# LL parser (From Wikipedia)

Top-down parser that parses input from left to right

In [computer science](https://en.wikipedia.org/wiki/Computer_science), an **LL parser** (Left-to-right, [leftmost derivation](https://en.wikipedia.org/wiki/Leftmost_derivation)) is a [top-down parser](https://en.wikipedia.org/wiki/Top-down_parsing) for a restricted [context-free language](https://en.wikipedia.org/wiki/Context-free_language). It parses the input from **L**eft to right, performing [**L**eftmost derivation](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees) of the sentence.

An LL parser is called an LL(*k*) parser if it uses *k* [tokens](https://en.wikipedia.org/wiki/Token_(parser)) of [lookahead](https://en.wikipedia.org/wiki/Parsing#Lookahead) when parsing a sentence. A grammar is called an [LL(*k*) grammar](https://en.wikipedia.org/wiki/LL_grammar) if an LL(*k*) parser can be constructed from it. A formal language is called an LL(*k*) language if it has an LL(*k*) grammar. The set of LL(*k*) languages is properly contained in that of LL(*k*+1) languages, for each *k* ≥ 0.[[1]](https://en.wikipedia.org/wiki/LL_parser#cite_note-1) A corollary of this is that not all context-free languages can be recognized by an LL(*k*) parser.

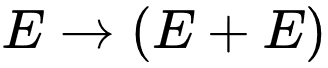
An LL parser is called LL-regular (LLR) if it parses an [LL-regular language](https://en.wikipedia.org/wiki/LL-regular_language).[[*clarification needed*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)][[2]](https://en.wikipedia.org/wiki/LL_parser#cite_note-2)[[3]](https://en.wikipedia.org/wiki/LL_parser#cite_note-3)[[4]](https://en.wikipedia.org/wiki/LL_parser#cite_note-4) The class of [LLR grammars](https://en.wikipedia.org/wiki/LL-regular_grammar) contains every LL(k) grammar for every k. For every LLR grammar there exists an LLR parser that parses the grammar in linear time.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

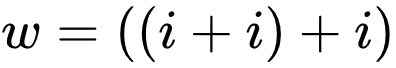
Two nomenclative outlier parser types are LL(\*) and LL(finite). A parser is called LL(\*)/LL(finite) if it uses the LL(\*)/LL(finite) parsing strategy. [[5]](https://en.wikipedia.org/wiki/LL_parser#cite_note-5)[[6]](https://en.wikipedia.org/wiki/LL_parser#cite_note-6) LL(\*) and LL(finite) parsers are functionally more closely resemblant to [PEG](https://en.wikipedia.org/wiki/Parsing_expression_grammar) parsers. An LL(finite) parser can parse an arbitrary LL(k) grammar optimally in the amount of lookahead and lookahead comparisons. The class of grammars parsable by the LL(\*) strategy encompasses some context-sensitive languages due to the use of syntactic and semantic predicates and has not been identified. It has been suggested that LL(\*) parsers are better thought of as [TDPL](https://en.wikipedia.org/wiki/Top-down_parsing_language) parsers.[[7]](https://en.wikipedia.org/wiki/LL_parser#cite_note-7) Against the popular misconception, LL(\*) parsers are not LLR in general, and are guaranteed by construction to perform worse on average (super-linear against linear time) and far worse in the worst-case (exponential against linear time).

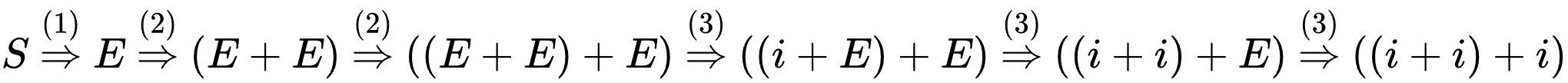
LL grammars, particularly LL(1) grammars, are of great practical interest, as parsers for these grammars are easy to construct, and many [computer languages](https://en.wikipedia.org/wiki/Computer_language) are designed to be LL(1) for this reason.[[8]](https://en.wikipedia.org/wiki/LL_parser#cite_note-8) LL parsers may be table-based,[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] i.e. similar to [LR parsers](https://en.wikipedia.org/wiki/LR_parser), but LL grammars can also be parsed by [recursive descent parsers](https://en.wikipedia.org/wiki/Recursive_descent_parser). According to Waite and Goos (1984),[[9]](https://en.wikipedia.org/wiki/LL_parser#cite_note-9) LL(*k*) grammars were introduced by Stearns and Lewis (1969).[[10]](https://en.wikipedia.org/wiki/LL_parser#cite_note-10)

## Overview[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=1)]

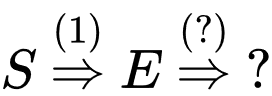
For a given [context-free grammar](https://en.wikipedia.org/wiki/Context-free_grammar), the parser attempts to find the [leftmost derivation](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees). Given an example grammar G {\displaystyle G} G:

1. S → E {\displaystyle S\to E} S \to E
2. E → ( E + E ) {\displaystyle E\to (E+E)} 
3. E → i {\displaystyle E\to i} E \to i

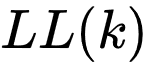
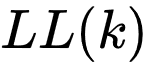
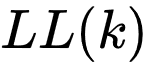
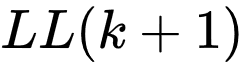
the leftmost derivation for w = ( ( i + i ) + i ) {\displaystyle w=((i+i)+i)} is:

S   ⇒ ( 1 )   E   ⇒ ( 2 )   ( E + E )   ⇒ ( 2 )   ( ( E + E ) + E )   ⇒ ( 3 )   ( ( i + E ) + E )   ⇒ ( 3 )   ( ( i + i ) + E )   ⇒ ( 3 )   ( ( i + i ) + i ) {\displaystyle S\ {\overset {(1)}{\Rightarrow }}\ E\ {\overset {(2)}{\Rightarrow }}\ (E+E)\ {\overset {(2)}{\Rightarrow }}\ ((E+E)+E)\ {\overset {(3)}{\Rightarrow }}\ ((i+E)+E)\ {\overset {(3)}{\Rightarrow }}\ ((i+i)+E)\ {\overset {(3)}{\Rightarrow }}\ ((i+i)+i)} 

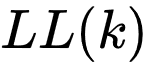
Generally, there are multiple possibilities when selecting a rule to expand the leftmost non-terminal. In step 2 of the previous example, the parser must choose whether to apply rule 2 or rule 3:

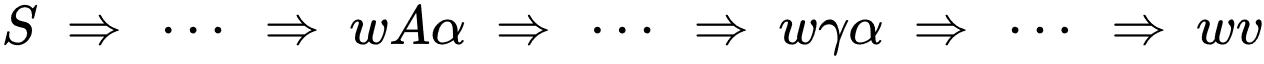
S   ⇒ ( 1 )   E   ⇒ ( ? )   ? {\displaystyle S\ {\overset {(1)}{\Rightarrow }}\ E\ {\overset {(?)}{\Rightarrow }}\ ?} 

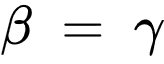
To be efficient, the parser must be able to make this choice deterministically when possible, without backtracking. For some grammars, it can do this by peeking on the unread input (without reading). In our example, if the parser knows that the next unread symbol is ( {\displaystyle (} (, the only correct rule that can be used is 2.

Generally, an L L ( k ) {\displaystyle LL(k)} parser can look ahead at k {\displaystyle k} ksymbols. However, given a grammar, the problem of determining if there exists a L L ( k ) {\displaystyle LL(k)} parser for some k {\displaystyle k} kthat recognizes it is undecidable. For each k {\displaystyle k} k, there is a language that cannot be recognized by an L L ( k ) {\displaystyle LL(k)} parser, but can be by an L L ( k + 1 ) {\displaystyle LL(k+1)} .

We can use the above analysis to give the following formal definition:

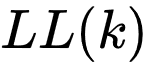
Let G {\displaystyle G} Gbe a context-free grammar and k ≥ 1 {\displaystyle k\geq 1} k \ge 1. We say that G {\displaystyle G} Gis L L ( k ) {\displaystyle LL(k)} , if and only if for any two leftmost derivations:

1. S   ⇒   ⋯   ⇒   w A α   ⇒   ⋯   ⇒   w β α   ⇒   ⋯   ⇒   w u {\displaystyle S\ \Rightarrow \ \cdots \ \Rightarrow \ wA\alpha \ \Rightarrow \ \cdots \ \Rightarrow \ w\beta \alpha \ \Rightarrow \ \cdots \ \Rightarrow \ wu} {\displaystyle S\ \Rightarrow \ \cdots \ \Rightarrow \ wA\alpha \ \Rightarrow \ \cdots \ \Rightarrow \ w\beta \alpha \ \Rightarrow \ \cdots \ \Rightarrow \ wu}
2. S   ⇒   ⋯   ⇒   w A α   ⇒   ⋯   ⇒   w γ α   ⇒   ⋯   ⇒   w v {\displaystyle S\ \Rightarrow \ \cdots \ \Rightarrow \ wA\alpha \ \Rightarrow \ \cdots \ \Rightarrow \ w\gamma \alpha \ \Rightarrow \ \cdots \ \Rightarrow \ wv} 

the following condition holds: the prefix of the string u {\displaystyle u} uof length k {\displaystyle k} kequals the prefix of the string v {\displaystyle v} vof length k {\displaystyle k} kimplies β   =   γ {\displaystyle \beta \ =\ \gamma } .

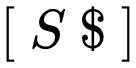
In this definition, S {\displaystyle S} Sis the start symbol and A {\displaystyle A} Aany non-terminal. The already derived input w {\displaystyle w} w, and yet unread u {\displaystyle u} uand v {\displaystyle v} vare strings of terminals. The Greek letters α {\displaystyle \alpha } \alpha , β {\displaystyle \beta } \beta and γ {\displaystyle \gamma } \gamma represent any string of both terminals and non-terminals (possibly empty). The prefix length corresponds to the lookahead buffer size, and the definition says that this buffer is enough to distinguish between any two derivations of different words.

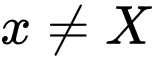
## Parser[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=2)]

The L L ( k ) {\displaystyle LL(k)} parser is a [deterministic pushdown automaton](https://en.wikipedia.org/wiki/Deterministic_pushdown_automaton) with the ability to peek on the next k {\displaystyle k} kinput symbols without reading. This peek capability can be emulated by storing the lookahead buffer contents in the finite state space, since both buffer and input alphabet are finite in size. As a result, this does not make the automaton more powerful, but is a convenient abstraction.

