Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string *s* is any string obtained by removing zero or more symbols from the end of *s*. For example, ban, banana, and e are prefixes of banana.

2. A *suffix* of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*. For example, nana, banana, and e are suffixes of banana.

3. A *substring* of *s* is obtained by deleting any prefix and any suffix from *s*. For instance, banana, nan, and e are substrings of banana.

4. The *proper* prefixes, suffixes, and substrings of a string *s* are those, prefixes, suffixes, and substrings, respectively, of *s* that are not e or not equal to *s* itself.

5. A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*. For example, baan is a subsequence of banana.

If *x* and *y* are strings, then the *concatenation* of *x* and *y,* denoted *xy,* is the string formed by appending *y* to *x*. For example, if *x* = dog and *y* = house, then *xy* — doghouse. The empty string is the identity under concatenation; that is, for any string *s, es = se — s.*

If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be e, and for all $i > 0$, define s^i to be $s^{i-1}s$. Since es = *s,* it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss,$ and so on.

3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The *(Kleene) closure* of a language *L,* denoted *L*,* is the set of strings you get by concatenating *L* zero or more times. Note that L^0, the "concatenation of *L* zero times," is defined to be {e}, and inductively, L^i is $L^{i-1}L$. Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, e will not be in L^+ unless it is in *L* itself.

OPERATION ,	DEFINITION AND NOTATION
Union of *L* and *M*	$L \cup M = \{s \mid s$ is in *L* or *s* is in M $\}$
Concatenation of *L* and *M*	L M = $\{$ si $\mid s$ is in L and *t* is in M $\}$
Kleene closure of *L*	$L* = \cup \pounds_\circ$
Positive closure of *L*	

Figure 3.6: Definitions of operations on languages

Example **3.3** i Let *L* be the set of letters $\{A, B, \ldots, Z, \mathbf{a}, b, \ldots, z\}$ and let *D* be the set of digits $\{0, 1, \ldots 9\}$. We may think of *L* and *D* in two, essentially equivalent, ways. One way is that *L* and *D* are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that *L* and *D* are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages *L* and *D,* using the operators of Fig. 3.6:

1. *L* $\cup$ *D* is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.

2. *LD* is the set c-f 520 strings of length two, each consisting of one letter followed by one digit.

3. L^4 is the set of all 4-letter strings.

4. *L** is the set of all strings of letters, including e, the empty string.

5. *L(L* $\cup$ *D)** is the set of all strings of letters and digits beginning with a letter.

6. $D^.$ is the set of all strings of one or more digits.

•

3.3.3   Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter-* is established to stand for any letter or the underscore, and *digit-* is

established to stand for any digit, then we could describe the language of C identifiers by:

<div align="center">

letter- (letter- \ digit)*

</div>

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of *letter,* with the remainder of the expression signifies concatenation.

   The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language *L(r),* which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet £ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. e is a regular expression, and *L(e)* is {e}, that is, the language whose sole member is the empty string.

2. If *a* is a symbol in E, then **a** is a regular expression, and L(**a**) = *{a},* that is, the language with one string, of length one, with *a* in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

1. (r)|(s) is a regular expression denoting the language *L(r)* U *L(s).*

2. (r)(s) is a regular expression denoting the language *L(r)L(s).*

3. (r)* is a regular expression denoting (L(r))*.

4. (r) is a regular expression denoting *L(r).* This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

   As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

   however, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression **(a)|((b)*(c))** by **a|b***c. Both expressions denote the set of strings that are either a single *a* or are zero or more 6's followed by one c.

Example 3.4: Let £ = {a, 6}.

1. The regular expression **a|b** denotes the language *{a, b}.*

2. **(a|b)(a|b)** denotes *{aa, ah, ba, bb},* the language of all strings of length two over the alphabet E. Another regular expression for the same language is **aa|ab|ba|bb.**

3. **a*** denotes the language consisting of all strings of zero or more a's, that is, { e, *a, a a, a a a, … }.*

4. **(a|b)*** denotes the set of all strings consisting of zero or more instances of *a* or *b,* that is, all strings of a's and 6's: {e,*a, b,aa, ab, ba, bb,aaa,…}.* Another regular expression for the same language is **(a*b*)*.**

5. **a|a*b** denotes the language *{a, b, ab, aab, aaab,…},* that is, the string *a* and all strings consisting of zero or more a's and ending in *b.*

•

   A language that can be defined by a regular expression is called a *regular set.* If two regular expressions r and *s* denote the same regular set, we say they are *equivalent* and write *r = s.* For instance, **(a|b) = (b|a).** There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions *r, s,* and *t.*

LAW	DESCRIPTION
$r\backslash s = s\backslash r$	| is commutative
$r\backslash(s\backslash t) = (r\backslash s)\backslash t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\backslash t) = rs\backslash rt;\ (s\backslash t)r = sr\backslash tr$	Concatenation distributes over |
$er = re = r$	e is the identity for concatenation
$r* = (r|e)*$	e is guaranteed in a closure
	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

3.3.4   Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If £ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$di \quad \text{->} \quad \mathbf{n}$$
$$d_2 \quad \text{-»} \quad \mathbf{r}_2$$

$$d_n \quad \text{^} \quad r_n$$

where:

1. Each *di* is a new symbol, not in E and not the same as any other of the cTs, and

2. Each *T{* is a regular expression over the alphabet E U *{d\,d₂... . ,*

By restricting   to E and the previously defined **GTS**, we avoid recursive definitions, and we can construct a regular expression over E alone, for each r$. We do so by first replacing uses of $d\backslash$ in r_2 (which cannot use any of the d's except for $d\backslash$), then replacing uses of $d\backslash$ and d_2 in r-$ by $r\backslash$ and (the substituted) r_2, and so on. Finally, in r_n we replace each *di,* for $i — 1,2,\dots ,n — 1$, by the substituted version of r$, each of which has only symbols of E.

E x a m p l e 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\textit{letter-} \quad \text{-+} \quad \mathbf{A \mid B \mid - - - \mid Z \mid a \mid b \mid - - - \mid z \mid \_}$$
$$\textit{digit} \quad \text{->} \quad \mathbf{0 \text{ j } 1 \text{ j } \bullet\bullet\bullet \text{ j } 9}$$
$$\textit{id} \quad \text{—)} \bullet \quad \textit{letter- (letter- } \backslash \textit{ digit)*}$$

E x a m p l e 3.6: Unsigned numbers (integer or floating point) are strings such as **5280, 0.01234, 6.336E4,** or **1.89E-4.** The regular definition

$$\textit{digit} \quad\quad \mathbf{0 \mid 1} \quad\quad \mathbf{9}$$
$$\textit{digits} \quad \text{-+} \quad \textit{digit digit*}$$
$$\textit{optionalFraction} \quad \text{-)} \bullet \quad \text{. } \textit{digits} \mid \text{e}$$
$$\textit{optionalExponent} \quad\quad \mathbf{(E (+ \mid - \mid e) } \textit{digits}) \mid \text{e}$$
$$\textit{number} \quad \text{->} \quad \textit{digits optionalFraction optionalExponent}$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent,* if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match **1**., but does match **1.0.**
•

3.3.5   Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.*   The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $(L(r))^+$. The operator $^+$ has the same precedence and associativity as the operator *. Two useful algebraic laws, $r* = r^+|e$ and $r^+ = rr* = r*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to r|e, or put another way, $L(r?) = L(r) \cup \{e\}$. The ? operator has the same precedence and associativity as * and +.

3. *Character classes.*   A regular expression $aifal \cdots \backslash a_s$, where the a^s are each symbols of the alphabet, can be replaced by the shorthand $[aia,2 \cdots a_s]$. More importantly, when $01, 02, \ldots, a_s$ f°'" a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by $oi-a_s$, that is, just the first and last separated by a hyphen. Thus, **[abc]** is shorthand for **a|b|c,** and **[a-z]** is shorthand for **a|b|.--|z.**

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{array}{rcl} \textit{letter.} & \text{->} & \textbf{[A-Za-z\_]} \\ \textit{digit} & \text{->} & \textbf{[0-9]} \\ \textit{id} & \text{->} & \textit{letter- (letter } \backslash \textit{ digit)*} \end{array}$$

The regular definition of Example 3.6 can also be simplified:

$$\begin{array}{rcl} \textit{digit} & \text{->} & \textbf{[0-9]} \\ \textit{digits} & \text{—>•} & \textit{digit}^+ \\ \textit{number} & \text{-»•} & \textit{digits (. digits)?} \ (\ \textbf{E [+-]?} \ \textit{digits)?} \end{array}$$

•

3.3.6   Exercises for Section 3.3

Exercise 3.3.1: Consult the language reference manuals to determine *(i)* the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), *(ii)* the lexical form of numerical constants, and *(Hi)* the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! Exercise 3.3.2: Describe the languages denoted by the following regular expressions:

     a) **a(a|b)*a.**

     b) **((e|a)b*)*.**

     c) **(a|b)*a(a|b)(a|b).**

     d) **a*ba*ba*ba*.**

 !! e) **(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*.**

Exercise 3.3.3: In a string of length n, how many of the following are there?

     a) Prefixes.

     b) Suffixes.

     c) Proper prefixes.

  ! d) Substrings.

  ! e) Subsequences.

