LR(1) and LALR(1) Grammars

All of the practical bottom-up parsing techniques start with something like the LR(0) parse tables. The techniques differ in the ways they resolve conflicts in the tables. For example, consider the grammar

P1: S' ::= S EOF
P2: S ::= a+x+E
P3: E ::= a+E
P4: E ::= a

The LR(0) automaton contains a state that has the following items in it: [E ::= a.+E] and [E ::= a.] The first item wants to do a shift if it sees a + token; the second item wants to do a reduction using rule P4. What do we do?

Here are 4 possible ways to resolve such conflicts:

LR(0): Don't allow conflicts.

SLR(1): Use the next token to determine what to do. On a shift/reduce conflict with [A ::= $\alpha.\beta$] and [B ::= $\alpha.$], shift the next token if it is in First(β), and do a reduction using the B-rule if the next token is in Follow(B). Of course, this requires First(β) and Follow(B) to have nothing in common. On a reduce/reduce conflict with [A ::= $\alpha.$] and [B ::= $\alpha.$], use the Follow(A) and Follow(B) sets to resolve the conflict.

LR(1): Modify the LR-automaton to include in each item the token that could possibly follow it. Use this for the disambiguation.

LALR(1): Use the SLR(1) automaton. At points of conflict work backwards to find the lookahead tokens that can be used to resolve the conflict.

An LR(1) item is [A ::= $\beta_1.\beta_2$, t], where [A ::= $\beta_1.\beta_2$] is an LR(0) item and t is a token. This is *viable* if we can derive the string $\alpha\ \beta_1\beta_2\ \omega$ and t is in First(ω). In other words, it means that t is the next input symbol to follow $\beta_1\beta_2$.

Consider the following grammar:

P1: S' ::= S EOF
P2: S ::= AaAb
P3: A ::= bA
P4: A ::= d
P5: S ::= BbBa
P6: B ::= d

This grammar is not SLR(1); the automaton for it has a state containing both [A ::= d.] and [B ::= d.]. Since Follow(A) and Follow(B) are both {a, b}, there is no hope of using the Follow sets to resolve the conflict.

We can build an LR(1) table for this grammar. It starts with a state containing all of the following items:

[S' ::= .S EOF, EOF]
[S ::= .AaAb, EOF]
[S ::= .BbBa, EOF]
[A ::= .bA, a]
[A ::= .d, a]
B ::= .d, b]

Note that we add the rule [A ::= .bA, a], because the dot in [S ::= .AaAb] is in front of the non-terminal A; after we find this initial A we expect it to be followed by a. This is the lookahead in [A ::= .bA, a]

This process continues until all states lead to reductions.

This is fine and very general, but the resulting automata and their tables are very large. The LR(1) table for typical programming language has several thousand states and hundreds of thousands of entries.

The LALR(1) technique builds the LR(0) automaton, then works backwards from conflict states to find the appropriate LR(1) lookaheads. The text has the details of an algorithm for this.

For a typical language like C there are about 100 grammar rules and somewhere between 50 and 100 valid tokens.

An LR(1) parse table for such a language has several thousand states, several hundred thousand entries, and a table size on the order of 1 MB.

An LALR(1) parse table for the language has several hundred states, perhaps 20,000 entries and occupies less than 100 KB of space.3