2

Lexical Analysis

1. What is the role of a lexical analyzer?

Ans: The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis. If lexical analyzer is placed as a separate pass in the compiler, it would require an intermediate file to place its output, from which the parser would then take its input. To eliminate the need for the intermediate file, the lexical analyzer and the syntactic analyzer (parser) are often grouped together into the same pass where the lexical analyzer operates either under the control of the parser or as a subroutine with the parser.

The parser requests the lexical analyzer for the next token, whenever it needs one. The lexical analyzer also interacts with the symbol table while passing tokens to the parser. Whenever a token is found, the lexical analyzer returns a representation for that token to the parser. If the token is a simple construct such as parentheses, comma, or a colon, then it returns an integer code. If the token is a more complex element such as an identifier or another token with a value, the value is also passed to the parser. The lexical analyzer provides this information by calling a bookkeeping routine which installs the actual value in the symbol table if it is not already there.

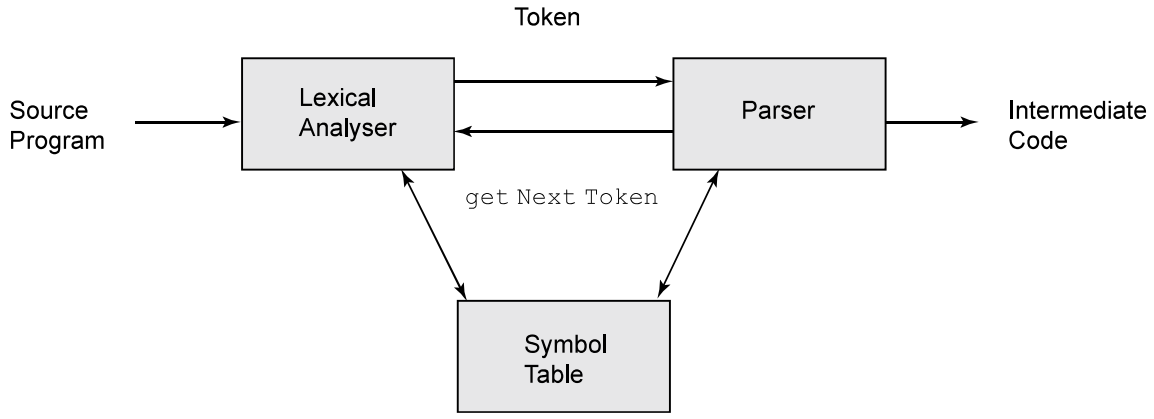

Figure 2.1 Role of the Lexical Analyzer

Besides generation of tokens, the lexical analyzer also performs certain other tasks such as:

- Stripping out comments and whitespace (tab, newline, blank, and other characters that are used to separate tokens in the input).
- Correlating error messages that are generated by the compiler during lexical analysis with the source program. For example, it can keep track of all newline characters so that it can associate an ambiguous statement line number with each error message.
- Performing the expansion of macros, in case macro preprocessors are used in the source program.

2. What do you understand by the terms tokens, patterns, and lexemes?

Ans: Tokens: The lexical analyzer separates the characters of the source language into groups that logically belong together, commonly known as **tokens**. A token consists of a token name and an optional attribute value. The **token name** is an abstract symbol that represents a kind of lexical unit and the optional attribute value is commonly referred to as **token value**. Each token represents a sequence of characters that can be treated as a single entity. Tokens can be identifiers, keywords, constants, operators, and punctuation symbols such as commas and parenthesis. In general, the tokens are broadly classified into two types:

- Specific strings such as if, else, comma, or a semicolon.
- Classes of strings such as identifiers, constants, or labels.

For example, consider an assignment statement in C

```
total = number1 + number2 * 5
```

After lexical analysis, the tokens generated are as follows:

```
<id,1> <=> <id,2> <+> <id,3> <*> <5>
```

Patterns: A rule that defines a set of input strings for which the same token is produced as output is known as **pattern**. Regular expressions play an important role for specifying patterns. If a keyword is considered as a token, the pattern is just the sequence of characters. But for identifiers and some other tokens, the pattern forms a complex structure.

