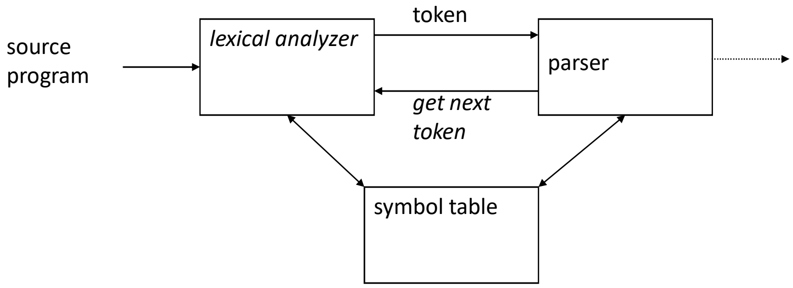
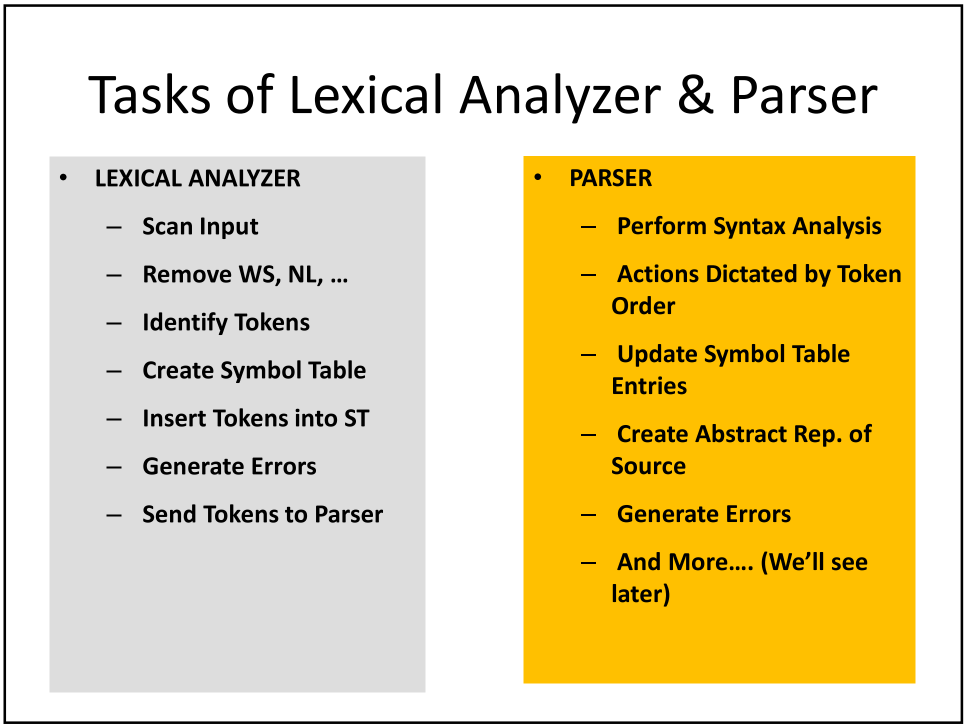
Compiler Construction

