

CHAPTER TWO

2. LEXICAL ANALYSIS

The lexical analysis is an essential step for language understanding, as well as for the compilation because it is also taken as the basis of understanding programs. The main purpose of lexical analysis is to make life **easier** for the subsequent syntax analysis phase. Lexical analysis is the **first phase of a compiler**. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer **finds a token invalid, it generates an error**. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

2.1. The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to **read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens** for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

The interactions are suggested in Figure 2.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

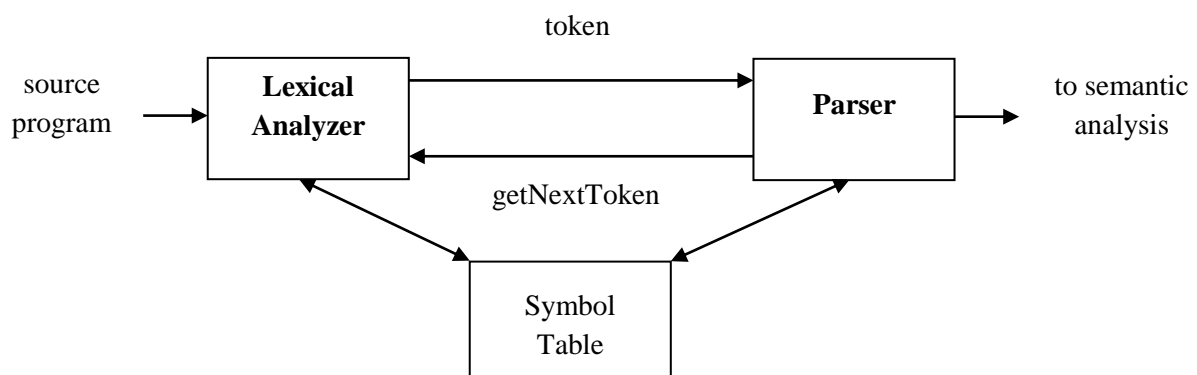


Figure 2.1: Interactions between the lexical analyzer and the parser

The following are **additional tasks** performed by the lexical analyzer other than identifying lexemes:

- **Stripping out comments and whitespace** (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- **Correlating error messages** generated by 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.

Sometimes, lexical analyzers are divided into a cascade of two processes:

- Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one. The scanner is in charge of the simple task.
- Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output. The lexical analyzer is really doing the more complex operations.

2.1.1. Issues in Lexical Analysis

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

- ✓ Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
- ✓ 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.
- ✓ Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

2.1.2. Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a keyword, constants, strings, numbers, operators, punctuation symbols or a sequence of input characters denoting an identifier. The token names are input symbols that the parser processes.
- A **pattern** is a description of the form that lexemes of a token may take. 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. In case of a keyword as a token, the pattern is the sequence of characters that forms the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.
- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

The following table gives some typical tokens, their informally described patterns, and some sample lexemes.

TOKEN	SAMPLE LEXEME	INFORMAL DESCRIPTION
if	if	characters i, f
const	const	characters c,o,n,s,t
comparison	<=, !=	< or > or <= or >= or == or !=
id	pi, score, D2	letter followed by letters and digit
num	3.14159, 0, 6.02e23	any numeric constant
literal	"core dumped"	anything but ", surrounded by " 's

- In many programming languages, the following classes cover most or all of the tokens:
 - ✓ One token for each keyword. The pattern for a keyword is the same as the keyword itself.
 - ✓ Tokens for operators, either individually or in classes such as comparison as mentioned above.
 - ✓ One token representing all identifiers
 - ✓ One or more tokens representing constants, such as numbers and literal strings
 - ✓ Tokens for each punctuation symbol, such as left and right parentheses, comma and semicolon

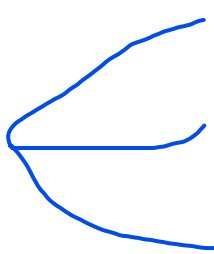
2.1.3. Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phase's additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

This is done by using attribute value that describes the lexeme represented by the token. The lexical analyzer collects information about tokens into their associated attributes. The token name influences parsing decisions; while 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. The symbol table entry contains various information about the token such as the lexeme, its type, the line number in which it was first seen, etc.

