Syntax Analysis – LR(0) and SLR Parsers

Wei Wang
Where We Are

Textbook Chapters

• Dragon Book
  - Chapter 4.6 “Introduction to LR Parsing: Simple LR”
LR Parers
LR Parsers

- **LR parsers** are a family of bottom-up parsers.
  - *L* means the parsers read input string from *left-to-right*.
  - *R* means the parsers eventually found a *right-most derivation* for the input string.
  - Recall that our bottom-up parsing example was a left-to-right, right-most parsing.

- The LR parsers discussed in this course:
  - LR(0)
  - SLR(1)
  - If we have time
    - LR(1)
    - LALR
Intuitions of LR Parsers

• Intuitively,
  – LR parsers predict:
    • Whether to shift or reduce.
    • If reduce, which rule should be use.
  – based on:
    • Symbols in the stack
    • Symbols in the input string
    • The production rules used so far.

• More on the intuitions later...
Components of LR Parsers

- Components of LR Parsers includes:
  - A stack,
  - A parsing table (with DFA states and actions),
  - An input buffer
  - A driver routine.
- Different LR parsers differ in nature of table only. Rest the same. The Power of LR parsers comes from accuracy of table.

- Two possible actions:
  - shift a terminal from input to stack
  - reduce a string $\alpha$ on top of the stack

- LR Parsers are also called shift-reduce parsers.
The Use of Stack in LR Parsers

• PDA and LL parsers expand a non-terminal symbol on the stack.
  – If the top of the stack is $A$, PDA and LL parsers pick a rule $A \rightarrow \omega$, and replace $A$ with $w$ in the stack (pop $A$ and push $\omega$).

• LR parsers reduces the symbols on top of the stack.
  – If the top of the stack is $\omega$, LR parsers pick a rule $A \rightarrow \omega$, and replace $\omega$ with $A$ in the stack (pop $\omega$ and push $A$).
  – The $\omega$ here is called handle. The key questions on when to reduce/shift and how to reduce are equivalent to determine whether the symbols on top of the stack constitute a valid handle.
The Use of Stack in LR Parsers

- In addition to symbols, the stack of LR parsers also include states traversed in the DFA.
  - These states are used to keep track of the rules that are (predicted to be) used for derivations.

- Stack contains configuration of the form \((s_0, X_1, s_1, X_2, \ldots, X_m, s_m)\)
  - \(X_i\): grammar symbol (terminal or non-terminal),
  - \(s_i\): state

- LR parser configuration:
  - \((\text{stack}, \text{input}) : (s_0, X_1, s_1, X_2, \ldots, X_m, s_m, t_i, t_{i+1}, \ldots, t_n)\)
  - \(t_j\): terminal symbols in input string.
LR(0) Parser
LR(0) Parser

• Bottom-up predictive parsing with:
  – \(L\): Left-to-right scan of the input.
  – \(R\): Rightmost derivation.
  – \(0\): Zero tokens of lookahead.

• Use the production-rule-tracking DFA, without any lookahead, to predict where handles are.
LR(0) Parsing Table

- LR(0) parsers are usually represented via two tables: an action table and a goto table.
- The action table maps each state to an action:
  - shift, which shifts the next terminal, and
  - reduce $A \rightarrow \omega$, which performs reduction $A \rightarrow \omega$.
- The goto table maps state/symbol pairs to a next state.
  - Used after a reduction, when a new non-terminal symbol is push on to stack.
An Example of LR(0) Parsing Table

- Consider the following grammar and parsing table:

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce E → T + E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>Reduce E → T ;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>Reduce T → (E)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>9</td>
<td>Reduce T → int</td>
<td></td>
</tr>
</tbody>
</table>

If the top of the stack is state-2, reduce using the specified rule. That is, pop T+E and push E.

If the top of the stack is state-8, and next input symbol is int, shift int to stack and push state-9 to stack.

Empty cells indicate parsing error.

After reduction, if the new symbol on stack is E and previous stack is 0, push state 1 to stack.
The LR(0) Algorithm

- Maintain a stack of in the format of ($s_0 X_1 s_1 X_2 ... X_m s_m$), which is initially in state 0 ($s_0$).

