study of natural languages. Their use in specifying the syntax of programming languages arose independently. While working with a draft of Algol 60, John Backus "hastily adapted [Emil Post's productions] to that use" (Wexelblat [1981, p.162]). The resulting notation was a variant of context-free grammars. The scholar Panini devised an equivalent syntactic notation to specify the rules of Sanskrit grammar between 400 B.C. and 200 B.C. (Ingerman [1967]).

The proposal that BNF, which began as an abbreviation of Backus Normal Form, be read as Backus-Naur Form, to recognize Naur's contributions as editor of the Algol 60 report (Naur [1963]), is contained in a letter by Knuth [1964].

Syntax-directed definitions are a form of inductive definitions in which the induction is on the syntactic structure. As such they have long been used informally in mathematics. Their application to programming languages came with the use of a grammar to structure the Algol 60 report. Shortly thereafter, Irons [1961] constructed a syntax-directed compiler.

Recursive-descent parsing has been used since the early 1960's. Bauer [1976] attributes the method to Lucas [1961]. Hoare [1962b, p.128] describes an Algol compiler organized as "a set of procedures, each of which is capable of processing one of the syntactic units of the Algol 60 report." Foster [1968] discusses the elimination of left recursion from productions containing semantic actions that do not affect attribute values.

McCarthy [1963] advocated that the translation of a language be based on abstract syntax. In the same paper McCarthy [1963, p.24] left "the reader to convince himself" that a tail-recursive formulation of the factorial function is equivalent to an iterative program.

The benefits of partitioning a compiler into a front end and a back end were explored in a committee report by Strong et al. [1958]. The report coined the name UNCOL (from universal computer oriented language) for a universal intermediate language. The concept has remained an ideal.

A good way to learn about implementation techniques is to read the code of existing compilers. Unfortunately, code is not often published. Randell and Russell [1964] give a comprehensive account of an early Algol compiler. Compiler code may also be seen in McKeeman, Horning, and Wortman [1970]. Barron [1981] is a collection of papers on Pascal implementation, including implementation notes distributed with the Pascal P compiler (Nori et al. [1981]), code generation details (Ammann [1977]), and the code for an implementation of Pascal S, a Pascal subset designed by Wirth [1981] for student use. Knuth [1985] gives an unusually clear and detailed description of the TeX translator.

Kernighan and Pike [1984] describe in detail how to build a desk calculator program around a syntax-directed translation scheme using the compiler-construction tools available on the UNIX operating system. Equation (2.17) is from Tantzen [1963].

CHAPTER 3

Lexical Analysis

This chapter deals with techniques for specifying and implementing lexical analyzers. A simple way to build a lexical analyzer is to construct a diagram that illustrates the structure of the tokens of the source language, and then to hand-translate the diagram into a program for finding tokens. Efficient lexical analyzers can be produced in this manner.

The techniques used to implement lexical analyzers can also be applied to other areas such as query languages and information retrieval systems. In each application, the underlying problem is the specification and design of programs that execute actions triggered by patterns in strings. Since pattern-directed programming is widely useful, we introduce a pattern-action language called Lex for specifying lexical analyzers. In this language, patterns are specified by regular expressions, and a compiler for Lex can generate an efficient finite-automaton recognizer for the regular expressions.

Several other languages use regular expressions to describe patterns. For example, the pattern-scanning language AWK uses regular expressions to select input lines for processing and the UNIX system shell allows a user to refer to a set of file names by writing a regular expression. The UNIX command `rm *.o`, for instance, removes all files with names ending in "`.o`".[1]

A software tool that automates the construction of lexical analyzers allows people with different backgrounds to use pattern matching in their own application areas. For example, Jarvis [1976] used a lexical-analyzer generator to create a program that recognizes imperfections in printed circuit boards. The circuits are digitally scanned and converted into "strings" of line segments at different angles. The "lexical analyzer" looked for patterns corresponding to imperfections in the string of line segments. A major advantage of a lexical-analyzer generator is that it can utilize the best-known pattern-matching algorithms and thereby create efficient lexical analyzers for people who are not experts in pattern-matching techniques.

[1] The expression `*.o` is a variant of the usual notation for regular expressions. Exercises 3.10 and 3.14 mention some commonly used variants of regular expression notations.

3.1 THE ROLE OF THE LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction, summarized schematically in Fig. 3.1, is commonly implemented by making the lexical analyzer be a subroutine or a coroutine of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

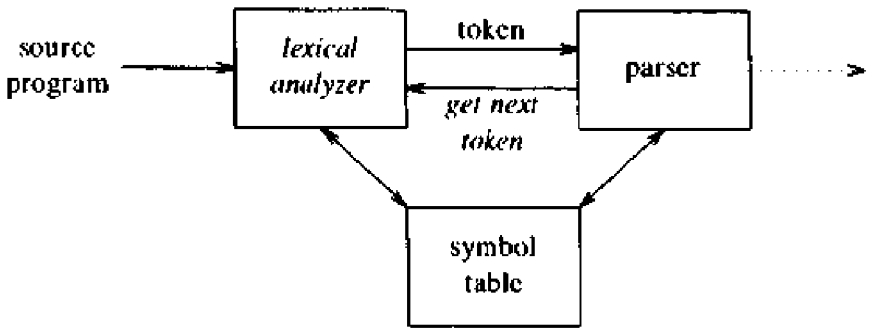


Fig. 3.1. Interaction of lexical analyzer with parser.

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, tab, and newline characters. Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message. In some compilers, the lexical analyzer is in charge of making a copy of the source program with the error messages marked in it. If the source language supports some macro preprocessor functions, then these preprocessor functions may also be implemented as lexical analysis takes place.

Sometimes, lexical analyzers are divided into a cascade of two phases, the first called "scanning," and the second "lexical analysis." The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations. For example, a Fortran compiler might use a scanner to eliminate blanks from the input.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

1. Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify

one or the other of these phases. For example, a parser embodying the conventions for comments and white space is significantly more complex than one that can assume comments and white space have already been removed by a lexical analyzer. If we are designing a new language, separating the lexical and syntactic conventions can lead to a cleaner overall language design.

