Syntax Analysis
Part II
Chapter 4

Bottom-Up Parsing

• LR methods (Left-to-right, Rightmost derivation)
  – SLR, Canonical LR, LALR
• Other special cases:
  – Shift-reduce parsing
  – Operator-precedence parsing

Operator-Precedence Parsing

• Special case of shift-reduce parsing
• We will not further discuss (you can skip textbook section 4.6)
Shift-Reduce Parsing

Grammar:

\[ S \rightarrow aABe \]
\[ A \rightarrow AbBe \]
\[ B \rightarrow d \]

These match production's right-hand sides

Reducing a sentence:

Shift-reduce corresponds to a rightmost derivation:

\[ S \Rightarrow_{rm} aABe \]
\[ \Rightarrow_{rm} aAde \]
\[ \Rightarrow_{rm} aAbcd e \]

Handles

A handle is a substring of grammar symbols in a right-sentential form that matches a right-hand side of a production

Grammar:

\[ S \rightarrow aABe \]
\[ A \rightarrow AbBe \]
\[ B \rightarrow d \]

\[ a b c d e \]

Handle

\[ a b c d e \]

\[ a A A e \]

\[ S \]

NOT a handle, because further reductions will fail

(result is not a sentential form)

Stack Implementation of Shift-Reduce Parsing

Grammar:

\[ E \rightarrow E + E \]
\[ E \rightarrow E * E \]
\[ E \rightarrow (E) \]
\[ E \rightarrow \text{id} \]

Find handles to reduce

Stack Implementation of Shift-Reduce Parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>+id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>*id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>*id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>shift reduce E \rightarrow id</td>
</tr>
</tbody>
</table>

How to resolve conflicts?
Conflicts

- Shift-reduce and reduce-reduce conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

Shift-Reduce Parsing: Shift-Reduce Conflicts

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ldots \text{if } E \text{ then } S \ldots$</td>
<td>$\ldots \text{else } S \ldots$</td>
<td>shift or reduce?</td>
</tr>
</tbody>
</table>

Ambiguous grammar:
- $S \rightarrow \text{if } E \text{ then } S$
- $\ldots \text{if } E \text{ then } S \ldots$
- $\text{other}$

Resolve in favor of shift, so else matches closest if

Shift-Reduce Parsing: Reduce-Reduce Conflicts

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$aB$</td>
<td>$a$</td>
<td>shift reduce $A \rightarrow a$ or $B \rightarrow a$?</td>
</tr>
</tbody>
</table>

Grammar:
- $C \rightarrow A B$
- $A \rightarrow a$
- $B \rightarrow a$

Resolve in favor of reduce $A \rightarrow a$, otherwise we’re stuck!
LR\(_{(k)}\) Parsers: Use a DFA for Shift/Reduce Decisions

Grammar:
\[
S \rightarrow C \\
C \rightarrow AB \\
A \rightarrow a \\
B \rightarrow a 
\]

State \(I_0\):
\[
S \rightarrow \ast \\
C \rightarrow \ast AB \\
A \rightarrow \ast a 
\]

Can only reduce \(A \rightarrow a\) (not \(B \rightarrow a\))

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ast)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(3)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(3)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(2)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(\ast)</td>
<td>(\ast)</td>
</tr>
</tbody>
</table>

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ast)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(3)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(3)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(2)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(B)</td>
<td>(a)</td>
<td>(\ast)</td>
</tr>
<tr>
<td>(a)</td>
<td>(\ast)</td>
<td>(\ast)</td>
</tr>
</tbody>
</table>
DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>start in state $0$</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>shift (and goto state 3)</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>reduce $A \to a$ (goto 2)</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>shift (goto 5)</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>reduce $B \to a$ (goto 4)</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>reduce $C \to AB$ (goto 1)</td>
</tr>
<tr>
<td>$0$</td>
<td>a$a$</td>
<td>accept ($S \to C$)</td>
</tr>
</tbody>
</table>

