# Lexical Analysis

Definition: **Lexical Analysis** is the operation of dividing the input program into sequence of lexemes(tokens)

# Input

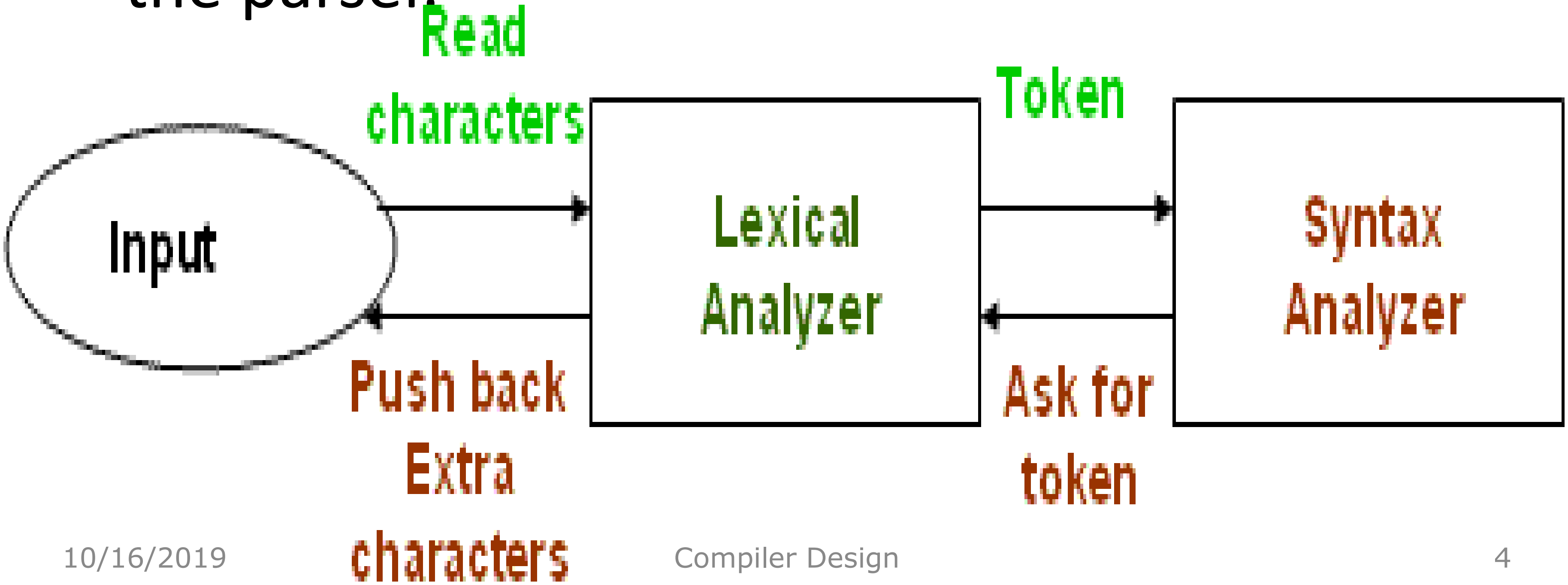
- Read string input
  - Might be sequence of characters
  - Might be sequence of lines
  - Character set:

# The Output

- A series of *tokens: kind, location, name (if any)*
  - Punctuation ( ) ; , [ ]
  - Operators + - ** :=
  - Keywords begin end if while try catch
  - Identifiers Square_Root
  - String literals “press Enter to continue”
  - Character literals ‘x’
  - Numeric literals
    - Integer: 123
    - Floating_point: 4_5.23e+2
    - Based representation: 16#ac#

# Introduction

- The role of the **lexical analyzer** is to read a sequence of characters from the source program and produce tokens to be used by the parser.



- The stream of tokens is sent to the parser for syntax analysis
- The lexical analyzer also interacts with the symbol table, e.g., when the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table

