CS 335: Bottom-up Parsing

Swarnendu Biswas

Semester 2019-2020-II
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.
Rightmost Derivation of \( abbcde \)

\[
S \rightarrow aABe \\
A \rightarrow Abc \mid b \\
B \rightarrow d
\]

\[
\text{Input string: } abbcde
\]

\[
S \rightarrow aABe \\
\rightarrow aAde \\
\rightarrow aAbcde \\
\rightarrow abbcde
\]
Bottom-up Parsing

Constructs the parse tree starting from the leaves and working up toward the root

Input string: $abbcde$

<table>
<thead>
<tr>
<th></th>
<th>$abbcde$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aABe$</td>
<td>$abbcde$</td>
</tr>
<tr>
<td>$\rightarrow aAde$</td>
<td>$\rightarrow aAde$</td>
</tr>
<tr>
<td>$\rightarrow aAbcde$</td>
<td>$\rightarrow aAbcde$</td>
</tr>
<tr>
<td>$\rightarrow abbcde$</td>
<td>$\rightarrow abbcde$</td>
</tr>
<tr>
<td>$\rightarrow S$</td>
<td>$\rightarrow S$</td>
</tr>
</tbody>
</table>
Bottom-up Parsing

\[ S \rightarrow aABe \]
\[ A \rightarrow Abc \mid b \]
\[ B \rightarrow d \]

**Input string:** \textit{abbcde}

- \textit{abbcde}  
- \textit{aAbcde}  
- \textit{aAde}  
- \textit{aABe}  
- \textit{S}
Reduction

• Bottom-up parsing reduces a string $w$ to the start symbol $S$
  • At each reduction step, a chosen substring that is the rhs (or body) of a production is replaced by the lhs (or head) nonterminal

\[
S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w
\]

Derivation

Bottom-up Parser
Handle

- Handle is a substring that matches the body of a production
  - Reducing the handle is one step in the reverse of the rightmost derivation

<table>
<thead>
<tr>
<th>Right Sentential Form</th>
<th>Handle</th>
<th>Reducing Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E + T \mid T$</td>
<td>$\text{id}_1 \ast \text{id}_2$</td>
<td>$\text{id}_1$</td>
</tr>
<tr>
<td>$T \rightarrow T \ast F \mid F$</td>
<td>$F \ast \text{id}_2$</td>
<td>$F$</td>
</tr>
<tr>
<td>$F \rightarrow (E) \mid \text{id}$</td>
<td>$T \ast \text{id}_2$</td>
<td>$\text{id}_2$</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>$T \ast F$</td>
<td>$T \ast F$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>
Handle

Although $T$ is the body of the production $E \rightarrow T$, $T$ is not a handle in the sentential form $T \ast \text{id}_2$

<table>
<thead>
<tr>
<th>Right Sentential Form</th>
<th>Handle</th>
<th>Reducing Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E + T \mid T$</td>
<td>$\text{id}_1 \ast \text{id}_2$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$T \rightarrow T \ast F \mid F$</td>
<td>$F \ast \text{id}_2$</td>
<td>$T \rightarrow F$</td>
</tr>
<tr>
<td>$F \rightarrow (E) \mid \text{id}$</td>
<td>$T \ast \text{id}_2$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$T \ast F$</td>
<td>$T \ast F$</td>
<td>$T \rightarrow T \ast F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$E \rightarrow T$</td>
</tr>
</tbody>
</table>
Handle

• If $S \Rightarrow^{*} \alpha Aw \Rightarrow \alpha \beta w$, then $A \rightarrow \beta$ is a handle of $\alpha \beta w$

• String $w$ right of a handle must contain only terminals

A handle $A \rightarrow \beta$ in the parse tree for $\alpha \beta w$
Handle

If grammar $G$ is unambiguous, then every right sentential form has only one handle.

If $G$ is ambiguous, then there can be more than one rightmost derivation of $\alpha\beta w$. 
Shift-Reduce Parsing
Shift-Reduce Parsing

- Type of bottom-up parsing with two primary actions, shift and reduce
  - Other obvious actions are accept and error

- The input string (i.e., being parsed) consists of two parts
  - Left part is a string of terminals and nonterminals, and is stored in stack
  - Right part is a string of terminals read from an input buffer
  - Bottom of the stack and end of input are represented by $
Shift-Reduce Actions

• **Shift**: shift the next input symbol from the right string onto the top of the stack

• **Reduce**: identify a string on top of the stack that is the body of a production, and replace the body with the head
Shift-Reduce Parsing

• Initial

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$w$</td>
</tr>
</tbody>
</table>

• Final goal

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S$</td>
</tr>
</tbody>
</table>
Shift-Reduce Parsing

$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow (E) | id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id_1 * id_2$</td>
<td>Shift</td>
</tr>
<tr>
<td>$id_1$</td>
<td>* id_2$</td>
<td>Reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$F$</td>
<td>* id_2$</td>
<td>Reduce by $T \rightarrow F$</td>
</tr>
<tr>
<td>$T$</td>
<td>* id_2$</td>
<td>Shift</td>
</tr>
<tr>
<td>$T *$</td>
<td>id_2$</td>
<td>Shift</td>
</tr>
<tr>
<td>$T * id_2$</td>
<td>$</td>
<td>Reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$T * F$</td>
<td>$</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$</td>
<td>Reduce by $E \rightarrow T$</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Handle on Top of the Stack

- Is the following scenario possible?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$αβγ$</td>
<td>$w$</td>
<td>Reduce by $A → γ$</td>
</tr>
<tr>
<td>$αβA$</td>
<td>$w$</td>
<td>Reduce by $B → β$</td>
</tr>
<tr>
<td>$αBA$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Possible Choices in Rightmost Derivation

1. \( S \Rightarrow_{rm} \alpha Az \Rightarrow_{rm} \alpha \beta Byz \Rightarrow_{rm} \alpha \beta \gamma yz \)

2. \( S \Rightarrow_{rm} \alpha BxAz \Rightarrow_{rm} \alpha Bxyz \Rightarrow_{rm} \alpha \gamma xyz \)
Handle on Top of the Stack

- Is the following scenario possible?

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Handle always eventually appears on top of the stack, never inside
Shift-Reduce Actions

• **Shift**: shift the next input symbol from the right string onto the top of the stack

• **Reduce**: identify a string on top of the stack that is the body of a production, and replace the body with the head

How do you decide when to shift and when to reduce?
Steps in Shift-Reduce Parsers

General shift-reduce technique
If there is no handle on the stack, then shift
If there is a handle on the stack, then reduce

• Bottom up parsing is essentially the process of detecting handles and reducing them
• Different bottom-up parsers differ in the way they detect handles
Challenges in Bottom-up Parsing

Which action do you pick when there is a choice?

- Both shift and reduce are valid, implies a shift-reduce conflict

Which rule to use if reduction is possible by more than one rule?

