* 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.

* In this scheme,

  → amount of look ahead is limited and this limited look-ahead 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 Eg)

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 paranthese

Code to advance forward pointer:

```
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 ;
```

2. Sentinels:

- * The previous code to advance the forward point 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.

* Upon receiving a "get next token" command from the parser. the lexical analyzer reads input characters until it can identify the next token

* Lexical analyzer also perform certain secondary tasks at the user interface

→ One such task is shipping out from the source program comments and white space in the form of blank, tab and new line characters

→ Another is correlating the error messages from the compiler with the source program

* Lexical analyzers are divided into a cascade of two phases

1. Scanning
2. Lexical Analysis

* The scanner is responsible for doing simple tasks while the lexical analyzer proper does the more complex operations

* For eg) a FORTRAN compiler might use a scanner to eliminate blanks from the input

Tokens, Patterns, Lexemes :

* The 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 eg) const pi = 3.1416.

pi → lexeme for the token identifier

Token	Sample Lexemes	Informal Description of Patterns
const	const	const
if	if	if
relational Operator	<, <=, >, >=, <>, ==	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letter and digit
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any character between " and " except the following

- In most programming languages, the following constructs are treated as tokens
  → keywords, operators, identifiers, constants, literal, strings and punctuation symbol such as paranthesis, commas and semicolons.

- A pattern is a rule describing the set of lexemes that can represent a particular token in the source programs.

Attributes of 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.
- The lexical analyzer collects information about tokens and their associated attributes.
- The tokens influence parsing decisions, the attributes influence the transition of tokens.
- 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 tokens and associated attribute values for

E = M * C * * 2

are written below as a sequence of pairs

<id, pointer to the symbol table Entry for E>

<assign_op>

<id, pointer to the symbol table entry for M>

<mult_op>

<id, pointer to the symbol table entry for C>

<exp_op>

<nump, integer value 2>

LEXICAL ERRORS:

* Few errors are discernible at the lexical level alone, because lexical analyzer has a very localized

* For eg) 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 mispelling of the keyword if or an undeclared function identifier.

* A lexical analyzer must return the token for an identifier and some other phase of the compiler may handle any error.

* But suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the pattern for tokens matches a prefix of the remaining input.

* The simplest error recovery strategy is "panic mode" recovery.

"We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token".

* 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.