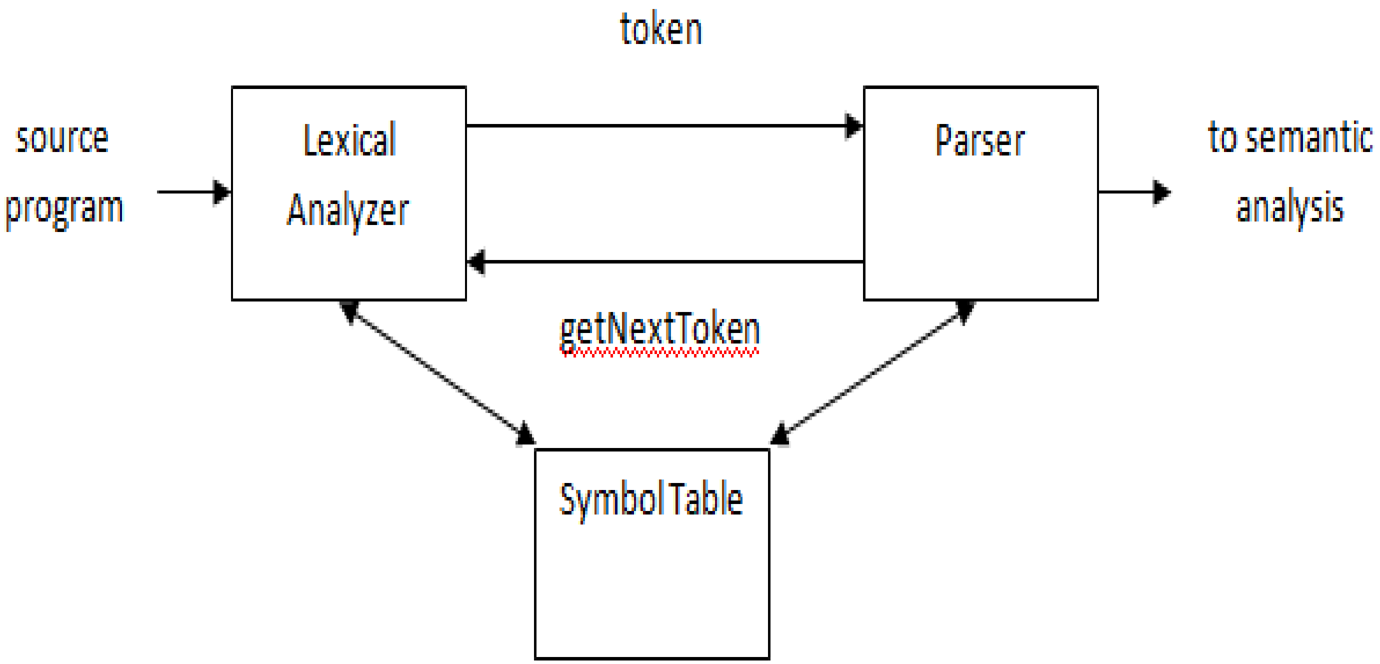


Figure 2.1: Interactions between the lexical analyzer and the parser

- “getNextToken” is a command sent from the parser to the lexical analyzer.
- On receipt of the command the lexical analyzer scans the input until it determines the next token, and returns it.
- It skips comments and whitespaces while creating these tokens

# Introduction (cont’d)

- The lexical analyzer can also perform the following secondary tasks:
  - stripping out blanks, tabs, new lines
  - stripping out comments
  - keeping track of line numbers **(to correlate an error with the source and line number)**
  - Expanding macros in some lexical analyzers

# Token, pattern, lexeme

- **Token:** a sequence of characters from the source program having a collective meaning.
- A classification for a common set of strings
- Examples Include <Identifier>, <number>, etc
- A single token can be produced by different sequences of characters.
- E.g. x, distance, count → IDENT
- Tokens are terminals in the grammar of the source language.

# Token, pattern, lexeme (cont’d)

- **Pattern:** a rule describing the set of lexemes that represent a token.
- Patterns are usually specified using regular expressions.

Example

[a-zA-Z]*

Lexemes matched: a, ab, count, …

# Token, pattern, lexeme (cont'd)

- **Lexeme**: a sequence of characters in the source program that is matched by a pattern for a certain token.
- Lexeme: The smallest logical units(words) of a program
  - **eg: i, sum, for,10,++,"%d\n",<=.**

Example:

x, distance, count ➔ IDENT

**token**: IDENT

**lexemes**: x, distance, count

# Token, pattern, lexeme (cont’d)

- Example: Some tokens and their lexemes in Pascal (a high level, case insensitive programming language)

| **Token** | **Some lexemes** | **Informal pattern** |
|---|---|---|
| begin | begin, Begin, BEGIN, beGin, … | Begin in small or capital letters |
| if | if, IF, iF, If | if in small or capital letters |
| ident | Distance, F1, x, Dist1, … | Letter followed by zero or more letters and/or digits |

Complete the above table for the C++ programming language.