- Reduce-reduce conflict
Shift-Reduce Conflict

$E \rightarrow E + E \mid E \ast E \mid id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id$</td>
<td>Shift</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E + E$</td>
<td>* id$</td>
<td>Reduce by $E \rightarrow E + E$</td>
</tr>
<tr>
<td>$E$</td>
<td>* id$</td>
<td>Shift</td>
</tr>
<tr>
<td>$E \ast$</td>
<td>id$</td>
<td>Shift</td>
</tr>
<tr>
<td>$E \ast id$</td>
<td>$</td>
<td>Reduce by $E \rightarrow id$</td>
</tr>
<tr>
<td>$E \ast E$</td>
<td>$</td>
<td>Reduce by $E \rightarrow E \ast E$</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

$E$ → $E + E$ | $E \ast E$ | $id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id$</td>
<td>Shift</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E + E$</td>
<td>* id$</td>
<td>Shift</td>
</tr>
<tr>
<td>$E + E \ast$</td>
<td>id$</td>
<td>Shift</td>
</tr>
<tr>
<td>$E + E \ast id$</td>
<td>$</td>
<td>Reduce by $E \rightarrow id$</td>
</tr>
<tr>
<td>$E + E \ast E$</td>
<td>$</td>
<td>Reduce by $E \rightarrow E \ast E$</td>
</tr>
<tr>
<td>$E + E$</td>
<td>$</td>
<td>Reduce by $E \rightarrow E + E$</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Shift-Reduce Conflict

\[ Stmt \rightarrow \text{if Expr then Stmt} \]
\[ \quad \text{if Expr then Stmt else Stmt} \]
\[ \quad \text{other} \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>... if Expr then Stmt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else ... $</td>
<td></td>
</tr>
</tbody>
</table>
Shift-Reduce Conflict

**Grammar:**

\[
Stmt \rightarrow \text{if } Expr \text{ then } Stmt \\
| \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\
| \text{other}
\]

**Table:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>... if Expr then Stmt</td>
<td>else ... $</td>
<td></td>
</tr>
</tbody>
</table>

What is a correct thing to do for this grammar – shift or reduce?
### Reduce-Reduce Conflict

#### $c + c$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$c + c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$c$</td>
<td>$+c$</td>
<td>Reduce by $R \rightarrow c$</td>
</tr>
<tr>
<td>$R$</td>
<td>$+c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$R +$</td>
<td>$c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$R + c$</td>
<td>$\cdot$</td>
<td>Reduce by $R \rightarrow c$</td>
</tr>
<tr>
<td>$R + R$</td>
<td>$\cdot$</td>
<td>Reduce by $R \rightarrow R + R$</td>
</tr>
<tr>
<td>$M$</td>
<td>$\cdot$</td>
<td></td>
</tr>
</tbody>
</table>

#### $c + c$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$c + c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$c$</td>
<td>$+c$</td>
<td>Reduce by $R \rightarrow c$</td>
</tr>
<tr>
<td>$R$</td>
<td>$+c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$R +$</td>
<td>$c$</td>
<td>Shift</td>
</tr>
<tr>
<td>$R + c$</td>
<td>$\cdot$</td>
<td>Reduce by $M \rightarrow R + c$</td>
</tr>
<tr>
<td>$M$</td>
<td>$\cdot$</td>
<td></td>
</tr>
</tbody>
</table>
LR Parsing
LR(k) Parsing

• Popular bottom-up parsing scheme
  • L is for left-to-right scan of input
  • R is for reverse of rightmost derivation
  • k is the number of lookahead symbols

• LR parsers are table-driven, like the nonrecursive LL parser

• LR grammar is one for which we can construct an LR parsing table
Popularity of LR Parsing

- Can recognize all language constructs with CFGs
- Most general nonbacktracking shift-reduce parsing method
- Works for a superset of grammars parsed with predictive or LL parsers

Why?
Popularity of LR Parsing

Can recognize all language constructs with CFGs

Most general nonbacktracking shift-reduce parsing method

Works for a superset of grammars parsed with predictive or LL parsers

- LL(k) parsing predicts which production to use having seen only the first k tokens of the right-hand side
- LR(k) parsing can decide after it has seen input tokens corresponding to the entire right-hand side of the production
Block Diagram of LR Parser

LR Parsing Program

Parse Table

Input

Output

Stack

$ a_i \ldots \ldots$
LR Parsing

• Remember the basic question: when to shift and when to reduce!
• Information is encoded in a DFA constructed using canonical LR(0) collection
  I. Augmented grammar $G'$ with new start symbol $S'$ and rule $S' \rightarrow S$
  II. Define helper functions Closure() and Goto()
LR(0) Item

• An LR(0) item (also called item) of a grammar $G$ is a production of $G$ with a dot at some position in the body.

• An item indicates how much of a production we have seen
  • Symbols on the left of “•” are already on the stack
  • Symbols on the right of “•” are expected in the input

<table>
<thead>
<tr>
<th>Production</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow XYZ$</td>
<td>$A \rightarrow \bullet XYZ$</td>
</tr>
<tr>
<td>$A \rightarrow X\bullet YZ$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow XY\bullet Z$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow XYZ\bullet$</td>
<td></td>
</tr>
</tbody>
</table>

$A \rightarrow \bullet XYZ$ indicates that we expect a string derivable from $XYZ$ next on the input.
Closure Operation

• Let $I$ be a set of items for a grammar $G$
• Closure($I$) is constructed by
  1. Add every item in $I$ to Closure($I$)
  2. If $A \rightarrow \alpha \cdot B \beta$ is in Closure($I$) and $B \rightarrow \gamma$ is a rule, then add $B \rightarrow \gamma$ to Closure($I$) if not already added
  3. Repeat until no more new items can be added to Closure($I$)
Example of Closure

Suppose $I = \{ E' \rightarrow \cdot E \}$, compute Closure($I$)

\begin{align*}
E' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid id
\end{align*}
Example of Closure

Suppose $I = \{E' \rightarrow \mathbf{E}\}$

Closure$(I) = \{E' \rightarrow \mathbf{E},$

$E \rightarrow \mathbf{E} + T ,

E \rightarrow \mathbf{T},

T \rightarrow \mathbf{T} * F ,

T \rightarrow \mathbf{F},

F \rightarrow \mathbf{(E)},

F \rightarrow \mathbf{id}\}$
Kernel and Nonkernel Items

• If one $B$-production is added to $\text{Closure}(I)$ with the dot at the left end, then all $B$-productions will be added to the closure

• Kernel items
  • Initial item $S' \rightarrow \bullet S$, and all items whose dots are not at the left end

• Nonkernel items
  • All items with their dots at the left end, except for $S' \rightarrow \bullet S$
Goto Operation