- While the stack is not empty:
  - Let state $s_i$ be the top state, and $t_i$ is the next input symbol.
  - If $\text{action}[s_i, t_i]$ is SJ (i.e., shift and goto state $j$, the “S” here means shift instead of state):
    - Push pair $(t_i, s_j)$ atop the stack.
  - If $\text{action}[s_i]$ is reduce $A \rightarrow \omega$:
    - Pop $|\omega|$ symbols and states from the top of the stack.
    - Let top-state $s_k$ be the state on top of the stack.
    - Push $(A, \text{goto}[s_k, A])$ atop the stack.
  - Otherwise, report an error.
An Example of LR(0) Parsing

- Considering using the previous parsing table to parse string “int + (int + int;);”

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce</td>
<td>E → T + E</td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>E → T;</td>
</tr>
<tr>
<td>5</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>Reduce</td>
<td>T → (E)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>9</td>
<td>Reduce</td>
<td>T → int</td>
</tr>
</tbody>
</table>

Initially in state 0

Stack | inputs | Action |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>int + (int + int;); $</td>
<td></td>
</tr>
</tbody>
</table>

CS5363
PL and Compilers
An Example of LR(0) Parsing cont.

- Considering using the previous parsing table to parse string “int + (int + int;);”

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s9$</td>
<td>$s8$</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce $E \rightarrow T + E$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$s5$</td>
<td>$s4$</td>
</tr>
<tr>
<td>4</td>
<td>Reduce $E \rightarrow T;$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$s9$</td>
<td>$s8$</td>
</tr>
<tr>
<td>6</td>
<td>Reduce $T \rightarrow (E)$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$s6$</td>
</tr>
<tr>
<td>8</td>
<td>$s9$</td>
<td>$s8$</td>
</tr>
<tr>
<td>9</td>
<td>Reduce $T \rightarrow int$</td>
<td></td>
</tr>
</tbody>
</table>

Stack | inputs | Action
--- | ------ | ---

$0$ | int + (int + int;); $ | Shift 9
$0$ int 9 + (int + int;); $
An Example of LR(0) Parsing cont.

- Considering using the previous parsing table to parse string “int + (int + int;);”

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>s9</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce E → T + E</td>
<td>s5 s4</td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>5</td>
<td>Reduce T → (E)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>7</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reduce T → int</td>
<td></td>
</tr>
</tbody>
</table>

Stack | inputs | Action
--- | --- | ---
$0$ | int + (int + int;); $ | Shift 9
$0$ int 9 | + (int + int;); $ | Reduce w. rule (4)
$0$ $T$ | $+$ (int + int;); $ | 

Action[9,*] = reduce T → int, which means pop “int 9” and push T.
An Example of LR(0) Parsing cont.

- Considering using the previous parsing table to parse string “int + (int + int;);”

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce E → T + E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>Reduce E → T;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>Reduce T → (E)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>9</td>
<td>Reduce T → int</td>
<td></td>
</tr>
</tbody>
</table>

Stack | inputs | Action |
$0$ | int + (int + int;); $ | Shift 9 |
$0$ | int 9 | + (int + int;); $ | Reduce w. rule (4); Goto 3 |
$0$ | T 3 | + (int + int;); $ |

\[ GOTO[0,T] = 3, \text{ which means push state 3.} \]
An Example of LR(0) Parsing cont.

- The rest of stacks:

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>s9</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce E → T + E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>Reduce E → T;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>Reduce T → (E)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>9</td>
<td>Reduce T → int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>inputs</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>int + (int + int);;</td>
<td>Shift 9</td>
</tr>
<tr>
<td>$0$ int 9</td>
<td>+ (int + int);</td>
<td>Redu 4; Go 3</td>
</tr>
<tr>
<td>$0$ T 3</td>
<td>+ (int + int);</td>
<td>Shift 5</td>
</tr>
<tr>
<td>$0$ T 3 + 5</td>
<td>(int + int);</td>
<td>Shift 8</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 int 9)</td>
<td>(int + int);</td>
<td>Shift 9</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3)</td>
<td>+ int);</td>
<td>Redu 4; Go 3</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3 + 5)</td>
<td>Int;</td>
<td>Shift 9</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3 + 5 int 9)</td>
<td>;); $</td>
<td>Redu 4; Go 3</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3 + 5 T 3)</td>
<td>;); $</td>
<td>Shift 4</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3 + 5 T 3 ; 4)</td>
<td>;); $</td>
<td>Redu 2; Go 2</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 T 3 + 5 E 2)</td>
<td>;); $</td>
<td>Redu 3; Go 7</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 E 7)</td>
<td>;); $</td>
<td>Shift 6</td>
</tr>
<tr>
<td>$0$ T 3 + 5 (8 E 7) 6</td>
<td>; $</td>
<td>Redu 5; Go 3</td>
</tr>
<tr>
<td>$0$ T 3 + 5 T 3</td>
<td>; $</td>
<td>Shift 4</td>
</tr>
<tr>
<td>$0$ T 3 + 5 T 3 ; 4</td>
<td>$</td>
<td>Redu 2; Go 2</td>
</tr>
<tr>
<td>$0$ T 3 + 5 E 2</td>
<td>$</td>
<td>Redu 3; Go 1</td>
</tr>
<tr>
<td>$0$ E 1</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
LR(0) Parsing Table Generation

• There are two steps to generate LR(0) parsing table:
  1) Generate the DFA (states and transitions) of LR(0).
  2) Generate the parsing table based on the DFA.
LR(0) DFA Construction

• What is a state in LR(0) DFA?
  – A state is a set of items.
  – An item is of the form \([A \rightarrow \alpha \cdot B \beta]\). If the parsing processing goes to the state with this item, it means, we
    • Expect to reduce by \(A \rightarrow \alpha B \beta\).
    • Have already see \(\alpha\) and is on stack. The “\(\cdot\)” determines how much has been seen.
    • Expects to see symbols that will be generated by \(B\).

• Steps for construction of the states:
  1) Start with an initial item with and take closure to construct first item to form the state.
  2) Determine what next item will be. Take its closure to determine the next state.
  3) Iterate previous step until no more transitions are possible.
LR(0) DFA Construction Details

- **Augmented grammar:**
  - If $s$ is the starting symbol grammar $G$,
  - Construct a new grammar $G'$ by
    - Add a new starting symbol $s'$ to $G$
    - Add a new production $s' \rightarrow s$ to $G$
  - $G'$ is called the augmented grammar of $G$. 
LR(0) DFA Construction Details cont.

1. The first item. Set initial item of state 0 to
   - Include only rule \( S' \rightarrow S \).
   - Add “·” to the beginning of the right hand (indicating no symbols has been seen yet).
   - That is, the initial item is \([S' \rightarrow \cdot S]\).

2. Add closure of \([S' \rightarrow \cdot S]\) to state \(0\).
   - If \(I\) is a set of items of grammar \(G\), the closure(\(I\)) is set of items constructed with the following operations:
     - if \([A \rightarrow \alpha \cdot B \beta]\) is in closure(\(I\)), add all items \([B \rightarrow \cdot \gamma]\) if there is a rule \(B \rightarrow \cdot \gamma\) and items do not exist already.
     - Apply until no more items can be added.
3. Use the following operations to construct a state:
   - If \( I \) is the set of items of an existing state \( j \), construct a new state \( k \) by
     - For each item of form \([A \rightarrow \alpha \cdot X \beta]\), add item \([A \rightarrow \alpha X \cdot \beta]\) to a new state \( k \). Note that \( X \) may be terminal or non-terminal.
     - Add closure of \([A \rightarrow \alpha X \cdot \beta]\) as well.
     - Add a transition between state \( j \) and state \( k \) under input \( X \).

4. Repeat until no new states can be added.
An Example of LR(0) DFA Construction

• Again, consider this grammar:

  (2) $E \rightarrow T;$
  (3) $E \rightarrow T + E$
  (4) $T \rightarrow \text{int}$
  (5) $T \rightarrow (E)$

• Extend it with a new start symbol $S$:

  (1) $S \rightarrow E$
  (2) $E \rightarrow T;$
  (3) $E \rightarrow T + E$
  (4) $T \rightarrow \text{int}$
  (5) $T \rightarrow (E)$
An Example of LR(0) DFA Construction cont.

