



WIKIPEDIA
The Free Encyclopedia

[Main page](#)

[Contents](#)

[Featured content](#)

[Current events](#)

[Random article](#)

[Donate to Wikipedia](#)

[Wikipedia store](#)

Interaction

[Help](#)

[About Wikipedia](#)

[Community portal](#)

[Recent changes](#)

[Contact page](#)

Tools

[What links here](#)

[Related changes](#)

[Upload file](#)

[Special pages](#)

[Permanent link](#)

[Page information](#)

[Wikidata item](#)

[Cite this page](#)

Print/export

[Create a book](#)

[Download as PDF](#)

[Printable version](#)

Languages

[العربية](#)

[Čeština](#)

[Deutsch](#)

[Español](#)

[Français](#)

[Italiano](#)

[日本語](#)

[Polski](#)

[Português](#)

[Русский](#)

[中文](#)

[Edit links](#)

Article [Talk](#)

[Read](#)

[Edit](#)

[View history](#)



LR parser

From Wikipedia, the free encyclopedia

In **computer science**, **LR parsers** are a type of **bottom-up parsers** that efficiently handle **deterministic context-free languages** in guaranteed linear time.^[1] The **LALR parsers** and the **SLR parsers** are common variants of LR parsers. LR parsers are often mechanically generated from a **formal grammar** for the language by a **parser generator** tool. They are very widely used for the processing of **computer languages**, more than other kinds of generated parsers.^[*citation needed*]

The name **LR** is an acronym. The **L** means that the **parser** reads input text in one direction without backing up; that direction is typically **Left** to right within each line, and top to bottom across the lines of the full input file. (This is true for most parsers.) The **R** means that the parser produces a **reversed Rightmost derivation**; it does a **bottom-up parse**, not a **top-down LL parse** or ad-hoc parse. The name LR is often followed by a numeric qualifier, as in **LR(1)** or sometimes **LR(*k*)**. To avoid **backtracking** or guessing, the LR parser is allowed to peek ahead at *k* **lookahead** input symbols before deciding how to parse earlier symbols. Typically *k* is 1 and is not mentioned. The name LR is often preceded by other qualifiers, as in **SLR** and **LALR**.

LR parsers are deterministic; they produce a single correct parse without guesswork or backtracking, in linear time. This is ideal for computer languages. But LR parsers are not suited for human languages which need more flexible but slower methods. Other parser methods (**CYK algorithm**, **Earley parser**, and **GLR parser**) that backtrack or yield multiple parses may take *O*(*n*²), *O*(*n*³) or even exponential time when they guess badly.

The above properties of **L**, **R**, and **k** are actually shared by all **shift-reduce parsers**, including **precedence parsers**. But by convention, the LR name stands for the form of parsing invented by **Donald Knuth**, and excludes the earlier, less powerful precedence methods (for example **Operator-precedence parser**).^[1] LR parsers can handle a larger range of languages and grammars than precedence parsers or top-down **LL parsing**.^[2] This is because the LR parser waits until it has seen an entire instance of some grammar pattern before committing to what it has found. An LL parser has to decide or guess what it is seeing much sooner, when it has only seen the leftmost input symbol of that pattern. LR is also better at error reporting. It detects syntax errors as early in the input stream as possible.

Contents [hide]

1 Overview

- 1.1 Bottom-up parse tree for example *A*2 + 1*
- 1.2 Shift and reduce actions
- 1.3 Bottom-up parse stack
- 1.4 Bottom-up parse steps for example *A*2 + 1*
- 1.5 LR parse steps for example *A*2 + 1*
- 1.6 Grammar for the example *A*2 + 1*
- 1.7 Parse table for the example grammar
- 1.8 LR parser loop

2 LR generator analysis

- 2.1 LR states
- 2.2 Finite state machine
- 2.3 Lookahead sets
- 2.4 Syntax error recovery
- 2.5 Variants of LR parsers
- 2.6 Theory

3 Additional example 1+1

- 3.1 Action and goto tables
- 3.2 Parsing steps
- 3.3 Walkthrough

4 Constructing LR(0) parsing tables

- 4.1 Items
- 4.2 Item sets
- 4.3 Extension of Item Set by expansion of non-terminals
- 4.4 Closure of item sets
- 4.5 Augmented grammar

5 Table construction

5.1 Finding the reachable item sets and the transitions between them

5.2 Constructing the action and goto tables

5.2.1 A note about LR(0) versus SLR and LALR parsing

5.3 Conflicts in the constructed tables

6 See also

7 References

8 Further reading

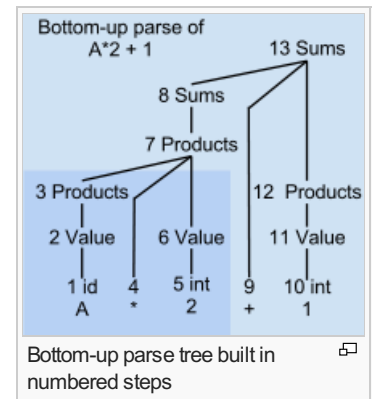
9 External links

Overview [\[edit\]](#)

Bottom-up parse tree for example $A*2 + 1$ [\[edit\]](#)

An LR parser scans and parses the input text in one forward pass over the text. The parser builds up the [parse tree](#) incrementally, bottom up, and left to right, without guessing or backtracking. At every point in this pass, the parser has accumulated a list of subtrees or phrases of the input text that have been already parsed. Those subtrees are not yet joined together because the parser has not yet reached the right end of the syntax pattern that will combine them.

At step 6 in the example parse, only " $A*2$ " has been parsed, incompletely. Only the shaded lower-left corner of the parse tree exists. None of the parse tree nodes numbered 7 and above exist yet. Nodes 3, 4, and 6 are the roots of isolated subtrees for variable A , operator $*$, and number 2, respectively. These three root nodes are temporarily held in a parse stack. The remaining unparsed portion of the input stream is " $+ 1$ ".



Shift and reduce actions [\[edit\]](#)

As with other shift-reduce parsers, an LR parser works by doing some combination of Shift steps and Reduce steps.

- A **Shift** step advances in the input stream by one symbol. That shifted symbol becomes a new single-node parse tree.
- A **Reduce** step applies a completed grammar rule to some of the recent parse trees, joining them together as one tree with a new root symbol.

