Summary of Lecture VI—Part 3

- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables
Intro to LR Parsers

- LR(k) Grammars are the most general Non-Backtracking Grammars that can be used in bottom-up parsers.
  - “L”: Left-to-right scanning of the input;
  - “R”: Rightmost derivations;
  - “k”: number of lookahead symbols to take a decision.

- Predictive Grammars, i.e., LL Grammars, are a proper subset of LR Grammars (e.g., if-then-else is not LL but it is LR).

- An LR parser can detect a syntactic error as soon as possible.

- Disadvantage. Is difficult to build an LR parser by hand. We need specialized tools like YACC.
LR Parser Architecture

An LR parser has: An input buffer (Tokens returned from the Lexical Analyzer); A stack containing Grammar symbols and States; A parsing table with two parts, Action and Goto, implementing a DFA to decide between shift and reduce.
LR Parsing Algorithm

- The stack stores a string of the form $s_0X_1s_1 \ldots s_{m-1}X_ms_m$, where:
  - $X_i$ is a Grammar Symbol;
  - $s_i$ is a state summarizing the information contained in the stack below it.
- The combination $\langle$ State on top of the stack, Lookahead symbol $\rangle$ is used to index the Action-Goto table.
- Configuration of an LR Parser. Is a pair made by the content of the stack ($s_m$ on top) and the right-part of the input (starting with the Lookahead):
  $$\langle s_0X_1s_1 \ldots s_{m-1}X_ms_m, a_ia_{i+1} \ldots a_n$\rangle$$
LR Parsing Algorithm (Cont.)

The next move of the parser is based on the pair \((s_m, a_i)\) and on what specified in the \textit{Action} table:

1. \textit{action}[s_m, a_i] = \textit{shift} \ s_j. The parser executes a \textit{shift} entering the configuration: \(\langle s_0X_1s_1 \ldots X_ms_ma_is_j, a_{i+1} \ldots a_n\rangle\).

2. \textit{action}[s_m, a_i] = \textit{reduce} \ A \rightarrow \beta. The parser executes a \textit{reduce} entering the configuration: \(\langle s_0X_1s_1 \ldots X_{m-r}s_{m-r}As, a_ia_{i+1} \ldots a_n\rangle\); where \(s = \textit{goto}(s_{m-r}, A)\) and \(r = |\beta|\). The parser pops \(2r\) symbols from the stack (\(r\) states and the \(r\) Grammar symbols \(\beta\)) and then pushes both \(A\) and \(s\). The production \(A \rightarrow \beta\) is in the output.

3. \textit{action}[s_m, a_i] = \textit{error}.

4. \textit{action}[s_m, \$] = \textit{accept}. The parser stops successfully.
Example: LR Parser on “id*id+id”

Grammar

r1. $E \rightarrow E + T$

r2. $E \rightarrow T$

r3. $T \rightarrow T * F$

r4. $T \rightarrow F$

r5. $F \rightarrow (E)$

r6. $F \rightarrow id$
Example: LR Parser on "id*id+id"

<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>acc</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>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>0 id 5</td>
<td>id * id + id $</td>
</tr>
<tr>
<td>(3)</td>
<td>0 F 3</td>
<td>* id + id $</td>
</tr>
<tr>
<td>(4)</td>
<td>0 T 2</td>
<td>* id + id $</td>
</tr>
<tr>
<td>(5)</td>
<td>0 T 2 * 7</td>
<td>* id + id $</td>
</tr>
<tr>
<td>(6)</td>
<td>0 T 2 * 7 id 5</td>
<td>id + id $</td>
</tr>
<tr>
<td>(7)</td>
<td>0 T 2 * 7 F 10</td>
<td>+ id $</td>
</tr>
<tr>
<td>(8)</td>
<td>0 T 2</td>
<td>+ id $</td>
</tr>
<tr>
<td>(9)</td>
<td>0 E 1</td>
<td>+ id $</td>
</tr>
<tr>
<td>(10)</td>
<td>0 E 1 + 6</td>
<td>+ id $</td>
</tr>
<tr>
<td>(11)</td>
<td>0 E 1 + 6 id 5</td>
<td>id $</td>
</tr>
<tr>
<td>(12)</td>
<td>0 E 1 + 6 F 3</td>
<td>$</td>
</tr>
<tr>
<td>(13)</td>
<td>0 E 1 + 6 T 9</td>
<td>$</td>
</tr>
<tr>
<td>(14)</td>
<td>0 E 1</td>
<td>$</td>
</tr>
</tbody>
</table>

shift | reduce by F → id
reduce by T → F
reduce by T → F
reduce by F → id
reduce by T → T*F
shift | reduce by E → T
shift | reduce by F → id
reduce by T → F
reduce by E → T
accept
Summary

• Parsing Algorithm: An Intro
• Automata and Bottom-up Parsing
• SLR Parsing
  ▶ Closure and Goto Operations, Canonical Collection;
  ▶ SLR Parsing Tables
• **Definition 1.** **Right-Sentential Form:** A string $\alpha$ derived from the scope of the language, $S \Rightarrow_{rm}^* \alpha$, by means of right-most derivations.

