System Programming and Compiler Construction

MODULE 5 (2)
COMPILERS : ANALYSIS PHASE

Prof. Sonal Shroff
Computer Engineering Department
TSEC

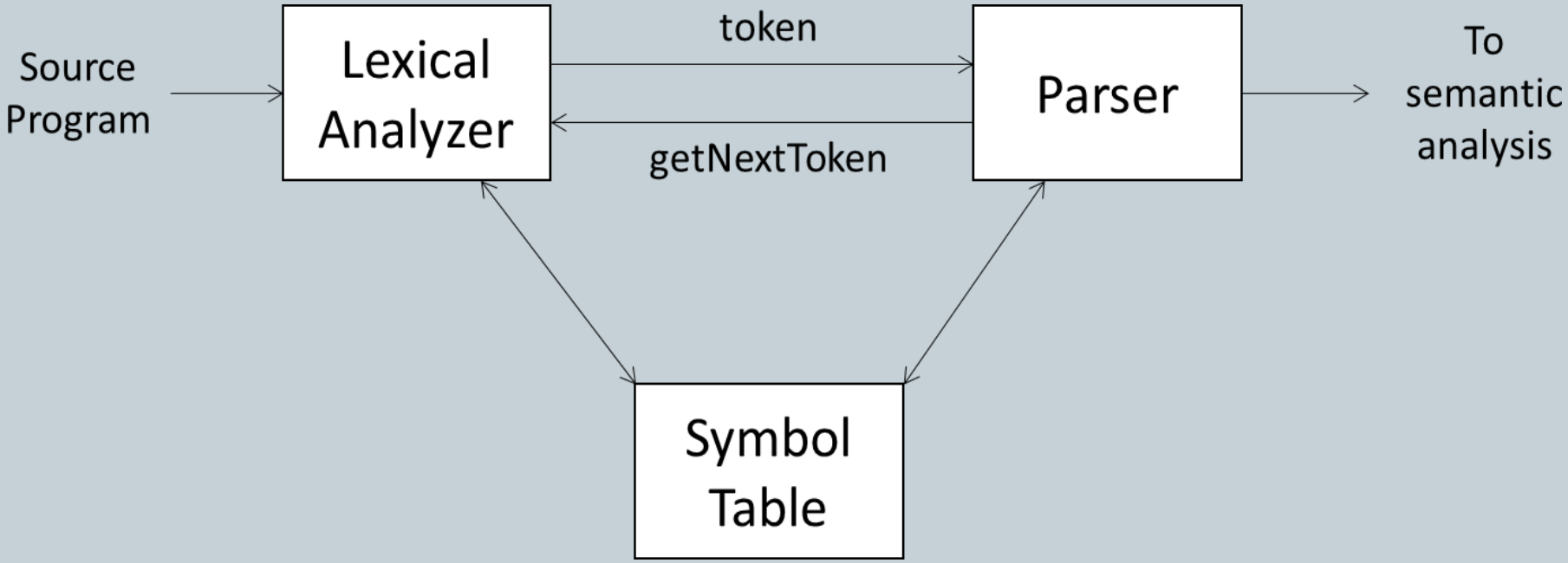
Lexical Analysis

The role of the Lexical Analyzer

- Lexical Analyzer is the first phase of a compiler
- Main Task: to read the input character from source program, group them into lexemes and produce output as sequence of tokens for each lexemes in the source program.
- The stream of tokens is sent to the parser for syntax analysis.
- Lexical analyzer interacts with the symbol table as well.
- Lexical analyzer enters lexemes into the symbol table.