Lexical Analysis

Chapter -3

Lexical Analysis

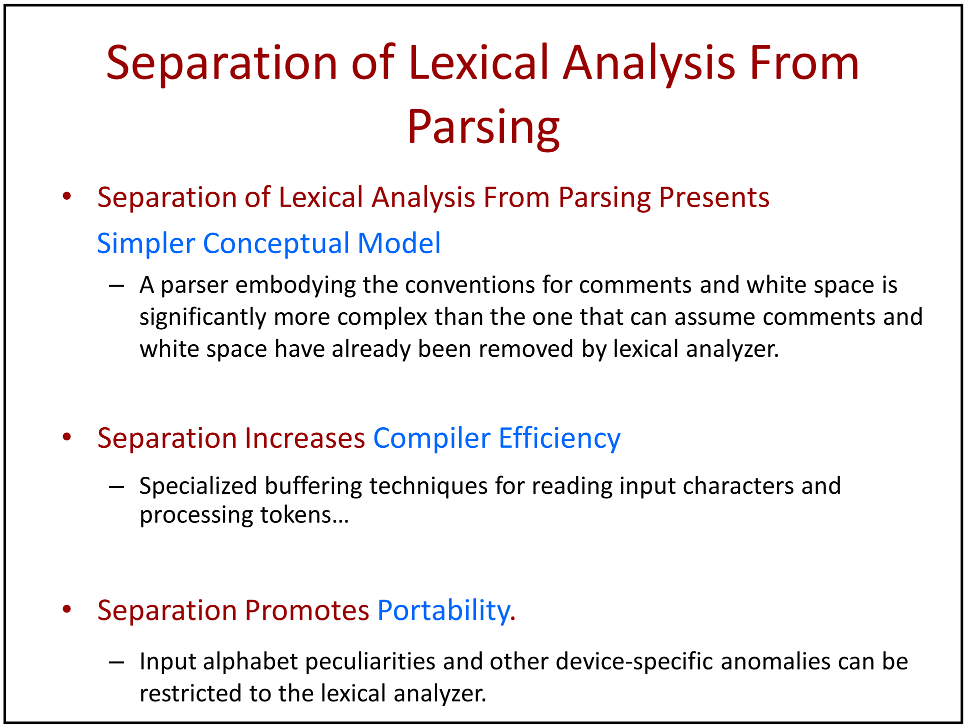




Tasks of Lexical Analyzer & Parser

- LEXICAL ANALYZER
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Generate Errors
  - Send Tokens to Parser

- PARSER
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Rep. of Source
  - Generate Errors
  - And More…. (We'll see later)



Separation of Lexical Analysis From Parsing

- Separation of Lexical Analysis From Parsing Presents Simpler Conceptual Model
  - A parser embodying the conventions for comments and white space is significantly more complex than the one that can assume comments and white space have already been removed by lexical analyzer.

- Separation Increases Compiler Efficiency
  - Specialized buffering techniques for reading input characters and processing tokens…

- Separation Promotes Portability.
  - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

Basic Concepts

- What are Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - Recall File and OS Wildcards ([A-Z]*.*)
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

Basic Concepts

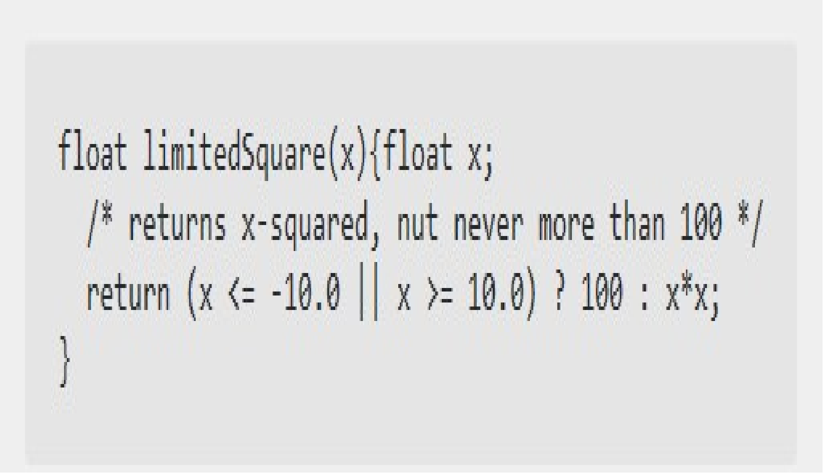
Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except “

Classifies
Pattern

Actual values are critical. Info is :

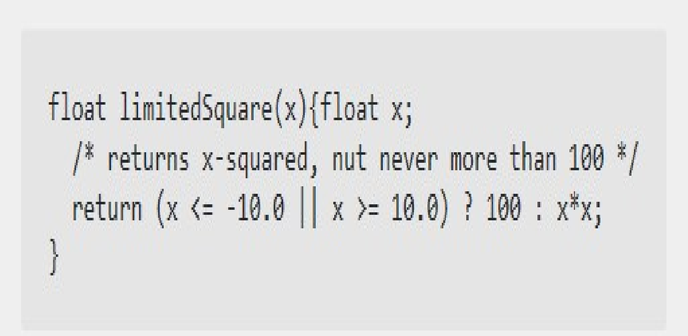
1. Stored in symbol table
2. Returned to parser

