Syntax Analysis – LR(1) and LALR Parsers

Wei Wang
Where We Are

Source Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

Machine Code
Canonical LR(1) Parser
The Limitation of SLR(1)

• Consider the follow grammar and its states

(1) \( S \rightarrow E \)
(2) \( E \rightarrow L = R \)
(3) \( E \rightarrow R \)
(4) \( L \rightarrow \text{id} \)
(5) \( L \rightarrow \ast R \)
(6) \( R \rightarrow L \)

\[ S \rightarrow \cdot E \]
\[ E \rightarrow \cdot L=R; \]
\[ E \rightarrow \cdot R \]
\[ L \rightarrow \cdot \text{id} \]
\[ L \rightarrow \cdot \ast R \]
\[ R \rightarrow \cdot L \]

\( E \rightarrow L \cdot = R \)
\( R \rightarrow L \cdot \)

\( E \rightarrow L \cdot = R, \) tells us to shift on seeing =,
\( R \rightarrow L \cdot, \) tells us to reduce on FOLLOW(R).

However, \( = \in \text{FOLLOW}(R), \) we have a shift/reduce conflict!
Why is SLR(1) Weak?

- Again, consider this grammar,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1 | $S \rightarrow E$
| 2 | $E \rightarrow L \cdot = R$
| 3 | $E \rightarrow R$
| 4 | $L \rightarrow \text{id}$
| 5 | $L \rightarrow \ast R$
| 6 | $R \rightarrow L$

State $0$:
- $S \rightarrow \cdot E$
- $E \rightarrow \cdot L=R$
- $E \rightarrow \cdot R$
- $L \rightarrow \cdot \text{id}$
- $L \rightarrow \cdot \ast R$
- $R \rightarrow \cdot L$

State $1$:
- $E \rightarrow L \cdot = R$
- $R \rightarrow L$

But we get to $R \rightarrow L \cdot$ in state $1$, $=$ is not a possible symbol after $R$ given the possible derivations.

The possible derivations are:
- $S \, \$\n- $E \, \$\n- $R \, \$\n- $L \, \$\n
Only $\$\$ can follow $R$ in this case.
Why is SLR(1) Weak?

- SLR(1) utilize minimal context to predict whether to shift or reduce.
- FOLLOW(\textit{A}) means “what could follow \textit{A} somewhere in the grammar?,” even if in a particular state \textit{A} couldn't possibly have that symbol after it.
Improving SLR(1)

• Since not of its FOLLOW(A) are valid following symbols for A in a particular state, can we also key track of the possible following symbols of A when generating the states as well?
Canonical LR(1) Parser

- Bottom-up predictive parsing with
  - L: Left-to-right scan
  - R: Rightmost derivation
  - (1): One token lookahead

- Substantially more powerful than the other methods we've covered so far.

- Tracks the potential following symbols for the non-terminals in a state.
  - If the lookahead symbols meets the potential following symbols, then a reduction may be used.

- LR(k) parsers are invented by Dr. Donald Knuth.
Constructing LR(1) DFA – LR(1) Items

• An item of LR(1) is of the form \([A \rightarrow \alpha \cdot \beta, t]\).

• Similar to LR(0) items, an LR(1) item in a state means,
  - Expect to reduce by \(A \rightarrow \alpha \beta\).
  - Have already see \(\alpha\) and is on stack. The “\(\cdot\)” determines how much has been seen.
  - Expects to see symbols that will be generated by \(\beta\).

• Unlike LR(0), an LR(1) item has an expected lookahead symbol \(t\), which means,
  - When expecting to reduce by \(A \rightarrow \alpha \beta\), \(t\) is also expected to be the symbol following \(A\).
  - If an item is in the form of \([A \rightarrow \alpha \cdot , t]\), reduce with \(A \rightarrow \alpha\) only if next input symbol is \(t\).
Constructing LR(1) DFA – LR(1) Closure

- If $I$ is a set of items of grammar $G$, the LR(1) closure($I$) is set of items constructed with the following operations:
  - if $[A \rightarrow a \cdot B \beta, t]$ is in closure($I$), for every production $B \rightarrow \gamma$ and every $b$ in FIRST($\beta t$) add all items $[B \rightarrow \cdot \gamma, b]$ to closure($I$).
    - Intuitively, $b$ is expected to be following $B$, if $B \rightarrow \gamma$ is used to reduce somewhere along the path from this state.
  - Apply until no more items can be added to closure($I$).
Constructing LR(1) DFA – LR(1) States and Transitions

- The initial state, state$_0$:
  - Add item [$s' \rightarrow \cdot s, \$]$ and its closure to state$_0$.
  - Recall that $\$ is in FOLLOW($s'$).

