```
fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

3.1.5 Exercises for Section 3.1

Exercise 3.1.1: Divide the following C++ program:

```
float limitedSquare(x) float x; {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

! Exercise 3.1.2: Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <B>m
<P><IMG SRC = "house.gi
See <A HREF = "morePix.
liked that one.<P>
```

into appropriate lexemes. Which
and what should those values be?

3.2 Input Buffering

Before discussing the problem of re
some ways that the simple but in
can be speeded. This task is m
to look one or more characters be
we have the right lexeme. The b
Tokens" in Section 3.1 gave an ex
where we need to look at least o
we cannot be sure we've seen the
that is not a letter or digit, and t
C, single-character operators like
two-character operator like ->, ==
scheme that handles large lookahe
involving "sentinels" that saves ti

3.2.1 Buffer Pairs

Because of the amount of time tak
of characters that must be proces
program, specialized buffering te
amount of overhead required to p
tant scheme involves two buffers t
Fig. 3.3.

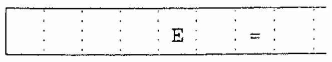

Figure 3.3: Us

Each buffer is of the same size
e.g., 4096 bytes. Using one syste
into a buffer, rather than using o
characters remain in the input file

```
Here is a photo of <B>my house</B>:
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. The box on "Tricky Problems When Recognizing Tokens" in Section 3.1 gave an extreme example, but there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like `-`, `=`, or `<` could also be the beginning of a two-character operator like `->`, `==`, or `<=`. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

3.2.1 Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.

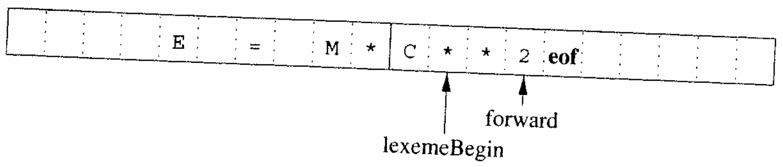

Figure 3.3: Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**,

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 (*iii*) 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) $\mathbf{a}(\mathbf{a}|\mathbf{b})^*\mathbf{a}$.

b) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

d) $\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*$.

!! e) $(\mathbf{aa}|\mathbf{bb})^*((\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*(\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*)^*$.

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 lexemes are 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 b's with an even number of a's and an odd number of b's.

g) The set of Chess moves, in the informal notation, such as `p-k4` or `kbp`$\times$`qn`.

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

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

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	M
c	the one nc
$\backslash c$	character
"s"	string s lit
.	any charac
^	beginning
$	end of a li
$[s]$	any one of
$[\hat{}s]$	any one ch
$r*$	zero or mc
$r+$	one or moi
$r?$	zero or one
$r\{m, n\}$	between m
$r_1 r_2$	an r_1 follo
$r_1 \mid r_2$	an r_1 or ai
(r)	same as r
r_1/r_2	r_1 when fc

Figure 3.8:

! **Exercise 3.3.9:** The regular exp rences of the pattern r. For examp Show that for every regular expr form, there is an equivalent regul

! **Exercise 3.3.10:** The operator the right end of a line. The opera character classes, but the contex tended. For example, `^[^aeiou]` contain a lowercase vowel.

