8 Parsing 4 - General LR Parsing

Non-SLR Grammars

All LR grammars are unambiguous, but there are unambiguous grammars that are not LR. Indeed there are LR grammars that are not SLR, i.e. for which useful parser tables cannot be generated by the SLR algorithm. To see this, consider the following example, from Aho and Ullman, Chapter 6.

$$\begin{aligned} S &\rightarrow L = R \\ S &\rightarrow R \\ L &\rightarrow * R \\ L &\rightarrow \textbf{id} \\ R &\rightarrow L \end{aligned}$$

This is a partial grammar of C language *l-values* and *r-values* in assignments. $*$ represents the dereferencing operator. If we try to build the SLR table for this grammar, we arrive at the sets of *LR(0)* items shown in Figure . This produces two definitions for $ACTION[2,=]$, from set I_2 item 1 gives "shift 6", while item 2 gives "reduce $R \rightarrow L$". This is an example of a *shift-reduce conflict*. This is not because the grammar is ambiguous, but because the *SLR* parsing technique is not sufficiently powerful.

Canonical LR Parsing

SLR parsing assumes that, in state i, the reduction $A \rightarrow \alpha$ is always made when symbol a is input if the corresponding item set I_i contains $[A \rightarrow \alpha \cdot]$ and a is in FOLLOW(A). This ignores situations where the stack may contain symbols before the sequence α which make no sense if α is reduced to A. Formally, if the stack contains a viable prefix $\beta\ \alpha$ and there is no right-sentential form where $\beta\ A$ occurs, a reduction of α to A cannot be allowed.

In the example grammar above, this means that the reduction of L to R on input of $=$ should not be legal, since there is no right hand side whose start matches the resultant $R\ =$.

The SLR tables do not embody this level of information, since they are built from $LR(0)$ items. To deal with this problem a more powerful table generating algorithm is needed, which works in terms of the $LR(1)$ items. In these items, a one terminal symbol lookahead is used. This means that every item now has a second value, which is used to note which symbols may validly be the next input following a reduction.

An item is now of the form $[A \rightarrow \alpha \cdot \beta,\ a]$, where a is the lookahead component. It is an $LR(1)$ item since the lookahead component is 1 terminal symbol (including \$ as a terminal) long. This value has no effect unless β is the empty string ϵ, which can be written as $[A \rightarrow \alpha \cdot]$. In that case it says that the reduction of α to A is only allowed if the next input symbol is a.

Constructing the parse tables for this approach is essentially the same as that for *SLR* parsing, except that now some states are split according to the lookahead component. CLOSURE and GOTO are still needed, but are more complicated.

$$
\begin{array}{lrcl}
I_0\text{:} & S' & \to & \cdot\, S \\
 & S & \to & \cdot L \;=\; R \\
 & S & \to & \cdot R \\
 & L & \to & \cdot\, \mathbf{id} \\
 & R & \to & \cdot\, L \\
 & & & \\
I_1\text{:} & S' & \to & S\;\cdot \\
 & & & \\
I_2\text{:} & S & \to & L\cdot \;=\; R \\
 & R & \to & L\;\cdot \\
 & & & \\
I_3\text{:} & S & \to & R\;\cdot \\
 & & & \\
I_4\text{:} & L & \to & *\cdot\; R \\
 & R & \to & \cdot\, L \\
 & L & \to & \cdot\; *\; R \\
 & L & \to & \cdot\, \mathbf{id} \\
 & & & \\
I_5\text{:} & L & \to & \mathbf{id}\;\cdot \\
 & & & \\
I_6\text{:} & S & \to & L = \cdot\, R \\
 & R & \to & \cdot\, L \\
 & L & \to & \cdot\; *\; R \\
 & L & \to & \cdot\, \mathbf{id} \\
 & & & \\
I_7\text{:} & L & \to & *R\;\cdot \\
 & & & \\
I_8\text{:} & R & \to & L\;\cdot \\
 & & & \\
I_9\text{:} & S & \to & L \;=\; R\;\cdot
\end{array}
$$

Figure 1: Shift/reduce conflict in SLR parsing

For an item of the form $[A \rightarrow \alpha \cdot B\ \beta,\ a]$, in the set of items for a viable prefix γ, there is a rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$, where $\gamma = \delta\alpha$.