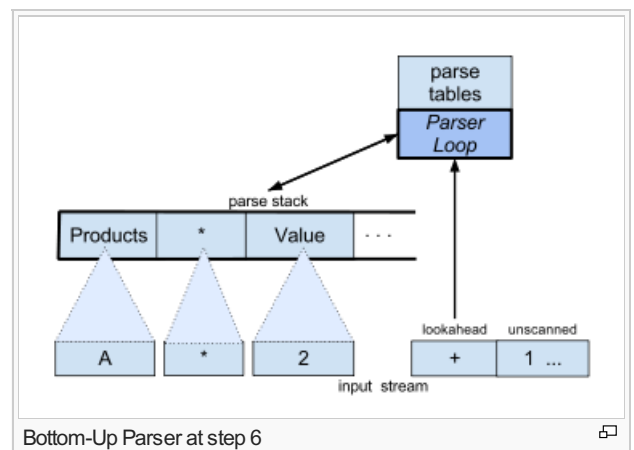
If the input has no syntax errors, the parser continues with these steps until all of the input has been consumed and all of the parse trees have been reduced to a single tree representing an entire legal input.

LR parsers differ from other shift-reduce parsers in how they decide when to reduce, and how to pick between rules with similar endings. But the final decisions and the sequence of shift or reduce steps are the same.

Much of the LR parser's efficiency is from being deterministic. To avoid guessing, the LR parser often looks ahead (rightwards) at the next scanned symbol, before deciding what to do with previously scanned symbols. The lexical scanner works one or more symbols ahead of the parser. The **lookahead** symbols are the 'right-hand context' for the parsing decision.^[3]

Bottom-up parse stack [\[edit\]](#)

Like other shift-reduce parsers, an LR parser lazily waits until it has scanned and parsed all parts of some construct before committing to what the combined construct is. The parser then acts immediately on the combination instead of waiting any further. In the parse tree example, the phrase A gets reduced to Value and then to Products in steps 1-3 as soon as lookahead $*$ is seen, rather than waiting any later to organize those parts of the parse tree. The decisions for how to handle A are based only on what the parser and scanner have already seen, without considering things that



appear much later to the right.

Reductions reorganize the most recently parsed things, immediately to the left of the lookahead symbol. So the list of already-parsed things acts like a [stack](#). This **parse stack** grows rightwards. The base or bottom of the stack is on the left and holds the leftmost, oldest parse fragment. Every reduction step acts only on the rightmost, newest parse fragments. (This accumulative parse stack is very unlike the predictive, leftward-growing parse stack used by [top-down parsers](#).)

Bottom-up parse steps for example $A*2 + 1$ [\[edit\]](#)

Step	Parse Stack	Unparsed	Shift/Reduce
0	<i>empty</i>	$A*2 + 1$	shift
1	<i>id</i>	$*2 + 1$	$\text{Value} \rightarrow id$
2	Value	$*2 + 1$	$\text{Products} \rightarrow \text{Value}$
3	Products	$*2 + 1$	shift
4	Products *	$2 + 1$	shift
5	Products * <i>int</i>	$+ 1$	$\text{Value} \rightarrow int$
6	Products * Value	$+ 1$	$\text{Products} \rightarrow \text{Products} * \text{Value}$
7	Products	$+ 1$	$\text{Sums} \rightarrow \text{Products}$
8	Sums	$+ 1$	shift
9	Sums +	1	shift
10	Sums + <i>int</i>	<i>eof</i>	$\text{Value} \rightarrow int$
11	Sums + Value	<i>eof</i>	$\text{Products} \rightarrow \text{Value}$
12	Sums + Products	<i>eof</i>	$\text{Sums} \rightarrow \text{Sums} + \text{Products}$
13	Sums	<i>eof</i>	done

Step 6 applies a grammar rule with multiple parts:

$\text{Products} \rightarrow \text{Products} * \text{Value}$

This matches the stack top holding the parsed phrases "... Products * Value". The reduce step replaces this instance of the rule's right hand side, "Products * Value" by the rule's left hand side symbol, here a larger Products. If the parser builds complete parse trees, the three trees for inner Products, *, and Value are combined by a new tree root for Products. Otherwise, [semantic](#) details from the inner Products and Value are output to some later compiler pass, or are combined and saved in the new Products symbol.^[4]

LR parse steps for example $A*2 + 1$ [\[edit\]](#)

In LR parsers, the shift and reduce decisions are potentially based on the entire stack of everything that has been previously parsed, not just on a single, topmost stack symbol. If done in an unclever way, that could lead to very slow parsers that get slower and slower for longer inputs. LR parsers do this with constant speed, by summarizing all the relevant left context information into a single number called the LR(0) **parser state**. For each grammar and LR analysis method, there is a fixed (finite) number of such states. Besides holding the already-parsed symbols, the parse stack also remembers the state numbers reached by everything up to those points.

At every parse step, the entire input text is divided into a stack of previously parsed phrases, and a current lookahead symbol, and the remaining unscanned text. The parser's next action is determined by its current LR(0) **state number** (rightmost on the stack) and the lookahead symbol. In the steps below, all the black details are exactly the same as in other non-LR shift-reduce parsers. LR parser stacks add the state information in purple, summarizing the black phrases to their left on the stack and what syntax possibilities to expect next. Users of an LR parser can usually ignore state information. These states are explained in a later section.

Step	Parse Stack <i>state</i> [Symbol <i>state</i>]*	Look Ahead	Unscanned	Parser Action	Grammar Rule	Next State
0	<i>0</i>	<i>id</i>	$*2 + 1$	shift		<i>9</i>
1	<i>0 id</i> <i>9</i>	*	$2 + 1$	reduce	$\text{Value} \rightarrow id$	<i>7</i>
2	<i>0 Value</i> <i>7</i>	*	$2 + 1$	reduce	$\text{Products} \rightarrow \text{Value}$	<i>4</i>

3	0 Products ₄	*	2 + 1	shift		5
4	0 Products ₄ * ₅	int	+ 1	shift		8
5	0 Products ₄ * ₅ int ₈	+	1	reduce	Value → int	6
6	0 Products ₄ * ₅ Value ₆	+	1	reduce	Products → Products * Value	4
7	0 Products ₄	+	1	reduce	Sums → Products	1
8	0 Sums ₁	+	1	shift		2
9	0 Sums ₁ + ₂	int	eof	shift		8
10	0 Sums ₁ + ₂ int ₈	eof		reduce	Value → int	7
11	0 Sums ₁ + ₂ Value ₇	eof		reduce	Products → Value	3
12	0 Sums ₁ + ₂ Products ₃	eof		reduce	Sums → Sums + Products	1
13	0 Sums ₁	eof		done		

At initial step 0, the input stream "A*2 + 1" is divided into

- an empty section on the parse stack,
- lookahead text "A" scanned as an *id* symbol, and
- the remaining unscanned text "*2 + 1".

The parse stack begins by holding only initial state 0. When state 0 sees the lookahead *id*, it knows to shift that *id* onto the stack, and scan the next input symbol *, and advance to state 9.

At step 4, the total input stream "A*2 + 1" is currently divided into

- the parsed section "A *" with 2 stacked phrases Products and *;
- lookahead text "2" scanned as an *int* symbol, and
- the remaining unscanned text " + 1".

The states corresponding to the stacked phrases are 0, 4, and 5. The current, rightmost state on the stack is state 5. When state 5 sees the lookahead *int*, it knows to shift that *int* onto the stack as its own phrase, and scan the next input symbol +, and advance to state 8.

At step 12, all of the input stream has been consumed but only partially organized. The current state is 3. When state 3 sees the lookahead *eof*, it knows to apply the completed grammar rule

Sums → Sums + Products

by combining the stack's rightmost three phrases for Sums, +, and Products into one thing. State 3 itself doesn't know what the next state should be. This is found by going back to state 0, just to the left of the phrase being reduced. When state 0 sees this new completed instance of a Sums, it advances to state 1 (again). This consulting of older states is why they are kept on the stack, instead of keeping only the current state.

Grammar for the example A*2 + 1 [\[edit\]](#)

LR parsers are constructed from a grammar that formally defines the syntax of the input language as a set of patterns. The grammar doesn't cover all language rules, such as the size of numbers, or the consistent use of names and their definitions in the context of the whole program. LR parsers use a [context-free grammar](#) that deals just with local patterns of symbols.

The example grammar used here is a tiny subset of the Java or C language:

r0: Goal → Sums eof
r1: Sums → Sums + Products
r2: Sums → Products
r3: Products → Products * Value
r4: Products → Value
r5: Value → int

r6: Value $\rightarrow id$

The grammar's [terminal symbols](#) are the multi-character symbols or 'tokens' found in the input stream by a [lexical scanner](#). Here these include $+$ $*$ and *int* for any integer constant, and *id* for any identifier name, and *eof* for end of input file. The grammar doesn't care what the *int* values or *id* spellings are, nor does it care about blanks or line breaks. The grammar uses these terminal symbols but does not define them. They are always leaf nodes (at the bottom bushy end) of the parse tree.

The capitalized terms like Sums are [nonterminal symbols](#). These are names for concepts or patterns in the language. They are defined in the grammar and never occur themselves in the input stream. They are always internal nodes (above the bottom) of the parse tree. They only happen as a result of the parser applying some grammar rule. Some terminals are defined with two or more rules; these are alternative patterns. Rules can refer back to themselves, which are called *recursive*. This grammar uses recursive rules to handle repeated math operators. Grammars for complete languages use recursive rules to handle lists, parenthesized expressions, and nested statements.

Any given computer language can be described by several different grammars. An LR(1) parser can handle many but not all common grammars. It is usually possible to manually modify a grammar so that it fits the limitations of LR(1) parsing and the generator tool.

The grammar for an LR parser must be [unambiguous](#) itself, or must be augmented by tie-breaking precedence rules. This means there is only one correct way to apply the grammar to a given legal example of the language, resulting in a unique parse tree with just one meaning, and a unique sequence of shift/reduce actions for that example. LR parsing is not a useful technique for human languages with ambiguous grammars that depend on the interplay of words. Human languages are better handled by parsers like [Generalized LR parser](#), the [Earley parser](#), or the [CYK algorithm](#) that can simultaneously compute all possible parse trees in one pass.

Parse table for the example grammar [\[edit\]](#)

Most LR parsers are table driven. The parser's program code is a simple generic loop that is the same for all grammars and languages. The knowledge of the grammar and its syntactic implications are encoded into unchanging data tables called **parse tables**. Entries in a table show whether to shift or reduce (and by which grammar rule), for every legal combination of parser state and lookahead symbol. The parse tables also tell how to compute the next state, given just a current state and a next symbol.

The parse tables are much larger than the grammar. LR tables are hard to accurately compute by hand for big grammars. So they are mechanically derived from the grammar by some [parser generator](#) tool like [Bison](#).^[5]

Depending on how the states and parsing table are generated, the resulting parser is called either a [SLR \(simple LR\) parser](#), [LALR \(look-ahead LR\) parser](#), or [canonical LR parser](#). LALR parsers handle more grammars than SLR parsers. Canonical LR parsers handle even more grammars, but use many more states and much larger tables. The example grammar is SLR.

LR parse tables are two-dimensional. Each current LR(0) parser state has its own row. Each possible next symbol has its own column. Some combinations of state and next symbol are not possible for valid input streams. These blank cells trigger syntax error messages.

The **Action** left half of the table has columns for lookahead terminal symbols. These cells determine whether the next parser action is shift (to state *n*), or reduce (by grammar rule *r_n*).

The **Goto** right half of the table has columns for nonterminal symbols. These cells show which state to advance to, after some reduction's Left Hand Side has created an expected new instance of that symbol. This is like a shift action but for nonterminals; the lookahead terminal symbol is unchanged.

The table column "Current Rules" documents the meaning and syntax possibilities for each state, as worked out by the parser generator. It is not included in the actual tables used at parsing time. The \bullet (pink dot) marker shows where the parser is now, within some partially recognized grammar rules. The things to the left of \bullet have been parsed, and the things to the right are expected soon. A state has several such current rules if the parser has not yet narrowed possibilities down to a single rule.