Exercise 3.3.4: Most languages are *case sensitive,* so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are *case insensitive,* so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword **SELECT** can also be written **select, Select,** or **sElEcT,** for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

! Exercise 3.3.5: Write regular definitions for the following languages:

     a) All strings of lowercase letters that contain the five vowels in order.

     b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

     c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (").

!! d) All strings of digits with no repeated digits. *Hint:* Try this problem first with a few digits, such as {0,1,2}.

!! e) All strings of digits with at most one repeated digit.

!! f) All strings of a's and 6's with an even number of a's and an odd number of 6's.

g) The set of Chess moves, in the informal notation, such as p-k4 or kbpxqn.

!! h) All strings of a's and 6's that do not contain the substring a66.

i) All strings of a's and 6's that do not contain the subsequence a66.

Exercise 3.3.6: Write character classes for the following sets of characters:

a) The first ten letters (up to "j") in either upper or lower case.

b) The lowercase consonants.

c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).

d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from Lex (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Fig. 3.8.

Exercise 3.3.7: Note that these regular expressions give all of the following symbols *(operator characters)* a special meaning:

\ » . ~ $ [ ] * + ? { } | /

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression "**" matches the string **. We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Exercise 3.3.8: In Lex, a *complemented character class* represents any character except the ones listed in the character class. We denote a complemented class by using " as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, [~A-Za-z] matches any character that is not an uppercase or lowercase letter, and [~\~] represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	**a**
V	character c literally	*
llgll	string s literally	"**"
	any character but newline	**a.** *b
	beginning of a line	~abc
$	end of a line	abc$
[*]	any one of the characters in string *s*	[abc]
	any one character not in string *s*	[~abc]
r*	zero or more strings matching *r*	**a***
r+	one or more strings matching *r*	**a+**
r?	zero or one r	**a?**
r{m, n}	between *m* and *n* occurrences of r	**a**[1,**5**]
rir₂	an ?*i followed by an *r₂*	ab
r\ | r₂	an ri or an *r₂*	a|b
(r)	same as r	(alb)
ri/r₂	7*1 when followed by *r₂*	abc/**123**

Figure **3.8: Lex** regular expressions

! **Exercise 3.3.9**: The regular expression *r{m, n]* matches from *m* to n occur-
rences of the pattern r. For example, **a[1,5]** matches a string of one to five a's.
Show that for every regular expression containing repetition operators of this
form, there is an equivalent regular expression without repetition operators.

! **Exercise 3.3.10**: The operator " matches the left end of a line, and $ matches
the right end of a line. The operator " is also used to introduce complemented
character classes, but the context always makes it clear which meaning is in-
tended. For example, **~[~aeiou]*$** matches any complete line that does not
contain a lowercase vowel.

   a) How do you tell which meaning of ^ is intended?

   b) Can you always replace a regular expression using the " and $ operators
      by an equivalent expression that does not use either of these operators?

! **Exercise 3.3.11**: The UNIX shell command **sh** uses the operators in Fig. **3.9**
in filename expressions to describe sets of file names. For example, the filename
expression ***.o** matches all file names ending in **.o; sortl.?** matches all file-
names of the form **sort**.c, where c is any character. Show how **sh** filename

EXPRESSION	MATCHES	EXAMPLE
v	string *s* literally	'v
V	character c literally	V
*	any string	*.o
?	any character	sortl.?
[s]	any character in *s*	sortl.[cso]

Figure 3.9: Filename expressions used by the shell command sh

expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

! Exercise 3.3.12: SQL allows a rudimentary form of patterns in which two characters have special meaning: underscore (_) stands for any one character and percent-sign (%) stands for any string of 0 or more characters. In addition, the programmer may define any character, say e, to be the escape character, so e preceding an e preceding _, % or another e gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

3.4   Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

$$
\begin{array}{rcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & e \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Figure 3.10: A grammar for branching statements

Example 3.8: The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

For **relop,** we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if, then, else, relop, id,** and **number,** are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. The patterns for *id* and *number* are similar to what we saw in Example 3.7.

$$
\begin{array}{rll}
digit & \rightarrow & [0\text{-}9] \\
digits & & digit^{+} \\
number & & digits\ (.\ digits)?\ (\ \mathbf{E}\ [\text{+-}]?\ digits\ )? \\
letter & & [\text{A-Za-z}] \\
id & & letter\ (\ letter\ \backslash\ digit\ )* \\
if & \rightarrow & \text{if} \\
then & & \mathbf{then} \\
else & & \mathbf{else} \\
relop & & <\ |\ >\ |\ <=\ |\ >=\ |\ =\ |\ <> \\
\end{array}
$$

Figure 3.11: Patterns for tokens of Example 3.8

For this language, the lexical analyzer will recognize the keywords if, **then,** and **else,** as well as lexemes that match the patterns for *relop, id,* and *number.* To simplify matters, we make the common assumption that keywords are also *reserved words:* that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" *ws* defined by:

$$ws \rightarrow (\ \mathbf{blank}\ |\ \mathbf{tab}\ |\ \mathbf{newline}\ )^{+}$$

Here, **blank, tab,** and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Our goal for the lexical analyzer is summarized in Fig. 3.12. That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, as discussed in Section 3.1.3, is returned. Note that for the six relational operators, symbolic constants **LT, LE,** and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found. The particular operator found will influence the code that is output from the compiler.   •

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	**-**	**-**
if	**if**	–
then	**then**	**-**
else	**else**	**-**
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Figure 3.12: Tokens, their patterns, and attribute values

3.4.1   Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams." In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand, but in Section 3.6, we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.

Transition diagrams have a collection of nodes or circles, called *states*. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3).

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state **5,** and the next input symbol is *a,* we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always

indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the *start state,* or *initial state;* it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Example 3.9: Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop.** We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.
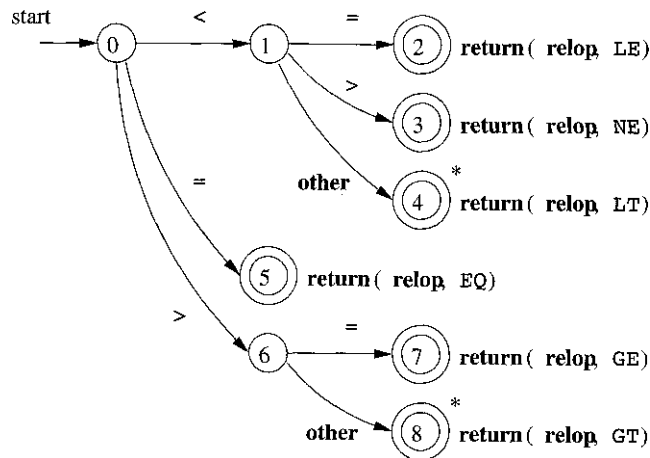

Figure 3.13: Transition diagram for **relop**

On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character). Note that if, in state 0, we see any character besides <, =, or >, we can not possibly be seeing a **relop** lexeme, so this transition diagram will not be used.   •

3.4.2   Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords if, **then,** and **else** of our running example.

letter or **digit**

start   **letter**   *C^)* **other**   */^zx* *
————*\9J*————————*\10)*————————**K®)**   **return(**getToken(),   *installlD* ())
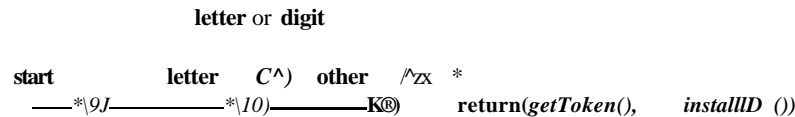
Figure 3.14: A transition diagram for **id**'s and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installlD* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id.** The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword **then** is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id,** with a lexeme like thenextvalue that has **then** as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word

tokens are recognized in preference to id, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

Figure 3.15: Hypothetical transition diagram for the keyword then

3.4.3   Completion of the Running Example

The transition diagram for id's that we saw in Fig. 3.14 has a simple structure. Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so. We stay in state 10 as long as the input contains letters and digits. When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found. Since the last character is not part of the identifier, we must retract the input one position, and as discussed in Section 3.4.2, we enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

The transition diagram for token number is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.
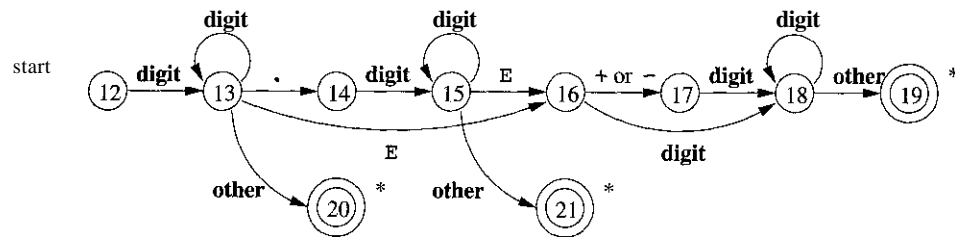

Figure 3.16: A transition diagram for unsigned numbers

If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an **E,** then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but **E** or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by **delim** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.
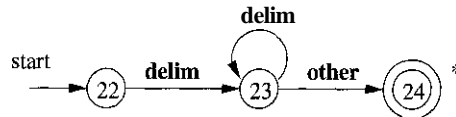

Figure 3.17: A transition diagram for whitespace

Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character. We retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

3.4.4   Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable s t a t e holding the number of the current state for a transition diagram. A switch based on the value of s t a t e takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Example 3.10: In Fig. 3.18 we see a sketch of getRelopO, a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type TOKEN, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case). getRelopO first creates a new object r e t Token and initializes its first component to RELOP, the symbolic code for token **relop.**

We see the typical behavior of a state in case 0, the case where the current state is 0. A function nextChar() obtains the next character from the input and assigns it to local variable c. We then check c for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case. For example, if the next input character is =, we go to state 5.

