

LEXICAL ANALYSIS

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 tokens for each lexeme in the source program. This 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. This process is shown in the following figure.

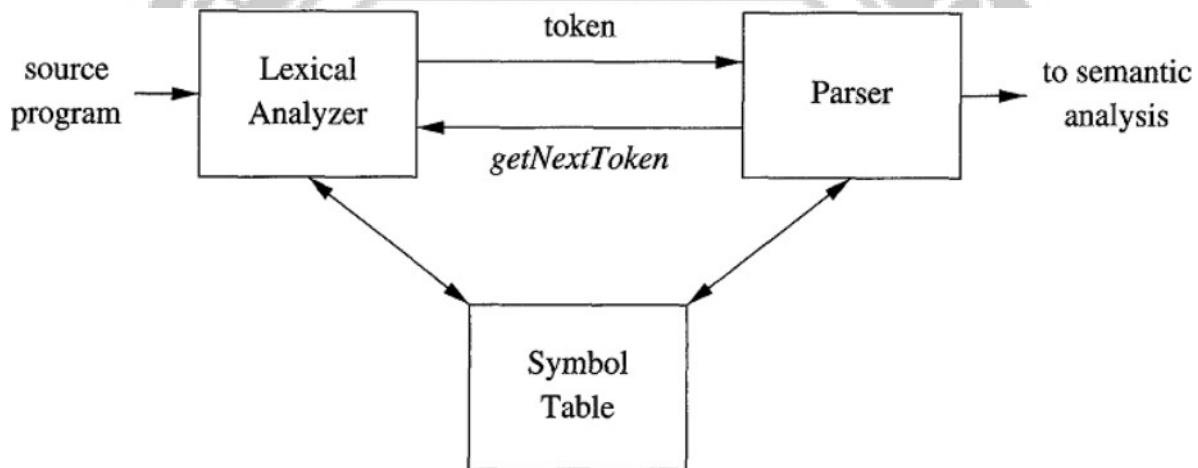


Figure : Lexical Analyzer

When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

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 particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form 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.

Example: In the following C language statement , printf

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

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n"** is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but " , surrounded by " 's	"core dumped"

Figure 1.7: Examples of Tokens

LEXICAL ANALYSIS Vs PARSING:

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

- 1. 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. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
- 2. Compiler efficiency is improved.** A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
- 3. Compiler portability is enhanced:** Input-device-specific peculiarities can be restricted to the lexical analyzer.

INPUT BUFFERING:

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

Buffer Pairs

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

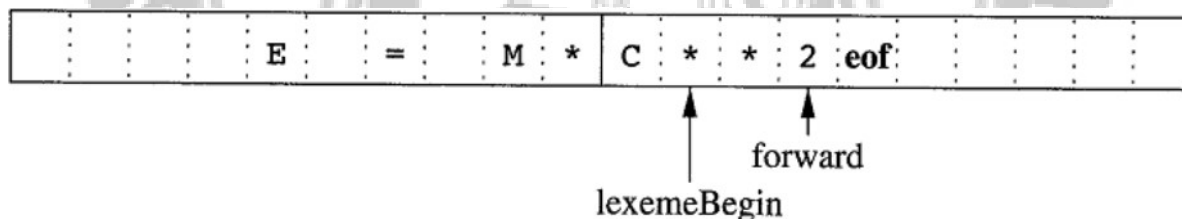


Figure : Using a Pair of Input Buffers

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 in to 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. The 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; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

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. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, 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 make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

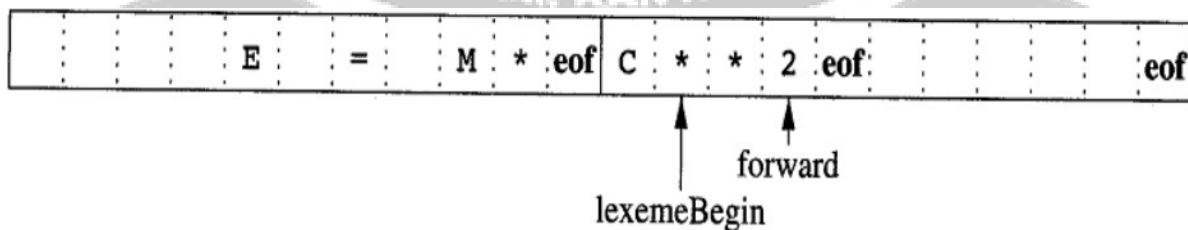


Figure : Sentinel at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of

a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```

switch ( *forward++ )
{
    case eof: if (forward is at end of first buffer )
        {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer )
        {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
    break;
}

```

Figure: use of switch-case for the sentential

OBSERVE OPTIMIZE OUTSPREAD