- Adding a new state:
  - Pick a state $i$, for each item [$A \rightarrow \alpha \cdot X\beta, t$] ($X$ can be any terminals/non-terminals),
    - Create a new state state$_j$ with item [$A \rightarrow \alpha X \cdot \beta, t$], if such a state does not exist.
    - Add the closure of [$A \rightarrow \alpha X \cdot \beta, t$] to state$_j$.
    - Add transition from state$_i$ to state$_j$ on input $X$.

- Repeatedly adding new states until no more states can be added.
An Example of LR(1) DFA

• Consider the following grammar:

\[
\begin{align*}
(1) & \quad S' \to S \\
(2) & \quad S \to CC \\
(3) & \quad C \to cC \mid d
\end{align*}
\]

<table>
<thead>
<tr>
<th>Symbols</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td>c, d</td>
<td>$</td>
</tr>
<tr>
<td>$S$</td>
<td>c, d</td>
<td>$</td>
</tr>
<tr>
<td>$C$</td>
<td>c, d</td>
<td>c, d, $</td>
</tr>
</tbody>
</table>
An Example of LR(1) DFA cont.

- Start with initial item \([S' \to \cdot S, \$$]\) of state \(0\):

1. \(S' \rightarrow S\)
2. \(S \rightarrow CC\)
3. \(C \rightarrow cC \mid d\)

State \(0\):
\[S' \rightarrow \cdot S, \$$\]
An Example of LR(1) DFA cont.

- Add the closure of \([S' \rightarrow \cdot S, \$]\) to state \(0\):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) (S' \rightarrow S)</td>
<td></td>
</tr>
<tr>
<td>(2) (S \rightarrow CC)</td>
<td></td>
</tr>
<tr>
<td>(3) (C \rightarrow cC \mid d)</td>
<td></td>
</tr>
</tbody>
</table>

State \(0\):
\[
S' \rightarrow \cdot S, \$
S \rightarrow \cdot CC, \$
\]

Add Rule \(S \rightarrow CC\), with lookahead symbol in FIRST(\$), which is \{\$\}. 
An Example of LR(1) DFA cont.

- Add the closure of $[S' \rightarrow \cdot S, \$]$ to state $0$:

$\begin{align*}
\text{(1)} & \quad S' \rightarrow S \\
\text{(2)} & \quad S \rightarrow CC \\
\text{(3)} & \quad C \rightarrow cC \mid d
\end{align*}$

State $0$:

$\begin{align*}
S' & \rightarrow \cdot S, \$ \\
S & \rightarrow \cdot CC, \$ \\
C & \rightarrow \cdot cC, c/d
\end{align*}$

Add Rule $C \rightarrow cC$, with lookahead symbol in FIRST$(C\$), which is $\{c, d\}$. To save space, we combine $[C \rightarrow \cdot cC, c]$ and $[C \rightarrow \cdot cC, d]$ into $[C \rightarrow \cdot cC, c/d]$. 

---

CS5363
PL and Compilers
An Example of LR(1) DFA cont.

- Add the closure of \([S' \rightarrow \cdot S, $] \) to state\( _0 \):

(1) \( S' \rightarrow S \)
(2) \( S \rightarrow CC \)
(3) \( C \rightarrow cC \mid d \)

State\( _0 \):
- \( S' \rightarrow \cdot S, \quad $ \)
- \( S \rightarrow \cdot CC, \quad $ \)
- \( C \rightarrow \cdot cC, \quad c/d \)
- \( C \rightarrow \cdot d, \quad c/d \)

Add Rule \( C \rightarrow d \), with lookahead symbol in \( \text{FIRST}(C$), which is \{c, d\}.