Lexical Analysis

- The role of the Lexical Analyzer



Interaction between the lexical analyzer and the parser

Lexical Analysis

The role of the Lexical Analyzer

- Identification of lexemes
- Stripping out comments and white spaces
- Correlating error messages from the compiler with the source program
- If source language uses some macro-preprocessor, then these macro expansion may also be performed

Lexical Analysis

- Lexical analyzer is divided into cascade of two processes

1. Scanning :
   - Deletion of comments
   - Compaction of consecutive whitespace characters into one, etc
2. Lexical analysis
   - To produce tokens from the output of the scanner

Lexical Analysis

- Lexical analysis Vs Parsing

Why analysis portion of a compiler is separated into lexical analysis and parsing (syntax analysis)

1. Simplicity of design is the most important consideration
2. Compiler efficiency is improved
3. Compiler portability is enhanced

Lexical Analysis

- Tokens Patterns and Lexemes

Token – a pair consisting of token name and an optional attribute value.

Token names are abstract symbol representing a kind of lexical unit.

Attribute value is commonly referred to as token value.

Each token represents a sequence of characters that can be treated as a single entity.

Ex. Identifiers, keywords, constants, operators, punctuation symbols.

Lexical Analysis

- Tokens Patterns and Lexemes

Token – broadly classified into 2 types

1. Specific strings such as if, else, comma, semicolon etc.
2. Classes of strings such as identifiers, constants, labels etc.

Ex. A = B + C * 5

After lexical analysis

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

Lexical Analysis

- Tokens Patterns and Lexemes

Pattern – a rule that defines a set of input strings for which the same token is produced as output.

Regular expressions play an important role in specifying patterns

For keyword, pattern is – sequence of characters.

For identifier, pattern is – letter (letter/ digit)*

Lexeme – 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.

Lexical Analysis

- Examples of tokens, patterns and 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, count, D2
number	Any Numeric Constant	3.1416, 0, 6.02E23
literal	Any character between " and "	"core dumped"

Lexical Analysis

- Classes that cover most of the Tokens

- Keyword – The pattern for a keyword is the same as the keyword itself.
- Operators – either individually or in classes.
- Identifier – one token for all identifiers
- Constants – one or more tokens representing constants such as numbers and literal strings
- Punctuation symbols – tokens for parenthesis, comma, semicolon, etc.

Lexical Analysis

- Example

```
int add(int a, int b)
//addition of two numbers
{
    int c;
    c=a+b;
    return
    c;
}
```

S.No.	Lexeme	Token	S.No.	Lexeme	Token
1.	int	Keyword	13.	;	Punctuation
2.	add	Identifier	14.	c	Identifier
3.	(	Punctuation	15.	=	Operator
4.	int	Keyword	16.	a	Identifier
5.	a	Identifier	17.	+	Operator
6.	,	Punctuation	18.	b	Identifier
7.	int	Keyword	19.	;	Punctuation
8.	b	Identifier	20.	return	Keyword
9.	)	Punctuation	21.	c	Identifier
10.	{	Punctuation	22.	;	Punctuation
11.	int	Keyword	23.	}	Punctuation
12.	c	Identifier			

Lexical Analysis

- Attributes for tokens

When more than one lexeme can match a pattern, then lexical analyzer must provide additional information about the lexeme.

Lexical analyzer provides token name and its attribute value.

The attribute value describes the lexeme represented by the token.

Token name influences parsing decisions.

Attribute value influences the translation of token after the parse.

Lexical Analysis

- Attributes for tokens

A token has a single attribute – A pointer to the symbol table entry in which the information about the token is kept

The pointer becomes attribute for the token

The attribute may have a structure that combines the several pieces of information.

Example – token **identifier**

Lexical Analysis

- Attributes for tokens

Example – token **identifier**

Information about an identifier – its lexeme, its type, its location etc is stored in symbol table.