If the next input character is not one that can begin a comparison operator, then a function f a i l () is called. What f a i l () does depends on the global error-recovery strategy of the lexical analyzer. It should reset the forward pointer to lexemeBegin, in order to allow another transition diagram to be applied to

```
TOKEN getRelopO
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextCharQ;
                    if ( c == '<» ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state - 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...

            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
```

Figure **3.18:** Sketch of implementation of **relop** transition diagram

the true beginning of the unprocessed input. It might then change the value of **state** to be the start state for another transition diagram, which will search for another token. Alternatively, if there is no other transition diagram that remains unused, **fail()** could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section **3.1.4.**

We also show the action for state **8** in Fig. **3.18.** Because state **8** bears a *, we must retract the input pointer one position (i.e., put c back on the input stream). That task is accomplished by the function **retract** (). Since state **8** represents the recognition of lexeme $\geq$=, we set the second component of the returned object, which we suppose is named **attribute,** to GT, the code for this operator.   •

To place the simulation of one transition diagram in perspective, let us consider the ways code like Fig. **3.18** could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function **f ail()** of Example **3.10** resets the pointer **forward** and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one suggested in Fig. **3.15.** We have oniy to use these before we use the diagram for id, in order for the keywords to be reserved words.

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier thenext to keyword then, or the operator -> to -, for example.

3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.

3.4.5   Exercises for Section 3.4

Exercise 3.4.1: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

Exercise 3.4.2: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

The following exercises, up to Exercise 3.4.12, introduce the Aho-Corasick algorithm for recognizing a collection of keywords in a text string in time proportional to the length of the text and the sum of the length of the keywords. This algorithm uses a special form of transition diagram called a *trie*. A trie is a tree-structured transition diagram with distinct labels on the edges leading from a node to its children. Leaves of the trie represent recognized keywords.

Knuth, Morris, and Pratt presented an algorithm for recognizing a single keyword $b_1 b_2 \cdots b_n$ in a text string. Here the trie is a transition diagram with n states, 0 through n. State 0 is the initial state, and state n represents acceptance, that is, discovery of the keyword. From each state s from 0 through $n - 1$, there is a transition to state $s + 1$, labeled by symbol b_s. For example, the trie for the keyword ababaa is:

In order to process text strings rapidly and search those strings for a keyword, it is useful to define, for keyword $b_1 b_2 \cdots b_n$ and position s in that keyword (corresponding to state s of its trie), a *failure function, f(s),* computed as in

Fig. **3.19.**   The objective is that $b_1 b_2 \cdots b_{f(s)}$ is the longest proper prefix of $b_1 b_2 \cdots b_s$ that is also a suffix of $b_1 b_2 \cdots b_s$. The reason $f(s)$ is important is that if we are trying to match a text string for $b_1 b_2 \cdots b_n$, and we have matched the first s positions, but we then fail (i.e., the next position of the text string does not hold b_{s+1}), then $f(s)$ is the longest prefix of $b_1 b_2 \cdots b_n$ that could possibly match the text string up to the point we are at. Of course, the next character of the text string must be $b_{f(s)+1}$, or else we still have problems and must consider a yet shorter prefix, which will be $b_{f(f(s))}$.

```
1)   t = 0;
2)   f(1)=0;
3)   for (s = 1; s < n; s++) {
4)        while (t > 0 && b_{s+1} != b_{t+1})  t = f(t);
5)        if (b_{s+1} == b_{t+1}) {
6)            t = t + l;
7)            f(s + 1) = t;
         }
8)        else f(s + 1) = 0;
     }
```

Figure **3.19:** Algorithm to compute the failure function for keyword $b_1 b_2 \cdots b_n$

As an example, the failure function for the trie constructed above for **ababaa** is:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

For instance, states **3** and **1** represent prefixes **aba** and **a,** respectively. $f(3) = 1$ because **a** is the longest proper prefix of **aba** that is also a suffix of **aba.** Also, $f(2) = 0$, because the longest proper prefix of **ab** that is also a suffix is the empty string.

Exercise 3.4.3: Construct the failure function for the strings:

   a) **abababaab.**

   b) **aaaaaa.**

   c) **abbaabb.**

 ! Exercise 3.4.4: Prove, by induction on s, that the algorithm of Fig. **3.19** correctly computes the failure function.

 !! Exercise 3.4.5: Show that the assignment $t = f(t)$ in line **(4)** of Fig. **3.19** is executed at most n times. Show that therefore, the entire algorithm takes only $O(n)$ time on a keyword of length n.

Having computed the failure function for a keyword $b_1b_2 \cdots b_n$, we can scan a string $a_1a_2 \cdots a_m$ in time $O(m)$ to tell whether the keyword occurs in the string. The algorithm, shown in Fig. 3.20, slides the keyword along the string, trying to make progress by matching the next character of the keyword with the next character of the string. If it cannot do so after matching s characters, then it "slides" the keyword right $s - f(s)$ positions, so only the first $f(s)$ characters of the keyword are considered matched with the string.

```
1)    s = 0;
2)    for (i = 1; i < m; i++) {
3)            while (s > 0        a_i ! =  b_{s+1})  s = f(s);
4)            if,(a* == b_s+i)  s = s + 1;
5)            if (s == n) return "yes";
      }
6)    return "no";
```

Figure 3.20: The KMP algorithm tests whether string $a_1a_2 \cdots a_m$ contains a single keyword $b_1b_2 \cdots b_n$ as a substring in $O(m + n)$ time

Exercise 3.4.6: Apply Algorithm KMP to test whether keyword **ababaa** is a substring of:

  a) **abababaab.**

  b) **abababbaa.**

!! **Exercise 3.4.7:** Show that the algorithm of Fig. 3.20 correctly tells whether the keyword is a substring of the given string. *Hint:* proceed by induction on *i.* Show that for all *i,* the value of *s* after line **(4)** is the length of the longest prefix of the keyword that is a suffix of $a_1a_2 \cdots a_i$.

!! **Exercise 3.4.8**: Show that the algorithm of Fig. 3.20 runs in time $O(m + n)$, assuming that function f is already computed and its values stored in an array indexed by *s.*

Exercise 3.4.9: The *Fibonacci strings* are defined as follows:

  1. s_1 = **b.**

  2. s_2 = **a.**

  3. $s_k = s_{k-1}s_{k-2}$ for $k > 2$.

For example, s_3 = **ab,** s_4 = **aba,** and s_5 = **abaab.**

  a) What is the length of s_n?

  b) Construct the failure function for *SQ.*

  c) Construct the failure function for *sj.*

!! d) Show that the failure function for any s_n can be expressed by $/(1) = /(2) = 0$, and for $2 < j < |s_n|$, $f(j)$ is $j - |s_n\text{-}i|$, where k is the largest integer such that $< j + 1$.

!! e) In the KMP algorithm, what is the largest number of consecutive applica-tions of the failure function, when we try to determine whether keyword *Sk* appears in text string s^+i?

   Aho and Corasick generalized the KMP algorithm to recognize any of a set of keywords in a text string.   In this case, the trie is a true tree, with branching from the root.   There is one state for every string that is a prefix (not necessarily proper) of any keyword.   The parent of a state corresponding to string $b \backslash b_2 \bullet \bullet \bullet bk$ is the state that corresponds to &1&2 • • • frfc-i- A state is accepting if it corresponds to a complete keyword.   For example, Fig. 3.21 shows the trie for the keywords he, she, his, and hers.
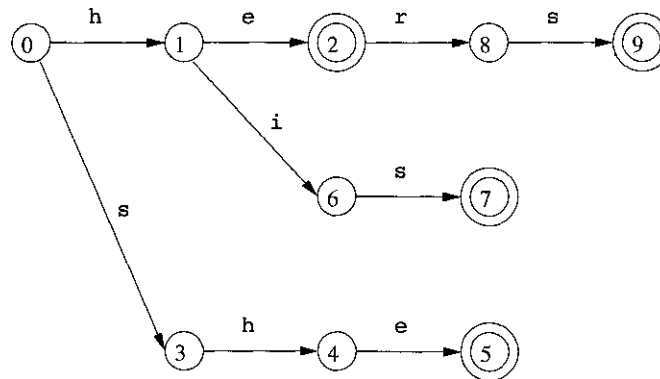

Figure 3.21: Trie for keywords he, she, his, hers

   The failure function for the general trie is defined as follows.   Suppose *s* is the state that corresponds to string $\&i\&_2 \bullet \bullet \bullet b_n$.   Then *f(s)* is the state that corresponds to the longest proper suffix of $\&i6_2 \bullet \bullet \text{-}b_n$ that is also a prefix of *some* keyword.   For example, the failure function for the trie of Fig. 3.21 is:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! Exercise 3.4.10: Modify the algorithm of Fig. 3.19 to compute the failure function for general tries.   *Hint:* The major difference is that we cannot simply test for equality or inequality of b_{n} and $b_n i$ in lines (4) and (5) of Fig. 3.19. Rather, from any state there may be several transitions out on several charac-ters, as there are transitions on both e and i from state 1 in Fig. 3.21.   Any of

those transitions could lead to a state that represents the longest suffix that is also a prefix.

