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
Српски / srpski
Edit links

# Canonical LR parser

From Wikipedia, the free encyclopedia

In computer science, a **canonical LR parser** or **LR(1) parser** is an LR(k) parser for *k=1*, i.e. with a single lookahead terminal. The special attribute of this parser is that all LR(k) parsers with *k>1* can be transformed into a LR(1) parser.[1] It can handle all deterministic context-free languages.[1] In the past this LR(k) parser has been avoided because of its huge memory requirements in favor of less powerful alternatives such as the LALR and the LL(1) parser. Recently, however, a "minimal LR(1) parser" whose space requirements are close to LALR parsers, is being offered by several parser generators.

Like most parsers, the LR(1) parser is automatically generated by compiler compilers like GNU Bison, MSTA, Menhir,[2] HYACC,[3] and LRSTAR.[4]

**Contents** [hide]



## History [edit]

In 1965 Donald Knuth invented the LR(k) parser (**L**eft to right, **R**ightmost derivation parser) a type of shift-reduce parser, as a generalization of existing precedence parsers. This parser has the potential of recognizing all deterministic context-free languages and can produce both left and right derivations of statements encountered in the input file. Knuth proved that it reaches its maximum language recognition power for k=1 and provided a method for transforming LR(k), k > 1 grammars into LR(1) grammars.[1]

Canonical LR(1) parsers have the practical disadvantage of having enormous memory requirements for their internal parser-table representation. In 1969, Frank DeRemer suggested two simplified versions of the LR parser called LALR and SLR. These parsers require much less memory than Canonical LR(1) parsers, but have slightly less language-recognition power.[5] LALR(1) parsers have been the most common implementations of the LR Parser.

However, a new type of LR(1) parser, some people call a "minimal LR(1) parser" was introduced in 1977 by David Pager [6] who showed that LR(1) parsers can be created whose memory requirements rival those of LALR(1) parsers. Recently, some parser generators are offering minimal LR(1) parsers, which not only solve the memory requirement problem, but also the mysterious-conflict-problem inherent in LALR(1) parser generators.

## Overview [edit]

The LR(1) parser is a deterministic automaton and as such its operation is based on static state transition tables. These codify the grammar of the language it recognizes and are typically called "parsing tables".

The parsing tables of the LR(1) parser are parameterized with a lookahead terminal. Simple parsing tables, like those used by the LR(0) parser represent grammar rules of the form

A1 → A, B

which means that if we go from state *A* to state *B* then we will go to state *A1*. After parameterizing such a rule with a lookahead we have:

A1 → A, B, a

which means that the transition will now be performed only if the lookahead terminal is *a*. This allows for richer languages where a simple rule can have different meanings depending on the lookahead context. For example, in a LR(1) grammar, all of the following rules transition to a different state in spite of being based on the same state sequence.

A1 → A, B, a
A2 → A, B, b
A3 → A, B, c
A4 → A, B, d

The same would not be true if a lookahead terminal was not being taken into account. Parsing errors can be identified without the parser having to read the whole input by declaring some rules as errors. For example

E1 → B, C, d

can be declared an error, causing the parser to stop. This means that the lookahead information can also be used to catch errors, as in the following example:

A1 → A, B, a
A1 → A, B, b
A1 → A, B, c
E1 → A, B, d

In this case A, B will be reduced to A1 when the lookahead is a, b or c and an error will be reported when the lookahead is d.

The lookahead can also be helpful in deciding when to reduce a rule. The lookahead can help avoid reducing a specific rule if the lookahead is not valid, which would probably mean that the current state should be combined with the following instead of the previous state. That means that in the following example

- State sequence: A, B, C
- Rules:
  A1 → A, B
  A2 → B, C

the state sequence can be reduced to

A, A2

instead of

A1, C

if the lookahead after the parser went to state B wasn't acceptable, i.e. no transition rule existed. It should be noted here that states can be produced directly from a terminal as in

X → y

which allows for state sequences to appear.