a) How do you tell which mea

b) Can you always replace a r by an equivalent expression

! **Exercise 3.3.11:** The UNIX sh in filename expressions to describ expression `*.o` matches all file n names of the form `sort1.c`, whe

```
s = s_0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

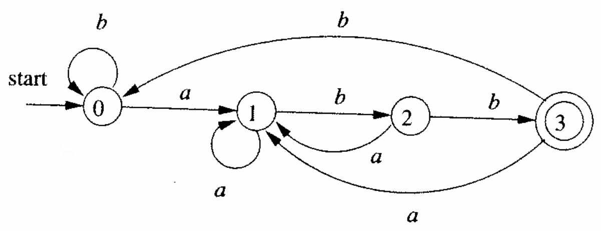

Figure 3.28: DFA accepting **(a|b)*abb**

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 $b_1b_2\cdots b_n$ an $n+1$-state DFA that recognizes $.^*b_1b_2\cdots b_n$, 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*?

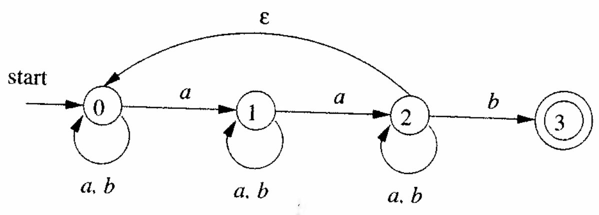

Figure 3.29: NFA for Exercise 3.6.3

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) $(\mathbf{a}|\mathbf{b})^*$.

b) $(\mathbf{a}^*|\mathbf{b}^*)^*$.

c) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

d) $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}(\mathbf{a}|\mathbf{b})^*$.

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.

3.8.1 The Structure of t

Figure 3.49 overviews the architect
The program that serves as the lex
simulates an automaton; at this po
is deterministic or nondeterministic
components that are created from t

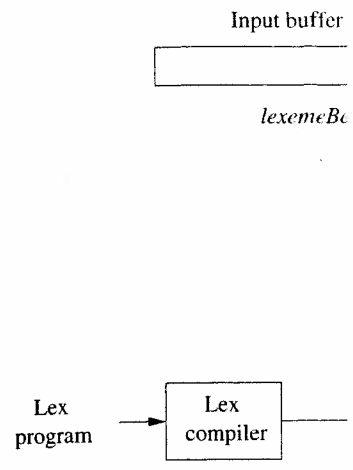

Figure 3.49: A Lex program is turn
are used by a finite-automaton sim

These components are:

1. A transition table for the aut

2. Those functions that are pas
Section 3.5.2).

3. The actions from the input p
to be invoked at the appropri

To construct the automaton, w
pattern in the Lex program and cor
We need a single automaton that
patterns in the program, so we cor
a new start state with ϵ-transition
for pattern p_i. This construction is

Example 3.26: We shall illustrat
simple, abstract example:

is valid, and the next state for state s on input a is $next[l]$. If $check[l] \neq s$, then we determine another state $t = default[s]$ and repeat the process as if t were the current state. More formally, the function $nextState$ is defined as follows:

```
int nextState(s, a) {
        if ( check[base[s] + a] == s ) return next[base[s] + a];
        else return nextState(default[s], a);
}
```

The intended use of the structure of Fig. 3.66 is to make the *next-check* arrays short by taking advantage of the similarities among states. For instance, state t, the default for state s, might be the state that says "we are working on an identifier," like state 10 in Fig. 3.14. Perhaps state s is entered after seeing the letters `th`, which are a prefix of keyword `then` as well as potentially being the prefix of some lexeme for an identifier. On input character `e`, we must go from state s to a special state that remembers we have seen `the`, but otherwise, state s behaves as t does. Thus, we set $check[base[s] + \texttt{e}]$ to s (to confirm that this entry is valid for s) and we set $next[base[s] + \texttt{e}]$ to the state that remembers `the`. Also, $default[s]$ is set to t.

While we may not be able to choose *base* values so that no *next-check* entries remain unused, experience has shown that the simple strategy of assigning *base* values to states in turn, and assigning each $base[s]$ value the lowest integer so that the special entries for state s are not previously occupied utilizes little more space than the minimum possible.

3.9.9 Exercises for Section 3.9

Exercise 3.9.1: Extend the table of Fig. 3.58 to include the operators (a) ? and (b) +.

Exercise 3.9.2: Use Algorithm 3.36 to convert the regular expressions of Exercise 3.7.3 directly to deterministic finite automata.

! Exercise 3.9.3: We can prove that two regular expressions are equivalent by showing that their minimum-state DFA's are the same up to renaming of states. Show in this way that the following regular expressions: $(\mathbf{a}|\mathbf{b})^*$, $(\mathbf{a}^*|\mathbf{b}^*)^*$, and $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$ are all equivalent. *Note*: You may have constructed the DFA's for these expressions in response to Exercise 3.7.3.

! Exercise 3.9.4: Construct the minimum-state DFA's for the following regular expressions:

a) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})$.

b) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

Do you see a pattern?

!! Exercise 3.9.5: To make form
that any deterministic finite aut

$(\mathbf{a}|\mathbf{b})^*$

where $(\mathbf{a}|\mathbf{b})$ appears $n-1$ times
Observe the pattern in Exercise .
inputs does each state represent?

3.10 Summary of C

✦ *Tokens.* The lexical analyz
output a sequence of tokens
the parser. Some tokens m
may also have an associate
the particular instance of tl

✦ *Lexemes.* Each time the le
it has an associated lexeme
token represents.

✦ *Buffering.* Because it is oft
order to see where the next
lexical analyzer to buffer its
ending each buffer's contents
techniques that accelerate tl

✦ *Patterns.* Each token has a
characters can form the lex
of words, or strings of chara
language.

✦ *Regular Expressions.* These
patterns. Regular expressi
union, concatenation, and t
ator.

✦ *Regular Definitions.* Comple
terns that describe the toke
fined by a regular definition,
define one variable to stand
pression for one variable can
expression.