• Suppose $I$ is a set of items and $X$ is a grammar symbol

• $\text{Goto}(I, X)$ is the closure of set all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X \beta]$ is in $I$
  • If $I$ is a set of items for some valid prefix $\alpha$, then $\text{Goto}(I, X)$ is set of valid items for prefix $\alpha X$

• Intuitively, $\text{Goto}(I, X)$ defines the transitions in the LR(0) automaton
  • $\text{Goto}(I, X)$ gives the transition from state $I$ under input $X$
Example of Goto

\[
\begin{align*}
  E' & \rightarrow E \\
  E & \rightarrow E + T \mid T \\
  T & \rightarrow T * F \mid F \\
  F & \rightarrow (E) \mid \text{id}
\end{align*}
\]

Suppose \( I = \{ \)
\[
\begin{align*}
  E' & \rightarrow E\star, \\
  E & \rightarrow E\star + T
\end{align*}
\]
\( \}

• Compute Goto(\( I, + \)
Example of Goto

\[ E' \rightarrow E \]
\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \ast F \mid F \]
\[ F \rightarrow (E) \mid \text{id} \]

Goto(I, +) = \{ 
\[ E \rightarrow E + \bullet T, \]
\[ T \rightarrow \bullet T \ast F, \]
\[ T \rightarrow \bullet F, \]
\[ F \rightarrow \bullet (E'), \]
\[ F \rightarrow \bullet \text{id} \]
\}
Canonical Collection of Sets of LR(0) Items

\[ C = \text{Closure} \{ S' \rightarrow \bullet S \} \]

repeat
   for each set of items \( I \) in \( C \)
      for each grammar symbol \( X \)
         if Goto(\( I, X \)) is not empty and not in \( C \)
            add Goto(\( I, X \)) to \( C \)
   until no new sets of items are added to \( C \)
Canonical Collection of Sets of LR(0) Items

- Compute the canonical collection for the expression grammar

\[
\begin{align*}
E' & \rightarrow E \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
\]
Canonical Collection of Sets of LR(0) Items

\[ I_0 = \text{Closure}(E' \to \bullet E) = \{ \]
\[ E' \to \bullet E, \]
\[ E \to \bullet E + T, \]
\[ E \to \bullet T, \]
\[ T \to \bullet T \ast F, \]
\[ T \to \bullet F, \]
\[ F \to \bullet (E), \]
\[ F \to \bullet \text{id}, \]
\[ \} \]

\[ I_1 = \text{Goto}(I_0, E) = \{ \]
\[ E' \to E \bullet, \]
\[ E \to E \bullet + T \]
\[ \} \]

\[ I_2 = \text{Goto}(I_0, T) = \{ \]
\[ E \to T \bullet, \]
\[ T \to T \bullet \ast F \]
\[ \} \]

\[ I_3 = \text{Goto}(I_0, F) = \{ \]
\[ T \to F \bullet \]
\[ \} \]

\[ I_4 = \text{Goto}(I_0, "\text{"}) = \{ \]
\[ F \to (\bullet E), \]
\[ E \to \bullet E + T, \]
\[ E \to \bullet T, \]
\[ T \to \bullet T \ast F, \]
\[ T \to \bullet F, \]
\[ F \to \bullet (E), \]
\[ F \to \bullet \text{id}, \]
\[ \} \]

\[ I_5 = \text{Goto}(I_0, \text{id}) = \{ \]
\[ F \to \text{id} \bullet \]
\[ \} \]

\[ I_6 = \text{Goto}(I_2, \text{*}) = \{ \]
\[ T \to T \ast \bullet F, \]
\[ F \to \bullet (E), \]
\[ F \to \bullet \text{id} \]
\[ \} \]
Canonical Collection of Sets of LR(0) Items

\[ I_6 = \text{Goto}(I_1, +) = \{ \]
\[ E \to E + \cdot T, \]
\[ T \to \cdot T \ast F, \]
\[ T \to \cdot F, \]
\[ F \to \cdot (E), \]
\[ F \to \cdot \text{id}, \]
\[ \} \]

\[ I_8 = \text{Goto}(I_4, E) = \{ \]
\[ E \to E \cdot + T, \]
\[ F \to (E \cdot) \]
\[ \} \]

\[ I_9 = \text{Goto}(I_6, T) = \{ \]
\[ E \to E + T \cdot, \]
\[ T \to T \cdot \ast F \]
\[ \} \]

\[ I_{10} = \text{Goto}(I_7, F) = \{ \]
\[ T \to T \ast F \cdot, \]
\[ \} \]

\[ I_{11} = \text{Goto}(I_8, ")") = \{ \]
\[ F \to (E) \cdot \]
\[ \} \]

\[ I_2 = \text{Goto}(I_4, T) \]

\[ I_3 = \text{Goto}(I_4, F) \]

\[ I_4 = \text{Goto}(I_4, "") \]

\[ I_5 = \text{Goto}(I_4, \text{id}) \]

\[ I_6 = \text{Goto}(I_8, +) \]

\[ I_7 = \text{Goto}(I_9, \ast) \]
LR(0) Automaton

• An LR parser makes shift-reduce decisions by maintaining states
• Canonical LR(0) collection is used for constructing a DFA for parsing
• States represent sets of LR(0) items in the canonical LR(0) collection
  • Start state is Closure({$S' \rightarrow \cdot S$}), where $S'$ is the start symbol of the augmented grammar
  • State $j$ refers to the state corresponding to the set of items $I_j$
LR(0) Automaton
Use of LR(0) Automaton

• How can LR(0) automata help with shift-reduce decisions?
• Suppose string $\gamma$ of grammar symbols takes the automaton from start state $S_0$ to state $S_j$
  • Shift on next input symbol $a$ if $S_j$ has a transition on $a$
  • Otherwise, reduce
    • Items in state $S_j$ help decide which production to use
## Shift-Reduce Parser with LR(0) Automaton

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>id * id$</td>
<td>Shift to 5</td>
</tr>
<tr>
<td>0 5</td>
<td>$id</td>
<td>* id$</td>
<td>Reduce by $F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>0 3</td>
<td>$F</td>
<td>* id$</td>
<td>Reduce by $T \rightarrow F$</td>
</tr>
<tr>
<td>0 2</td>
<td>$T</td>
<td>* id$</td>
<td>Shift to 7</td>
</tr>
<tr>
<td>0 2 7</td>
<td>$T *$</td>
<td>id$</td>
<td>Shift to 5</td>
</tr>
<tr>
<td>0 2 7 5</td>
<td>$T * \text{id}$</td>
<td>$\text{id}$</td>
<td>Reduce by $F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>0 2 7 10</td>
<td>$T * F$</td>
<td>$\text$id$</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>0 2 10</td>
<td>$T$</td>
<td>$\text{id}$</td>
<td>Reduce by $E \rightarrow T$</td>
</tr>
<tr>
<td>0 1</td>
<td>$E$</td>
<td>$\text{id}$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Viable Prefix

- A viable prefix is a prefix of a right sentential form that can appear on the stack of a shift-reduce parser
  - $\alpha$ is a viable prefix if $\exists w$ such that $\alpha w$ is a right sentential form

$$E \rightarrow T \rightarrow T * F \rightarrow T * \text{id} \rightarrow F * \text{id} \rightarrow \text{id} * \text{id}$$

- $\text{id} *$ is a prefix of a right sentential form, but it can never appear on the stack
  - Always reduce by $F \rightarrow \text{id}$ before shifting *
  - Not all prefixes of a right sentential form can appear on the stack

- There is no error as long as the parser has viable prefixes on the stack
Example of a Viable Prefix

\[ S \rightarrow X_1X_2X_3X_4 \]
\[ A \rightarrow X_1X_2 \]

Let \( w = X_1X_2X_3 \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$X_1X_2X_3$</td>
</tr>
<tr>
<td>$X_1</td>
<td>$X_2X_3$</td>
</tr>
<tr>
<td>$X_1X_2</td>
<td>$X_3$</td>
</tr>
<tr>
<td>$A</td>
<td>$X_3$</td>
</tr>
<tr>
<td>$AX_3</td>
<td>$</td>
</tr>
</tbody>
</table>

\( X_1X_2X_3 \) can never appear on a stack
Challenges with LR(0) Parsing

• An LR(0) parser works only if each state with a reduce action has only one possible reduce action and no shift action

{ \{ L \rightarrow L, S\bullet \} }

{ \{ L \rightarrow L, S\bullet \\ S \rightarrow S\bullet, L \} }

{ \{ L \rightarrow S, L\bullet \\ L \rightarrow S\bullet \} }

- Ok
- Shift-reduce conflict
- Reduce-reduce conflict

• Takes shift/reduce decisions without any lookahead token
  • Lacks the power to parse programming language grammars
Challenges with LR(0) Parsing

• Consider the following grammar for adding numbers

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

Left associative

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

Right associative
Challenges with LR(0) Parsing

- Consider the following grammar for adding numbers

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

Left associative

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

Right associative

Not LR(0)

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

Shift-reduce conflict
Simple LR Parsing

SLR(1)
Block Diagram of LR Parser

- Same driver program is used for all LR parsers
- Different LR parsing techniques produce different parse tables
LR Parsing Algorithm

- The parser driver is same for all LR parsers
  - Only the parsing table changes across parsers
- A shift-reduce parser shifts a symbol, and an LR parser shifts a state

- By construction, all transitions to state $j$ is for the same symbol $X$
  - Each state, except the start state, has a unique grammar symbol associated with it
SLR(1) Parsing

• Extends LR(0) parser to eliminate a few conflicts
  • Uses LR(0) items and LR(0) automaton
• For each reduction $A \rightarrow \beta$, look at the next symbol $c$
• Apply reduction only if $c \in \text{FOLLOW}(A)$ or $c = \epsilon$ and $S \Rightarrow \gamma A$
Structure of SLR Parsing Table

• Assume $S_i$ is top of the stack and $a_i$ is the current input symbol
• Parsing table consists of two parts: an Action and a Goto function
• Action table is indexed by state and terminal symbols
  • Action[$S_i$, $a_i$] can have four values
    • Shift $a_i$ to the stack, goto state $S_j$
    • Reduce by rule $k$
    • Accept
    • Error (empty cell in the table)
• Goto table is indexed by state and nonterminal symbols
Constructing SLR Parsing Table

1) Construct LR(0) canonical collection $C = \{I_0, I_1, ..., I_n\}$ for grammar $G'$

2) State $i$ is constructed from $I_i$
   a) If $[A \rightarrow \alpha \cdot a\beta]$ is in $I_i$ and Goto($I_i$, $a$) = $I_j$, then set Action[$i$, $a$] = “Shift $j$”
   b) If $[A \rightarrow \alpha \cdot]$ is in $I_i$, then set Action [$i$, $a$] = “Reduce $A \rightarrow \alpha$” for all $a$ in FOLLOW($A$)
   c) If $[S' \rightarrow S\cdot]$ is in $I_i$, then set Action [$i$, $\$$] = “Accept”

3) If Goto($I_i$, $A$) = $I_j$, then Goto[$i$, $A$] = $j$

4) All entries left undefined are “errors”
## SLR Parsing for Expression Grammar

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3</td>
<td>$T \rightarrow T * F$</td>
</tr>
<tr>
<td>4</td>
<td>$T \rightarrow F$</td>
</tr>
<tr>
<td>5</td>
<td>$F \rightarrow (E)$</td>
</tr>
<tr>
<td>6</td>
<td>$F \rightarrow id$</td>
</tr>
</tbody>
</table>

- *sj* means shift and stack state $i$
- *r{j}* means reduce by rule #$j$
- *acc* means accept
- blank means error
<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>8</td>
<td>s6</td>
<td>s11</td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>s7</td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>
LR Parser Configurations

• A LR parser configuration is a pair \(<s_0, s_1, ..., s_m, a_i a_{i+1} ... a_n \>$>
  • Left half is stack content, and right half is the remaining input
• Configuration represents the right sentential form \(X_1X_2 ... X_m a_i a_{i+1} ... a_n\)
LR Parsing Algorithm

• If Action\([s_m, a_i]\) = shift \(s\), new configuration is \(<s_0, s_1, ..., s_ms, a_{i+1}... a_n>\)

• If Action\([s_m, a_i]\) = reduce \(A \rightarrow \beta\), new configuration is \(<s_0, s_1, ..., s_{m−r}, a_i a_{i+1}... a_n>\)
  • Assume \(r\) is \(|\beta|\) and \(s = \text{Goto}[s_{m−r}, A]\)

• If Action\([s_m, a_i]\) = accept, parsing is successful

• If Action\([s_m, a_i]\) = error, parsing has discovered an error
LR Parsing Program

