CSE6339 3.0 Introduction to Computational Linguistics

Insructor: Nick Cercone – 3050 LAS – nick@cse.yorku.ca

Winter Semester, 2014

LR Parsers

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.

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. Other qualifiers, as in SLR and LALR, often precede the name LR.

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.

Variants of LR Parsers

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.

In the recursive ascent parser variation, the implicit stack used by subroutine calls also replaces the explicit parse stack structure. 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 highly ambiguous grammars such as 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 an unambiguous, 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

Donald Knuth invented LR parsers 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."
>
> "A language can be generated by an LR(*k*) grammar if and only if it is deterministic, if and only if it can be generated by an LR(1) grammar."[1]

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).

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.

For full details on LR theory and how LR parsers are derived from grammars, see *The Theory of Parsing, Translation, and Compiling, Volume 1* (Aho and Ullman). [3]

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

References

1. Knuth, D. E. (July 1965). "On the translation of languages from left to right". *Information and Control* **8** (6): 607–639. doi:10.1016/S0019-9958(65)90426-2.
2. Simple LR(k) Grammars, by Frank DeRemer, Comm. ACM 14:7 1971.
3. The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing, by Alfred Aho and Jeffrey Ullman, Prentice Hall 1972.