- For example, in assignment statement (in FORTRAN): $E = M * C ** 2$, the tokens and their attributes 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>
- <number, integer value 2>

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute.

2.1.4. Lexical Errors

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognize a lexeme as a valid token for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of already recognized valid tokens don't match any of the right sides of your grammar rules.

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For example, if the string **fi** is encountered for the first time in a C program in the context:

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

a lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or undeclared function identifier since **fi** is a valid lexeme for the token identifier.

When an error occurs, the lexical analyzer recovers by:

- ✓ Skipping (deleting) successive characters from the remaining input until the lexical analyzer can find a well-formed token (known as panic mode recovery)
- ✓ Deleting extraneous characters from the remaining input
- ✓ Inserting missing characters from the remaining input
- ✓ Replacing an incorrect character by a correct character
- ✓ Transposing two adjacent characters

2.2. Input Buffering

Here we will be looking at some ways that the 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. 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**. 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 look a heads safely. We then consider an improvement involving sentinels that saves time checking for the ends of buffers.

2.2.1. Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead to process a single input character. An important scheme involves two buffers that are alternatively reloaded, as suggested in Figure 2.2

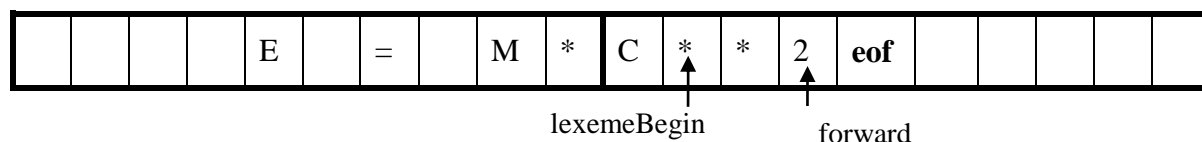


Figure 2.2: Using a pair of input buffers

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**, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer **lexemeBegin**, marks the **beginning of the current lexeme**, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead **until a pattern match is found**.

Once the lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is **recorded as an attribute value of the token returned to the parser**, **lexemeBegin** is set to the character immediately after the lexeme just found.

Advancing **forward** requires that we first **test** whether we have reached the **end of one of the buffers**, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

2.2.2. Sentinels

If we use the previous scheme, we must check each time we advance **forward**, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we must make **two tests: one for the end of the buffer**, and **one to determine which character is read**. We can **combine** the buffer-end test with the test for the current character if we extend each buffer to hold *sentinel* character at the end.

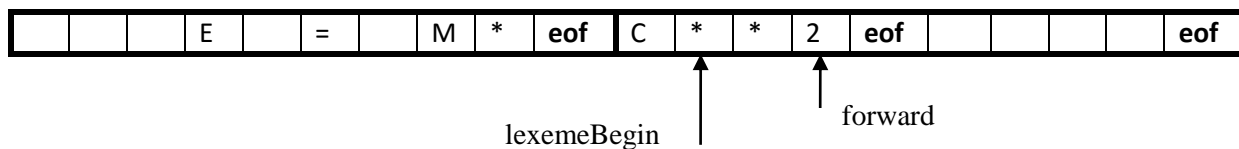


Figure 2.3: Sentinels at the end of each buffer

The *sentinel* is a special character that **cannot be part of the source program**, and a natural choice is the character **eof**.

Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of buffer means that the input is at an end. Figure 2.3 shows the same arrangement as Figure 2.2, but with the *sentinels* added.

2.3. Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While **they cannot express all possible patterns**, they are effective in specifying those types of patterns that we need for tokens.