Let $a$ be the first symbol of input $w$
while (1)
  let $s$ be the top of the stack
  if Action[$a$] == shift $t$
    push $t$ onto the stack
    let $a$ be the next input symbol
  else if Action[$s$, $a$] == reduce $A \rightarrow \beta$
    pop $|\beta|$ symbols off the stack
    push Goto[$t$, $A$] onto the stack
    output production $A \rightarrow \beta$
  else if Action[$s$, $a$] == accept
    break
else
  invoke error recovery
Moves of an LR Parser on $id \ast id + id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$id \ast id + id$</td>
<td>Shift</td>
</tr>
<tr>
<td>2</td>
<td>0 5</td>
<td>id</td>
<td>$* id + id$</td>
</tr>
<tr>
<td>3</td>
<td>0 3</td>
<td>$F$</td>
<td>$* id + id$</td>
</tr>
<tr>
<td>4</td>
<td>0 2</td>
<td>$T$</td>
<td>$* id + id$</td>
</tr>
<tr>
<td>5</td>
<td>0 2 7</td>
<td>$T \ast$</td>
<td>id + id$</td>
</tr>
<tr>
<td>6</td>
<td>0 2 7 5</td>
<td>$T \ast id$</td>
<td>$+id$</td>
</tr>
<tr>
<td>7</td>
<td>0 2 7 10</td>
<td>$T \ast F$</td>
<td>$+id$</td>
</tr>
<tr>
<td>8</td>
<td>0 2</td>
<td>$T$</td>
<td>$+id$</td>
</tr>
<tr>
<td>9</td>
<td>0 1</td>
<td>$E$</td>
<td>$+id$</td>
</tr>
<tr>
<td>10</td>
<td>0 1 6</td>
<td>$E +$</td>
<td>id$</td>
</tr>
</tbody>
</table>
Moves of an LR Parser on $\text{id} \ast \text{id} + \text{id}$

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0 1 6 5</td>
<td>$E + \text{id}$</td>
<td>$,$</td>
<td>$,$</td>
</tr>
<tr>
<td>12</td>
<td>0 1 6 3</td>
<td>$E + F$</td>
<td>$,$</td>
<td>$,$</td>
</tr>
<tr>
<td>13</td>
<td>0 1 6 9</td>
<td>$E + T$</td>
<td>$,$</td>
<td>$,$</td>
</tr>
<tr>
<td>14</td>
<td>0 1</td>
<td>$E$</td>
<td>$,$</td>
<td>$,$</td>
</tr>
</tbody>
</table>

Reduce by $F \rightarrow \text{id}$
Reduce by $T \rightarrow F$
Reduce by $E \rightarrow E + T$
Accept
Limitations of SLR Parsing

• If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous

• Every SLR(1) grammar is unambiguous, but there are unambiguous grammars that are not SLR(1)
Limitations of SLR Parsing

Unambiguous grammar

\[
S \rightarrow L = R | R \\
L \rightarrow *R | \text{id} \\
R \rightarrow L
\]

Example Derivation

\[
S \Rightarrow L = R \Rightarrow *R = R
\]

FIRST:

\[
\text{FIRST}(S) = \text{FIRST}(L) = \text{FIRST}(R) = \{*, \text{id}\}
\]

FOLLOW:

\[
\text{FOLLOW}(S) = \text{FOLLOW}(L) = \text{FOLLOW}(R) = \{=, \}$
\]
Canonical LR(0) Collection

$I_0 = \text{Closure}(S' \rightarrow S) = \{ 
S' \rightarrow \cdot S,
S \rightarrow \cdot L = R,
S \rightarrow \cdot R,
L \rightarrow \cdot *R,
L \rightarrow \cdot \text{id},
R \rightarrow \cdot L
\}$

$I_1 = \text{Goto}(I_0, S) = \{ 
S' \rightarrow S \cdot
\}$

$I_2 = \text{Goto}(I_0, L) = \{ 
S \rightarrow L \cdot = R,
R \rightarrow L \cdot
\}$

$I_3 = \text{Goto}(I_0, R) = \{ 
S \rightarrow R \cdot
\}$

$I_4 = \text{Goto}(I_0, R) = \{ 
L \rightarrow * \cdot R,
R \rightarrow \cdot L,
L \rightarrow \cdot *R,
L \rightarrow \cdot \text{id}
\}$

$I_5 = \text{Goto}(I_0, \text{id}) = \{ 
L \rightarrow \cdot \text{id}
\}$

$I_6 = \text{Goto}(I_2, '=' ) = \{ 
S \rightarrow L = \cdot R,
R \rightarrow \cdot L,
L \rightarrow \cdot *R,
L \rightarrow \cdot \text{id}
\}$

$I_7 = \text{Goto}(I_4, R) = \{ 
L \rightarrow * \cdot R
\}$

$I_8 = \text{Goto}(I_4, L) = \{ 
R \rightarrow L \cdot
\}$

$I_9 = \text{Goto}(I_6, R) = \{ 
S \rightarrow L = R \cdot
\}$
### SLR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s4, s5</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6, r6</td>
<td>r6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s4, s5</td>
<td>8, 7</td>
</tr>
<tr>
<td>5</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>6</td>
<td>s4, s5</td>
<td>8, 9</td>
</tr>
<tr>
<td>7</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>8</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r2</td>
</tr>
</tbody>
</table>
1. Action[2,\rightarrow]= \text{Shift 6}
2. Action[2,\rightarrow]= \text{Reduce } R \rightarrow L \text{ since } \prime = \rightarrow \in \text{FOLLOW}(R)
## Moves of an LR Parser on \texttt{id=id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{id=id}$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0 \texttt{id} 5</td>
<td>\texttt{=id}$</td>
<td>Reduce by ( L \rightarrow \texttt{id} )</td>
</tr>
<tr>
<td>0 \texttt{L} 2</td>
<td>\texttt{=id}$</td>
<td>Reduce by ( R \rightarrow L )</td>
</tr>
<tr>
<td>0 \texttt{R} 3</td>
<td>\texttt{=id}$</td>
<td>Error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{id=id}$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0 \texttt{id} 5</td>
<td>\texttt{=id}$</td>
<td>Reduce by ( L \rightarrow \texttt{id} )</td>
</tr>
<tr>
<td>0 \texttt{L} 2</td>
<td>\texttt{=id}$</td>
<td>Shift 6</td>
</tr>
<tr>
<td>0 \texttt{L} 2 6</td>
<td>\texttt{id}$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0 \texttt{L} 2 6 \texttt{id} 5</td>
<td>$</td>
<td>Reduce by ( L \rightarrow \texttt{id} )</td>
</tr>
<tr>
<td>0 \texttt{L} 2 6 \texttt{L} 8</td>
<td>$</td>
<td>Reduce by ( R \rightarrow L )</td>
</tr>
<tr>
<td>0 \texttt{L} 2 6 \texttt{R} 9</td>
<td>$</td>
<td>Reduce by ( S \rightarrow L = R )</td>
</tr>
<tr>
<td>0 \texttt{S} 1</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Moves of an LR Parser on $\text{id}=\text{id}$

• State $i$ calls for a reduction by $A \rightarrow \alpha$ if the set of items $I_i$ contains item $[A \rightarrow \alpha \bullet]$ and $\alpha \in \text{FOLLOW}(A)$

• Suppose $\beta A$ is a viable prefix on top of the stack

• There may be no right sentential form where $\alpha$ follows $\beta A$
  • No right sentential form begins with $R = \cdots$
  ➢ Parser should not reduce by $A \rightarrow \alpha$

| 0 L 2 = 6 R 9 | $\varepsilon$ | Reduce by $S \rightarrow L = R$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 S 1</td>
<td>$\varepsilon$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Moves of an LR Parser on **id=id**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id=id$</td>
<td>Shift 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id=id$</td>
<td>Shift 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLR parsers cannot remember the left context**

