CSE 309 (Compilers)

Lexical Analysis

Dr. Muhammad Masroor Ali

Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

November, 2009

Version: 1.1, Last modified: November 1, 2009

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.

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.

The Role of the Lexical Analyzer

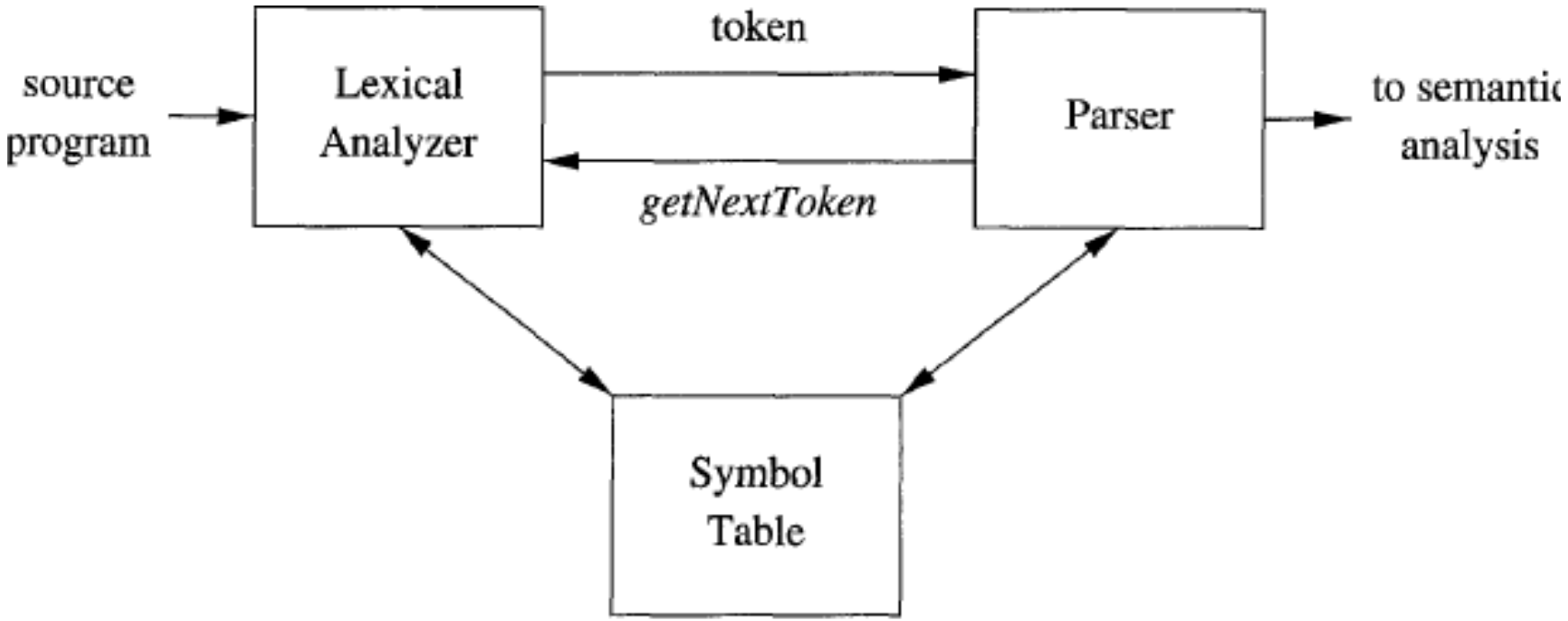
what is the role of the lexical analyzer?

- 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.

The Role of the Lexical Analyzer — *continued*

- 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.