Exercise 3.4.11: Construct the tries and compute the failure function for the following sets of keywords:

   a) aaa, abaaa, and ababaaa.

   b) all, fall, fatal, llama, and lame.

   c) pipe, pet, item, temper, and perpetual.

! Exercise 3.4.12: Show that your algorithm from Exercise 3.4.10 still runs in time that is linear in the sum of the lengths of the keywords.

3.5   The Lexical-Analyzer Generator Lex

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

3.5.1   Use of Lex

Figure 3.22 suggests how Lex is used. An input file, which we call lex.l, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c. The latter file is compiled by the C compiler into a file called a.out, as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as a.out in Fig. 3.22, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable yylval,[2] which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

incidentally, the **yy** that appears in **yylval** and **lex.yy.c** refers to the **Yacc** parser-generator, which we shall describe in Section 4.9, and which is commonly used in conjunction with **Lex.**

Lex source program
lex.l   Lex
                                compiler   **lex.yy.c**

lex.yy.c ———————| C
                     | compiler |———————— **a.out**

Input stream ————————| **a.out** |———————— Sequence of tokens
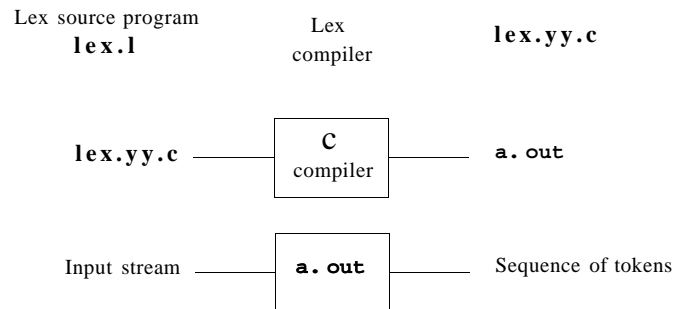
Figure **3.22:** Creating a lexical analyzer with **Lex**

3.5.2 Structure of Lex Programs

A **Lex** program has the following form:

> declarations
> **%%**
> translation rules
> %%
> auxiliary functions

The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions, in the style of Section **3.3.4.**

The translation rules each have the form

> Pattern   { Action }

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of **Lex** using other languages have been created.

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns P_i. It then executes the associated action A_i. Typically, A_i will return to the parser, but if it does not (e.g., because P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

Example 3.11: Figure 3.23 is a Lex program that recognizes the tokens of Fig. 3.12 and returns the token found. A few observations about this code will introduce us to many of the important features of Lex.

In the declarations section we see a pair of special brackets, %{ and %}. Anything within these brackets is copied directly to the file lex.yy.c, and is not treated as a regular definition. It is common to place there the definitions of the manifest constants, using C #define statements to associate unique integer codes with each of the manifest constants. In our example, we have listed in a comment the names of the manifest constants, LT, **IF,** and so on, but have not shown them defined to be particular integers.[3]

Also in the declarations section is a sequence of regular definitions. These use the extended notation for regular expressions described in Section 3.3.5. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, *delim* is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively. Then, *ws* is defined to be one or more delimiters, by the regular expression {delim}+.

Notice that in the definition of *id* and *number,* parentheses are used as grouping metasymbols and do not stand for themselves. In contrast, E in the definition of *number* stands for itself. If we wish to use one of the Lex meta-symbols, such as any of the parentheses, +, *, or ?, to stand for themselves, we may precede them with a backslash. For instance, we see \. in the definition of *number,* to represent the dot, since that character is a metasymbol representing "any character," as usual in UNIX regular expressions.

In the auxiliary-function section, we see two such functions, installlD() and installNumO. Like the portion of the declaration section that appears between %{...%}, everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.

Finally, let us examine some of the patterns and rules in the middle section of Fig. 3.23. First, *ws,* an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern if. Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for id), then the lexical analyzer consumes these two letters from the input and returns the token name **IF,** that is, the integer for which the manifest constant **IF** stands. Keywords then and else are treated similarly.

The fifth token has the pattern defined by *id.* Note that, although keywords like if match this pattern as well as an earlier pattern, Lex chooses whichever

---

[3]If **Lex** is used along with **Yacc,** then it would be normal to define the manifest constants in the **Yacc** program and use them without definition in the **Lex** program. Since **lex.yy.c** is compiled with the **Yacc** output, the constants thus will be available to the actions in the **Lex** program.

U

```
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNumO ; return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%

int installlDQ {/* function to install the lexeme, whose
                first character is pointed to by yytext,
                arid whose length is yyleng, into the
                symbol table and return a pointer
                thereto */
}

int installNumO {/* similar to installlD, but puts numer-
                ical constants into a separate table */
}
```

Figure **3.23:** **Lex** program for the tokens of Fig. **3.12**

pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when *id* is matched is threefold:

1. Function installID() is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler. Note that installID() has available to it two variables that are set automatically by the lexical analyzer that Lex generates:

   (a) yytext is a pointer to the beginning of the lexeme, analogous to lexemeBegin in Fig. 3.3.

   (b) yyleng is the length of the lexeme found.

3. The token name ID is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function installNumO.   •

3.5.3   Conflict Resolution in Lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Example 3.12: The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme. The second rule makes keywords reserved, if we list the keywords before id in the program. For instance, if then is determined to be the longest prefix of the input that matches any pattern, and the pattern then precedes {id}, as it does in Fig. 3.23, then the token THEN is returned, rather than ID.   •

3.5.4   The Lookahead Operator

Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the

pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

Example 3.13: In Fortran and some other languages, keywords are not re-served. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where **IF** is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where **IF** is a keyword. Fortunately, we can be sure that the keyword **IF** is always followed by a left parenthesis, some text — the condition —- that may contain parentheses, a right parenthesis and a letter. Thus, we could write a **Lex** rule for the keyword **IF** like:

```
IF / \(  .* \) {letter}
```

This rule says that the pattern the lexeme matches is just the two letters **IF.** The slash says that additional pattern follows but does not match the lexeme. In this pattern, the first character is the left parentheses. Since that character is a **Lex** metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning. The dot and star match "any string without a newline." Note that the dot is a **Lex** metasymbol meaning "any character except newline." It is followed by a right parenthesis, again with a backslash to give that character its literal meaning. The additional pattern is followed by the symbol *letter,* which is a regular definition representing the character class of all letters.

Note that in order for this pattern to be foolproof, we must preprocess the input to delete whitespace. We have in the pattern neither provision for whitespace, nor can we deal with the possibility that the condition extends over lines, since the dot will not match a newline character.

For instance, suppose this pattern is asked to match a prefix of input:

```
IF(A<(B+C)*D)THEN...
```

the first two characters match **IF,** the next character matches \(, the next nine characters match .*, and the next two match \) and *letter*. Note the fact that the first right parenthesis (after C) is not followed by a letter is irrelevant; we only need to find some way of matching the input to the pattern. We conclude that the letters **IF** constitute the lexeme, and they are an instance of token **if.**
•

3.5.5   Exercises for Section 3.5

Exercise 3.5.1: Describe how to make the following modifications to the **Lex** program of Fig. **3.23:**

a) Add the keyword while.

b) Change the comparison operators to be the C operators of that kind.

c) Allow the underscore (_) as an additional letter.

! d) Add a new pattern with token **STRING.** The pattern consists of a double-quote ("), any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash (\), and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to escape a character removed. Strings are to be installed in a table of strings.

Exercise 3.5.2: Write a **Lex** program that copies a file, replacing each non-empty sequence of white space by a single blank.

Exercise 3.5.3: Write a **Lex** program that copies a C program, replacing each instance of the keyword **float** by **double.**

! **Exercise 3.5.4:** Write a **Lex** program that converts a file to "Pig latin." Specifically, assume the file is a sequence of words (groups of letters) separated by whitespace. Every time you encounter a word:

  1. If the first letter is a consonant, move it to the end of the word and then add **ay.**

  2. If the first letter is a vowel, just add **ay** to the end of the word.

All nonletters are copied intact to the output.

! **Exercise 3.5.5:** In SQL, keywords and identifiers are case-insensitive. Write a **Lex** program that recognizes the keywords **SELECT, FROM,** and **WHERE** (in any combination of capital and lower-case letters), and token **ID,** which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the "install" function would differ from that described for case-sensitive identifiers as in Fig. **3.23.**

3.6   Finite Automata

We shall now discover how Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as *finite automata.* These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers;* they simply say "yes" or "no" about each possible input string.

2. Finite automata come in two flavors:

    (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and e, the empty string, is a possible label.

    (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the *regular languages,* that regular expressions can describe.[4]

3.6.1   Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states 5.

2. A set of input symbols E, the *input alphabet.* We assume that e, which stands for the empty string, is never a member of E.