The stack alphabet is Γ = N ∪ Σ {\displaystyle \Gamma =N\cup \Sigma } \Gamma = N \cup \Sigma, where:

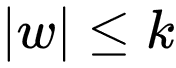
* N {\displaystyle N} Nis the set of non-terminals;
* Σ {\displaystyle \Sigma } \Sigma the set of terminal (input) symbols with a special end-of-input (EOI) symbol $ {\displaystyle \$} {\displaystyle \$}.

The parser stack initially contains the starting symbol above the EOI: [   S   $   ] {\displaystyle [\ S\ \$\ ]} . During operation, the parser repeatedly replaces the symbol X {\displaystyle X} Xon top of the stack:

* with some α {\displaystyle \alpha } \alpha , if X ∈ N {\displaystyle X\in N} X \in Nand there is a rule X → α {\displaystyle X\to \alpha } X\to \alpha ;
* with ϵ {\displaystyle \epsilon } \epsilon (in some notations λ {\displaystyle \lambda } \lambda ), i.e. X {\displaystyle X} Xis popped off the stack, if X ∈ Σ {\displaystyle X\in \Sigma } X \in \Sigma. In this case, an input symbol x {\displaystyle x} xis read and if x ≠ X {\displaystyle x\neq X} , the parser rejects the input.

If the last symbol to be removed from the stack is the EOI, the parsing is successful; the automaton accepts via an empty stack.

The states and the transition function are not explicitly given; they are specified (generated) using a more convenient *parse table* instead. The table provides the following mapping:

* row: top-of-stack symbol X {\displaystyle X} X
* column: | w | ≤ k {\displaystyle |w|\leq k} lookahead buffer contents
* cell: rule number for X → α {\displaystyle X\to \alpha } X\to \alpha or ϵ {\displaystyle \epsilon } \epsilon 

If the parser cannot perform a valid transition, the input is rejected (empty cells). To make the table more compact, only the non-terminal rows are commonly displayed, since the action is the same for terminals.

## Concrete example

### Set up[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=4)]

To explain an LL(1) parser's workings we will consider the following small (1) grammar:

1. S → F
2. S → ( S + F )
3. F → a

and parse the following input:

**( a + a )**

An LL(1) parsing table for a grammar has a row for each of the non-terminals and a column for each terminal (including the special terminal, represented here as **$**, that is used to indicate the end of the input stream).

Each cell of the table may point to at most one rule of the grammar (identified by its number). For example, in the parsing table for the above grammar, the cell for the non-terminal 'S' and terminal '(' points to the rule number 2:

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

The algorithm to construct a parsing table is described in a later section, but first let's see how the parser uses the parsing table to process its input.

### Parsing procedure[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=5)]

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 by removing 'S' from stack and pushing ')', 'F', '+', 'S', '(' onto the stack, and this writes the rule number 2 to the output. The stack then becomes:

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

In the second step, the parser removes the '**(**' from its input stream and from its stack, since they now 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, **)**, **$** ]

The parser now has an '**a'** on the input stream and an **'a'** at its stack top. Because they are the same, it removes it from the input stream and pops it from the top of the stack. The parser then has an '**+'** on the input stream and **'+'** is at the top of the stack meaning, like with 'a', it is popped from the stack and removed from the input stream. 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](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees) of the input string, which is:

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

### Parser implementation in C++[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=6)]

Below follows a C++ implementation of a table-based LL parser for the example language:

#include <iostream>

#include <map>

#include <stack>

enum Symbols {

// the symbols:

// Terminal symbols:

TS\_L\_PARENS, // (

TS\_R\_PARENS, // )

TS\_A, // a

TS\_PLUS, // +

TS\_EOS, // $, in this case corresponds to '\0'

TS\_INVALID, // invalid token

// Non-terminal symbols:

NTS\_S, // S

NTS\_F // F

};

/\*

Converts a valid token to the corresponding terminal symbol

\*/

Symbols lexer(char c)

{

switch (c)

{

case '(': return TS\_L\_PARENS;

case ')': return TS\_R\_PARENS;

case 'a': return TS\_A;

case '+': return TS\_PLUS;

case '\0': return TS\_EOS; // end of stack: the $ terminal symbol

default: return TS\_INVALID;

}

}

int main(int argc, char \*\*argv)

{

using namespace std;

if (argc < 2)

{

cout << "usage:\n\tll '(a+a)'" << endl;

return 0;

}

// LL parser table, maps < non-terminal, terminal> pair to action

map< Symbols, map<Symbols, int> > table;

stack<Symbols> ss; // symbol stack

char \*p; // input buffer

// initialize the symbols stack

ss.push(TS\_EOS); // terminal, $

ss.push(NTS\_S); // non-terminal, S

// initialize the symbol stream cursor

p = &argv[1][0];

// set up the parsing table

table[NTS\_S][TS\_L\_PARENS] = 2;

table[NTS\_S][TS\_A] = 1;

table[NTS\_F][TS\_A] = 3;

while (ss.size() > 0)

{

if (lexer(\*p) == ss.top())

{

cout << "Matched symbols: " << lexer(\*p) << endl;

p++;

ss.pop();

}

else

{

cout << "Rule " << table[ss.top()][lexer(\*p)] << endl;

switch (table[ss.top()][lexer(\*p)])

{

case 1: // 1. S → F

ss.pop();

ss.push(NTS\_F); // F

break;

case 2: // 2. S → ( S + F )

ss.pop();

ss.push(TS\_R\_PARENS); // )

ss.push(NTS\_F); // F

ss.push(TS\_PLUS); // +

ss.push(NTS\_S); // S

ss.push(TS\_L\_PARENS); // (

break;

case 3: // 3. F → a

ss.pop();

ss.push(TS\_A); // a

break;

default:

cout << "parsing table defaulted" << endl;

return 0;

