**Parsing**

From Wikipedia, the free encyclopedia

"Parse" redirects here. For other uses, see [Parse (disambiguation)](https://en.wikipedia.org/wiki/Parse_%28disambiguation%29).

"Parser" redirects here. For the computer programming language, see [Parser (CGI language)](https://en.wikipedia.org/wiki/Parser_%28CGI_language%29).

**Parsing** or **syntactic analysis** is the process of analysing a [string](https://en.wikipedia.org/wiki/String_%28computer_science%29) of [symbols](https://en.wikipedia.org/wiki/Symbol_%28programming%29), either in [natural language](https://en.wikipedia.org/wiki/Natural_language) or in [computer languages](https://en.wikipedia.org/wiki/Computer_languages), conforming to the rules of a [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar). The term *parsing* comes from Latin *pars* (*orationis*), meaning [part (of speech)](https://en.wikipedia.org/wiki/Part_of_speech).[[1]](https://en.wikipedia.org/wiki/Parsing#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Parsing#cite_note-dictionary.com-2)

The term has slightly different meanings in different branches of [linguistics](https://en.wikipedia.org/wiki/Linguistics) and [computer science](https://en.wikipedia.org/wiki/Computer_science). Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence, sometimes with the aid of devices such as [sentence diagrams](https://en.wikipedia.org/wiki/Sentence_diagram). It usually emphasizes the importance of grammatical divisions such as [subject](https://en.wikipedia.org/wiki/Subject_%28grammar%29) and [predicate](https://en.wikipedia.org/wiki/Predicate_%28grammar%29).

Within [computational linguistics](https://en.wikipedia.org/wiki/Computational_linguistics) the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a [parse tree](https://en.wikipedia.org/wiki/Parse_tree) showing their syntactic relation to each other, which may also contain semantic and other information.

The term is also used in [psycholinguistics](https://en.wikipedia.org/wiki/Psycholinguistics) when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc."[[2]](https://en.wikipedia.org/wiki/Parsing#cite_note-dictionary.com-2) This term is especially common when discussing what linguistic cues help speakers to interpret [garden-path sentences](https://en.wikipedia.org/wiki/Garden_path_sentence).

Within computer science, the term is used in the analysis of [computer languages](https://en.wikipedia.org/wiki/Computer_languages), referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of [compilers](https://en.wikipedia.org/wiki/Compilers) and [interpreters](https://en.wikipedia.org/wiki/Interpreter_%28computing%29).

**Contents**

* [1 Human languages](https://en.wikipedia.org/wiki/Parsing#Human_languages)
  + [1.1 Traditional methods](https://en.wikipedia.org/wiki/Parsing#Traditional_methods)
  + [1.2 Computational methods](https://en.wikipedia.org/wiki/Parsing#Computational_methods)
  + [1.3 Psycholinguistics](https://en.wikipedia.org/wiki/Parsing#Psycholinguistics)
* [2 Computer languages](https://en.wikipedia.org/wiki/Parsing#Computer_languages)
  + [2.1 Parser](https://en.wikipedia.org/wiki/Parsing#Parser)
  + [2.2 Overview of process](https://en.wikipedia.org/wiki/Parsing#Overview_of_process)
* [3 Types of parsers](https://en.wikipedia.org/wiki/Parsing#Types_of_parsers)
* [4 Parser development software](https://en.wikipedia.org/wiki/Parsing#Parser_development_software)
* [5 Lookahead](https://en.wikipedia.org/wiki/Parsing#Lookahead)
* [6 See also](https://en.wikipedia.org/wiki/Parsing#See_also)
* [7 References](https://en.wikipedia.org/wiki/Parsing#References)
* [8 Further reading](https://en.wikipedia.org/wiki/Parsing#Further_reading)
* [9 External links](https://en.wikipedia.org/wiki/Parsing#External_links)

**Human languages**

Main category: [Natural language parsing](https://en.wikipedia.org/wiki/Category:Natural_language_parsing)

**Traditional methods**

The traditional grammatical exercise of parsing, sometimes known as *clause analysis*, involves breaking down a text into its component parts of speech with an explanation of the form, function, and syntactic relationship of each part.[[3]](https://en.wikipedia.org/wiki/Parsing#cite_note-3) This is determined in large part from study of the language's [conjugations](https://en.wikipedia.org/wiki/Conjugation_%28grammar%29) and [declensions](https://en.wikipedia.org/wiki/Declensions), which can be quite intricate for heavily inflected languages. To parse a phrase such as 'man bites dog' involves noting that the singular noun 'man' is the subject of the sentence, the verb 'bites' is the third person singular of the present tense of the verb 'to bite', and the singular noun 'dog' is the object of the sentence. Techniques such as [sentence diagrams](https://en.wikipedia.org/wiki/Sentence_diagram) are sometimes used to indicate relation between elements in the sentence.

Parsing was formerly central to the teaching of grammar throughout the English-speaking world, and widely regarded as basic to the use and understanding of written language. However the teaching of such techniques is no longer current.

**Computational methods**

|  |  |
| --- | --- |
| [https://upload.wikimedia.org/wikipedia/en/thumb/9/99/Question_book-new.svg/50px-Question_book-new.svg.png](https://en.wikipedia.org/wiki/File:Question_book-new.svg) | This section **does not** [**cite**](https://en.wikipedia.org/wiki/Wikipedia:Citing_sources) **any** [**sources**](https://en.wikipedia.org/wiki/Wikipedia:Verifiability). Please help improve this section by [adding citations to reliable sources](https://en.wikipedia.org/wiki/Help:Introduction_to_referencing_with_Wiki_Markup/1). Unsourced material may be challenged and [removed](https://en.wikipedia.org/wiki/Wikipedia:Verifiability#Burden_of_evidence). *(February 2013)* |

Main category: [Natural language parsing](https://en.wikipedia.org/wiki/Category:Natural_language_parsing)

In some [machine translation](https://en.wikipedia.org/wiki/Machine_translation) and [natural language processing](https://en.wikipedia.org/wiki/Natural_language_processing) systems, written texts in human languages are parsed by computer programs[[*clarification needed*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)]. Human sentences are not easily parsed by programs, as there is substantial [ambiguity](https://en.wikipedia.org/wiki/Syntactic_ambiguity) in the structure of human language, whose usage is to convey meaning (or [semantics](https://en.wikipedia.org/wiki/Semantics)) amongst a potentially unlimited range of possibilities but only some of which are germane to the particular case. So an utterance "Man bites dog" versus "Dog bites man" is definite on one detail but in another language might appear as "Man dog bites" with a reliance on the larger context to distinguish between those two possibilities, if indeed that difference was of concern. It is difficult to prepare formal rules to describe informal behaviour even though it is clear that some rules are being followed.

In order to parse natural language data, researchers must first agree on the [grammar](https://en.wikipedia.org/wiki/Grammar) to be used. The choice of syntax is affected by both [linguistic](https://en.wikipedia.org/wiki/Language) and computational concerns; for instance some parsing systems use [lexical functional grammar](https://en.wikipedia.org/wiki/Lexical_functional_grammar), but in general, parsing for grammars of this type is known to be [NP-complete](https://en.wikipedia.org/wiki/NP-complete). [Head-driven phrase structure grammar](https://en.wikipedia.org/wiki/Head-driven_phrase_structure_grammar) is another linguistic formalism which has been popular in the parsing community, but other research efforts have focused on less complex formalisms such as the one used in the Penn [Treebank](https://en.wikipedia.org/wiki/Treebank). [Shallow parsing](https://en.wikipedia.org/wiki/Shallow_parsing) aims to find only the boundaries of major constituents such as noun phrases. Another popular strategy for avoiding linguistic controversy is [dependency grammar](https://en.wikipedia.org/wiki/Dependency_grammar) parsing.

Most modern parsers are at least partly [statistical](https://en.wikipedia.org/wiki/Statistics); that is, they rely on a corpus of training data which has already been annotated (parsed by hand). This approach allows the system to gather information about the frequency with which various constructions occur in specific contexts. *(See* [*machine learning*](https://en.wikipedia.org/wiki/Machine_learning)*.)* Approaches which have been used include straightforward [PCFGs](https://en.wikipedia.org/wiki/PCFG) (probabilistic context-free grammars), [maximum entropy](https://en.wikipedia.org/wiki/Maximum_entropy_classifier), and [neural nets](https://en.wikipedia.org/wiki/Neural_net). Most of the more successful systems use *lexical* statistics (that is, they consider the identities of the words involved, as well as their [part of speech](https://en.wikipedia.org/wiki/Part_of_speech)). However such systems are vulnerable to [overfitting](https://en.wikipedia.org/wiki/Overfitting) and require some kind of [smoothing](https://en.wikipedia.org/wiki/Smoothing) to be effective.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Parsing algorithms for natural language cannot rely on the grammar having 'nice' properties as with manually designed grammars for programming languages. As mentioned earlier some grammar formalisms are very difficult to parse computationally; in general, even if the desired structure is not [context-free](https://en.wikipedia.org/wiki/Context-free), some kind of context-free approximation to the grammar is used to perform a first pass. Algorithms which use context-free grammars often rely on some variant of the [CYK algorithm](https://en.wikipedia.org/wiki/CYK_algorithm), usually with some [heuristic](https://en.wikipedia.org/wiki/Heuristic_%28computer_science%29) to prune away unlikely analyses to save time. *(See* [*chart parsing*](https://en.wikipedia.org/wiki/Chart_parsing)*.)* However some systems trade speed for accuracy using, e.g., linear-time versions of the [shift-reduce](https://en.wikipedia.org/wiki/Shift-reduce_parsing) algorithm. A somewhat recent development has been [parse reranking](https://en.wikipedia.org/w/index.php?title=Parse_reranking&action=edit&redlink=1) in which the parser proposes some large number of analyses, and a more complex system selects the best option.

**Psycholinguistics**

In [psycholinguistics](https://en.wikipedia.org/wiki/Psycholinguistics), parsing involves not just the assignment of words to categories, but the evaluation of the meaning of a sentence according to the rules of syntax drawn by inferences made from each word in the sentence. This normally occurs as words are being heard or read. Consequently, psycholinguistic models of parsing are of necessity *incremental*, meaning that they build up an interpretation as the sentence is being processed, which is normally expressed in terms of a partial syntactic structure. Creation of initially wrong structures occurs when interpreting [garden path sentences](https://en.wikipedia.org/wiki/Garden_path_sentences).

**Computer languages**

|  |  |
| --- | --- |
| [https://upload.wikimedia.org/wikipedia/en/thumb/9/99/Question_book-new.svg/50px-Question_book-new.svg.png](https://en.wikipedia.org/wiki/File:Question_book-new.svg) | This section **does not** [**cite**](https://en.wikipedia.org/wiki/Wikipedia:Citing_sources) **any** [**sources**](https://en.wikipedia.org/wiki/Wikipedia:Verifiability). Please help improve this section by [adding citations to reliable sources](https://en.wikipedia.org/wiki/Help:Introduction_to_referencing_with_Wiki_Markup/1). Unsourced material may be challenged and [removed](https://en.wikipedia.org/wiki/Wikipedia:Verifiability#Burden_of_evidence). *(February 2013)* |

**Parser**

A **parser** is a software component that takes input data (frequently text) and builds a [data structure](https://en.wikipedia.org/wiki/Data_structure) – often some kind of [parse tree](https://en.wikipedia.org/wiki/Parse_tree), [abstract syntax tree](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Lexical_analysis), which creates tokens from the sequence of input characters; alternatively, these can be combined in [scannerless parsing](https://en.wikipedia.org/wiki/Scannerless_parsing). Parsers may be programmed by hand or may be automatically or semi-automatically generated by a [parser generator](https://en.wikipedia.org/wiki/Parser_generator). Parsing is complementary to [templating](https://en.wikipedia.org/wiki/Templating_language), which produces formatted *output.* These may be applied to different domains, but often appear together, such as the [scanf](https://en.wikipedia.org/wiki/Scanf)/[printf](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Scanf), to complex programs such as the frontend of a [C++ compiler](https://en.wikipedia.org/wiki/C%2B%2B_compiler) or the [HTML](https://en.wikipedia.org/wiki/HTML) parser of a [web browser](https://en.wikipedia.org/wiki/Web_browser). An important class of simple parsing is done using [regular expressions](https://en.wikipedia.org/wiki/Regular_expression), in which a regular expression defines a [regular language](https://en.wikipedia.org/wiki/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.

The use of parsers varies by input. In the case of data languages, a parser is often found as the file reading facility of a program, such as reading in HTML or [XML](https://en.wikipedia.org/wiki/XML) text; these examples are [markup languages](https://en.wikipedia.org/wiki/Markup_language). In the case of [programming languages](https://en.wikipedia.org/wiki/Programming_language), a parser is a component of a [compiler](https://en.wikipedia.org/wiki/Compiler) or [interpreter](https://en.wikipedia.org/wiki/Interpreter_%28computing%29), which parses the [source code](https://en.wikipedia.org/wiki/Source_code) of a [computer programming language](https://en.wikipedia.org/wiki/Computer_programming_language) to create some form of internal representation; the parser is a key step in the [compiler frontend](https://en.wikipedia.org/wiki/Compiler_frontend). Programming languages tend to be specified in terms of a [deterministic context-free grammar](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/One-pass_compiler) and [multi-pass compiler](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Semantic_analysis_%28compilers%29) (contextual analysis) step.

For example, in [Python](https://en.wikipedia.org/wiki/Python_%28programming_language%29) the following is syntactically valid code:

x = 1

print(x)

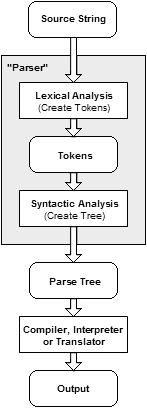
The following code, however, is syntactically valid in terms of the context-free grammar, yielding a syntax tree with the same structure as the previous, but is syntactically invalid in terms of the context-sensitive grammar, which requires that variables be initialized before use:

x = 1

print(y)

Rather than being analyzed at the parsing stage, this is caught by checking the *values* in the syntax tree, hence as part of *semantic* analysis: context-sensitive syntax is in practice often more easily analyzed as semantics.

**Overview of process**

[](https://en.wikipedia.org/wiki/File:Parser_Flow%D5%B8.gif)

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](https://en.wikipedia.org/wiki/Lexical_analysis), by which the input character stream is split into meaningful symbols defined by a grammar of [regular expressions](https://en.wikipedia.org/wiki/Regular_expression). For example, a calculator program would look at an input such as "12\*(3+4)^2" and split it into the tokens 12, \*, (, 3, +, 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Attribute_grammar).

The final phase is [semantic parsing](https://en.wikipedia.org/wiki/Semantic_analysis_%28computer_science%29) 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:

* [Top-down parsing](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Parse_tree) using a top-down expansion of the given [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar) rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate [ambiguity](https://en.wikipedia.org/wiki/Ambiguity) by expanding all alternative right-hand-sides of grammar rules.[[4]](https://en.wikipedia.org/wiki/Parsing#cite_note-AhoSethiUllman_1986-4)
* [Bottom-up parsing](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/LR_parser) are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

[LL parsers](https://en.wikipedia.org/wiki/LL_parser) and [recursive-descent parser](https://en.wikipedia.org/wiki/Recursive-descent_parser) are examples of top-down parsers which cannot accommodate [left recursive](https://en.wikipedia.org/wiki/Left_recursion) [production rules](https://en.wikipedia.org/wiki/Formal_grammar#The_syntax_of_grammars). Although it has been believed that simple implementations of top-down parsing cannot accommodate direct and indirect left-recursion and may require exponential time and space complexity while parsing ambiguous [context-free grammars](https://en.wikipedia.org/wiki/Context-free_grammar), more sophisticated algorithms for top-down parsing have been created by Frost, Hafiz, and Callaghan[[5]](https://en.wikipedia.org/wiki/Parsing#cite_note-FrostHafizCallaghan_2007-5)[[6]](https://en.wikipedia.org/wiki/Parsing#cite_note-FrostHafizCallaghan_2008-6) which accommodate [ambiguity](https://en.wikipedia.org/wiki/Ambiguity) and [left recursion](https://en.wikipedia.org/wiki/Left_recursion) in polynomial time and which generate polynomial-size representations of the potentially exponential number of parse trees. Their algorithm is able to produce both left-most and right-most derivations of an input with regard to a given [context-free grammar](https://en.wikipedia.org/wiki/Context-free_grammar).

An important distinction with regard to parsers is whether a parser generates a *leftmost derivation* or a *rightmost derivation* (see [context-free grammar](https://en.wikipedia.org/wiki/Context-free_grammar)). LL parsers will generate a leftmost [derivation](https://en.wikipedia.org/wiki/Parse_tree) and LR parsers will generate a rightmost derivation (although usually in reverse).[[4]](https://en.wikipedia.org/wiki/Parsing#cite_note-AhoSethiUllman_1986-4)

**Parser development software**

Some of the well known parser development tools include the following. Also see [comparison of parser generators](https://en.wikipedia.org/wiki/Comparison_of_parser_generators).

* [ANTLR](https://en.wikipedia.org/wiki/ANTLR)
* [Bison](https://en.wikipedia.org/wiki/GNU_bison)
* [Coco/R](https://en.wikipedia.org/wiki/Coco/R)
* [GOLD](https://en.wikipedia.org/wiki/GOLD_%28parser%29)
* [JavaCC](https://en.wikipedia.org/wiki/JavaCC)
* [Lemon](https://en.wikipedia.org/wiki/Lemon_Parser)
* [Lex](https://en.wikipedia.org/wiki/Lex_%28software%29)
* [LuZ](http://qyxz.netau.net)

* [Parboiled](https://en.wikipedia.org/wiki/Parboiled_%28Java%29)
* [Parsec](https://en.wikipedia.org/wiki/Parsec_%28parser%29)
* [Ragel](https://en.wikipedia.org/wiki/Ragel)
* [Spirit Parser Framework](https://en.wikipedia.org/wiki/Spirit_Parser_Framework)
* [Syntax Definition Formalism](https://en.wikipedia.org/wiki/Syntax_Definition_Formalism)
* [SYNTAX](https://en.wikipedia.org/wiki/SYNTAX)
* [XPL](https://en.wikipedia.org/wiki/XPL)
* [Yacc](https://en.wikipedia.org/wiki/Yacc)

**Lookahead**

|  |  |
| --- | --- |
| [https://upload.wikimedia.org/wikipedia/en/thumb/9/99/Question_book-new.svg/50px-Question_book-new.svg.png](https://en.wikipedia.org/wiki/File:Question_book-new.svg) | This section **does not** [**cite**](https://en.wikipedia.org/wiki/Wikipedia:Citing_sources) **any** [**sources**](https://en.wikipedia.org/wiki/Wikipedia:Verifiability). Please help improve this section by [adding citations to reliable sources](https://en.wikipedia.org/wiki/Help:Introduction_to_referencing_with_Wiki_Markup/1). Unsourced material may be challenged and [removed](https://en.wikipedia.org/wiki/Wikipedia:Verifiability#Burden_of_evidence). *(April 2012)* |

Lookahead establishes the maximum incoming tokens that a parser can use to decide which rule it should use. Lookahead is especially relevant to [LL](https://en.wikipedia.org/wiki/LL_parser), [LR](https://en.wikipedia.org/wiki/LR_parser), and [LALR parsers](https://en.wikipedia.org/wiki/LALR_parser), where it is often explicitly indicated by affixing the lookahead to the algorithm name in parentheses, such as LALR(1).

Most [programming languages](https://en.wikipedia.org/wiki/Programming_language), the primary target of parsers, are carefully defined in such a way that a parser with limited lookahead, typically one, can parse them, because parsers with limited lookahead are often more efficient. One important change[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] to this trend came in 1990 when [Terence Parr](https://en.wikipedia.org/wiki/Terence_Parr) created [ANTLR](https://en.wikipedia.org/wiki/ANTLR) for his Ph.D. thesis, a [parser generator](https://en.wikipedia.org/wiki/Parser_generator) for efficient LL(*k*) parsers, where *k* is any fixed value.

Parsers typically have only a few actions after seeing each token. They are shift (add this token to the stack for later reduction), reduce (pop tokens from the stack and form a syntactic construct), end, error (no known rule applies) or conflict (does not know whether to shift or reduce).

Lookahead has two advantages.

* It helps the parser take the correct action in case of conflicts. For example, parsing the if statement in the case of an else clause.
* It eliminates many duplicate states and eases the burden of an extra stack. A C language non-lookahead parser will have around 10,000 states. A lookahead parser will have around 300 states.

Example: Parsing the Expression 1 + 2 \* 3

|  |  |  |
| --- | --- | --- |
| Set of expression parsing rules (called grammar) is as follows, | | |
| Rule1: | E → E + E | Expression is the sum of two expressions. |
| Rule2: | E → E \* E | Expression is the product of two expressions. |
| Rule3: | E → number | Expression is a simple number |
| Rule4: | + has less precedence than \* | |

Most programming languages (except for a few such as APL and Smalltalk) and algebraic formulas give higher precedence to multiplication than addition, in which case the correct interpretation of the example above is (1 + (2\*3)). Note that Rule4 above is a semantic rule. It is possible to rewrite the grammar to incorporate this into the syntax. However, not all such rules can be translated into syntax.

Simple non-lookahead parser actions

Initially Input = [1,+,2,\*,3]

1. Shift "1" onto stack from input (in anticipation of rule3). Input = [+,2,\*,3] Stack = [1]
2. Reduces "1" to expression "E" based on rule3. Stack = [E]
3. Shift "+" onto stack from input (in anticipation of rule1). Input = [2,\*,3] Stack = [E,+]
4. Shift "2" onto stack from input (in anticipation of rule3). Input = [\*,3] Stack = [E,+,2]
5. Reduce stack element "2" to Expression "E" based on rule3. Stack = [E,+,E]
6. Reduce stack items [E,+] and new input "E" to "E" based on rule1. Stack = [E]
7. Shift "\*" onto stack from input (in anticipation of rule2). Input = [3] Stack = [E,\*]
8. Shift "3" onto stack from input (in anticipation of rule3). Input = [] (empty) Stack = [E,\*,3]
9. Reduce stack element "3" to expression "E" based on rule3. Stack = [E,\*,E]
10. Reduce stack items [E,\*] and new input "E" to "E" based on rule2. Stack = [E]

The parse tree and resulting code from it is not correct according to language semantics.

To correctly parse without lookahead, there are three solutions:

* The user has to enclose expressions within parentheses. This often is not a viable solution.
* The parser needs to have more logic to backtrack and retry whenever a rule is violated or not complete. The similar method is followed in LL parsers.
* Alternatively, the parser or grammar needs to have extra logic to delay reduction and reduce only when it is absolutely sure which rule to reduce first. This method is used in LR parsers. This correctly parses the expression but with many more states and increased stack depth.

Lookahead parser actions

1. Shift 1 onto stack on input 1 in anticipation of rule3. It does not reduce immediately.
2. Reduce stack item 1 to simple Expression on input + based on rule3. The lookahead is +, so we are on path to E +, so we can reduce the stack to E.
3. Shift + onto stack on input + in anticipation of rule1.
4. Shift 2 onto stack on input 2 in anticipation of rule3.
5. Reduce stack item 2 to Expression on input \* based on rule3. The lookahead \* expects only E before it.
6. Now stack has E + E and still the input is \*. It has two choices now, either to shift based on rule2 or reduction based on rule1. Since \* has more precedence than + based on rule4, so shift \* onto stack in anticipation of rule2.
7. Shift 3 onto stack on input 3 in anticipation of rule3.
8. Reduce stack item 3 to Expression after seeing end of input based on rule3.
9. Reduce stack items E \* E to E based on rule2.
10. Reduce stack items E + E to E based on rule1.

The parse tree generated is correct and simply more efficient[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] than non-lookahead parsers. This is the strategy followed in [LALR parsers](https://en.wikipedia.org/wiki/LALR_parser).