3. A *transition function* that gives, for each state, and for each symbol in E U {e} a set of *next states.*

4. A state so from *S* that is distinguished as the *start state* (or *initial state).*

5. A set of states *F,* a subset of *S,* that is distinguished as the *accepting states* (or *final   states).*

We can represent either an NFA or DFA by a *transition graph,* where the nodes are states and the labeled edges represent the transition function. There is an edge labeled *a* from state *s* to state *t* if and only if *t* is one of the next states for state *s* and input *a.* This graph is very much like a transition diagram, except:

[4]There is a small lacuna: as we defined them, regular expressions cannot describe the empty language, since we would never want to use this pattern in practice. However, finite automata **can** define the empty language. In the theory, 0 is treated as an additional regular expression for the sole purpose of defining the empty language.

a) The same symbol can label edges from one state to several different states, and

b) An edge may be labeled by e, the empty string, instead of, or in addition to, symbols from the input alphabet.

Example 3.14: The transition graph for an NFA recognizing the language of regular expression (a|b)*abb is shown in Fig. 3.24. This abstract example, describing all strings of a's and &'s ending in the particular string *abb,* will be used throughout this section. It is similar to regular expressions that describe languages of real interest, however. For instance, an expression describing all files whose name ends in .o is any*.o, where any stands for any printable character.
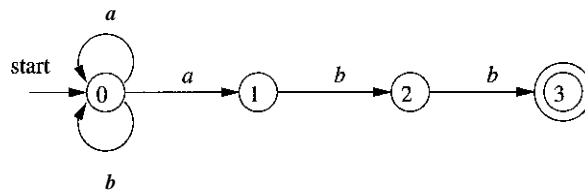
Figure 3.24: A nondeterministic finite automaton

Following our convention for transition diagrams, the double circle around state 3 indicates that this state is accepting. Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading *abb* from the input. Thus, the only strings getting to the accepting state are those that end in *abb.*   •

3.6.2   Transition Tables

We can also represent an NFA by a *transition table,* whose rows correspond to states, and whose columns correspond to the input symbols and e. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put 0 in the table for the pair.

Example 3.15: The transition table for the NFA of Fig. 3.24 is shown in Fig. 3.25.   •

The transition table has the advantage that we can easily find the transitions on a given state and input. Its disadvantage is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

STATE	a	b	e
0	{ 0 , 1 }	{ 0 }	0
1	0	{ 2 }	0
2	0	{ 3 }	0
3	0	0	0

Figure 3.25: Transition table for the NFA of Fig. 3.24

3.6.3 Acceptance of Input Strings by Automata

An NFA *accepts* input string *x* if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out *x*. Note that e labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

Example 3.16: The string *aabb* is accepted by the NFA of Fig. 3.24. The path labeled by *aabb* from state 0 to state 3 demonstrating this fact is:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$
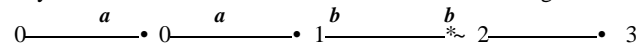
Note that several paths labeled by the same string may lead to different states. For instance, path

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$
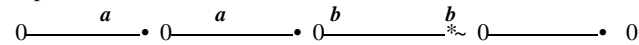
is another path from state 0 labeled by the string *aabb*. This path leads to state 0, which is not accepting. However, remember that an NFA accepts a string as long as *some* path labeled by that string leads from the start state to an accepting state. The existence of other paths leading to a nonaccepting state is irrelevant.   •

   The *language defined* (or *accepted)* by an NFA is the set of strings labeling some path from the start to an accepting state. As was mentioned, the NFA of Fig. 3.24 defines the same language as does the regular expression **(a|b)*abb,** that is, all strings from the alphabet *{a, b}* that end in *abb*. We may use *L(A)* to stand for the language accepted by automaton *A*.
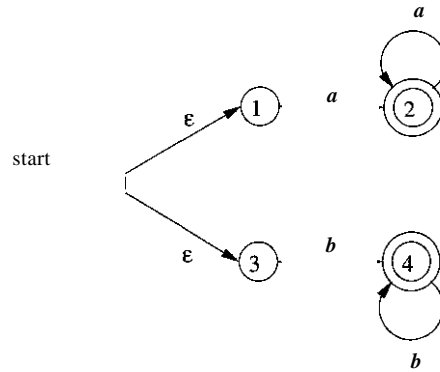
Example 3.17: Figure 3.26 is an NFA accepting L**(aa*|bb*).** String *aaa* is accepted because of the path

$$0 \xrightarrow{e} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$
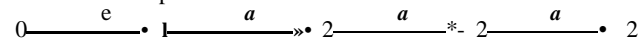
Note that e's "disappear" in a concatenation, so the label of the path is *aaa*.
•

3.6.4 Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a special case of an NFA where:

Figure 3.26: NFA accepting **aa*|bb***

1. There are no moves on input e, and

2. For each state *s* and input symbol a, there is exactly one edge out of *s* labeled *a.*

If we are using a transition table to represent a DFA, then each entry is a single state, we may therefore represent this state without the curly braces that we use to form sets.

   While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers. The following algorithm shows how to apply a DFA to a string.

Algorithm 3.18: Simulating a DFA.

INPUT: An input string *x* terminated by an end-of-file character **eof.** A DFA *D* with start state *so,* accepting states *F,* and transition function *move.*

OUTPUT: Answer "yes" if *D* accepts *x;* "no" otherwise.

METHOD: Apply the algorithm in Fig. 3.27 to the input string *x.* The function *move(s,c)* gives the state to which there is an edge from state *s* on input c. The function *next Char* returns the next character of the input string *x.* •

