Compilers

LR & LALR Parsing Tables
Canonical sets of LR(1) items

Number of states much larger than in the SLR construction

\[
\begin{align*}
\text{LR}(1) &= \text{Order of thousands for a standard prog. Lang.} \\
\text{SLR}(1) &= \text{Order of hundreds for a standard prog. Lang.}
\end{align*}
\]

LALR(1) (lookahead-LR)

A tradeoff:

- Collapse states of the LR(1) table that have the same core (the “LR(0)” part of each state)
- LALR never introduces a Shift/Reduce Conflict if LR(1) doesn’t.
- It might introduce a Reduce/Reduce Conflict (that did not exist in the LR(1))...
- Still much better than SLR(1) (larger set of languages)
- ... but smaller than LR(1)

What Yacc and most compilers employ.
Conflict Example

\[
\begin{align*}
S & \rightarrow L=R & I_6: & S' \rightarrow .S & I_1: & S' \rightarrow S. & I_2: & S \rightarrow L=R \rightarrow L. & I_3: & S \rightarrow R. \\
S & \rightarrow R & & S \rightarrow L=R & & R \rightarrow .L \\
L & \rightarrow *R & & S \rightarrow .R & & L \rightarrow .*R \\
L & \rightarrow id & & L \rightarrow .*R & & L \rightarrow .id \\
R & \rightarrow L & & L \rightarrow .id & & R \rightarrow L. \\
R & \rightarrow L & & L \rightarrow .id & & I_4: L \rightarrow .*R \\
\end{align*}
\]

Problem

FOLLOW(R) = \{=,$\}

= \text{shift 6} \quad \text{reduce by } R \rightarrow L \quad \text{shift/reduce conflict} \\
I_7: & L \rightarrow .*R \\
I_8: & R \rightarrow L. \\
I_5: & L \rightarrow id. \\
I_4: & L \rightarrow .*R \\
I_7: & L \rightarrow .*R.
S → AaAb
S → BbBa
A → ε
B → ε

Problem
FOLLOW(A)={a,b}
FOLLOW(B)={a,b}

a → reduce by A → ε
reduce by B → ε
reduce/reduce conflict

b → reduce by A → ε
reduce by B → ε
reduce/reduce conflict
In SLR method, the state $i$ makes a reduction by $A \rightarrow \alpha$ when the current token is $a$:

$$\text{if the } A \rightarrow \alpha \text{ in the } I_i \text{ and } a \text{ is FOLLOW}(A)$$

In some situations, $\beta A$ cannot be followed by the terminal $a$ in a right-sentential form when $\beta \alpha$ and the state $i$ are on the top stack. This means that making reduction in this case is not correct.

**Grammar:**

1) $S \rightarrow AaAb$

2) $S \rightarrow BbBa$

3) $A \rightarrow \epsilon$

4) $B \rightarrow \epsilon$

**RMD:**

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S $\Rightarrow$ AaAb $\Rightarrow$ Aab $\Rightarrow$ ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S $\Rightarrow$ BbBa $\Rightarrow$ Bba $\Rightarrow$ ba</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parse (Reverse of RMD):**

3) $Aab \Rightarrow \epsilon \ ab$ (correct follow is a)

4) $AaAb \Rightarrow Aa \ \epsilon \ b$ (correct follow is b)

**Parse (Reverse of RMD):**

3) $Bba \Rightarrow \epsilon \ ba$

4) $BbBa \Rightarrow Bb \ \epsilon \ a$
LR(1) Items

To avoid some of invalid reductions, the states need to carry more information. Extra information is put into a state by including a terminal symbol as a second component in an item.

A LR(1) item is:

\[ A \rightarrow \alpha \cdot \beta, a \]

where \( a \) is the look-head of the LR(1) item (\( a \) is a terminal or end-marker.)
LR(1) Items

- When $\beta$ (in the LR(1) item $A \rightarrow \alpha \cdot \beta, a$) is not empty, the look-head $a$ does not have any affect.
- When $\beta$ is empty ($A \rightarrow \alpha \cdot, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is $a$ (not for any terminal in FOLLOW($A$)).
- A state will contain $A \rightarrow \alpha \cdot, a_1$ where \{a_1,...,a_n\} $\subseteq$ FOLLOW($A$)

...
Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

**closure(I)** is: (where I is a set of LR(1) items)
  - every LR(1) item in I is in closure(I)
  - if $A \rightarrow \alpha \cdot B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow \cdot \gamma, b$ will be in the closure(I) for each terminal $b$ in FIRST($\beta a$).
goto operation

• If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  o If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in $\text{closure}({A \rightarrow \alphaX.\beta,a})$ will be in goto(I,X).
Construction of The Canonical LR(1) Collection

- **Algorithm:**
  
  \[ C \text{ is } \{ \text{closure}\({S' \rightarrow \cdot .S, \cdot $}\}) \}\]

  **repeat** the followings until no more set of LR(1) items can be added to \( C \).

  **for each** \( I \) in \( C \) and each grammar symbol \( X \)

  **if** goto(I,X) is not empty and not in \( C \)

  add goto(I,X) to \( C \)

- goto function is a DFA on the sets in \( C \).
A Short Notation for The Sets of LR(1) Items

A set of LR(1) items containing the following items

\[ A \rightarrow \alpha \cdot \beta, a_1 \]

\[ \ldots \]

\[ A \rightarrow \alpha \cdot \beta, a_n \]

can be written as

\[ A \rightarrow \alpha \cdot \beta, \{a_1, a_2, \ldots, a_n\} \]
Canonical LR(1) Collection - Example

\[ S \rightarrow AaAb \]
\[ S \rightarrow BbBa \]
\[ A \rightarrow \varepsilon \]
\[ B \rightarrow \varepsilon \]

\[ I_0: S' \rightarrow .S ,$ \]
\[ S \rightarrow .AaAb ,$ \]
\[ S \rightarrow .BbBa ,$ \]
\[ A \rightarrow . ,a \]
\[ B \rightarrow . ,b \]

\[ I_1: S' \rightarrow S. ,$ \]
\[ S \rightarrow .S ,$ \]
\[ I_2: S \rightarrow A.aAb ,$ \]
\[ a \rightarrow \text{to } I_4 \]
\[ I_3: S \rightarrow B.bBa ,$ \]
\[ b \rightarrow \text{to } I_5 \]

\[ I_4: S \rightarrow Aa.Ab ,$ \]
\[ A \rightarrow . ,b \]

\[ I_6: S \rightarrow AaA.b ,$ \]
\[ a \rightarrow I_8: S \rightarrow AaAb. ,$ \]

\[ I_5: S \rightarrow Bb.Ba ,$ \]
\[ B \rightarrow . ,a \]

\[ I_7: S \rightarrow BbB.a ,$ \]
\[ b \rightarrow I_9: S \rightarrow BbBa. ,$ \]
Canonical LR(1) Collection - Example

S' → S
1) S → L=R
   S → .L=R,$
2) S → R
   S → .R,$
3) L → *R
   L → .*R, {$,=}
4) L → id
   L → .id, {$,=}
5) R → L
   R → .L,$

I_6:S → L=.R,$
   R → .L,$
   L → .*R,$
   L → .id,$
I_7:L → *R., {$,=}
I_8: R → L.,$
I_9:S → L=R.,$
I_10: R → L.,$
I_11: L → .*R.,$
I_12: L → id.,$
I_13: L → *R.,$

R → .L, {$,=}
L → .*R, {$,=}
L → .id, {$,=}

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for $G'$.
   \[ C \leftarrow \{ I_0, \ldots, I_n \} \]

2. Create the parsing action table as follows
   - If $a$ is a terminal, $A \rightarrow \alpha \cdot a \beta, b$ in $I_i$ and $\text{goto}(I_i, a) = I_j$ then action\[i, a]\] is \textit{shift} $j$.
   - If $A \rightarrow \alpha \cdot \cdot a$ is in $I_i$, then action\[i, a]\] is \textit{reduce} $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S \cdot \cdot \$, is in $I_i$, then action\[i, \$]\] is \textit{accept}.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals $A$, if $\text{goto}(I_i, A) = I_j$ then goto\[i, A]\] = $j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow S, \$$. 
**LR(1) Parsing Tables**

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or no reduce/reduce conflict

so, it is a LR(1) grammar
LALR Parsing Tables

- **LALR** stands for **LookAhead LR**.

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.

- The number of states in SLR and LALR parsing tables for a grammar G are equal.

- But LALR parsers recognize more grammars than SLR parsers.

- **YACC** creates a LALR parser for the given grammar.

- A state of LALR parser will be again a set of LR(1) items.
LALR Parsing Tables

Canonical LR(1) Parser $\rightarrow$ LALR Parser  
shrink # of states

- This shrink process may introduce a reduce/reduce conflict in the resulting LALR parser (so the grammar is NOT LALR)

- But, this shrink process does not produce a shift/reduce conflict.
The Core of A Set of LR(1) Items

The core of a set of LR(1) items is the set of its first component.

Ex: \( S \rightarrow L \cdot =R,\$ \) \( \Rightarrow \) \( S \rightarrow L \cdot =R \) (Core)
\( R \rightarrow L \cdot ,\$ \) \( R \rightarrow L \cdot \)

We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

\( I_1: L \rightarrow id \cdot ,= \)

\( \Rightarrow \) A new state: \( I_{12}: L \rightarrow id \cdot ,\{=,\$\} \)

\( I_2: L \rightarrow id \cdot ,\$ \) (have same core, merge the lookaheads)

We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.
Creation of LALR Parsing Tables

• Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
• Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.
  \[ C = \{I_0, \ldots, I_n\} \rightarrow C' = \{J_1, \ldots, J_m\} \text{ where } m \leq n \]
• Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  o Note that: If \( J = I_1 \cup \ldots \cup I_k \) since \( I_1, \ldots, I_k \) have same cores
    \[ \Rightarrow \text{cores of goto}(I_1, X), \ldots, \text{goto}(I_2, X) \] must be same.
  o So, \( \text{goto}(J, X) = K \) where \( K \) is the union of all sets of items having same cores as \( \text{goto}(I_1, X) \).
• If no conflict is introduced, the grammar is LALR(1) grammar.
  (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)
Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.

- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

  \[ A \rightarrow \alpha \cdot , a \quad \text{and} \quad B \rightarrow \beta \cdot a \gamma , b \]

- This means that a state of the canonical LR(1) parser must have:

  \[ A \rightarrow \alpha \cdot , a \quad \text{and} \quad B \rightarrow \beta \cdot a \gamma , c \]

  But, this state has also a shift/reduce conflict.

  i.e. The original canonical LR(1) parser has a conflict.

  (Reason for this, the shift operation does not depend on lookaheads)
Reduce/Reduce Conflict

But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

I_1: A → α • ,a
    B → β • ,b

I_2: A → α • ,b
    B → β • ,c

I_{12}: A → α • ,\{a,b\} \rightarrow \text{reduce/reduce conflict}
    B → β • ,\{b,c\}
Canonical LALR(1) Collection

1) \( S \to L=R \quad I_1: S' \to S \cdot, \$
   \quad I_2: S \to \cdot L=R, \$
   \quad R \to \cdot L, \$
   \quad I_6: L \to \cdot R, \$
   \quad I_4: L \to \cdot R, \$

2) \( S \to R \quad I_1: S' \to S \cdot, \$
   \quad I_2: S \to \cdot R, \$
   \quad R \to \cdot L, \$
   \quad I_6: L \to \cdot R, \$
   \quad I_4: L \to \cdot R, \$

3) \( L \to \cdot R \quad I_2: S \to \cdot L=R, \$
   \quad R \to \cdot L, \$
   \quad I_6: L \to \cdot R, \$
   \quad I_4: L \to \cdot R, \$
   \quad L \to \cdot R, \$
   \quad I_6: L \to \cdot R, \$
   \quad I_4: L \to \cdot R, \$

4) \( L \to \cdot id, \$
   \quad I_3: S \to \cdot R, \$
   \quad L \to \cdot id, \$
   \quad I_5: L \to \cdot R, \$
   \quad I_6: L \to \cdot R, \$

5) \( R \to L \quad I_3: S \to \cdot R, \$
   \quad R \to \cdot L, \$
   \quad I_6: L \to \cdot R, \$
   \quad I_4: L \to \cdot R, \$

Same Cores

\( I_4 \) and \( I_{11} \)

\( I_5 \) and \( I_{12} \)

\( I_7 \) and \( I_{13} \)

\( I_8 \) and \( I_{10} \)
## Canonical LALR(1) Collection

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>411</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>713</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>713</td>
<td></td>
<td></td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>810</td>
<td></td>
<td></td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or no reduce/reduce conflict

so it is a LALR(1) grammar
Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.

- Can we create LR-parsing tables for ambiguous grammars?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.

- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are much natural, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may eliminate unnecessary reductions.

Ex.

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$
- $E \rightarrow E + E \mid E * E \mid (E) \mid id$
Sets of LR(0) Items for Ambiguous Grammar

I₀:   E’ → .E
     E → .E+E
     E → .E*E
     E → .(E)
     E → .id

I₁:   E’ → E.
     E → E .+E
     E → E .*E

I₂:   E → (.E)
     E → .E+E
     E → .E*E
     E → .(E)
     E → .id

I₃:   E → id.

I₄:   E → E .+E
     E → E .*E

I₅:   E → E .*E
     E → .E+E
     E → .E*E
     E → .(E)
     E → .id

I₆:   E → (E .)
     E → E .+E
     E → E .*E

I₇:   E → E+E.

I₈:   E → E*E.

I₉:   E → (E).

LR Parsing
SLR-Parsing Tables for Ambiguous Grammar

\[ \text{FOLLOW}(E) = \{ \$, +, *, ) \} \]

State \( I_7 \) has shift/reduce conflicts for symbols + and *.

When current token is +
- shift \( \Rightarrow \) + is right-associative
- reduce \( \Rightarrow \) + is left-associative

When current token is *
- shift \( \Rightarrow \) * has higher precedence than +
- reduce \( \Rightarrow \) + has higher precedence than *
SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *, ) }

State $I_8$ has shift/reduce conflicts for symbols $+$ and $*$.

\[
I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_8
\]

when current token is $*$

shift $\Rightarrow *$ is right-associative

reduce $\Rightarrow *$ is left-associative

when current token is $+$

shift $\Rightarrow +$ has higher precedence than $*$

reduce $\Rightarrow *$ has higher precedence than $+$
### SLR-Parsing Tables for Ambiguous Grammar

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
</tr>
</tbody>
</table>
Error Recovery in LR Parsing

• An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
• Errors are never detected by consulting the goto table.
• An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
• A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
• The SLR and LALR parsers may make several reductions before announcing an error.
• But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.
Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state \( s \) with a goto on a particular nonterminal \( A \) is found. (Get rid of everything from the stack before this state \( s \)).

- Discard zero or more input symbols until a symbol \( a \) is found that can legitimately follow \( A \).
  - The symbol \( a \) is simply in FOLLOW(\( A \)), but this may not work for all situations.

- The parser stacks the nonterminal \( A \) and the state goto[\( s, A \)], and it resumes the normal parsing.

- This nonterminal \( A \) is normally is a basic programming block (there can be more than one choice for \( A \)).
  - stmt, expr, block, ...
Phrase-Level Error Recovery in LR Parsing

• Each empty entry in the action table is marked with a specific error routine.

• An error routine reflects the error that the user most likely will make in that case.

• An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis
LR Parsing

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

Diagram illustrating the relationship between different LR parsing methods.