}

}

}

cout << "finished parsing" << endl;

return 0;

}

### Parser implementation in Python[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=7)]

# All constants are indexed from 0

TERM = 0

RULE = 1

# Terminals

T\_LPAR = 0

T\_RPAR = 1

T\_A = 2

T\_PLUS = 3

T\_END = 4

T\_INVALID = 5

# Non-Terminals

N\_S = 0

N\_F = 1

# Parse table

table = [[ 1, -1, 0, -1, -1, -1],

[-1, -1, 2, -1, -1, -1]]

RULES = [[(RULE, N\_F)],

[(TERM, T\_LPAR), (RULE, N\_S), (TERM, T\_PLUS), (RULE, N\_F), (TERM, T\_RPAR)],

[(TERM, T\_A)]]

stack = [(TERM, T\_END), (RULE, N\_S)]

def lexical\_analysis(inputstring):

print("Lexical analysis")

tokens = []

for c in inputstring:

if c == "+": tokens.append(T\_PLUS)

elif c == "(": tokens.append(T\_LPAR)

elif c == ")": tokens.append(T\_RPAR)

elif c == "a": tokens.append(T\_A)

else: tokens.append(T\_INVALID)

tokens.append(T\_END)

print(tokens)

return tokens

def syntactic\_analysis(tokens):

print("Syntactic analysis")

position = 0

while len(stack) > 0:

(stype, svalue) = stack.pop()

token = tokens[position]

if stype == TERM:

if svalue == token:

position += 1

print("pop", svalue)

if token == T\_END:

print("input accepted")

else:

print("bad term on input:", token)

break

elif stype == RULE:

print("svalue", svalue, "token", token)

rule = table[svalue][token]

print("rule", rule)

for r in reversed(RULES[rule]):

stack.append(r)

print("stack", stack)

inputstring = "(a+a)"

syntactic\_analysis(lexical\_analysis(inputstring))

## Remarks[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=8)]

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 the parser 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 nonterminal 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 the parser reports an error and stops.
* If the top is a terminal then the parser 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](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees) to the output stream or it will have reported an error.

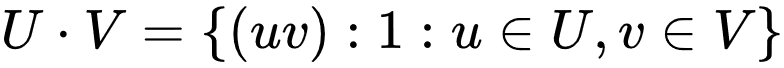
## Constructing an LL(1) parsing table[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=9)]

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 some 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**(*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(*aw'*) = { *a* } for every terminal *a*
   * Fi(*Aw'*) = **Fi**(*A*) for every nonterminal *A* with ε not in **Fi**(*A*)
   * Fi(*Aw'* ) = (**Fi**(*A*) \ { ε }) ∪ Fi(*w'* ) for every nonterminal *A* with ε in **Fi**(*A*)
   * Fi(ε) = { ε }
3. add Fi(*wi*) to **Fi**(*Ai*) for every rule *Ai* → *wi*
4. do steps 2 and 3 until all **Fi** sets stay the same.

The result is the least fixed point solution to the following system:

* **Fi**(*A*) ⊇ **Fi**(*w*) for each rule A → *w*
* **Fi**(*a*) ⊇ { *a* }, for each terminal *a*
* **Fi**(*w*0 *w*1) ⊇ **Fi**(*w*0)·**Fi**(*w1*), for all words *w*0 and *w*1
* **Fi**(ε) ⊇ {ε}

where, for sets of words U and V, the truncated product is defined by U ⋅ V = { ( u v ) : 1 : u ∈ U , v ∈ V } {\displaystyle U\cdot V=\{(uv):1:u\in U,v\in V\}} , and w:1 denotes the initial length-1 prefix of words w of length 2 or more, or w, itself, if w has length 0 or 1.

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. We use **$** as a special terminal indicating end of input stream, and *S* as start symbol.

Computing the Follow-sets for the nonterminals in a grammar can be done as follows:

1. initialize **Fo**(*S*) with { **$** } and every other **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*)
   * if *w'* has length 0, then add **Fo**(*Aj*) to **Fo**(*Ai*)
3. repeat step 2 until all **Fo** sets stay the same.

This provides the least fixed point solution to the following system:

* **Fo**(*S*) ⊇ {**$**}
* **Fo**(*A*) ⊇ **Fi**(*w*)·**Fo**(*B*) for each rule of the form B → ... A w

Now we can define exactly which rules will appear 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

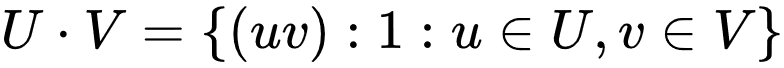
ε is in **Fi**(*w*) and *a* is in **Fo**(*A*).

