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
  - **ok** shift/reduce reduce / reduce

<table>
<thead>
<tr>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → S</td>
<td>L → S .</td>
</tr>
</tbody>
</table>

LR(0) Parsing Table

```
\begin{array}{cccc|cc}
1 & ) & id & $ & S & L \\
2 & s3 & s2 & & g4 & g5 \\
3 & S \rightarrow & S & \cdot & S & \cdot \\
4 & s3 & s2 & accept & & \\
5 & s6 & s8 & & & \\
6 & S \rightarrow & S & \cdot & S & \cdot \\
7 & L \rightarrow & L & \cdot & L & \cdot \\
8 & s3 & s2 & & & \\
9 & L \rightarrow & L & \cdot & L & \cdot \\
\end{array}
```

A Non-LR(0) Grammar

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

LR(0) Parsing Table

```
\begin{array}{cccc|cc}
1 & S' \rightarrow S \cdot $ & & & & \\
2 & S \rightarrow E + S & & & & \\
3 & S \rightarrow E \cdot S & & & & \\
4 & E \rightarrow \text{num} \cdot & & & & \\
5 & S' \rightarrow S \cdot $ & & & & \\
6 & E \rightarrow \text{num} & & & & \\
7 & S' \rightarrow S \cdot $ & & & & \\
8 & E \rightarrow \text{num} \cdot & & & & \\
9 & S \rightarrow E + S & & & & \\
10 & S \rightarrow E \cdot S & & & & \\
11 & E \rightarrow \text{num} \cdot & & & & \\
12 & S \rightarrow E + S & & & & \\
13 & S \rightarrow E \cdot S & & & & \\
14 & E \rightarrow \text{num} \cdot & & & & \\
15 & S \rightarrow E + S & & & & \\
16 & S \rightarrow E \cdot S & & & & \\
17 & E \rightarrow \text{num} \cdot & & & & \\
18 & S \rightarrow E + S & & & & \\
19 & S \rightarrow E \cdot S & & & & \\
20 & E \rightarrow \text{num} \cdot & & & & \\
\end{array}
```
SLR Parsing

- SLR Parsing = easy extension of LR(0)
  - For each reduction X → γ look at the next symbol C
  - Apply reduction only if C is in FOLLOW(X)

- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions X → γ only in the columns of symbols in FOLLOW(X)

- Example: FOLLOW(S) = {S, $}

<table>
<thead>
<tr>
<th>num</th>
<th>S</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s4</td>
<td>$</td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>S→E</td>
<td>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>S</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s4</td>
<td>$</td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>S→E</td>
<td>g6</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>S→E</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S→E</td>
<td>S→E</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S→E</td>
<td>S→E</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S→E+</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>accept</td>
<td></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

<table>
<thead>
<tr>
<th>LR(0) Item</th>
<th>LR(1) Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → S + E</td>
<td>S → . S + E +</td>
</tr>
</tbody>
</table>

LR(1) States

- LR(1) state = set of LR(1) items
- LR(1) item = \( X \rightarrow \alpha \cdot \beta \cdot \gamma \) 
- Meaning: \( \alpha \) already matched at top of the stack; next expect to see \( \beta \gamma \)

<table>
<thead>
<tr>
<th>Shorthand Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \rightarrow \alpha \cdot \beta \cdot \gamma )</td>
<td>( S \rightarrow S \cdot E + , $ )</td>
</tr>
<tr>
<td>( X \rightarrow \alpha \cdot \beta \cdot \gamma )</td>
<td>( S \rightarrow S \cdot E \cdot E ) num</td>
</tr>
<tr>
<td>( X \rightarrow \alpha \cdot \beta \cdot \gamma )</td>
<td>( S \rightarrow S \cdot E \cdot E \cdot num )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( S \rightarrow S \cdot E + S \cdot , $ )</td>
</tr>
</tbody>
</table>

| Extend closure and goto operations |

LR(1) Closure

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

- Similar to LR(0) closure, but also keeps track of the look-ahead symbol

<table>
<thead>
<tr>
<th>closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>S' → S $ S' → S $</td>
</tr>
<tr>
<td>S → E + S $ S → E $</td>
</tr>
<tr>
<td>E → num $</td>
</tr>
</tbody>
</table>
LR(1) Goto Operation

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

\[ S \rightarrow E \cdot S \]
\[ S \rightarrow E \cdot \]
\[ S \rightarrow E \cdot +,\$

LR(1) DFA Construction

- If \( S' \) = 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 \((X \rightarrow \gamma, \gamma)\)

\[ S' \rightarrow S \]
\[ S \rightarrow E \cdot \]
\[ S \rightarrow E \cdot +,\$

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 \((X \rightarrow \gamma, \gamma) \in S\), then:
  \[ \text{Reduce}(X \rightarrow \gamma) \subseteq \text{Table}[S, \gamma] \]

LR(1) Parsing Table Example

- 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)

\[ S \rightarrow \text{id.} + \]
\[ S \rightarrow \text{id.} + \]
\[ S \rightarrow \text{id.} + \]
\[ S \rightarrow \text{id.} + \]

- 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)
- LL(k) ⊆ LL(k+1)
- LR(k) ⊆ LR(k+1)

Shift/Reduce Conflict

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

\[ E \rightarrow E + E. + \]
\[ E \rightarrow E. + E \]

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

When shifting ‘+’ conflicts with reducing a production, choose reduce

Grammar in CUP

non terminal E;
terminal PLUS, LPAREN...
precedence left PLUS;

When shifting ‘+’ conflicts with reducing a production, choose reduce

\[ E ::= E \ PLUS E \]
\| LPAREN E RPAREN
\| NUMBER ;

Associativity

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

What happens if we run this grammar through LALR construction?
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

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

\[ E \rightarrow E + E \mid E \times E \]
\[ E \rightarrow E + E \mid E \times E \]

Precedence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
\[ E ::= E \text{PLUS} E \mid E \text{TIMES} E \mid \ldots \]

RULE: in conflict, choose reduce if production symbol
higher precedence than shifted symbol; choose shift if vice-versa

\[ E \rightarrow E + E \]
\[ E \rightarrow E \times E \]
reduce \( E \rightarrow E \times E \)

Shift \( \times \)

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