Example: Identify the Tokens



```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

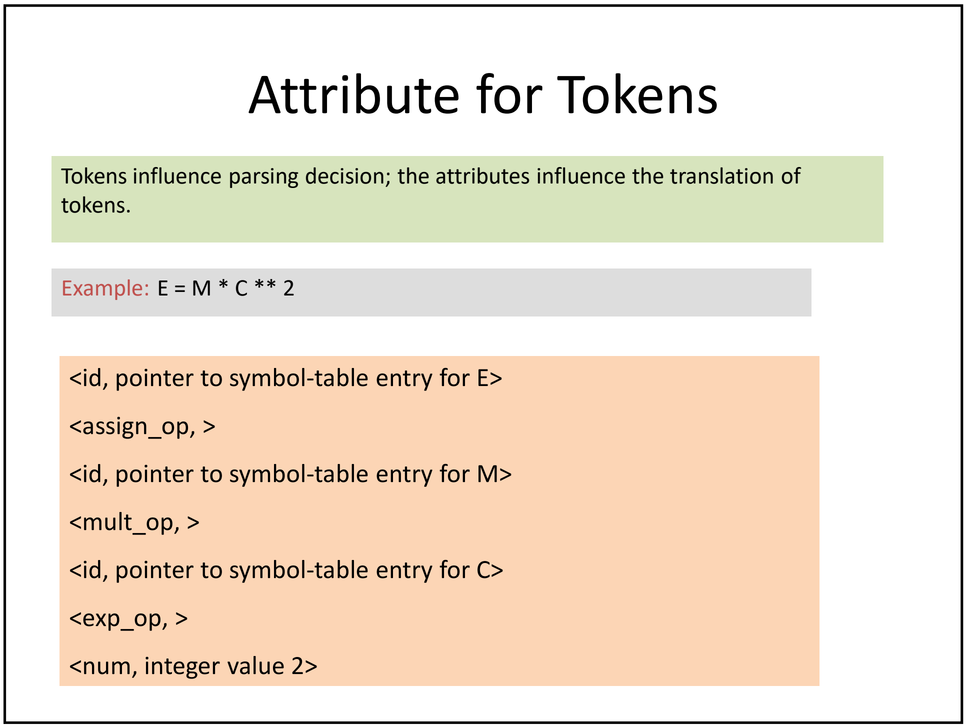
Example: Identify the Tokens



```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

<float>
<id, limitedSquare>
<(>
<id, x>
<) >
<{>
<float>
<id, x>

<return>
<(>
<id,x>
<op, “<=“>
<num,-10.0>
<op, “||“>
<id,x>
<op, “>=“>
<num,10.0>
<) >
<op, “?”>
<num,100>
<op, “:”>
<id, x>
<op,“*”>
<id, x>
<;>
<}>



Attribute for Tokens

Tokens influence parsing decision; the attributes influence the translation of tokens.

Example: E = M * C ** 2

<id, pointer to symbol-table entry for E>

<assign_op, >

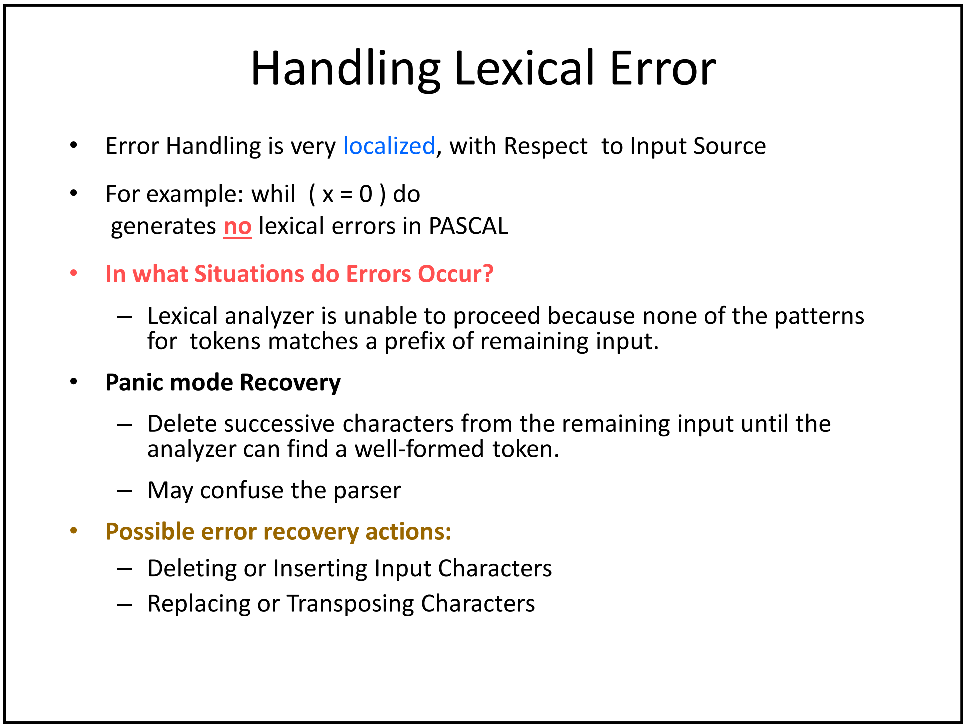
<id, pointer to symbol-table entry for M>