- Start with initial item \([S \rightarrow \cdot E]\) of state \(_0^\cdot\):

\[
\begin{align*}
(1) & \quad S \rightarrow E \\
(2) & \quad E \rightarrow T; \\
(3) & \quad E \rightarrow T + E \\
(4) & \quad T \rightarrow \text{int} \\
(5) & \quad T \rightarrow (E)
\end{align*}
\]
An Example of LR(0) DFA Construction cont.

- Add the closure of item \([S' \rightarrow \cdot S]\) to state \(_0\):

  (1) \(S \rightarrow E\)
  (2) \(E \rightarrow T;\)
  (3) \(E \rightarrow T + E\)
  (4) \(T \rightarrow \text{int}\)
  (5) \(T \rightarrow (E)\)

- State \(_0\):
  \[
  \begin{align*}
  S & \rightarrow \cdot E \\
  E & \rightarrow \cdot T; \\
  E & \rightarrow \cdot T + E
  \end{align*}
  \]

- The the right hand of \(S \rightarrow \cdot E\), which is \(E\), and add all rules of \(E \rightarrow \omega\).
An Example of LR(0) DFA Construction cont.

- Add the closure of item \([s' \to \cdot s]\) to state \(_0\):

\begin{align*}
(1) & S \to E \\
(2) & E \to T; \\
(3) & E \to T + E \\
(4) & T \to \text{int} \\
(5) & T \to (E)
\end{align*}

\[
\text{State}_0: \\
S \to \cdot E \\
E \to \cdot T; \\
E \to \cdot T + E \\
T \to \cdot \text{int} \\
T \to \cdot (E)
\]

Since \(T\)'s productions all start with terminal symbols, there is no new rules can be added to the closure.

The the right hand of \(E \to \cdot T\), which is \(T\), and add all rules of \(T \to \omega\). Do the same with rule \(E \to \cdot T + E\), which adds the two same \(T \to \omega\) rules.
An Example of LR(0) DFA Construction cont.

- Take item $S \rightarrow \cdot E$ in state $\text{state}_0$ and add a new state $\text{state}_1$ on input $E$, with new item $S \rightarrow E \cdot$:

1. $S \rightarrow E$
2. $E \rightarrow T$
3. $E \rightarrow T + E$
4. $T \rightarrow \text{int}$
5. $T \rightarrow (E)$

State $\text{state}_0$:
- $S \rightarrow \cdot E$
- $E \rightarrow \cdot T$
- $E \rightarrow \cdot T + E$
- $T \rightarrow \cdot \text{int}$
- $T \rightarrow \cdot (E)$

State $\text{state}_1$:
- $S \rightarrow E \cdot$

The closure of $S \rightarrow \cdot E$ is just itself, so no other items in state $\text{state}_1$.

On reading $E$, we travel from $S \rightarrow \cdot E$ to $S \rightarrow E \cdot$. 
An Example of LR(0) DFA Construction cont.

- Take item $E \rightarrow \cdot T$; and $E \rightarrow \cdot T + E$ in state $0$ and add a new state $3$ (don’t worry about the index) on input $T$:

  - State $0$:
    - $S \rightarrow \cdot E$
    - $E \rightarrow \cdot T$
    - $E \rightarrow \cdot T + E$
    - $T \rightarrow \cdot \text{int}$
    - $T \rightarrow \cdot (E)$

  - State $1$:
    - $S \rightarrow E \cdot$

  - State $3$:
    - $E \rightarrow T \cdot$
    - $E \rightarrow T \cdot + E$

  On reading $T$, it travels from $E \rightarrow \cdot T$; to $E \rightarrow T \cdot$; or from $E \rightarrow \cdot T + E$ to $E \rightarrow T \cdot + E$. Note that we combine two rules together since their next symbol are both $T$.

  The closure of state $3$ has only two items since the symbols after $\cdot$ are terminals.
An Example of LR(0) DFA Construction cont.

- Take item $T \rightarrow \cdot (E)$ in state $0$ and add a new state $8$ on input $($.