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language **(a|b)*abb,** the same as that accepted by the NFA of Fig. 3.24. Given the input string *ababb,* this DFA enters the sequence of states 0,1,2,1,2,3 and returns "yes."   •
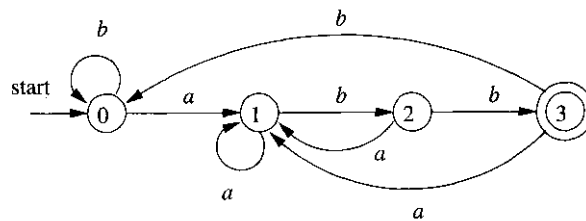
```
8 = s,]
c  =  nextCharQ;
while ( c != eof) {
        s  =  move(s,c);
        c  =  nextCharQ;
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

Figure 3.28: DFA accepting (a|b)*abb

3.6.5   Exercises for Section 3.6

! **Exercise 3.6.1**: Figure **3.19** in the exercises of Section **3.4** computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword $bib_2 \cdots b_n$ an. $n + 1$-state DFA that recognizes $*bib_2 \cdots b_n$, where the dot stands for "any character." Moreover, this DFA can be constructed in $O(n)$ time.

Exercise 3.6.2: Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise **3.3.5.**

Exercise 3.6.3: For the NFA of Fig. **3.29,** indicate all the paths labeled *aabb*. Does the NFA accept *aabb?*
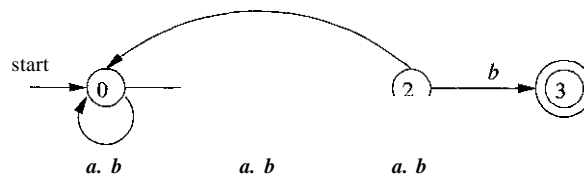
Figure **3.29:** NFA for Exercise **3.6.3**
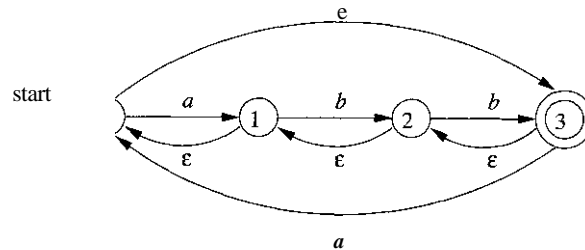
Figure 3.30: NFA for Exercise 3.6.4

Exercise 3.6.4: Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

Exercise 3.6.5: Give the transition tables for the NFA of:

   a) Exercise 3.6.3.

   b) Exercise 3.6.4.

   c) Figure 3.26.

3.7   Prom Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software, as was reflected in Section 3.5. However, implementation of that software requires the simulation of a DFA, as in Algorithm 3.18, or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol (as Fig. 3.24 does oh input *a* from state 0) or on e (as Fig. 3.26 does from state 0), or even a choice of making a transition on e or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

   In this section we shall first show how to convert NFA's to DFA's. Then, we use this technique, known as "the subset construction," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation. Next, we show how to convert regular expressions to NFA's, from which a DFA can be constructed if desired. We conclude with a discussion of the time-space tradeoffs inherent in the various methods for implementing regular expressions, and see how to choose the appropriate method for your application.

3.7.1   Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states.   After reading input

flifl2 • • • $Q>n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a\backslash a_2$ • • • a_i.

It is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA. However, part of the power of the automaton-based approach to lexical analysis is that for real languages, the NFA and DFA have approximately the same number of states, and the exponential behavior is not seen.

Algorithm 3.20: The *subset construction* of a DFA from an NFA.

INPUT: An NFA JV.

OUTPUT: A DFA *D* accepting the same language as *N*.

METHOD: Our algorithm constructs a transition table *Dtran* for *D*. Each state of *D* is a set of NFA states, and we construct *Dtran* so *D* will simulate "in parallel" all possible moves *N* can make on a given input string. Our first problem is to deal with e-transitions of *N* properly. In Fig. 3.31 we see the definitions of several functions that describe basic computations on the states of *N* that are needed in the algorithm. Note that *s* is a single state of *N,* while *T* is a set of states of *N*.

OPERATION	DESCRIPTION
e-closure(s)	Set of NFA states reachable from NFA state *s* on e-transitions alone.
e-closure(T)	Set of NFA states reachable from some NFA state *s* in set *T* on e-transitions alone; $= U_s i_n T$ *e-closure(s).*
move(T, a)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T*.

Figure 3.31: Operations on NFA states

We must explore those sets of states that *N* can be in after seeing some input string. As a basis, before reading the first input symbol, *N* can be in any of the states of *e-closure(so),* where *SQ* is its start state. For the induction, suppose that *N* can be in set of states *T* after reading input string *x*. If it next reads input a, then *N* can immediately go to any of the states in *move(T, a)*. However, after reading a, it may also make several e-transitions; thus *N* could be in any state of *e-closure(move(T, a))* after reading input *xa*. Following these ideas, the construction of the set of Z?'s states, *Dstates,* and its transition function *Dtran,* is shown in Fig. 3.32.

The start state of *D* is *e-closure(so),* and the accepting states of *D* are all those sets of ATs states that include at least one accepting state of *N*. To complete our description of the subset construction, we need only to show how

initially, *e-closure(s₀)* is the only state in *Dstates,* and it is unmarked;
while (there is an unmarked state *T* in *Dstates*) {
      mark *T;*
      for (each input symbol *a*) {
          U   =   e-closure(move(T,a));
          if (*U* is not in *Dstates*)
             add *U* as an unmarked state to *Dstates;*
          Dtran[T, a] = U;
      }
}

Figure 3.32: The subset construction

e-closure(T) is computed for any set of NFA states *T*. This process, shown in
Fig. 3.33, is a straightforward search in a graph from a set of states. In this
case, imagine that only the e-labeled edges are available in the graph.   •

push all states of *T* onto *stack;*
initialize *e~closure(T)* to *T;*
while (*stack* is not empty) {
      pop *t,* the top element, off *stack;*
      for (each state *u* with an edge from *t* to *u* labeled e)
          if (*u* is not in *e-closure(T)*) {
             add *u* to *e-closure(T);*
             push *u* onto *stack;*
          }
    }

Figure 3.33: Computing *e-closure(T)*

Example 3.21: Figure 3.34 shows another NFA accepting **(a|b)*abb;** it hap-
pens to be the one we shall construct directly from this regular expression in
Section 3.7. Let us apply Algorithm 3.20 to Fig. 3.29.

The start state *A* of the equivalent DFA is *e-closure(0),* or *A* = {0,1,2,4,7},
since these are exactly the states reachable from state 0 via a path all of whose
edges have label e. Note that a path can have zero edges, so state 0 is reachable
from itself by an e-labeled path.

The input alphabet is *{a, b}.* Thus, our first step is to mark *A* and compute
Dtran[A,a]   =   e-closure(move(A,a))   and   *Dtran[A,b]   =   e-closure(move(A,b)).*
Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to
3 and 8, respectively. Thus, *move(A,a)* = {3,8}. Also, *e-closure({3,8}* =
{1,2,3,4,6,7,8}, so we conclude
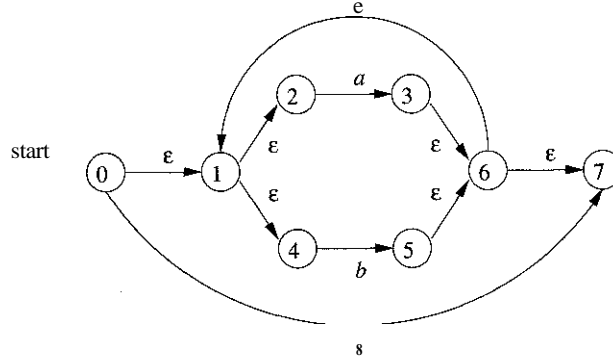
Figure 3.34: NFA *N* for **(a|b)*abb**

Dtran[A,a]   =   e-closure(move(A,a))   —   e-closure({3,8})   =   { 1 , 2 , 3 , 4 , 6 , 7 , 8 }

Let us call this set *B*, so *Dtran[A, a] = B*.

Now, we must compute *Dtran[A,b]*. Among the states in *A,* only 4 has a transition on 6, and it goes to 5. Thus,

Dtran[A,b]   =   e-closure({5})   =   { 1 , 2 , 4 , 6 , 7 }

Let us call the above set C, so *Dtran[A, b] — C*.

NFA STATE	DFA STATE	*a*	*b*
{ 0 , 1 , 2 , 4 , 7 }	*A*	*B*	*C*
{ 1 , 2 , 3 , 4 , 6 , 7 , 8 }	*B*	*B*	*D*
{ 1 , 2 , 4 , 5 , 6 , 7 }	*C*	*B*	*C*
{ 1 , 2 , 4 , 5 , 6 , 7 , 9 }	*D*	*B*	*E*
{ 1 , 2 , 3 , 5 , 6 , 7 , 1 0 }	*E*	*B*	*C*

Figure 3.35: Transition table *Dtran* for DFA *D*

If we continue this process with the unmarked sets *B* and C, we eventually reach a point where all the states of the DFA are marked. This conclusion is guaranteed, since there are "only" 2^{11} different subsets of a set of eleven NFA states. The five different DFA states we actually construct, their corresponding sets of NFA states, and the transition table for the DFA *D* are shown in Fig. 3.35, and the transition graph for *D* is in Fig. 3.36. State *A* is the start state, and state *E,* which contains state 10 of the NFA, is the only accepting state.

Note that *D* has one more state than the DFA of Fig. 3.28 for the same language. States *A* and *C* have the same move function, and so can be merged. We discuss the matter of minimizing the number of states of a DFA in Section 3.9.6.
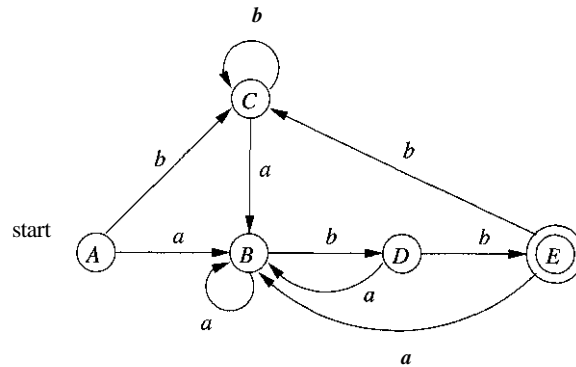.

Figure 3.36: Result of applying the subset construction to Fig. 3.34

3.7.2   Simulation of an NFA

A strategy that has been used in a number of text-editing programs is to con-
struct an NFA from a regular expression and then simulate the NFA using
something like an on-the-fly subset construction.   The simulation is outlined
below.

Algorithm 3.22: Simulating an NFA.

INPUT: An input string x terminated by an end-of-file character **eof.** An NFA
N with start state *SQ,* accepting states *F,* and transition function *move.*

OUTPUT: Answer "yes" if *M* accepts *x;* "no" otherwise.

METHOD: The algorithm keeps a set of current states *S,* those that are reached
from so following a path labeled by the inputs read so far. If c is the next input
character, read by the function *nextCharQ,* then we first compute *move(S,c)*
and then close that set using *e-closureQ.* The algorithm is sketched in Fig. 3.37.
•

```
1)   S  =  e-closure(so);
2)   c  =  nextCharQ;
3)   while ( c != eof) {
4)          S    =    e-closure(move(S,c));
5)          c  =  nextCharQ;
6)   }
7)   if ( 5 n F != 0 ) return "yes";
8)   else return "no";
```

Figure 3.37: Simulating an NFA

3.7.3   Efficiency of NFA Simulation

If carefully implemented, Algorithm 3.22 can be quite efficient. As the ideas involved are useful in a number of similar algorithms involving search of graphs, we shall look at this implementation in additional detail. The data structures we need are:

1. Two stacks, each of which holds a set of NFA states. One of these stacks, *oldStates,* holds the "current" set of states, i.e., the value of *S* on the right side of line (4) in Fig. 3.37. The second, *newStates,* holds the "next" set of states — 5 on the left side of line (4). Unseen is a step where, as we go around the loop of lines (3) through (6), *newStates* is transferred to *oldStates.*

2. A boolean array *alreadyOn,* indexed by the NFA states, to indicate which states are in *newStates.* While the array and stack hold the same information, it is much faster to interrogate *alreadyOn[s]* than to search for state *s* on the stack *newStates.* It is for this efficiency that we maintain both representations.

3. A two-dimensional array *move[s,a]* holding the transition table of the NFA. The entries in this table, which are sets of states, are represented by linked lists.

To implement line (1) of Fig. 3.37, we need to set each entry in array *alreadyOn* to **FALSE,** then for each state *s* in *e-closure(so),* push *s* onto *oldStates* and set *alreadyOn[s]* to **TRUE.** This operation on state *s,* and the implementation of line (4) as well, are facilitated by a function we shall call *addState(s).* This function pushes state *s* onto *newStates,* sets *alreadyOn[s]* to **TRUE,** and calls itself recursively on the states in *move[s,e]* in order to further the computation of *e-closure(s).* However, to avoid duplicating work, we must be careful never to call *addState* on a state that is already on the stack *newStates.* Figure 3.38 sketches this function.