This is a DFA for Shift/Reduce Decisions. The states of the DFA are used to determine if a handle is on top of the stack.
DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>a</td>
<td>push and goto state 3</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>A</td>
<td>reduce A → a (goto 2)</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>a</td>
<td>shift (goto 5)</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>A</td>
<td>reduce B → a (goto 4)</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>C</td>
<td>reduce C → AB (goto 1)</td>
</tr>
</tbody>
</table>

Grammar:

\[
S \rightarrow C \\
C \rightarrow AB \\
A \rightarrow a \\
B \rightarrow a
\]

State $I_0$:

\[
S \rightarrow \cdot C \\
C \rightarrow \cdot A B \\
A \rightarrow \cdot a \\
\]

State $I_1$:

\[
S \rightarrow C \cdot \text{goto} (I_0, C) \\
\]

Model of an LR Parser

Configuration ( = LR parser state): 

\[
(s_0, X_1 s_1, X_2 s_2 \ldots X_m s_m, a_t, a_{t+1} \ldots a_n, S)
\]

stack

\[
\downarrow
\]

LR Parsing Program (driver)

\[
\downarrow
\]

input

\[
\downarrow
\]

output

action

\[
\downarrow
\]

goto

DFA

Built with LR(0) method, SLR method, LR(1) method, or LALR(1) method

LR Parsing (Driver)

\[
(s_0, X_1 s_1, X_2 s_2 \ldots X_m s_m, a_t, a_{t+1} \ldots a_n, S)
\]

If action[\(s_0,a_t]\] = shift \(s\) then 
push \(a_t\), push \(s\), and advance input:

\[
(s_0, X_1 s_1, X_2 s_2 \ldots X_m s_m, a_t, a_{t+1} \ldots a_n, S)
\]

If action[\(s_0,a_t]\] = reduce \(A \rightarrow \beta\) and goto[\(s_0,a_t\)] = \(\epsilon\) with \(\epsilon\) \# symbols, push \(A\), and push \(s\):

\[
(s_0, X_1 s_1, X_2 s_2 \ldots X_m s_m, a_t, a_{t+1} \ldots a_n, S)
\]

If action[\(s_0,a_t]\] = accept then stop

If action[\(s_0,a_t]\] = error then attempt recovery
Example LR Parse Table

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s_5$</td>
<td>1 2 3</td>
</tr>
<tr>
<td>1</td>
<td>$s_6$</td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>$s_7$</td>
</tr>
<tr>
<td>3</td>
<td>$s_4$</td>
<td>$s_4$</td>
</tr>
<tr>
<td>4</td>
<td>$s_5$</td>
<td>$s_6$</td>
</tr>
<tr>
<td>5</td>
<td>$s_3$</td>
<td>$s_4$</td>
</tr>
<tr>
<td>6</td>
<td>$s_5$</td>
<td>$s_6$</td>
</tr>
<tr>
<td>7</td>
<td>$s_3$</td>
<td>$s_4$</td>
</tr>
<tr>
<td>8</td>
<td>$s_6$</td>
<td>$s_{11}$</td>
</tr>
<tr>
<td>9</td>
<td>$s_7$</td>
<td>$s_7$</td>
</tr>
<tr>
<td>10</td>
<td>$s_3$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>11</td>
<td>$s_5$</td>
<td>$s_5$</td>
</tr>
</tbody>
</table>

Example LR Parsing

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$id$</td>
<td>shift 3</td>
</tr>
<tr>
<td>$S$</td>
<td>$id$</td>
<td>shift 3</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>reduce 4 goto 2</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>shift 7</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>shift 5</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>reduce 6 goto 10</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>reduce 1 goto 1</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>shift 6</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
<tr>
<td>$S$</td>
<td>$E + E$</td>
<td>id</td>
</tr>
</tbody>
</table>

SLR Grammars

- SLR (Simple LR): a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions $A \rightarrow \alpha$ on symbols in FOLLOW(A)
SLR Parsing Table

