[](http://www.tutorialspoint.com/index.htm)

Top of Form



Bottom of Form

* [Quizzes](http://www.tutorialspoint.com/questions_and_answers.htm)
* [Shared](http://www.tutorialspoint.com/shared-tutorials.php)
* [Articles](http://www.tutorialspoint.com/articles/index.php)
* [**HOME**](http://www.tutorialspoint.com/index.htm)
* [**TUTORIALS LIBRARY**](javascript:void(0);)
* [**CODING GROUND**](javascript:void(0);)
* [**DEV TOOLS**](http://www.tutorialspoint.com/online_dev_tools.htm)



* Compiler Design Tutorial
* [Compiler Design - Home](http://www.tutorialspoint.com/compiler_design/index.htm)
* [Compiler Design - Overview](http://www.tutorialspoint.com/compiler_design/compiler_design_overview.htm)
* [Compiler Design - Architecture](http://www.tutorialspoint.com/compiler_design/compiler_design_architecture.htm)
* [Compiler Design - Phases of Compiler](http://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm)
* [Compiler Design - Lexical Analysis](http://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm)
* [Compiler - Regular Expressions](http://www.tutorialspoint.com/compiler_design/compiler_design_regular_expressions.htm)
* [Compiler Design - Finite Automata](http://www.tutorialspoint.com/compiler_design/compiler_design_finite_automata.htm)
* [Compiler Design - Syntax Analysis](http://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm)
* [Compiler Design - Types of Parsing](http://www.tutorialspoint.com/compiler_design/compiler_design_types_of_parsing.htm)
* [Compiler Design - Top-Down Parser](http://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm)
* [Compiler Design - Bottom-Up Parser](http://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm)
* [Compiler Design - Error Recovery](http://www.tutorialspoint.com/compiler_design/compiler_design_error_recovery.htm)
* [Compiler Design - Semantic Analysis](http://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm)
* [Compiler - Run-time Environment](http://www.tutorialspoint.com/compiler_design/compiler_design_runtime_environment.htm)
* [Compiler Design - Symbol Table](http://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm)
* [Compiler - Intermediate Code](http://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm)
* [Compiler Design - Code Generation](http://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm)
* [Compiler Design - Code Optimization](http://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm)
* Compiler Design Useful Resources
* [Compiler Design - Quick Guide](http://www.tutorialspoint.com/compiler_design/compiler_design_quick_guide.htm)
* [Compiler Design - Useful Resources](http://www.tutorialspoint.com/compiler_design/compiler_design_useful_resources.htm)
* [Compiler Design - Discussion](http://www.tutorialspoint.com/compiler_design/compiler_design_discussion.htm)
* Selected Reading
* [Developer's Best Practices](http://www.tutorialspoint.com/developers_best_practices/index.htm)
* [Questions and Answers](http://www.tutorialspoint.com/questions_and_answers.htm)
* [Effective Resume Writing](http://www.tutorialspoint.com/effective_resume_writing.htm)
* [HR Interview Questions](http://www.tutorialspoint.com/hr_interview_questions/index.htm)
* [Computer Glossary](http://www.tutorialspoint.com/computer_glossary.htm)
* [Who is Who](http://www.tutorialspoint.com/computer_whoiswho.htm)

Compiler Design - Lexical Analysis

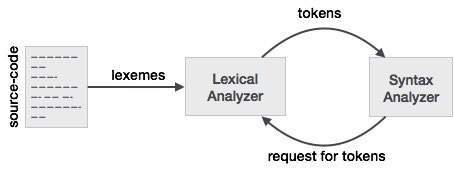
Advertisements

[Previous Page](http://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm)

[Next Page](http://www.tutorialspoint.com/compiler_design/compiler_design_regular_expressions.htm)

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

|  |  |
| --- | --- |
| Arithmetic Symbols | Addition(+), Subtraction(-), Modulo(%), Multiplication(\*), Division(/) |
| Punctuation | Comma(,), Semicolon(;), Dot(.), Arrow(->) |
| Assignment | = |
| Special Assignment | +=, /=, \*=, -= |
| Comparison | ==, !=, <, <=, >, >= |
| Preprocessor | # |
| Location Specifier | & |
| Logical | &, &&, |, ||, ! |
| Shift Operator | >>, >>>, <<, <<< |

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

**For example:**

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

[Previous Page](http://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm)

[Print](http://www.tutorialspoint.com/cgi-bin/printpage.cgi)

[PDF](http://www.tutorialspoint.com/compiler_design/pdf/compiler_design_lexical_analysis.pdf)

[Next Page](http://www.tutorialspoint.com/compiler_design/compiler_design_regular_expressions.htm)

Advertisements

[[img](javascript:void(0)) [img](javascript:void(0)) [img](javascript:void(0)) [img](javascript:void(0)) [img](javascript:void(0)) img](javascript:void(0))

[](http://www.tutorialspoint.com/index.htm)

* [Write for us](http://www.tutorialspoint.com/about/tutorials_writing.htm)

* [FAQ's](http://www.tutorialspoint.com/about/faq.htm)

* [Helping](http://www.tutorialspoint.com/about/about_helping.htm)

* [Contact](http://www.tutorialspoint.com/about/contact_us.htm)

© Copyright 2015. All Rights Reserved.

 go

Applications:

A lexer forms the first phase of a [compiler frontend](https://en.wikipedia.org/wiki/Compiler_frontend) in modern processing,[[a]](https://en.wikipedia.org/wiki/Lexical_analysis#cite_note-2) and is generally done in a single pass.

Lexers and parsers are most often used for compilers, but can be used for other computer language tools, such as [prettyprinters](https://en.wikipedia.org/wiki/Prettyprint" \o "Prettyprint) or [linters](https://en.wikipedia.org/wiki/Lint_(software)). Lexing itself can be divided into two stages: the *scanning,* which segments the input sequence into groups and categorizes these into token classes; and the *evaluating,* which converts the raw input characters into a processed value.

Lexers are generally quite simple, with most of the complexity deferred to the parser or [semantic analysis](https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)) phases, and can often be generated by a [lexer generator](https://en.wikipedia.org/wiki/Lexical_analysis" \l "Lexer_generator), notably [lex](https://en.wikipedia.org/wiki/Lex_(software)" \o "Lex (software)) or derivatives. However, lexers can sometimes include some complexity, such as [phrase structure](https://en.wikipedia.org/wiki/Lexical_analysis#Phrase_structure) processing to make input easier and simplify the parser, and may be written partially or completely by hand, either to support additional features or for performance.