Now suppose that βax can be matched by bz, a string starting with the terminal symbol b. For every production of the form $B \rightarrow \eta$, there is a derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta\gamma Bbz \underset{rm}{\Rightarrow} \gamma\eta bz$. This means that the item $[B \rightarrow \cdot\eta,\ b]$ is valid for γ.

b could be the first terminal in the derivation of β or, if β derives the empty string ϵ, it could be the terminal symbol a. Since a is a terminal, b is never the first terminal derived in x and so the general definition of b is any terminal in FIRST(βa).

Closure for LR(1) items

We start with the closure of $\{[S' \rightarrow \cdot\ S,\ \$]\}$.

Matching this with the general form $[A \rightarrow \alpha \cdot B\beta,\ a]$, so that $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ and $a = \$$. Now for each production of the form $B \rightarrow \gamma$ and terminal b in FIRST(βa), we add an item $[B \rightarrow \cdot\ \gamma,\ b]$.

For the grammar:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow \mathrm{c}C \mid \mathrm{d} \end{aligned}$$

the only match for $B \rightarrow \gamma$ is $S \rightarrow CC$. β is ϵ and a is \$ and so the only terminal matching b must be \$. Thus the item $[S \rightarrow \cdot\ CC,\ \$]$ is added to the set.

Now the items with C as a right hand side and a dot at the start of the left hand side are added, i.e. those matching $[C \rightarrow \cdot\ \gamma,\ b]$. The terminal b is any in FIRST($C\$$), which is actually the same as FIRST(C) as C never derives the empty string ϵ. As there are two values in this set, c and d, and 2 productions are of the required form $C \rightarrow \gamma$, 4 $LR(1)$ items are added. Thus the set of items making up state 0 are:

$$\begin{aligned} I_0\colon S' &\rightarrow \cdot S,\ \$ \\ S &\rightarrow \cdot CC,\ \$ \\ C &\rightarrow \cdot cC,\ c \\ C &\rightarrow \cdot cC,\ d \\ C &\rightarrow \cdot d,\ c \\ C &\rightarrow \cdot d,\ d \end{aligned}$$

In future we follow Aho and Ullman and abbreviate sets of items which differ only in their lookahead terminal as one item with a forward slash separated list of terminals, e.g. $C \rightarrow \cdot d,\ c/d$.

GOTO is computed in the same way as before, so that GOTO($I_0,\ S$) is the closure of $[S' \rightarrow S\cdot,\ \$]$, which gives the set I_1.

GOTO($I_0,\ C$) is the closure of $[S \rightarrow C \cdot\ C,\ \$]$, which gives the set I_2. Eventually this yields 10 sets of items, as in Figure 2.

The sets I_3 and I_6 differ only in their lookahead components. In the SLR table generation algorithm they would have been amalgamated. Thus the $LR(1)$ items are able to differentiate states which might lead to shift reduce conflicts in SLR tables, although this example would not have caused problems.

Figure 2: LR(1) items

$$
\begin{array}{rcl}
I_0\colon S' & \to & \cdot S,\ \$ \\
S & \to & \cdot CC,\ \$ \\
C & \to & \cdot cC,\ c \\
C & \to & \cdot cC,\ d \\
C & \to & \cdot d,\ c \\
C & \to & \cdot d,\ d
\end{array}
$$

$$
\begin{array}{rcl}
I_1\colon S' & \to & S\cdot,\ \$
\end{array}
$$

$$
\begin{array}{rcl}
I_2\colon S & \to & C \cdot C,\ \$ \\
C & \to & \cdot cC,\ \$ \\
C & \to & \cdot d,\ \$
\end{array}
$$

$$
\begin{array}{rcl}
I_3\colon C & \to & c \cdot C,\ c/d \\
C & \to & \cdot cC,\ c/d \\
C & \to & \cdot d,\ c/d
\end{array}
$$

