Lecture 16
SLR, LR(1) and LALR
Instructor: Hossein Hojjat
February 21, 2018
Consider the grammar: $S \rightarrow (S) \mid \text{num}$
For each state:

- Transition to another state using a terminal symbol is a **shift** to that state.
- Transition to another state using a non-terminal is a **goto** to that state.
- If there is a single item $A \rightarrow \alpha\cdot$ in the state **reduce** with that production for all terminals.
Building Parse Table Example

<table>
<thead>
<tr>
<th></th>
<th>(   )</th>
<th>num</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td>s3</td>
<td></td>
<td>g1</td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>s3</td>
<td></td>
<td>g4</td>
</tr>
<tr>
<td>3</td>
<td>r(S→num)</td>
<td>r(S→num)</td>
<td>r(S→num)</td>
<td>r(S→num)</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r(S→(S))</td>
<td>r(S→(S))</td>
<td>r(S→(S))</td>
<td>r(S→(S))</td>
</tr>
</tbody>
</table>

```
S' → •S$
S → •(S)
S → •num
S → (•S)
S → •(S)
S → •num
S → (S)$
```
• LR(0) only works if states with reduce actions have a single reduce action

\[
E \rightarrow T. \]

• In those states it always reduce without looking at lookahead

• LR(0) is vulnerable to unnecessary conflicts

• Shift/Reduce Conflicts (may reduce too soon in some cases)

\[
E \rightarrow E \cdot +T \\
S \rightarrow E. 
\]

• Reduce/Reduce Conflicts

\[
E \rightarrow \text{num}. \\
T \rightarrow \text{num}. 
\]
LR(0) Parsing Table With Conflicts

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>+</th>
<th>num</th>
<th>$</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td></td>
<td>s4</td>
<td></td>
<td>g2</td>
<td>g1</td>
</tr>
<tr>
<td>1</td>
<td>r1</td>
<td>r1</td>
<td>r1/s6</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s8</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
E
S' → •S$
S → •E
S → •E + num
E → •(E)
S → •num

