CHAPTER 3

Scanning—Theory and Practice

3.1 Overview

The primary function of a scanner is to read in characters from a source file and group them into tokens. A scanner is sometimes called a *lexical analyzer* or *lexer.* The names scanner, lexical analyzer and lexer are used interchangeably. The `ac` scanner we saw in Chapter 2 was quite simple and could easily be coded by any competent programmer. We will now develop a thorough and systematic approach to scanning that will allow us to create scanners for complete programming languages.

We will introduce formal notations for specifying the precise structure of tokens. At first glance this may seem an unnecessary complication, given the simple token structure found in most programming languages. However token structure can be more detailed and subtle than one might expect. For example, we are all familiar with simple quoted strings in C, C++ or 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, not unless it is escaped with a backslash. This is to avoid a "runaway string" which, lacking a closing quote, matches characters intended to be part of other tokens. While C, C++ and Java allow escaped newlines in strings, Pascal is even stricter and forbids them entirely. Ada goes further still and forbids all unprintable characters (precisely because they are normally unreadable). Similarly, are null strings (of length zero) allowed? In C, C++, Java and Ada they are, 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 obviously 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 allowed, 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, we wish this to be interpreted as a range specifier (`1` to `10`). 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 reals *cannot* be adjacent is reflected in the context-free grammar, which is enforced by the parser, not the scanner.)

Given a formal specification of token and program structure, it is possible to examine a language for design flaws. For example, we could analyze all pairs of tokens that can be adjacent and determine whether the catenation of the two might be incorrectly scanned. If so, a separator is required (as is the case for adjacent identifiers and reserved words), or 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 common to all scanners, a significant duplication of effort. The goal of a *scanner generator* is to limit the effort in building a scanner to specifying which tokens the scanner is to recognize. Using a formal notation, we tell the scanner generator what tokens we want recognized; it 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. The 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 ordinary programming, which we call *procedural*, we do not tell a scanner generator *how* to scan but simply *what* we want scanned. 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. (Database query languages and Prolog, a "logic" programming language, are declarative.) Declarative programming is most successful in limited domains, such as scanning, where the range of implementation decisions that must be *automatically* made 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.

Though our primary focus in this text 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, if poorly implemented, they can be significant performance bottlenecks. The reason is that scanners must wade through the text of a program character by character.

Assume we wish to implement a very fast compiler that can compile a program in a few seconds. Let's use 30,000 lines a minute (500 lines a second) as our goal. (Compiler such as "Turbo C++" achieve such speeds.) If an average line contains 20 characters, we 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 1000 instructions per input character to spend. Since scanning *isn't* the only thing a compiler does, 250 instructions per character is more realistic. This is a rather tight budget given that even a simple assignment takes several instructions on a typical processor. Though multi-MIPS processors are common these days and 30,000 lines per minute is an ambitious speed, it's clear that a poorly coded scanner can dramatically impact the performance of a compiler.

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* will be 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. A well-known scanner generator, *Lex*, will be considered in some detail in Section 3.4. A few alternatives will also be considered. Lex takes token definitions (in a declarative form—regular expressions) and produces a complete scanner subprogram, ready to be compiled and executed.

Our next topic of discussion, in Section 3.5, is 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 3.6 that explains how scanner generators, like Lex, translate regular expressions into finite automata and how finite automata may be converted to equivalent regular expressions. Readers who wish 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 earlier. The section also illustrates how finite automata can be built, merged, simplified, and even optimized.

3.2 Regular Expressions

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

Regular expressions are widely used in computer applications other than compilers. The Unix utility *grep* uses regular expressions 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 specifies desired matches using regular expressions.

The sets of strings defined by *regular expressions* are termed *regular sets.* For purposes of scanning, a token class will be a regular set, whose structure is defined by a regular expression. Particular instances of a token class are sometimes called *lexemes,* though we will simply call a string in a token class an instance of that token. For example, we will 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 V). This vocabulary is normally the character set used by a computer. Today, the *ASCII* character set, which contains a total of 128 characters, is very widely used.

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

Strings are built from characters in the character set V via *catenation*. As characters are catenated to a string, it grows in length. 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 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9). (In this text we will often use abbreviations like (0 | … | 9) rather than enumerate a complete list of alternatives. The “…” symbol *is not* part of our regular expression notation.)

The characters (,), ' , ∗, +, and | are *meta-characters* (punctuation and regular expression operators). Meta-characters must be quoted when used as ordinary characters to avoid ambiguity. (Any character or string may be quoted, but unnecessary quotation is avoided to enhance readability.) 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 lower-case letters and UC is the set of upper-case letters, then (LC | 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,* is also allowed. The operator $*$ will represent the postfix *Kleene closure* operator.

Let P be a set of strings. Then P^* 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^* is the set of all words composed only of lower-case letters, of any length (including the zero length word, λ).

Precisely stated, a string $s \in P^*$ if and only if s can be broken into zero or more pieces: $s = s_1\, s_2 \ldots s_n$ 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^*.

Now that we are familiar with 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 contains 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 | B, A B, and A^* are also regular expressions, denoting the alternation, catenation, and Kleene closure of the corresponding regular sets.

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

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

- P^+ denotes all strings consisting of *one* or more strings in P catenated together: $P^* = (P^+ \mid \lambda)$ and $P^+ = P\,P^*$. 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 (V – A); that is, all *characters* in V not included in A. Since Not(A) contains characters rather than strings, it must be finite, and is automatically regular. Not(A) does not contain λ since 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, '\n' in Java or C).

  It is possible to extend Not to strings, rather than just V. That is, if S is a set of strings, we can define $\overline{S}$ to be $(V^* - S)$; that is the set of all strings except those in S. Though $\overline{S}$ is usually infinite, it is also 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, $A^k = (A\,A\,A \ldots)$ (k copies). Thus $(0 \mid 1)^{32}$ is the set of all bit strings exactly 32 bits long.

Examples

We will now explore how regular expressions can be used to specify tokens. Let D be the set of the ten single digits and let L be the set of all letters (52 in all). Then

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

$$\text{Comment} = //\ \text{Not(Eol)}^*\ \text{Eol}$$

This regular expression says a comment begins with two slashes and ends at the *first* end of line. Within the comment we allow any sequence of characters not containing an end of line (this guarantees the first end of line we see ends the comment).

- A fixed decimal literal (e.g., `12.345`) can be defined as:

  $$\text{Lit} = D^{+}\textbf{.}\, D^{+}$$

  We require one or more digits on both sides of the decimal point, so this definition excludes `.12` and `35.`
- An optionally signed integer literal can be defined as:

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

  An integer literal is one or more digits preceded by a plus or minus or no sign at all (λ). So that the plus 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:

  $$\text{Comment2} = \#\#\ ((\# \mid \lambda)\ \ \text{Not}(\#)\ )^{*}\ \#\#$$

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

All finite sets and many infinite sets are regular. But not all infinite sets are regular. For example, consider the set of balanced brackets of the form [[[...]]]. This set is defined formally as $\{\ [^{m}\ ]^{m} \mid m \ge 1\ \}$. 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 it includes extra, unwanted strings. (Exercise 14 proves this.)

