LR(0) Parsing Summary

- LR(0) state = set of LR(0) items
- LR(0) item = a production with a dot in RHS
- Compute LR(0) states and build DFA:
  - Use the closure operation to compute states
  - Use the goto operation to compute transitions between states
- Build the LR(0) parsing table from the DFA
- Use the LR(0) parsing table to determine whether to reduce or to shift

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

```
gk:  shift/reduce       reduce/reduce
    L → L, S.         L → S, L.
    S → S., L        L → S, L.
```

A Non-LR(0) Grammar

- Grammar for addition of numbers:
  - S → S + E | E
  - E → num
- Left-associative version is LR(0)
- Right-associative version is not LR(0)
  - S → E + S | E
  - E → num

LR(0) Parsing Table
**SLR Parsing**
- SLR Parsing = easy extension of LR(0)
  - For each reduction \( X \rightarrow \gamma \) look at the next symbol \( C \)
  - Apply reduction only if \( C \) is not in FOLLOW(\( X \))
- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions \( X \rightarrow \gamma \) only in the columns of symbols in FOLLOW(\( X \))
- Example:

<table>
<thead>
<tr>
<th>FOLLOW(S)</th>
<th>($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 num +</td>
<td>E S</td>
</tr>
<tr>
<td>2 S Э S</td>
<td>g2 g6</td>
</tr>
</tbody>
</table>

**SLR Parsing Table**
- Reductions do not fill entire rows
- Otherwise, same as LR(0)

<table>
<thead>
<tr>
<th>num +</th>
<th>E S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 g4</td>
<td>S Э S</td>
</tr>
<tr>
<td>2 g4</td>
<td>S Э S</td>
</tr>
<tr>
<td>3 S Э S</td>
<td></td>
</tr>
<tr>
<td>4 S Э S</td>
<td></td>
</tr>
<tr>
<td>5 S Э S</td>
<td></td>
</tr>
<tr>
<td>6 S Э S</td>
<td></td>
</tr>
<tr>
<td>7 accept</td>
<td></td>
</tr>
</tbody>
</table>

**LR(1) Parsing**
- Get as much power as possible out of 1 look-ahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead
- LR(1) parsing uses similar concepts as LR(0)
  - Parser states = sets of items
  - LR(1) item = LR(0) item + look-ahead symbol possibly following production

- LR(0) item: \( S \rightarrow S + E \)
- LR(1) item: \( S \rightarrow S + E + \)

**LR(1) States**
- LR(1) states = set of LR(1) items
- LR(1) item = \( (X \rightarrow \alpha \cdot \beta, \gamma) \)
- Meaning: \( \alpha \) already matched at top of the stack; next expect to see \( \beta \gamma \)
- Shorthand notation
  \( (X \rightarrow \alpha \cdot \beta, x_i) \)
  means:
  \( S \rightarrow S + E + \$, \ S \rightarrow S + E \ num \)
  ...
  \( (X \rightarrow \alpha \cdot \beta, x_i) \)
- Extend closure and goto operations

**LR(1) Closure**
- LR(1) closure operation:
  - Start with Closure(\( S \)) = \( S \)
  - For each item in \( S \):
    \( X \rightarrow \alpha \cdot Y \beta, z \)
    and for each production \( Y \rightarrow \gamma \) add the following item to the closure of \( S \):
    \( Y \rightarrow \gamma, \text{ FIRST}(\beta z) \)
  - Repeat until nothing changes
- Similar to LR(0) closure, but also keeps track of the look-ahead symbol

**LR(1) Start State**
- Initial state: start with \( (S' \rightarrow . S, \$) \), then apply the closure operation
- Example: sum grammar

| S' \rightarrow S \$ |
| S \rightarrow E + S | E |
| E \rightarrow num |

| S' \rightarrow . S  |
| S \rightarrow . E + S  |
| S \rightarrow . E  |
| E \rightarrow . num  | 

LR(1) Goto Operation
- LR(1) goto operation describes transitions between LR(1) states.
- Algorithm: for a state S and a symbol Y
  \[ S' = \{ (X \rightarrow \alpha Y \beta, z) \mid (X \rightarrow \alpha \beta, z) \in S \} \]
  \[ \text{Goto}(S, X) = \text{Closure}(S') \]

LR(1) DFA Construction
- If \( S' = \text{goto} (S, x) \) then add an edge labeled \( x \) from \( S \) to \( S' \):

LR(1) Reductions
- Reductions correspond to LR(1) items of the form \( (X \rightarrow \gamma, y) \):

LR(1) Parsing Table Construction
- Same as construction of LR(0) parsing table, except for reductions:
  - For a transition \( S \rightarrow S' \) on terminal \( x \):
    \[ \text{Shift}(S') \subseteq \text{Table}[S,x] \]
  - For a transition \( S \rightarrow S' \) on non-terminal \( N \):
    \[ \text{Goto}(S') \subseteq \text{Table}[S,N] \]
  - If \( (X \rightarrow \gamma, y) \in S_y \) then:
    \[ \text{Reduce}(X \rightarrow \gamma) \subseteq \text{Table}[S,y] \]

LR(1) Parsing Table Example

LALR(1) Grammars
- Problem with LR(1): too many states
- LALR(1) Parsing (Look-Ahead LR):
  - Constructs LR(1) DFA and then merge any two LR(1) states whose items are identical except look-ahead
  - Results in smaller parser tables
  - Theoretically less powerful than LR(1)
- LALR(1) Grammar = a grammar whose LALR(1) parsing table has no conflicts
**LL/LR Grammar Summary**

- **LL parsing tables**
  - Nonterminals \( \rightarrow \) terminals \( \rightarrow \) productions
  - Computed using FIRST/FOLLOW
- **LR parsing tables**
  - LR(0) states \( \rightarrow \) terminals \( \rightarrow \) (shift/reduce)
  - LR(k) states \( \rightarrow \) nonterminals \( \rightarrow \) grammar
  - Computed using closure/growth operations on LR states

- A grammar is:
  - LL(1) if its LL(1) parsing table has no conflicts
  - LR(0) if its LR(0) parsing table has no conflicts
  - SLR if its SLR parsing table has no conflicts
  - LALR(1) if its LALR(1) parsing table has no conflicts
  - LR(1) if its LR(1) parsing table has no conflicts

---

**Classification of Grammars**

LR(0) \( \subseteq \) SLR

LR(k) \( \subseteq \) LR(k+1)

LL(k) \( \subseteq \) LR(k)

LALR(1) \( \subseteq \) LR(1)

---

**Automate the Parsing Process**

- Can automate:
  - The construction of LR parsing tables
  - The construction of shift-reduce parsers based on these parsing tables
- **Automatic parser generators:** yacc, bison, CUP
- **LALR(1) parser generators**
  - No much difference compared to LR(1) in practice
  - Smaller parsing tables than LR(1)
  - Augment LALR(1) grammar specification with declarations of precedence, associativity
- **output:** LALR(1) parser program

---

**Shift/Reduce Conflict**

\[
E \rightarrow E + E
\]

\[
E \rightarrow \text{num}
\]

\[
\begin{align*}
\text{shift/reduce conflict} & \quad \text{shift: } 1+(2+3) \\
& \quad \text{reduce: } (1+2)+3
\end{align*}
\]

\[
E \rightarrow E + E + E + \text{num}
\]

**Associativity**

\[
S \rightarrow S + E \mid E
\]

\[
E \rightarrow E + E
\]

What happens if we run this grammar through LALR construction?

---

**Grammar in CUP**

\[
E ::= E \ PLUS \ E
\]

\[
\mid \ LPAREN \ E \ RPAREN
\]

\[
\mid \ NUMBER ;
\]
Precedence

- CUP can also handle operator precedence

\[
E \rightarrow E + E \mid T \\
T \rightarrow T \times T \mid \text{num} \mid (E)
\]

Conflicts without Precedence

\[
E \rightarrow E + E \mid E \times E \\
\mid \text{num} \mid (E)
\]

Precedence in CUP

precedence left PLUS; precedence left TIMES; // TIMES > PLUS

\[
E ::= E + E \mid E \times E \mid ...
\]

RULE: in conflict, choose \textit{reduce} if production symbol has higher precedence than shifted symbol; choose \textit{shift} if vice-versa

\[
\begin{align*}
E &\rightarrow E + E \quad \text{reduce} \\
E &\rightarrow E \times E \quad \text{reduce} \\
E &\rightarrow E + E \times E \\
E &\rightarrow E \times E + E
\end{align*}
\]

Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1) grammars
- Precedence, associativity declarations simplify grammar writing