CS5363
PL and Compilers
An Example of LR(1) DFA cont.

- Add new state based on \([S' \rightarrow \cdot S, \$]\) of state \(0\):

\[
\begin{align*}
(1) & \quad S' \rightarrow S \\
(2) & \quad S \rightarrow CC \\
(3) & \quad C \rightarrow cC \mid d
\end{align*}
\]

State \(0\):
\[
\begin{align*}
S' & \rightarrow \cdot S, \$ \\
S & \rightarrow \cdot CC, \$ \\
C & \rightarrow \cdot CC, c/d \\
C & \rightarrow \cdot d, c/d
\end{align*}
\]

State \(1\):
\[
\begin{align*}
S' & \rightarrow S \cdot, \$
\end{align*}
\]

From state \(0\) to state \(1\) on \(S\).
An Example of LR(1) DFA cont.

- Add new state based on \([S \to \cdot CC, \$]\) of state \(0\):

  (1) \(S' \to S\)

  (2) \(S \to CC\)

  (3) \(C \to cC \mid d\)

  State: \(0\):
  - \(S' \to \cdot S, \$\)
  - \(S \to \cdot CC, \$\)
  - \(C \to cC, c/d\)
  - \(C \to \cdot d, c/d\)

  State: \(2\):
  - \(S \to C \cdot C, \$\)

  From state \(0\) to state \(2\) on \(C\).
An Example of LR(1) DFA cont.

- Add closure of \([S \rightarrow C \cdot C, \$$] to state_2:

  \begin{align*}
  (1) & \quad S' \rightarrow S \\
  (2) & \quad S \rightarrow CC \\
  (3) & \quad C \rightarrow cC \mid d
  \end{align*}

  \begin{align*}
  \text{State}_0: & \quad S' \rightarrow \cdot s, \quad \$$ \\
  & \quad s \rightarrow \cdot CC, \quad \$$ \\
  & \quad C \rightarrow \cdot cC, \quad c/d \\
  & \quad C \rightarrow \cdot d, \quad c/d
  \end{align*}

  Add Rule \( C \rightarrow cC \), with lookahead symbol in \( \text{FIRST}(\$$), which is \{\$$\}.

  Add Rule \( C \rightarrow d \), with lookahead symbol in \( \text{FIRST}(\$$), which is \{\$$\}.
An Example of LR(1) DFA cont.

- Add new state based on \([C \to \cdot cC, \$]\) of state_0:
  
  \[(1) \ S' \to S\]
  \[(2) \ S \to CC\]
  \[(3) \ C \to cC \mid d\]

---

State_0:

- \(S' \to \cdot S, \$\)
- \(S \to \cdot CC, \$\)
- \(C \to \cdot cC, c/d\)
- \(C \to \cdot d, c/d\)

State_2:

- \(S \to C \cdot C, \$_\)
- \(C \to \cdot cC, \$\)
- \(C \to \cdot d, \$\)

State_3:

- \(C \to \cdot C, c/d\)

From state_0 to state_2 on c.
An Example of LR(1) DFA cont.

- Add new state based on \([C\rightarrow c \cdot C, c/d]\) of state_0:

(1) \(S' \rightarrow S\)
(2) \(S \rightarrow CC\)
(3) \(C \rightarrow cC \mid d\)

State_0:
- \(S' \rightarrow S\)
- \(S \rightarrow CC\)
- \(C \rightarrow cC, c/d\)
- \(C \rightarrow d, c/d\)

State_2:
- \(S \rightarrow C \cdot C, \$
- \(C \rightarrow cC, \$
- \(C \rightarrow d, \$

State_3:
- \(C \rightarrow c \cdot C, c/d\)

From state_0 to state_2 on \(c\).
An Example of LR(1) DFA cont.

- Add the closure of \([C \rightarrow \cdot cC, c/d]\) to state \(_3\):

\begin{align*}
(1) & \quad S' \rightarrow S \\
(2) & \quad S \rightarrow CC \\
(3) & \quad C \rightarrow cC \mid d
\end{align*}

State\(_0\):

\begin{align*}
S' & \rightarrow \cdot S, \quad $ \\
S & \rightarrow \cdot CC, \quad $ \\
C & \rightarrow \cdot cC, \quad c/d \\
C & \rightarrow \cdot d, \quad c/d
\end{align*}

State\(_1\):

\begin{align*}
S & \rightarrow \cdot S, \quad $ \\
S & \rightarrow \cdot CC, \quad $ \\
C & \rightarrow \cdot cC, \quad c/d \\
C & \rightarrow \cdot d, \quad c/d
\end{align*}

State\(_3\):

\begin{align*}
C & \rightarrow c \cdot C, \quad c/d \\
C & \rightarrow \cdot cC, \quad c/d \\
C & \rightarrow \cdot d, \quad c/d
\end{align*}

Add Rule \(C \rightarrow cC\), with lookahead symbol in \(FIRST(c/d)\), which is \{c/d\}.

Add Rule \(C \rightarrow d\), with lookahead symbol in \(FIRST(c/d)\), which is \{c/d\}.
An Example of LR(1) DFA cont.

• The rest of the DFA:

State₀:
S' → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c/d

State₁:
S' → S ·, $
S → .CC ·, $
C → .cC ·, $
C → .d ·, $

State₂:
S → C ·C, $
C → .cC ·, $
C → .d ·, $

State₃:
C → c ·C ·, c/d
C → .cC ·, c/d
C → .d ·, c/d

State₄:
C → d ·, c/d

State₅:
S → .CC ·, $
S → .CC ·, $
C → .cC ·, $
C → .d ·, $

State₆:
C → c ·C ·, $
C → .cC ·, $
C → .d ·, $

State₇:
C → d ·, $

State₈:
C → cC ·, c/d

State₉:
C → cC ·, $
C → .cC ·, $
C → .d ·, $

CS5363
PL and Compilers
LR(1) Parsing Table

• LR(1) parsing table is similar as SLR(1) table.
• As with LR(0) and SLR(1), use action and goto tables.
• goto table defined as before; encodes transition table as map from (state, token) to states.
• action table maps pairs (state, lookahead) to actions.
LR(1) Parsing Table Construction

• For each state X:
  - If there is an item \([A \to \omega \cdot, t]\), \(\text{action}[X, \ t] = \text{reduce } A \to \omega\).
  - If there is an item \([S' \to S \cdot, \$]\) where \(S'\) is the new start symbol of \(G'\) and \(S\) is old start symbol of \(G\), set \(\text{action}[X, \ \$] = \text{accept}\).
  - If there is a transition out of \(X\) on terminal symbol \(t\) to state \(Y\), set \(\text{action}[X, \ t] = \text{shift } Y\).
  - If there is a transition out of \(X\) on non-terminal symbol \(N\) to state \(Y\), set \(\text{goto}[X, \ N] = \text{state } Y\).

• Set all other actions to error.

• If any table entry contains two or more actions, the grammar is not LR(0).
An Example of LR(1) Parsing Table

- The parsing table for the previous grammar and its DFA is,

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s6</td>
<td>s7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) $S' \rightarrow S$
(2) $S \rightarrow CC$
(3) $C \rightarrow cC \mid d$
(4) $C \rightarrow d$
The LR(1) Parsing Algorithm

- LR(1) Algorithm is similar to SLR(1) and LR(0). Basically follow the table.
- Maintain a stack of in the format of \( ($s_0 X_1 s_1 X_2 ... X_m s_m$), which is initially in state \( s_0 \) ($0$).
- While the stack is not empty:
  - Let state \( s_i \) be the top state, and \( t_i \) is the next input symbol.
  - If \( \text{action}[s_i, t_i] \) is SJ (i.e., shift and goto state \( j \), the “S” here means shift instead of state):
    - Push pair \( (t_i, s_j) \) atop the stack.
  - If \( \text{action}[s_i, t_i] \) is reduce \( A \rightarrow \omega \):
    - Pop \( |\omega| \) symbols and states from the top of the stack.
    - Let top-state \( s_k \) be the state on top of the stack after pop.
    - Push \( (A, \text{goto}[s_k, A]) \) atop the stack.
  - Otherwise, report an error.
An Example of LR(1) Parsing

- Consider the previous grammar and its parsing table. To parse string “cccd”.