• **Definition 2.** **Handle:** Substring of a right-sentential form that matches a right hand side of a production.

• **The Handle will always appear on top of the stack, never inside.**
• The *Action* and *Goto* tables define the transition function of an Automaton that recognizes handles on top of the stack.

• The automaton does not need to read the stack every time: The state on top of the stack is the state the automaton would be after reading the symbols of the stack.

• This is why an LR parser has full control on the content of the stack just knowing the state on top of the stack.
Summary

- Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables
SLR Parsers

- **Simple LR (SLR)** is the simplest LR parsing Grammar.
- **Definition.** An LR(0) Item of a Grammar, $G$, is a production with a dot at some position in the right side.
- **Example.** The production $A \rightarrow XYZ$ gives rise to four items:
  
  $$
  A \rightarrow .\, XYZ \\
  A \rightarrow X\, .\, YZ \\
  A \rightarrow XY\, .\, Z \\
  A \rightarrow XYZ.
  $$

  The production $A \rightarrow \epsilon$ generates the item $A \rightarrow$.

- **Intuition.** An item indicates how much of a production we have seen in the parsing process, and can be represented by a pair of integers:

  $$\langle \text{Number of Production, Dot Position} \rangle$$
Constructing SLR Parsing Tables

• Items are useful to build the transition function of the Automaton recognizing handles.
• Items representing the same situation are grouped together into sets.
• Each of these sets represents a state of the DFA recognizing handles.
• The Canonical Collection of LR(0) Items provides the basis to construct the SLR parsing tables (implementing the DFA).
• The canonical collection is defined in terms of two operations, Closure and Goto, and an Augmented Grammar, i.e., a Grammar with a new scope $S'$ and a new production $S' \rightarrow S$.
  ▶ The production $S' \rightarrow S$ indicates acceptance, i.e., the parser accepts iff it is about to reduce by $S' \rightarrow S$. 
Summary

- Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables
The Closure Operation

• **Algorithm.** *Closure(I).*
  If \( I \) is a set of items for an augmented Grammar \( G' \), then \( \text{closure}(I) \) is the set of items such that:
  
  1. Initially every item in \( I \) is added to \( \text{closure}(I) \);
  2. If \( A \rightarrow \alpha. B\beta \in \text{closure}(I) \) and \( B \rightarrow \gamma \), then we add the item \( B \rightarrow \gamma \) to \( \text{closure}(I) \). Go to step 1 until no more items can be added to \( \text{closure}(I) \).

• **Intuition.** \( A \rightarrow \alpha. B\beta \in \text{closure}(I) \) indicates that:
  
  1. We expect to see something derivable from \( A \), and
  2. \( \alpha \) is already on top of the stack, thus
  3. we expect to see something derivable from \( B\beta \), and then
  4. if \( B \rightarrow \gamma \) we could also expect something derivable from \( \gamma \).
The Closure Operation: An Example

- **Example.** Consider the augmented grammar on the left, then, $\text{closure}(\{E' \rightarrow .E\})$ contains the items shown on the right:

<table>
<thead>
<tr>
<th>Augmented Grammar</th>
<th>$\text{closure}({E' \rightarrow .E})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E' \rightarrow E$</td>
<td>$E' \rightarrow .E$</td>
</tr>
<tr>
<td>$E \rightarrow E + T$</td>
<td>$E \rightarrow .E + T$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E \rightarrow .T$</td>
</tr>
<tr>
<td>$T \rightarrow T * F$</td>
<td>$T \rightarrow .T * F$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T \rightarrow .F$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow .(E)$</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>$F \rightarrow .\text{id}$</td>
</tr>
</tbody>
</table>
The Goto Operation

- **Definition.** If $I$ is a set of items and $X \in V_N \cup V_T$, then, $\text{goto}(I, X)$ is the *closure* of the set of all items $A \rightarrow \alpha X.\beta$ such that $A \rightarrow \alpha. X\beta$ is in $I$.

- **Intuition 1.** $\text{goto}(I, X)$ represents the transition of the automaton from state $I$ and input $X$.

- **Intuition 2.** If $I$ is a set of items valid for a prefix $\alpha$ of a right-sentential form, then, $\text{goto}(I, X)$ is valid for the prefix $\alpha X$. 
Example. If \( I = \{ E' \to E, \ E \to E. + T \} \), then:
\[
goto(I, +) = \text{closure}(\{ E \to E. + T \})
\]
is the set:
\[
\begin{align*}
E & \to \ E+. T \\
T & \to \ . T * F \\
T & \to \ . F \\
F & \to \ . (E) \\
F & \to \ . \text{id}
\end{align*}
\]
Algorithm. *Canonical Collection for an Augmented Grammar $G'$*

1. Initially, $C = \{\text{closure}(\{S' \to S\})\}$;
2. For each set of items $I$ in $C$ and each Grammar symbol $X$
   If $\text{goto}(I, X) \neq \emptyset$ and $\text{goto}(I, X) \notin C$, then
   add $\text{goto}(I, X)$ to $C$;
3. Go to step 2 if new items have been added, otherwise stop.
Example: Canonical Collection for Arithmetic Expressions