Equivalently: *T*[*A*, *a*] contains the rule *A* → *w* for each *a* ∈ **Fi**(*w*)·**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[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=10)]

The construction for LL(1) parsers can be adapted to LL(*k*) for *k* > 1 with the following modifications:

* the truncated product is defined U ⋅ V = { ( u v ) : 1 : u ∈ U , v ∈ V } {\displaystyle U\cdot V=\{(uv):1:u\in U,v\in V\}} , where w:k denotes the initial length-k prefix of words of length > k, or w, itself, if w has length k or less,
* **Fo**(*S*) = {**$**k}
* Apply **Fi**(αβ) = **Fi**(α) ⋅ {\displaystyle \cdot } \cdot **Fi**(β) also in step 2 of the **Fi** construction given for LL(1).
* In step 2 of the **Fo** construction, for *Aj* → *wAiw'* simply add **Fi**(*w'*) ⋅ {\displaystyle \cdot } \cdot **Fo**(*Aj*) to **Fo**(*Ai*).

where an input is suffixed by k end-markers **$**, to fully account for the k lookahead context. This approach eliminates special cases for ε, and can be applied equally well in the LL(1) case.

Until the mid-1990s, it was widely believed that LL(*k*) parsing (for *k* > 1) was impractical,[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] since the parser table would have [exponential](https://en.wikipedia.org/wiki/Exponential_function) size in *k* in the worst case. This perception changed gradually after the release of the [Purdue Compiler Construction Tool Set](https://en.wikipedia.org/wiki/Antlr) around 1992, when it was demonstrated that many [programming languages](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Yacc) use [LALR(1)](https://en.wikipedia.org/wiki/LALR_parser) parser tables to construct a restricted [LR parser](https://en.wikipedia.org/wiki/LR_parser) with a fixed one-token lookahead.

## Conflicts[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=11)]

As described in the introduction, LL(1) parsers recognize languages that have LL(1) grammars, which are a special case of context-free grammars; LL(1) parsers cannot recognize all context-free languages. The LL(1) languages are a proper subset of the LR(1) languages, which in turn are a proper subset of all context-free languages. In order for a context-free grammar to be an LL(1) grammar, certain conflicts must not arise, which we describe in this section.

### Terminology[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=12)]

Let *A* be a non-terminal. FIRST(*A*) is (defined to be) the set of terminals that can appear in the first position of any string derived from *A*. FOLLOW(*A*) is the union over:[[11]](https://en.wikipedia.org/wiki/LL_parser#cite_note-11)

1. FIRST(*B*) where *B* is any non-terminal that immediately follows *A* in the right-hand side of a [production rule](https://en.wikipedia.org/wiki/Formal_grammar#The_syntax_of_grammars).
2. FOLLOW(*B*) where *B* is any head of a rule of the form *B* → *wA*.

### LL(1) conflicts[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=13)]

There are two main types of LL(1) conflicts:

#### FIRST/FIRST conflict[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=14)]

The FIRST sets of two different grammar rules for the same non-terminal intersect. An example of an LL(1) FIRST/FIRST conflict:

S -> E | E 'a'

E -> 'b' | ε

FIRST(*E*) = {*b*, ε} and FIRST(*E* *a*) = {*b*, *a*}, so when the table is drawn, there is conflict under terminal *b* of production rule *S*.

##### Special case: left recursion[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=15)]

[Left recursion](https://en.wikipedia.org/wiki/Left_recursion) will cause a FIRST/FIRST conflict with all alternatives.

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

#### FIRST/FOLLOW conflict[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=16)]

The FIRST and FOLLOW set of a grammar rule overlap. With an [empty string](https://en.wikipedia.org/wiki/Empty_string) (ε) in the FIRST set, it is unknown which alternative to select. An example of an LL(1) conflict:

S -> A 'a' 'b'

A -> 'a' | ε

The FIRST set of *A* is {*a*, ε}, and the FOLLOW set is {*a*}.

### Solutions to LL(1) conflicts[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=17)]

#### Left factoring[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=18)]

A common left-factor is "factored out".

A -> X | X Y Z

becomes

A -> X B

B -> Y Z | ε

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

Another example (more complex) using above FIRST/FIRST conflict example:

S -> E | E 'a'

E -> 'b' | ε

becomes (merging into a single non-terminal)

S -> 'b' | ε | 'b' 'a' | 'a'

then through left-factoring, becomes

S -> 'b' E | E

E -> 'a' | ε

#### Substitution[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=19)]

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[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=20)]

See.[[12]](https://en.wikipedia.org/wiki/LL_parser#cite_note-12)

For a general method, see [removing left recursion](https://en.wikipedia.org/wiki/Left_recursion#Removing_left_recursion). A simple example for left recursion removal: The following production rule has left recursion on E

E -> E '+' T

E -> T

This rule is nothing but list of Ts separated by '+'. In a regular expression form T ('+' T)\*. So the rule could be rewritten as

E -> T Z

Z -> '+' T Z

Z -> ε

Now there is no left recursion and no conflicts on either of the rules.

However, not all context-free grammars have an equivalent LL(k)-grammar, e.g.:

S -> A | B

A -> 'a' A 'b' | ε

B -> 'a' B 'b' 'b' | ε

It can be shown that there does not exist any LL(k)-grammar accepting the language generated by this grammar.

