WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Čeština
Dansk
Deutsch
Ελληνικά
Español
فارسی
Français
한국어
Հայերեն
Hrvatski
Bahasa Indonesia
Italiano
עברית
Македонски
Nederlands
日本語
Polski
Português
Русский
Svenska
தமிழ்
Українська
Tiếng Việt
中文
Edit links

Lexical analysis

From Wikipedia, the free encyclopedia

In computer science, **lexical analysis** is the process of converting a sequence of characters (such as a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a **lexer**, **tokenizer**,[1] or **scanner** (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.

Contents [hide]

Applications [edit]

A lexer forms the first phase of a compiler frontend in modern processing,[a] 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 or linters. 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 phases, and can often be generated by a lexer generator, notably lex or derivatives. However, lexers can sometimes include some complexity, such as 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.

Lexeme [edit]

A *lexeme* is a string of characters which forms a syntactic unit.[2]

Some authors (for example, [1], [2]), just call this a *token*, using 'token' interchangeably to represent (a) the string being tokenized, and also (b) the token datastructure resulting from putting this string through the tokenization process.

Note that the usage of the word 'lexeme' in computer science is different from the meaning of the word 'lexeme' in linguistics. A lexeme in computer science roughly corresponds to what in linguistics might be called a word (in computer science, 'word' has a different meaning than the meaning of 'word' in linguistics), although in some cases it may be more similar to a morpheme.

Token [edit]

A **token** is a structure representing a lexeme that explicitly indicates its categorization for the purpose of parsing.[3] A category of tokens is what in linguistics might be called a part-of-speech. Examples of token categories may include "identifier" and "integer literal", although the set of token categories differ in different programming languages. The process of forming tokens from an input stream of characters is called **tokenization**. Consider this expression in the C programming language:

```
sum = 3 + 2;
```

Tokenized and represented by the following table:

Lexeme	Token category
sum	"Identifier"
=	"Assignment operator"
3	"Integer literal"
+	"Addition operator"
2	"Integer literal"
;	"End of statement"

Lexical grammar [edit]

Main article: Lexical grammar

The specification of a programming language often includes a set of rules, the lexical grammar, which defines the lexical syntax. The lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions; they define the set of possible character sequences that are used to form individual tokens or lexemes. A lexer recognizes strings, and for each kind of string found the lexical program takes an action, most simply producing a token.

Two important common lexical categories are white space and comments. These are also defined in the grammar and processed by the lexer, but may be discarded (not producing any tokens) and considered *non-significant,* at most separating two tokens (as in `if x` instead of `ifx`). There are two important exceptions to this. Firstly, in off-side rule languages that delimit blocks with indentation, initial whitespace is significant, as it determines block structure, and is generally handled at the lexer level; see phrase structure, below. Secondly, in some uses of lexers, comments and whitespace must be preserved – for examples, a prettyprinter also needs to output the comments and some debugging tools may provide messages to the programmer showing the original source code. In the 1960s, notably for ALGOL, whitespace and comments were eliminated as part of the line reconstruction phase (the initial phase of the compiler frontend), but this separate phase has been eliminated and these are now handled by the lexer.

Tokenization [edit]

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

'Tokenization' has a different meaning within the field of computer security.

Take, for example,

```
The quick brown fox jumps over the lazy dog
```

The string isn't implicitly segmented on spaces, as an English speaker would do. The raw input, the 43 characters, must be explicitly split into the 9 tokens with a given space delimiter (i.e. matching the string `" "` or regular expression `/\s{1}/`).

The tokens could be represented in XML,

```xml
<sentence>
  <word>The</word>
  <word>quick</word>
  <word>brown</word>
  <word>fox</word>
  <word>jumps</word>
  <word>over</word>
  <word>the</word>
  <word>lazy</word>
  <word>dog</word>
</sentence>
```

Or an s-expression,

```
 (sentence
   (word The)
   (word quick)
   (word brown)
   (word fox)
   (word jumps)
   (word over)
   (word the)
   (word lazy)
   (word dog))
```

When a token class represents more than one possible lexeme, the lexer often saves enough information to reproduce the original lexeme, so that it can be used in semantic analysis. The parser typically retrieves this information from the lexer and stores it in the abstract syntax tree. This is necessary in order to avoid information loss in the case of numbers and identifiers.