<mult_op, >

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

<exp_op, >

<num, integer value 2>



Handling Lexical Error

- Error Handling is very localized, with Respect to Input Source
- For example: whil (x = 0) do
  generates no lexical errors in PASCAL
- **In what Situations do Errors Occur?**
  - Lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of remaining input.
- **Panic mode Recovery**
  - Delete successive characters from the remaining input until the analyzer can find a well-formed token.
  - May confuse the parser
- **Possible error recovery actions:**
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters

Fortran Example

DO 5 I = 1.25

Here, there are 3 tokens: DO5I, = and 1.25

DO 5 I = 1,25

Here, there are 7 tokens: DO, 5, I, =, 1, ‘,’ and 25

Buffering Technique

- Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- Use a function **ungetc** to push lookahead characters back into the input stream.
- Large amount of time can be consumed moving characters.

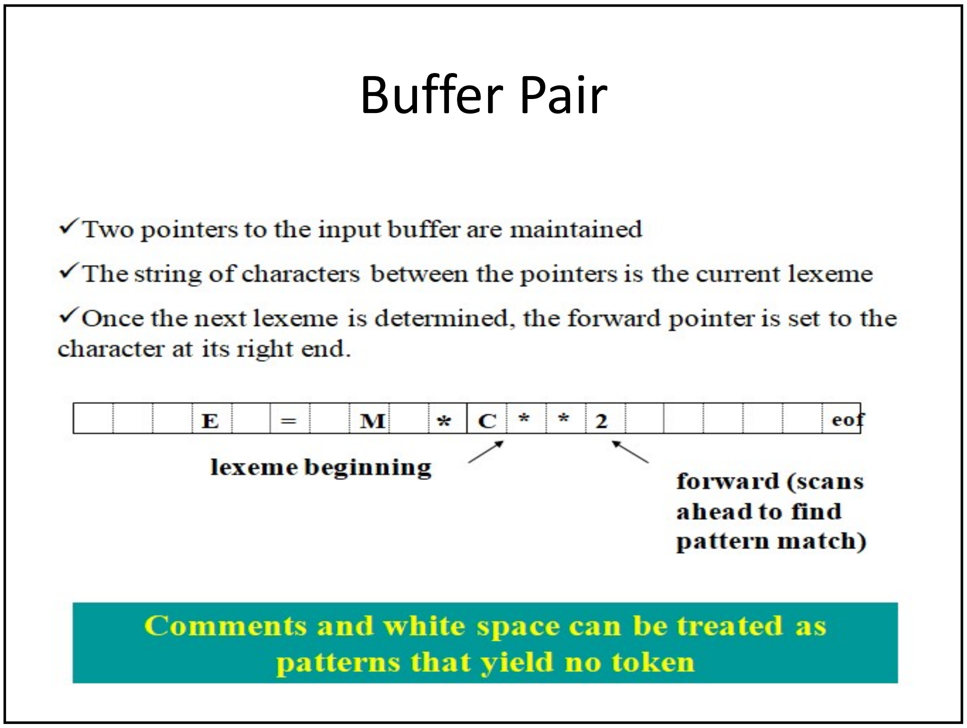
Special Buffering Technique

✓Use a buffer divided into two N-character halves

✓N = Number of characters on one disk block

✓One system command read N characters

✓Fewer than N character => eof



Buffer Pair

✓Two pointers to the input buffer are maintained

✓The string of characters between the pointers is the current lexeme

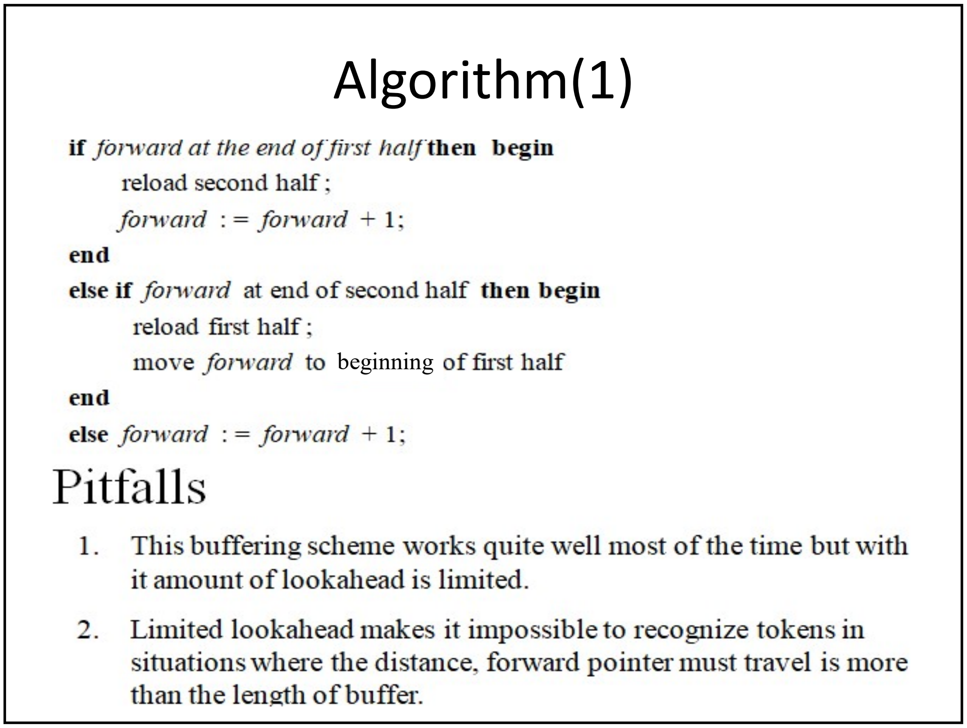
✓Once the next lexeme is determined, the forward pointer is set to the character at its right end.

			E		=		M		*	C	*	*	2							eof

lexeme beginning

forward (scans ahead to find pattern match)

Comments and white space can be treated as patterns that yield no token

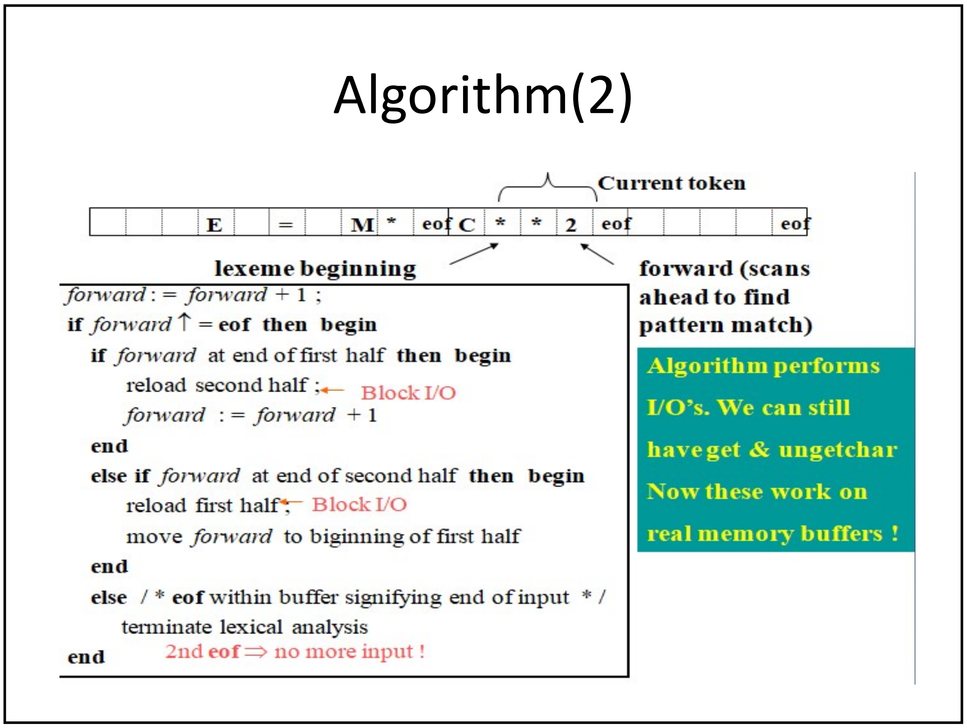


Algorithm(1)

```
if forward at the 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;
```

