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
Dansk
Deutsch
فارسی
Français
한국어
Italiano
日本語
Polski
Русский
Српски / srpski
中文
Edit links

LALR parser

From Wikipedia, the free encyclopedia
(Redirected from Look-ahead LR parser)

In computer science, an **LALR parser**[a] or **Look-Ahead LR parser** is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language. ("LR" means left-to-right, rightmost derivation.)

The LALR parser was invented by Frank DeRemer in his 1969 PhD dissertation, *Practical Translators for LR(k) languages*,[1] in his treatment of the practical difficulties at that time of implementing LR(1) parsers. He showed that the LALR parser has more language recognition power than the LR(0) parser, while requiring the same number of states as the LR(0) parser for a language that can be recognized by both parsers. This makes the LALR parser a memory-efficient alternative to the LR(1) parser for languages that are not LR(0). It was also proved that there exist LR(1) languages that are not LALR. Despite this weakness, the power of the LALR parser is enough for many mainstream computer languages,[2] including Java,[3] though the reference grammars for many languages fail to be LALR due to being ambiguous.[2]

The original dissertation gave no algorithm for constructing such a parser given some formal grammar. The first algorithms for LALR parser generation were published in 1973.[4] In 1982, DeRemer and Penello published an algorithm that generated highly memory-efficient LALR parsers.[5] LALR parsers can be automatically generated from some grammar by an LALR parser generator such as Yacc or GNU Bison. The automatically generated code may be augmented by hand-written code to augment the power of the resulting parser.

Contents [hide]

1 History
2 Overview
3 Relation to other parsers
3.1 LR parsers
3.2 LL parsers
4 Implementation issues
5 See also
6 Notes
7 References
8 External links

History [edit]

In 1965, Donald Knuth invented the LR parser (**L**eft to Right, **R**ightmost derivation). The LR parser can recognize any deterministic context-free language in linear-bounded time.[6] Rightmost derivation has very large memory requirements and implementing an LR parser was impractical due to the limited memory of computers at that time. To address this shortcoming, in 1969, Frank DeRemer proposed two simplified versions of the LR parser, namely the **Look-Ahead LR** (LALR)[1] and the **Simple LR parser** that had much lower memory requirements at the cost of less language-recognition power, with the LALR parser being the most-powerful alternative.[1] In 1977, memory optimizations for the LR parser were invented[7] but still the LR parser was less memory-efficient than the simplified alternatives.

In 1979, Frank DeRemer and Tom Pennello announced a series of optimizations for the LALR parser that would further improve its memory efficiency.[8] Their work was published in 1982.[5]

Overview [edit]

Generally, the LALR parser refers to the LALR(1) parser,[b] just as the LR parser generally refers to the LR(1) parser. The "(1)" denotes one-token lookahead, to resolve differences between rule patterns during parsing. Similarly, there is an LALR(2) parser with two-token lookahead, and LALR(*k*) parsers with *k*-token lookup, but these are rare in actual use. The LALR parser is based on the LR(0) parser, so it can also be denoted LALR(1) = LA(1)LR(0) (1 token of lookahead, LR(0)) or more generally LALR(*k*) = LA(*k*)LR(0) (k tokens of lookahead, LR(0)). There is in fact a two-parameter family of LA(*k*)LR(*j*) parsers for all combinations of *j* and *k*, which can be derived from the LR(*j* + *k*) parser,[9] but these do not see practical use.

As with other types of LR parsers, an LALR parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, because it does not need to use backtracking. Being a lookahead parser by definition, it always uses a lookahead, with LALR(1) being the most-common case.

Relation to other parsers [edit]

LR parsers [edit]

The LALR(1) parser is less powerful than the LR(1) parser, and more powerful than the SLR(1) parser, though they all use the same production rules. The simplification that the LALR parser introduces consists in merging rules that have identical **kernel item sets**, because during the LR(0) state-construction process the lookaheads are not known. This reduces the power of the parser because not knowing the lookahead symbols can confuse the parser as to which grammar rule to pick next, resulting in **reduce/reduce conflicts**. All conflicts that arise in applying a LALR(1) parser to an unambiguous LR(1) grammar are reduce/reduce conflicts. The SLR(1) parser performs further merging, which introduces additional conflicts.

The standard example of an LR(1) grammar that cannot be parsed with the LALR(1) parser, exhibiting such a reduce/reduce conflict, is:[10][11]

```
S → a E c
  → a F d
  → b F c
  → b E d
E → e
F → e
```

In the LALR table construction, two states will be merged into one state and later the lookaheads will be found to be ambiguous. The one state with lookaheads is:

```
E → e. {c,d}
F → e. {c,d}
```

An LR(1) parser will create two different states (with non-conflicting lookaheads), neither of which is ambiguous. In an LALR parser this one state has conflicting actions (given lookahead c or d, reduce to E or F), a "reduce/reduce conflict"; the above grammar will be declared ambiguous by a LALR parser generator and conflicts will be reported.