Tokens are identified based on the specific rules of the lexer. Some methods used to identify tokens include: regular expressions, specific sequences of characters known as a flag, specific separating characters called delimiters, and explicit definition by a dictionary. Special characters, including punctuation characters, are commonly used by lexers to identify tokens because of their natural use in written and programming languages.

Tokens are often categorized by character content or by context within the data stream. Categories are defined by the rules of the lexer. Categories often involve grammar elements of the language used in the data stream. Programming languages often categorize tokens as identifiers, operators, grouping symbols, or by data type. Written languages commonly categorize tokens as nouns, verbs, adjectives, or punctuation. Categories are used for post-processing of the tokens either by the parser or by other functions in the program.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

When a lexer feeds tokens to the parser, the representation used is typically an enumerated list of number representations. For example "Identifier" is represented with 0, "Assignment operator" with 1, "Addition operator" with 2, etc.

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing". If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

Scanner [edit]

The first stage, the **scanner**, is usually based on a finite-state machine (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes). For instance, an *integer* token may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule, or longest match rule). In some languages, the lexeme creation rules are more complicated and may involve backtracking over previously read characters. For example, in C, a single 'L' character is not enough to distinguish between an identifier that begins with 'L' and a wide-character string literal.

Evaluator [edit]

A lexeme, however, is only a string of characters known to be of a certain kind (e.g., a string literal, a sequence of letters). In order to construct a token, the lexical analyzer needs a second stage, the **evaluator**, which goes over the characters of the lexeme to produce a *value*. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing: only the type is needed. Similarly, sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments. The evaluators for identifiers are usually simple (literally representing the identifier), but may include some unstropping. The evaluators for integer literals may pass the string on (deferring evaluation to the semantic analysis phase), or may perform evaluation themselves, which can be involved for different bases or floating point numbers. For a simple quoted string literal, the evaluator only needs to remove the quotes, but the evaluator for an escaped string literal itself incorporates a lexer, which unescapes the escape sequences.

For example, in the source code of a computer program, the string

`net_worth_future = (assets - liabilities);`

might be converted into the following lexical token stream; note that whitespace is suppressed and special characters have no value:

```
NAME net_worth_future
EQUALS
OPEN_PARENTHESIS
NAME assets
MINUS
NAME liabilities
CLOSE_PARENTHESIS
SEMICOLON
```

Though it is possible and sometimes necessary, due to licensing restrictions of existing parsers or if the list of tokens is small, to write a lexer by hand, lexers are often generated by automated tools. These tools generally accept regular expressions that describe the tokens allowed in the input stream. Each regular expression is associated with a production rule in the lexical grammar of the programming language that evaluates the lexemes matching the regular expression. These tools may generate source code that can be compiled and executed or construct a state table for a finite-state machine (which is plugged into template code for compilation and execution).

Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, for an English-based language, a NAME token might be any English alphabetical character or an underscore, followed by any number of instances of ASCII alphanumeric characters and/or underscores. This could be represented compactly by the string `[a-zA-Z_][a-zA-Z_0-9]*`. This means "any character a-z, A-Z or _, followed by 0 or more of a-z, A-Z, _ or 0-9".

Regular expressions and the finite-state machines they generate are not powerful enough to handle recursive patterns, such as "*n* opening parentheses, followed by a statement, followed by *n* closing parentheses." They are not capable of keeping count, and verifying that *n* is the same on both sides — unless you have a finite set of permissible values for *n*. It takes a full-fledged parser to recognize such patterns in their full generality. A parser can push parentheses on a stack and then try to pop them off and see if the stack is empty at the end. (see example[4] in the SICP book).

The Lex programming tool and its compiler is designed to generate code for fast lexical analysers based on a formal description of the lexical syntax. It is not generally considered sufficient for applications with a complicated set of lexical rules and severe performance requirements; for instance, the GNU Compiler Collection (gcc) uses hand-written lexers.

Lexer generator [edit]

See also: Parser generator

Lexers are often generated by a **lexer generator**, analogous to parser generators, and such tools often come together. The most established is lex, paired with the yacc parser generator, and the free equivalents flex/bison. These generators are a form of domain-specific language, taking in a lexical specification – generally regular expressions with some markup – and outputting a lexer.

These tools yield very fast development, which is particularly important in early development, both to get a working lexer and because the language specification may be changing frequently. Further, they often provide advanced features, such as pre- and post-conditions which are hard to program by hand. However, automatically generated lexer may lack flexibility, and thus may require some manual modification or a completely manually written lexer.

Lexer performance is a concern, and optimization of the lexer is worthwhile, particularly in stable languages where the lexer is run very frequently (such as C or HTML). lex/flex-generated lexers are reasonably fast, but improvements of two to three times are possible using more tuned generators. Hand-written lexers are sometimes used, but modern lexer generators produce faster lexers than most hand-coded ones. The lex/flex family of generators uses a table-driven approach which is much less efficient than the directly coded approach.[*dubious – discuss*] With the latter approach the generator produces an engine that directly jumps to follow-up states via goto statements. Tools like re2c[5] and Quex have proven to produce engines that are between two to three times faster than flex produced engines.[*citation needed*] It is in general difficult to hand-write analyzers that perform better than engines generated by these latter tools.

List of lexer generators [edit]

See also: List of parser generators

- ANTLR - Can generate lexical analyzers and parsers.
- DFASTAR - Generates DFA matrix table-driven lexers in C++.
- Flex - Alternative variant of the classic "lex" (C/C++).
- Ragel - A state machine and lexer generator with output in C, C++, C#, Objective-C, D, Java, Go and Ruby.

The following lexical analysers can handle Unicode:

- JavaCC - JavaCC generates lexical analyzers written in Java.
- JFLex - A lexical analyzer generator for Java.
- Quex - A fast universal lexical analyzer generator for C and C++.
- FsLex - A lexer generator for byte and Unicode character input for F#

Phrase structure [edit]

Lexical analysis primarily segments the input stream of characters into tokens, simply grouping the characters into pieces and categorizing them. However, the lexing may be significantly more complex; most simply, lexers may omit tokens or insert additional tokens. Omitting tokens, notably whitespace and comments, is very common, when these are not needed by the compiler. Less commonly, additional tokens may be inserted. This is primarily done to group tokens into statements, or statements into blocks, to simplify the parser.

Line continuation [edit]

Line continuation is a feature of some languages where a newline is normally a statement terminator. Most frequently, ending a line with a backslash (immediately followed by a newline) results in the line being *continued* – the following line is *joined* to the previous line. This is generally done in the lexer: the backslash and newline are discarded, rather than the newline being tokenized. Examples include bash,[6] other shell scripts and Python.[7]

Semicolon insertion [edit]

Many languages use the semicolon as a statement terminator. Most often this is mandatory, but in some languages the semicolon is optional in many contexts. This is primarily done at the lexer level, where the lexer outputs a semicolon into the token stream, despite one not being present in the input character stream, and is known as **semicolon insertion** or **automatic semicolon insertion**. In these cases semicolons are part of the formal phrase grammar of the language, but may not be found in input text, as they can be inserted by the lexer. Note that optional semicolons or other terminators or separators are also sometimes handled at the parser level, notably in the case of trailing commas or semicolons.

Semicolon insertion is a feature of BCPL and its distant descendent Go,[8] though it is not present in B or C.[9] Semicolon insertion is present in JavaScript, though the rules are somewhat complicated and much-criticized; to avoid bugs, some recommend always using semicolons, while others use initial semicolons, known as defensive semicolons, at the start of potentially ambiguous statements.

Semicolon insertion (in languages with semicolon-terminated statements) and line continuation (in languages with newline-terminated statements) can be seen as complementary: semicolon insertion adds a token, even though newlines generally do *not* generate tokens, while line continuation prevents a token from being generated, even though newlines generally *do* generate tokens.

Off-side rule [edit]

Further information: Off-side rule

The off-side rule (blocks determined by indentation) can be implemented in the lexer, as in Python, where increasing the indentation results in the lexer outputting an INDENT token, and decreasing the indentation results in the lexer outputting a DEDENT token.[10] These tokens correspond to the opening brace `{` and closing brace `}` in languages that use braces for blocks, and means that the phrase grammar does not depend on whether braces or indentation are used. This requires that the lexer hold state, namely the current indentation level, and thus can detect changes in indentation when this changes, and thus the lexical grammar is not context-free – INDENT/DEDENT depend on the contextual information of previous indentation level.

Context-sensitive lexing [edit]

Generally lexical grammars are context-free or almost context-free, and do not require any looking back, looking ahead, or backtracking, which allows a simple, clean, and efficient implementation. This also allows simple one-way communication from the lexer to the parser, without needing any information flowing back to the lexer.

There are exceptions, however. Simple examples include: semicolon insertion in Go, which requires looking back one token; concatenation of consecutive string literals in Python,[11] which requires holding one token in a buffer before outputting it (to see if the next token is another string literal); and the off-side rule in Python, which requires maintaining a count of indentation level (indeed, a stack of each indentation level). These examples all only require lexical context, and while they complicate the lexer some, they are invisible to the parser and later phases.

A more complicated example is the lexer hack in C, where the token class of a sequence of characters cannot be determined until the semantic analysis phase, since typedef names and variable names are lexically identical but constitute different token classes – thus in the lexer hack, the lexer calls the semantic analyzer (say, symbol table) and checks if the sequence requires a typedef name. In this case, information has to flow back not simply from the parser, but from the semantic analyzer back to the lexer, which complicates the design.

Notes [edit]

a. ^ In older languages such as ALGOL, the initial stage was instead line reconstruction, which performed unstropping and removed whitespace and comments (and in fact had scannerless parsers, without a separate lexer). These steps are now done as part of the lexer.

References [edit]

1. ^ www.cs.man.ac.uk
2. ^ "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." – page 111, "Compilers Principles, Techniques, & Tools, 2nd Ed." (WorldCat) by Aho, Lam, Sethi and Ullman, as quoted in http://stackoverflow.com/questions/14954721/what-is-the-difference-between-token-and-lexeme
3. ^ "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 sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes." — page 111, "Compilers Principles, Techniques, & Tools, 2nd Ed." (WorldCat) by Aho, Lam, Sethi and Ullman, as quoted in http://stackoverflow.com/questions/14954721/what-is-the-difference-between-token-and-lexeme
4. ^ mitpress.mit.edu
5. ^ Bumbulis, P.; Cowan, D. D. (Mar–Dec 1993). "RE2C: A more versatile scanner generator". *ACM Letters on Programming Languages and Systems* **2** (1–4): 70–84. doi:10.1145/176454.176487.
6. ^ *Bash Reference Manual*, 3.1.2.1 Escape Character
7. ^ Python Documentation, 2. Lexical analysis: 2.1.5. Explicit line joining
8. ^ *Effective Go*, "Semicolons"
9. ^ "Semicolons in Go", golang-nuts, Rob 'Commander' Pike, 12/10/09
10. ^ Python Documentation, 2. Lexical analysis: 2.1.8. Indentation
11. ^ Python Documentation, 2. Lexical analysis: 2.4.2. String literal concatenation

- *Compiling with C# and Java*, Pat Terry, 2005, ISBN 032126360X
- *Algorithms + Data Structures = Programs*, Niklaus Wirth, 1975, ISBN 0-13-022418-9
- *Compiler Construction*, Niklaus Wirth, 1996, ISBN 0-201-40353-6
- Sebesta, R. W. (2006). Concepts of programming languages (Seventh edition) pp. 177. Boston: Pearson/Addison-Wesley.

External links [edit]

- Yang, W.; Tsay, Chey-Woei; Chan, Jien-Tsai (2002). "On the applicability of the longest-match rule in lexical analysis". *Computer Languages, Systems and Structures* (Elsevier Science) **28** (3): 273–288. doi:10.1016/S0096-0551(02)00014-0. NSC 86-2213-E-009-021 and NSC 86-2213-E-009-079.
- Trim, Craig (Jan 23, 2013). "The Art of Tokenization". *Developer Works*. IBM.

- Word Mention Segmentation Task, an analysis

Categories: Compiler construction | Interpreters (computing) | Programming language implementation | Parsing