- SLR(1) states only tell us about the sequence on top of the stack, not what is below on the stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 L 2 = 6 id 5</td>
<td>$</td>
<td>Reduce by $L \rightarrow id$</td>
</tr>
<tr>
<td>0 L 2 = 6 L 8</td>
<td>$</td>
<td>Reduce by $R \rightarrow L$</td>
</tr>
<tr>
<td>0 L 2 = 6 R 9</td>
<td>$</td>
<td>Reduce by $S \rightarrow L = R$</td>
</tr>
<tr>
<td>0 S 1</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Canonical LR Parsing
LR(1) Item

• An LR(1) item of a CFG $G$ is a string of the form $[A \to \alpha \cdot \beta, a]$
  • $A \to \alpha \beta$ is a production in $G$, and $a \in T \cup \{\}$
  • There is now one symbol lookahead
• Suppose $[A \to \alpha \cdot \beta, a]$ where $\beta \neq \epsilon$, then the lookahead does not help
• If $[A \to \alpha \cdot, a]$, reduce only if next input symbol is $a$
  • Set of possible terminals will always be a subset of FOLLOW($A$), but can be a proper subset
LR(1) Item

• An LR(1) item \([A \rightarrow \alpha \cdot \beta, a]\) is valid for a viable prefix \(\gamma\) if there is a derivation

\[
S \stackrel{\delta}{\Rightarrow}^* \delta Aw \Rightarrow_{rm} \delta\alpha\beta w
\]

where

i. \(\gamma = \delta a\), and

ii. \(a\) is first symbol of \(w\) or \(w = \varepsilon\) and \(a = \$\)
Constructing LR(1) Sets of Items

**Closure(I)**

```
repeat
    for each item \([A \rightarrow \alpha \bullet B \beta, a]\) in \(I\)
        for each production \(B \rightarrow \gamma\) in \(G'\)
            for each terminal \(b\) in \(\text{FIRST}(\beta a)\)
                add \([B \rightarrow \bullet \gamma, b]\) to set \(I\)
        until no more items are added to \(I\)
return \(I\)
```

**Goto(I, X)**

```
initialize \(J\) to be the empty set
for each item \([A \rightarrow \alpha \bullet X \beta, a]\) in \(I\)
    add item \([A \rightarrow aX \bullet \beta, a]\) to set \(J\)
return \(\text{Closure}(J)\)
```
Constructing LR(1) Sets of Items

\[
\text{Constructing LR(1) Sets of Items}
\]

\[
\text{Items}(G')
\]

\[
C = \text{Closure}([[S' \rightarrow \bullet S, \$]])
\]

\[
\text{repeat}
\]

\[
\text{for each set of items } I \text{ in } C
\]

\[
\text{for each grammar symbol } X
\]

\[
\text{if } \text{Goto}(I, X) \neq \phi \text{ and } \text{Goto}(I, X) \notin C
\]

\[
\text{add } \text{Goto}(I, X) \text{ to } C
\]

\[
\text{until no new sets of items are added to } C
\]
Example Construction of LR(1) Items

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S' \rightarrow S$</td>
</tr>
<tr>
<td>1</td>
<td>$S \rightarrow CC$</td>
</tr>
<tr>
<td>2</td>
<td>$C \rightarrow cC$</td>
</tr>
<tr>
<td>3</td>
<td>$C \rightarrow d$</td>
</tr>
</tbody>
</table>

$I_0 = \text{Closure}([S' \rightarrow \bullet S, \$]) = \{ 
S' \rightarrow \bullet S, \$
S \rightarrow \bullet CC, \$
C \rightarrow \bullet cC, c/d,
C \rightarrow \bullet d, c/d
\}$

$I_1 = \text{Goto}(I_0, S) = \{ 
S' \rightarrow S \bullet, \$
\}$

generates the regular language $c^* dc^* d$
Example Construction of LR(1) Items

\[ I_0 = \text{Closure}([S' \to S, \$, S \to \$]) = \{ \]
\[ S' \to \$S, \$
\[ S \to \$CC, \$
\[ C \to \$cC, c/d, \]
\[ C \to \$d, c/d \]
\}\n
\[ I_1 = \text{Goto}(I_0, S) = \{ \]
\[ S' \to S \$
\}\n
\[ I_2 = \text{Goto}(I_0, C) = \{ \]
\[ S \to \$C, \$
\[ C \to \$cC, \$
\[ C \to \$d, \$
\}\n
\[ I_3 = \text{Goto}(I_0, c) = \{ \]
\[ C \to \$cC, c/d, \]
\[ C \to \$cC, c/d, \]
\[ C \to \$d, c/d \]
\}\n
\[ I_4 = \text{Goto}(I_0, d) = \{ \]
\[ C \to \$d, c/d \]
\}\n
\[ I_5 = \text{Goto}(I_2, C) = \{ \]
\[ C \to \$CC, \$
\}\n
\[ I_6 = \text{Goto}(I_2, c) = \{ \]
\[ C \to \$cC, \$
\[ C \to \$cC, \$
\[ C \to \$d, \$
\}\n
\[ I_7 = \text{Goto}(I_2, d) = \{ \]
\[ C \to \$d, \$
\}\n
\[ I_8 = \text{Goto}(I_3, C) = \{ \]
\[ C \to \$cC, c/d \]
\}\n
\[ I_9 = \text{Goto}(I_6, C) = \{ \]
\[ C \to \$cC, \$
\}\}
LR(1) Automaton
Construction of Canonical LR(1) Parsing Tables

• Construct $C' = \{I_0, I_1, ..., I_n\}$

• State $i$ of the parser is constructed from $I_i$
  • If [$A \rightarrow \alpha \cdot a \beta, b$] is in $I_i$ and Goto($I_i, a$) = $I_j$, then set Action[$i, a$] = “shift $j$”
  • If [$A \rightarrow \alpha \cdot, a$] is in $I_i$, $A \neq S'$, then set Action[$i, a$] = “reduce $A \rightarrow \alpha \cdot$”
  • If [$S' \rightarrow S \cdot, \$] is in $I_i$, then set Action[$i, \$]$ = “accept”

• If Goto($I_i, A$) = $I_j$, then Goto[$i, A$] = $j$

• Initial state of the parser is constructed from the set of items containing [$S' \rightarrow \cdot S, \$]$
### Canonical LR(1) Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c$</td>
<td>$d$</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s6</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s6</td>
<td>s7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Canonical LR(1) Parsing

• If the parsing table has no multiply-defined cells, then the corresponding grammar $G$ is LR(1)