Pitfalls

1. This buffering scheme works quite well most of the time but with it amount of lookahead is limited.
2. Limited lookahead makes it impossible to recognize tokens in situations where the distance, forward pointer must travel is more than the length of buffer.



Algorithm(2)

Current token

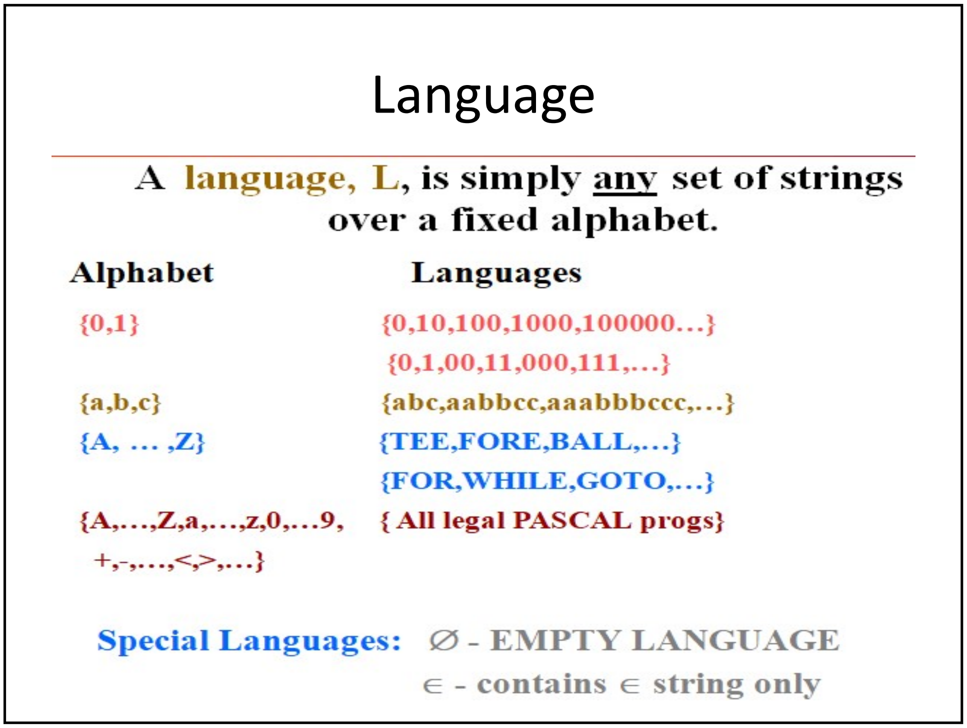
			E		=		M	*	eof	C	*	*	2	eof					eof

lexeme beginning

forward (scans ahead to find pattern match)

```
forward : = forward + 1 ;
if forward ↑ = eof then begin
   if forward at end of first half then begin
      reload second half ; ← Block I/O
      forward : = forward + 1
   end
   else if forward at end of second half then begin
      reload first half; ← Block I/O
      move forward to biginning of first half
   end
   else / * eof within buffer signifying end of input * /
      terminate lexical analysis
end      2nd eof ⇒ no more input !
```

Algorithm performs I/O's. We can still have get & ungetchar Now these work on real memory buffers !



Language

A language, L, is simply any set of strings over a fixed alphabet.