- Refer to the previous course (Formal Language Theory) to recap on regular expressions

2.3.1. Strings and Languages

An *alphabet* (Σ) is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set $\{0, 1\}$ is the binary alphabet. ASCII is an important example of an alphabet.

A *string* over an 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 "string." The length of a string s , usually written as $|s|$, is the number of occurrences of symbols in s . For example, *banana* is a string of length six. The empty string, denoted by ϵ , is the string of length zero.

A language is any countable set of strings over some fixed alphabet. For example, the set of legal English words, the set of strings consisting of n 0's followed by n 1's $\{\epsilon, 01, 0011, 000111, \dots\}$, the set of binary numbers whose value is prime $\{10, 11, 101, 111, 1011, \dots\}$ are languages.

2.3.2. Operations on Languages

In **lexical analysis**, the most important **operations** on languages are **union, concatenation, and closure**.

- ❖ 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 $\{\epsilon\}$, 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, ϵ will not be in L^+ unless it is in L itself.

- ✓ Union of L and M : $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
- ✓ Concatenation of L and M : $LM = \{xy \mid x \in L \text{ and } y \in M\}$
- ✓ Exponentiation of L : $L^0 = \{\epsilon\}$; $L^i = L^{i-1}L$
- ✓ Kleene closure of L : $L^* = \cup_{i=0, \dots, \infty} L^i$
- ✓ Positive closure of L : $L^+ = \cup_{i=1, \dots, \infty} L^i$

2.3.3. Regular expressions

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 Σ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
2. If a is a symbol in Σ , 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.

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. $(r) \mid (s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
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.

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, $(a \mid b) = (b \mid a)$

2.3.4. 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 Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

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

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d 's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

By restricting r_i to Σ and the previously defined d 's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i . We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d 's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2 , and so on. Finally, in r_n we replace each d_i , for $i = 1, 2, \dots, n-1$, by the substituted version of r_i , each of which has only symbols of Σ .

Example:

- C identifiers are strings of letters, digits, and underscores. Here is a regular definition:

$$\begin{aligned} \text{letter_} &\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _ \\ \text{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\ \text{id} &\rightarrow \text{letter_}(\text{letter_} \mid \text{digit})^* \end{aligned}$$

2.3.5. Extensions of Regular Expressions

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 specification of lexical analyzers. Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them. Some regular expression variants that use today are the following.

1. **One or more instances.** The unary postfix operator **+** represents the **positive closure** of a regular expression and its language. The operator **+** has the same precedence and associativity as the operator *****. $r^* = r^+ \mid \epsilon$ and $r^+ = rr^* = r^*r$
2. **Zero or one instance.** The unary postfix operator **?**, means “zero or one occurrence”. That is $r?$ is equivalent to $r \mid \epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The **?** operator has **the same precedence** and associativity as ***** and **+**.
3. **Character classes.** A regular expression $a_1 \mid a_2 \mid \dots \mid a_n$, where a_i 's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2 \dots a_n]$. More importantly, when a_1, a_2, \dots, a_n form a logical sequence, e.g., consecutive uppercase letter, lowercase letters, or digits, we can replace them by $a_1 - a_n$, that is, just the first and the last separated by hyphen. Thus $[abc]$ is shorthand for $a \mid b \mid c$, and $[a-z]$ is shorthand for $a \mid b \mid \dots \mid z$

Using these short hands, we can rewrite the above regular definitions as follows.

- ✓ $\text{letter_} \rightarrow [A - Za - z _]$
- ✓ $\text{digit} \rightarrow [0 - 9]$
- ✓ $\text{id} \rightarrow \text{letter_} (\text{letter_} \mid \text{digit})^*$

- The following is a regular definition for unsigned number in Pascal (note that it uses extensions regular expressions)

- ✓ $\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$
- ✓ $\text{digits} \rightarrow \text{digit}^+$
- ✓ $\text{num} \rightarrow \text{digits} (\text{digits})? (E[+-]? \text{digits})?$

2.4. Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and a prefix that is a lexeme matching one of the patterns.