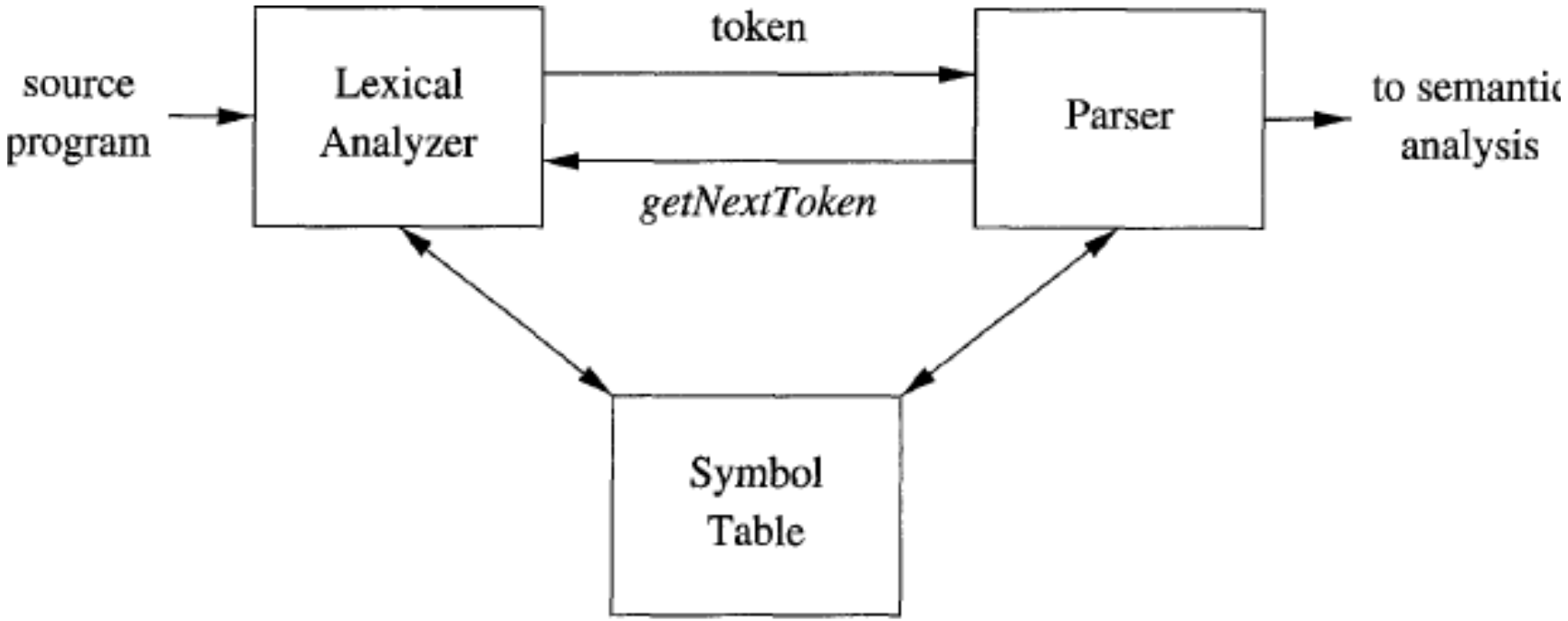
The Role of the Lexical Analyzer — *continued*



Interactions between the lexical analyzer and the parser

- These interactions are suggested in figure.
- Commonly, the interaction is implemented by having the parser call the lexical analyzer.

The Role of the Lexical Analyzer — *continued*



Interactions between the lexical analyzer and the parser

- 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.

The Role of the Lexical Analyzer — *continued*

- The lexical analyzer is the part of the compiler that reads the source text.
- It may perform certain other tasks besides identification of lexemes.
- One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

The Role of the Lexical Analyzer — *continued*

- Another task is correlating error messages generated by the compiler with the source program.
- For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

The Role of the Lexical Analyzer — *continued*

- In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

The Role of the Lexical Analyzer — *continued*

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

  a) *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.

  b) *Lexical* analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

The Role of the Lexical Analyzer — *continued*

- Sometimes, lexical analyzers are divided into a cascade of two processes:
  a) *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.
  b) *Lexical* analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis Versus 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.

Lexical Analysis Versus Parsing — *continued*

What are the differences between lexical analysis and parsing?

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.
   - If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

Lexical Analysis Versus Parsing — *continued*

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.

Lexical Analysis Versus Parsing — *continued*

3. Compiler portability is enhanced.
   - Input-device-specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns, and Lexemes

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

*** What is a token?

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.
- We shall generally write the name of a token in boldface.
- We will often refer to a token by its token name.