Alphabet	Languages

{0,1}	{0,10,100,1000,100000...}
	{0,1,00,11,000,111,...}

{a,b,c}	{abc,aabbcc,aaabbbccc,...}

{A, ... ,Z}	{TEE,FORE,BALL,...}
	{FOR,WHILE,GOTO,...}

{A,...,Z,a,...,z,0,...9, +,-,...,<>,...}	{ All legal PASCAL progs}

Special Languages: ∅ - EMPTY LANGUAGE

∈ - contains ∈ string only

Language Operations

OPERATION	DEFINITION
union of L and M written L ∪ M	L ∪ M = {s | s is in L or s is in M}
concatenation of L and M written LM	LM = {st | s is in L and t is in M}
Kleene closure of L written L*	L*= $\bigcup_{i=0}^{\infty} L^i$ L* denotes "zero or more concatenations of " L
positive closure of L written L^+	L^+= $\bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of " L

Regular Language

- A Regular Expression is a Set of Rules / Techniques for Constructing Sequences of Symbols (Strings) From an Alphabet.
- Let Σ Be an Alphabet, r a Regular Expression Then L(r) is the Language That is Characterized by the Rules of r

Algebraic laws for regular Expressions

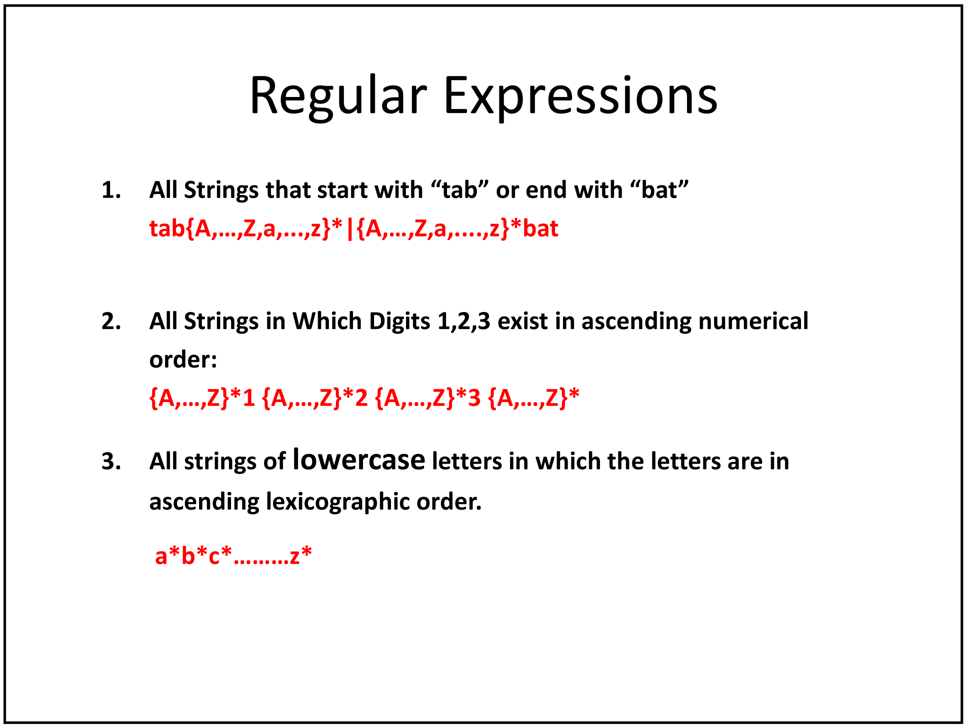
AXIOM	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = r s | r t (s | t) r = s r | t r	concatenation distributes over |
∈r = r r∈ = r	∈ Is the identity element for concatenation
r* = (r | ∈)*	relation between * and ∈
r** = r*	* is idempotent

Regular Expression

Regular expressions over alphabet Σ

- ∈ is a regular expression denoting {∈}, set containing the empty string
- If a is in Σ, a is a regular expression that denotes {a}
- Let r and s be regular expressions with languages L(r) and L(s). Then
  - (a) (r) | (s) is a regular expression ⇒ L(r) ∪ L(s)
  - (b) (r)(s) is a regular expression ⇒ L(r) L(s)
  - (c) (r)* is a regular expression ⇒ (L(r))*
  - (d) (r) is a regular expression ⇒ L(r)

All are Left-Associative. Parentheses are dropped as allowed by precedence rules.



Regular Expressions

1. All Strings that start with “tab” or end with “bat”
   tab{A,…,Z,a,...,z}*|{A,…,Z,a,....,z}*bat

2. All Strings in Which Digits 1,2,3 exist in ascending numerical order:
   {A,…,Z}*1 {A,…,Z}*2 {A,…,Z}*3 {A,…,Z}*

3. All strings of lowercase letters in which the letters are in ascending lexicographic order.

   a*b*c*………z*