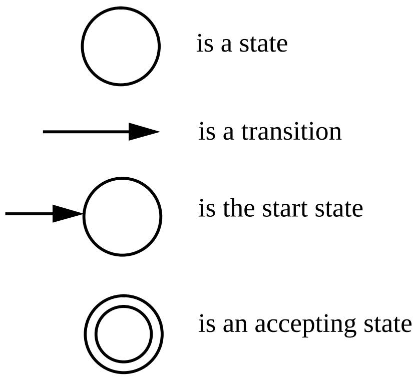
It is easy to write a context-free grammar (CFG) that defines balanced brackets precisely. In fact, all regular sets can be defined by CFGs. Thus, our bracket example shows that CFGs are a more powerful descriptive mechanism than regular expressions. Regular expressions are, however, quite adequate for 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

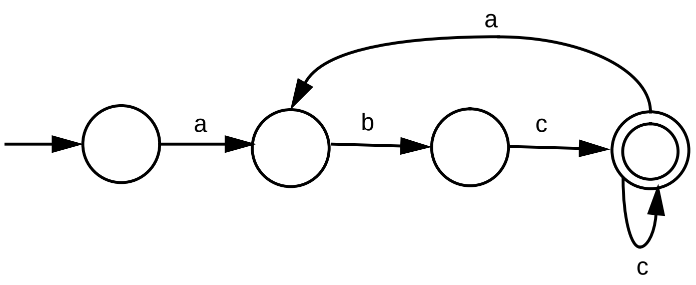
A *finite automaton* (FA) can be used to recognize the tokens specified by a regular expression. An FA is a simple, idealized computer that recognizes strings belonging to regular sets. It consists of:

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

These four components of a finite automaton are often represented graphically:



Finite automata (the plural of automaton is automata) can be represented graphically using *transition diagrams.* Using these diagrams, we start at the start state. If the next input character matches the label on a transition from the current state, we go to the state it points to. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a *valid* token; otherwise, we have not seen a valid token. In the diagram shown below, the valid tokens are the strings described by the regular expression $(a\ b\ (c)^+)^+$.

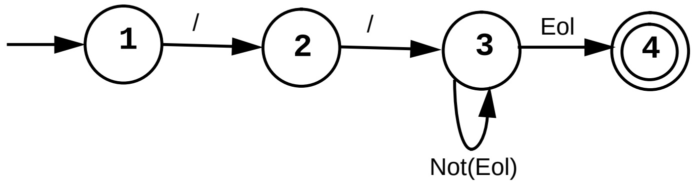


As an abbreviation, a transition may be labeled with more than one character (for example, Not(c)). The transition may be taken if the current input character matches any of the characters labeling the transition.

If an FA always has a *unique* transition (for a given state and character), the FA is *deterministic* (that is, a deterministic FA, or DFA). Deterministic finite automata 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 be part of the current token. For example, the regular expression

// Not(Eol)^* Eol

which defines a Java or C++ single-line comment, might be translated into



The corresponding transition table is

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

A full transition table will contain one column for each character. To save space, *table compression* is sometimes utilized. That is, only non-error entries are explicitly represented in the table, 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 using a scanner generator.

A DFA can be coded in a

- Table-driven form
- Explicit control form

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 direct 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. Using the DFA for the Java comments illustrated above, the two approaches would produce the programs illustrated in Figures 1 and 2.

The first form is commonly produced by a scanner generator; it is token-independent. It uses a simple driver that can scan *any* token if the transition table is properly stored in T. The latter 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 is specific to a single token definition.

```
     ● Asume CurrentChar contains the first character to be scanned
 1.  State ← StartState
 2.  while true
 3.          do  if CurrentChar = eof
 4.                  then break
 5.              NextState ← T[State, CurrentChar]
 6.              if NextState = error
 7.                  then break
 8.              State ← NextState
 9.              READ(CurrentChar)
10.  if State ∈ AcceptingStates
11.      then ● Return or process valid token
12.      else ● Signal a lexical error
```

FIGURE 1 Scanner Driver Interpreting a Transition Table

```
     ● Asume CurrentChar contains the first character to be scanned
1.   if CurrentChar = '/'
2.      then READ(CurrentChar)
3.           if CurrentChar = '/'
4.              then repeat
5.                        READ(CurrentChar)
6.                   until CurrentChar $\in$ { eol, eof }
7.              else ● Signal a lexical error
8.      else ● Signal a lexical error
9.   if CurrentChar = eol
10.     then ● Return or process valid token
11.     else ● Signal a lexical error
```

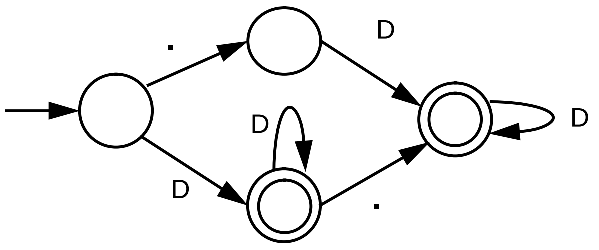
FIGURE 2 Explicit Control Scanner

The following are two more examples of regular expressions and their corresponding DFAs:

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

  RealLit = $(D^+ (\lambda \mid \mathbf{.})) \mid (D^* \mathbf{.} D^+)$

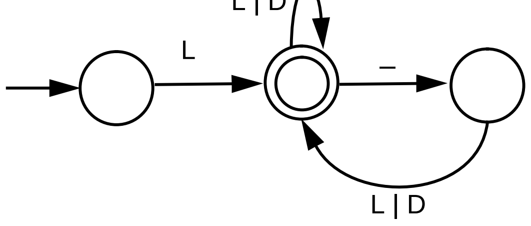
  which corresponds to the DFA



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

  ID = $L (L \mid D)^* (\_ (L \mid D)^+)^*$

  This definition includes identifiers like `sum` or `unit_cost`, but excludes `_one` and `two_` and `grand___total`. The corresponding DFA is

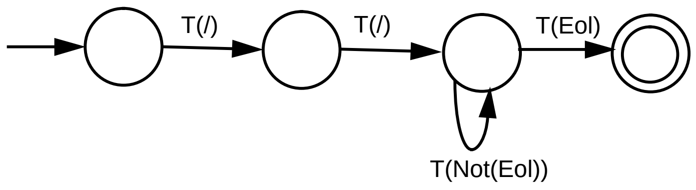


So far we haven't saved or processed the characters we've scanned—they are matched and then thrown away. It is useful to add an output facility to an FA; this makes the FA a *transducer.* As characters are read, they can be transformed and catenated to an output string. For our purposes, we shall limit 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 this notation:

$\xrightarrow{\text{a}}$ means save `a` in a token buffer

$\xrightarrow{\text{T(a)}}$ means don't save `a` (Toss it away)

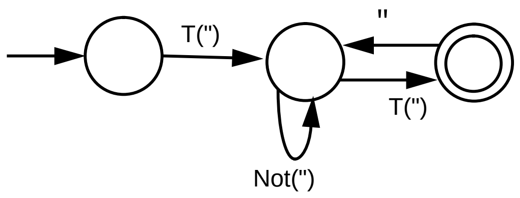
For example, for Java and C++ comments, we might write



A more interesting example is given by Pascal-style quoted strings, according to the regular expression

(" (Not(") | " ")* ")

A corresponding transducer might be



The input `"""Hi"""` would produce output `"Hi"`.

3.4 The Lex Scanner Generator

We now discuss a very popular scanner generator, Lex. We will later briefly discuss a number of 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 [Lesk and Schmidt 1975] and [Levine, Mason and Brown 1992]. Flex (see [Paxson 1988]) is a widely used 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. A scanner specification, defining the tokens to be scanned and how they are to be processed, is presented to Lex. Lex then generates a complete scanner, coded in C. This scanner is then compiled and linked with other compiler components to create a complete compiler.

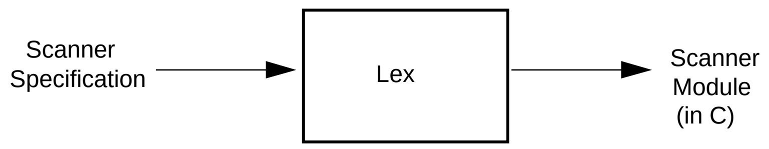


FIGURE 3 The Operation of the Lex Scanner Generator

Using Lex saves us a great deal of effort in programming a scanner. We are relieved of the necessity of explicitly programming many low level details of the scanner (reading character efficiently, buffering them, matching characters against token definitions, and so on). Rather, we can focus on the character structure of tokens, and how they are to be processed.

Our primary purpose in this section is to show how regular expressions and related information are presented to scanner generators. A good way to learn 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 say *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 returned by `yylex`() is the token code of the token scanned by Lex. Tokens like white space are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed. Figure 4 illustrates a simple Lex definition for the three reserved words of the `ac` language (which was introduced in Chapter 2). When a string matching `f` or `i` or `p` is found, the appropriate token code is returned. It is vital that the token codes 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.