Curr		Lookahead					LHS Goto		
State	Current Rules	<i>int</i>	<i>id</i>	$*$	$+$	<i>eof</i>	Sums	Products	Value
0	Goal $\rightarrow \bullet$ Sums <i>eof</i>	8	9				1	4	7
1	Goal \rightarrow Sums \bullet <i>eof</i> Sums \rightarrow Sums \bullet $+$ Products				2	done			

2	Sums → Sums + • Products	8	9					3	7
3	Sums → Sums + Products • Products → Products • * Value			5	r1	r1			
4	Sums → Products • Products → Products • * Value			5	r2	r2			
5	Products → Products * • Value	8	9						6
6	Products → Products * Value •			r3	r3	r3			
7	Products → Value •			r4	r4	r4			
8	Value → int •			r5	r5	r5			
9	Value → id •			r6	r6	r6			

In state 2 above, the parser has just found and shifted-in the + of grammar rule

r1: Sums → Sums + • Products

The next expected phrase is Products. Products begins with terminal symbols *int* or *id*. If the lookahead is either of those, the parser shifts them in and advances to state 8 or 9, respectively. When a Products has been found, the parser advances to state 3 to accumulate the complete list of summands and find the end of rule r0. A Products can also begin with nonterminal Value. For any other lookahead or nonterminal, the parser announces a syntax error.

In state 3, the parser has just found a Products phrase, that could be from two possible grammar rules:

r1: Sums → Sums + Products •

r3: Products → Products • * Value

The choice between r1 and r3 can't be decided just from looking backwards at prior phrases. The parser has to check the lookahead symbol to tell what to do. If the lookahead is *, it is in rule 3, so the parser shifts in the * and advances to state 5. If the lookahead is eof, it is at the end of rule 1 and rule 0, so the parser is done.

In state 9 above, all the non-blank, non-error cells are for the same reduction r6. Some parsers save time and table space by not checking the lookahead symbol in these simple cases. Syntax errors are then detected somewhat later, after some harmless reductions, but still before the next shift action or parser decision.

Individual table cells must not hold multiple, alternative actions, otherwise the parser would be nondeterministic with guesswork and backtracking. If the grammar is not LR(1), some cells will have shift/reduce conflicts between a possible shift action and reduce action, or reduce/reduce conflicts between multiple grammar rules. LR(k) parsers resolve these conflicts (where possible) by checking additional lookahead symbols beyond the first.

LR parser loop [\[edit\]](#)

The LR parser begins with a nearly empty parse stack containing just the start state 0, and with the lookahead holding the input stream's first scanned symbol. The parser then repeats the following loop step until done, or stuck on a syntax error:

The topmost state on the parse stack is some state *s*, and the current lookahead is some terminal symbol *t*. Look up the next parser action from row *s* and column *t* of the Lookahead Action table. That action is either Shift, Reduce, Done, or Error:

- Shift *n*:
Shift the matched terminal *t* onto the parse stack and scan the next input symbol into the lookahead buffer.
Push next state *n* onto the parse stack as the new current state.
- Reduce *r_m*: Apply grammar rule *r_m*: Lhs → S₁ S₂ ... S_L
Remove the matched topmost L symbols (and parse trees and associated state numbers) from the parse stack.
This exposes a prior state *p* that was expecting an instance of the Lhs symbol.
Join the L parse trees together as one parse tree with new root symbol Lhs.
Look up the next state *n* from row *p* and column *Lhs* of the LHS Goto table.

Push the symbol and tree for Lhs onto the parse stack.
Push next state n onto the parse stack as the new current state.
The lookahead and input stream remain unchanged.

- Done: Lookahead t is the *eof* marker. End of parsing. If the state stack contains just the start state report success. Otherwise, report a syntax error.
- No action: Report a syntax error. The parser ends, or attempts some recovery.

Note: LR parser stack usually stores just the LR(0) automaton states, as the grammar symbols may be derived from them (in the automaton, all input transitions to some state are marked with the same symbol, which is the symbol associated with this state). Moreover these symbols are almost never needed as the state is all that matters when making the parsing decision.^[6]

LR generator analysis [\[edit\]](#)

This section of the article can be skipped by most users of LR parser generators.

LR states [\[edit\]](#)

State 2 in the example parse table is for the partially parsed rule

r1: Sums \rightarrow Sums + • Products

This shows how the parser got here, by seeing Sums then + while looking for a larger Sums. The • marker has advanced beyond the beginning of the rule. It also shows how the parser expects to eventually complete the rule, by next finding a complete Products. But more details are needed on how to parse all the parts of that Products.

The partially parsed rules for a state are called its "core LR(0) items". The parser generator adds additional rules or items for all the possible next steps in building up the expected Products:

r3: Products \rightarrow • Products * Value

r4: Products \rightarrow • Value

r5: Value \rightarrow • *int*

r6: Value \rightarrow • *id*

Note that the • marker is at the beginning of each of these added rules; the parser has not yet confirmed and parsed any part of them. These additional items are called the "closure" of the core items. For each nonterminal symbol immediately following a •, the generator adds the rules defining that symbol. This adds more • markers, and possibly different follower symbols. This closure process continues until all follower symbols have been expanded. The follower nonterminals for state 2 begins with Products. Value is then added by closure. The follower terminals are *int* and *id*.

The kernel and closure items together show all possible legal ways to proceed from the current state to future states and complete phrases. If a follower symbol appears in only one item, it leads to a next state containing only one core item with the • marker advanced. So *int* leads to next state 8 with core

r5: Value \rightarrow *int* •

If the same follower symbol appears in several items, the parser cannot yet tell which rule applies here. So that symbol leads to a next state that shows all remaining possibilities, again with the • marker advanced. Products appears in both r1 and r3. So Products leads to next state 4 with core

r1: Sums \rightarrow Sums + Products •

r3: Products \rightarrow Products • * Value

In words, that means if the parser has seen a single Products, it might be done, or it might still have even more things to multiply together. Note that all the core items have the same symbol preceding the • marker; all transitions into this state are always with that same symbol.

Some transitions will be to cores and states that have been enumerated already. Other transitions lead to new states. The generator starts with the grammar's goal rule. From there it keeps exploring known states and transitions until all needed states have been found.

These states are called "LR(0)" states because they use a lookahead of $k=0$, i.e. no lookahead. The only checking of input symbols occurs when the symbol is shifted in. Checking of lookaheads for reductions is done

separately by the parse table, not by the enumerated states themselves.

Finite state machine [\[edit\]](#)

The parse table describes all possible LR(0) states and their transitions. They form a [finite state machine](#) (FSM). An FSM is a simple engine for parsing simple unnested languages, without using a stack. In this LR application, the FSM's modified "input language" has both terminal and nonterminal symbols, and covers any partially parsed stack snapshot of the full LR parse.

Recall step 5 of the Parse Steps Example:

Step	Parse Stack	Look	Unscanned
	state Symbol state ...	Ahead	
5	0 Products ₄ * ₅ int ₈	+	1

The parse stack shows a series of state transitions, from the start state 0, to state 4 and then on to 5 and current state 8. The symbols on the parse stack are the shift or goto symbols for those transitions. Another way to view this, is that the finite state machine can scan the stream "Products * int + 1" (without using yet another stack) and find the leftmost complete phrase that should be reduced next. And that is indeed its job!

How can a mere FSM do this, when the original unparsed language has nesting and recursion and definitely requires an analyzer with a stack? The trick is that everything to the left of the stack top has already been fully reduced. This eliminates all the loops and nesting from those phrases. The FSM can ignore all the older beginnings of phrases, and track just the newest phrases that might be completed next. The obscure name for this in LR theory is "viable prefix".

Lookahead sets [\[edit\]](#)

The states and transitions give all the needed information for the parse table's shift actions and goto actions. The generator also needs to calculate the expected lookahead sets for each reduce action.

In **SLR** parsers, these lookahead sets are determined directly from the grammar, without considering the individual states and transitions. For each nonterminal S, the SLR generator works out Follows(S), the set of all the terminal symbols which can immediately follow some occurrence of S. In the parse table, each reduction to S uses Follow(S) as its LR(1) lookahead set. Such follow sets are also used by generators for LL top-down parsers. A grammar that has no shift/reduce or reduce/reduce conflicts when using Follow sets is called an SLR grammar.

LALR parsers have the same states as SLR parsers, but use a more complicated, more precise way of working out the minimum necessary reduction lookaheads for each individual state. Depending on the details of the grammar, this may turn out to be the same as the Follow set computed by SLR parser generators, or it may turn out to be a subset of the SLR lookaheads. Some grammars are okay for LALR parser generators but not for SLR parser generators. This happens when the grammar has spurious shift/reduce or reduce/reduce conflicts using Follow sets, but no conflicts when using the exact sets computed by the LALR generator. The grammar is then called LALR(1) but not SLR.

An SLR or LALR parser avoids having duplicate states. But this minimization is not necessary, and can sometimes create unnecessary lookahead conflicts. **Canonical LR** parsers use duplicated (or "split") states to better remember the left and right context of a nonterminal's use. Each occurrence of a symbol S in the grammar can be treated independently with its own lookahead set, to help resolve reduction conflicts. This handles a few more grammars. Unfortunately, this greatly magnifies the size of the parse tables if done for all parts of the grammar. This splitting of states can also be done manually and selectively with any SLR or LALR parser, by making two or more named copies of some nonterminals. A grammar that is conflict-free for a canonical LR generator but has conflicts in an LALR generator is called LR(1) but not LALR(1), and not SLR.

SLR, LALR, and canonical LR parsers make exactly the same shift and reduce decisions when the input stream is correct language. When the input has a syntax error, the LALR parser may do some additional (harmless) reductions before detecting the error than would the canonical LR parser. And the SLR parser may do even more. This happens because the SLR and LALR parsers are using a generous superset approximation to the true, minimal lookahead symbols for that particular state.

Syntax error recovery [\[edit\]](#)

LR parsers can generate somewhat helpful error messages for the first syntax error in a program, by simply enumerating all the terminal symbols that could have appeared next instead of the unexpected bad lookahead

symbol. But this does not help the parser work out how to parse the remainder of the input program to look for further, independent errors. If the parser recovers badly from the first error, it is very likely to mis-parse everything else and produce a cascade of unhelpful spurious error messages.

In the [yacc](#) and bison parser generators, the parser has an ad hoc mechanism to abandon the current statement, discard some parsed phrases and lookahead tokens surrounding the error, and resynchronize the parse at some reliable statement-level delimiter like semicolons or braces. This often works well for allowing the parser and compiler to look over the rest of the program.

Many syntactic coding errors are simple typos or omissions of a trivial symbol. Some LR parsers attempt to detect and automatically repair these common cases. The parser enumerates every possible single-symbol insertion, deletion, or substitution at the error point. The compiler does a trial parse with each change to see if it worked okay. (This requires backtracking to snapshots of the parse stack and input stream, normally unneeded by the parser.) Some best repair is picked. This gives a very helpful error message and resynchronizes the parse well. However, the repair is not trustworthy enough to permanently modify the input file. Repair of syntax errors is easiest to do consistently in parsers (like LR) that have parse tables and an explicit data stack.

Variants of LR parsers [\[edit\]](#)

The LR parser generator decides what should happen for each combination of parser state and lookahead symbol. These decisions are usually turned into read-only data tables that drive a generic parser loop that is grammar- and state-independent. But there are also other ways to turn those decisions into an active parser.

Some LR parser generators create separate tailored program code for each state, rather than a parse table. These parsers can run several times faster than the generic parser loop in table-driven parsers. The fastest parsers use generated assembler code.

In the [recursive ascent parser](#) variation, the explicit parse stack structure is also replaced by the implicit stack used by subroutine calls. Reductions terminate several levels of subroutine calls, which is clumsy in most languages. So recursive ascent parsers are generally slower, less obvious, and harder to hand-modify than [recursive descent parsers](#).

Another variation replaces the parse table by pattern-matching rules in non-procedural languages such as [Prolog](#).

GLR [Generalized LR parsers](#) use LR bottom-up techniques to find all possible parses of input text, not just one correct parse. This is essential for ambiguous grammars such as used for human languages. The multiple valid parse trees are computed simultaneously, without backtracking. GLR is sometimes helpful for computer languages that are not easily described by a conflict-free LALR(1) grammar.

[Left corner parsers](#) use LR bottom-up techniques for recognizing the left end of alternative grammar rules. When the alternatives have been narrowed down to a single possible rule, the parser then switches to top-down LL(1) techniques for parsing the rest of that rule. LC parsers have smaller parse tables than LALR parsers and better error diagnostics. There are no widely used generators for deterministic LC parsers. Multiple-parse LC parsers are helpful with human languages with very large grammars.

Theory [\[edit\]](#)

LR parsers were invented by [Donald Knuth](#) in 1965 as an efficient generalization of [precedence parsers](#). Knuth proved that LR parsers were the most general-purpose parsers possible that would still be efficient in the worst cases.