• Every SLR(1) grammar is an LR(1) grammar
  • Canonical LR parser may have more states than SLR
LALR Parsing
Example Construction of LR(1) Items

\[ I_0 = \text{Closure}([S' \to S, \$]) = \{ \]
\[ S' \to \cdot S, \$
\[ S \to \cdot C C, \$
\[ C \to \cdot c C, c/d,
\[ C \to \cdot d, c/d \]
\}\n
\[ I_1 = \text{Goto}(I_0, S) = \{ \]
\[ S' \to S \cdot , \$
\}\n
\[ I_2 = \text{Goto}(I_0, C) = \{ \]
\[ S \to C \cdot C, \$
\[ C \to \cdot c C, \$
\[ C \to \cdot d, \$
\}\n
\[ I_3 = \text{Goto}(I_0, c) = \{ \]
\[ C \to c \cdot C, c/d,
\[ C \to \cdot c C, c/d,
\[ C \to \cdot d, c/d \]
\}\n
\[ I_6 = \text{Goto}(I_2, c) = \{ \]
\[ C \to c \cdot C, \$
\[ C \to \cdot c C, \$
\[ C \to \cdot d, \$
\}\n
\[ I_4 = \text{Goto}(I_0, d) = \{ \]
\[ C \to d \cdot , c/d \]
\}\n
\[ I_7 = \text{Goto}(I_2, d) = \{ \]
\[ C \to d \cdot , \$
\}\n
\[ I_5 = \text{Goto}(I_2, C) = \{ \]
\[ C \to C C \cdot , \$
\]\n
\[ I_8 = \text{Goto}(I_3, C) = \{ \]
\[ C \to c C \cdot , c/d \]
\}\n
\[ I_9 = \text{Goto}(I_6, C) = \{ \]
\[ C \to c C \cdot , \$
\}\n
\[ I_3 \text{ and } I_6, I_4 \text{ and } I_7, \text{ and } I_8 \text{ and } I_9 \text{ only differ in the second components} \]
Lookahead LR (LALR) Parsing

• CLR(1) parser has a large number of states

• Lookahead LR (LALR) parser
  • Merge sets of LR(1) items that have the same core, i.e., first component
    • A core is a set of LR(0) items
  • LALR parser is used in many parser generators (for e.g., Yacc and Bison)
    • Fewer number of states, same as SLR
Construction of LALR Parsing Table

• Construct $C = \{I_0, I_1, ..., I_n\}$, the collection of sets of LR(1) items
• For each core present in LR(1) items, find all sets having the same core and replace these sets by their union
• Let $C' = \{J_0, J_1, ..., J_n\}$ be the resulting sets of LR(1) items
  • Also called LALR collection
• Construct Action table as was done earlier, parsing actions for state $i$ is constructed from $J_i$
• Let $J = I_1 \cup I_2 \cup \cdots \cup I_k$, where the cores of $I_1, I_2, ..., I_k$ are same.
  • Cores of $\text{Goto}(I_1, X), \text{Goto}(I_2, X), ..., \text{Goto}(I_k, X)$ will also be the same.
  • Let $K = \text{Goto}(I_1, X) \cup \text{Goto}(I_2, X) \cup \cdots \cup \text{Goto}(I_k, X)$, then $\text{Goto}(J, X) = K$
LALR Grammar

• If there are no parsing action conflicts, then the grammar is LALR(1)

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S' \rightarrow S$</td>
</tr>
<tr>
<td>1</td>
<td>$S \rightarrow CC$</td>
</tr>
<tr>
<td>2</td>
<td>$C \rightarrow cC$</td>
</tr>
<tr>
<td>3</td>
<td>$C \rightarrow d$</td>
</tr>
</tbody>
</table>

$I_{36} = \text{Goto}(I_0, c) = \{ 
\quad C \rightarrow c\cdot C, c/d/$, 
\quad C \rightarrow \cdot cC, c/d/$, 
\quad C \rightarrow \cdot d, c/d/$
\}$

$I_{89} = \text{Goto}(I_3, C) = \{ 
\quad C \rightarrow cC\cdot, c/d/$
\}$

$I_{47} = \text{Goto}(I_0, d) = \{ 
\quad C \rightarrow d\cdot, c/d/$
\}$
## LALR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$$$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$s36$</td>
<td>$s47$</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$s36$</td>
<td>$s47$</td>
</tr>
<tr>
<td>36</td>
<td>$s36$</td>
<td>$s47$</td>
</tr>
<tr>
<td>47</td>
<td>$r3$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>$r2$</td>
<td></td>
</tr>
</tbody>
</table>
Notes on LALR Parsing Table

- Modified parser behaves as original
- Merging items can **never** produce shift/reduce conflicts
  - Suppose there is a conflict on lookahead $a$
  - Shift due to item $[B \rightarrow \beta \cdot a\gamma, b]$ and reduce due to item $[A \rightarrow \alpha\cdot, a]$
  - But merged state was formed from states with same cores
- Merging items **may** produce reduce/reduce conflicts
Reduce-Reduce Conflicts due to Merging

LR(1) grammar

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow aAd | bBd | aBe | bAe \\
A & \rightarrow c \\
B & \rightarrow c
\end{align*}
\]

acd, ace, bcd, bce

\[
\begin{align*}
\{[A \rightarrow c \cdot , d], [B \rightarrow c \cdot , e]\} & \quad \{[A \rightarrow c \cdot , e], [B \rightarrow c \cdot , d]\} \\
\{[A \rightarrow c \cdot , d/c], [B \rightarrow c \cdot , d/e]\}
\end{align*}
\]
Dealing with Errors with LALR Parsing

• Consider an erroneous input \( ccd \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( s3 )</td>
<td>( s4 )</td>
</tr>
<tr>
<td>1</td>
<td>( acc )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( s6 )</td>
<td>( s7 )</td>
</tr>
<tr>
<td>3</td>
<td>( s3 )</td>
<td>( s4 )</td>
</tr>
<tr>
<td>4</td>
<td>( r3 )</td>
<td>( r3 )</td>
</tr>
<tr>
<td>5</td>
<td>( r1 )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( s6 )</td>
<td>( s7 )</td>
</tr>
<tr>
<td>7</td>
<td>( r3 )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( r2 )</td>
<td>( r2 )</td>
</tr>
<tr>
<td>9</td>
<td>( r2 )</td>
<td></td>
</tr>
</tbody>
</table>

CLR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( s36 )</td>
<td>( s47 )</td>
</tr>
<tr>
<td>1</td>
<td>( acc )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( s36 )</td>
<td>( s47 )</td>
</tr>
<tr>
<td>3</td>
<td>( s36 )</td>
<td>( s47 )</td>
</tr>
<tr>
<td>47</td>
<td>( r3 )</td>
<td>( r3 )</td>
</tr>
<tr>
<td>5</td>
<td>( r1 )</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>( r2 )</td>
<td>( r2 )</td>
</tr>
</tbody>
</table>