2. Compiler efficiency is improved. A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. A large amount of time is spent reading the source program and partitioning it into tokens. Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

3. Compiler portability is enhanced. Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer. The representation of special or non-standard symbols, such as ↑ in Pascal, can be isolated in the lexical analyzer.

Specialized tools have been designed to help automate the construction of lexical analyzers and parsers when they are separated. We shall see several examples of such tools in this book.

Tokens, Patterns, Lexemes

When talking about lexical analysis, we use the terms "token," "pattern," and "lexeme" with specific meanings. Examples of their use are shown in Fig. 3.2. In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a *pattern* associated with the token. The pattern is said to *match* each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. For example, in the Pascal statement

```
const pi = 3.1416;
```

the substring `pi` is a lexeme for the token "identifier."

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	`<` or `<=` or `=` or `<>` or `>=` or `>`
id	`pi, count, D2`	letter followed by letters and digits
num	`3.1416, 0, 6.02E23`	any numeric constant
literal	`"core dumped"`	any characters between " and " except "

Fig. 3.2. Examples of tokens.

We treat tokens as terminal symbols in the grammar for the source language, using boldface names to represent tokens. The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit.

In most programming languages, the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons. In the example above, when the character sequence `pi` appears in the source program, a token representing an identifier is returned to the parser. The returning of a token is often implemented by passing an integer corresponding to the token. It is this integer that is referred to in Fig. 3.2 as boldface **id**.

A pattern is a rule describing the set of lexemes that can represent a particular token in source programs. The pattern for the token **const** in Fig. 3.2 is just the single string `const` that spells out the keyword. The pattern for the token **relation** is the set of all six Pascal relational operators. To describe precisely the patterns for more complex tokens like **id** (for identifier) and **num** (for number) we shall use the regular-expression notation developed in Section 3.3.

Certain language conventions impact the difficulty of lexical analysis. Languages such as Fortran require certain constructs in fixed positions on the input line. Thus the alignment of a lexeme may be important in determining the correctness of a source program. The trend in modern language design is toward free-format input, allowing constructs to be positioned anywhere on the input line, so this aspect of lexical analysis is becoming less important.

The treatment of blanks varies greatly from language to language. In some languages, such as Fortran or Algol 68, blanks are not significant except in literal strings. They can be added at will to improve the readability of a program. The conventions regarding blanks can greatly complicate the task of identifying tokens.

A popular example that illustrates the potential difficulty of recognizing tokens is the DO statement of Fortran. In the statement

```
DO 5 I = 1.25
```

we cannot tell until we have seen the decimal point that DO is not a keyword, but rather part of the identifier DO5I. On the other hand, in the statement

```
DO 5 I = 1,25
```

we have seven tokens, corresponding to the keyword DO, the statement label 5, the identifier I, the operator =, the constant 1, the comma, and the constant 25. Here, we cannot be sure until we have seen the comma that DO is a keyword. To alleviate this uncertainty, Fortran 77 allows an optional comma between the label and index of the DO statement. The use of this comma is encouraged because it helps make the DO statement clearer and more readable.

In many languages, certain strings are *reserved*; i.e., their meaning is predefined and cannot be changed by the user. If keywords are not reserved, then the lexical analyzer must distinguish between a keyword and a user-defined identifier. In PL/I, keywords are not reserved; thus, the rules for distinguishing keywords from identifiers are quite complicated as the following PL/I statement illustrates:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Attributes for Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern **num** matches both the strings 0 and 1, but it is essential for the code generator to know what string was actually matched.

The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions; the attributes influence the translation of tokens. As a practical matter, a token has usually only a single attribute — a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token. For diagnostic purposes, we may be interested in both the lexeme for an identifier and the line number on which it was first seen. Both these items of information can be stored in the symbol-table entry for the identifier.

Example 3.1. The tokens and associated attribute-values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs:

<**id**, pointer to symbol-table entry for E>

<**assign_op**, >

<**id**, pointer to symbol-table entry for M>

<**mult_op**, >

<**id**, pointer to symbol-table entry for C>

<**exp_op**, >

<**num**, integer value 2>

Note that in certain pairs there is no need for an attribute value; the first component is sufficient to identify the lexeme. In this small example, the token **num** has been given an integer-valued attribute. The compiler may store the character string that forms a number in a symbol table and let the attribute of token **num** be a pointer to the table entry. □

Lexical Errors

Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of a source program. If the string `fi` is encountered in a C program for the first time in the context

```
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 identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

But, suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input. Perhaps 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. This recovery technique may occasionally confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. deleting an extraneous character
2. inserting a missing character
3. replacing an incorrect character by a correct character
4. transposing two adjacent characters.

Error 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 just a single error transformation. This strategy assumes most lexical errors are the result of a single error transformation, an assumption usually, but not always, borne out in practice.

One way of finding the errors in a program is to compute the minimum number of error transformations required to transform the erroneous program into one that is syntactically well-formed. We say that the erroneous program has k errors if the shortest sequence of error transformations that will map it into some valid program has length k. Minimum-distance error correction is a convenient theoretical yardstick, but it is not generally used in practice because it is too costly to implement. However, a few experimental compilers have used the minimum-distance criterion to make local corrections.

3.2 INPUT BUFFERING

This section covers some efficiency issues concerned with the buffering of input. We first mention a two-buffer input scheme that is useful when look-ahead on the input is necessary to identify tokens. Then we introduce some useful techniques for speeding up the lexical analyzer, such as the use of "sentinels" to mark the buffer end.

There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator, such as the Lex compiler discussed in Section 3.5, to produce the lexical analyzer from a regular-expression-based specification. In this case, the generator provides routines for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of input.

The three choices are listed in order of increasing difficulty for the implementor. Unfortunately, the harder-to-implement approaches often yield faster lexical analyzers. Since the lexical analyzer is the only phase of the compiler that reads the source program character-by-character, it is possible to spend a considerable amount of time in the lexical analysis phase, even though the later phases are conceptually more complex. Thus, the speed of lexical analysis is a concern in compiler design. While the bulk of the chapter is devoted to the first approach, the design and use of an automatic generator, we also consider techniques that are helpful in manual design. Section 3.4 discusses transition diagrams, which are a useful concept for the organization of a hand-designed lexical analyzer.

Buffer Pairs

For many source languages, there are times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced. The lexical analyzers in Chapter 2 used a function `ungetc` to push lookahead characters back into the input stream. Because a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. Many buffering schemes can be used, but since the techniques are somewhat dependent on system parameters, we shall only outline the principles behind one class of schemes here.

We use a buffer divided into two N-character halves, as shown in Fig. 3.3. Typically, N is the number of characters on one disk block, e.g., 1024 or 4096.

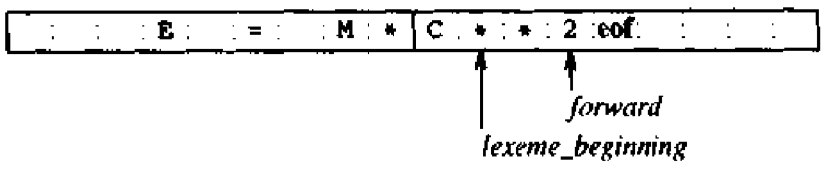


Fig. 3.3. An input buffer in two halves.

We read N input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character. If fewer than N characters remain in the input, then a special character **eof** is read into the buffer after the input characters, as in Fig. 3.3. That is, **eof** marks the end of the source file and is different from any input character.

Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme. Initially, both pointers point to the first character of the next lexeme to be found. One, called the forward pointer, scans ahead until a match for a pattern is found. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately past the lexeme. With this scheme, comments and white space can be treated as patterns that yield no token.

If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

This buffering scheme works quite well most of the time, but with it the amount of lookahead is limited, and this limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer. For example, if we see