It is standard for the scanner and parser to share the definition of token codes to guarantee consistent values are seen by both. The file `y.tab.h`, produced by the Yacc parser generator (see Chapter 6) is often used to define shared token codes.

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

FIGURE 4 A Lex Definition for ac's Reserved Words

The pair %% delimits sections of a Lex token specification. Three sections exist; the general form of a Lex specification is

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

In our simple example, we've used only the second section in which regular expressions and corresponding C code are specified. The regular expressions illustrated in Figure 4 are simple—single character strings that match only themselves. The code executed returns a constant value representing the appropriate ac token.

If we wished, we could have quoted the strings representing the reserved words ("f" or "i" or "p"), but since these strings contain no delimiters or operators, quoting it is unnecessary. If you want to quote such strings to avoid any chance of misinterpretation, that's fine with Lex.

Our specification so far is quite incomplete. None of the other tokens in ac are handled, particularly identifiers and numbers. To handle these tokens correctly, we'll introduce a useful concept—*character classes*.

Characters often naturally fall into classes, with all characters in a class 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 digit characters can be used.

Character classes are delimited by [and]; individual characters are catenated without any quotation or separators. However \, ^,] and -, because of their special meaning in character classes, must be escaped. Thus [xyz] represents the class that can match a single x, y, or z. The expression [\])] represents the class 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 -; [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, upper- and lower-case. \ is the escape character, 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 octal 10.

The ^ symbol complements a character class (it is Lex's representation of the Not operation). [^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 use of \0 in character classes; it can be confused with the null character's special use as end of string terminator in C.) Table 1 illustrates a variety of character classes and the character sets they define.

TABLE 1. Lex Character Class Definitions

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 one five character *string*: `[abc]`

With character classes we can easily define ac identifiers, as shown in Figure 5. The character class includes the range of characters, a to e, then g and h, then the range j to o, followed by the range q to z. We are able to concisely represent the 23 characters that may form an ac identifier without having to enumerate them all.

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

FIGURE 5 A Lex Definition for ac's Identifiers

Tokens are defined using regular expressions. Lex provides the standard regular expression operators, as well as some additions. Catenation is specified by the juxtaposition of two expressions; no explicit operator is used. Thus `[ab][cd]` will match any of `ad`, `ac`, `bc`, and `bd`. When outside of character class brackets, individual letters and numbers match themselves; 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 if we wish to match the reserved word `while` allowing any mixture of upper- 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). `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 can now define all of ac's tokens using Lex's regular expression facilities. This is shown in Figure 6.

```
%%
(" ")+                         { /* 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 6 A Lex Definition for ac's Tokens

Recall that a Lex specification of a scanner consists of three sections. The first section, which we've 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, given the definition

```
Letter [a–zA–Z]
```

the expression `{Letter}` expands to `[a-zA-Z]`. Symbolic definitions can often make Lex specifications easier to read, as illustrated in Figure 7.

```
%%
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 7 An Alternative Lex Definition for ac's Tokens

In the first section we 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 and variable, procedure and type declarations that are needed to allow the commands of section two to be compiled. For example, we might use

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

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

As we have seen, Lex's second section defines a table of regular expressions and corresponding commands in C. The first blank or tab not escaped or part of a quoted string or character class is taken as the end of the regular expression, so avoid embedded blanks 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 [Lesk and Schmidt 1975] or [Levine, Mason and Brown 1992]). 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 contents 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 contents of `yytext` (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. First, the longest possible match is performed. Lex automatically buffers characters while deciding how many characters can be matched. Second, 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. Their definitions are therefore 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 like `.*` which will consume all characters up to the next newline.

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. It may therefore be necessary 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 definition 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 rerunning Lex. (Some implementations of Lex generate scanners rather slowly.)

In Lex, end of file is not handled by regular expressions. A predefined `EOF` 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 `EOF` 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 input and output. They are

`input()`	read a single character, 0 on end of file.
`output(c)`	write a single character to the output.
`unput(c)`	put a single character back into the input to be re-read.

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 one if there is no more input. Otherwise, it returns zero and arranges for `input`() to provide more characters.

The compiler writer may supply the `input`(), `output`(), `unput`(), and `yywrap`() functions (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 one, 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 like `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. That is, `r/s` tells Lex to match regular expression `r` but only if regular expression s immediately follows it. `s` is *right context*; 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 Table 2 . Note that a symbol sometimes has one meaning in a regular expression and an entirely different meaning in a character class (i.e., within a pair of brackets). If you find Lex behaving unexpectedly, it’s a good idea to check this table to be sure of how the operators and symbols you’ve used behave. Ordinary letters and digits, and symbols not mentioned (like `@`) represent themselves. If you’re not sure if a character is special or not, 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 is learning Lex’s notation and rules. Once you’ve done this, Lex will relieve you of the many of chores of writing a scanner (reading characters, buffering them, deciding which token pattern matches, etc.). 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. Lex does require that code segments be written in C, and hence is not language-independent.

TABLE 2. Meaning of Operators and Special Symbols in Lex

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
`(`	Matches with) to group sub-expressions.	Represents itself.
`)`	Matches with (to group sub-expressions.	Represents itself.
`[`	Begins a character class.	Represents itself.
`]`	Represents itself.	Ends a character class.
`{`	Matches with } to signal macro-expansion.	Represents itself.

TABLE 2. Meaning of Operators and Special Symbols in Lex

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
}	Matches with { to signal macro-expansion.	Represents itself.
"	Matches with " to delimit strings (only \ is special within 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 zero matches).	Represents itself.
/	Context sensitive matching operator.	Represents itself.
^	Matches only at beginning of a line.	Complements remaining characters in the class.
$	Matches only at end of a line.	Represents itself.
-	Represents itself.	Range of characters operator.

3.4.2 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, and produces scanners too slow to be used in production compilers. [Jacobsen 1987] has shown that Lex can be improved so that it is always faster than a hand-written scanner. As noted earlier, Flex (Fast Lex) is a freely distributabled Lex clone. It produces scanners that are considerably faster than the ones produced by Lex. Flex also provides options that allow tuning of the scanner size versus its speed, as well as some features that Lex does not have (such as support for eight-bit characters). If Flex is available on your system you should use it instead of Lex.

Lex has also been implemented in languages other than C. JLex [Berk 1997] is a Lex-like scanner generator written in Java that generates Java scanner classes. It is of particular interest to individuals writing compilers in Java. Alex, [Self 1990], is an Ada version of Lex. Lexgen, [Appel 1989], is an ML version of Lex.

An interesting alternative to Lex is GLA (Generator for Lexical Analyzers), [Gray 1988]. 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 hand-coded scanners. Another tool that produces directly executable scanners is RE2C, [Bumbulis 1993]. 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 DOS and Macintosh systems as well as Unix systems. Among the most widely-used and highly-recommended of these are DLG (part of the PCCTS tools suite, [Parr 1991]), CoCo/R, [Moessenboeck 1991], an integrated scanner/parser generator, and Rex, [Grosch 1989], part of the Karlsruhe Cocktail tools suite.

3.5 Practical Considerations

In this section we discuss the practical considerations necessary to build real scanners for real programming languages. As one might expect, the finite automaton model we have developed sometimes falls short and must be supplemented. Efficiency concerns must be addressed. Further, some provision for error handling must be incorporated into any practical scanner.

We shall discuss a number of potential problem areas. In each case, solutions will be weighed, particularly in conjunction with the Lex scanner generator we have studied.

3.5.1 Processing Identifiers and Literals

In simple languages with only global variables and declarations, it is common to have the scanner 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, we usually don't ask the scanner to enter or look up identifiers in the symbol table because an identifier can be used in many contexts (as a variable, in a declaration, as a member of a class, a label, and more). It is not possible, in general, for the scanner to 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, will resolve the identifier's intended usage.