LR(1) parsers have the requirement that each rule should be expressed in a complete LR(1) manner, i.e. a sequence of two states with a specific lookahead. That makes simple rules such as

X → y

requiring a great many artificial rules that essentially enumerate the combinations of all the possible states and lookahead terminals that can follow. A similar problem appears for implementing non-lookahead rules such as

A1 → A, B

where all the possible lookaheads must be enumerated. That is the reason why LR(1) parsers cannot be practically implemented without significant memory optimizations.[6]

## Constructing LR(1) parsing tables [edit]

LR(1) parsing tables are constructed in the same way as LR(0) parsing tables with the modification that each Item contains a lookahead terminal. This means, contrary to LR(0) parsers, a different action may be executed, if the item to process is followed by a different terminal.

### Parser items [edit]

Starting from the production rules of a language, at first the item sets for this language have to be determined.

In plain words, an item set is the list of production rules, which the currently processed symbol might be part of. An item set has a one-to-one correspondence to a parser state, while the items within the set, together with the next symbol, are used to decide which state transitions and parser action are to be applied. Each item contains a marker, to note at which point the currently processed symbol appears in the rule the item represents. For LR(1) parsers, each item is specific to a lookahead terminal, thus the lookahead terminal has also been noted inside each item.

For example assume a language consisting of the terminal symbols 'n', '+', '(', ')', the nonterminals 'E', 'T', the starting rule 'S' and the following production rules:

S → E
E → T
E → ( E )
T → n
T → + T
T → T + n

Items sets will be generated by analog to the procedure for LR(0) parsers. The item set 0 which represents the initial state will be created from the starting rule:

[S → • E, $]

The dot '•' denotes the marker of the current parsing position within this rule. The expected lookahead terminal to apply this rule is noted after the comma. The '$' sign is used to denote 'end of input' is expected, as is the case for the starting rule.

This is not the complete item set 0, though. Each item set must be 'closed', which means all production rules for each nonterminal following a '•' have to be recursively included into the item set until all of those nonterminals are dealt with. The resulting item set is called the closure of the item set we began with.

For LR(1) for each production rule an item has to be included for each possible lookahead terminal following the rule. For more complex languages this usually results in very large item sets, which is the reason for the large memory requirements of LR(1) parsers.

In our example, the starting symbol requires the nonterminal 'E' which in turn requires 'T', thus all production rules will appear in item set 0. At first, we ignore the problem of finding the lookaheads and just look at the case of an LR(0), whose items do not contain lookahead terminals. So the item set 0 (without lookaheads) will look like this:

[S → • E]
[E → • T]
[E → • ( E )]
[T → • n]
[T → • + T]
[T → • T + n]

### FIRST and FOLLOW sets [edit]

To determine lookhead terminals, so-called FIRST and FOLLOW sets are used. FIRST(A) is the set of terminals which can appear as the first element of any chain of rules matching nonterminal A. FOLLOW(I) of an Item I [A → α • B β, x] is the set of terminals that can appear immediately after nonterminal B, where α, β are arbitrary symbol strings, and x is an arbitrary lookahead terminal. FOLLOW(k,B) of an item set k and a nonterminal B is the union of the follow sets of all items in k where '•' is followed by B. The FIRST sets can be determined directly from the closures of all nonterminals in the language, while the FOLLOW sets are determined from the items under usage of the FIRST sets.