LALR Parsing Table

<table>
<thead>
<tr>
<th>#</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S' \rightarrow S )</td>
</tr>
<tr>
<td>1</td>
<td>( S \rightarrow CC )</td>
</tr>
<tr>
<td>2</td>
<td>( C \rightarrow cC )</td>
</tr>
<tr>
<td>3</td>
<td>( C \rightarrow d )</td>
</tr>
</tbody>
</table>
Dealing with Errors with LALR Parsing

- Consider an erroneous input \( ccd \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( c )</td>
<td>( S )</td>
</tr>
<tr>
<td>1</td>
<td>( d )</td>
<td>( C )</td>
</tr>
<tr>
<td>2</td>
<td>( $ )</td>
<td>( S )</td>
</tr>
</tbody>
</table>

**CLR Parsing Table**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>( c )</td>
<td>( S )</td>
</tr>
<tr>
<td>37</td>
<td>( d )</td>
<td>( C )</td>
</tr>
<tr>
<td>38</td>
<td>( $ )</td>
<td>( S )</td>
</tr>
</tbody>
</table>

**LALR Parsing Table**

- CLR parser will not even reduce before reporting an error
- SLR and LALR parsers may reduce several times before reporting an error
  - Will never shift an erroneous input symbol onto the stack
Using Ambiguous Grammars
Dealing with Ambiguous Grammars

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

\[ I_0 = \text{Closure}\{E' \rightarrow \text{\cdot}E\} = \{ \]
\[ E' \rightarrow \text{\cdot}E, \]
\[ E \rightarrow \text{\cdot}E + E, \]
\[ E \rightarrow \text{\cdot}E * E, \]
\[ E \rightarrow \text{\cdot}(E), \]
\[ E \rightarrow \text{\cdot}\text{id} \]
\[ \} \]

\[ I_1 = \text{Goto}(I_0, E) = \{ \]
\[ E' \rightarrow E \text{\cdot}, \]
\[ E \rightarrow E \text{\cdot} + E, \]
\[ E \rightarrow E \text{\cdot} * E \]
\[ \} \]

\[ I_2 = \text{Goto}(I_0, ') = \{ \]
\[ E \rightarrow (\text{\cdot}E), \]
\[ E \rightarrow \text{\cdot}E + E, \]
\[ E \rightarrow \text{\cdot}E * E, \]
\[ E \rightarrow \text{\cdot}(E), \]
\[ E \rightarrow \text{\cdot}\text{id} \]
\[ \} \]

\[ I_3 = \text{Goto}(I_0, \text{id}) = \{ \]
\[ E \rightarrow \text{id}\text{\cdot} \]
\[ \} \]

\[ I_4 = \text{Goto}(I_0, '+') = \{ \]
\[ E \rightarrow E + \text{\cdot}E, \]
\[ E \rightarrow \text{\cdot}E + E, \]
\[ E \rightarrow \text{\cdot}E * E, \]
\[ E \rightarrow \text{\cdot}(E), \]
\[ E \rightarrow \text{\cdot}\text{id} \]
\[ \} \]

\[ I_5 = \text{Goto}(I_0, '*') = \{ \]
\[ E \rightarrow E * \text{\cdot}E, \]
\[ E \rightarrow \text{\cdot}E + E, \]
\[ E \rightarrow \text{\cdot}E * E, \]
\[ E \rightarrow \text{\cdot}(E), \]
\[ E \rightarrow \text{\cdot}\text{id} \]
\[ \} \]

\[ I_6 = \text{Goto}(I_2, E) = \{ \]
\[ E \rightarrow (E\text{\cdot}), \]
\[ E \rightarrow E\text{\cdot} + E, \]
\[ E \rightarrow E\text{\cdot} * E, \]
\[ \} \]

\[ I_7 = \text{Goto}(I_4, E) = \{ \]
\[ E \rightarrow E + E\text{\cdot}, \]
\[ E \rightarrow E\text{\cdot} + E, \]
\[ E \rightarrow E\text{\cdot} * E, \]
\[ \} \]

\[ I_8 = \text{Goto}(I_5, E) = \{ \]
\[ E \rightarrow E * E\text{\cdot}, \]
\[ E \rightarrow E\text{\cdot} + E, \]
\[ E \rightarrow E\text{\cdot} * E \]
\[ \} \]
## SLR(1) Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>s4, s5</td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4, r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>6</td>
<td>s4, s5</td>
<td>s9</td>
</tr>
<tr>
<td>7</td>
<td>s4, r1</td>
<td>s5, r1</td>
</tr>
<tr>
<td>8</td>
<td>s4, r2</td>
<td>s5, r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>

### State
- **id**: Production rule
- **+**: Production rule
- **\(\ast\)**: Production rule
- **( )**: Production rule
- **\(\$\)**: Production rule

### Goto
- **E**: Production rule
Moves of an SLR Parser on \( \text{id + id} \ast \text{id} \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>id + id * id$</td>
<td>Shift 3</td>
</tr>
<tr>
<td>2</td>
<td>0 3</td>
<td>id</td>
<td>+id * id$</td>
</tr>
<tr>
<td>3</td>
<td>0 1</td>
<td>( E )</td>
<td>( +id \ast id$ )</td>
</tr>
<tr>
<td>4</td>
<td>0 1 4</td>
<td>( E + )</td>
<td>( id \ast id$ )</td>
</tr>
<tr>
<td>5</td>
<td>0 1 4 3</td>
<td>( E + \text{id} )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0 1 4 7</td>
<td>( E + E )</td>
<td>( \ast id$ )</td>
</tr>
</tbody>
</table>
## SLR(1) Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>s4, s5</td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>$</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>$</td>
</tr>
<tr>
<td>6</td>
<td>s4, s5</td>
<td>$</td>
</tr>
<tr>
<td>7</td>
<td>s4, r1</td>
<td>$</td>
</tr>
<tr>
<td>8</td>
<td>s4, r2</td>
<td>$</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>$</td>
</tr>
</tbody>
</table>
Summary
Comparisons across Techniques

• SLR(1) = LR(0) items + FOLLOW
  • SLR(1) parsers can parse a larger number of grammars than LR(0)
  • Any grammar that can be parsed by an LR(0) parser can be parsed by an SLR(1) parser

• SLR(1) ≤ LALR(1) ≤ LR(1)
• SLR(k) ≤ LALR(k) ≤ LR(k)
• LL(k) ≤ LR(k)
• Ambiguous grammars are not LR
Summary

• Bottom-up parsing is a more powerful technique compared to top-down parsing
  • LR grammars can handle left recursion
  • Detects errors as soon as possible, and allows for better error recovery
• Automated parser generators such as Yacc and Bison
References