Sometimes a *string space* is used to store identifiers (see Chapter 8). This avoids frequent calls to memory allocators like `new` or `malloc` to allocate private space for a string and avoids the space overhead of storing multiple copies of the same string. If a string space is used, 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.* All identifiers that have the same text get the same serial number; identifiers with different texts always get different serial numbers. A serial number is a small integer that can be used instead of a string space pointer. 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, but in others, such as Ada and Pascal, case is insignificant. If 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, if case is insignificant, we need to guarantee that case differences in the spelling of an identifier or reserved word do 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 like `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, the above character sequence can also 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 separating 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 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 `id` token). This allows the parser to easily distinguish the two constructs—they now begin with different tokens.

Why does this complication exist in C? `typedef` statements were 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 above had to be devised to resolve the correct usage.

3.5.2 Reserved Words

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

Most programming languages choose to make key words 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 are required). Now assume that `begin` and `end` are not reserved and some devious programmer has declared procedures named `begin` and `end`. The following program can be parsed in many ways; its meaning is not well defined:

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

With careful design, outright ambiguities can be avoided. For example, in PL/I key words are not reserved, but procedures are called using an explicit `call` key word. Nonetheless, opportunities for convoluted usage abound because key words may be used as variable names:

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

The problem with reserved words is that if they are too numerous, they may confuse inexperienced programmers who 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 were the symbol in question not reserved. COBOL is infamous for this problem, having 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 were able to recognize reserved words by creating distinct regular expressions for each reserved word. This approach was feasible because Lex (and Flex) allow 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 will increase the number of states in the transition table a scanner generator creates. [Gray 1988] reports that in as simple a language as Pascal (which has only 35 reserved words), the number of states increases from 37 to 165 when each reserved word is defined by its own regular expression. In uncompressed form with 127 columns for ASCII characters (excluding null), the number of transition table entries would increase from 4699 to 20,955. This may not be a problem with modern multi-megabyte memories. Nonetheless, some scanner generators, like Flex, allow you to choose to optimize scanner size or scanner speed.

In Exercise 18 it is established 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 we can write a regular expression for nonreserved identifiers: $\overline{(\overline{\mathtt{ident}}|\mathtt{if}|\mathtt{while}|\ldots)}$

That is, if we take the complement of the set containing reserved words and all non-identifier strings, we get all strings that *are* identifiers *excluding* the reserved words. Unfortunately, neither Lex nor Flex provide 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 seriously consider. Suppose END is the only reserved word, and identifiers contain only letters. Then

$$L \mid (L\ L) \mid ((L\ L\ L)\ L^{+}) \mid ((L - \text{'E'})\ L^{*}) \mid (L\ (L - \text{'N'})\ L^{*}) \mid (L\ L\ (L - \text{'D'})\ L^{*})$$

defines identifiers shorter or longer than three letters, or not starting with E or without N in position two, and so forth.

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 what looks like an identifier is scanned, a table of exceptions is consulted to see if a reserved word has been recognized. If case is significant in reserved words, the exception lookup will require an exact match; otherwise, the token should be translated to a standard form (all upper- or lowercase) before the lookup.

An exception table may be organized in a variety of ways. An obvious organization is a sorted list of exceptions suitable for a binary search. A hash table may also 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. It has been shown by [Cichelli 1980] 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 it is 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. If what looks like an identifier is found to have a serial number or string space position *smaller* than the initial position assigned to identifiers, then we immediately know that a reserved word rather than an identifier has been scanned. In fact with a little care it is possible to 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.

3.5.3 Compiler Directives and Listing Source Lines

Compiler directives and pragmas are used to control compiler options (listings, source file inclusion, conditional compilation, optimizations, profiling, and so on). They may be processed either by the scanner or later 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, like 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, which cause it to suspend reading the current file and to begin reading and scanning 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, it is 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, etc.

Some languages (like 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, and hence a conditional expression will be evaluated. Characters following the expression will then be scanned and passed to the parser or 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. This is straightforward, but a bit of care is required. The most obvious way to produce a source listing is to echo characters as they are read, using end of line conditions to terminate a line, increment line counters, and so on. This approach has a number of short-comings, however:

- Error messages may need to be printed, and 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).
- 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 a priori 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, line numbers ought to correspond to source lines.) 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. The message can specify where in the source file the error occurred. It is straightforward to open the source file and 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.

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.

3.5.4 Scanner Termination

A scanner is designed to read input characters and partition them into tokens. What happens when the end of the input file is reached? It is convenient to create an `Eof` pseudocharacter when this occurs. 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 `Eof` *token* that can be passed back to the parser. An `Eof` 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 (Chapter 5) and LALR(1) parsers (Chapter 6) require an end of file token.

What should happen if a scanner is called after the 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 `Eof` token to the parser. This allows the parser to handle termination cleanly, especially since the `Eof` token is normally syntactically valid only after a complete program is parsed. If the `Eof` token appears too soon or too late, the parser can perform error repair or issue a suitable error message.

3.5.5 Multicharacter Lookahead

We can generalize finite automata 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! 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.)

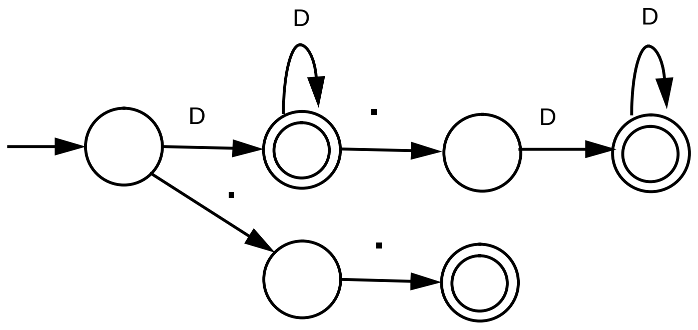


FIGURE 8 An FA That Scans Integer and Real Literals and the Subrange Operator

We've already seen a milder form of the extended lookahead problem that occurs in Pascal and Ada. To scan `10..100` we need two-character lookahead after the `10`. Suppose we use the FA of Figure 8. Given `10..100` we would scan three characters and stop in a non-accepting state. Whenever we stop reading in a non-accepting state, we can back up along 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 we know we never have more than two-character lookahead, which simplifies buffering characters to be rescanned. Alternatively, we can add a new accepting state to the above automaton, corresponding 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 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 (`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, but 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 reach a situation in which 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 3.5.7.)

Buffering and backup are essential in general purpose scanners like 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 deterministic finite automaton) 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 backup 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 our earlier example of `12.3e+q`. The following table illustrates how the buffer is built and flags are set:

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)

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.

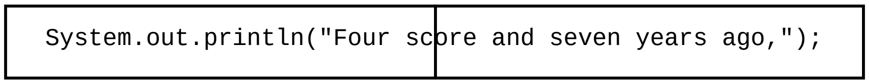
3.5.6 Performance Considerations

Our main concern in this chapter is learning 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 briefly what to do to increase scanning speed. One approach is to use a scanner generator like Flex or GLA that is designed to generate fast scanners. These generators will incorporate many “tricks” that increase speed in non-obvious ways.

If you code a scanner by hand, a few general principles, if followed, can increase scanner performance dramatically. First, try to block character-level operations whenever possible. That is, 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 our examples we’ve read input one character as a time, perhaps using Java’s `InputStream.read` (or a C or C++ equivalent). Single character reads make our discussion simpler, but they are quite inefficient. A subroutine call can cost hundreds or thousands of instructions to execute—far too much 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 disc block (512 or perhaps 1024 bytes) so that an entire disc 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, and an end of file 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 we might see the beginning of a quoted string, but not its end. If we do another read operation to get the rest of the string, the first part may be overwritten.

To avoid this problem, we may do *double-buffering*, as shown in Figure 9. Input is first read into the left buffer, then 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 we make the buffer size large enough (say 512 or 1024 characters), the chance of losing part of a token is very low. If we find that the length of a token is near the buffer length, we can extend the buffer size, perhaps by using Java-style `Vector` objects rather than arrays to implement buffers.



```
System.out.println("Four sc|ore and seven years ago,");
```

FIGURE 9 An Example of Double Buffering

Whenever we fetch a character from one of the buffers, we must ask if we are at the end of the left buffer or at the end of the right buffer. We can speed this “end of buffer” check by using a *sentinel* character. This is a character that can’t appear in the input. We place the sentinel character just beyond the end of a buffer. When we fetch a character for scanning, we check to see if it is a sentinel. If it is, we’re at the end of a buffer and it is time to read more input into the other buffer.