In our example, as one can verify from the full list of item sets below, the first sets are:

FIRST(S) = { n, '+', '(' }
FIRST(E) = { n, '+', '(' }
FIRST(T) = { n, '+' }

### Determining lookahead terminals [edit]

Within item set 0 the follow sets can be found to be:

FOLLOW(0,S) = { $ }
FOLLOW(0,E) = { $, ')'}

FOLLOW(0,T) = { $, '+', ')' }

From this the full item set 0 for an LR(1) parser can be created, by creating for each item in the LR(0) item set one copy for each terminal in the follow set of the LHS nonterminal. Each element of the follow set may be a valid lookahead terminal:

[S → • E, $]
[E → • T, $]
[E → • ( E ), $]
[T → • n, $]
[T → • n, +]
[T → • + T, $]
[T → • + T, +]
[T → • T + n, $]
[T → • T + n, +]

### Creating new item sets [edit]

The rest of the item sets can be created by the following algorithm

1. For each terminal and nonterminal symbol A appearing after a '•' in each already existing item set k, create a new item set m by adding to m all the rules of k where '•' is followed by A, but only if m will not be the same as an already existing item set after step 3.
2. shift all the '•'s for each rule in the new item set one symbol to the right
3. create the closure of the new item set
4. Repeat from step 1 for all newly created item sets, until no more new sets appear

In the example we get 5 more sets from item set 0, item set 1 for nonterminal E, item set 2 for nonterminal T, item set 3 for terminal n, item set 4 for terminal '+' and item set 5 for '('.

Item set 1 (E):

[S → E •, $]

Item set 2 (T):

[E → T •, $]
[T → T • + n, $]
[T → T • + n, +]

Item set 3 (n):

[T → n •, $]
[T → n •, +]

Item set 4 ('+'):

[T → + • T, $]
[T → + • T, +]
[T → • n, $]
[T → • n, +]
[T → • + T, $]
[T → • + T, +]
[T → • T + n, $]
[T → • T + n, +]

Item set 5 ('('):

[E → ( • E ), $]
[E → • T, )]
[E → • ( E ), )]
[T → • n, )]
[T → • n, +]
[T → • + T, )]
[T → • + T, +]
[T → • T + n, )]
[T → • T + n, +]

From item sets 2, 4 and 5 several more item sets will be produced. The complete list is quite long and thus will

not be stated here. Detailed LR(k) treatment of this grammar can e.g. be found in [1].

### Goto [edit]

The lookahead of an LR(1) item is used directly only when considering reduce actions (i.e., when the • marker is at the right end).

The **core** of an LR(1) item [S → a A • B e, c] is the LR(0) item S → a A • B e. Different LR(1) items may share the same core.

For example in item set 2

[E → T •, $]
[T → T • + n, $]
[T → T • + n, +]

the parser is required to perform the reduction [E → T] if the next symbol is '$', but to do a shift if the next symbol is '+'. Note that a LR(0) parser would not be able to make this decision, as it only considers the core of the items, and would thus report a shift/reduce conflict.

A state containing [A → α • X β, a] will move to a state containing [A → α X • β, a] with label X.

Every state has transitions according to Goto.

### Shift actions [edit]

If [A → α • b β, a] is in state $I_k$ and $I_k$ moves to state $I_m$ with label b, then we add the action

action[k, b] = "shift m"

### Reduce actions [edit]

If [A→α •, a] is in state $I_k$, then we add the action: "Reduce A → α" to action[$I_k$, a].

## References [edit]


1. ^ [a] [b] [c] 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. ^ "What is Menhir?". INRIA, CRISTAL project. Retrieved 29 June 2012.
3. ^ "HYACC, minimal LR(1) parser generator".
4. ^ "LRSTAR, minimal LR(1) parser generator".
5. ^ Franklin L. DeRemer (1969). "Practical Translators for LR(k) languages" (PDF). MIT, PhD Dissertation.
6. ^ [a] [b] Pager, D. (1977), "A Practical General Method for Constructing LR(k) Parsers", *Acta Informatica* 7: 249–268


## External links [edit]

- Syntax Analysis MS/Powerpoint presentation, Aggelos Kiayias, University of Connecticut
- LR parsing MS/Powerpoint presentation, Aggelos Kiayias, University of Connecticut
- Practical LR(k) Parser Construction HTML page, David Tribble

Categories: Parsing algorithms