$$
\begin{array}{rcl}
I_4\colon C & \to & d\cdot,\ c/d
\end{array}
$$

$$
\begin{array}{rcl}
I_5\colon S & \to & CC\cdot,\ \$
\end{array}
$$

$$
\begin{array}{rcl}
I_6\colon C & \to & c \cdot C,\ \$ \\
C & \to & \cdot cC,\ \$ \\
C & \to & \cdot d,\ \$
\end{array}
$$

$$
\begin{array}{rcl}
I_7\colon C & \to & d\cdot,\ \$
\end{array}
$$

$$
\begin{array}{rcl}
I_8\colon C & \to & cC\cdot,\ c/d
\end{array}
$$

$$
\begin{array}{rcl}
I_9\colon c & \to & cC\cdot,\ \$
\end{array}
$$

Figure 3: Canonical LR parse table

State	Action			Goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Parser table generation

Following Aho and Ullman, the following algorithm will generate the parser table from the canonical LR(1) sets. Any shift/reduce conflicts or reduce/reduce conflicts in the resulting table mean the grammar was not $LR(1)$.

1. Construct the collection of sets of LR(1) items.
2. State i matches set I_i. Actions are found by:
   (a) If $[A \rightarrow \alpha \cdot a\ \beta,\ b]$ is in I_i and GOTO$(I_i,\ a) = I_j$, ACTION$[i,\ a]$ is "shift j".
   (b) If $[A \rightarrow \alpha \cdot,\ a]$ is in I_i, ACTION$[i,\ a]$ is "reduce $A \rightarrow \alpha$"
   (c) If $[S' \rightarrow S \cdot,\ \$]$ is in I_i, ACTION$[i,\ \$]$ is "accept"
3. The GOTO entries for state I_i follow directly from the construction of the sets of items.
4. All unfilled entries are defined as "error".
5. The initial state matches I_0, i.e. contains item $[S' \rightarrow \cdot\ S,\ \$]$

LALR parsing

Although they are more powerful, canonical LR parsers have much larger state spaces than SLR parsers. To reduce the state space needed from the thousands of states needed for a realistic language in canonical LR to the hundreds needed if SLR techniques work for the same language, $LALR$ parse table generation can be used. This is more general than SLR but occupies the same number of states. Informally this approach can be thought of as merging sets of items in the $LR(1)$ collection for a grammar, wherever sets have the same first components or *core*. As long as the resulting set has no conflicts the new union of these sets can replace both of them. Thus in the example of Figure 2 I_4 and I_7 can be merged to form a new set I_{47}. This has no conflicts since the ACTION for both the original sets was r3.

$$I_{47}\text{:}\ C \rightarrow d\cdot,\ c/d/\$$$

The full replacement also involves I_3 and I_6 being replaced by I_36 and I_8 and I_9 replaced by I_{89}. The algorithm for this is essentially:

Figure 4: LALR parse table

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			

1. Construct the set of $LR(1)$ items as before.
2. Replace sets with the same core with their union (i.e. the common core and the union of the lookahead terminals for each item).
3. Find the parsing actions in the same way as before, but if any conflicts arise in the merged sets the grammar is not $LALR(1)$
4. Build the GOTO table using the rule that the value of the core of GOTO($I_i,\ X$) is the same for all I_i with the same core. Thus the union of all sets with the same core as any GOTO($I_i,\ X$) is the value for GOTO($\bigcup I_i,\ X$).‘

The table produced in this way is shown below.

The major difference in LR and $LALR$ parsers is in error detection. The canonical LR parser will detect errors on the first symbol that is wrong, but the $LALR$ parser may continue to reduce for some number of levels beyond the point where the erroneous symbol reaches the front of the input.

Grammar with reduce/reduce conflict in LALR

An example of a grammar with a reduce/reduce conflict when the LALR approach is used is:

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

The construction of the $LR(1)$ sets includes
$\{[A \rightarrow c\cdot,\ d], [B \rightarrow c\cdot,\ e]\}$ and
$\{[A \rightarrow c\cdot,\ e], [B \rightarrow c\cdot,\ d]\}$
which have the same core and give a union

$$\begin{aligned} A &\rightarrow c\cdot,\ d/e \\ B &\rightarrow c\cdot,\ d/e \end{aligned}$$

which is clearly a conflict. Thus this grammar is not $LALR$ even though it is $LR(1)$