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.

Consider the following grammar fragment:

$$\begin{aligned} \text{stmt} &\rightarrow \text{if expr then stmt} \\ &\quad \mid \text{if expr then stmt else stmt} \\ &\quad \mid \epsilon \\ \text{expr} &\rightarrow \text{term relop term} \\ &\quad \mid \text{term} \end{aligned}$$

$$\begin{array}{l} \text{term} \rightarrow \text{id} \\ \quad | \text{ num} \end{array}$$

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

$$\text{if} \rightarrow \text{if}$$

$$\text{then} \rightarrow \text{then}$$

$$\text{else} \rightarrow \text{else}$$

$$\text{relop} \rightarrow < \mid <= \mid < > \mid > \mid >= \mid =$$

$$\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$$

$$\text{num} \rightarrow \text{digit}^+ (. \text{digit}^+)? (\text{E} (+ \mid -)? \text{digit}^+)?$$

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.

In addition, we assume lexemes are separated by white space, consisting of non-null 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.

$$\text{delim} \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})$$

$$\text{ws} \rightarrow \text{delim}^+$$

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 the following. 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	GT
<=	relop	LE
=	relop	EQ

< >	relop	NE
>=	relop	GE

Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called “transition diagrams”. Transition diagrams have a collection of nodes 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.

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or a set of symbols. If we are in some state *s*, 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 and enter the state of the transition diagram to which that edge leads. Here we assume that our transition diagram is **deterministic**, meaning that there is never more than one edge out a given state with a given symbol among its labels. Some important conventions about the transition diagram 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**. Accepting states are indicated 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* one 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**. Any number of *s can be attached depending on the number of positions to retract
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

Figure 2.4 is a transition diagram that recognizes the lexemes matching the token **relop**. Being on state 0 (initial state), if we see < as the first input symbol, then 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 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 <, and we enter state 4 to return information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

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 the 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 $>=$ (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 cannot possibly be seeing a **relop** lexeme, so this transition diagram will not be used.

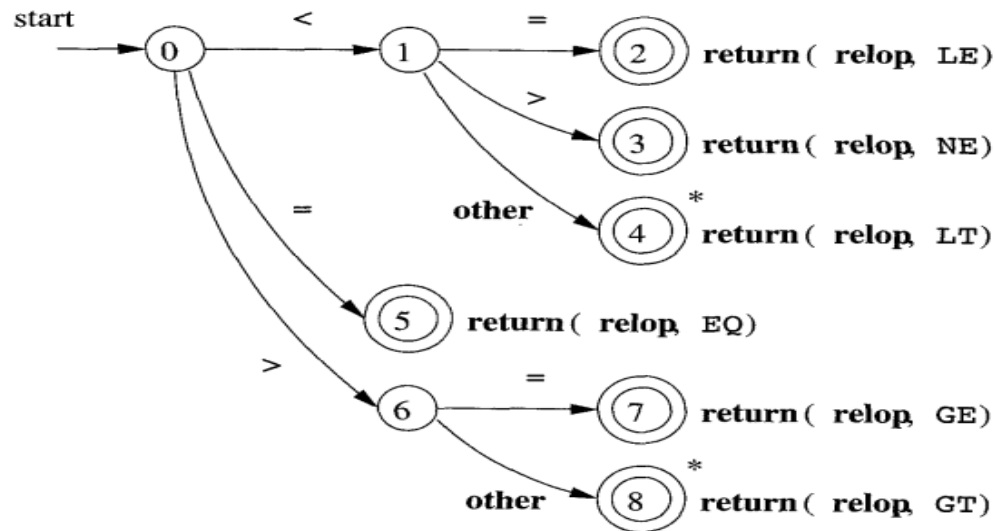


Figure 2.4 Transition diagram for **relop**