<table>
<thead>
<tr>
<th>State</th>
<th>c</th>
<th>d</th>
<th>$</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Acc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>s7</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s6</td>
<td>s7</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack | inputs | Action
$0   | cddc $ | Shift 3
$0 c3 | dcd $ | Shift 4
$0 c3 d4 | cd $ | Reduce (4)
$0 c3 C | cd $ | Goto 8
$0 c3 C 8 | cd $ | Reduce (C)
$0 C | cd $ | Goto (C)
$0 C 2 | cd $ | Shift 2
$0 C 2 c6 | d $ | Shift 6
$0 C 2 c6 d7 | $ | Reduce (4)
$0 C 2 c6 C | $ | Goto 9
$0 C 2 c6 C 9 | $ | Reduce (3)
$0 C 2 C | $ | Goto 5
$0 C 2 C 5 | $ | Reduce (2)
$0 S | $ | Goto 1
$0 S 1 | | Accept
The Power of LR(1)

- Any LR(0) grammar is LR(1).
- Any LL(1) grammar is LR(1).
- Any deterministic CFL (a CFL parse-able by a deterministic pushdown automaton) has an LR(1) grammar.
The Old Practical Problem of LR(1)

- LR(1) Automata are Huge.
- In a grammar with n terminals, could in theory be $O(2^n)$ times as large as the LR(0) automaton.
- Replicate each state with all $O(2^n)$ possible lookaheads.
- LR(1) tables for practical programming languages can have hundreds of thousands or even millions of states.
- In the old days, the huge LR(1) DFA was a headache for machines with small memory.
  - Program were designed to be not LR(1). E.g., Java 1.0 was LALR(1).
- Modern computers are more powerful, modern languages are way more complex. LR(1) is not that bad for contemporary systems and languages.
  - C++ is even too complex for LR(1).
LALR Parser
LALR Parser

- **Lookahead(1) LR Parser.**
  - Well, LR(1) also uses lookahead…

- LALR parser often has smaller tables than LR(1) parser, makes it popular for practical use.

- LALR parser is based on LR(1) by consolidating LR(1) states.
Intuition of LALR

• The core item of a state:
  – If \([A \rightarrow \alpha \cdot B \beta, \text{t}]\) is an item of a state, \(A \rightarrow \alpha \cdot B \beta\) is the core of this item.

• If we look at the LR(1) states from the previous example, we can see state\(_3\) and state\(_6\) have exactly the same core items. The same is true for state\(_4\) – state\(_7\), and state\(_8\) – state\(_9\).
  – Sometime it is OK to merge two states with the same core items to save space.
Merging LR(1) States to Create LALR States

1. We can merge two states \( \text{state}_i \) and \( \text{state}_j \) in LR(1) with the same core items into one LALR state \( \text{state}_{ij} \) by unionizing them.

<table>
<thead>
<tr>
<th>State (_3):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{C} \rightarrow \text{c} \cdot \text{C}, \text{c/d} )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{cC}, \text{c/d} )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{d}, \text{c/d} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State (_6):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{C} \rightarrow \text{c} \cdot \text{C}, \cdot )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{cC}, \cdot )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{d}, \cdot )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State (_{36}):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{C} \rightarrow \text{c} \cdot \text{C}, \text{c/d/} )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{cC}, \text{c/d/} )</td>
</tr>
<tr>
<td>( \text{C} \rightarrow \cdot \text{d}, \text{c/d/} )</td>
</tr>
</tbody>
</table>
Merging LR(1) States to Create LALR States cont.

2. For any transition that ends at state \( i \) or state \( j \), it now ends at state \( ij \).

3. For any transition the starts from state \( i \) or state \( j \), it now starts from state \( ij \).

State 3:
- \( C \rightarrow c \cdot C, \ c/d \)
- \( C \rightarrow \cdot cC, \ c/d \)
- \( C \rightarrow \cdot d, \ c/d \)

State 6:
- \( C \rightarrow c \cdot C, \ $ \)
- \( C \rightarrow \cdot cC, \ $ \)
- \( C \rightarrow \cdot d, \ $ \)

State 36:
- \( C \rightarrow c \cdot C, \ c/d/$
- \( C \rightarrow \cdot cC, \ c/d/$
- \( C \rightarrow \cdot d, \ c/d/$

From \( S_0 \) to \( S_8 \)
From \( S_2 \) to \( S_8 \)
From \( S_2 \) to \( S_7 \)
To \( S_8 \)
To \( S_7 \)
An Example of Converting LR(1) DFA to LALR DFA

- After the merges:

State_0:
- $S' \rightarrow S'$,
- $S \rightarrow CC'$,
- $C \rightarrow CC', c/d$
- $C \rightarrow d', c/d$

State_1:
- $S' \rightarrow S', s$

State_2:
- $S \rightarrow CC'$,
- $C \rightarrow cC', $,
- $C \rightarrow d', c/d$

State_36:
- $C \rightarrow cC', c/d/d/$
- $C \rightarrow cC', c/d/d/$
- $C \rightarrow d', c/d/d/$

State_47:
- $C \rightarrow d', c/d/d/$

State_5:
- $S \rightarrow CC', $

State_89:
- $C \rightarrow cC', c/d/d/$
LALR Parsing Table

- LALR parsing table is constructed similarly as LR(1)
- For example, for the previous LALR DFA, we have parsing table,

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>36</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>47</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>
LALR Parsing Algorithm

- LALR parsing algorithm is the same as LR(1).
- E.g., given the previous table, we can parse “cdcd”,

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Acc</td>
</tr>
<tr>
<td>2</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>36</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>47</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>89</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>inputs</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>c</td>
<td>Shift 36</td>
</tr>
<tr>
<td>$0 c 36</td>
<td>dcd</td>
<td>$</td>
</tr>
<tr>
<td>$0 c 3 d 47</td>
<td>cd</td>
<td>$</td>
</tr>
<tr>
<td>$0 c 3 C</td>
<td>cd</td>
<td>$</td>
</tr>
<tr>
<td>$0 c 3 C 89</td>
<td>cd</td>
<td>$</td>
</tr>
<tr>
<td>$0 C</td>
<td>cd</td>
<td>$</td>
</tr>
<tr>
<td>0 C 2</td>
<td>cd</td>
<td>$</td>
</tr>
<tr>
<td>0 C 2 c 36</td>
<td>d</td>
<td>$</td>
</tr>
<tr>
<td>0 C 2 c 36 d 47</td>
<td>$</td>
<td>Reduce (4)</td>
</tr>
<tr>
<td>0 C 2 c 36 C</td>
<td>$</td>
<td>Goto 89</td>
</tr>
<tr>
<td>0 C 2 c 36 C 89</td>
<td>$</td>
<td>Reduce (3)</td>
</tr>
<tr>
<td>0 C 2 C</td>
<td>$</td>
<td>Goto 5</td>
</tr>
<tr>
<td>0 C 2 C 5</td>
<td>$</td>
<td>Reduce (2)</td>
</tr>
<tr>
<td>0 S</td>
<td>$</td>
<td>Goto 1</td>
</tr>
<tr>
<td>0 S 1</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
LALR is NOT LR(1)

- LALR is less powerful than LR(1), because it removed some useful information in the states by merging.