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

\[ \begin{array}{c|cc} \text{SLR Parsing Table} \\ \hline \text{r2} & s2 & r3 \\ \hline 1. S & E \\ 2. E & \rightarrow \text{id} + E \\ 3. E & \rightarrow \text{id} \\ \hline \end{array} \]

\[ \text{FOLLOW(E)} = \{ \$ \} \text{ thus reduce on } \$

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - Closure operation to construct LR(0) items
  - Goto operation to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

Constructing SLR Parsing Tables

1. Augment the grammar with \( S' \rightarrow S \)
2. Construct the set \( C = \{ I_0, I_1, \ldots, I_n \} \) of LR(0) items
3. If \( [A \rightarrow \alpha \beta] \in I_i \) and goto \((I_i, \alpha) = I_j\) then set \( \text{action}[i, \alpha] = \text{shift } j \)
4. If \( [A \rightarrow \alpha] \in I_i \) then set \( \text{action}[i, \alpha] = \text{reduce } A \rightarrow \alpha \)
   for all \( \alpha \in \text{FOLLOW}(A) \) (apply only if \( A \neq S' \))
5. If \( [S' \rightarrow S\bullet] \) is in \( I_i \) then set \( \text{action}[i, \$] = \text{accept} \)
6. If goto \((I_i, A) = I_j\) then set goto \([i, A] = j\)
7. Repeat 3-6 until no more entries added
8. The initial state \( i \) is the \( I_i \) holding item \( [S' \rightarrow S] \)
LR(0) Items of a Grammar

• An LR(0) item of a grammar G is a production of G with a • at some position of the right-hand side

• Thus, a production
  \[ A \rightarrow XYZ \]
  has four items:
  \[ \left[ A \rightarrow \cdot XYZ \right] \]
  \[ \left[ A \rightarrow X \cdot YZ \right] \]
  \[ \left[ A \rightarrow XY \cdot Z \right] \]
  \[ \left[ A \rightarrow XYZ \cdot \right] \]

• Note that production \( A \rightarrow \varepsilon \) has one item \( [A \rightarrow \cdot] \)

Constructing the set of LR(0) Items of a Grammar

1. The grammar is augmented with a new start symbol \( S' \) and production \( S' \rightarrow S \)

2. Initially, set \( C = \text{closure}([S' \rightarrow S]) \) (this is the start state of the DFA)

3. For each set of items \( I \subseteq C \) and each grammar symbol \( X \in (N \cup T) \) such that \( \text{goto}(I, X) \notin C \) and \( \text{goto}(I, X) \neq \emptyset \), add the set of items \( \text{goto}(I, X) \) to \( C \)

4. Repeat 3 until no more sets can be added

The Closure Operation for LR(0) Items

1. Start with \( \text{closure}(I) = I \)

2. If \( [A \rightarrow \alpha \bullet B] \in \text{closure}(I) \) then for each production \( B \rightarrow \gamma \) in the grammar, add the item \( [B \rightarrow \gamma \bullet] \) to \( I \) if not already in \( I \)

3. Repeat 2 until no new items can be added
The Closure Operation
(Example)

closure([E → • E]) =
(E → • E)  (E → • E)  (E → • E)  (E → • E)
[E → • E + T]  [E → • E + T]  [E → • E + T]  [E → • E + T]

Grammar:
E → E + T | T
T → T * F | F
F → ( E ) | id

The Goto Operation for LR(0)

Items
1. For each item [A → αXβ] ∈ I, add the set of items closure([A → αXβ]) to goto(I,X) if not already there
2. Repeat step 1 until no more items can be added to goto(I,X)
3. Intuitively, goto(I,X) is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

The Goto Operation (Example 1)

Suppose I = { [E → • E]  [E → • E + T]  [E → • T]  [T → • T * F]  [T → • T]  [F → • ( E )]  [F → • id] }

Then goto(I,E)
= closure({[E → E * E → E * T]})
= { [E → E * E]  [E → E * T] }
The Goto Operation (Example 2)

Suppose \( I = \{ [E \to E \cdot], [E \to E \cdot + T] \} \)

Then \( \text{goto}(I, \cdot) = \text{closure}([E \to E \cdot + T]) = \{ [E \to E \cdot + T], [T \to \cdot T F], [T \to \cdot F], [F \to \cdot (E)], [F \to \cdot \text{id}] \} \)

Grammar:

\[
\begin{align*}
E & \to E + T | T \\
T & \to T * F | F \\
F & \to (E) \\
F & \to \text{id}
\end{align*}
\]

Example SLR Grammar and LR(0) Items

Augmented grammar:

\[
I_0 = \text{closure}([C \to \cdot C])
\]

1. \( C' \to C \)
2. \( C \to A B \)
3. \( A \to a \)
4. \( B \to a \)

State \( I_0 \):

\[
\begin{align*}
C' & \to \cdot C \\
C & \to \cdot A B \\
A & \to \cdot a \\
B & \to \cdot a
\end{align*}
\]

Example SLR Parsing Table

\[
\begin{align*}
\text{Grammar:} & \\
1. & C' \to C \\
2. & C \to A B \\
3. & A \to a \\
4. & B \to a
\end{align*}
\]
SLR and Ambiguity

- Every SLR grammar is unambiguous, but not every unambiguous grammar is SLR
- Consider for example the unambiguous grammar
  \[ S \rightarrow L \equiv R \mid R \]
  \[ L \rightarrow * R \mid \text{id} \]
  \[ R \rightarrow L \]

\[ I_0: S' \rightarrow \]
\[ I_1: S \rightarrow L \equiv R \mid R \]
\[ I_2: S \rightarrow L \equiv R \mid R \]
\[ I_3: L \rightarrow * R \]
\[ I_4: L \rightarrow \text{id} \]
\[ I_5: R \rightarrow L \equiv R \mid R \]
\[ I_6: S \rightarrow \text{id} \]
\[ I_7: L \rightarrow R \mid R \]
\[ I_8: R \rightarrow L \]
\[ I_9: S \rightarrow L \equiv R \mid R \]

Action
\[ [2, \pi] = s_6 \]
\[ [2, \pi] = r_5 \]

LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

LR(0) item: \[ [A \rightarrow \alpha \cdot \beta] \]
LR(1) item: \[ [A \rightarrow \alpha \cdot \beta, a] \]

SLR Versus LR(1)

- Split the SLR states by adding LR(1) lookahead
- Unambiguous grammar
  1. \[ S \rightarrow L \equiv R \]
  2. \[ L \rightarrow R \]
  3. \[ L \rightarrow * R \]
  4. \[ L \rightarrow \text{id} \]
  5. \[ R \rightarrow L \]

Should not reduce on \( \pi \), because no right-sentential form begins with \( R \equiv \)
LR(1) Items

- An LR(1) item 
  \[ A \rightarrow \alpha \beta, a \]
  contains a lookahead terminal \( a \), meaning \( \alpha \) already on top of the stack, expect to see \( \beta \).
- For items of the form 
  \[ A \rightarrow \alpha \beta, a \]
  the lookahead \( a \) is used to reduce \( A \rightarrow \alpha \) only if the next input is \( a \).
- For items of the form 
  \[ A \rightarrow \alpha \beta, a \]
  with \( \beta \neq \epsilon \), the lookahead has no effect.

The Closure Operation for LR(1) Items

1. Start with \( \text{closure}(I) = I \)
2. If \( [A \rightarrow \alpha \beta, a] \in \text{closure}(I) \) then for each production \( B \rightarrow \gamma \) in the grammar and each terminal \( b \in \text{FIRST}(\beta) \), add the item 
   \[ [B \rightarrow \gamma, b] \] to \( I \) if not already in \( I \).
3. Repeat step 2 until no new items can be added.