# Token, pattern, lexeme (cont'd)

- In general, in programming languages, the following are tokens:
  - keywords
  - operators
  - identifiers
  - constants
  - literals
  - punctuation symbols
  - …

# Attributes of tokens

- When more than one lexeme matches a pattern, the scanner must provide additional information about the particular lexeme to the subsequent phases of the compiler.
- For ex., both **0** and **1** match the pattern for the token **num**. But the code generator needs to know which number is recognized.

# Attributes of tokens (cont'd)

- The lexical analyzer collects information about tokens into their associated attributes.
- Practically, a token has one attribute: a pointer to the symbol table entry in which the information about the token is kept.
- Symbol table entry contains information about the token such as the lexeme, the line number in which it was first seen, …

# Attributes of tokens (cont'd)

- For ex. consider x = y + 2

  The tokens and their attributes are written as:

  <**id**, pointer to symbol-table entry for x>

  <**assign_op**, >

  <**id**, pointer to symbol-table entry for y>

  <**plus_op**, >

  <**num**, integer value 2>

- **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>

# Errors

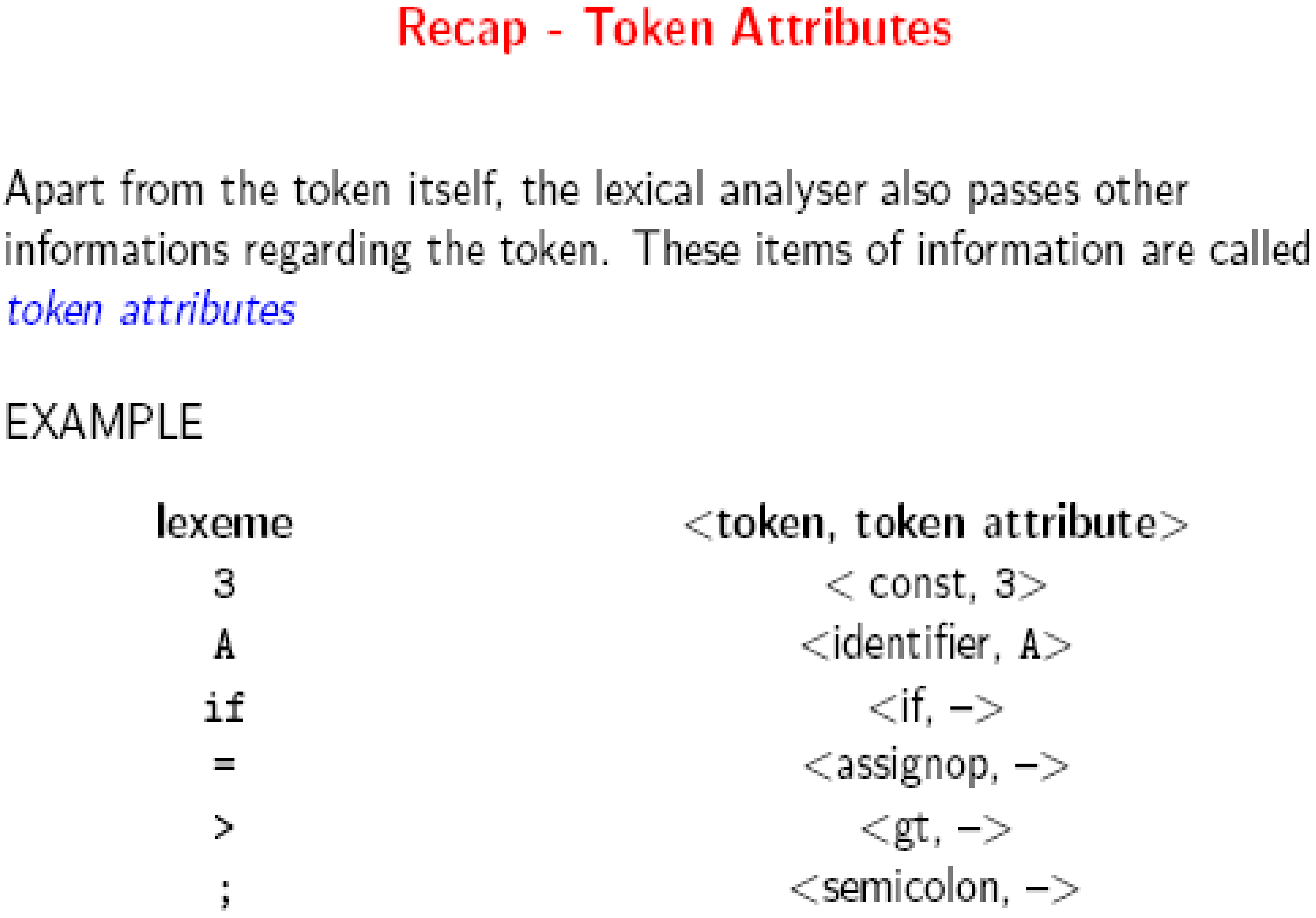
- Very few errors are detected by the lexical analyzer.
- For ex., if the programmer mistakes **wihle** for **while**, the lexical analyzer cannot detect the error (why?)
- Nonetheless, if a certain sequence of characters follows none of the specified patterns, the lexical analyzer can detect the error.
- **Examples of Lexical Errors:**
  - **Illegal characters in** source program
    - For ex. if there is a ? symbol in the source program and no pattern contains ?
  - Exceeding length of identifier or numeric constants
  - Unterminated strings or comments

# Handling Lexical Errors

- When an error occurs, the lexical analyzer recovers by:
  - **Panic mode recovery** :skipping (deleting) successive characters from the remaining input until the lexical analyzer can find a well-formed token (panic mode recovery)
  - deleting extraneous characters
  - inserting missing characters
  - replacing an incorrect character by a correct character
  - transposing two adjacent characters

# Example – tokens in Java

1. **Identifier:** A *Javaletter* followed by zero or more *Javaletterordigits*. A *Javaletter* includes the characters `a-z`, `A-Z`, `_` and `$`.
2. **Constants:**
   - 2.1 Integer Constants
     - Octal, Hex and Decimal
     - 4 byte and 8 byte representation
   - 2.2 Floating point constants
     - float - ends with `f`
     - double
   - 2.3 Boolean constants – `true` and `false`
   - 2.4 Character constants – `'a'`, `'\u0034'`, `'\t'`
   - 2.5 String constants – `""`, `"\""`, `"A string"`.
   - 2.6 Null constant – `null`.
3. **Delimiters:** (, ), {, }, [, ] , ;, . and ,
4. **Operators:** =, >, < . . . >>>=
5. **Keywords:** `abstract`, `boolean` . . . `volatile`, `while`.



# Recap - Token Attributes

Apart from the token itself, the lexical analyser also passes other informations regarding the token. These items of information are called *token attributes*

EXAMPLE

| lexeme | <token, token attribute> |
|---|---|
| `3` | < const, 3> |
| `A` | <identifier, `A`> |
| `if` | <if, –> |
| `=` | <assignop, –> |
| `>` | <gt, –> |
| `;` | <semicolon, –> |

# Specifying and recognizing tokens

- Regular expressions are used to specify the patterns of tokens.
- **Regular Definitions**
- For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols
- If ∑ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

where:

- Each $d_i$ is a new symbol, not in ∑ and not the same as any other of the d's, and
- Each $r_i$ is a regular expression over the alphabet ∑ $\{d_1,d_2, . . . , d_{i-1}\}$

- **Example:**

C identifiers are strings of letters, digits, and underscores. Here is a regular definition:

*letter_* → A | B | . . . | Z | a | b | . . . | z | _

*digit* → 0 | 1 | . . . | 9

*id*→ *letter_*(*letter_* | *digit*)*

- Regular expressions (RE) defined by an alphabet (terminal symbols) and three operations:
  - Alternation $RE_1 \mid RE_2$
  - Concatenation $RE_1 \, RE_2$
  - Repetition RE* (zero or more RE’s)

- Extensions of Regular Expressions
- ***One or more instances.*** The unary postfix operator + represents the positive closure of a regular expression and its language. The operator + has the same precedence and associativity as the operator *. $r^* = r^+ \mid \lambda$ and $r^+ = rr^* = r^*r$
- ***Zero or one instance.*** The unary postfix operator ?, means “zero or one occurrence”. That is r? is equivalent to r | λ. The ? operator has the same precedence and associativity as * and +
- ***Character classes.*** A regular expression a1 | a2 | . . . | an, where ai’s are each symbols of the alphabet, can be replaced by the shorthand [a1a2 . . . an]. More importantly, when a1, a2, . . ., an form a logical sequence, e.g., consecutive uppercase letter, lowercase letters, or digits, we can replace them by a1-an, that is, just the first and the last separated by hyphen. Thus [abc] is shorthand for a | b | c, and [a-z] is shorthand for a | b | . . . | z

# Specifying RE's in Unix Tools

- Single characters        a b c d \x
- Alternation        [bcd]  [b-z]  ab|cd
- Any character        .        *(period)*
- Match sequence of characters x* y+
- Concatenation        abc[d-q]
- Optional RE        [0-9]+(\.[0-9]*)?

Example

letter → A|B|C|…|Z|a|b|c|…|z

digit → 0|1|…|9

identifier → letter (letter|digit)*

# Specifying (cont’d)

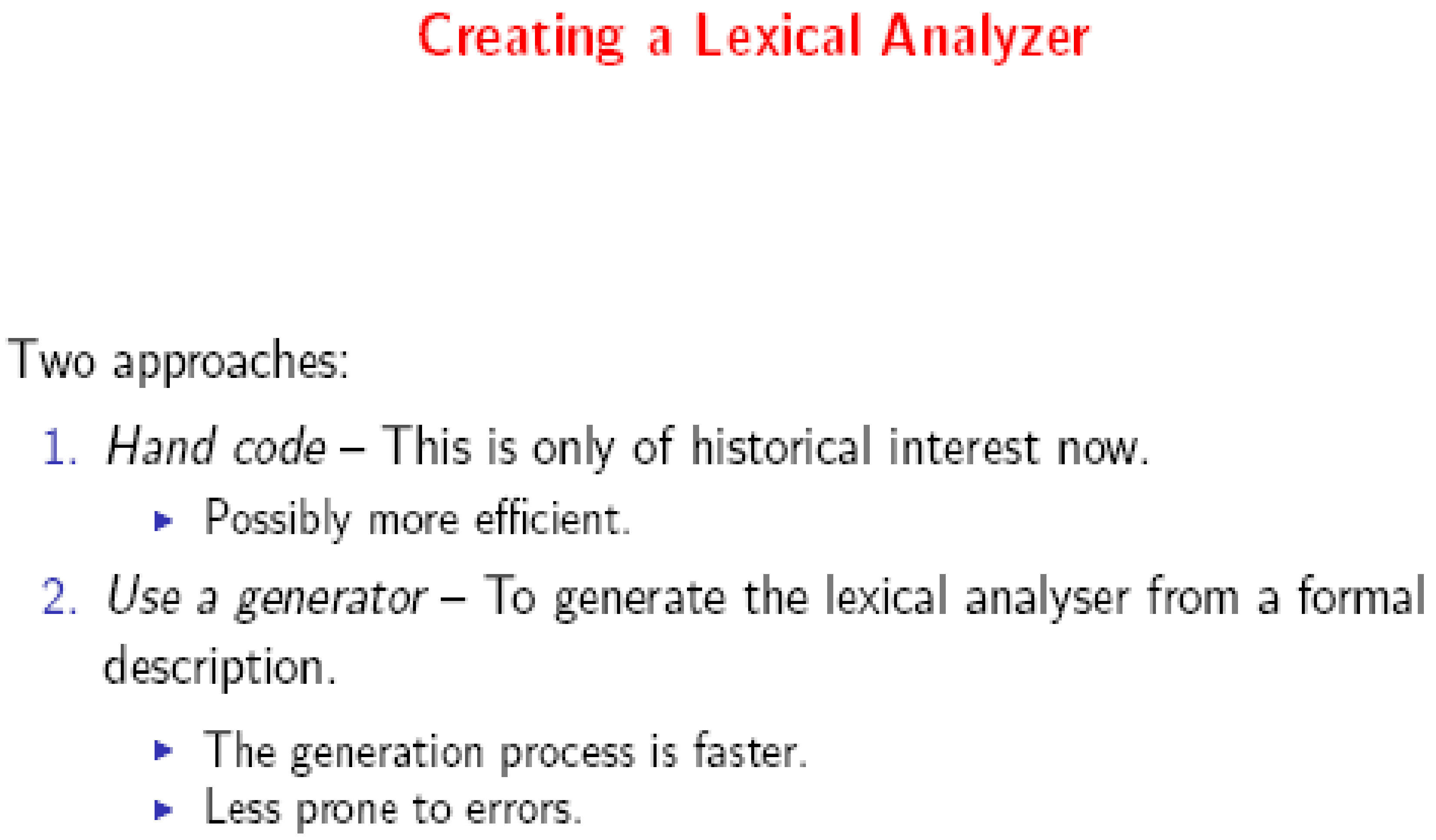
- Lex provides shortcuts for describing regular expressions in a compact manner.

  Example

  [a-z] stands for a|b|c|…|z

  [0-9] stands for 0|1|…|9

  [abc] stands for a|b|c



# Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
   - Possibly more efficient.
2. *Use a generator* – To generate the lexical analyser from a formal description.
   - The generation process is faster.
   - Less prone to errors.

# Automatic Generation of Lexical Analysers

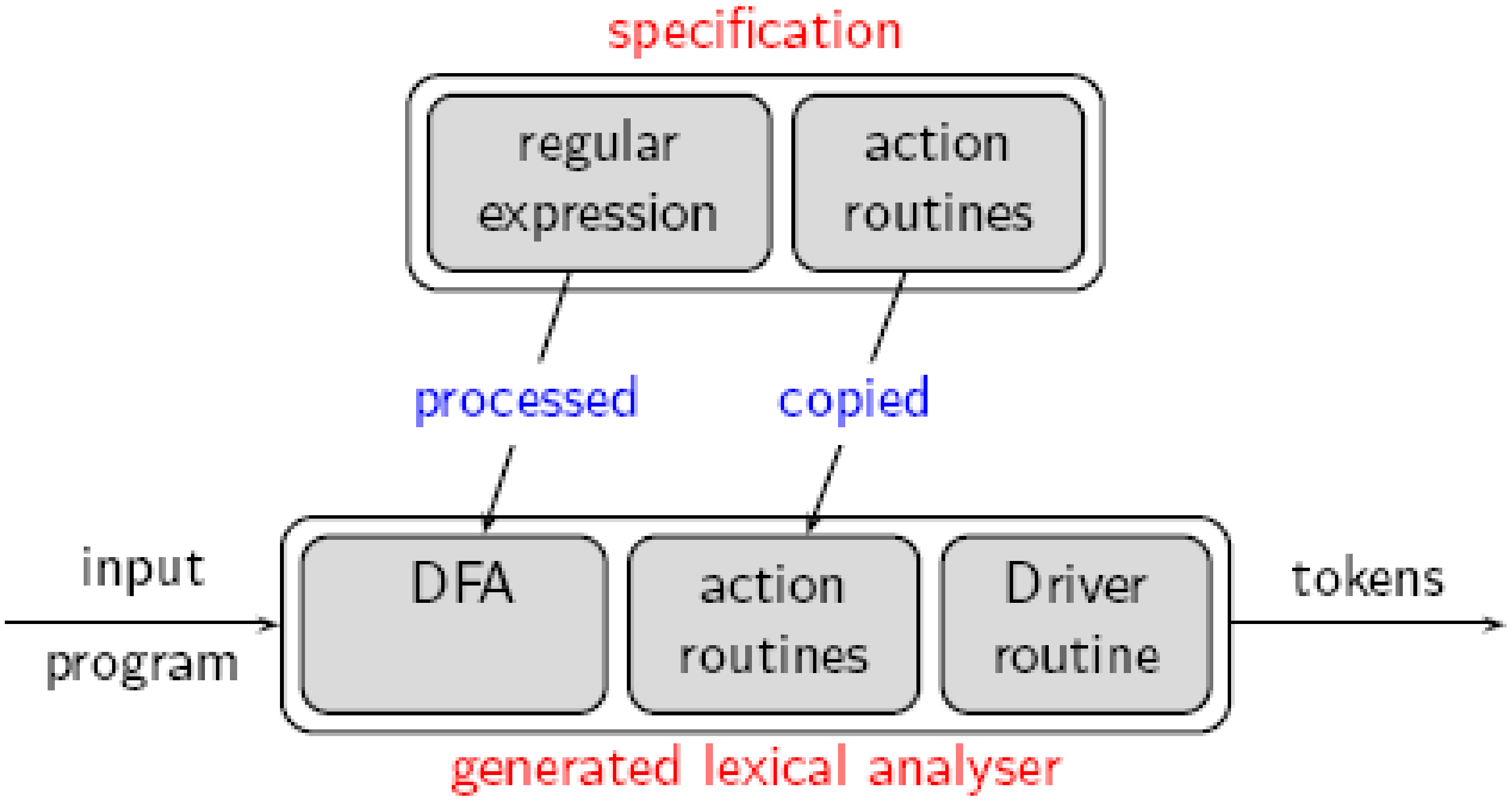
Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, consisting of:
  - a regular expression describing each token, and
  - a code fragment describing the action to be performed, on identifying each token.

The generated lexical analyser consists of:

- A *deterministic finite automaton (DFA)* constructed from the token specification.
- A *code fragment* (a driver routine) which can traverse *any DFA*.
- Code for the *action specifications*.

# Automatic Generation of Lexical Analysers

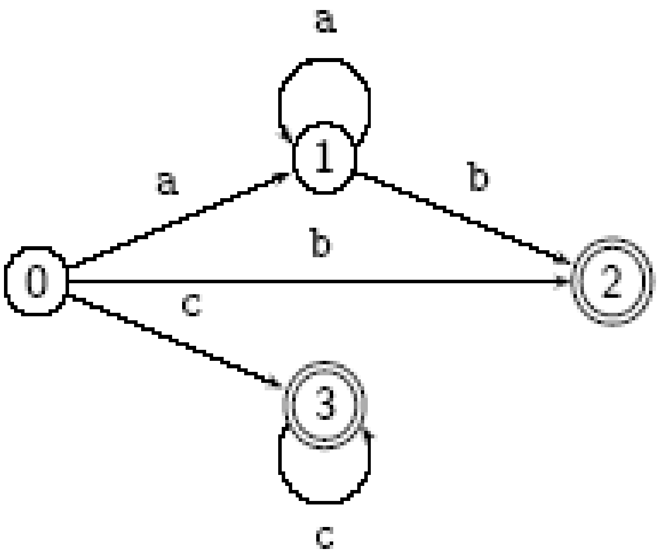


# Example of Lexical Analyser Generation

Suppose a language has two tokens

| Pattern | Action |
|---|---|
| `a*b` | `{ printf( "Token 1 found");}` |
| `c+` | `{ printf( "Token 2 found");}` |

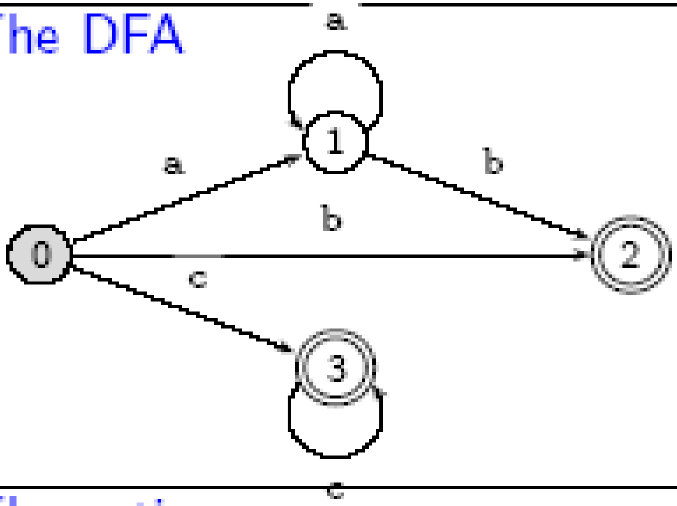
From the description, construct a structure called a deterministic finite automaton (DFA).



# Example of Lexical Analyser Generation

Now consider the following together:

## The DFA



## The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

## The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

## The input and output

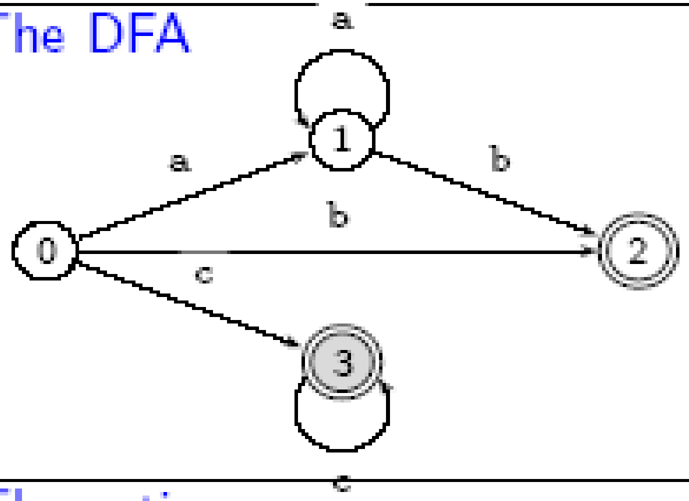
Input: aabadbcc↩

Output:

# Example of Lexical Analyser Generation

Now consider the following together:

## The DFA



## The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

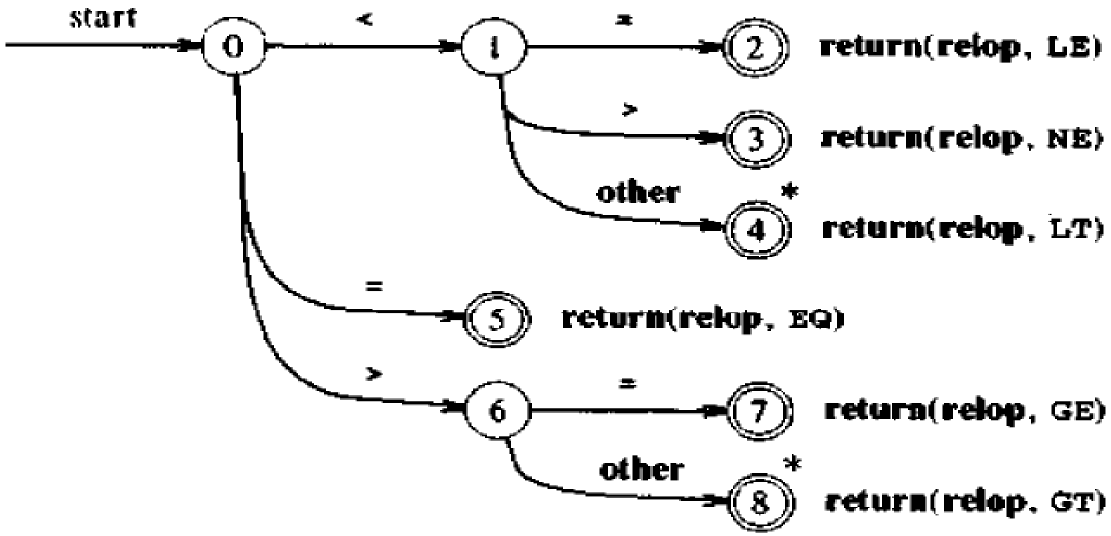
## The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
      break;}
  3:{printf("Token 2 found");
      break;}
}
```

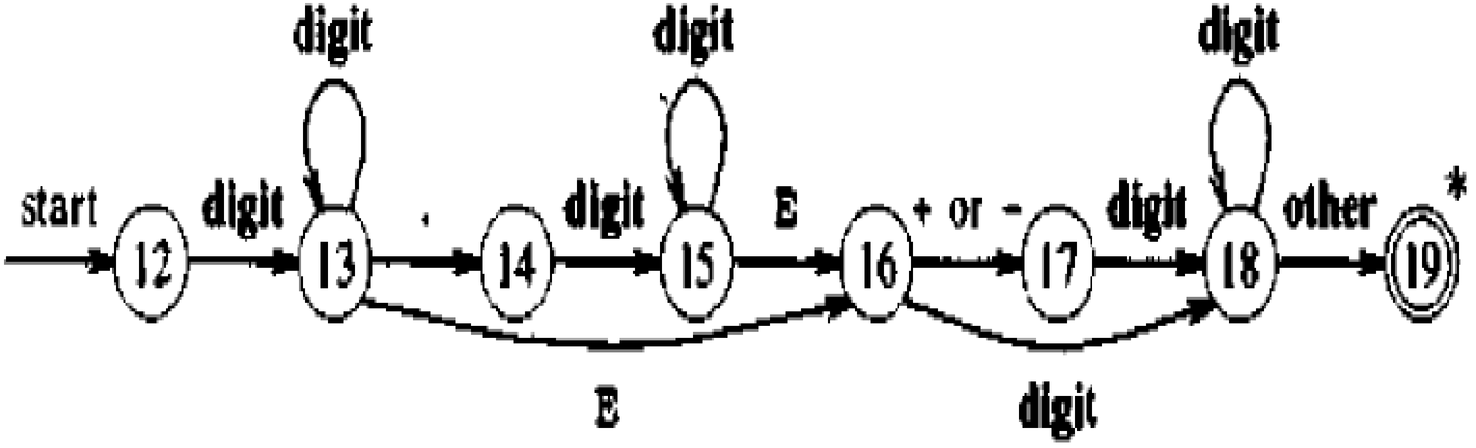
## The input and output

Input: aabadbcc↩

Output:
```
Token 1 found
Token 1 found
Token 2 found
```



- Fig:Transition diagram for relop



- Fig:diagram for unsigned numbers in Pascal

# Exercise

- All Strings that start with “ab” and end with “ba”
- All Strings in Which {1,2,3} exist in ascending order