Tokens, Patterns, and Lexemes

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

*** What is a patern?

A pattern is a description of the form that the lexemes of a token may take.

  - 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.

Tokens, Patterns, and Lexemes

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

*** What is a lexeme?

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.

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"`

Examples of tokens

- Figure gives some typical tokens, their informally described patterns, and some sample lexemes.

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"`

Examples of tokens

- To see how these concepts are used in practice, in the C statement `printf("Total = %d\n", score);}` both `printf` and `score` are lexemes matching the pattern for token **id**, and `"Total = %d\n"` a lexeme matching **literal**.

Tokens, Patterns, and Lexemes — *continued*

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword.
   - The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases 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.

Attributes for Tokens — *continued*

- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token.
- The token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

Attributes for Tokens — *continued*

- We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information.
- The most important example is the token **id**, where we need to associate with the token a great deal of information.

Attributes for Tokens — *continued*

- Normally, information about an identifier e.g.,
  - its lexeme,
  - its type,
  - and the location at which it is first found (in case an error message about that identifier must be issued)

  — is kept in the symbol table.
- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example

- The token names 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**>
  <**number**, integer value 2>

Example — *continued*

- 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.
- In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string `fi` is encountered for the first time in a C program in the context:
  `fi ( a == f(x)) dots`
  a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier.
- Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string `fi` is encountered for the first time in a C program in the context:
  `fi ( a == f(x)) dots`
  a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier.
- Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

Lexical Errors — *continued*

- 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.

Lexical Errors — *continued*

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.

Lexical Errors — *continued*

lexerror1.cpp

```
#include <iostream>
int main()
{
   `int i, j, k;
   return 0;
}
```

Response from gcc

```
lexerror1.cpp:4: error: stray ` in program
```

Lexical Errors — *continued*

lexerror2.cpp

```
int main()
{
  int 5test;

  return 0;
}
```

Response from gcc

```
lexerror2.cpp:3:7: error: invalid suffix
"test" on integer constant
```

Lexical Errors — *continued*

- 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 a single transformation.
- This strategy makes sense, since in practice most lexical errors involve a single character.

Lexical Errors — *continued*

- A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes.
- But this approach is considered too expensive in practice to be worth the effort.

Tricky Problems When Recognizing Tokens

- Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input.
- However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token.

Tricky Problems When Recognizing Tokens — *continued*

- The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90.
- In the statement

  `DO 5 I = 1.25`

  it is not apparent that the first lexeme is `D05I`, an instance of the identifier token, until we see the dot following the 1.
- Note that blanks in fixed-format Fortran are ignored (an archaic convention).

Tricky Problems When Recognizing Tokens — *continued*

- Had we seen a comma instead of the dot, we would have had a do-statement

  `DO 5 I = 1,25`

  in which the first lexeme is the keyword `DO`.

Input Buffering

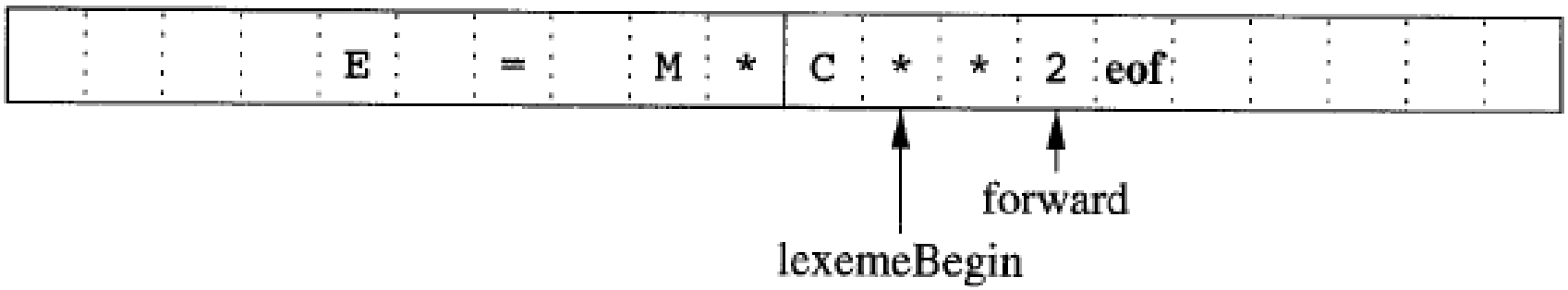
- Let us examine some ways that the simple but important task of reading the source program can be speeded.
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.
- There are many situations where we need to look at least one additional character ahead.