On reading $($, it travels from $T \rightarrow \cdot (E)$ to $T \rightarrow (\cdot E)$.
An Example of LR(0) DFA Construction cont.

- Add the closure of $T \rightarrow \cdot (E)$ to state $\delta$:

  1. The the right hand of $T \rightarrow (\cdot E)$, which is $E$, and add all rules of $E \rightarrow \omega$
  2. The the right hand of $E \rightarrow \cdot T$, which is $T$, and add all rules of $T \rightarrow \omega$

  Do the same with rule $E \rightarrow \cdot T + E$, which adds the two same $T \rightarrow \omega$ rules.
An Example of LR(0) DFA Construction cont.

- Take item $T \rightarrow \cdot \text{int}$ in state $\text{state}_0$ and add a new state $\text{state}_9$ on input int:

  \[
  \begin{align*}
  \text{State}_0: \\
  S & \rightarrow \cdot E \\
  E & \rightarrow \cdot T; \\
  E & \rightarrow \cdot T + E \\
  T & \rightarrow \cdot \text{int} \\
  T & \rightarrow \cdot (E)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{State}_1: \\
  S & \rightarrow E \cdot \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{State}_3: \\
  E & \rightarrow T \cdot; \\
  E & \rightarrow T \cdot + E \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{State}_8: \\
  T & \rightarrow (\cdot E) \\
  E & \rightarrow \cdot T; \\
  E & \rightarrow \cdot T + E \\
  T & \rightarrow \cdot \text{int} \\
  T & \rightarrow \cdot (E)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{State}_9: \\
  T & \rightarrow \cdot \text{int}.
  \end{align*}
  \]

On reading int, it travels from $T \rightarrow \cdot \text{int}$ to $T \rightarrow \cdot \text{int}$.
Note that state $\text{state}_9$ does not transit to any other states, as there are no symbols after "\cdot".
An Example of LR(0) DFA Construction cont.

- The DFA with all states:

State₀:
\[
\begin{align*}
S & \rightarrow \cdot E \\
E & \rightarrow \cdot T; \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₁:
\[
\begin{align*}
S & \rightarrow E
\end{align*}
\]

State₂:
\[
\begin{align*}
E & \rightarrow T + E
\end{align*}
\]

State₃:
\[
\begin{align*}
E & \rightarrow \cdot T; \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₄:
\[
\begin{align*}
E & \rightarrow \cdot T;
\end{align*}
\]

State₅:
\[
\begin{align*}
E & \rightarrow T + \cdot E \\
E & \rightarrow \cdot T; \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₆:
\[
\begin{align*}
T & \rightarrow \cdot (E) \\
E & \rightarrow T + \cdot E \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₇:
\[
\begin{align*}
T & \rightarrow \cdot (E) \\
E & \rightarrow T + \cdot E \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₈:
\[
\begin{align*}
T & \rightarrow \cdot (E) \\
E & \rightarrow T + \cdot E \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State₉:
\[
\begin{align*}
T & \rightarrow \cdot \text{int}
\end{align*}
\]
From LR(0) DFA to Parsing Table

- For each state X:
  - If there is an item $A \rightarrow \omega \cdot$, set $\text{action}[X, \ast] = \text{reduce } A \rightarrow \omega$.
  - If there is the special production $S' \rightarrow S \cdot$, where $S'$ is the new start symbol of $G'$ and $S$ is old start symbol of $G$, set $\text{action}[X, \ast] = \text{accept}$.
  - If there is a transition out of X on terminal symbol $t$ to state Y, set $\text{action}[X, t] = \text{shift } Y$.
  - If there is a transition out of X on non-terminal symbol $N$ to state Y, set $\text{goto}[X, N] = \text{state } Y$.
- Set all other actions to error.
- If any table entry contains two or more actions, the grammar is not LR(0).
An Example of LR(0) Construction

- Consider part of the previous DFA:

  The DFA goes from state 0 to state 9 on “int”. Therefore, shift9 (s9) in action[0, int].
An Example of LR(0) Construction cont.

- Consider part of the previous DFA:

  The DFA goes from state 0 to state 3 on “T”. Therefore, 3 in Goto[0, T].

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>s9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Example of LR(0) Construction cont.

- Consider part of the previous DFA:

State$_1$: 
\[ S \rightarrow E. \]

State$_0$: 
\[
\begin{align*}
S & \rightarrow \cdot E \\
E & \rightarrow \cdot T; \\
E & \rightarrow \cdot T + E \\
T & \rightarrow \cdot \text{int} \\
T & \rightarrow \cdot (E)
\end{align*}
\]

State$_9$: 
\[
\begin{align*}
T & \rightarrow \text{int} \\
E & \rightarrow T \cdot ; \\
E & \rightarrow T \cdot + E
\end{align*}
\]

State$_3$: 
\[
\begin{align*}
E & \rightarrow T \cdot ; \\
E & \rightarrow T \cdot + E
\end{align*}
\]

State$_4$: 
\[
\begin{align*}
E & \rightarrow T \cdot ;
\end{align*}
\]

State$_4$ has only one item, whose “·” is at then end. Therefore, Action[4,*]=reduce with \( E \rightarrow T; \).
An Example of LR(0) Construction cont.

• Consider part of the previous DFA:

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>s9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Reduce E → T;</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

State 1 has only the extended new rule $S \rightarrow E\cdot$. Therefore, action[1, ] = Accept.
An Example of LR(0) Construction cont.

- The final parsing table for the previous DFA:

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>1</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reduce $E \rightarrow T + E$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>4</td>
<td>Reduce $E \rightarrow T;$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>6</td>
<td>Reduce $T \rightarrow (E)$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td>s9</td>
<td>s8</td>
</tr>
<tr>
<td>9</td>
<td>Reduce $T \rightarrow \text{int}$</td>
<td></td>
</tr>
</tbody>
</table>
The Limitation of LR(0)

- A Non-LR(0) grammar,
  
  
  (1) $S \rightarrow E$
  (2) $E \rightarrow T$
  (3) $E \rightarrow T + E$
  (4) $T \rightarrow \text{int}$
  (5) $T \rightarrow (E)$

- If a grammar is not LR(0), there may be shift/reduce or reduce/reduce conflicts.
  
  - Unlike more powerful LR(1) and LALR parsers, these conflicts cannot be simply resolved by always shift or always reduce with the first rule.
Why LR(0) is Weak

- LR(0) only accepts languages where the handle can be found with no right context.
  - Our shift/reduce parser only looks to the left of the handle, not to the right.

- How do we exploit the tokens after a possible handle to determine what to do?
SLR(1) Parser
Resolving Shift/Reduce Conflicts in LR(0)

- A shift/reduce conflict in a state of LR(0) looks like this:

\[
\begin{align*}
\text{A} & \rightarrow \omega . \\
\text{B} & \rightarrow \alpha . \beta
\end{align*}
\]

- Can we use lookahead to predict whether we should shift or reduce based on next input symbol?
SLR(1)

- **Simple LR(1)**
  - We will learn the no-so-simple LR(1) later.
- **Minor modification to LR(0) automaton that uses lookahead to avoid shift/reduce conflicts.**
- **Idea:** Only choose to reduce with $A \rightarrow \omega$ if the next token $t$ is in FOLLOW($A$). Otherwise, shift.
- **Automaton identical to LR(0) automaton; only change is when we choose to reduce.**
From LR(0) DFA to SLR(1) Parsing Table

- For each state X:
  - If there is a production \( A \rightarrow \omega \cdot \), for every terminal symbol \( t, t \in \text{FOLLOW}(A) \), set \( \text{action}[X, t] = \text{reduce } A \rightarrow \omega \).
  - If there is the special production \( S' \rightarrow S \cdot \), where \( S' \) is the new start symbol of \( G' \) and \( S \) is old start symbol of \( G \), set \( \text{action}[X, $] = \text{accept} \).
  - If there is a transition out of \( X \) on terminal symbol \( t \) to state \( Y \), set \( \text{action}[X, t] = \text{shift } Y \).
  - If there is a transition out of \( X \) on non-terminal symbol \( N \) to state \( Y \), set \( \text{goto}[X, N] = \text{state } Y \).

