# Introduction

Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term has slightly different meanings in different branches of linguistics and computer science.

Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate. Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process.

The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator.

Parsing is complementary to templating, which produces formatted output. These may be applied to different domains, but often appear together, such as the scanf/printf pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser.

An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer programming language to create some form of internal representation; the parser is a key step in the compiler frontend. Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers, the parsing itself can be done in one pass or multiple passes – see one-pass compiler and multi-pass compiler.

The implied disadvantages of a one-pass compiler can largely be overcome by adding fix-ups, where provision is made for fix-ups during the forward pass, and the fix-ups are applied backwards when the current program segment has been recognized as having been completed. An example where such a fix-up mechanism would be useful would be a forward GOTO statement, where the target of the GOTO is unknown until the program segment is completed. In this case, the application of the fix-up would be delayed until the target of the GOTO was recognized. Obviously, a backward GOTO does not require a fix-up.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out at the semantic analysis (contextual analysis) step.

## https://upload.wikimedia.org/wikipedia/commons/d/d6/Parser_Flow%D5%B8.gifOverview of the Process

The following example demonstrates the common case of parsing a computer language with two levels of grammar: lexical and syntactic. The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "11 \* (5 + 4)^2" and split it into the tokens 11, \*, (, 5, +, 4, ), ^, 2, each of which is a meaningful symbol in the context of an arithmetic expression. The lexer would contain rules to tell it that the characters \*, +, ^, ( and ) mark the start of a new token, so meaningless tokens like "12\*" or "(3" will not be generated.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars.

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator or interpreter, the action is to evaluate the expression or program, a compiler, on the other hand, would generate some kind of code. Attribute grammars can also be used to define these actions.

## Types of Parsers

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

1. Top-down parsing - Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.
2. Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

LL parsers and recursive-descent parser are examples of top-down parsers which cannot accommodate left recursive production rules. An important distinction with regard to parsers is whether a parser generates a leftmost derivation or a rightmost derivation (see context-free grammar). LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).