```
DECLARE ( ARG1, ARG2, ... , ARGn )
```

in a PL/I program, we cannot determine whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the lexeme ends at the second E, but the amount of lookahead needed is proportional to the number of arguments, which in principle is unbounded.

```
if forward at end of first half then begin
    reload second half;
    forward := forward + 1
end
else if forward at end of second half then begin
    reload first half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

Fig. 3.4. Code to advance forward pointer.

Sentinels

If we use the scheme of Fig. 3.3 exactly as shown, we must check each time we move the forward pointer that we have not moved off one half of the buffer; if we do, then we must reload the other half. That is, our code for advancing the forward pointer performs tests like those shown in Fig. 3.4.

Except at the ends of the buffer halves, the code in Fig. 3.4 requires two tests for each advance of the forward pointer. We can reduce the two tests to one if we extend each buffer half to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program. A natural choice is **eof**; Fig. 3.5 shows the same buffer arrangement as Fig. 3.3, with the sentinels added.

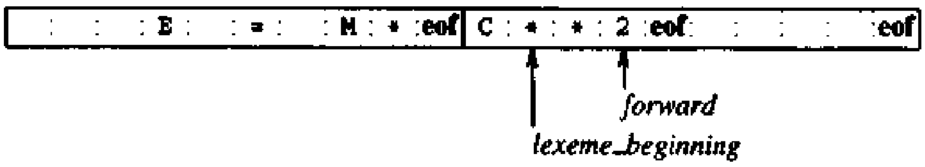


Fig. 3.5. Sentinels at end of each buffer half.

With the arrangement of Fig. 3.5, we can use the code shown in Fig. 3.6 to advance the forward pointer (and test for the end of the source file). Most of the time the code performs only one test to see whether *forward* points to an **eof**. Only when we reach the end of a buffer half or the end of the file do we perform more tests. Since N input characters are encountered between **eof**'s, the average number of tests per input character is very close to 1.

```
forward := forward + 1;
if forward↑ = eof then begin
      if forward at end of first half then begin
            reload second half;
            forward := forward + 1
      end
      else if forward at end of second half then begin
            reload first half;
            move forward to beginning of first half
      end
      else /* eof within a buffer signifying end of input */
            terminate lexical analysis
end
```

Fig. 3.6. Lookahead code with sentinels.

We also need to decide how to process the character scanned by the forward pointer; does it mark the end of a token, does it represent progress in finding a particular keyword, or what? One way to structure these tests is to use a case statement, if the implementation language has one. The test

```
if forward↑ = eof
```

can then be implemented as one of the different cases.

3.3 SPECIFICATION OF TOKENS

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings. Section 3.5 extends this notation into a pattern-directed language for lexical analysis.

Strings and Languages

The term *alphabet* or *character class* denotes any finite set of symbols. Typical examples of symbols are letters and characters. The set {0,1} is the *binary alphabet*. ASCII and EBCDIC are two examples of computer alphabets.

A *string* over some alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms *sentence* and *word* are often used as synonyms for the term "string." The length of a string s, usually written $|s|$, is the number of occurrences of symbols in s. For example, banana is a string of length six. The *empty* string, denoted ϵ, is a special string of length zero. Some common terms associated with parts of a string are summarized in Fig. 3.7.

The term *language* denotes any set of strings over some fixed alphabet. This definition is very broad. Abstract languages like $\emptyset$, the *empty* set, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed Pascal programs and the set of all grammatically correct English sentences, although the latter two sets are much more difficult to specify. Also note that this definition does not ascribe any meaning to the strings in a language. Methods for ascribing meanings to strings are discussed in Chapter 5.

If x and y are strings, then the *concatenation* of x and y, written 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 element under concatenation. That is, $s\epsilon = \epsilon s = s$.

If we think of concatenation as a "product", we can define string "exponentiation" as follows. Define s^0 to be ϵ, and for $i>0$ define s^i to be $s^{i-1}s$. Since ϵs is s itself, $s^1 = s$. Then, $s^2 = ss$, $s^3 = sss$, and so on.

TERM	DEFINITION
prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g., ban is a prefix of banana.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g., nana is a suffix of banana.
substring of s	A string obtained by deleting a prefix and a suffix from s; e.g., nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and ϵ are prefixes, suffixes, and substrings of s.
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g., baaa is a subsequence of banana.

Fig. 3.7. Terms for parts of a string.

Operations on Languages

There are several important operations that can be applied to languages. For lexical analysis, we are interested primarily in union, concatenation, and closure, which are defined in Fig. 3.8. We can also generalize the "exponentiation" operator to languages by defining L^0 to be $\{\epsilon\}$, and L^i to be $L^{i-1}L$. Thus, L^i is L concatenated with itself $i-1$ times.