```
 9)    addState(s)   {
10)          push  s  onto  newStates;
11)          alreadyOn[s]  =  TRUE;
12)          for (  t  on  move[s, e]  )
13)                  if ( !alreadyOn(t) )
14)                          addState(t);
15)    }
```

Figure 3.38: Adding a new state *s,* which is known not to be on *newStates*

We implement line (4) of Fig. 3.37 by looking at each state *s* on *oldStates.* We first find the set of states *move[s,* c], where c is the next input, and for each

of those states that is not already on *newStates,* we apply *addState* to it. Note that *addState* has the effect of computing the e- *closure* and adding all those states to *newStates* as well, if they were not already on. This sequence of steps is summarized in Fig. 3.39.

```
16)    for  ( s o n  oldStates )  {
17)         for  ( t on  move[s, c]  )
18)              if  (  \alreadyOn[t]  )
19)                        addState(t);
20)         pop  s  from  oldStates;
21)    }

22)    for  ( s  on  newStates )  {
23)         pop  s  from  newStates;
24)         push  s  onto  oldStates;
25)         a/readyOn[s]  =  FALSE;
26)    }
```

Figure 3.39: Implementation of step (4) of Fig. 3.37

Now, suppose that the NFA N has n states and m transitions; i.e., m is the sum over all states of the number of symbols (or e) on which the state has a transition out. Not counting the call to *addState* at line (19) of Fig. 3.39, the time spent in the loop of lines (16) through (21) is *0(n).* That is, we can go around the loop at most n times, and each step of the loop requires constant work, except for the time spent in *addState.* The same is true of the loop of lines (22) through (26).

During one execution of Fig. 3.39, i.e., of step (4) of Fig. 3.37, it is only possible to call *addState* on a given state once. The reason is that whenever we call *addState(s),* we set *alreadyOn[s]* to TRUE at line (11) of Fig. 3.39. Once *alreadyOn[s]* is TRUE, the tests at line (13) of Fig. 3.38 and line (18) of Fig. 3.39 prevent another call.

The time spent in one call to *addState,* exclusive of the time spent in recursive calls at line (14), is $0(1)$ for lines (10) and (11). For lines (12) and (13), the time depends on how many e-transitions there are out of state s. We do not know this number for a given state, but we know that there are at most m transitions in total, out of all states. As a result, the aggregate time spent in lines (11) over all calls to *addState* during one execution of the code of Fig. 3.39 is *0(m).* The aggregate for the rest of the steps of *addState* is *0(n),* since it is a constant per call, and there are at most n calls.

We conclude that, implemented properly, the time to execute line (4) of Fig. 3.37 is *0(n* + m). The rest of the while-loop of lines (3) through (6) takes $0(1)$ time per iteration. If the input x is of length $k,$ then the total work in that loop is *0((k(n* + m)). Line (1) of Fig. 3.37 can be executed in $0(n + m)$ time, since it is essentially the steps of Fig. 3.39 with *oldStates* containing only

Big-Oh Notation

An expression like $O(n)$ is a shorthand for "at most some constant times n." Technically, we say a function $/(n)$, perhaps the running time of some step of an algorithm, is $O(g(n))$ if there are constants c and no, such that whenever $n > n_0$, it is true that $/(n) < cg(n)$. A useful idiom is "$O(1)$," which means "some constant." The use of this *big-oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time, yet lets us express the rate at which the running time of an algorithm grows.

the state so- Lines (2), (7), and (8) each take $O(1)$ time. Thus, the running time of Algorithm 3.22, properly implemented, is $O((k(n + m))$. That is, the time taken is proportional to the length of the input times the size (nodes plus edges) of the transition graph.

3.7.4 Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that defines the same language. The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression. For each subexpression the algorithm constructs an NFA with a single accepting state.

Algorithm 3.23: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

INPUT: A regular expression r over alphabet S.

OUTPUT: An NFA N accepting $L(r)$.

METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

BASIS: For expression e construct the NFA
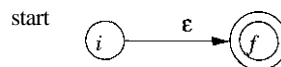
Here, i is a new state, the start state of this NFA, and $/$ is another new state, the accepting state for the NFA.

For any subexpression a in S, construct the NFA
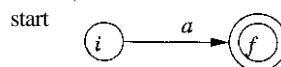
where again *i* and / are new states, the start and accepting states, respectively. Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of e or some o as a subexpression of *r*.

INDUCTION: Suppose *N(s)* and *N(t)* are NFA's for regular expressions *s* and *t,* respectively.

a) Suppose *r = s\t.* Then *N(r),* the NFA for r, is constructed as in Fig. 3.40. Here, *i* and / are new states, the start and accepting states of *N(r),* respectively. There are e-transitions from *i* to the start states of *N(s)* and *N(t),* and each of their accepting states have e-transitions to the accepting state /. Note that the accepting states of *N(s)* and *N(t)* are not accepting in *N(r).* Since any path from *i* to / must pass through either *N(s)* or *N(t)* exclusively, and since the label of that path is not changed by the e's leaving *i* or entering /, we conclude that *N(r)* accepts *L(s)* U *L(t),* which is the same as *L(r).* That is, Fig. 3.40 is a correct construction for the union operator.
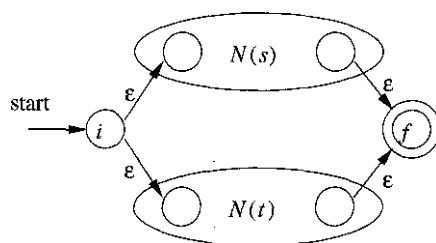

Figure 3.40: NFA for the union of two regular expressions

b) Suppose r = *st.* Then construct *N(r)* as in Fig. 3.41. The start state of *N(s)* becomes the start state of *N(r),* and the accepting state of *N(t)* is the only accepting state of *N(r).* The accepting state of *N(s)* and the start state of *N(t)* are merged into a single state, with all the transitions in or out of either state. A path from *i* to / in Fig. 3.41 must go first through *N(s),* and therefore its label will begin with some string in *L(s).* The path then continues through *N(t),* so the path's label finishes with a string in *L(t).* As we shall soon argue, accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter *N(s)* after leaving it. Thus, *N(r)* accepts exactly *L(s)L(i),* and is a correct NFA for *r = st.*
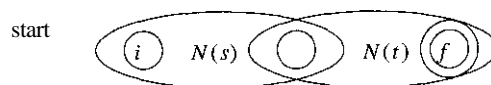

Figure 3.41: NFA for the concatenation of two regular expressions

c) Suppose $r = s*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.42.
Here, i and / are new states, the start state and lone accepting state of
$N(r)$. To get from i to /, we can either follow the introduced path labeled
e, which takes care of the one string in $L(s)^\circ$, or we can go to the start
state of $N(s)$, through that NFA, then from its accepting state back to
its start state zero or more times. These options allow $N(r)$ to accept all
the strings in $L(s)^1$, $L(s)^2$, and so on, so the entire set of strings accepted
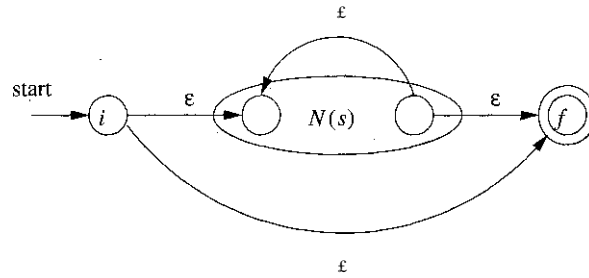by $N(r)$ is $L(s*)$.

Figure 3.42: NFA for the closure of a regular expression

d) Finally, suppose r $= (s)$. Then $L(r) = L(s)$, and we can use the NFA
$N(s)$ as $N(r)$.

•

The method description in Algorithm 3.23 contains hints as to why the
inductive construction works as it should. We shall not give a formal correctness
proof, but we shall list several properties of the constructed NFA's, in addition
to the all-important fact that $N(r)$ accepts language $L(r)$.   These properties
are interesting in their own right, and helpful in making a formal proof.

1. $N(r)$ has at most twice as many states as there are operators and operands
in r. This bound follows from the fact that each step of the algorithm
creates at most two new states.

2. $N(r)$ has one start state and one accepting state. The accepting state has
no outgoing transitions, and the start state has no incoming transitions.

3. Each state of $N(r)$ other than the accepting state has either one outgoing
transition on a symbol in E or two outgoing transitions, both on e.

Example 3.24: Let us use Algorithm 3.23 to construct an NFA for $r =$
$(a|b)*abb$. Figure 3.43 shows a parse tree for r that is analogous to the parse
trees constructed for arithmetic expressions in Section 2.2.3. For subexpression
ri, the first a, we construct the NFA:

10

Figure 3.43: Parse tree for **(a|b)*abb**

start

State numbers have been chosen for consistency with what follows. For r_2 we construct:

start

• © -