## See also[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=21)]

* [Comparison of parser generators](https://en.wikipedia.org/wiki/Comparison_of_parser_generators)
* [Parse tree](https://en.wikipedia.org/wiki/Parse_tree)
* [Top-down parsing](https://en.wikipedia.org/wiki/Top-down_parsing)
* [Bottom-up parsing](https://en.wikipedia.org/wiki/Bottom-up_parsing)

## Notes[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=22)]

1. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-1) Rosenkrantz, D. J.; Stearns, R. E. (1970). [*"Properties of Deterministic Top Down Grammars"*](https://doi.org/10.1016%2Fs0019-9958%2870%2990446-8). Information and Control. **17** (3): 226–256. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1016/s0019-9958(70)90446-8*](https://doi.org/10.1016%2Fs0019-9958%2870%2990446-8).
2. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-2) Jarzabek, Stanislav; Krawczyk, Tomasz (1974). "LL-Regular Grammars". Instytutu Maszyn Matematycznych: 107–119.
3. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-3) Jarzabek, Stanislav; Krawczyk, Tomasz (Nov 1975). [*"LL-Regular Grammars"*](https://www.sciencedirect.com/science/article/abs/pii/0020019075900095). [*Information Processing Letters*](https://en.wikipedia.org/wiki/Information_Processing_Letters). **4** (2): 31–37. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1016/0020-0190(75)90009-5*](https://doi.org/10.1016%2F0020-0190%2875%2990009-5).
4. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-4) David A. Poplawski (Aug 1977). [*Properties of LL-Regular Languages*](https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1176&context=cstech) (Technical Report). [*Purdue University*](https://en.wikipedia.org/wiki/Purdue_University), Department of Computer Science.
5. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-5) Parr, Terence and Fisher, Kathleen (2011). "LL (\*) the foundation of the ANTLR parser generator". ACM SIGPLAN Notices. **46** (6): 425–436. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1145/1993316.1993548*](https://doi.org/10.1145%2F1993316.1993548).{{[cite journal](https://en.wikipedia.org/wiki/Template:Cite_journal)}}: CS1 maint: multiple names: authors list ([link](https://en.wikipedia.org/wiki/Category:CS1_maint:_multiple_names:_authors_list))
6. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-6) Belcak, Peter (2020). "The LL(finite) parsing strategy for optimal LL(k) parsing". [arXiv](https://en.wikipedia.org/wiki/ArXiv_(identifier)):[*2010.07874*](https://arxiv.org/abs/2010.07874) [[*cs.PL*](https://arxiv.org/archive/cs.PL)].
7. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-7) Ford, Bryan (2004). "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". ACM SIGPLAN Notices. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1145/982962.964011*](https://doi.org/10.1145%2F982962.964011).
8. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-8) Pat Terry (2005). [*Compiling with C# and Java*](https://books.google.com/books?id=4O9ffYfX_H0C). Pearson Education. pp. 159–164. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [*9780321263605*](https://en.wikipedia.org/wiki/Special:BookSources/9780321263605).
9. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-9) William M. Waite and Gerhard Goos (1984). Compiler Construction. Texts and Monographs in Computer Science. Heidelberg: Springer. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [*978-3-540-90821-0*](https://en.wikipedia.org/wiki/Special:BookSources/978-3-540-90821-0). Here: Sect. 5.3.2, p. 121-127; in particular, p. 123.
10. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-10) [*Richard E. Stearns*](https://en.wikipedia.org/wiki/Richard_E._Stearns) and P.M. Lewis (1969). [*"Property Grammars and Table Machines"*](https://doi.org/10.1016%2FS0019-9958%2869%2990312-X). [Information and Control](https://en.wikipedia.org/wiki/Information_and_Control). **14** (6): 524–549. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[*10.1016/S0019-9958(69)90312-X*](https://doi.org/10.1016%2FS0019-9958%2869%2990312-X).
11. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-11) [*"LL Grammars"*](http://www.cs.uaf.edu/~cs331/notes/LL.pdf) *(PDF)*. [*Archived*](https://web.archive.org/web/20100618193203/http://www.cs.uaf.edu/~cs331/notes/LL.pdf) *(PDF)* from the original on 2010-06-18*. Retrieved 2010-05-11*.
12. [**^**](https://en.wikipedia.org/wiki/LL_parser#cite_ref-12) [Modern Compiler Design](https://books.google.com/books?id=zkpFTBtK7a4C&printsec=frontcover#v=onepage&q=%22left%20recursion%22&f=false), Grune, Bal, Jacobs and Langendoen

## External links[[edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit&section=23)]

* [A tutorial on implementing LL(1) parsers in C# (archived)](https://web.archive.org/web/20131228024914/http://www.itu.dk/people/kfl/parsernotes.pdf)
* [Parsing Simulator](http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php) This simulator is used to generate parsing tables LL(1) and to resolve the exercises of the book.
* [LL(1) DSL PEG parser (toolkit framework)](http://expressionscompiler.codeplex.com/)
* [Language theoretic comparison of LL and LR grammars](https://cs.stackexchange.com/q/43)
* [LL(k) Parsing Theory](http://www.h8dems.com/llkparse.html)