Example 3.2. Let L be the set {A, B, . . . , Z, a, b, . . . , z} and D the set {0, 1, . . . , 9}. We can think of L and D in two ways. We can think of L as the alphabet consisting of the set of upper and lower case letters, and D as the alphabet consisting of the set of the ten decimal digits. Alternatively, since a symbol can be regarded as a string of length one, the sets L and D are each finite languages. Here are some examples of new languages created from L and D by applying the operators defined in Fig. 3.8.

1. $L \cup D$ is the set of letters and digits.
2. LD is the set of strings consisting of a letter followed by a digit.
3. L^4 is the set of all four-letter strings.
4. L^* is the set of all strings of letters, including ϵ, 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. □

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Fig. 3.8. Definitions of operations on languages.

Regular Expressions

In Pascal, an identifier is a letter followed by zero or more letters or digits; that is, an identifier is a member of the set defined in part (5) of Example 3.2. In this section, we present a notation, called regular expressions, that allows us to define precisely sets such as this. With this notation, we might define Pascal identifiers as

letter (**letter** | **digit**) *

The vertical bar here means "or," the parentheses are used to group subexpressions, the star means "zero or more instances of" the parenthesized expression, and the juxtaposition of **letter** with the remainder of the expression means concatenation.

A regular expression is built up out of simpler regular expressions using a set of defining rules. Each regular expression r denotes a language $L(r)$. The defining rules specify how $L(r)$ is formed by combining in various ways the languages denoted by the subexpressions of r.

Here are the rules that define the *regular expressions over alphabet* Σ. Associated with each rule is a specification of the language denoted by the regular expression being defined.

1. ϵ is a regular expression that denotes $\{\epsilon\}$, that is, the set containing the empty string.

2. If a is a symbol in Σ, then a is a regular expression that denotes $\{a\}$, i.e., the set containing the string a. Although we use the same notation for all three, technically, the regular expression a is different from the string a or the symbol a. It will be clear from the context whether we are talking about a as a regular expression, string, or symbol.

3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,

   a) $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$.
   b) $(r)(s)$ is a regular expression denoting $L(r)L(s)$.
   c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
   d) (r) is a regular expression denoting $L(r)$.[2]

A language denoted by a regular expression is said to be a *regular set*.

The specification of a regular expression is an example of a recursive definition. Rules (1) and (2) form the basis of the definition; we use the term *basic symbol* to refer to ϵ or a symbol in Σ appearing in a regular expression. Rule (3) provides the inductive step.

Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1. the unary operator * has the highest precedence and is left associative,
2. concatenation has the second highest precedence and is left associative,
3. | has the lowest precedence and is left associative.

Under these conventions, $(a)|((b)^*(c))$ is equivalent to $a|b^*c$. Both expressions denote the set of strings that are either a single a or zero or more b's followed by one c.

Example 3.3. Let $\Sigma = \{a, b\}$.

1. The regular expression $a|b$ denotes the set $\{a, b\}$.

2. The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$, the set of all strings of a's and b's of length two. Another regular expression for this same set is $aa \mid ab \mid ba \mid bb$.

3. The regular expression a^* denotes the set of all strings of zero or more a's, i.e., $\{\epsilon, a, aa, aaa, \cdots\}$.

4. The regular expression $(a|b)^*$ denotes the set of all strings containing zero or more instances of an a or b, that is, the set of all strings of a's and b's. Another regular expression for this set is $(a^*b^*)^*$.

5. The regular expression $a \mid a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's followed by a b. □

If two regular expressions r and s denote the same language, we say r and s are *equivalent* and write $r = s$. For example, $(a|b) = (b|a)$.

There are a number of algebraic laws obeyed by regular expressions and these can be used to manipulate regular expressions into equivalent forms. Figure 3.9 shows some algebraic laws that hold for regular expressions r, s, and t.

[2] This rule says that extra pairs of parentheses may be placed around regular expressions if we desire.

Axiom	Description
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenation distributes over $\mid$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r \mid \epsilon)^*$	relation between $*$ and ϵ
$r^{**} = r^*$	$*$ is idempotent

Fig. 3.9. Algebraic properties of regular expressions.

Regular Definitions

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, i.e., the basic symbols and the previously defined names. By restricting each r_i to symbols of Σ and the previously defined names, we can construct a regular expression over Σ for any r_i by repeatedly replacing regular-expression names by the expressions they denote. If r_i used d_j for some $j \geq i$, then r_i might be recursively defined, and this substitution process would not terminate.

To distinguish names from symbols, we print the names in regular definitions in boldface.

Example 3.4. As we have stated, the set of Pascal identifiers is the set of strings of letters and digits beginning with a letter. Here is a regular definition for this set.

letter → A | B | · · · | Z | a | b | · · · | z
digit → 0 | 1 | · · · | 9
id → **letter** (**letter** | **digit**)* □

Example 3.5. Unsigned numbers in Pascal are strings such as 5280, 39.37, `6.336E4`, or `1.894E-4`. The following regular definition provides a precise specification for this class of strings:

$$\begin{array}{rcl}
\textbf{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textbf{digits} & \rightarrow & \textbf{digit}\ \textbf{digit}^* \\
\textbf{optional\_fraction} & \rightarrow & \texttt{.}\ \textbf{digits} \mid \epsilon \\
\textbf{optional\_exponent} & \rightarrow & (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textbf{digits}\ ) \mid \epsilon \\
\textbf{num} & \rightarrow & \textbf{digits}\ \textbf{optional\_fraction}\ \textbf{optional\_exponent}
\end{array}$$

This definition says that an **optional_fraction** is either a decimal point followed by one or more digits, or it is missing (the empty string). An **optional_exponent**, if it is not missing, is an `E` followed by an optional `+` or `-` sign, followed by one or more digits. Note that at least one digit must follow the period, so **num** does not match `1.` but it does match `1.0`. □

