بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر

(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

Constructing LALR Parsing Tables

We now introduce our last parser construction method, the LALR (lookahead-LR) technique. This method is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar. The same is almost true for SLR grammars, but there are a few constructs that cannot be conveniently handled by SLR techniques (see Example 4.48, for example).

For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like C. The canonical LR table would typically have several thousand states for the same-size language. Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

نکاتی دربارهٔ چیستی و چرایی تجزیهٔ $\mathrm{LALR}(1)$

☞ $\mathrm{LR}(1)$ *parsers would be impractical in that the space required for their deterministic automata would be prohibitive. A modest grammar might already require hundreds of thousands or even millions of states.*

> $\mathrm{LR}(1)$ *parsing tables can be very large, with many states. A smaller table can be made by merging any two states whose items are identical except for lookahead sets. The result parser is called an* $\mathrm{LALR}(1)$ *parser, for lookahead* $\mathrm{LR}(1)$*. The surprising thing is that this procedure preserves almost all the original lookahead power and still saves an enormous amount of memory.*

☞ $\mathrm{LALR}(1)$ *parsers are powerful, almost as powerful as* $\mathrm{LR}(1)$ *parsers, they have fairly modest memory requirements, only slightly inferior to (= larger than) those of* $\mathrm{LR}(0)$ *parsers, and they are time-efficient.* $\mathrm{LALR}(1)$ *parsing may very well be the most-used parsing method in the world today. Probably the most famous* $\mathrm{LALR}(1)$ *parser generators are yacc and its GNU version bison.*

☞ *Even though* $\mathrm{LALR}(1)$ *grammars are not as powerful as* $\mathrm{LR}(1)$ *grammars, they are sufficiently powerful to describe most programming languages. This, together with their small (relative to LR) table size, makes the* $\mathrm{LALR}(1)$ *family of grammars an excellent candidate for the automatic generation of parsers. Stephen C. Johnson's YACC, for "Yet Another Compiler-Compiler", based on* $\mathrm{LALR}(1)$ *techniques, was probably the first practical bottom-up parser generator. GNU has developed an open-source version called Bison.*

☞ LALR(1) *grammars make for parsers that are almost as powerful as* LR(1) *grammars but result in much more space-efficient parsing tables. This goes some way in explaining the popularity of parser generators such as YACC and Bison.*

☞ *An* LR(1) *parsing table for a typical programming language such as Java can have thousands of states, and so thousands of rows. One could argue that, given the inexpensive memory nowadays, this is not a problem. On the other hand, smaller programs and data make for faster running programs so it would be advantageous if we might be able to reduce the number of states.* LALR(1) *is a parsing method that does just this.*

Example 4.60 : Again consider grammar (4.55) whose GOTO graph was shown in Fig. 4.41. As we mentioned, there are three pairs of sets of items that can be merged. I_3 and I_6 are replaced by their union:

$$\begin{array}{ll} I_{36}: & C \to c{\cdot}C,\ c/d/\$ \\ & C \to {\cdot}cC,\ c/d/\$ \\ & C \to {\cdot}d,\ c/d/\$ \end{array}$$

I_4 and I_7 are replaced by their union:

$$I_{47}: \quad C \to d{\cdot},\ c/d/\$$$

and I_8 and I_9 are replaced by their union:

$$I_{89}: \quad C \to cC{\cdot},\ c/d/\$$$

The LALR action and goto functions for the condensed sets of items are shown in Fig. 4.43.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figure 4.43: LALR parsing table for the grammar of Example 4.54

To see how the GOTO's are computed, consider GOTO(I_{36}, C). In the original set of LR(1) items, GOTO$(I_3,\ C) = I_8$, and I_8 is now part of I_{89}, so we make GOTO(I_{36}, C) be I_{89}. We could have arrived at the same conclusion if we considered I_6, the other part of I_{36}. That is, GOTO$(I_6, C) = I_9$, and I_9 is now part of I_{89}. For another example, consider GOTO(I_2, c), an entry that is exercised after the shift action of I_2 on input c. In the original sets of LR(1) items, GOTO$(I_2, c) = I_6$. Since I_6 is now part of I_{36}, GOTO(I_2, c) becomes I_{36}. Thus, the entry in Fig. 4.43 for state 2 and input c is made s36, meaning shift and push state 36 onto the stack. □

ساخت جدول LALR(1)

Algorithm 4.59: An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of GOTO$(I_1, X)$, GOTO$(I_2, X), \ldots,$ GOTO(I_k, X) are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X). Then GOTO$(J, X) = K$.