|  |  |
| --- | --- |
| **hide**   * [**v**](https://en.wikipedia.org/wiki/Template:Parsers) * [**t**](https://en.wikipedia.org/wiki/Template_talk:Parsers) * [**e**](https://en.wikipedia.org/w/index.php?title=Template:Parsers&action=edit)   [**Parsing algorithms**](https://en.wikipedia.org/wiki/Parsing) | |
| [**Top-down**](https://en.wikipedia.org/wiki/Top-down_parsing) | * LL * [Recursive descent](https://en.wikipedia.org/wiki/Recursive_descent_parser)   + [Tail recursive](https://en.wikipedia.org/wiki/Tail_recursive_parser) |
| [**Bottom-up**](https://en.wikipedia.org/wiki/Bottom-up_parsing) | * Precedence   + [Simple](https://en.wikipedia.org/wiki/Simple_precedence_parser)   + [Operator](https://en.wikipedia.org/wiki/Operator-precedence_parser)     - [Shunting-yard](https://en.wikipedia.org/wiki/Shunting_yard_algorithm) * [LR](https://en.wikipedia.org/wiki/LR_parser)   + [Simple](https://en.wikipedia.org/wiki/Simple_LR_parser)   + [Look-ahead](https://en.wikipedia.org/wiki/LALR_parser)   + [Canonical](https://en.wikipedia.org/wiki/Canonical_LR_parser)   + [Generalized](https://en.wikipedia.org/wiki/GLR_parser) * [CYK](https://en.wikipedia.org/wiki/CYK_algorithm) * [Recursive ascent](https://en.wikipedia.org/wiki/Recursive_ascent_parser) * [Shift-reduce](https://en.wikipedia.org/wiki/Shift-reduce_parser) |
| **Mixed, other** | * [Combinator](https://en.wikipedia.org/wiki/Parser_combinator) * [Chart](https://en.wikipedia.org/wiki/Chart_parser) * [Earley](https://en.wikipedia.org/wiki/Earley_parser) |
| **Related topics** | * [PEG](https://en.wikipedia.org/wiki/Parsing_expression_grammar) * [Definite clause grammar](https://en.wikipedia.org/wiki/Definite_clause_grammar) * [Deterministic parsing](https://en.wikipedia.org/wiki/Deterministic_parsing) * [Dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming) * [Memoization](https://en.wikipedia.org/wiki/Memoization) * [Parser generator](https://en.wikipedia.org/wiki/Compiler-compiler)   + [LALR](https://en.wikipedia.org/wiki/LALR_parser_generator) * [Parse tree](https://en.wikipedia.org/wiki/Parse_tree) * [AST](https://en.wikipedia.org/wiki/Abstract_syntax_tree) * [Scannerless parsing](https://en.wikipedia.org/wiki/Scannerless_parsing) * [History of compiler construction](https://en.wikipedia.org/wiki/History_of_compiler_construction) * [Comparison of parser generators](https://en.wikipedia.org/wiki/Comparison_of_parser_generators) * [Operator-precedence grammar](https://en.wikipedia.org/wiki/Operator-precedence_grammar) |

Retrieved from "<https://en.wikipedia.org/w/index.php?title=LL_parser&oldid=1133787821>"

[Categories](https://en.wikipedia.org/wiki/Help:Category):

* [Parsing algorithms](https://en.wikipedia.org/wiki/Category:Parsing_algorithms)

Hidden categories:

* [CS1 maint: multiple names: authors list](https://en.wikipedia.org/wiki/Category:CS1_maint:_multiple_names:_authors_list)
* [Articles with short description](https://en.wikipedia.org/wiki/Category:Articles_with_short_description)
* [Short description is different from Wikidata](https://en.wikipedia.org/wiki/Category:Short_description_is_different_from_Wikidata)
* [Wikipedia articles needing clarification from August 2021](https://en.wikipedia.org/wiki/Category:Wikipedia_articles_needing_clarification_from_August_2021)
* [All articles with unsourced statements](https://en.wikipedia.org/wiki/Category:All_articles_with_unsourced_statements)
* [Articles with unsourced statements from June 2022](https://en.wikipedia.org/wiki/Category:Articles_with_unsourced_statements_from_June_2022)
* [Articles with unsourced statements from February 2019](https://en.wikipedia.org/wiki/Category:Articles_with_unsourced_statements_from_February_2019)
* [Articles with unsourced statements from February 2007](https://en.wikipedia.org/wiki/Category:Articles_with_unsourced_statements_from_February_2007)
* [Articles with example C++ code](https://en.wikipedia.org/wiki/Category:Articles_with_example_C%2B%2B_code)
* [Articles with example Python (programming language) code](https://en.wikipedia.org/wiki/Category:Articles_with_example_Python_(programming_language)_code)
* [Pages that use a deprecated format of the math tags](https://en.wikipedia.org/wiki/Category:Pages_that_use_a_deprecated_format_of_the_math_tags)

## Navigation menu

### Personal tools

* Not logged in
* [Talk](https://en.wikipedia.org/wiki/Special:MyTalk)
* [Contributions](https://en.wikipedia.org/wiki/Special:MyContributions)
* [Create account](https://en.wikipedia.org/w/index.php?title=Special:CreateAccount&returnto=LL+parser)
* [Log in](https://en.wikipedia.org/w/index.php?title=Special:UserLogin&returnto=LL+parser)

### Namespaces

* [Article](https://en.wikipedia.org/wiki/LL_parser)
* [Talk](https://en.wikipedia.org/wiki/Talk:LL_parser)

English

### Views

* [Read](https://en.wikipedia.org/wiki/LL_parser)
* [Edit](https://en.wikipedia.org/w/index.php?title=LL_parser&action=edit)
* [View history](https://en.wikipedia.org/w/index.php?title=LL_parser&action=history)

More

Top of Form



Bottom of Form

### Navigation

* [Main page](https://en.wikipedia.org/wiki/Main_Page)
* [Contents](https://en.wikipedia.org/wiki/Wikipedia:Contents)
* [Current events](https://en.wikipedia.org/wiki/Portal:Current_events)
* [Random article](https://en.wikipedia.org/wiki/Special:Random)
* [About Wikipedia](https://en.wikipedia.org/wiki/Wikipedia:About)
* [Contact us](https://en.wikipedia.org/wiki/Wikipedia:Contact_us)
* [Donate](https://donate.wikimedia.org/wiki/Special:FundraiserRedirector?utm_source=donate&utm_medium=sidebar&utm_campaign=C13_en.wikipedia.org&uselang=en)