Besides doing block reads, we can speed a scanner by avoiding unnecessary copying of characters. Because we see so many characters during scanning, moving them from one place to another can be costly. With block reads we directly read into our scanning buffer rather than into an intermediate input buffer. As we scan characters, 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, like `qpt`, `prof`, `gprof`, or `pixie` may allow you to find unexpected performance bottlenecks in your scanner.

3.5.7 Lexical Error Recovery

A character sequence that can't be scanned into any valid token is a *lexical error.* Though such errors are uncommon, they 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:

- Delete the characters read so far and restart scanning at the next unread character.
- Delete the first character read by the scanner and resume scanning at the character following it.

Both of these approaches are reasonable. The former is easy to do. We just reset the scanner and begin scanning anew. The latter is a bit harder but also is a bit safer (in that less is immediately deleted). It 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 will usually appear as the beginning of a token. In such a 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. In effect we'd get `...for tnight...`, which will cause a syntax error. Such occurrences are unavoidable, though a good syntactic error-repair algorithm will often make some reasonable repair.

If the parser has a syntactic error-repair mechanism, it can be useful to return a special warning token when a lexical error occurs. The semantic value of the warning token is the character string deleted to restart scanning. When the parser sees the warning token, it is warned that the next token is unreliable and that error repair may be required. Furthermore, 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 ought to receive special error messages. First consider runaway strings. In Java strings are not allowed to cross line boundaries, so a runaway string is detected when an end of a line 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 as 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'll use an error token to represent a string terminated by an end of line rather than a double quote character. For a valid string, in which internal double quotes and back slashes are escaped (and no other escaped characters are allowed), we can use

$$" \; (\; \mathrm{Not}(\; " \mid \mathrm{Eol} \mid \backslash \;) \mid \backslash" \mid \backslash\backslash \;)^{*} \; "$$

For a runaway string we can use

$$" \; (\; \mathrm{Not}(\; " \mid \mathrm{Eol} \mid \backslash \;) \mid \backslash" \mid \backslash\backslash \;)^{*} \; \mathrm{Eol}$$

(Eol is the end of line character.) When a runaway string token is recognized, a special error message should be issued. Further, the string may be repaired 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, 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. Though legal, such strings almost always result from errors that cause a string to extend father 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 like C, C++, Java and Pascal, which allow multiline comments, improperly terminated (*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 the end of file is reached. Clearly a special error message is required.

Let's look at Pascal-style comments that begin with a `{` and end with a `}`. Comments that begin and end with a pair of characters, like `/*` and `*/` in Java, C and C++, are a bit trickier to get right (see Exercise 6). Correct Pascal comments are defined quite simply:

$$\{\ \mathrm{Not}(\,\}\,)^{*}\ \}$$

To handle comments terminated by Eof, our error token approach can be used:

$$\{\ \mathrm{Not}(\,\}\,)^{*}\ \mathrm{Eof}$$

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 above. We therefore split our legal comment definition into two tokens. The one that accepts an open comment in its body causes a warning message (`"Possible unclosed comment"`) to be printed. We now have three token definitions:

$$\{\ \mathrm{Not}(\,\}\,)^{*}\ \}\quad \text{and}\quad \{\ (\mathrm{Not}(\,\{\,|\,\}\,)^{*}\ \{\ \mathrm{Not}(\,\{\,|\,\}\,)^{*}\ )^{+}\ \}\quad \text{and}\quad \{\ \mathrm{Not}(\,\}\,)^{*}\ \mathrm{Eof}$$

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

Of course, single line comments, found in Java and C++, are always terminated by Eol, and 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, as we saw above, nested comments normally fail because FAs and regular expressions cannot recognize properly balanced open comment/close comment sequences. This failure to recognize such sequences 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, like `#if` and `#endif` used in C and C++ are designed to safely disable compilation of selected parts of a program.

3.6 Translating Regular Expressions and Finite Automata

Regular expressions are equivalent to FAs. In fact, the main job of a scanner generator program like 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). Upon reading a particular input, an NFA is not required to make a unique (deterministic) choice of which state to visit. For example, as shown in Figure 10, an NFA is allowed to have a state that has two transitions (arrows) coming out of it, labeled by the same symbol. As shown in Figure 11, an NFA may also have transitions labeled with λ.

Transitions are normally labeled with individual characters in V, 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 automaton is in the state at the left and the next input character is a, it may choose to use the transition labeled a or 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.*

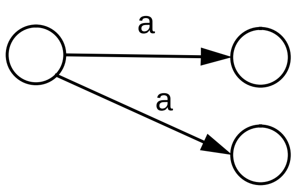


FIGURE 10 An NFA with Two a Transitions

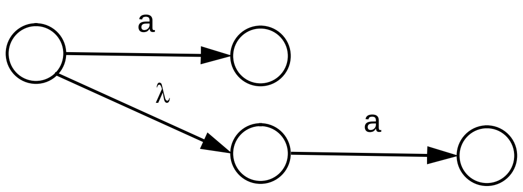


FIGURE 11 An NFA with a λ Transition

The algorithm to make an FA from a regular expression proceeds in two steps: First, it transforms the regular expression into an NFA, and then it transforms the NFA into a deterministic one. This first step is easy.

Regular expressions are all built out of the *atomic* regular expressions a (where a is a character in V) and λ by using the three operations A B and A | B and A^*. Other operations (like A^+) are just abbreviations for combinations of these. As shown in Figure 12, NFAs for a and λ are trivial.

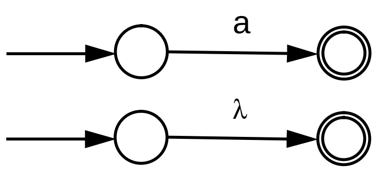


FIGURE 12 NFAs for a and λ

Now suppose we have NFAs for A and B and want one for A | B. We construct the NFA shown in Figure 13. 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 automaton.

As shown in Figure 14, the construction for A B is even easier. The accepting state of the combined automaton is the same state that was the accepting state of B. We could also just merge the accepting state of A with the initial state of B. We chose not to only because the picture would be more difficult to draw.

Finally, the NFA for A^* is shown in Figure 15. 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 zero or more strings that belong to A are matched.

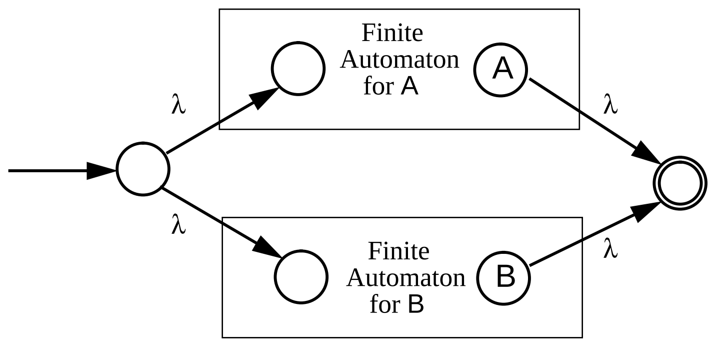


FIGURE 13 An NFA for A | B

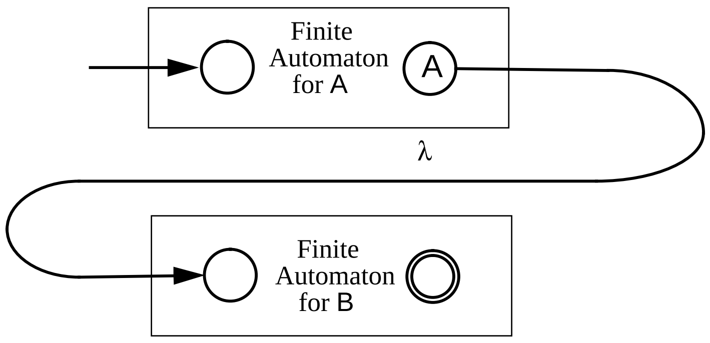


FIGURE 14 An NFA for A B

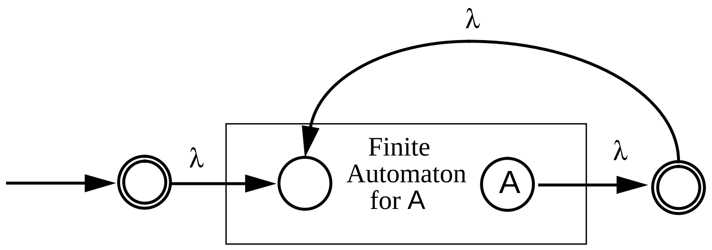


FIGURE 15 An NFA for A^*

3.6.1 Creating Deterministic Automata

The transformation from an NFA *N* to an equivalent DFA *D* works by what is sometimes called the *subset construction.* Each state of *D* corresponds to 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 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, which makes tracking various sets of states feasible.