Notational Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances.* The unary postfix operator $^+$ means "one or more instances of." If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$. Thus, the regular expression a^+ denotes the set of all strings of one or more a's. The operator $^+$ has the same precedence and associativity as the operator *. The two algebraic identities $r^* = r^+|\epsilon$ and $r^+ = rr^*$ relate the Kleene and positive closure operators.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one instance of." The notation $r?$ is a shorthand for $r|\epsilon$. If r is a regular expression, then $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$. For example, using the $^+$ and ? operators, we can rewrite the regular definition for **num** in Example 3.5 as

$$\begin{array}{rcl}
\textbf{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textbf{digits} & \rightarrow & \textbf{digit}^+ \\
\textbf{optional\_fraction} & \rightarrow & (\ \texttt{.}\ \textbf{digits}\ )? \\
\textbf{optional\_exponent} & \rightarrow & (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-}\ )?\ \textbf{digits}\ )? \\
\textbf{num} & \rightarrow & \textbf{digits}\ \textbf{optional\_fraction}\ \textbf{optional\_exponent}
\end{array}$$

3. *Character classes.* The notation `[abc]` where `a`, `b`, and `c` are alphabet symbols denotes the regular expression `a | b | c`. An abbreviated character class such as `[a-z]` denotes the regular expression `a | b | ··· | z`. Using character classes, we can describe identifiers as being strings generated by the regular expression

`[A-Za-z][A-Za-z0-9]*`

Nonregular Sets

Some languages cannot be described by any regular expression. To illustrate the limits of the descriptive power of regular expressions, here we give examples of programming language constructs that cannot be described by regular expressions. Proofs of these assertions can be found in the references.

Regular expressions cannot be used to describe balanced or nested constructs. For example, the set of all strings of balanced parentheses cannot be described by a regular expression. On the other hand, this set can be specified by a context-free grammar.

Repeating strings cannot be described by regular expressions. The set

$$\{wcw \mid w \text{ is a string of } a\text{'s and } b\text{'s}\}$$

cannot be denoted by any regular expression, nor can it be described by a context-free grammar.

Regular expressions can be used to denote only a fixed number of repetitions or an unspecified number of repetitions of a given construct. Two arbitrary numbers cannot be compared to see whether they are the same. Thus, we cannot describe Hollerith strings of the form $n\mathtt{H}a_1a_2 \cdots a_n$ from early versions of Fortran with a regular expression, because the number of characters following `H` must match the decimal number before `H`.

3.4 RECOGNITION OF TOKENS

In the previous section, we considered the problem of how to specify tokens. In this section, we address the question of how to recognize them. Throughout this section, we use the language generated by the following grammar as a running example.

Example 3.6. Consider the following grammar fragment:

$$\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term}\ \textbf{relop}\ \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{num}
\end{array}$$

where the terminals **if**, **then**, **else**, **relop**, **id**, and **num** generate sets of strings given by the following regular definitions:

$$\begin{array}{rcl}
\textbf{if} & \rightarrow & \mathtt{if} \\
\textbf{then} & \rightarrow & \mathtt{then} \\
\textbf{else} & \rightarrow & \mathtt{else} \\
\textbf{relop} & \rightarrow & \mathtt{<} \mid \mathtt{<=} \mid \mathtt{=} \mid \mathtt{<>} \mid \mathtt{>} \mid \mathtt{>=} \\
\textbf{id} & \rightarrow & \textbf{letter}\ (\ \textbf{letter} \mid \textbf{digit}\ )^* \\
\textbf{num} & \rightarrow & \textbf{digit}^+\ (\ .\ \textbf{digit}^+\ )?\ (\ \mathtt{E}(\ + \mid -\ )?\ \textbf{digit}^+\ )?
\end{array}$$

where **letter** and **digit** are as defined previously.

For this language fragment the lexical analyzer will recognize the keywords `if`, `then`, `else`, as well as the lexemes denoted by **relop**, **id**, and **num**. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers. As in Example 3.5, **num** represents the unsigned integer and real numbers of Pascal.

In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines. Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws**, below.

$$\begin{aligned} \textbf{delim} &\rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\ \textbf{ws} &\rightarrow \textbf{delim}^+ \end{aligned}$$

If a match for ws is found, the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space and returns that to the parser.

Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value, using the translation table given in Fig. 3.10. The attribute-values for the relational operators are given by the symbolic constants `LT`, `LE`, `EQ`, `NE`, `GT`, `GE`. □

Regular Expression	Token	Attribute-Value
ws	-	-
`if`	**if**	-
`then`	**then**	-
`else`	**else**	-
id	**id**	pointer to table entry
num	**num**	pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Fig. 3.10. Regular-expression patterns for tokens.

Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first produce a stylized flowchart, called a *transition diagram*. Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token, as suggested by Fig. 3.1. Suppose the input buffer is as in Fig. 3.3 and the lexeme-beginning pointer points to the character following the last lexeme found. We use a transition diagram to keep track of information about characters that are seen as the forward pointer scans the input. We do so by moving from position to position in the diagram as characters are read.

Positions in a transition diagram are drawn as circles and are called *states*. The states are connected by arrows, called *edges*. Edges leaving state *s* have labels indicating the input characters that can next appear after the transition diagram has reached state *s*. The label **other** refers to any character that is not indicated by any of the other edges leaving *s*.

We assume the transition diagrams of this section are *deterministic*; that is, no symbol can match the labels of two edges leaving one state. Starting in Section 3.5, we shall relax this condition, making life much simpler for the designer of the lexical analyzer and, with proper tools, no harder for the implementor.

One state is labeled the *start* state; it is the initial state of the transition diagram where control resides when we begin to recognize a token. Certain states may have actions that are executed when the flow of control reaches that state. On entering a state we read the next input character. If there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge. Otherwise, we indicate failure.

Figure 3.11 shows a transition diagram for the patterns >= and >. The transition diagram works as follows. Its start state is state 0. In state 0, we read the next input character. The edge labeled > from state 0 is to be followed to state 6 if this input character is >. Otherwise, we have failed to recognize either > or >=.

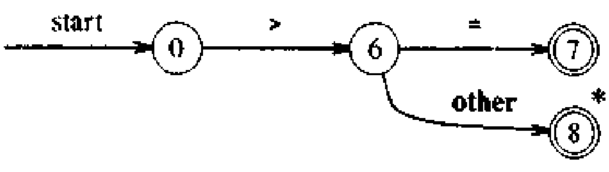


Fig. 3.11. Transition diagram for >=.