- After merging, there may be reduce/reduce conflicts in the parsing table.
  - There will be no new shift/reduce conflicts as we only touch the following symbols in the states.
Efficient Construction of LALR

- Creating a complete set of LR(1) states is tedious.
- We can create LALR one the fly by slightly changing the LR(1) DFA generation algorithm.
  - Pick a state $i$, for each item $[A \rightarrow \alpha \cdot x \beta \cdot t]$ ($x$ can be any terminals/non-terminals),
    - If there is a state $j$ with an item $[A \rightarrow \alpha x \cdot \beta \cdot a]$, add $[A \rightarrow \alpha x \cdot \beta \cdot t]$ to state $j$ to create item $[A \rightarrow \alpha x \cdot \beta \cdot a/t]$.
    - if no such a state exists, create a new state state $j$ with item $[A \rightarrow \alpha x \cdot \beta \cdot t]$,
    - Add the closure of $[A \rightarrow \alpha x \cdot \beta \cdot t]$ to state $j$.
    - Add transition from state $i$ to state $j$ on input $x$. 
An Example of LALR(1) DFA cont.

- Create state₀ from item \([S' \rightarrow \cdot S, \$]\):

1. \(S' \rightarrow S\)
2. \(S \rightarrow CC\)
3. \(C \rightarrow cC \mid d\)

State₀:
- \(S' \rightarrow \cdot S, \$\)
- \(S \rightarrow \cdot CC, \$\)
- \(C \rightarrow \cdot cC, c/d\)
- \(C \rightarrow \cdot d, c/d\)
An Example of LALR DFA cont.

- Create state $s_1$ from item $[S' → · S, \$]$ in state $s_0$:

(1) $S' → S$
(2) $S → CC$
(3) $C → cC \mid d$

State $s_0$:
- $S' → · S, \$$
- $S → · CC, \$
- $C → · cC, c/d$
- $C → · d, c/d$

State $s_1$:
- $S' → · S, \$

CS5363
PL and Compilers
An Example of LR(1) DFA cont.

- Add state $S_2$ based on $[S \rightarrow C \cdot C, \$]$ of state $S_0$:

(1) $S' \rightarrow S$
(2) $S \rightarrow CC$
(3) $C \rightarrow cC \mid d$

State $S_0$:
- $S' \rightarrow \cdot S, \$
- $S \rightarrow \cdot CC, \$
- $C \rightarrow \cdot cC, c/d$
- $C \rightarrow \cdot d, c/d$

State $S_1$:
- $S' \rightarrow \cdot S, \$

State $S_2$:
- $S \rightarrow C \cdot C, \$
- $C \rightarrow \cdot cC, \$
- $C \rightarrow \cdot d, \$

CS5363
PL and Compilers
An Example of LR(1) DFA cont.

- Add state $3$ based on $[C \rightarrow \cdot cC, \$]$ of state $0$:

  \[(1) \ S' \rightarrow S \]
  \[(2) \ S \rightarrow CC \]
  \[(3) \ C \rightarrow cC \mid d \]

  \begin{align*}
  \text{State}_0: \\
  & S' \rightarrow S \cdot, \$ \\
  & S \rightarrow \cdot CC, \$
  & C \rightarrow \cdot cC, c/d \\
  & C \rightarrow \cdot d, c/d
  \end{align*}

  \begin{align*}
  \text{State}_2: \\
  & S \rightarrow \cdot CC, \$
  & C \rightarrow \cdot cC, \$
  & C \rightarrow \cdot d, \$
  \end{align*}

  \begin{align*}
  \text{State}_3: \\
  & C \rightarrow \cdot C, c/d \\
  & C \rightarrow \cdot cC, c/d \\
  & C \rightarrow \cdot d, c/d
  \end{align*}
An Example of LR(1) DFA cont.

- Extend state\(_3\) based on \([S \rightarrow C \cdot C, \$_\)] of state\(_2\):

\[\begin{align*}
(1) & \quad S' \rightarrow S \\
(2) & \quad S \rightarrow CC \\
(3) & \quad C \rightarrow cC \mid d
\end{align*}\]

State\(_0\):
- \(S' \rightarrow \cdot S, \$_\)
- \(S \rightarrow \cdot CC, \$_\)
- \(C \rightarrow \cdot cC, \cdot d, \cdot c/d\)

State\(_2\):
- \(S \rightarrow C \cdot C, \$_\)
- \(C \rightarrow \cdot cC, \cdot c/d\)
- \(C \rightarrow \cdot d, \$_\)

State\(_3\):
- \(C \rightarrow c \cdot C, \cdot c/d/\$_\)
- \(C \rightarrow \cdot cC, \cdot c/d\)
- \(C \rightarrow \cdot d, \cdot c/d\)
An Example of LR(1) DFA cont.

- **Extend state** \( \text{state}_3 \) based on \([C \rightarrow \cdot cC, \$] \) of state \( \text{state}_2 \):

  \begin{align*}
  (1) & \quad S' \rightarrow S \\
  (2) & \quad S \rightarrow CC \\
  (3) & \quad C \rightarrow cC \mid d
  \end{align*}

State \( \text{state}_0 \):
- \( S' \rightarrow \cdot S, \$ \)
- \( S \rightarrow \cdot CC, \$ \)
- \( C \rightarrow \cdot cC, c/d \)
- \( C \rightarrow \cdot d, c/d \)

State \( \text{state}_2 \):
- \( S' \rightarrow \cdot S, \$ \)
- \( S \rightarrow \cdot CC, \$ \)
- \( C \rightarrow \cdot cC, \$ \)
- \( C \rightarrow \cdot d, \$ \)

State \( \text{state}_3 \):
- \( C \rightarrow \cdot cC, c/d/\$ \)
- \( C \rightarrow \cdot cC, c/d \)
- \( C \rightarrow \cdot d, c/d \)

State \( \text{state}_2 \) transits to state \( \text{state}_3 \) on \( c \), due to item \([C \rightarrow \cdot cC, \$]\) of state \( \text{state}_2 \).

Extend item \([C \rightarrow \cdot cC, c/d]\) to item \([C \rightarrow \cdot cC, c/d/\$]\).
An Example of LR(1) DFA cont.

- Add closure of \([C \rightarrow c \cdot C, c/d/\$]\) to state 3:

  (1) \(S' \rightarrow S\)
  (2) \(S \rightarrow CC\)
  (3) \(C \rightarrow cC \mid d\)

  State 0:
  - \(S' \rightarrow S\), \$\n  - \(S \rightarrow CC\), \$\n  - \(C \rightarrow cC\), \(c/d\)
  - \(C \rightarrow d\), \(c/d\)

  State 1:
  - \(S' \rightarrow S\), \$\n  - \(S \rightarrow CC\), \$\n  - \(C \rightarrow cC\), \(c/d\)
  - \(C \rightarrow d\), \(c/d\)

  State 2:
  - \(S \rightarrow CC\), \$\n  - \(C \rightarrow cC\), \(c/d\)\$
  - \(C \rightarrow d\), \(c/d\)\$

  State 3:
  - \(C \rightarrow c \cdot C\), \(c/d/\$\)
  - \(C \rightarrow cC\), \(c/d/\$\)
  - \(C \rightarrow d\), \(c/d/\$\)

Add \$\ to the rest two items.
Parsing in Practice
A Hierarchy of Grammar Classes

Unambiguous Grammars

- LR(k)
- LR(1)
- LALR
- SLR(1)
- LR(0)

Ambiguous Grammars

- LL(k)
- LL(1)
- LL(0)

CS5363
PL and Compilers
Parsing is Very Very Hard

• Modern programming languages are too complex.
  – Some are inherently ambiguities, e.g., the dangling else.
  – Some are strangely context-sensitive, e.g., C++.
  – There are not just simple statements, but also other compiler directives in the code, e.g., macros and pragmas.
  – After all, programming languages are designed for human, not machines.

• Real compilers cannot completely rely on theoretical LL or LR parsers.
  – Even with bison’s LR parser, we still have to rely on shift-over-reduce, or first-rule-reduce, to resolve conflicts in real programming languages.
  – GCC parser was built with bison, but after 2004, it uses a hand-write recursive descent parser (tree-traversal based push-down automata). Hand-write implementation is easier to tune to handle strange things such as ambiguities and contexts.
  – Clang also uses a manually implemented recursive descent parser.

• Parser performance (execution time) is still a concern as the size of code base keeps growing.
Acknowledgment

- This lectures is partially based on the Compiler slides of Keith Schwarz and the Dragon book.