An accepting state of D will be any set containing an accepting state of N, reflecting the convention that N accepts if there is *any* way it could get to its accepting state by choosing the “right” transitions.

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 computes those states that can be reached following only λ transitions. Once the start state of D is built, we begin to create successor states. To do this, we take each state S of D, and each character c, and compute S’s successor under c. S is identified with some set of N’s states, $\{ n_1, n_2, \ldots \}$. We find all the possible successor states to $\{ n_1, n_2, \ldots \}$ under c, obtaining a set $\{ m_1, m_2, \ldots \}$. Finally, we compute T = CLOSE($\{ m_1, m_2, \ldots \}$). T is included as a state in D, and a transition from S to T 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 corresponds to a (finite) subset of N’s states, the process of adding new states to D must eventually terminate.

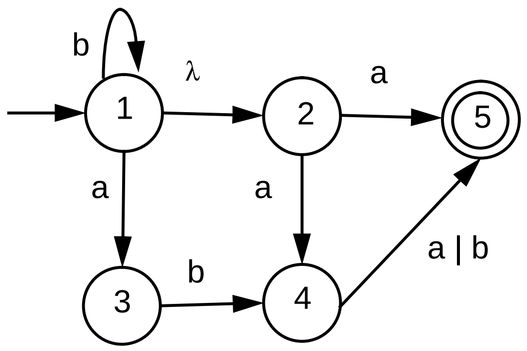
An algorithms for λ-closure follows. We utilize standard set operations and set notation.

```
● This algorithm adds to set S all states reachable from it using only λ transitions
CLOSE( SetofFaStates S )
 1.  while there exists a state $x \in S$ and there exists a state $y \notin S$ such that $x \xrightarrow{\lambda} y$
 2.      do  $S \leftarrow S \cup \{y\}$
```

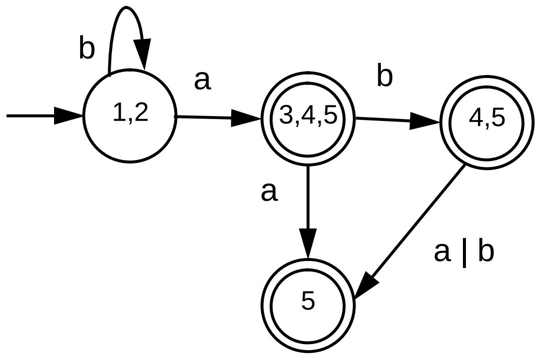
Using CLOSE, we can define the construction of a DFA, D, from an NFA, N:

```
DeterministicFa MAKEDETERMINISTIC( NonDeterministicFa N )
 1.  DeterministicFa D
 2.  SetofFaStates T
 3.  D.StartState ← { N.StartState }
 4.  CLOSE(D.StartState)
 5.  D. States ← { D.StartState }
 6.  while states or transitions can be added to D
 7.      do  choose any state $S \in D.$States and any character $c \in$ Alphabet
 8.          $T \leftarrow \{ y \in N.\text{States} \mid x \xrightarrow{c} y$ for some $x \in S\}$
 9.          CLOSE(T);
10.          if  $T \notin D.$States
11.              then  $D.\text{States} \leftarrow D.\text{States} \cup \{T\}$
12.                    $D.\text{Transitions} \leftarrow D.\text{Transitions} \cup \{\text{the transition } S \xrightarrow{c} T\}$
13.  $D.\text{AcceptingStates} \leftarrow \{ S \in D.\text{States} \mid \text{an accepting state of } N \in S\}$
```

To see how the subset construction operates, consider the following NFA:



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:



It is not too difficult to establish that the DFA constructed by MAKEDETERMINISTIC is equivalent to the original NFA (see Exercise 20). What is less 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.

In the case that creating a DFA is impractical (either because of size or speed-of-generation 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 construction of a DFA is not cost-effective.

3.6.2 Optimizing Finite Automata

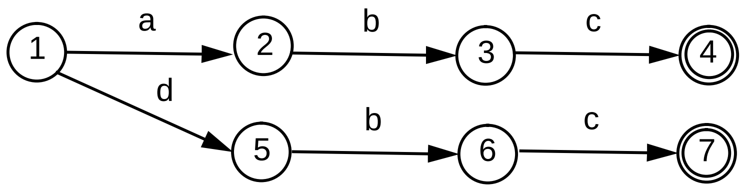
We do not have to stop with the DFA created by MAKEDETERMINISTIC. Sometimes this DFA will have more states than necessary. For every DFA there is a *unique* smallest (in terms of number of states) equivalent DFA. In other words, suppose a DFA (D) has 75 states and there is a DFA D′ with 50 states that 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 below, it is possible to optimize D by replacing it with D′.

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

We optimize the resulting DFA by merging together states we know to be equivalent. For example, two accepting states that have no transitions at all 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 together into one common state.

How do we decide what states to merge together? We take a *greedy* approach and try the most optimistic merger of states. By definition, accepting and non-accepting 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 non-accepting states. This merger into only two states is almost certainly too optimistic. In particular, all the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all the merged states must have no successor under c or they must all 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 following automaton:



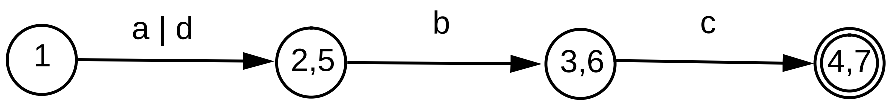
Initially we have a merged non-accepting state {1,2,3,5,6} and a merged accepting state {4,7}. A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur. We will 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 every state has a successor under every character. s_E is never merged with any real state.

Algorithm SPLIT, shown in Figure 16, splits merged states whose constituents do not agree on a common successor state for all characters. When SPLIT terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

Returning to our 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, have the error state s_E). This forces a split, yielding {1,2,5}, {3,6} and {4,7}. Now, for character b states 2 and 5 would go to the merged state {3,6}, but state 1 would not, so another split occurs. We now have: {1}, {2,5}, {3,6} and {4,7}. At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

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 state 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 containing the original start state; accepting states are those merged states containing accepting states (recall that accepting and non-accepting states are never merged).