Lexemes: A lexeme is a group of logically related characters in the source program that matches the pattern for a token. It is identified as an instance of that token by the lexical analyzer. For example, consider a C statement:

```
printf("Total = %d\n", total);
```

Here, `printf` is a keyword; parentheses, semicolon, and comma are punctuation symbols; `total` is a lexeme matching the pattern for token **id**; and `"Total = %d\n"` is a lexeme matching the pattern for token **literal**.

Some examples of tokens, patterns, and lexemes are given in Table 2.1.

Table 2.1 Examples of Tokens, Patterns and Lexemes

Token	Informal Description	Sample Lexeme
`while`	Characters `w, h, i, l, e`	`while`
`then`	Characters `t, h, e, n`	`then`
`comparison`	< or > or <= or >= or == or !=	<=, !=
`id`	Letter followed by letters and digits	`total, number1`
`number`	Any numeric constant	50, 3.12134, 0, 4.02e45
`literal`	Anything within double quotes(" ") except "	"Total"

3. What is the role of input buffering scheme in lexical analyzer?

Ans: The lexical analyzer scans the characters of source program one by one to find the tokens. Moreover, it needs to look ahead several characters beyond the next token to determine the next token itself. So, an input buffer is needed by the lexical analyzer to read its input. In a case of large source program, significant amount of time is required to process the characters during the compilation. To reduce the amount of overhead needed to process a single character from input character stream, specialized buffering techniques have been developed. An important technique that uses two input buffers that are reloaded alternately is shown in Figure 2.2.

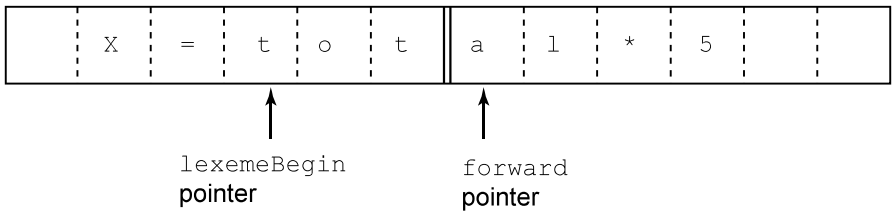

Figure 2.2 Input Buffer

Each buffer is of the same size `N`, where `N` is the size of a disk block, for example 1024 bytes. Thus, instead of one character, `N` characters can be read at a time. The pointers used in the input buffer for recognizing the lexeme are as follows:

- Pointer **`lexemeBegin`** points the beginning of the current lexeme being discovered.
- Pointer **`forward`** scans ahead until a pattern match is found for lexeme.

Initially, both pointers point to the first character of the next lexeme to be found. The `forward` pointer is scanned ahead until a match for a pattern is found. After the lexeme is processed, both the pointers are set to the character following that processed lexeme. For example, in Figure 2.2 the `lexemeBegin` pointer is at character `t` and `forward` pointer is at character `a`. The `forward` pointer is scanned until the lexeme `total` is found. Once it is found, both these pointers point to `*`, which is the next lexeme to be discovered.

4. What are strings and languages in lexical analysis? What are the operations performed on the languages?

Ans: Before defining the terms strings and languages, it is necessary to understand the term alphabet. An **alphabet** (or **character class**) denotes any finite set of symbols. Symbols include letters, digits, punctuation, etc. The ASCII, Unicode, and EBCDIC are the most important examples of alphabet. The set {0, 1} is the binary alphabet.

A **string** (also termed as **sentence** or **word**) is defined as a finite sequence of symbols drawn from an alphabet. The length of a string `s` is measured as the number of occurrences of symbols in `s` and is denoted by |`s`|. For example, the word 'orange' is a string of length six. The *empty string* (`∈`) is the string of length zero.

A **language** is any finite set of strings over some specific alphabet. This is an enormously broad definition. Simple sets such as ϕ, the empty set, or {`∈`}, the set containing only the empty string, are also the languages under this definition.

In lexical analysis, there are several important operations like union, concatenation, and closure that can be applied to languages. **Union** operation means taking all the strings of both the set of languages and creating a new set of language containing all the strings. The **concatenation** of languages is done