"LR(k) grammars can be efficiently parsed with an execution time essentially proportional to the length of the string."^[7]

For every $k \geq 1$, "a language can be generated by an LR(k) grammar if and only if it is deterministic [and context-free], if and only if it can be generated by an LR(1) grammar."^[8]

In other words, if a language was reasonable enough to allow an efficient one-pass parser, it could be described by an LR(k) grammar. And that grammar could always be mechanically transformed into an equivalent (but larger) LR(1) grammar. So an LR(1) parsing method was, in theory, powerful enough to handle any reasonable language. In practice, the natural grammars for many programming languages are close to being LR(1).^[citation needed]

The canonical LR parsers described by Knuth had too many states and very big parse tables that were impractically large for the limited memory of computers of that era. LR parsing became practical when [Frank DeRemer](#) invented [SLR](#) and [LALR](#) parsers with much fewer states.^{[9][10]}

For full details on LR theory and how LR parsers are derived from grammars, see *The Theory of Parsing*,

Earley parsers apply the techniques and \bullet notation of LR parsers to the task of generating all possible parses for ambiguous grammars such as for human languages.

While $LR(k)$ grammars have equal generative power for all $k \geq 1$, the case of $LR(0)$ grammars is slightly different. A language L is said to have the *prefix property* if no word in L is a **proper prefix** of another word in L .^[12] A language L has an $LR(0)$ grammar if and only if L is a **deterministic context-free language** with the prefix property.^[13] As a consequence, a language L is deterministic context-free if and only if $L\$$ has an $LR(0)$ grammar, where "\$" is not a symbol of L 's **alphabet**.^[14]

Additional example 1+1 [\[edit\]](#)

This example of LR parsing uses the following small grammar with goal symbol E :

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

to parse the following input:

1 + 1

Action and goto tables [\[edit\]](#)

The two $LR(0)$ parsing tables for this grammar look as follows:

state	action					goto	
	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

The **action table** is indexed by a state of the parser and a terminal (including a special terminal \$ that indicates the end of the input stream) and contains three types of actions:

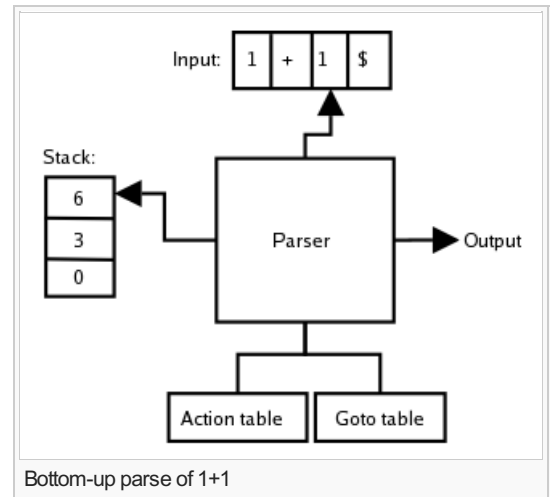
- *shift*, which is written as 'sn' and indicates that the next state is n
- *reduce*, which is written as 'rm' and indicates that a reduction with grammar rule m should be performed
- *accept*, which is written as 'acc' and indicates that the parser accepts the string in the input stream.

The **goto table** is indexed by a state of the parser and a nonterminal and simply indicates what the next state of the parser will be if it has recognized a certain nonterminal. This table is important to find out the next state after every reduction. After a reduction, the next state is found by looking up the **goto table** entry for top of the stack (i.e. current state) and the reduced rule's LHS (i.e. non-terminal).

Parsing steps [\[edit\]](#)

The table below illustrates each step in the process. Here the state refers to the element at the top of the stack (the right-most element), and the next action is determined by referring to the action table above. Also note that a \$ is appended to the input string to denote the end of the stream.

State	Input stream	Output stream	Stack	Next action
0	1+1\$		[0]	Shift 2



2	+1\$		[0,2]	Reduce 5
4	+1\$	5	[0,4]	Reduce 3
3	+1\$	5,3	[0,3]	Shift 6
6	1\$	5,3	[0,3,6]	Shift 2
2	\$	5,3	[0,3,6,2]	Reduce 5
8	\$	5,3,5	[0,3,6,8]	Reduce 2
3	\$	5,3,5,2	[0,3]	Accept

Walkthrough [\[edit\]](#)

The parser starts out with the stack containing just the initial state ('0'):

[0]

The first symbol from the input string that the parser sees is '1'. In order to find out what the next action is (shift, reduce, accept or error), the action table is indexed with the current state (remember that the "current state" is just whatever is on the top of the stack), which in this case is 0, and the current input symbol, which is '1'. The action table specifies a shift to state 2, and so state 2 is pushed onto the stack (again, remember that all the state information is in the stack, so "shifting to state 2" is the same thing as pushing 2 onto the stack). The resulting stack is

[0 '1' 2]

where the top of the stack is 2. For the sake of explanation the symbol (e.g., '1', B) is shown that caused the transition to the next state, although strictly speaking it is not part of the stack.

In state 2 the action table says that regardless of what terminal the parser sees on the input stream, it should do a reduction with grammar rule 5. If the table is correct, this means that the parser has just recognized the right-hand side of rule 5, which is indeed the case. In this case the parser writes 5 to the output stream, pops one state from the stack (since the right-hand side of the rule has one symbol), and pushes on the stack the state from the cell in the goto table for state 0 and B, i.e., state 4. The resulting stack is:

[0 B 4]

However, in state 4 the action table says the parser should now do a reduction with rule 3. So it writes 3 to the output stream, pops one state from the stack, and finds the new state in the goto table for state 0 and E, which is state 3. The resulting stack:

[0 E 3]

The next terminal that the parser sees is a '+' and according to the action table it should then go to state 6:

[0 E 3 '+' 6]

Note that the resulting stack can be interpreted as the history of a [finite state automaton](#) that has just read a nonterminal E followed by a terminal '+'. The transition table of this automaton is defined by the shift actions in the action table and the goto actions in the goto table.

The next terminal is now '1' and this means that the parser performs a shift and go to state 2:

[0 E 3 '+' 6 '1' 2]

Just as the previous '1' this one is reduced to B giving the following stack:

[0 E 3 '+' 6 B 8]

Again note that the stack corresponds with a list of states of a finite automaton that has read a nonterminal E, followed by a '+' and then a nonterminal B. In state 8 the parser always performs a reduce with rule 2. Note that the top 3 states on the stack correspond with the 3 symbols in the right-hand side of rule 2.

[0 E 3]

Finally, the parser reads a '\$' (end of input symbol) from the input stream, which means that according to the action table (the current state is 3) the parser accepts the input string. The rule numbers that will then have been written to the output stream will be [5, 3, 5, 2] which is indeed a [rightmost derivation](#) of the string "1 + 1" in reverse.