The Goto Operation for LR(1) Items

1. For each item \( [A \rightarrow \alpha \beta, a] \in I \), add the set of items \( \text{closure}([A \rightarrow \alpha \beta, a]) \) to \( \text{goto}(I, X) \) if not already there.
2. Repeat step 1 until no more items can be added to \( \text{goto}(I, X) \).
Constructing the set of LR(1) Items of a Grammar

1. Augment the grammar with a new start symbol \( S' \) and production \( S' \rightarrow S \)
2. Initially, set \( C = \text{closure}([[S' \rightarrow S, \$]]) \)
   (this is the start state of the DFA)
3. For each set of items \( I \in C \) and each grammar symbol \( X \in (N \cup T) \) such that \( \text{goto}(LX) \notin C \) and \( \text{goto}(LX) \neq \emptyset \), add the set of items \( \text{goto}(LX) \) to \( C \)
4. Repeat 3 until no more sets can be added to \( C \)

Example Grammar and LR(1) Items

- Unambiguous LR(1) grammar:
  \[
  S \rightarrow L = R \\
  | R \\
  L \rightarrow * R \\
  | \text{id} \\
  R \rightarrow L \\
  \]
- Augment with \( S' \rightarrow S \)
- LR(1) items (next slide)
Constructing Canonical LR(1)
Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C=\{I_0, I_1, \ldots, I_n\}$ of LR(1) items
3. If $[A \rightarrow \alpha \cdot b] \in I_i$ and $\text{goto}(I_i, a) = I_j$ then set $\text{action}(i, a) = \text{shift}$ $j$
4. If $[A \rightarrow \alpha \cdot a] \in I_i$ then set $\text{action}(i, a) = \text{reduce} A \rightarrow \alpha$ (apply only if $A \neq S'$)
5. If $[S' \rightarrow \cdot S, \cdot] \in I_i$ then set $\text{action}(i, \cdot) = \text{accept}$
6. If $\text{goto}(I_i, A) = I_j$ then set $\text{goto}(i, A) = j$
7. Repeat 3-6 until no more entries added
8. The initial state $i$ is the $I_i$ holding item $[S' \rightarrow \cdot S, \cdot]$

Example LR(1) Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>s5</th>
<th>s4</th>
<th>s3</th>
<th>s2</th>
<th>s1</th>
<th>s0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></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></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>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar:
1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow \text{id}$
6. $R \rightarrow L$

LALR(1) Grammars

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages
Constructing LALR(1) Parsing Tables

1. Construct sets of LR(1) items
2. Combine LR(1) sets with sets of items that share the same first part

Example LALR(1) Grammar

- Unambiguous LR(1) grammar:
  
  \[ S \to L = R \]
  
  \[ L \to * R \]
  
  \[ \text{id} \]
  
  \[ R \to L \]

- Augment with \( S' \to S \)
- LALR(1) items (next slide)
Example LALR(1) Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>$</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>s5</td>
<td>a5</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>r6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>r3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td>r5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>s5</td>
<td>s4</td>
<td>s5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
<td>r4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
<td>r4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td>r2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>r6</td>
<td>r6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals × terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states × terminals → shift/reduce actions
  - LR states × nonterminals → goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

LL, SLR, LR, LALR Grammars
Dealing with Ambiguous Grammars

1. $S \rightarrow E$
2. $E \rightarrow E + E$
3. $E \rightarrow id$

Shift/reduce conflict:
- action[4,+] = shift 4
- action[4,+] = reduce $E \rightarrow E + E$

Using Associativity and Precedence to Resolve Conflicts

- Left-associative operators: reduce
- Right-associative operators: shift
- Operator of higher precedence on stack: reduce
- Operator of lower precedence on stack: shift

Error Detection in LR Parsing

- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol
Error Recovery in LR Parsing

- Panic mode
  - Pop until state with a goto on a nonterminal A is found, (where A represents a major programming construct), push A
  - Discard input symbols until one is found in the FOLLOW set of A
- Phrase-level recovery
  - Implement error routines for every error entry in table
- Error productions
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue