An **LL parser** is a [top-down](http://www.answers.com/topic/top-down-parsing) [parser](http://www.answers.com/topic/parsing) for a subset of the [context-free grammars](http://www.answers.com/topic/context-free-grammar). It parses the input from **L**eft to right, and constructs a [**L**eftmost derivation](http://www.answers.com/topic/context-free-grammar) of the sentence (hence LL, compared with [LR parser](http://www.answers.com/topic/lr-parser)). The class of grammars which are parsable in this way is known as the *LL grammars*.

The remainder of this article describes the table-based kind of parser, the alternative being a [recursive descent parser](http://www.answers.com/topic/recursive-descent-parser) which is usually coded by hand (although not always; see e.g. [ANTLR](http://www.answers.com/topic/antlr) for an LL(\*) recursive-descent parser generator).

An LL parser is called an LL(*k*) parser if it uses *k* [tokens](http://www.answers.com/topic/lexical-analysis) of [lookahead](http://www.answers.com/topic/lookahead) when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without [backtracking](http://www.answers.com/topic/backtracking) then it is called an LL(*k*) grammar. Of these grammars, LL(1) grammars, although fairly restrictive, are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions. Languages based on grammars with a high value of *k* require considerable effort to parse.

There is contention between the "European school" of language design, who prefer LL-based grammars, and the "US-school", who predominantly prefer LR-based grammars.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] This is largely due to teaching traditions and the detailed description of specific methods and tools in certain text books; another influence may be [Niklaus Wirth](http://www.answers.com/topic/niklaus-wirth) at [ETH Zürich](http://www.answers.com/topic/swiss-federal-institute-of-technology-z-rich) in Switzerland, whose research has described a number of ways of optimising LL(1) languages and compilers.

|  |
| --- |
|  |

**General case**

The parser works on strings from a particular [context-free grammar](http://www.answers.com/topic/context-free-grammar).

The parser consists of

* an *input buffer*, holding the input string (built from the grammar)
* a *stack* on which to store the [terminals](http://www.answers.com/topic/terminal-and-nonterminal-symbols) and [non-terminals](http://www.answers.com/topic/terminal-and-nonterminal-symbols) from the grammar yet to be parsed
* a *parsing table* which tells it what (if any) grammar rule to apply given the symbols on top of its stack and the next input token

The parser applies the rule found in the table by matching the top-most symbol on the stack (row) with the current symbol in the input stream (column).

When the parser starts, the stack already contains two symbols:

[ S, $ ]

where '$' is a special terminal to indicate the bottom of the stack and the end of the input stream, and 'S' is the start symbol of the grammar. The parser will attempt to rewrite the contents of this stack to what it sees on the input stream. However, it only keeps on the stack what still needs to be rewritten.

**Concrete example**

**Set up**

To explain its workings we will consider the following small grammar:

1. S → F
2. S → **(** S **+** F **)**
3. F → **a**

and parse the following input:

**( a + a )**

The parsing table for this grammar looks as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **(** | **)** | **a** | **+** | **$** |
| S | 2 | - | 1 | - | - |
| F | - | - | 3 | - | - |

(Note that there is also a column for the special terminal, represented here as **$**, that is used to indicate the end of the input stream.)

**Parsing procedure**

In each step, the parser reads the next-available symbol from the input stream, and the top-most symbol from the stack. If the input symbol and the stack-top symbol match, the parser discards them both, leaving only the unmatched symbols in the input stream and on the stack.

Thus, in its first step, the parser reads the input symbol '**('** and the stack-top symbol 'S'. The parsing table instruction comes from the column headed by the input symbol '**('**) and the row headed by the stack-top symbol 'S'; this cell contains '2', which instructs the parser to apply rule (2). The parser has to rewrite 'S' to '**(** S **+** F **)'** on the stack and write the rule number 2 to the output. The stack then becomes:

[ **(**, S, **+**, F, **)**, **$** ]

Since the '**('** from the input stream did not match the top-most symbol, 'S', from the stack, it was not removed, and remains the next-available input symbol for the following step.

In the second step, the parser removes the '**('** from its input stream and from its stack, since they match. The stack now becomes:

[ S, **+**, F, **)**, **$** ]

Now the parser has an '**a'** on its input stream and an 'S' as its stack top. The parsing table instructs it to apply rule (1) from the grammar and write the rule number 1 to the output stream. The stack becomes:

[ F, **+**, F, **)**, **$** ]

The parser now has an '**a'** on its input stream and an 'F' as its stack top. The parsing table instructs it to apply rule (3) from the grammar and write the rule number 3 to the output stream. The stack becomes:

[ **a**, **+**, F, **)**, **$** ]

In the next two steps the parser reads the '**a'** and '**+'** from the input stream and, since they match the next two items on the stack, also removes them from the stack. This results in:

[ F, **)**, **$** ]

In the next three steps the parser will replace '**F'** on the stack by '**a'**, write the rule number 3 to the output stream and remove the '**a'** and '**)'** from both the stack and the input stream. The parser thus ends with '**$'** on both its stack and its input stream.

In this case the parser will report that it has accepted the input string and write the following list of rule numbers to the output stream:

[ 2, 1, 3, 3 ]

This is indeed a list of rules for a [leftmost derivation](http://www.answers.com/topic/context-free-grammar) of the input string, which is:

S → **(** S **+** F **)** → **(** F **+** F **)** → **( a +** F **)** → **( a + a )**

**Remarks**

As can be seen from the example the parser performs three types of steps depending on whether the top of the stack is a nonterminal, a terminal or the special symbol **$**:

* If the top is a nonterminal then it looks up in the parsing table on the basis of this nonterminal and the symbol on the input stream which rule of the grammar it should use to replace it with on the stack. The number of the rule is written to the output stream. If the parsing table indicates that there is no such rule then it reports an error and stops.
* If the top is a terminal then it compares it to the symbol on the input stream and if they are equal they are both removed. If they are not equal the parser reports an error and stops.
* If the top is **$** and on the input stream there is also a **$** then the parser reports that it has successfully parsed the input, otherwise it reports an error. In both cases the parser will stop.

These steps are repeated until the parser stops, and then it will have either completely parsed the input and written a [leftmost derivation](http://www.answers.com/topic/context-free-grammar) to the output stream or it will have reported an error.

**Constructing an LL(1) parsing table**

In order to fill the parsing table, we have to establish what grammar rule the parser should choose if it sees a nonterminal *A* on the top of its stack and a symbol *a* on its input stream. It is easy to see that such a rule should be of the form *A* → *w* and that the language corresponding to *w* should have at least one string starting with *a*. For this purpose we define the *First-set* of *w*, written here as **Fi**(*w*), as the set of terminals that can be found at the start of any string in *w*, plus ε if the empty string also belongs to *w*. Given a grammar with the rules *A*1 → *w*1, ..., *An* → *wn*, we can compute the **Fi**(*wi*) and **Fi**(*Ai*) for every rule as follows:

1. initialize every **Fi**(*wi*) and **Fi**(*Ai*) with the empty set
2. add *Fi*(*wi*) to **Fi**(*Ai*) for every rule *Ai* → *w*i, where *Fi* is defined as follows:
   * *Fi*(*a* *w'* ) = { *a* } for every terminal *a*
   * *Fi*(*A* *w'* ) = **Fi**(*A*) for every nonterminal *A* with ε not in **Fi**(*A*)
   * *Fi*(*A* *w'* ) = **Fi**(*A*) \ { ε } ∪ *Fi*(*w'* ) for every nonterminal *A* with ε in **Fi**(*A*)
   * *Fi*(ε) = { ε }
3. add **Fi**(*wi*) to **Fi**(*A*i) for every rule *Ai* → *wi*
4. do steps 2 and 3 until all **Fi** sets stay the same.

Unfortunately, the First-sets are not sufficient to compute the parsing table. This is because a right-hand side *w* of a rule might ultimately be rewritten to the empty string. So the parser should also use the rule *A* → *w* if ε is in **Fi**(*w*) and it sees on the input stream a symbol that could follow *A*. Therefore we also need the *Follow-set* of *A*, written as **Fo**(*A*) here, which is defined as the set of terminals *a* such that there is a string of symbols *αAaβ* that can be derived from the start symbol. Computing the Follow-sets for the nonterminals in a grammar can be done as follows:

1. initialize every **Fo**(*Ai*) with the empty set
2. if there is a rule of the form *Aj* → *wAiw'* , then
   * if the terminal *a* is in *Fi*(*w'* ), then add *a* to **Fo**(*Ai*)
   * if ε is in *Fi*(*w'* ), then add **Fo**(*Aj*) to **Fo**(*Ai*)
3. repeat step 2 until all *Fo* sets stay the same.

Now we can define exactly which rules will be contained where in the parsing table. If *T*[*A*, *a*] denotes the entry in the table for nonterminal *A* and terminal *a*, then

*T*[*A*,*a*] contains the rule *A* → *w* if and only if

*a* is in **Fi**(*w*) or

ε is in **Fi**(*w*) and *a* is in **Fo**(*A*).

If the table contains at most one rule in every one of its cells, then the parser will always know which rule it has to use and can therefore parse strings without backtracking. It is in precisely this case that the grammar is called an *LL(1) grammar*.

**Constructing an LL(*k*) parsing table**

Until the mid 1990s, it was widely believed that LL(*k*) parsing (for *k* > 1) was impractical[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)], since the size of the [parse table](http://www.answers.com/topic/parsing-table) would (in general, in the worst case) have to have [exponential](http://www.answers.com/topic/exponential-function-1) [complexity](http://www.answers.com/topic/complexity) in *k*. This perception changed gradually after the release of the [PCCTS](http://www.answers.com/topic/antlr) around 1992, when it was demonstrated that many [programming languages](http://www.answers.com/topic/programming-language) can be parsed efficiently by an LL(*k*) parser without triggering the worst-case behavior of the parser. Moreover, in certain cases LL parsing is feasible even with unlimited lookahead. By contrast, traditional parser generators, like [yacc](http://www.answers.com/topic/yacc) use [LALR(1)](http://www.answers.com/topic/lalr-parser) parse tables to construct a restricted [LR parser](http://www.answers.com/topic/lr-parser) with a fixed one-token lookahead.

**Constructing an LL(*\**) parser**

See [ANTLR](http://www.answers.com/topic/antlr)

**Conflicts**

**LL(1) Conflicts**

There are 3 types of LL(1) conflicts:

* FIRST/FIRST conflict

The FIRST sets of two different non-terminals are overlapping.

* FIRST/FOLLOW conflict

The FIRST and FOLLOW set of a grammar rule overlap. With an epsilon in the FIRST set it is unknown which alternative to select. An example of an LL(1) conflict:

S -> A 'a' 'b'

A -> 'a' | epsilon

The FIRST set of A now is { 'a' epsilon } and the FOLLOW set { 'a' }.

* left-recursion

Left recursion will cause a FIRST/FIRST conflict with all alternatives.

E -> E '+' term | alt1 | alt2

**Solutions to LL(1) Conflicts**

* Left-factoring

A common left-factor is factored out like 3x + 3y = 3(x+y).

A -> X | X Y Z

becomes

A -> X ( Y Z )?

Can be applied when two alternatives start with the same symbol like a FIRST/FIRST conflict.

* Substitution

Substituting a rule into another rule to remove indirect or FIRST/FOLLOW conflicts. Note that this may cause a FIRST/FIRST conflict.

* Left recursion removal[[1]](http://www.answers.com/topic/ll-parser#cite_note-0)