- Set all other actions to error.

- If any table entry contains two or more actions, the grammar is not LR(0).
An Example of SLR(1)

- Consider the following Grammar with its LR(0) states:

(1)  $S \rightarrow E$
(2)  $E \rightarrow T$
(3)  $E \rightarrow T + E$
(4)  $T \rightarrow \text{int}$
(5)  $T \rightarrow (E)$

Only reduce with $E \rightarrow T$ if next input symbol is in Follow (E).
An Example of SLR(1) cont.

- The FIRST and FOLLOW sets of this grammar are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>int, (</td>
<td>$$$</td>
</tr>
<tr>
<td>$E$</td>
<td>int, (</td>
<td>), $$</td>
</tr>
<tr>
<td>$T$</td>
<td>int, (</td>
<td>+, ), $$</td>
</tr>
</tbody>
</table>
An Example of SLR(1) cont.

- The SLR(1) parsing table is then:

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>s8</td>
<td>s7</td>
</tr>
<tr>
<td>2</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>s8</td>
<td>s7</td>
</tr>
<tr>
<td>5</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>s5</td>
</tr>
<tr>
<td>7</td>
<td>s8</td>
<td>s7</td>
</tr>
<tr>
<td>8</td>
<td>r4</td>
<td>r4</td>
</tr>
</tbody>
</table>

In state $3$, shift if next symbol is "+", reduce with rule (2) if next symbol is ")" or "+".
The Limitation of SLR(1)

- Consider the follow grammar and its states:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$S \rightarrow E$</td>
</tr>
<tr>
<td>(2)</td>
<td>$E \rightarrow L = R$</td>
</tr>
<tr>
<td>(3)</td>
<td>$E \rightarrow R$</td>
</tr>
<tr>
<td>(4)</td>
<td>$L \rightarrow \text{id}$</td>
</tr>
<tr>
<td>(5)</td>
<td>$L \rightarrow *R$</td>
</tr>
<tr>
<td>(6)</td>
<td>$R \rightarrow L$</td>
</tr>
</tbody>
</table>

$S \rightarrow \cdot E$
$E \rightarrow \cdot L=R$
$E \rightarrow \cdot R$
$L \rightarrow \cdot \text{id}$
$L \rightarrow \cdot *R$
$R \rightarrow \cdot L$

$E \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

$E \rightarrow L \cdot = R$, tells us to shift on seeing $=$,
$R \rightarrow L \cdot$, tells us to reduce on FOLLOW(R).

However, $\text{=} \in \text{FOLLOW}(R)$, we have a shift/reduce conflict!
Why is SLR(1) Weak?

• Again, consider this grammar,

\[
\begin{align*}
(1) & \quad S \rightarrow E \\
(2) & \quad E \rightarrow L = R \\
(3) & \quad E \rightarrow R \\
(4) & \quad L \rightarrow id \\
(5) & \quad L \rightarrow *R \\
(6) & \quad R \rightarrow L
\end{align*}
\]

State_0:

\[
\begin{align*}
S & \rightarrow \cdot E \\
E & \rightarrow \cdot L=R; \\
E & \rightarrow \cdot R \\
L & \rightarrow \cdot id \\
L & \rightarrow \cdot *R \\
R & \rightarrow \cdot L
\end{align*}
\]

State_1:

\[
\begin{align*}
E & \rightarrow L \cdot =R \\
R & \rightarrow L.
\end{align*}
\]

But we we get to $R \rightarrow L \cdot$ in state1, $=$ is not a possible symbols after $R$ given the possible derivations.

The possible derivations are

\[
\begin{align*}
S & \rightarrow E \rightarrow R \rightarrow L \rightarrow $, only $ can follow $R$ in this case.
\]

CS5363
PL and Compilers
Why is SLR(1) Weak?

• SLR(1) utilize minimal context to predict whether to shift or reduce.

• FOLLOW(A) means “what could follow A somewhere in the grammar?,” even if in a particular state A couldn't possibly have that symbol after it.
Acknowledgement

• This lectures is partially based on the Compiler slides of Keith Schwarz and Dr. Raju Pandey.