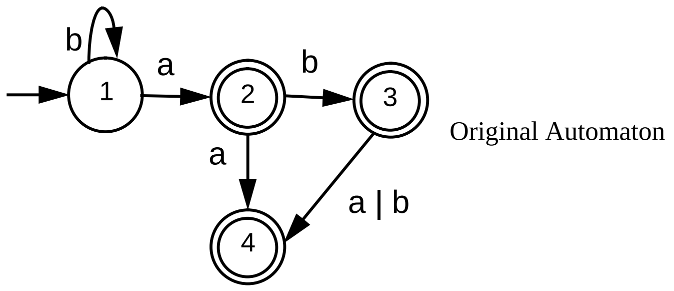
Returning to our example, the minimum state automaton we obtain is



A proof of the correctness and optimality of this minimization algorithm may be found in most texts on automata theory, such as [Hopcroft and Ullman 1979].

```
SPLIT( SetofFaStates  StateSet )
 1.  repeat
 2.          for each merged state $S \in StateSet$
 3.          do
 4.                  Assume $S$ corresponds to $\{s_1,..., s_n\}$
 5.                  for each character $c \in$ Alphabet
 6.                  do
 7.                          Let $t_1,..., t_n$ be the successor states to $s_1,..., s_n$ under $c$
 8.                          if  $t_1,..., t_n$ do not all belong to the same merged state
 9.                              then  Split $S$ into two or more new states such that
                                       $s_i$ and $s_j$ remain in the same merged state if
                                       and only if $t_i$ and $t_j$ are in the same merged state
10.  until no more splits are possible
```

FIGURE 16 An Algorithm to Split FA States



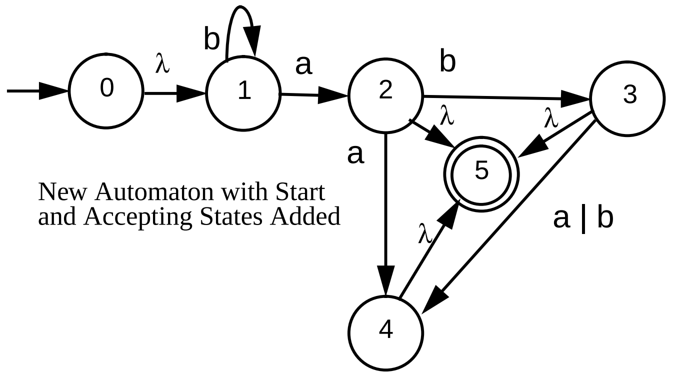


FIGURE 17 An FA with New Start and Accepting States Added

3.6.3 Translating Finite Automata to Regular Expressions

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

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

The algorithm we'll use is adapted from [Brozozowski and McCluskey 1963]. The idea is simple and elegant. We start with a finite automaton and simplify it by removing states, one by one. Simplified automata are equivalent to the original except for the fact that transitions are now labeled with regular expressions rather than individual characters. We continue removing states until we have an automaton with a single transition from the start state to a single accepting state. The regular expression labeling that single transition correctly describes the effect of the original automaton.

To start, we'll assume our finite automaton has a start state with no transitions into it and a single accepting state with no transitions out of it. If the automaton we start with 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 17 using the automaton we created with MAKEDETERMINISTIC in Section 3.6.1.We will define three simple transformations, T1, T2 and T3 that will allow us to progressively simplify finite automata. The first, illustrated in Figure 18(a), notes that if we have two different transitions between the same pair of states, with one transition labeled with R and the other labeled with S, then we can replace the two transitions with a new one, labeled with R | S. T1 simply reflects that we can choose to use the first transition *or* the second.

Transformation T2, illustrated in Figure 18(b) allows us to "by-pass" a state. That is, if state s has a transition to state r labeled with X and state r has a transition to state u labeled with Y, then we can go directly from state s to state u with a transition labeled with XY.

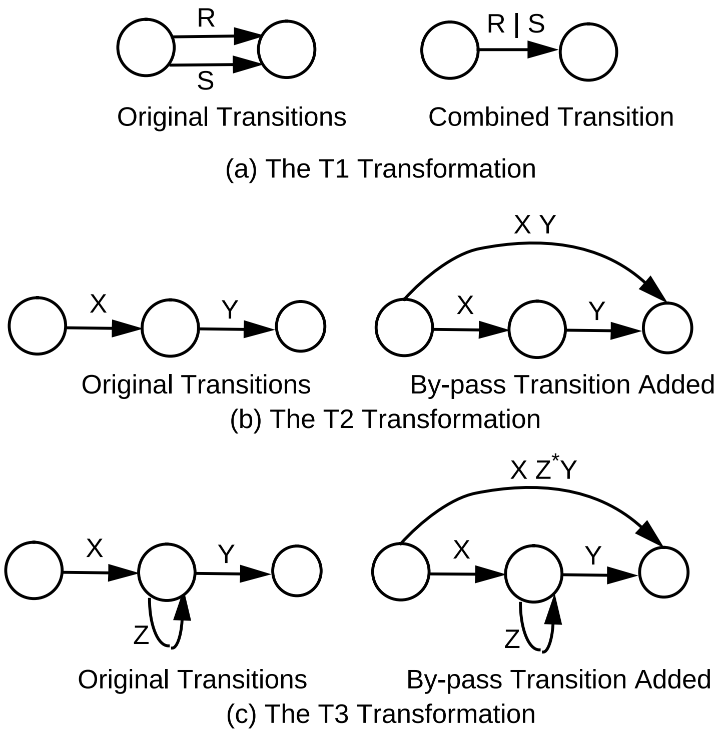


FIGURE 18 The T1, T2 and T3 Transformations

Transformation T3, illustrated in Figure 18(c) is similar to transformation T2. It again allows us to by-pass a state. If state s has a transition to state r labeled with X, and state r has a transition to itself labeled with Z, and state r also has a transition to state u labeled with Y, then we can go directly from state s to state u with a transition labeled with XZ^*Y. The Z^* term reflects that once we reach state r we can cycle back into r zero or more times before finally proceeding to u.

We will use transformations T2 and T3 as follows. If 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, then s is no longer needed—all paths through the finite automaton can by-pass it! Since s isn't needed, we will remove it. The finite automaton is now simpler because it has one fewer states. If we remove 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 automaton with only one transition, and the label on this transition will be the regular expression we want. FINDRE, shown in Figure 19, implements this algorithm.

```
RegularExpr FINDRE ( NonDeterministicFa Fa )
 1.  if Fa's start state has a transition into it
 2.      then Create a new start state; link it to the original start state with a λ-transition
 3.  if Fa has > 1 accepting state or Fa has an accepting state with out transitions
 4.      then Create a new and unique accepting state and link it to the original accepting states
              with λ-transitions
 5.  while Fa has > 1 transition
 6.       do while any pair of states have more than 1 transition between them
 7.               do Use a T1 transform to obtain a single transition
 8.          Let S be any state ≠ the start or accepting state
 9.          for each predecessor P of S where P ≠ S
10.               do for each successor U of S where U ≠ S
11.                       do if S has no transition to itself
12.                               then Create a transition from P to U using a
                                         T2 transformation
13.                               else Create a transition from P to U using a
                                         T3 transformation
14.          Remove S from Fa
15.  return the regular expression labeling the last remaining transition
```

FIGURE 19 An Algorithm to Generate a Regular Expression from a Finite Automaton

As an example, we will determine the regular expression corresponding to the FA we used in Section 3.6.1. The original automaton, with a new start state and accepting state added, is shown in Figure 20(a). State 1 has a single predecessor, state 0 and a single successor, 2. Using a T3 transformation, an arc directly from state 0 to state 2 is added, and state 1 is removed. This is shown in Figure 20(b). State 2 has a single predecessor, state 0 and three successors, 2, 4 and 5. Using three T2 transformations, arcs directly from state 0 to states 3, 4 and 5 are added. State 2 is removed. This is shown in Figure 20(c).

State 4 has two predecessors, states 0 and 3. It has one successor, state 5. Using two T2 transformations, arcs directly from states 0 and 3 to state 5 are added. State 4 is removed. This is shown in Figure 20(d). Two pairs of transitions are merged using T1 transformations, producing the finite automaton in Figure 20(e). Finally, state 3 is by-passed with a T2 transformation and a pair of transitions are merged with a T1 transformation, as shown in Figure 20(f). The regular expression we obtain is $b^*\,a\,b\,(a \mid b \mid \lambda) \mid b^*\,a\,a \mid b^*\,a$. By expanding the parenthesized subterm and then factoring a common term, we obtain $b^*\,a\,b\,a \mid b^*\,a\,b\,b \mid b^*\,a\,b \mid b^*\,a\,a \mid b^*\,a \equiv b^*\,a\,(ba \mid bb \mid b \mid a \mid \lambda)$.

Careful examination of the original automaton will verify that this expression correctly describes it.