Input Buffering — *continued*

- 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 lookaheads safely.
- We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

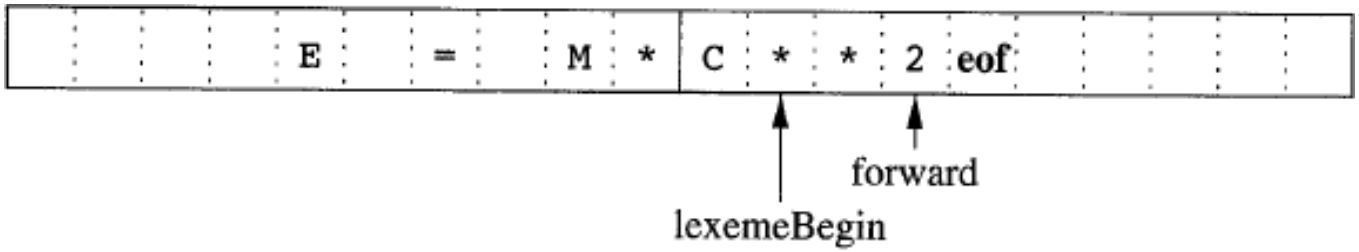
Buffer Pairs

- The amount of time taken is high to process characters 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, as suggested in figure.



Using a pair of input buffers

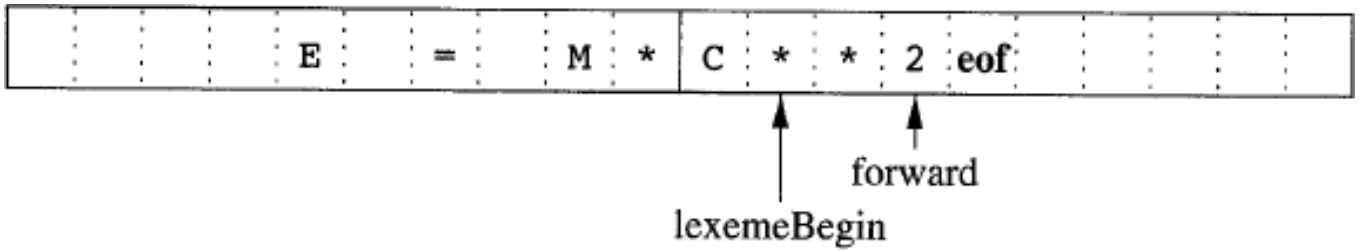
Buffer Pairs — *continued*



Using a pair of input buffers

- Each buffer is of the same size N.
- 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.

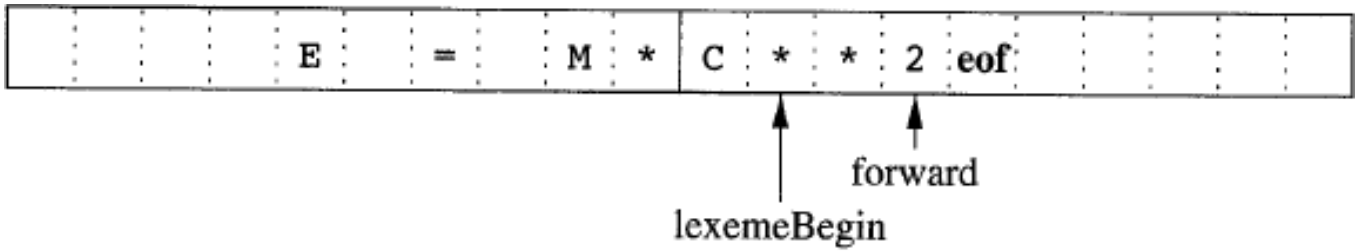
Buffer Pairs — *continued*



Using a pair of input buffers

- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- This **eof** is different from any possible character of the source program.

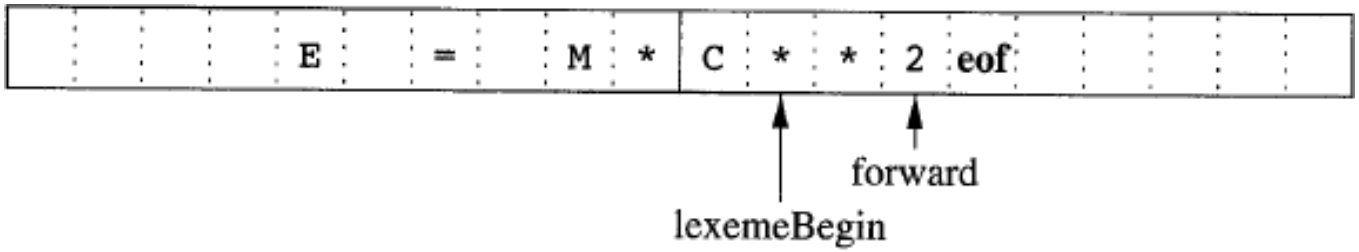
Buffer Pairs — *continued*



Using a pair of input buffers

- 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.

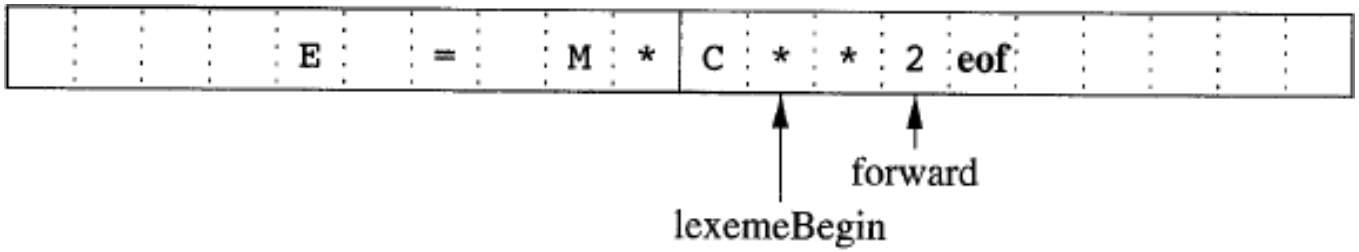
Buffer Pairs — *continued*



Using a pair of input buffers

- 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 figure, we see `forward` has passed the end of the next lexeme, `**` (the Fortran exponentiation operator), and must be retracted one position to its left.

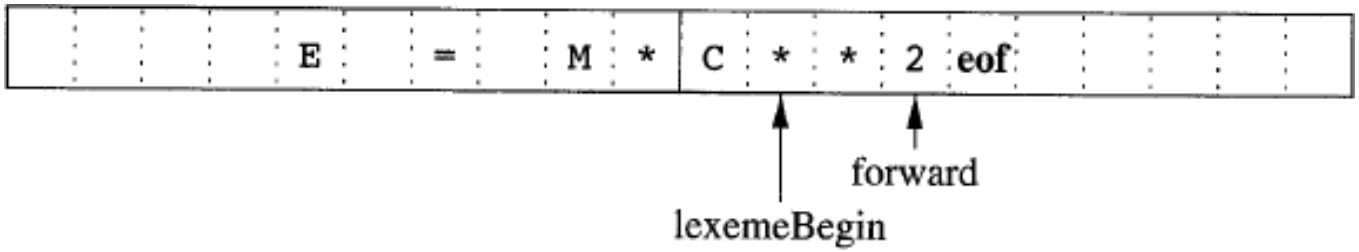
Buffer Pairs — *continued*



Using a pair of input buffers

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

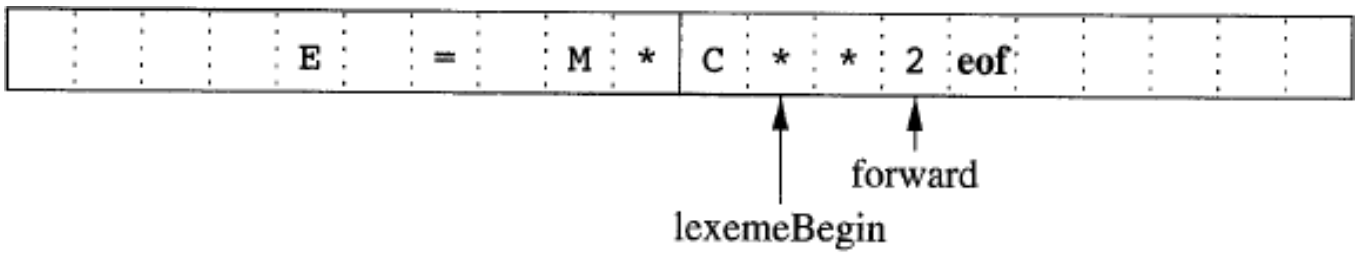
Buffer Pairs — *continued*



Using a pair of input buffers

- 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



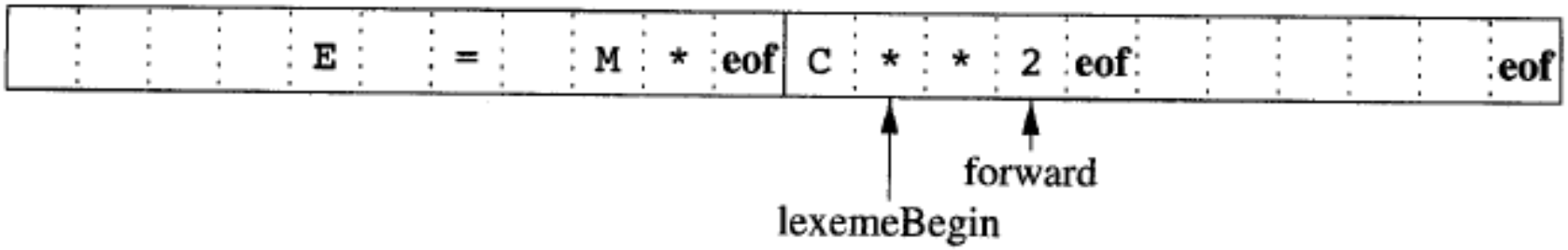
Using a pair of input buffers

- If we use the previous 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 multiway branch).

Sentinels — *continued*

- 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**.

Sentinels — *continued*



Sentinels at the end of each buffer

- Figure shows the same arrangement as previous, but with the sentinels added.
- 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 a buffer means that the input is at an end.

```
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;
        Cases for the other characters
}
```

Lookahead code with sentinels

- Figure 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.

Specification of Tokens

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

Recognition of Tokens

- We can express patterns using regular expressions.

Recognition of Tokens

- Our discussion will make use of the following running example.

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

A grammar for branching statements

Example

- The grammar fragment describes a simple form of branching statements and conditional expressions.
- This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

A grammar for branching statements

Example

- For **relop**, we use the comparison operators of languages like Pascal or SQL, where $=$ is "equals" and $<>$ is "not equals," because it presents an interesting structure of lexemes.

stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	$\rightarrow$	*term* **relop** *term*
	|	*term*
term	$\rightarrow$	**id**
	|	**number**

A grammar for branching statements

Example

- The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned.

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

A grammar for branching statements

Example — *continued*

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

- The patterns for these tokens are described using regular definitions.

Example — *continued*

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

- For this language, the lexical analyzer will recognize the keywords **if**, **then**, and **else**, as well as lexemes that match the patterns for *relop*, *id*, and *number*.

Example — *continued*

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

- To simplify matters, we make the common assumption that keywords are also *reserved words*.
- They are not identifiers, even though their lexemes match the pattern for identifiers.

Example — *continued*

- In addition, we assign the lexical analyzer the job of stripping out white- space, by recognizing the "token" *ws* defined by:

$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

- Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names.
- Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser.
- We rather restart the lexical analysis from the character that follows the whitespace.
- It is the following token that gets returned to the parser.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

- Our goal for the lexical analyzer is summarized in figure.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

- That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value is returned.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

- Note that for the six relational operators, symbolic constants `LT`, `LE`, and so on are used as the attribute value, in order to indicate which instance of the token relop we have found.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

- The particular operator found will influence the code that is output from the compiler.

End of Slides