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
Deutsch
日本語
Polski
Português
Русский
Edit links

GLR parser

From Wikipedia, the free encyclopedia

A **GLR parser** (GLR standing for "generalized LR", where L stands for "left-to-right" and R stands for "rightmost (derivation)") is an extension of an LR parser algorithm to handle nondeterministic and ambiguous grammars. The theoretical foundation was provided in a 1974 paper[1] by Bernard Lang (along with other general Context-Free parsers such as GLL). It describes a systematic way to produce such algorithms, and provides uniform results regarding correctness proofs, complexity with respect to grammar classes, and optimization techniques. The first actual implementation of GLR was described in a 1984 paper by Masaru Tomita, it has also been referred to as a "parallel parser". Tomita presented five stages in his original work,[2] though in practice it is the second stage that is recognized as the GLR parser.

Though the algorithm has evolved since its original forms, the principles have remained intact. As shown by an earlier publication,[3] Lang was primarily interested in more easily used and more flexible parsers for extensible programming languages. Tomita's goal was to parse natural language text thoroughly and efficiently. Standard LR parsers cannot accommodate the nondeterministic and ambiguous nature of natural language, and the GLR algorithm can.

Contents [hide]

Algorithm [edit]

Briefly, the GLR algorithm works in a manner similar to the LR parser algorithm, except that, given a particular grammar, a GLR parser will process all possible interpretations of a given input in a breadth-first search. On the front-end, a GLR parser generator converts an input grammar into parser tables, in a manner similar to an LR generator. However, where LR parse tables allow for only one state transition (given a state and an input token), GLR parse tables allow for multiple transitions. In effect, GLR allows for shift/reduce and reduce/reduce conflicts.

When a conflicting transition is encountered, the parse stack is forked into two or more parallel parse stacks, where the state corresponding to each possible transition is at the top. Then, the next input token is read and used to determine the next transition(s) for each of the "top" states – and further forking can occur. If any given top state and input token do not result in at least one transition, then that "path" through the parse tables is invalid and can be discarded.

A crucial optimization allows sharing of common prefixes and suffixes of these stacks, which constrains the overall search space and memory usage required to parse input text. The complex structures that arise from this improvement make the search graph a directed acyclic graph (with additional restrictions on the "depths" of various nodes), rather than a tree.

Advantages [edit]

Recognition using the GLR algorithm has the same worst-case time complexity as the CYK algorithm and Earley algorithm: $O(n^3)$.[*citation needed*] However, GLR carries two additional advantages:

- The time required to run the algorithm is proportional to the degree of nondeterminism in the grammar: on deterministic grammars the GLR algorithm runs in $O(n)$ time (this is not true of the Earley[*citation needed*] and CYK algorithms, but the original Earley algorithms can be modified to ensure it)
- The GLR algorithm is "online" – that is, it consumes the input tokens in a specific order and performs as much work as possible after consuming each token.

In practice, the grammars of most programming languages are deterministic or "nearly deterministic", meaning that any nondeterminism is usually resolved within a small (though possibly unbounded) number of tokens. Compared to other algorithms capable of handling the full class of context-free grammars (such as Earley or CYK), the GLR algorithm gives better performance on these "nearly deterministic" grammars, because only a single stack will be active during the majority of the parsing process.

GLR can be combined with the LALR(1) algorithm, in a hybrid parser, allowing still higher performance.[4]

See also [edit]

- Comparison of parser generators
- ASF+SDF Meta Environment
- DMS Software Reengineering Toolkit
- GNU Bison, a parser generator that can create LALR and GLR parsers

References [edit]

1. ^ Lang, Bernard (1974). Loeckx, J., ed. "Deterministic techniques for efficient non-deterministic parsers". *Automata, Languages and Programming, 2nd Colloquium*. Lecture Notes in Computer Science (Saarbrücken: Springer) **14**: 255–269. doi:10.1007/3-540-06841-4_65. ISSN 0302-9743.
2. ^ Masaru Tomita. Efficient parsing for natural language. Kluwer Academic Publishers, Boston, 1986.
3. ^ Lang, Bernard (December 1971). "Parallel non-deterministic bottom-up parsing". *ACM SIGPLAN Notices*. Proceedings of the international symposium on Extensible languages **6** (12): 56–57. doi:10.1145/942582.807982.
4. ^ "Elkhound, Elsa and Cqual++: Open-Source Static Analysis for C++".

Further reading [edit]

- Grune, Dick; Jacobs, Ceriel J.H (2008). *Parsing Techniques*. Springer Science+Business Media. ISBN 978-0-387-20248-8.
- Tomita, Masaru (1984). "LR parsers for natural languages". *COLING*. 10th International Conference on Computational Linguistics. pp. 354–357.
- Tomita, Masaru (1985). "An efficient context-free parsing algorithm for natural languages". *IJCAI*. International Joint Conference on Artificial Intelligence. pp. 756–764.

Categories: Parsing algorithms