There are algorithms for directly constructing the tables for $\mathrm{LALR}(1)$ ***parsing without first constructing the tables for*** $\mathrm{LR}(1)$ ***parsing, but we do not consider them here.***

یک پارسر $\mathrm{LALR}(1)$ برای گرامری که اتوماتون $\mathrm{LR}(1)$ آن ۲۲ استیت داشت

$$
\begin{array}{lll}
0.\ E' & ::= & E \\
1.\ E & ::= & E + T \\
2.\ E & ::= & T \\
3.\ T & ::= & T * F \\
4.\ T & ::= & F \\
5.\ F & ::= & (E) \\
6.\ F & ::= & \mathtt{id}
\end{array}
$$

یک پارسر LALR(1) برای گرامری که اتوماتون LR(1) آن ۲۲ استیت داشت

$s_0 =$

$$\begin{array}{ll} \{[E' ::= \cdot\ E,\ \#], & \text{goto}(s_0, E) = s_1 \\ [E ::= \cdot\ E + T,\ +/\#], & \text{goto}(s_0, T) = s_{2.9} \\ [E ::= \cdot\ T,\ +/\#], & \text{goto}(s_0, F) = s_{3.10} \\ [T ::= \cdot\ T * F,\ +/*/\#], & \text{goto}(s_0, () = s_{4.11} \\ [T ::= \cdot\ F,\ +/*/\#], & \text{goto}(s_0, \texttt{id}) = s_{5.12} \\ [F ::= \cdot (E),\ +/*/\#], & \\ [F ::= \cdot\ \texttt{id},\ +/*/\#]\} & \end{array}$$

$s_1 =$

$$\begin{array}{ll} \{[E' ::= E\ \cdot,\ \#], & \text{goto}(s_1, +) = s_{6.16} \\ [E ::= E\ \cdot + T,\ +/\#]\} & \end{array}$$

$s_{6.16} =$

$$\begin{array}{ll} \{[E ::= E + \cdot\ T,\ +/)/\#], & \text{goto}(s_{6.16}, T) = s_{13.19} \\ [T ::= \cdot\ T * F,\ +/*/)/\#], & \text{goto}(s_{6.16}, F) = s_{3.10} \\ [T ::= \cdot\ F,\ +/*/)/\#], & \text{goto}(s_{6.16}, () = s_{4.11} \\ [F ::= \cdot\ (E),\ +/*/)/\#], & \text{goto}(s_{6.16}, \texttt{id}) = s_{5.12} \\ [F ::= \cdot\ \texttt{id},\ +/*/)/\#]\} & \end{array}$$

$s_{7.17} =$

$$\begin{array}{ll} \{[T ::= T * \cdot\ F,\ +/*/)/\#], & \text{goto}(s_{7.17}, F) = s_{14.20} \\ [F ::= \cdot\ (E),\ +/*/)/\#], & \text{goto}(s_{7.17}, () = s_{4.11} \\ [F ::= \cdot\ \texttt{id},\ +/*/)/\#]\} & \text{goto}(s_{7.17}, \texttt{id}) = s_{5.12} \end{array}$$

یک پارسر LALR(1) برای گرامری که اتوماتون LR(1) آن ۲۲ استیت داشت

$s_{2.9} =$

$\{[E ::= T \cdot, +/)/\#],$ $\mathrm{goto}(s_{2.9}, *) = s_{7.17}$
$[T ::= T \cdot * F, +/*/)/\#]\}$

$s_{3.10} =$

$\{[T ::= F \cdot, +/*/)/\#]\}$

$s_{4.11} =$

$\{[F ::= (\cdot E), +/*/)/\#],$ $\mathrm{goto}(s_{4.11}, E) = s_{8.18}$
$[E ::= \cdot E + T, +/)],$ $\mathrm{goto}(s_{4.11}, T) = s_{2.9}$
$[E ::= \cdot T, +/)],$ $\mathrm{goto}(s_{4.11}, F) = s_{3.10}$
$[T ::= \cdot T * F, +/*/)],$ $\mathrm{goto}(s_{4.11}, () = s_{4.11}$
$[T ::= \cdot F, +/*/)],$ $\mathrm{goto}(s_{4.11}, \mathtt{id}) = s_{5.12}$
$[F ::= \cdot (E), +/*/)],$
$[F ::= \cdot \mathtt{id}, +/*/)]\}$

