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) Parsing Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td>g4</td>
<td>g5</td>
</tr>
<tr>
<td>2</td>
<td>S=id</td>
<td>S=id</td>
<td>S=id</td>
<td>S=id</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S=id(L)</td>
<td>S=id(L)</td>
<td>S=id(L)</td>
<td>S=id(L)</td>
</tr>
<tr>
<td>6</td>
<td>L=E</td>
<td>L=E</td>
<td>L=E</td>
<td>L=E</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L=id(L)</td>
<td>L=id(L)</td>
<td>L=id(L)</td>
<td>L=id(L)</td>
</tr>
<tr>
<td>9</td>
<td>L=id(S)</td>
<td>L=id(S)</td>
<td>L=id(S)</td>
<td>L=id(S)</td>
</tr>
</tbody>
</table>

A Non-LR(0) Grammar

- Grammar for addition of numbers:
  \[ S \rightarrow S + E | E \]
  \[ E \rightarrow \text{num} \]
- Left-associative version is LR(0)
- Right-associative version is not LR(0)
  \[ S \rightarrow E + S | E \]
  \[ E \rightarrow \text{num} \]
SLR Parsing

- SLR Parsing = easy extension of LR(0)
  - For each reduction \( X \rightarrow Y \) look at the next symbol \( C \)
  - Apply reduction only if \( C \) is in \( \text{FOLLOW}(X) \)

- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions \( X \rightarrow Y \) only in the columns of symbols in \( \text{FOLLOW}(X) \)

- Example: \( \text{FOLLOW}(S) = \{ , , \} \)

\[
\begin{array}{ccc}
1 & s4 & + \$ \\
2 & s3 & S \rightarrow E \\
\end{array}
\]

\[ g2 \ g6 \]

---

SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise, same as LR(0)

\[
\begin{array}{ccc}
1 & s4 & + \$ \\
2 & s3 & S \rightarrow E \\
3 & s4 & S \rightarrow E \\
4 & 5 & S \rightarrow E + S \\
6 & 7 & S \rightarrow E + S \\
7 & & \text{accept} \\
\end{array}
\]

---

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

\[
\begin{align*}
\text{LR(0) item:} & \quad S \rightarrow S + E \\
\text{LR(1) item:} & \quad S \rightarrow S + E +
\end{align*}
\]

---

LR(1) States

- LR(1) state = set of LR(1) items
- LR(1) item = \(( X \rightarrow a \cdot b , \gamma \) )
- Meaning: \( a \) already matched at top of the stack; next expect to see \( b \gamma \)
- Shorthand notation
  \[
  ( X \rightarrow a \cdot b , (x_0 , \ldots , x_n ) )
  \]
  means:
  \[
  S \rightarrow S . + E +,$$
  S \rightarrow S . + E \text{ num}
  \]
  \[
  ( X \rightarrow a \cdot b , x_i )
  \]
  \[
  ( X \rightarrow a \cdot b , x_i )
  \]
- Extend closure and goto operations

---

LR(1) Start State

- Initial state: start with \(( S' \rightarrow . S , , )\), then apply the closure operation
- Example: sum grammar

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow E + S | E \\
E & \rightarrow \text{num}
\end{align*}
\]

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow E + S \\
S & \rightarrow E \\
E & \rightarrow \text{num} +,$$
\]

---

LR(1) Closure

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

- **LR(1) goto operation** describes transitions between LR(1) states.

  **Algorithm:** for a state $S$ and a symbol $Y$
  
  $S' = \{ (X \to \alpha Y \beta, z) \mid (X \to \alpha Y \beta, z) \in S \}$

  $\text{Goto}(S, Y) = \text{Closure}(S')$


**LR(1) DFA Construction**

- If $S' = \text{goto} (S, x)$ then add an edge labeled $x$ from $S$ to $S'$

```
S' \rightarrow S. S
S \rightarrow . E + S $ $ S \rightarrow . E $ $ E \rightarrow . num +,$
```

**LR(1) Reductions**

- Reductions correspond to LR(1) items of the form $(X \to \gamma, Y)$

```
S' \rightarrow S. S
S \rightarrow . E + S $ S \rightarrow . E $ E \rightarrow . num +,$
```

**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 $(X \to \gamma, Y) \in S$, then:
    
    $\text{Reduce}(X \to \gamma) \subseteq \text{Table}[S, Y]$

**LR(1) Parsing Table Example**

```
1 S' \rightarrow S. S $ S \rightarrow . E + S $ S \rightarrow . E $ E \rightarrow . num +,$
2 S \rightarrow . E + S $ S \rightarrow . E $ E \rightarrow . num +,$
3 S \rightarrow . E + S $ S \rightarrow . E + S $ S \rightarrow . E $ E \rightarrow . num +,$
```

**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).

```
S \rightarrow \text{id. } + S \rightarrow \text{id. } S \rightarrow \text{E. } S \rightarrow \text{E. } + S \rightarrow \text{E. }
```

- **LALR(1) Grammar** = a grammar whose LALR(1) parsing table has no conflicts.
LL/LR Grammar Summary
- LL parsing tables
  - Nonterminals x terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables
  - LR states x terminals → (shift/reduce)
  - LR states x non-terminals → goto
  - Computed using closure/goto 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

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

Classification of Grammars
LR(0) ⊆ SLR
LR(1) ⊆ LALR(1)
LR(k) ⊆ LR(k+1)
LL(k) ⊆ LL(k+1)

Associativity
S → S + E | E
E → num

What happens if we run this grammar through LALR construction?

Shift/Reduce Conflict
E → E + E
E → num
E → E + E . +
E → E . + E ,+$

shift/reduce conflict
shift: 1+(2+3) reduce: (1+2)+3
1+2+3

“when shifting ‘+’ conflicts with reducing a production, choose reduce”
E ::= E PLUS E
| LPAREN E RPAREN
| NUMBER ;

Grammar in CUP
non terminal E; terminal PLUS, LPAREN...
precedence left PLUS;
Precedence

- CUP can also handle operator precedence

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

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

Conflicts without Precedence

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

Precedence in CUP

- 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

Summary