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

Lexical Analysis

E = M * C ** 2

Token names and attributes for above statements

1. <id, pointer to symbol-table entry for E>
2. <assign_op,>
3. <id, pointer to symbol-table entry for M>
4. <mult_op,>
5. <id, pointer to symbol-table entry for C>
6. <exp_op,>
7. <num, integer value 2>

Lexical Analysis

Lexical errors

Error Recovery Actions

1. Panic Mode Recovery- Delete successive characters from the remaining input until the lexical analyzer finds a well-formed token
2. Deleting one character from remaining input
3. Inserting a missing character into the remaining input
4. Replacing an incorrect character by a correct character
5. Transposing two adjacent characters

Lexical Analysis

Input Buffering

- **Scanner** is the only part of the compiler which reads a complete program.
- Lexical Analyzer is the only phase of the compiler that reads the source program character by character
- It takes approx. 25-30% of the compiler's time.
- So, Lexical Analysis phase consumes considerable amount of time, due to which, compilation time goes low.
- Hence, speed of Lexical Analysis is a major concern!

Lexical Analysis

- Input Buffering

As large amount of time consumption takes place in moving characters, Specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character by lexical Analyzer.

A block of data is first read into a BUFFER and then it is read by Lexical Analyzer

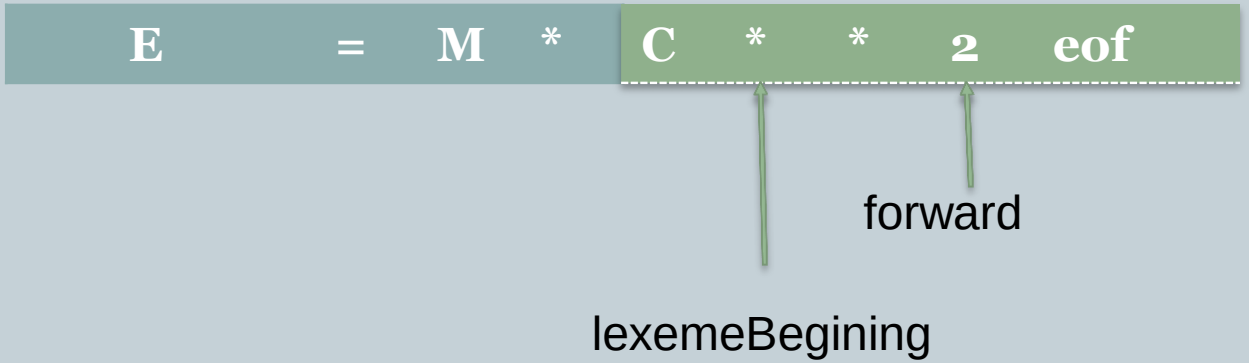
Lexical Analysis

Buffer Pairs

- 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.
- “eof’ is different from any possible character of the source program.

Lexical Analysis

- Buffer Pairs

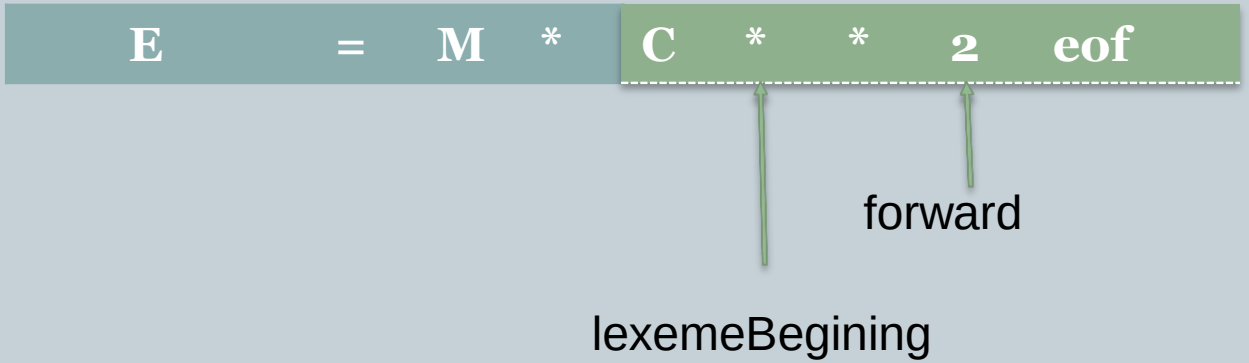


Each buffer is of the same size N.

N is usually the size of the disk block. Eg. 4096 bytes

Lexical Analysis

- Buffer Pairs



Two pointers are maintained

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

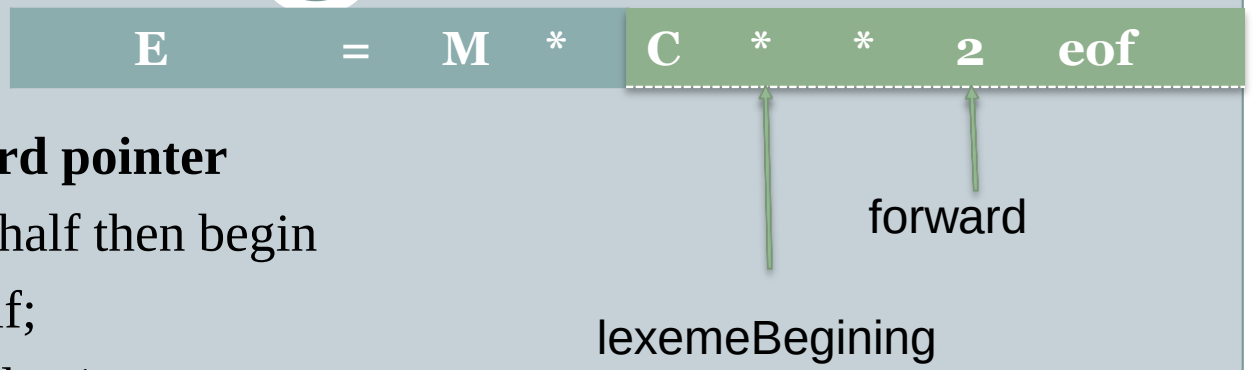
Lexical Analysis

Buffer Pairs

- Once the next lexeme is determined, forward is set to the character at its right end.
- 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

Lexical Analysis

Buffer Pairs



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;
```

Lexical Analysis

Sentinels

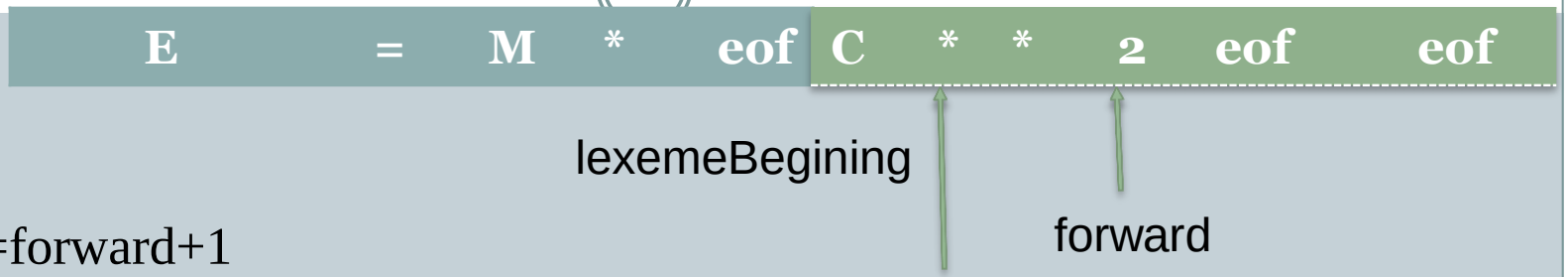
- In Two Buffer Scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

1. For end of buffer.
2. To determine what character is read.

The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

The sentinel is a special character that cannot be part of the source program. ***(eof* character is used as sentinel)**

Lexical Analysis




```
forward:=forward+1
if forward= eof then begin
      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 /* eof within a buffer signifying end of input */
          terminate
end
```