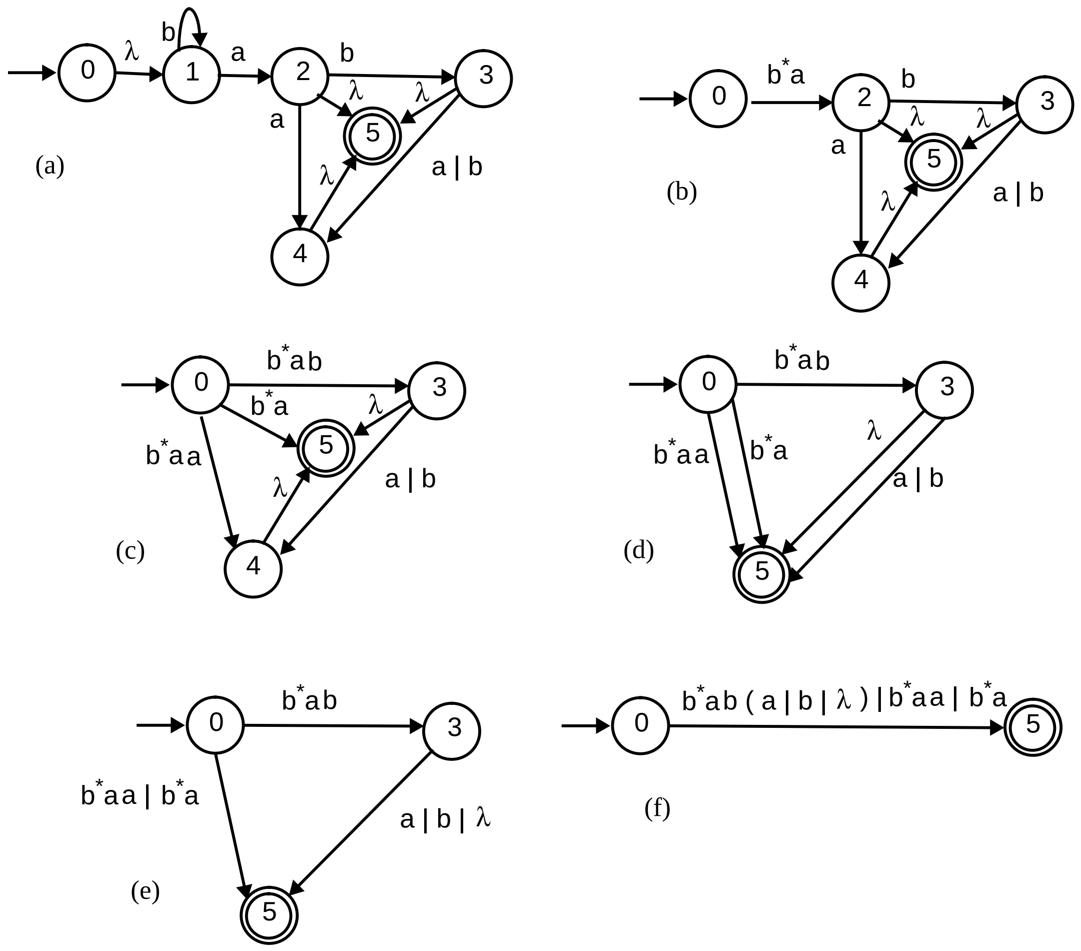


FIGURE 20 An Example of FINDRE

Summary

We have seen three equivalent and interchangeable mechanisms for defining tokens: regular expressions, deterministic finite automata and non-deterministic finite automata. 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. Non-deterministic finite automata form a middle ground. Sometimes they are used for definitional purposes, when it is convenient to just draw a simple automaton as a “flow diagram” of characters are to be matched. Sometimes non-deterministic finite automata are directly executed (see Exercise 17) when translation to deterministic finite automata is too costly or inconvenient. Familiarity with all three of these 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:

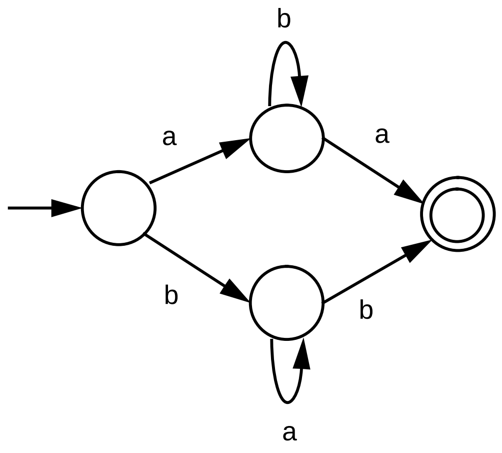
```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d  Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
}   }
```

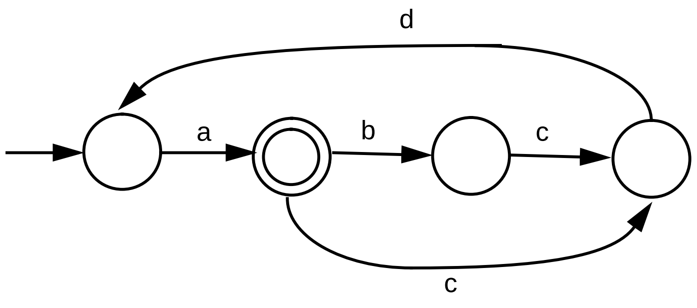
What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

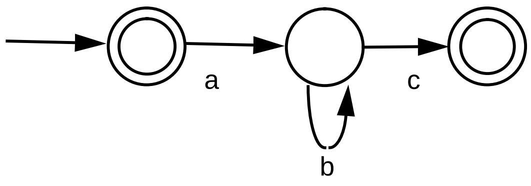
2. How many lexical errors (if any) appear in the following C program? How should each error be handled by the scanner?

```
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 following FAs:







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

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

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

$(a\ Not(a))^*\ aaa$

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.

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, it may be necessary 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 we wish to produce a prettyprinted listing (that is, a listing with text properly indented, `if-else` pairs aligned, and so on). How would you modify a Lex scanner specification to produce a prettyprinted listing?

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` produces seven tokens, whereas `DO 10 I = 1 . 10` produces three tokens. How would you design a scanner to handle the extended lookahead that FORTRAN requires?

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. Assume we are designing the ultimate programming language, “Utopia 2000.” We have already specified the language’s tokens using regular expressions and the language’s syntax using a context-free grammar. Now we wish to determine those token pairs that require white space to separate them (like `else a`) and those that require extra lookahead while scanning (like `10.0e-22`). Explain how we could use the regular expressions and context-free grammar 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 3.6:

$$(a\, b^* c) \mid (a\, b\, c^*)$$

Using MAKEDETERMINISTIC, translate the NFA into a DFA. Using the techniques of Section 3.6.2, optimize the DFA you created into a minimal state equivalent.

16. Consider the following regular expression: $(0 \mid 1)^* \, 0 \, (0 \mid 1) \, (0 \mid 1) \, (0 \mid 1) \ldots (0 \mid 1)$

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 3.6.1) while scanning. Rather than maintaining a single current state, a set of possible states is maintained. As characters are read, transitions from each state in the current set are followed, creating a new set of states. If any state in the current set is final, the characters read 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, following the set-of-states approach outlined above. 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 3.6.1 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. Assume we have scanned an integer literal into a character buffer (perhaps `yytext`). We wish 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 sets if you wish) that match the following sets of strings:

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

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

(c) The string x^{12345} (your solution should be far less than 12345 characters in length).

23. Write a Lex program that examines the words in an ASCII file and lists the ten most frequent 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 (e.g., 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 | s = ww }. Double contains only strings composed of two identical repeated pieces. For example, if we 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 we have a vocabulary consisting only of the single letter a. Is Double a regular set? Why?

Assume we 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, ...

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(λ) 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 non-deterministic finite automation that contains λ-transitions. Give an algorithm that transforms F into an equivalent non-deterministic finite automaton F′ that contains no λ-transitions.

Note: You need not use the "subset" construction since you're creating a non-deterministic finite automaton, not a deterministic finite automaton.

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 a finite automaton for R, construct one for Insert(R).

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