<table>
<thead>
<tr>
<th>I</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$E' \rightarrow .E$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow .E + T$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow .T$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow .T \ast F$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow .F$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .(E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .id$</td>
</tr>
<tr>
<td>1</td>
<td>$E' \rightarrow E.$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow E+.T$</td>
</tr>
<tr>
<td>2</td>
<td>$E \rightarrow T.$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T\ast F$</td>
</tr>
<tr>
<td>3</td>
<td>$T \rightarrow F.$</td>
</tr>
<tr>
<td>4</td>
<td>$F \rightarrow .(E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .id$</td>
</tr>
<tr>
<td>5</td>
<td>$F \rightarrow id.$</td>
</tr>
<tr>
<td>6</td>
<td>$E \rightarrow E+.T$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow .T \ast F$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow .F$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .(E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .id$</td>
</tr>
<tr>
<td>7</td>
<td>$T \rightarrow T\ast F$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .(E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow .id$</td>
</tr>
<tr>
<td>8</td>
<td>$F \rightarrow (E.)$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow E+.T$</td>
</tr>
<tr>
<td>9</td>
<td>$E \rightarrow E+.T.$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T\ast F$</td>
</tr>
<tr>
<td>10</td>
<td>$T \rightarrow T\ast F$</td>
</tr>
<tr>
<td>11</td>
<td>$F \rightarrow (E.)$</td>
</tr>
</tbody>
</table>
Example: Canonical Collection for Arithmetic Expressions

Augmented Grammar

r0. $E' \rightarrow E$

r1. $E \rightarrow E + T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \ast F$

r4. $T \rightarrow F$

r5. $F \rightarrow (E)$

r6. $F \rightarrow \text{id}$
Example: Canonical Collection for Arithmetic Expressions
Example: Canonical Collection for Arithmetic Expressions

\[ \begin{align*}
I_0 : & \quad E' \rightarrow .E \\
& \quad E \rightarrow .E + T \\
& \quad E \rightarrow .T \\
& \quad T \rightarrow .T*F \\
& \quad T \rightarrow .F \\
& \quad F \rightarrow .(E) \\
& \quad F \rightarrow .id \\
I_1 : & \quad E' \rightarrow E. \\
& \quad E \rightarrow E.+T \\
I_2 : & \quad E \rightarrow T. \\
& \quad T \rightarrow T.*F \\
I_3 : & \quad T \rightarrow F. \\
I_4 : & \quad F \rightarrow .(E) \\
& \quad E \rightarrow .E + T \\
& \quad E \rightarrow .T \\
& \quad T \rightarrow .T*F \\
& \quad T \rightarrow .F \\
& \quad F \rightarrow .(E) \\
& \quad F \rightarrow .id \\
I_5 : & \quad F \rightarrow \text{id.} \\
I_6 : & \quad E \rightarrow E.+T \\
& \quad T \rightarrow .T*F \\
& \quad T \rightarrow .F \\
& \quad F \rightarrow .(E) \\
& \quad F \rightarrow .id \\
I_7 : & \quad T \rightarrow T.*F \\
& \quad F \rightarrow .(E) \\
& \quad F \rightarrow .id \\
I_8 : & \quad F \rightarrow .(E.) \\
& \quad E \rightarrow E.+T \\
I_9 : & \quad E \rightarrow E + T. \\
& \quad T \rightarrow T.*F \\
I_{10} : & \quad T \rightarrow T.*F. \\
I_{11} : & \quad F \rightarrow .(E). \\
\end{align*} \]
The above figure represents the transition function of the DFA recognizing *viable prefixes* of the Grammar for Arithmetic Expressions.

**Viable Prefix**: Prefix of right-sentential form that could be on top of the stack of an SLR Parser.
Summary

- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables
Algorithm. SLR Parsing Tables *Action* and *Goto*.

1. Construct $C = \{l_0, l_1, \ldots, l_n\}$, the canonical collection for the augmented grammar $G'$.
2. To each item set $l_k$ we create a new state $s_k$. Then the *action* table is:
   - $\text{action}[s_k, a] = \text{"shift } s_j\text{"}$, if $A \rightarrow \alpha. a\beta \in l_k$, and $\text{goto}(l_k, a) = l_j$.
   - $\text{action}[s_k, a] = \text{"reduce } A \rightarrow \alpha\text{"}$, for all $A \rightarrow \alpha. \in l_k$, and for all $a$ in FOLLOW($A$). Here $A \neq S'$.
   - $\text{action}[s_k, $] = \text{"accept"}$, if $S' \rightarrow S. \in l_k$.
3. $\text{goto}[s_k, A] = s_j$, if $\text{goto}(l_k, A) = l_j$.
4. All the entries not defined by rules (2) and (3) are made “error”.
5. The initial state, $l_0$, is the one constructed from the closure of $S' \rightarrow S$.

**Note.** The Parsing table does not contains multiple entries if and only if the Grammar is SLR.
Summary of Lecture VI—Part 3

- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables