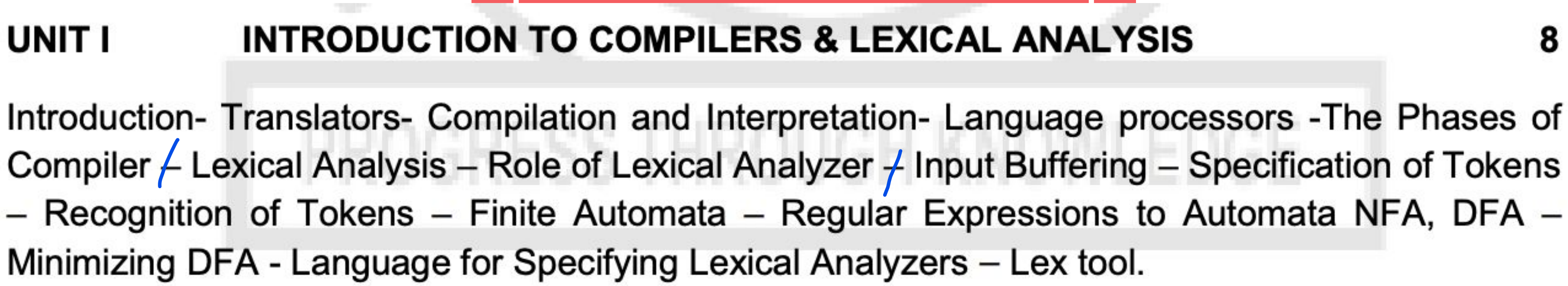
LEXICAL ANALYSER



UNIT I INTRODUCTION TO COMPILERS & LEXICAL ANALYSIS 8

Introduction- Translators- Compilation and Interpretation- Language processors -The Phases of Compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Finite Automata – Regular Expressions to Automata NFA, DFA – Minimizing DFA - Language for Specifying Lexical Analyzers – Lex tool.

PROGRESS THROUGH KNOWLEDGE

ROLE OF LEXICAL ANALYSER

- Lexical Analysis Vs Parsing
- Tokens, Patterns and Lexemes
- Attributes for tokens
- Lexical Errors

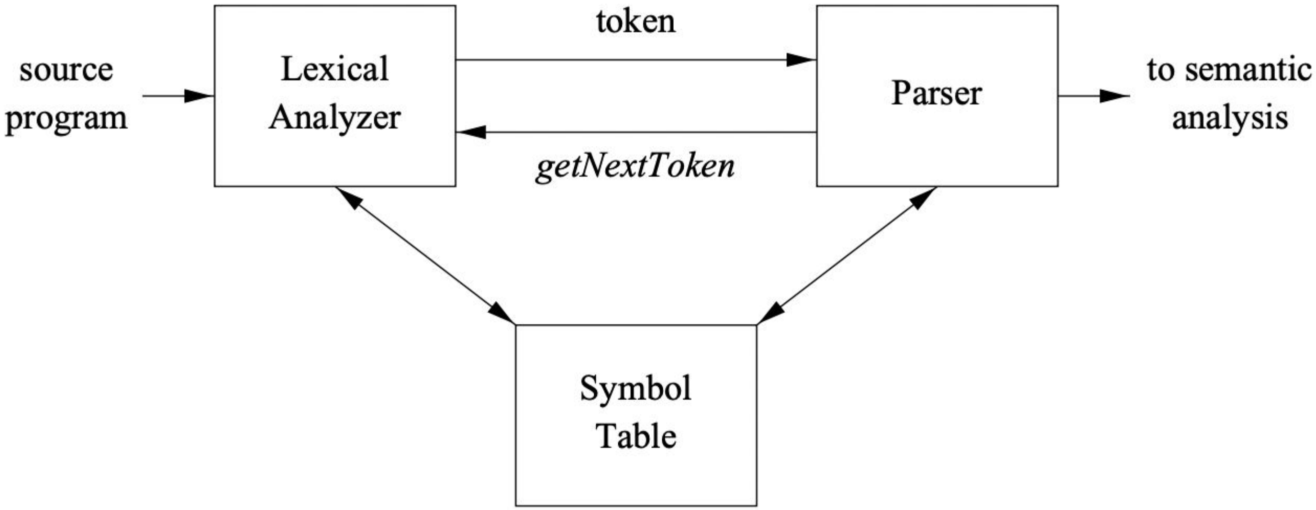


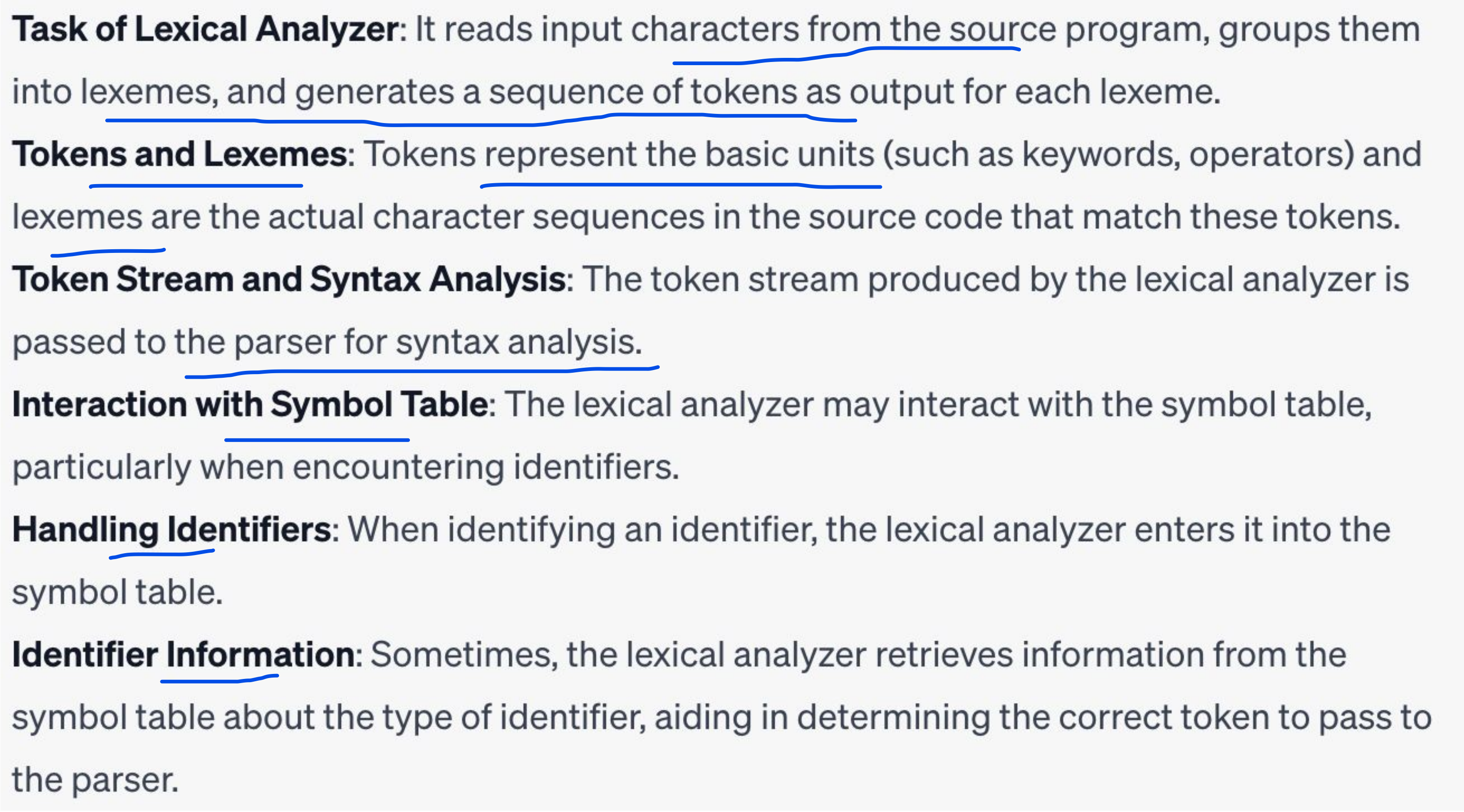
Figure 3.1: Interactions between the lexical analyzer and the parser

Role of Lexical Analyser

int a = 5 —

=

≡ a = a+5 ✓



Task of Lexical Analyzer: It reads input characters from the source program, groups them into lexemes, and generates a sequence of tokens as output for each lexeme.

Tokens and Lexemes: Tokens represent the basic units (such as keywords, operators) and lexemes are the actual character sequences in the source code that match these tokens.

Token Stream and Syntax Analysis: The token stream produced by the lexical analyzer is passed to the parser for syntax analysis.

Interaction with Symbol Table: The lexical analyzer may interact with the symbol table, particularly when encountering identifiers.

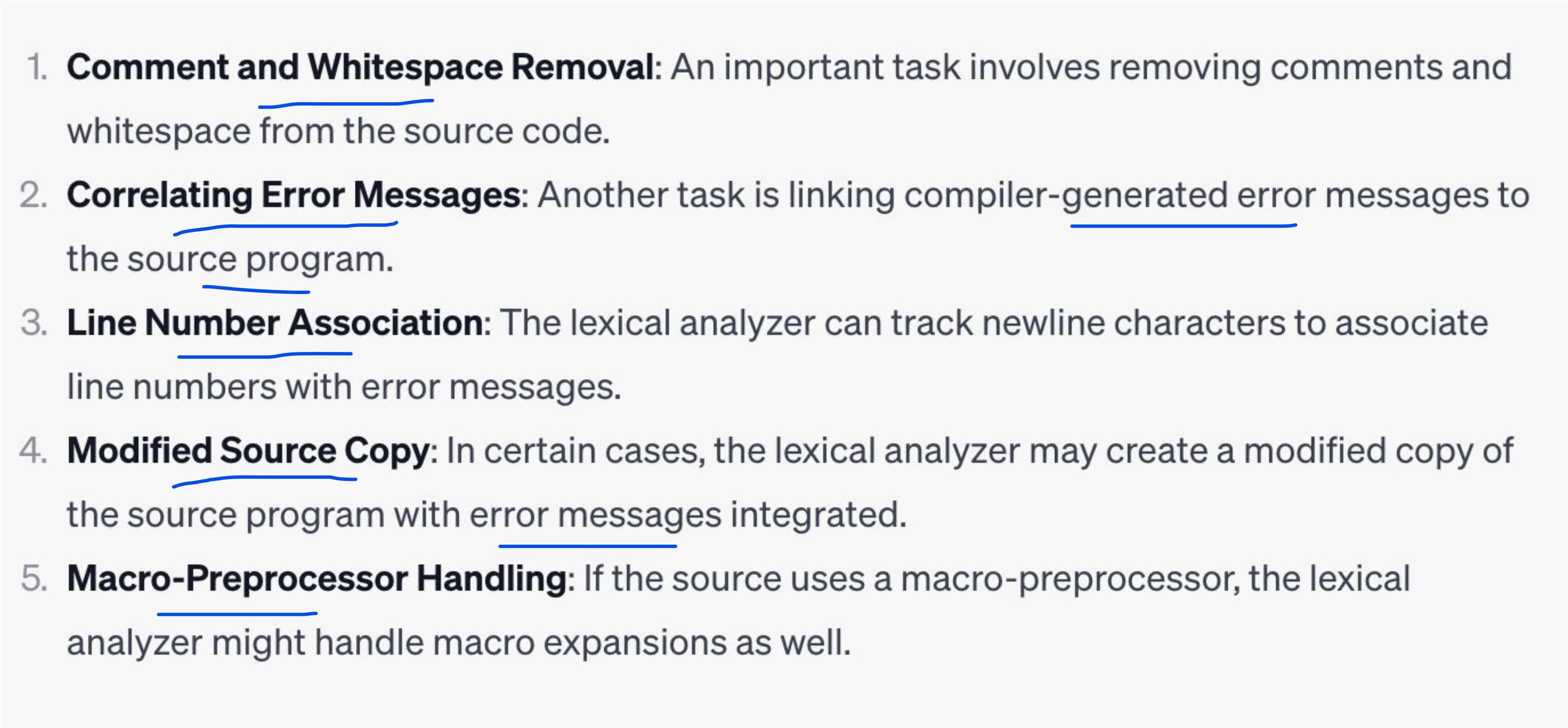
Handling Identifiers: When identifying an identifier, the lexical analyzer enters it into the symbol table.

Identifier Information: Sometimes, the lexical analyzer retrieves information from the symbol table about the type of identifier, aiding in determining the correct token to pass to the parser.

Other Tasks

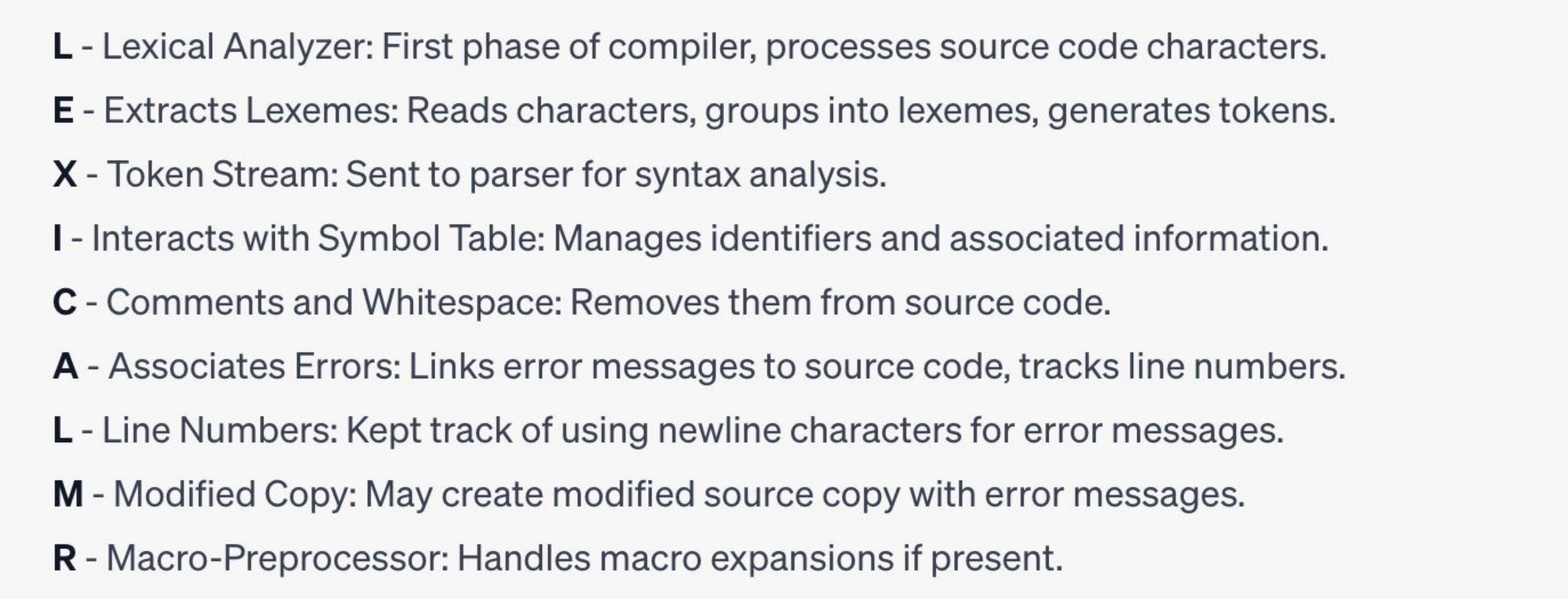
1) - - - \n
\n
\n

Syntax error in line no 45



1. **Comment and Whitespace Removal**: An important task involves removing comments and whitespace from the source code.
2. **Correlating Error Messages**: Another task is linking compiler-generated error messages to the source program.
3. **Line Number Association**: The lexical analyzer can track newline characters to associate line numbers with error messages.
4. **Modified Source Copy**: In certain cases, the lexical analyzer may create a modified copy of the source program with error messages integrated.
5. **Macro-Preprocessor Handling**: If the source uses a macro-preprocessor, the lexical analyzer might handle macro expansions as well.

Short Form



L - Lexical Analyzer: First phase of compiler, processes source code characters.

E - Extracts Lexemes: Reads characters, groups into lexemes, generates tokens.

X - Token Stream: Sent to parser for syntax analysis.

I - Interacts with Symbol Table: Manages identifiers and associated information.

C - Comments and Whitespace: Removes them from source code.

A - Associates Errors: Links error messages to source code, tracks line numbers.

L - Line Numbers: Kept track of using newline characters for error messages.

M - Modified Copy: May create modified source copy with error messages.

R - Macro-Preprocessor: Handles macro expansions if present.

1. Lexical Analysis vs Parsing

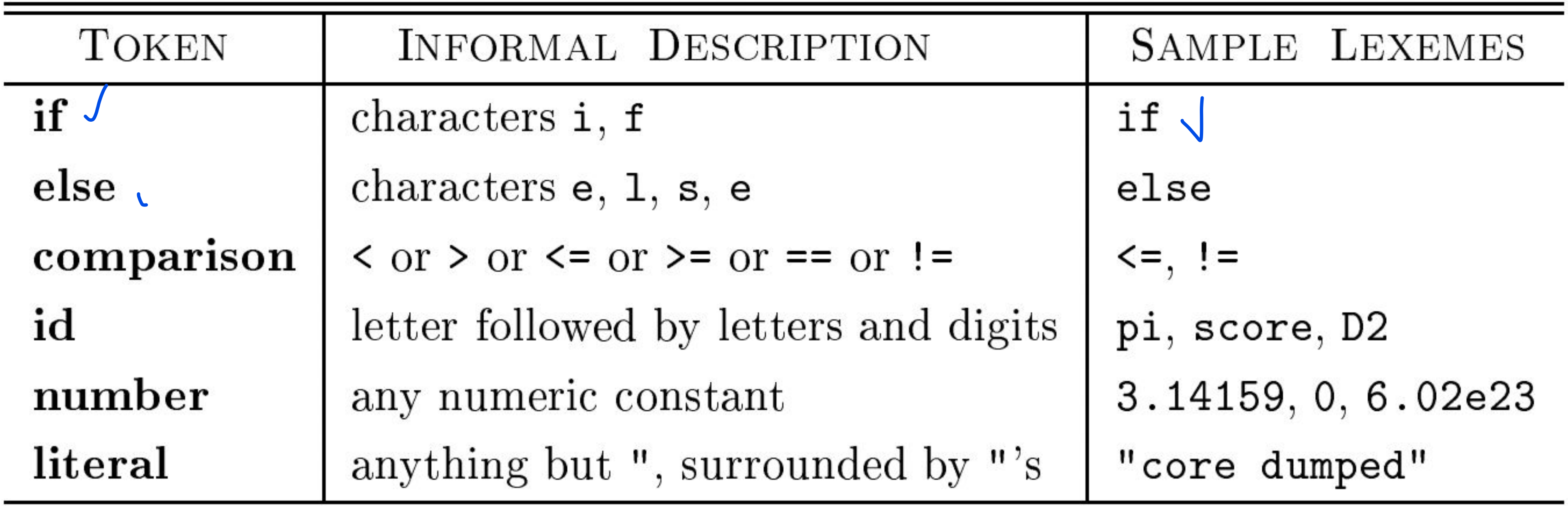
1. **Simplicity of design** ✓
   a. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
   b. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
2. **Compiler efficiency is improved.**
   a. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
   b. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. **Compiler portability is enhanced.**
   a. Input-device-specific peculiarities can be restricted to the lexical analyzer.

2. Tokens, Patterns and Lexemes

→ symbol table

- **Token** Var ⇒ $\langle id, 1000 \rangle$ $\langle id_1, 4000 \rangle$
    - 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.
- **Pattern** xyz, _xy, 1xyX
    - 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.
- **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.

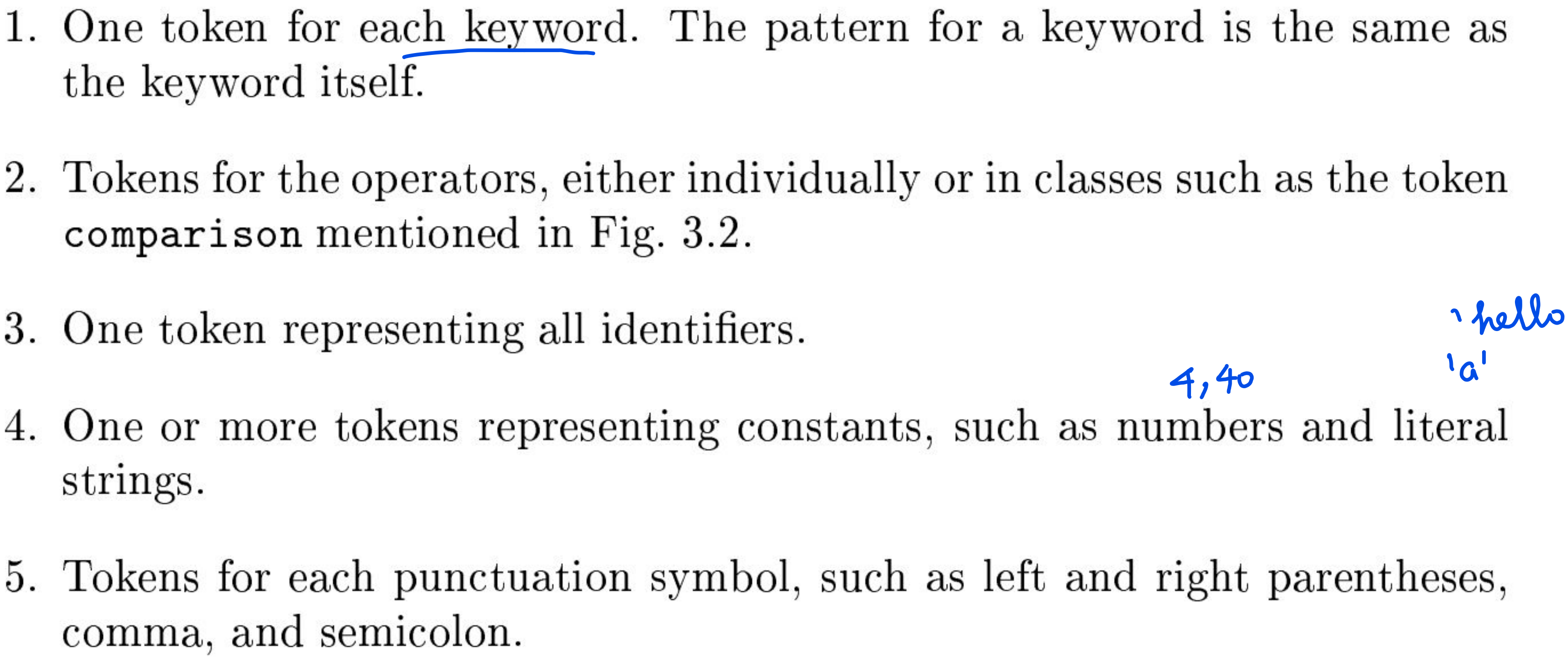
count



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"`

Figure 3.2: Examples of tokens

Classes of 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 in Fig. 3.2.

3. One token representing all identifiers.

'hello'

4,40 'a'

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.

3. Attributes for Tokens - Why value?

<id, value>
↳ Why?

Example 3.2 : The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

Variable
E, M, C

<**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>

ST

E	
M	

4. Lexical Errors

if (a==4):

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:

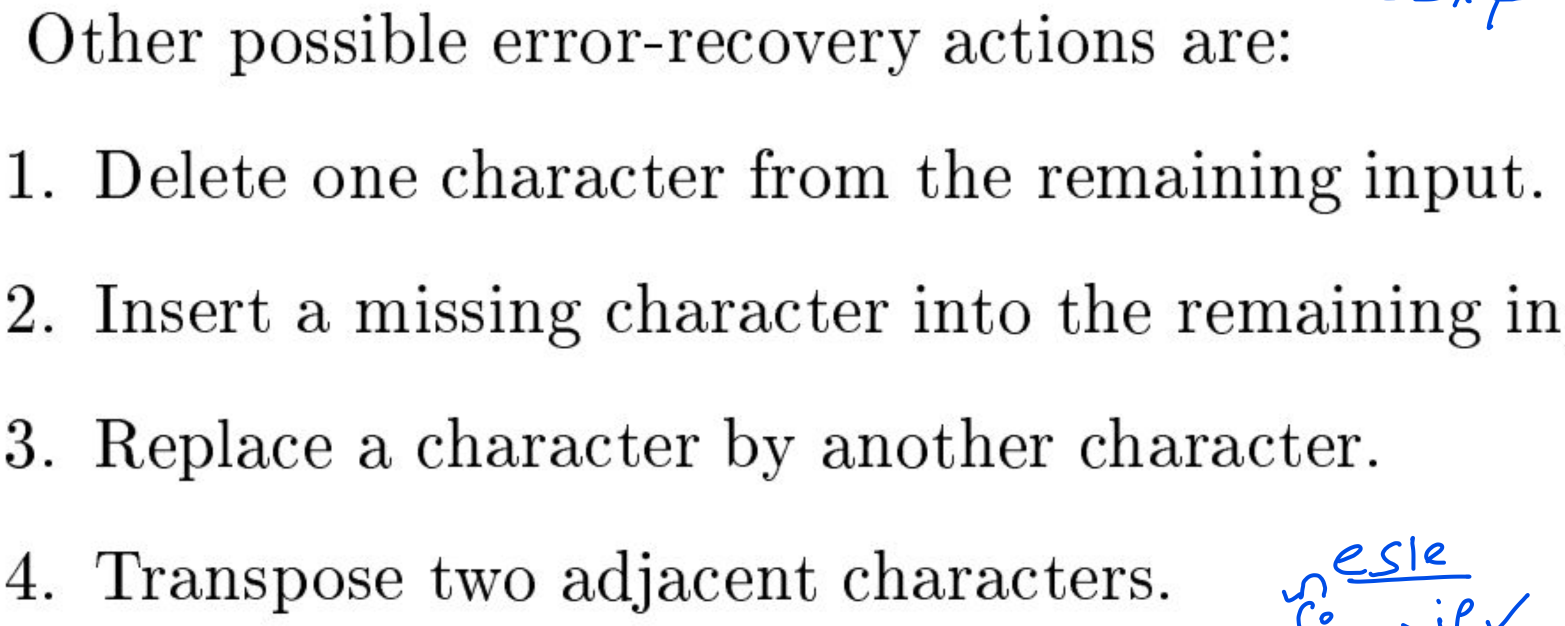
fi (a==4)

```
fi ( a == f(x)) ...
```

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.

Panic mode recovery. We delete successive characters from the remaining input, until the lexical analyzer can ~~find~~ find a well-formed token at the beginning of what input is left

else → esl̸e̸



Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input
3. Replace a character by another character.
4. Transpose two adjacent characters.

esle

fi → if ✓

THANK YOU