Constructing LR(0) parsing tables [\[edit\]](#)

Items [\[edit\]](#)

The construction of these parsing tables is based on the notion of $LR(0)$ items (simply called *items* here) which are grammar rules with a special dot added somewhere in the right-hand side. For example the rule $E \rightarrow E + B$ has the following four corresponding items:

$E \rightarrow \cdot E + B$
 $E \rightarrow E \cdot + B$
 $E \rightarrow E + \cdot B$
 $E \rightarrow E + B \cdot$

Rules of the form $A \rightarrow \epsilon$ have only a single item $A \rightarrow \cdot$. The item $E \rightarrow E \cdot + B$, for example, indicates that the parser has recognized a string corresponding with E on the input stream and now expects to read a '+' followed by another string corresponding with B .

Item sets [\[edit\]](#)

It is usually not possible to characterize the state of the parser with a single item because it may not know in advance which rule it is going to use for reduction. For example if there is also a rule $E \rightarrow E * B$ then the items $E \rightarrow E \cdot + B$ and $E \rightarrow E \cdot * B$ will both apply after a string corresponding with E has been read. Therefore it is convenient to characterize the state of the parser by a set of items, in this case the set $\{ E \rightarrow E \cdot + B, E \rightarrow E \cdot * B \}$.

Extension of Item Set by expansion of non-terminals [\[edit\]](#)

An item with a dot before a nonterminal, such as $E \rightarrow E + \cdot B$, indicates that the parser expects to parse the nonterminal B next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must include all items describing how B itself will be parsed. This means that if there are rules such as $B \rightarrow 1$ and $B \rightarrow 0$ then the item set must also include the items $B \rightarrow \cdot 1$ and $B \rightarrow \cdot 0$. In general this can be formulated as follows:

If there is an item of the form $A \rightarrow v \cdot Bw$ in an item set and in the grammar there is a rule of the form $B \rightarrow w'$ then the item $B \rightarrow \cdot w'$ should also be in the item set.

Closure of item sets [\[edit\]](#)

Thus, any set of items can be extended by recursively adding all the appropriate items until all nonterminals preceded by dots are accounted for. The minimal extension is called the *closure* of an item set and written as $\text{clos}(I)$ where I is an item set. It is these closed item sets that are taken as the states of the parser, although only the ones that are actually reachable from the begin state will be included in the tables.

Augmented grammar [\[edit\]](#)

Before the transitions between the different states are determined, the grammar is augmented with an extra rule

(0) $S \rightarrow E$

where S is a new start symbol and E the old start symbol. The parser will use this rule for reduction exactly when it has accepted the whole input string.

For this example, the same grammar as above is augmented thus:

(0) $S \rightarrow E$
(1) $E \rightarrow E * B$
(2) $E \rightarrow E + B$
(3) $E \rightarrow B$
(4) $B \rightarrow 0$
(5) $B \rightarrow 1$

It is for this augmented grammar that the item sets and the transitions between them will be determined.

Table construction [\[edit\]](#)

Finding the reachable item sets and the transitions between them [\[edit\]](#)

The first step of constructing the tables consists of determining the transitions between the closed item sets. These transitions will be determined as if we are considering a finite automaton that can read terminals as well

as nonterminals. The begin state of this automaton is always the closure of the first item of the added rule: $S \rightarrow$

• E:

Item set 0

$S \rightarrow \bullet E$

$+ E \rightarrow \bullet E * B$

$+ E \rightarrow \bullet E + B$

$+ E \rightarrow \bullet B$

$+ B \rightarrow \bullet 0$

$+ B \rightarrow \bullet 1$

The **boldfaced** "+" in front of an item indicates the items that were added for the closure (not to be confused with the mathematical '+' operator which is a terminal). The original items without a "+" are called the *kernel* of the item set.

Starting at the begin state (S_0), all of the states that can be reached from this state are now determined. The possible transitions for an item set can be found by looking at the symbols (terminals and nonterminals) found following the dots; in the case of item set 0 those symbols are the terminals '0' and '1' and the nonterminals E and B. To find the item set that each symbol x leads to, the following procedure is followed for each of the symbols:

1. Take the subset, S , of all items in the current item set where there is a dot in front of the symbol of interest, x .
2. For each item in S , move the dot to the right of x .
3. Close the resulting set of items.

For the terminal '0' (i.e. where $x = '0'$) this results in:

Item set 1

$B \rightarrow 0 \bullet$

and for the terminal '1' (i.e. where $x = '1'$) this results in:

Item set 2

$B \rightarrow 1 \bullet$

and for the nonterminal E (i.e. where $x = E$) this results in:

Item set 3

$S \rightarrow E \bullet$

$E \rightarrow E \bullet * B$

$E \rightarrow E \bullet + B$

and for the nonterminal B (i.e. where $x = B$) this results in:

Item set 4

$E \rightarrow B \bullet$

Note that the closure does not add new items in all cases - in the new sets above, for example, there are no nonterminals following the dot. This process is continued until no more new item sets are found. For the item sets 1, 2, and 4 there will be no transitions since the dot is not in front of any symbol. For item set 3, note that the dot is in front of the terminals '*' and '+'. For '*' the transition goes to:

Item set 5

$E \rightarrow E * \bullet B$

$+ B \rightarrow \bullet 0$

$+ B \rightarrow \bullet 1$

and for '+' the transition goes to:

Item set 6

$E \rightarrow E + \bullet B$

$+ B \rightarrow \bullet 0$

$+ B \rightarrow \bullet 1$

For item set 5, the terminals '0' and '1' and the nonterminal B must be considered. For the terminals, note that

the resulting closed item sets are equal to the already found item sets 1 and 2, respectively. For the nonterminal B the transition goes to:

Item set 7

$E \rightarrow E * B \cdot$

For item set 6, the terminal '0' and '1' and the nonterminal B must be considered. As before, the resulting item sets for the terminals are equal to the already found item sets 1 and 2. For the nonterminal B the transition goes to:

Item set 8

$E \rightarrow E + B \cdot$

These final item sets have no symbols beyond their dots so no more new item sets are added, so the item generating procedure is complete. The finite automaton, with item sets as its states is shown below.

The transition table for the automaton now looks as follows:

Item Set	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

Constructing the action and goto tables [\[edit\]](#)

From this table and the found item sets, the action and goto table are constructed as follows:

1. The columns for nonterminals are copied to the goto table.
2. The columns for the terminals are copied to the action table as shift actions.
3. An extra column for '\$' (end of input) is added to the action table that contains *acc* for every item set that contains $S \rightarrow E \cdot$.
4. If an item set i contains an item of the form $A \rightarrow w \cdot$ and $A \rightarrow w$ is rule m with $m > 0$ then the row for state i in the action table is completely filled with the reduce action rm .

The reader may verify that this results indeed in the action and goto table that were presented earlier on.

A note about LR(0) versus SLR and LALR parsing [\[edit\]](#)

Note that only step 4 of the above procedure produces reduce actions, and so all reduce actions must occupy an entire table row, causing the reduction to occur regardless of the next symbol in the input stream. This is why these are LR(0) parse tables: they don't do any lookahead (that is, they look ahead zero symbols) before deciding which reduction to perform. A grammar that needs lookahead to disambiguate reductions would require a parse table row containing different reduce actions in different columns, and the above procedure is not capable of creating such rows.

Refinements to the LR(0) table construction procedure (such as SLR and LALR) are capable of constructing reduce actions that do not occupy entire rows. Therefore, they are capable of parsing more grammars than LR(0) parsers.

Conflicts in the constructed tables [\[edit\]](#)

The automaton is constructed in such a way that it is guaranteed to be deterministic. However, when reduce actions are added to the action table it can happen that the same cell is filled with a reduce action and a shift action (a *shift-reduce conflict*) or with two different reduce actions (a *reduce-reduce conflict*). However, it can be shown that when this happens the grammar is not an LR(0) grammar. A classic real-world example of a shift-reduce conflict is the [dangling else](#) problem.

A small example of a non-LR(0) grammar with a shift-reduce conflict is:

- (1) $E \rightarrow 1 E$
- (2) $E \rightarrow 1$

One of the item sets found is:

Item set 1

- $E \rightarrow 1 \cdot E$
- $E \rightarrow 1 \cdot$
- $+ E \rightarrow \cdot 1 E$
- $+ E \rightarrow \cdot 1$

There is a shift-reduce conflict in this item set because in the cell in the action table for this item set and the terminal '1' there will be both a shift action to state 1 and a reduce action with rule 2.

A small example of a non-LR(0) grammar with a reduce-reduce conflict is:

- (1) $E \rightarrow A 1$
- (2) $E \rightarrow B 2$
- (3) $A \rightarrow 1$
- (4) $B \rightarrow 1$

In this case the following item set is obtained:

Item set 1

- $A \rightarrow 1 \cdot$
- $B \rightarrow 1 \cdot$


There is a reduce-reduce conflict in this item set because in the cells in the action table for this item set there will be both a reduce action for rule 3 and one for rule 4.

Both examples above can be solved by letting the parser use the follow set (see [LL parser](#)) of a nonterminal A to decide if it is going to use one of A 's rules for a reduction; it will only use the rule $A \rightarrow w$ for a reduction if the next symbol on the input stream is in the follow set of A . This solution results in so-called [Simple LR parsers](#).

See also [\[edit\]](#)

- [Canonical LR parser](#)
- [Simple LR](#)
- [Look-Ahead LR](#)
- [Generalized LR](#)

References [\[edit\]](#)

1. ^{[a](#)} ^{[b](#)} Knuth, D. E. (July 1965). "On the translation of languages from left to right"  (PDF). *Information and Control* **8** (6): 607–639. doi:10.1016/S0019-9958(65)90426-2 [↗](#). Retrieved 29 May 2011.
2. ^{[a](#)} [Language theoretic comparison of LL and LR grammars](#)
3. ^{[a](#)} Engineering a Compiler (2nd edition), by Keith Cooper and Linda Torczon, Morgan Kaufman 2011.
4. ^{[a](#)} Crafting and Compiler, by Charles Fischer, Ron Cytron, and Richard LeBlanc, Addison Wesley 2009.
5. ^{[a](#)} Flex & Bison: Text Processing Tools, by John Levine, O'Reilly Media 2009.
6. ^{[a](#)} ^{[b](#)} Compilers: Principles, Techniques, and Tools (2nd Edition), by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall 2006.
7. ^{[a](#)} Knuth (1965), p.638
8. ^{[a](#)} Knuth (1965), p.635. Knuth didn't mention the restriction $k \geq 1$ there, but it is required by his theorems he referred to, viz. on p.629 and p.630. Similarly, the restriction to [context-free languages](#) is tacitly understood from the context.
9. ^{[a](#)} Practical Translators for LR(k) Languages, by Frank DeRemer, MIT PhD dissertation 1969.
10. ^{[a](#)} Simple LR(k) Grammars, by Frank DeRemer, Comm. ACM 14:7 1971.
11. ^{[a](#)} The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing, by Alfred Aho and Jeffrey Ullman, Prentice Hall 1972.
12. ^{[a](#)} Hopcroft, John E.; Ullman, Jeffrey D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley. ISBN 0-201-02988-X. Here: Exercise 5.8, p.121.
13. ^{[a](#)} Hopcroft, Ullman (1979), Theorem 10.12, p.260
14. ^{[a](#)} Hopcroft, Ullman (1979), Corollary p.260

Further reading [edit]

- Chapman, Nigel P., *LR Parsing: Theory and Practice* [↗](#), Cambridge University Press, 1987. ISBN 0-521-30413-X
- Pager, D., A Practical General Method for Constructing LR(k) Parsers. Acta Informatica 7, 249 - 268 (1977)
- "Compiler Construction: Principles and Practice" by Kenneth C. Louden. ISBN 0-534-939724

External links [edit]

- [dickgrune.com](#) [↗](#), Parsing Techniques - A Practical Guide 1st Ed. web page of book includes downloadable pdf.
- [Parsing Simulator](#) [↗](#) This simulator is used to generate parsing tables LR and to resolve the exercises of the book
- [Internals of an LALR\(1\) parser generated by GNU Bison](#) [↗](#) - Implementation issues
- [Course notes on LR parsing](#) [↗](#)
- [Shift-reduce and Reduce-reduce conflicts in an LALR parser](#) [↗](#)
- [A LR parser example](#) [↗](#)
- [Practical LR\(k\) Parser Construction](#) [↗](#)
- [The Honalee LR\(k\) Algorithm](#) [↗](#)

Categories: [Parsing algorithms](#)

This page was last modified on 28 August 2015, at 13:58.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