$s_{5.12} = \{[F ::= \mathtt{id} \cdot, +/*/)/\#]\}$

$s_{8.18} =$

$\{[F ::= (E \cdot), +/*/)/\#],$ $\mathrm{goto}(s_{8.18},)) = s_{15.21}$
$[E ::= E \cdot + T, +/)]\}$ $\mathrm{goto}(s_{8.18}, +) = s_{6.16}$

$s_{13.19} =$

$\{[E ::= E + T \cdot, +/)/\#],$ $\mathrm{goto}(s_{13.19}, *) = s_{7.17}$
$[T ::= T \cdot * F, +/*/)/\#]\}$

$s_{14.20} = \{[T ::= T * F \cdot, +/*/)/\#]\}$

$s_{15.21} = \{[F ::= (E) \cdot, +/*/)]\}$

جدول LALR(1) بجای ۲۲ سطر، ۱۲ سطر دارد

	Action						Goto		
	+	*	(	)	id	#	E	T	F
0			s4		s5		1	2	3
1	s6					accept			
2.9	r2	s7.17		r2		r2			
3.10	r4	r4		r4		r4			
4.11			s11		s12		8.18	2.9	3.10
5.12	r6	r6		r6		r6			
6.16			s4		s5			13.19	3.10
7.17			s4		s5				14.20
8.18	s16			s15					
13.19	r1	s7				r1			
14.20	r3	r3				r3			
15.21	r5	r5				r5			

دربارهٔ ماهیت کانفلیکت‌ها در تجزیهٔ $\mathrm{LALR}(1)$

Suppose we have an $\mathrm{LR}(1)$ grammar, that is, one whose sets of $\mathrm{LR}(1)$ items produce no parsing-action conflicts. If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict, but it is unlikely for the following reason: Suppose in the union there is a conflict on lookahead a because there is an item $[A \to \alpha\bullet, a]$ calling for a reduction by $A \to \alpha$, and there is another item $[B \to \beta \bullet a\gamma, b]$ calling for a shift. Then some set of items from which the union was formed has item $[A \to \alpha\bullet, a]$, and since the cores of all these states are the same, it must have an item $[B \to \beta \bullet a\gamma, c]$ for some c. But then this state has the same shift/reduce conflict on a, and the grammar was not $\mathrm{LR}(1)$ as we assumed. Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on the core, not the lookahead.

It is possible, however, that a merger will produce a reduce/reduce conflict, as the following example shows.

Example 4.58 : Consider the grammar

$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\
A & \rightarrow & c \\
B & \rightarrow & c
\end{array}
$$

which generates the four strings acd, ace, bcd, and bce. The reader can check that the grammar is LR(1) by constructing the sets of items. Upon doing so, we find the set of items $\{[A \rightarrow c\cdot,\ d], [B \rightarrow c\cdot,\ e]\}$ valid for viable prefix ac and $\{[A \rightarrow c\cdot, e], [B \rightarrow c\cdot,\ d]\}$ valid for bc. Neither of these sets has a conflict, and their cores are the same. However, their union, which is

$$
\begin{array}{l}
A \rightarrow c\cdot,\ d/e \\
B \rightarrow c\cdot,\ d/e
\end{array}
$$

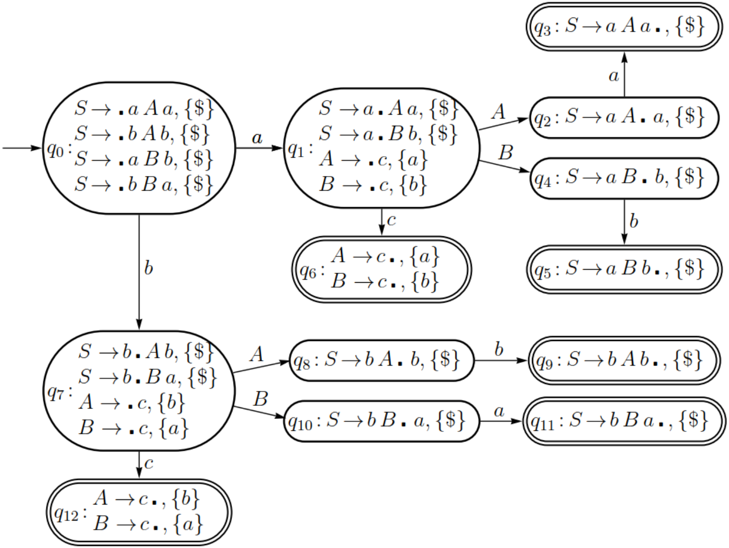
generates a reduce/reduce conflict, since reductions by both $A \rightarrow c$ and $B \rightarrow c$ are called for on inputs d and e. □

Example (a non-$\mathrm{LALR}(1)$ grammar): ***Let us consider the grammar G with axiom S and the following productions:***

$$
\begin{array}{ll}
1. & S \to a\,A\,a \\
2. & S \to b\,A\,b \\
3. & S \to a\,B\,b \\
4. & S \to b\,B\,a \\
5. & A \to c \\
6. & B \to c
\end{array}
$$

Since the axiom S does not occur on the right hand side of any production, we can take the given grammar G to be the augmented grammar of the grammar G itself.

The finite automaton for the $\mathrm{LR}(1)$ *parsing of the grammar* G



LR(1) Parsing table for the grammar G

	action				goto	
	a	b	c	$\$$	A	B
q_0	$sh\ q_1$	$sh\ q_7$				
q_1			$sh\ q_6$		q_2	q_4
q_2	$sh\ q_3$					
q_3				$red\ 1$		
q_4		$sh\ q_5$				
q_5				$red\ 3$		
q_6	$red\ 5$	$red\ 6$				
q_7			$sh\ q_{12}$		q_8	q_{10}
q_8		$sh\ q_9$				
q_9				$red\ 2$		
q_{10}	$sh\ q_{11}$					
q_{11}				$red\ 4$		
q_{12}	$red\ 6$	$red\ 5$				

where:

1. $S \to a\,A\,a$
2. $S \to b\,A\,b$
3. $S \to a\,B\,b$
4. $S \to b\,B\,a$
5. $A \to c$
6. $B \to c$

We fuse the states q_6 and q_{12}, thereby getting the state: $\boxed{q_{6\,12}: \begin{array}{l} A \to c\,\centerdot\,, \{a, b\} \\ B \to c\,\centerdot\,, \{a, b\} \end{array}}$ *. Thus, in the* $\mathrm{LALR}(1)$ *parsing table we will get two reduce-reduce conflicts, because in the columns* a *and* b *of that table we have both the* $red\ 5$ *action (for the production* $A \to c$*) and also the* $red\ 6$ *action (for the production* $B \to c$*). Hence, the given grammar* G *is not an* $\mathrm{LALR}(1)$ *grammar.*

جمع‌بندی و نکات نهایی بحث تجزیهٔ پایین به بالا

Theoretically, $\mathrm{LR}(1)$ grammars are the largest category of grammars that can be parsed deterministically while looking ahead just one token. Of course, $\mathrm{LR}(k)$ grammars for $k > 1$ are even more powerful, but one must look ahead k tokens, more importantly, the parsing tables must (in principle) keep track of all possible token strings of length k. So, in principle, the tables can grow exponentially with k.

FACT **[Relationships Between $LALR(k)$ Grammars and $LR(k)$ Grammars]** The class of $LALR(0)$ grammars coincides with the class of $LR(0)$ grammars. For $k \geq 1$ the class of $LALR(k)$ grammars is properly contained in the class of $LR(k)$ grammars.

EXAMPLE The grammar with axiom S and productions:

$$S \to aAa \mid bAb \mid aBb \mid bBa$$

$$A \to c$$

$$B \to c$$

is an $LR(1)$ grammar which is *not* $LALR(k)$ for any $k \geq 0$.

FACT **[Hierarchy of the $LR(k)$ Grammars]** For all $k \geq 0$, the grammar with axiom S and the productions:

$$S \to ab^k c \mid Ab^k d$$

$$A \to a$$

is an $LR(k+1)$ grammar which is *not* an $LR(k)$ grammar.

A hierarchy of grammar classes