On reaching state 6 we read the next input character. The edge labeled = from state 6 is to be followed to state 7 if this input character is an =. Otherwise, the edge labeled **other** indicates that we are to go to state 8. The double circle on state 7 indicates that it is an accepting state, a state in which the token >= has been found.

Notice that the character > and another extra character are read as we follow the sequence of edges from the start state to the accepting state 8. Since the extra character is not a part of the relational operator >, we must retract the forward pointer one character. We use a * to indicate states on which this input retraction must take place.

In general, there may be several transition diagrams, each specifying a group of tokens. If failure occurs while we are following one transition diagram, then we retract the forward pointer to where it was in the start state of this diagram, and activate the next transition diagram. Since the lexeme-beginning and forward pointers marked the same position in the start state of the diagram, the forward pointer is retracted to the position marked by the lexeme-beginning pointer. If failure occurs in all transition diagrams, then a lexical error has been detected and we invoke an error-recovery routine.

Example 3.7. A transition diagram for the token **relop** is shown in Fig. 3.12. Notice that Fig. 3.11 is a part of this more complex transition diagram. □

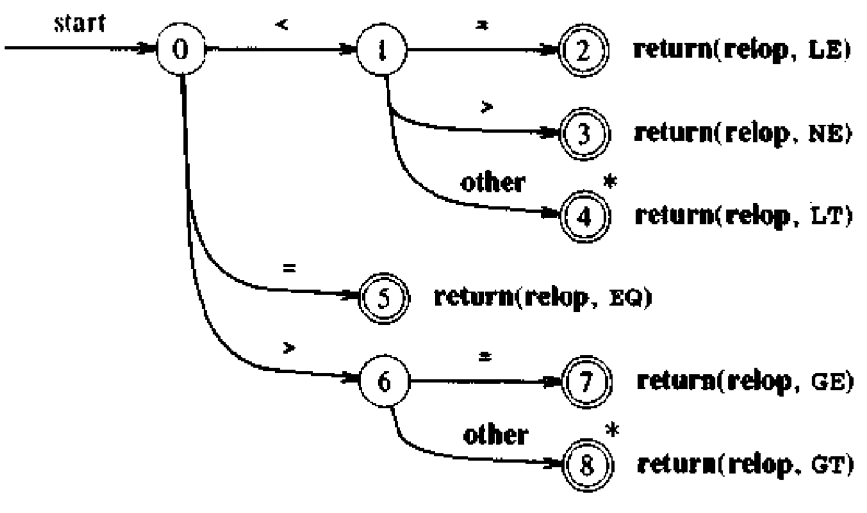


Fig. 3.12. Transition diagram for relational operators.

Example 3.8. Since keywords are sequences of letters, they are exceptions to the rule that a sequence of letters and digits starting with a letter is an identifier. Rather than encode the exceptions into a transition diagram, a useful trick is to treat keywords as special identifiers, as in Section 2.7. When the accepting state in Fig. 3.13 is reached, we execute some code to determine if the lexeme leading to the accepting state is a keyword or an identifier.

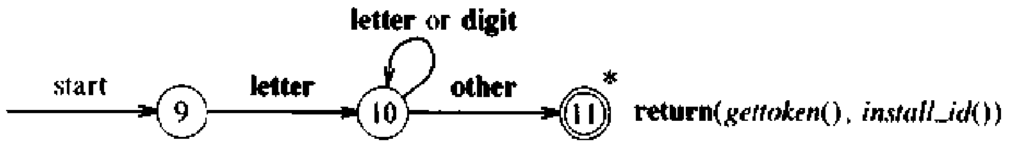


Fig. 3.13. Transition diagram for identifiers and keywords.

A simple technique for separating keywords from identifiers is to initialize appropriately the symbol table in which information about identifiers is saved. For the tokens of Fig. 3.10 we need to enter the strings `if`, `then`, and `else` into the symbol table before any characters in the input are seen. We also make a note in the symbol table of the token to be returned when one of these strings is recognized. The return statement next to the accepting state in Fig. 3.13 uses *gettoken*() and *install_id*() to obtain the token and attribute-value, respectively, to be returned. The procedure *install_id*() has access to the buffer, where the identifier lexeme has been located. The symbol table is examined and if the lexeme is found there marked as a keyword, *install_id*() returns 0. If the lexeme is found and is a program variable, *install_id*() returns a pointer to the symbol table entry. If the lexeme is not found in the symbol table, it is installed as a variable and a pointer to the newly created entry is returned.

The procedure *gettoken*() similarly looks for the lexeme in the symbol table. If the lexeme is a keyword, the corresponding token is returned; otherwise, the token **id** is returned.

Note that the transition diagram does not change if additional keywords are to be recognized; we simply initialize the symbol table with the strings and tokens of the additional keywords. □

The technique of placing keywords in the symbol table is almost essential if the lexical analyzer is coded by hand. Without doing so, the number of states in a lexical analyzer for a typical programming language is several hundred, while using the trick, fewer than a hundred states will probably suffice.

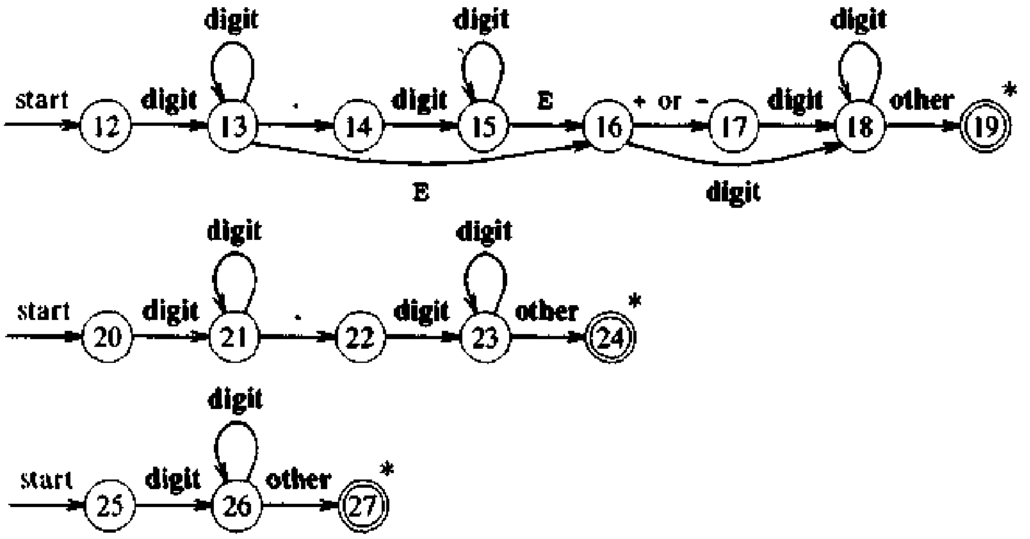


Fig. 3.14. Transition diagrams for unsigned numbers in Pascal.

Example 3.9. A number of issues arise when we construct a recognizer for unsigned numbers given by the regular definition

num → **digit**$^+$ (.**digit**$^+$)? (E(+|-)? **digit**$^+$)?

Note that the definition is of the form **digits fraction**? **exponent**? in which **fraction** and **exponent** are optional.

The lexeme for a given token must be the longest possible. For example, the lexical analyzer must not stop after seeing 12 or even 12.3 when the input is 12.3E4. Starting at states 25, 20, and 12 in Fig. 3.14, accepting states will be reached after 12, 12.3, and 12.3E4 are seen, respectively, provided 12.3E4 is followed by a non-digit in the input. The transition diagrams with start states 25, 20, and 12 are for **digits**, **digits fraction**, and **digits fraction**? **exponent**, respectively, so the start states must be tried in the reverse order 12, 20, 25.

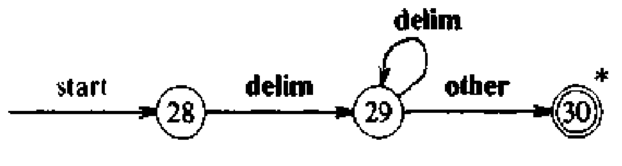
The action when any of the accepting states 19, 24, or 27 is reached is to call a procedure *install_num* that enters the lexeme into a table of numbers and returns a pointer to the created entry. The lexical analyzer returns the token **num** with this pointer as the lexical value. □

Information about the language that is not in the regular definitions of the tokens can be used to pinpoint errors in the input. For example, on input 1.<x, we fail in states 14 and 22 in Fig. 3.14 with next input character <. Rather than returning the number 1, we may wish to report an error and continue as if the input were 1.0<x. Such knowledge can also be used to simplify the transition diagrams, because error-handling may be used to recover from some situations that would otherwise lead to failure.

There are several ways in which the redundant matching in the transition diagrams of Fig. 3.14 can be avoided. One approach is to rewrite the transition diagrams by combining them into one, a nontrivial task in general. Another is to change the response to failure during the process of following a diagram. An approach explored later in this chapter allows us to pass through several accepting states; we revert back to the last accepting state that we passed through when failure occurs.

Example 3.10. A sequence of transition diagrams for all tokens of Example 3.6 is obtained if we put together the transition diagrams of Fig. 3.12, 3.13, and 3.14. Lower-numbered start states are to be attempted before higher numbered states.

The only remaining issue concerns white space. The treatment of **ws**, representing white space, is different from that of the patterns discussed above because nothing is returned to the parser when white space is found in the input. A transition diagram recognizing **ws** by itself is



Nothing is returned when the accepting state is reached; we merely go back to the start state of the first transition diagram to look for another pattern.

Whenever possible, it is better to look for frequently occurring tokens before less frequently occurring ones, because a transition diagram is reached only after we fail on all earlier diagrams. Since white space is expected to occur frequently, putting the transition diagram for white space near the beginning should be an improvement over testing for white space at the end. □

Implementing a Transition Diagram

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagrams. We adopt a systematic approach that works for all transition diagrams and constructs programs whose size is proportional to the number of states and edges in the diagrams.

Each state gets a segment of code. If there are edges leaving a state, then its code reads a character and selects an edge to follow, if possible. A function `nextchar()` is used to read the next character from the input buffer, advance the forward pointer at each call, and return the character read.[3] If there is an edge labeled by the character read, or labeled by a character class containing the character read, then control is transferred to the code for the state pointed to by that edge. If there is no such edge, and the current state is not one that indicates a token has been found, then a routine `fail()` is invoked to retract the forward pointer to the position of the beginning pointer and to initiate a search for a token specified by the next transition diagram. If there are no other transition diagrams to try, `fail()` calls an error-recovery routine.

To return tokens we use a global variable `lexical_value`, which is assigned the pointers returned by functions `install_id()` and `install_num()` when an identifier or number, respectively, is found. The token class is returned by the main procedure of the lexical analyzer, called `nexttoken()`.

We use a case statement to find the start state of the next transition diagram. In the C implementation in Fig. 3.15, two variables `state` and `start` keep track of the present state and the starting state of the current transition diagram. The state numbers in the code are for the transition diagrams of Figures 3.12 – 3.14.

Edges in transition diagrams are traced by repeatedly selecting the code fragment for a state and executing the code fragment to determine the next state as shown in Fig. 3.16. We show the code for state 0, as modified in Example 3.10 to handle white space, and the code for two of the transition diagrams from Fig. 3.13 and 3.14. Note that the C construct

`while( 1 )` *stmt*

repeats *stmt* "forever," i.e., until a `return` occurs.

[3] A more efficient implementation would use an in-line macro in place of the function `nextchar()`.

```
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

int fail()
{
    forward = token_beginning;
    switch (start) {
       case 0:   start = 9; break;
       case 9:   start = 12; break;
       case 12:  start = 20; break;
       case 20:  start = 25; break;
       case 25:  recover(); break;
       default:   /* compiler error */
    }
    return start;
}
```

Fig. 3.15. C code to find next start state.

Since C does not allow both a token and an attribute-value to be returned, `install_id()` and `install_num()` appropriately set some global variable to the attribute-value corresponding to the table entry for the **id** or **num** in question.

If the implementation language does not have a case statement, we can create an array for each state, indexed by characters. If *state* 1 is such an array, then *state* 1[*c*] is a pointer to a piece of code that must be executed whenever the lookahead character is *c*. This code would normally end with a goto to code for the next state. The array for state *s* is referred to as the indirect transfer table for *s*.

3.5 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS

Several tools have been built for constructing lexical analyzers from special-purpose notations based on regular expressions. We have already seen the use of regular expressions for specifying token patterns. Before we consider algorithms for compiling regular expressions into pattern-matching programs, we give an example of a tool that might use such an algorithm.

In this section, we describe a particular tool, called Lex, that has been widely used to specify lexical analyzers for a variety of languages. We refer to the tool as the *Lex compiler*, and to its input specification as the *Lex language*. Discussion of an existing tool will allow us to show how the specification of patterns using regular expressions can be combined with actions, e.g., making entries into a symbol table, that a lexical analyzer may be required to perform. Lex-like specifications can be used even if a Lex

```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c is lookahead character */
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
                    /* advance beginning of lexeme */
            }
            else if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else state = fail();
            break;

            .../* cases 1-8 here */

        case 9:   c = nextchar();
            if (isletter(c)) state = 10;
            else state = fail();
            break;
        case 10:  c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        case 11:  retract(1); install_id();
            return ( gettoken() );

            .../* cases 12-24 here */

        case 25:  c = nextchar();
            if  (isdigit(c)) state = 26;
            else state = fail();
            break;
        case 26:  c = nextchar();
            if (isdigit(c)) state = 26;
            else state = 27;
            break;
        case 27:  retract(1); install_num();
            return ( NUM );
        }
    }
}
```

Fig. 3.16. C code for lexical analyzer.

compiler is not available; the specifications can be manually transcribed into a working program using the transition diagram techniques of the previous section.

Lex is generally used in the manner depicted in Fig. 3.17. First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`. The program `lex.yy.c` consists of a tabular representation of a transition diagram constructed from the regular expressions of `lex.l`, together with a standard routine that uses the table to recognize lexemes. The actions associated with regular expressions in `lex.l` are pieces of C code and are carried over directly to `lex.yy.c`. Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

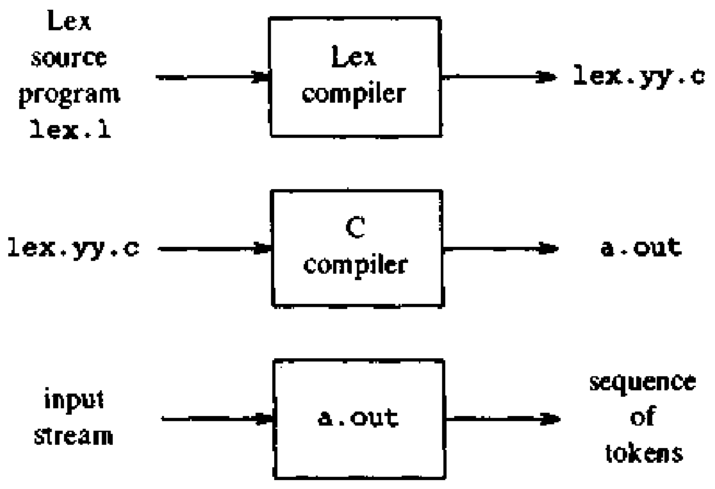


Fig. 3.17. Creating a lexical analyzer with Lex.

Lex Specifications

A Lex program consists of three parts:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

The declarations section includes declarations of variables, manifest constants, and regular definitions. (A manifest constant is an identifier that is declared to represent a constant.) The regular definitions are statements similar to those given in Section 3.3 and are used as components of the regular expressions appearing in the translation rules.

The translation rules of a Lex program are statements of the form

$$\begin{array}{ll} p_1 & \{ \mathit{action}_1 \} \\ p_2 & \{ \mathit{action}_2 \} \\ \cdots & \cdots \\ p_n & \{ \mathit{action}_n \} \end{array}$$

where each p_i is a regular expression and each *action*$_i$ is a program fragment describing what action the lexical analyzer should take when pattern p_i matches a lexeme. In Lex, the actions are written in C; in general, however, they can be in any implementation language.

The third section holds whatever auxiliary procedures are needed by the actions. Alternatively, these procedures can be compiled separately and loaded with the lexical analyzer.

A lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p_i. Then, it executes *action*$_i$. Typically, *action*$_i$ will return control to the parser. However, if it does not, then the lexical analyzer proceeds to find more lexemes, until an action causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.

The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set a global variable called `yylval`.

Example 3.11. Figure 3.18 is a Lex program that recognizes the tokens of Fig. 3.10 and returns the token found. A few observations about the code will introduce us to many of the important features of Lex.

In the declarations section, we see (a place for) the declaration of certain manifest constants used by the translation rules.[4] These declarations are surrounded by the special brackets `%{` and `%}`. Anything appearing between these brackets is copied directly into the lexical analyzer `lex.yy.c`, and is not treated as part of the regular definitions or the translation rules. Exactly the same treatment is accorded the auxiliary procedures in the third section. In Fig. 3.18, there are two procedures, `install_id` and `install_num`, that are used by the translation rules; these procedures will be copied into `lex.yy.c` verbatim.

Also included in the definitions section are some regular definitions. Each such definition consists of a name and a regular expression denoted by that name. For example, the first name defined is `delim`; it stands for the

[4] It is common for the program `lex.yy.c` to be used as a subroutine of a parser generated by Yacc, a parser generator to be discussed in Chapter 4. In this case, the declaration of the manifest constants would be provided by the parser, when it is compiled with the program `lex.yy.c`.

```
%{
    /* 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 = install_id(); return(ID);}
{number}    {yylval = install_num(); 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);}

%%

install_id() {
    /* procedure to install the lexeme, whose
    first character is pointed to by yytext and
    whose length is yyleng, into the symbol table
    and return a pointer thereto */
}

install_num() {
    /* similar procedure to install a lexeme that
    is a number */
}
```

Fig. 3.18. Lex program for the tokens of Fig. 3.10.