S

S → E •+ num
S → E •
E → •num
S

S' → S •$
S' → S •$
S → E •+ num
S → E •
E → •num
S

S → E + num •
S

S → E + num •
S

S → E + num •
S
```

Rules:
1. $r_1$  $S → E$
2. $r_2$  $E → E + num$
3. $r_3$  $E → (E)$
4. $r_4$  $E → num$
Simple LR parsing (SLR) is a simple extension of LR(0) parsing.

For each reduction $A \rightarrow \gamma$, look at the lookahead symbol $c$.

Apply reduction only if $c$ is in FOLLOW($A$).

**SLR Parsing Table**

- Eliminates some conflicts.
- Same as LR(0) table except reduction rows.
- Reductions do not fill entire rows.
- Add reductions $A \rightarrow \gamma$ only in the columns of symbols in FOLLOW($A$).
LR(0) Parsing Table

<table>
<thead>
<tr>
<th>( )</th>
<th>+</th>
<th>num</th>
<th>$</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3</td>
<td>s4</td>
<td>g2</td>
<td>g1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1/s6</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>s4</td>
<td></td>
<td>g5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FOLLOW(S) = $
FOLLOW(E) = \{+, (\), $\}

r1 \quad S \rightarrow E
r2 \quad E \rightarrow E + num
r3 \quad E \rightarrow (E)

r4 \quad E \rightarrow num
### SLR Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>+</th>
<th>num</th>
<th>$</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s4</td>
<td>g2</td>
<td>g1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s6</td>
<td>r1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s4</td>
<td>g5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s8</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FOLLOW(S) = $
FOLLOW(E) = \{+, ), $\}$

**Rules:**

- **r1**  $S \rightarrow E$
- **r2**  $E \rightarrow E + \text{num}$
- **r3**  $E \rightarrow (E)$
- **r4**  $E \rightarrow \text{num}$
LR(1) Parsing

- **Idea**: Get as much as possible out of 1 lookahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 lookahead
- LR(1) parsing uses similar concepts as LR(0)
- Parser states = set of LR(1) items
- LR(1) item = LR(0) item + lookahead symbols possibly following production
- LR(0) item: $S \rightarrow \cdot S + E$
- LR(1) item: $S \rightarrow \cdot S + E$, +
- Lookahead only has impact on reduce operations: apply when lookahead = next input
• LR(1) state = set of LR(1) items
• LR(1) item = $(X \rightarrow \alpha \cdot \beta, y)$
• Meaning: $\alpha$ already matched at top of the stack, next expect to see $\beta y$
• Shorthand notation: $(X \rightarrow \alpha \cdot \beta, \{x_1, \cdots, x_n\})$ means:
  - $(X \rightarrow \alpha \cdot \beta, x_1)$
  - $\cdots$
  - $(X \rightarrow \alpha \cdot \beta, x_n)$
• Need to extend closure and goto operations
LR(1) Closure

Similar to LR(0) closure, but also keeps track of lookahead symbol

If $L$ is a set of items, CLOSURE($L$) is the set of items such that:

- every item in $L$ is in CLOSURE($L$)
- if item $(X \rightarrow \alpha \cdot Y \beta, z)$ is in CLOSURE($L$) and $Y \rightarrow \gamma$ is a production then $(Y \rightarrow \cdot \gamma, \text{FIRST}(\beta z))$ is also in CLOSURE($L$)
LR(1) Start State

Initial state: start with $(S' \rightarrow S, \$$), then apply closure operation

Goto is analogous to goto in LR(0) parsing

**Goto(L, X)**

$I = \emptyset$

for any item $[A \rightarrow \alpha \cdot X \beta, x]$ in $L$

$I = I \cup \{[A \rightarrow \alpha X \cdot \beta, x]\}$

return CLOSURE$(I)$
Exercise

Construct the LR(1) automaton for the following grammar:

\[ S' \rightarrow S\$
\[ S \rightarrow E + S \mid E \]
\[ E \rightarrow \text{num} \]
### LR(0) Automaton Example

#### LR(0) Automaton Example

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>×</th>
<th>num</th>
<th>$</th>
<th>S</th>
<th>R</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>$</td>
<td>S</td>
<td>R</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>r5</td>
<td>r5/a8</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>g7</td>
<td>g6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>g9</td>
<td>g6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LR(0) Grammar

- **r1**: $S \rightarrow L \times R$
- **r2**: $S \rightarrow R$
- **r3**: $L \rightarrow \text{num}$
- **r4**: $L \rightarrow + R$
- **r5**: $R \rightarrow L$

#### LR(0) Rules

- $S' \rightarrow S\$
- $S \rightarrow \cdot L \times R$
- $S \rightarrow \cdot R$
- $S \rightarrow \cdot num$
- $L \rightarrow \cdot L$
- $L \rightarrow \cdot + R$
- $L \rightarrow \cdot num$
- $R \rightarrow \cdot L$
- $R \rightarrow \cdot R$
- $R \rightarrow \cdot L$
- $R \rightarrow \cdot num$
- $L \rightarrow \cdot + R$
- $L \rightarrow \cdot + R$
- $L \rightarrow \cdot num$

#### FOLLOW

- FOLLOW($S$) = $\{\times, $
- FOLLOW($L$) = $\{\}
- FOLLOW($R$) = $\{\times, $
SLR Automaton Example

\[
\begin{array}{cccccc}
+ & \times & \text{num} & $ & S & R & L \\
\hline
s5 & s4 & g3 & g2 & g1 \\
\hline
r5/s8 & r5 & \\
r2 & \\
\hline
\text{accept} & \\
\hline
r3 & r3 \\
\hline
s5 & s4 & g7 & g6 \\
r5 & r5 \\
r4 & r4 \\
s5 & s4 & g9 & g6 \\
r1 & \\
\end{array}
\]

\[
\text{FOLLOW}(S) = \$
\text{FOLLOW}(L) =
\text{FOLLOW}(R) = \{\times, $\}
\]

\[
\begin{array}{c}
r1 & S \rightarrow L \times R \\
r2 & S \rightarrow R \\
r3 & L \rightarrow \text{num} \\
r4 & L \rightarrow \text{+} R \\
r5 & R \rightarrow \text{L} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & S' \rightarrow \cdot S$ \\
1 & R \rightarrow \cdot L \\
2 & S \rightarrow \cdot R \\
3 & L \rightarrow \text{num} \\
4 & L \rightarrow \text{+} R \\
5 & S \rightarrow \text{L} \times \text{R} \\
6 & R \rightarrow \cdot L \\
7 & L \rightarrow \text{+} R \\
8 & S \rightarrow \text{L} \times \text{R} \\
9 & R \rightarrow \text{L} \times \text{R} \\
\end{array}
\]
Grammar is not LR(0) and SLR, but it is LR(1)

There is no more shift/reduce conflict in the automaton:
LALR

- Drawback: LR(1) parse engine has a large number of states
- LALR (Look-Ahead LR parser): Simple technique to eliminate states
- If two LR(1) states are identical except for the look ahead symbol of their items, merge them
- Result is LALR(1) DFA
- It is more memory efficient, typically merges several LR(1) states
- May also have more reduce/reduce conflicts
- Power of LALR parsing is enough for many mainstream computer languages
- Several automatic parser generators such as Yacc or GNU Bison
Consider for example these two LR(1) states

\[
\begin{align*}
  X &\rightarrow \alpha \cdot, a \\
  Y &\rightarrow \beta \cdot, c
\end{align*}
\]

\[
\begin{align*}
  X &\rightarrow \alpha \cdot, b \\
  Y &\rightarrow \beta \cdot, d
\end{align*}
\]

They will be merged into the following LALR(1) states

\[
\begin{align*}
  X &\rightarrow \alpha \cdot, \{a, b\} \\
  Y &\rightarrow \beta \cdot, \{c, d\}
\end{align*}
\]
“Modern Compiler Implementation in Java”,
Andrew W. Appel, Jens Palsberg