We can now combine JV(n) and $N(r_2)$, using the construction of Fig. 3.40 to obtain the NFA for $r_3 = ri\backslash r_2]$ this NFA is shown in Fig. 3.44.
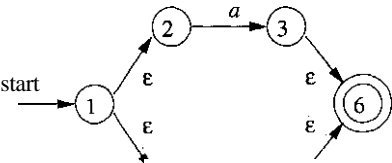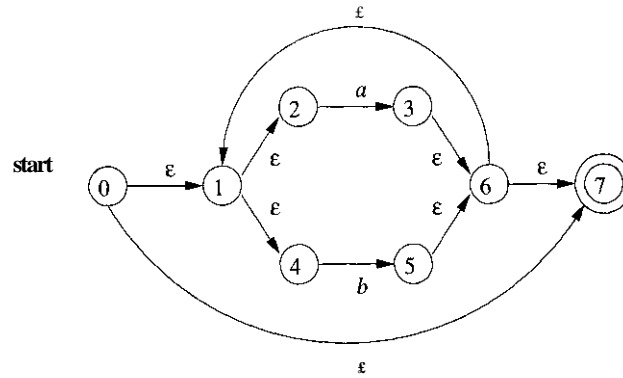
Figure 3.44: NFA for r_3

The NFA for $r_4 = (73)$ is the same as that for 7-3. The NFA for $r_5 = (r_3)^*$ is then as shown in Fig. 3.45. We have used the construction in Fig. 3.42 to build this NFA from the NFA in Fig. 3.44.

Now, consider subexpression r_6, which is another **a.** We use the basis construction for a again, but we must use new states. It is not permissible to reuse

Figure **3.45:** NFA for r,

the NFA we constructed for $r\backslash$, even though $r\backslash$ and $r\%$ are the same expression. The NFA for r_6 is:
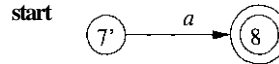
To obtain the NFA for $rr = r_5 re$, we apply the construction of Fig. **3.41.** We merge states **7** and **7',** yielding the NFA of Fig. **3.46.** Continuing in this fashion with new NFA's for the two subexpressions **b** called rg and rio, we eventually construct the NFA for **(a|b)*abb** that we first met in Fig. **3.34.**   •
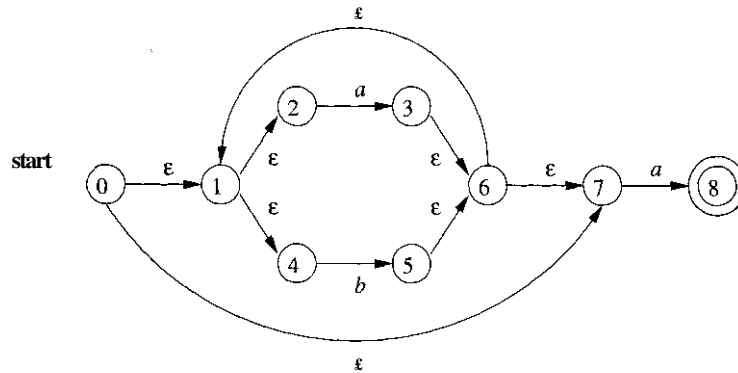
Figure **3.46:** NFA for r,

3.7.5   Efficiency of String-Processing Algorithms

We observed that Algorithm **3.18** processes a string x in time $\mathbf{0(|a?|)}$, while in Section **3.7.3** we concluded that we could simulate an NFA in time proportional to the product of $\backslash x\backslash$ and the size of the NFA's transition graph. Obviously, it

is faster to have a DFA to simulate than an NFA, so we might wonder whether it ever makes sense to simulate an NFA.

One issue that may favor an NFA is that the subset construction can, in the worst case, exponentiate the number of states. While in principle, the number of DFA states does not influence the running time of Algorithm 3.18, should the number of states become so large that the transition table does not fit in main memory, then the true running time would have to include disk I/O and therefore rise noticeably.

Example 3.25: Consider the family of languages described by regular expressions of the form $L_n = (\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$, that is, each language L_n consists of strings of a's and &'s such that the nth character to the left of the right end holds a. An n + 1-state NFA is easy to construct. It stays in its initial state under any input, but also has the option, on input a, of going to state 1. From state 1, it goes to state 2 on any input, and so on, until in state n it accepts. Figure 3.47 suggests this NFA.
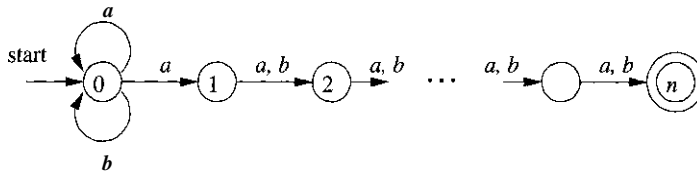

Figure 3.47: An NFA that has many fewer states than the smallest equivalent DFA

However, any DFA for the language L_n must have at least 2^n states. We shall not prove this fact, but the idea is that if two strings of length n can get the DFA to the same state, then we can exploit the last position where the strings differ (and therefore one must have a, the other *b)* to continue the strings identically, until they are the same in the last $n - 1$ positions. The DFA will then be in a state where it must both accept and not accept. Fortunately, as we mentioned, it is rare for lexical analysis to involve patterns of this type, and we do not expect to encounter DFA's with outlandish numbers of states in practice.   •

However, lexical-analyzer generators and other string-processing systems often start with a regular expression. We are faced with a choice of converting the regular expression to an NFA or DFA. The additional cost of going to a DFA is thus the cost of executing Algorithm 3.23 on the NFA (one could go directly from a regular expression to a DFA, but the work is essentially the same). If the string-processor is one that will be executed many times, as is the case for lexical analysis, then any cost of converting to a DFA is worthwhile. However, in other string-processing applications, such as **grep,** where the user specifies one regular expression and one or several files to be searched for the pattern

of that expression, it may be more efficient to skip the step of constructing a DFA, and simulate the NFA directly.

Let us consider the cost of converting a regular expression r to an NFA by Algorithm 3.23. A key step is constructing the parse tree for *r*. In Chapter 4 we shall see several methods that are capable of constructing this parse tree in linear time, that is, in time $0(|r|)$, where $|r|$ stands for the *size* of *r* — the sum of the number of operators and operands in r. It is also easy to check that each of the basis and inductive constructions of Algorithm 3.23 takes constant time, so the entire time spent by the conversion to an NFA is $0([r|)$.

Moreover, as we observed in Section 3.7.4, the NFA we construct has at most $|r|$ states and at most $2\backslash r\backslash$ transitions. That is, in terms of the analysis in Section 3.7.3, we have $n < \backslash r\backslash$ and $m < 2\backslash r\backslash$. Thus, simulating this NFA on an input string *x* takes time *$0(\backslash r\backslash$ x $\backslash x\backslash)$*. This time dominates the time taken by the NFA construction, which is $0(|r|)$, and therefore, we conclude that it is possible to take a regular expression r and string *x,* and tell whether *x* is in *L(r)* in time *$0(\backslash r\backslash$ x $\backslash x\backslash)$*.

The time taken by the subset construction is highly dependent on the number of states the resulting DFA has. To begin, notice that in the subset construction of Fig. 3.32, the key step, the construction of a set of states *U* from a set of states *T* and an input symbol a, is very much like the construction of a new set of states from the old set of states in the NFA simulation of Algorithm 3.22. We already concluded that, properly implemented, this step takes time at most proportional to the number of states and transitions of the NFA.

Suppose we start with a regular expression r and convert it to an NFA. This NFA has at most $|r|$ states and at most $2\backslash r\backslash$ transitions. Moreover, there are at most $|r|$ input symbols. Thus, for every DFA state constructed, we must construct at most $|r|$ new states, and each one takes at most *$0(\backslash r\backslash + 2\backslash r\backslash)$* time. The time to construct a DFA of *s* states is thus *$0(\backslash r\backslash^{2}s)$*.

In the common case where *s* is about $|r|$, the subset construction takes time *$0(\backslash r\backslash^{3})$*. However, in the worst case, as in Example 3.25, this time is *$0(\backslash r\backslash^{2}2^{\wedge})$*. Figure 3.48 summarizes the options when one is given a regular expression *r* and wants to produce a recognizer that will tell whether one or more strings *x* are in *L(r)*.

AUTOMATON	INITIAL	PER STRING
NFA	$0(|r|)$	*$0(\backslash r\backslash$ x $\backslash x\backslash)$*
D F A typical case	$0(|rf)$	$0(|s|)$
D F A worst case	$0(|rf 2M)$	$0(|*|)$

Figure 3.48: Initial cost and per-string-cost of various methods of recognizing the language of a regular expression

If the per-string cost dominates, as it does when we build a lexical analyzer,

we clearly prefer the DFA. However, in commands like grep, where we run the automaton on only one string, we generally prefer the NFA. It is not until $|x|$ approaches $|r|^3$ that we would even think about converting to a DFA.

There is, however, a mixed strategy that is about as good as the better of the NFA and the DFA strategy for each expression r and string *x*. Start off simulating the NFA, but remember the sets of NFA states (i.e., the DFA states) and their transitions, as we compute them. Before processing the current set of NFA states and the current input symbol, check to see whether we have already computed this transition, and use the information if so.

3.7.6   Exercises for Section 3.7

Exercise 3.7.1: Convert to DFA's the NFA's of:

   a) Fig. 3.26.

   b) Fig. 3.29.

   c) Fig. 3.30.

Exercise 3.7.2: use Algorithm 3.22 to simulate the NFA's:

   a) Fig. 3.29.

   b) Fig. 3.30.

on input *aabb*.

Exercise 3.7.3:   Convert the following regular expressions to deterministic finite automata, using algorithms 3.23 and 3.20:

   a) **(a|b)*.**

   b) **(a*|b*)*.**

   c) **((e|a)b*)*.**

   d) **(a|b)*abb(a|b)*.**

3.8   Design of a Lexical-Analyzer Generator

In this section we shall apply the techniques presented in Section 3.7 to see how a lexical-analyzer generator such as Lex is architected. We discuss two approaches, based on NFA's and DFA's; the latter is essentially the implementation of Lex.