To recover, this ambiguity is resolved by choosing E, because it occurs before F in the grammar. However, the resultant parser will not be able to recognize the valid input sequence `b e c`, since the ambiguous sequence `e c` is reduced to `(E → e) c`, rather than the correct `(F → e) c`, but `b E c` is not in the grammar.

LL parsers [edit]

The LALR(*k*) parsers are incomparable with LL(*k*) parsers: for any *j* and *k* both greater than 0, there are LALR(*j*) grammars that are not LL(*k*) grammars and conversely. In fact, it is undecidable whether a given LL(1) grammar is LALR(*k*) for any $k > 0$.[2]

Depending on the presence of empty derivations, a LL(1) grammar can be equal to a SLR(1) or a LALR(1) grammar. If the LL(1) grammar has no empty derivations it is SLR(1) and if all symbols with empty derivations have non-empty derivations it is LALR(1). If symbols having only an empty derivation exist, the grammar may or may not be LALR(1).[12]

Implementation issues [edit]

Because the LALR parser performs a right derivation instead of the more intuitive left derivation, understanding how it works is quite difficult. This makes the process of finding a correct and efficient LALR grammar very demanding and time-consuming.[*citation needed*] For the same reason, error-reporting can be quite hard because LALR parser errors cannot always be interpreted into messages with high-level terms meaningful for the end user.[*citation needed*] However, any LR(k > 0) table makes it trivial to at least enumerate the various tokens that would have been valid options when a syntax error occurred, for low-level error messages. For this reason, the recursive descent parser is sometimes preferred over the LALR parser. This parser requires more hand-written code because of its lower language-recognition power. However, it does not have the special difficulties of the LALR parser because it performs left-derivation. Notable examples of this phenomenon are the C-language and

C++ parsers of the Gnu Compiler Collection. These started as LALR parsers but were later changed to recursive-descent parsers.[13][14]

See also [edit]

- Comparison of parser generators
- Context-free grammar
- Lookahead in parsing
- Parser generator
- Token scanner

Notes [edit]

a. ^ "LALR" is pronounced as the initialism "el-ay-el-arr"
b. ^ "LALR(1)" is pronounced as the initialism "el-ay-el-arr-one"

References [edit]

1. ^ a b c DeRemer 1969.
2. ^ a b c *LR Parsing: Theory and Practice,* Nigel P. Chapman, p. 86–87
3. ^ "Generate the Parser". Eclipse JDT Project. Retrieved 29 June 2012.
4. ^ Anderson, T.; Eve, J.; Horning, J. (1973). "Efficient LR(1) parsers". *Acta Informatica* (2): 2–39.
5. ^ a b DeRemer, Frank; Penello, Thomas (October 1982). "Efficient Computation of LALR(1) Look-Ahead Sets" (PDF). *Tansactions on Programming Languages and Systems* (ACM) **4** (4): 615–649. Retrieved 25 July 2014.
6. ^ 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.
7. ^ Pager, D. (1977), "A Practical General Method for Constructing LR(k) Parsers", *Acta Informatica 7*: 249–268
8. ^ Frank DeRemer, Thomas Pennello (1979), "Efficient Computation of LALR(1) Look-Ahead Sets", *Sigplan Notices - SIGPLAN, vol. 14, no. 8*: 176–187
9. ^ *Parsing Techniques: A Practical Guide,* by Dick Grune and Ceriel J. H. Jacobs, "9.7 LALR(1)", p. 302
10. ^ "7.9 LR(1) but not LALR(1)", *CSE 756: Compiler Design and Implementation,* Eitan Gurari, Spring 2008
11. ^ "Why is this LR(1) grammar not LALR(1)?"
12. ^ (Beatty 1982)
13. ^ "GCC 3.4 Release Series Changes, New Features, and Fixes", GCC.gnu.org.
14. ^ "GCC 4.1 Release Series Changes, New Features, and Fixes", GCC.gnu.org.

- DeRemer, Franklin L. (1969). *Practical Translators for LR(k) languages* (PDF) (Ph.D.). MIT.
- Beatty, J. C. (1982). "On the relationship between LL(1) and LR(1) grammars" (PDF). *Journal of the ACM* **29** (4 (Oct)): 1007–1022. doi:10.1145/322344.322350.

External links [edit]

- Parsing Simulator This simulator is used to generate parsing tables LALR and resolve the exercises of the book.
- JS/CC JavaScript based implementation of a LALR(1) parser generator, which can be run in a web-browser or from the command-line.
- LALR(1) tutorial, a flash card-like tutorial on LALR(1) parsing.

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