### Contribute

* [Help](https://en.wikipedia.org/wiki/Help:Contents)
* [Learn to edit](https://en.wikipedia.org/wiki/Help:Introduction)
* [Community portal](https://en.wikipedia.org/wiki/Wikipedia:Community_portal)
* [Recent changes](https://en.wikipedia.org/wiki/Special:RecentChanges)
* [Upload file](https://en.wikipedia.org/wiki/Wikipedia:File_upload_wizard)

### Tools

* [What links here](https://en.wikipedia.org/wiki/Special:WhatLinksHere/LL_parser)
* [Related changes](https://en.wikipedia.org/wiki/Special:RecentChangesLinked/LL_parser)
* [Upload file](https://en.wikipedia.org/wiki/Wikipedia:File_Upload_Wizard)
* [Special pages](https://en.wikipedia.org/wiki/Special:SpecialPages)
* [Permanent link](https://en.wikipedia.org/w/index.php?title=LL_parser&oldid=1133787821)
* [Page information](https://en.wikipedia.org/w/index.php?title=LL_parser&action=info)
* [Cite this page](https://en.wikipedia.org/w/index.php?title=Special:CiteThisPage&page=LL_parser&id=1133787821&wpFormIdentifier=titleform)
* [Wikidata item](https://www.wikidata.org/wiki/Special:EntityPage/Q932615)

### Print/export

* [Download as PDF](https://en.wikipedia.org/w/index.php?title=Special:DownloadAsPdf&page=LL_parser&action=show-download-screen)
* [Printable version](https://en.wikipedia.org/w/index.php?title=LL_parser&printable=yes)

### Languages

* [العربية](https://ar.wikipedia.org/wiki/%D8%A7%D9%84%D9%85%D8%AD%D9%84%D9%84_%D8%A7%D9%84%D9%84%D8%BA%D9%88%D9%8A_LL)
* [Čeština](https://cs.wikipedia.org/wiki/LL_syntaktick%C3%BD_analyz%C3%A1tor)
* [Deutsch](https://de.wikipedia.org/wiki/LL-Parser)
* [Español](https://es.wikipedia.org/wiki/Analizador_sint%C3%A1ctico_LL)
* [فارسی](https://fa.wikipedia.org/wiki/%D8%AA%D8%AC%D8%B2%DB%8C%D9%87%E2%80%8C%DA%A9%D9%86%D9%86%D8%AF%D9%87_%D8%A7%D9%84%E2%80%8C%D8%A7%D9%84)
* [Français](https://fr.wikipedia.org/wiki/Analyse_LL)
* [한국어](https://ko.wikipedia.org/wiki/LL_%ED%8C%8C%EC%84%9C)
* [日本語](https://ja.wikipedia.org/wiki/LL%E6%B3%95)
* [Norsk bokmål](https://no.wikipedia.org/wiki/LL-parser)
* [Polski](https://pl.wikipedia.org/wiki/Parser_LL)
* [Português](https://pt.wikipedia.org/wiki/Analisador_sint%C3%A1tico_LL)
* [Русский](https://ru.wikipedia.org/wiki/LL-%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80)
* [Српски / srpski](https://sr.wikipedia.org/wiki/%D0%9B%D0%9B_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80)
* [Українська](https://uk.wikipedia.org/wiki/LL-%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%82%D0%BE%D1%80)
* [中文](https://zh.wikipedia.org/wiki/LL%E5%89%96%E6%9E%90%E5%99%A8)

6 more

[Edit links](https://www.wikidata.org/wiki/Special:EntityPage/Q932615#sitelinks-wikipedia)

* This page was last edited on 15 January 2023, at 14:39 (UTC).
* Text is available under the [Creative Commons Attribution-ShareAlike License 3.0](https://en.wikipedia.org/wiki/Wikipedia:Text_of_the_Creative_Commons_Attribution-ShareAlike_3.0_Unported_License); additional terms may apply. By using this site, you agree to the [Terms of Use](https://foundation.wikimedia.org/wiki/Terms_of_Use) and [Privacy Policy](https://foundation.wikimedia.org/wiki/Privacy_policy). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](https://www.wikimediafoundation.org/), a non-profit organization.
* [Privacy policy](https://foundation.wikimedia.org/wiki/Privacy_policy)
* [About Wikipedia](https://en.wikipedia.org/wiki/Wikipedia:About)
* [Disclaimers](https://en.wikipedia.org/wiki/Wikipedia:General_disclaimer)
* [Contact Wikipedia](https://en.wikipedia.org/wiki/Wikipedia:Contact_us)
* [Mobile view](https://en.m.wikipedia.org/w/index.php?title=LL_parser&mobileaction=toggle_view_mobile)
* [Developers](https://developer.wikimedia.org/)
* [Statistics](https://stats.wikimedia.org/#/en.wikipedia.org)
* [Cookie statement](https://foundation.wikimedia.org/wiki/Cookie_statement)
* [Edit preview settings](https://en.wikipedia.org/wiki/LL_parser)
* [Wikimedia Foundation](https://wikimediafoundation.org/)
* [Powered by MediaWiki](https://www.mediawiki.org/)