Bottom-Up Parsing

- Attempts to traverse a parse tree bottom up (**post-order traversal**)
- Reduces a sequence of tokens to the start symbol
- At each reduction step, the RHS of a production is replaced with LHS
- A reduction step corresponds to the **reverse of a rightmost derivation**

Example: given the following grammar

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

A rightmost derivation for \( \text{id} + \text{id} * \text{id} \) is shown below:

\[
\begin{align*}
E & \Rightarrow_{\text{rm}} E + T \Rightarrow_{\text{rm}} E + T * F \Rightarrow_{\text{rm}} E + T * \text{id} \\
& \Rightarrow_{\text{rm}} E + F * \text{id} \Rightarrow_{\text{rm}} E + \text{id} * \text{id} \Rightarrow_{\text{rm}} T + \text{id} * \text{id} \\
& \Rightarrow_{\text{rm}} F + \text{id} * \text{id} \Rightarrow_{\text{rm}} \text{id} + \text{id} * \text{id}
\end{align*}
\]
Handles

- If $S \Rightarrow^+_rm \alpha$ then $\alpha$ is called a **right sentential form**
- A **handle** of a right sentential form is:
  - A substring $\beta$ that matches the RHS of a production $A \rightarrow \beta$
  - The reduction of $\beta$ to $A$ is a step along the reverse of a rightmost derivation
- If $S \Rightarrow^+_rm \gamma Aw \Rightarrow^+_rm \gamma \beta w$, where $w$ is a sequence of tokens then
  - The substring $\beta$ of $\gamma \beta w$ and the production $A \rightarrow \beta$ make the handle
- Consider the reduction of $\text{id} + \text{id} \ast \text{id}$ to the start symbol $E$

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Production</th>
<th>Sentential Form</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{id} + \text{id} \ast \text{id}$</td>
<td>$F \rightarrow \text{id}$</td>
<td>$E + T \ast \text{id}$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$E + \text{id} \ast \text{id}$</td>
<td>$T \rightarrow F$</td>
<td>$E + T \ast F$</td>
<td>$T \rightarrow T \ast F$</td>
</tr>
<tr>
<td>$T + \text{id} \ast \text{id}$</td>
<td>$E \rightarrow T$</td>
<td>$E + T$</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>$E + \text{id} \ast \text{id}$</td>
<td>$F \rightarrow \text{id}$</td>
<td>$E$</td>
<td></td>
</tr>
<tr>
<td>$E + F \ast \text{id}$</td>
<td>$T \rightarrow F$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack Implementation of a Bottom-Up Parser

- A bottom-up parser uses an explicit stack in its implementation
- The main actions are shift and reduce
  - A bottom-up parser is also known as a shift-reduce parser
- Four operations are defined: shift, reduce, accept, and error
  - Shift: parser shifts the next token on the parser stack
  - Reduce: parser reduces the RHS of a production to its LHS
    - The handle always appears on top of the stack
  - Accept: parser announces a successful completion of parsing
  - Error: parser discovers that a syntax error has occurred
- The parser operates by:
  - Shifting tokens onto the stack
  - When a handle $\beta$ is on top of stack, parser reduces $\beta$ to LHS of production
  - Parsing continues until an error is detected or input is reduced to start symbol
Example on Bottom-Up Parsing

- Consider the parsing of the input string `id + id * id`

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id $</td>
<td>shift</td>
<td>$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+ id * id $</td>
<td>reduce F → id</td>
<td>$</td>
<td>reduce F → id</td>
</tr>
<tr>
<td>$F</td>
<td>+ id * id $</td>
<td>reduce T → F</td>
<td>$</td>
<td>reduce T → F</td>
</tr>
<tr>
<td>$T</td>
<td>+ id * id $</td>
<td>reduce E → T</td>
<td>$</td>
<td>reduce E → T</td>
</tr>
<tr>
<td>$E</td>
<td>+ id * id $</td>
<td>shift</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E +</td>
<td>id * id $</td>
<td>shift</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + id</td>
<td>* id $</td>
<td>reduce F → id</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + F</td>
<td>* id $</td>
<td>reduce T → F</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + T</td>
<td>* id $</td>
<td>shift</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + T *</td>
<td>id $</td>
<td>shift</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + T *id</td>
<td>$</td>
<td>reduce F → id</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + T *F</td>
<td>$</td>
<td>reduce T → T * F</td>
<td>$</td>
<td>accept</td>
</tr>
<tr>
<td>$E + T</td>
<td>$</td>
<td>reduce E → E + T</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

We use $ to mark the bottom of the stack as well as the end of input.
LR Parsing

- To have an operational shift-reduce parser, we must determine:
  - Whether a handle appears on top of the stack
  - The reducing production to be used
  - The choice of actions to be made at each parsing step

- LR parsing provides a solution to the above problems
  - Is a general and efficient method of shift-reduce parsing
  - Is used in a number of automatic parser generators

- The LR($k$) parsing technique was introduced by Knuth in 1965
  - L is for Left-to-right scanning of input
  - R corresponds to a Rightmost derivation done in reverse
  - $k$ is the number of lookahead symbols used to make parsing decisions
LR Parsing – cont'd

- LR parsing is attractive for a number of reasons …
  - Is the most general deterministic parsing method known
  - Can recognize virtually all programming language constructs
  - Can be implemented very efficiently
  - The class of LR grammars is a proper superset of the LL grammars
  - Can detect a syntax error as soon as an erroneous token is encountered
  - A LR parser can be generated by a parser generating tool

- Four LR parsing techniques will be considered
  - LR(0) : LR parsing with no lookahead token to make parsing decisions
  - SLR(1) : Simple LR, with one token of lookahead
  - LR(1) : Canonical LR, with one token of lookahead
  - LALR(1) : Lookahead LR, with one token of lookahead

- LALR(1) is the preferable technique used by parser generators
LR Parsers

- An LR parser consists of …
  - Driver program
    - Same driver is used for all LR parsers
  - Parsing stack
    - Contains state information, where $s_i$ is state $i$
    - States are obtained from grammar analysis
  - Parsing table, which has two parts
    - Action section: specifies the parser actions
    - Goto section: specifies the successor states

- The parser driver receives tokens from the scanner one at a time
- Parser uses top state and current token to lookup parsing table
- Different LR analysis techniques produce different tables
LR Parsing Table Example

- Consider the following grammar $G_1$ ...
  1: $E \rightarrow E + T$
  2: $E \rightarrow T$
  3: $T \rightarrow \text{ID}$
  4: $T \rightarrow (E)$

- The following parsing table is obtained after grammar analysis

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S1 S2</td>
<td>G4 G3</td>
</tr>
<tr>
<td>1</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>2</td>
<td>S1 S2</td>
<td>G6 G3</td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>S1 S2</td>
<td>G7</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S8</td>
</tr>
<tr>
<td>7</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>8</td>
<td>R4</td>
<td>R4</td>
</tr>
</tbody>
</table>

Entries are labeled with …
- $S_n$: Shift token and goto state $n$  
  (call scanner for next token)
- $R_n$: Reduce using production $n$
- $G_n$: Goto state $n$ (after reduce)
- A: Accept parse  
  (terminate successfully)
- blank: Syntax error
**LR Parsing Example**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ id + ( id + id )$</td>
<td>$ S1 $</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>$ id $</td>
<td>$ E + ( id + id )$</td>
<td>R3, G3</td>
</tr>
<tr>
<td>0 3</td>
<td>$ T $</td>
<td>$ E + ( id + id )$</td>
<td>R2, G4</td>
</tr>
<tr>
<td>0 4</td>
<td>$ E $</td>
<td>$ id + ( id + id )$</td>
<td>S5</td>
</tr>
<tr>
<td>0 4 5</td>
<td>$ E + $</td>
<td>$ ( id + id )$</td>
<td>S2</td>
</tr>
<tr>
<td>0 4 5 2</td>
<td>$ E + ( $</td>
<td>$ id + id )$</td>
<td>S1</td>
</tr>
<tr>
<td>0 4 5 2 1</td>
<td>$ E + ( id $</td>
<td>$ + id )$</td>
<td>R3, G3</td>
</tr>
<tr>
<td>0 4 5 2 3</td>
<td>$ E + ( T $</td>
<td>$ + id )$</td>
<td>R2, G6</td>
</tr>
<tr>
<td>0 4 5 2 6</td>
<td>$ E + ( E $</td>
<td>$ + id )$</td>
<td>S5</td>
</tr>
<tr>
<td>0 4 5 2 6 5</td>
<td>$ E + ( E + $</td>
<td>$ id )$</td>
<td>S1</td>
</tr>
<tr>
<td>0 4 5 2 6 5 1</td>
<td>$ E + ( E + id $</td>
<td>$ )$</td>
<td>R3, G7</td>
</tr>
<tr>
<td>0 4 5 2 6 5 7</td>
<td>$ E + ( E + T $</td>
<td>$ )$</td>
<td>R1, G6</td>
</tr>
<tr>
<td>0 4 5 2 6 8</td>
<td>$ E + ( $</td>
<td>$ E )$</td>
<td>S8</td>
</tr>
<tr>
<td>0 4 5 7</td>
<td>$ E + T $</td>
<td>$ $</td>
<td>R4, G7</td>
</tr>
<tr>
<td>0 4</td>
<td>$ E $</td>
<td>$ $</td>
<td>A</td>
</tr>
</tbody>
</table>

1: $ E \rightarrow E + T $
2: $ E \rightarrow T $
3: $ T \rightarrow id $
4: $ T \rightarrow ( E ) $

Grammar symbols do not appear on the parsing stack. They are shown here for clarity.
LR Parser Driver

- Let $s$ be the parser stack top state and $t$ be the current input token
- If $\text{action}[s,t] = \text{shift} \ n$ then
  - Push state $n$ on the stack
  - Call scanner to obtain next token
- If $\text{action}[s,t] = \text{reduce} \ A \rightarrow X_1 X_2 \ldots X_m$ then
  - Pop the top $m$ states off the stack
  - Let $s'$ be the state now on top of the stack
  - Push $\text{goto}[s', A]$ on the stack (using the goto section of the parsing table)
- If $\text{action}[s,t] = \text{accept}$ then return
- If $\text{action}[s,t] = \text{error}$ then call error handling routine
- All LR parsers behave the same way
  - The difference depends on how the parsing table is computed from a CFG
LR(0) Parse Generation – Items and States

- LR(0) grammars can be parsed looking only at the stack
- Making shift/reduce decisions without any lookahead token
- Based on the idea of an item or a configuration
- An LR(0) item consists of a production and a dot

\[ A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_n \]

- The dot symbol \( \bullet \) may appear anywhere on the right-hand side
  - Marks how much of a production has already been seen
  - \( X_1 \ldots X_i \) appear on top of the stack
  - \( X_{i+1} \ldots X_n \) are still expected to appear

- An LR(0) state is a set of LR(0) items
  - It is the set of all items that apply at a given point in parse
LR(0) Parser Generation – Initial State

- Consider the following grammar \( G_1 \):
  1: \( E \rightarrow E + T \)
  2: \( E \rightarrow T \)
  3: \( T \rightarrow \text{ID} \)
  4: \( T \rightarrow (E) \)

- For LR parsing, grammars are augmented with a . . .
  - New start symbol \( S \), and a
  - New start production 0: \( S \rightarrow E \$ \)

- The input should be reduced to \( E \) followed by $ 
  - We indicate this by the item: \( S \rightarrow \bullet E \$ \)

- The initial state (numbered 0) will have the item: \( S \rightarrow \bullet E \$ \)
- An LR parser will start in state 0
- State 0 is initially pushed on top of parser stack
Identifying the Initial State

- Since the dot appears before \( E \), an \( E \) is expected
  - There are two productions of \( E \): \( E \rightarrow E + T \) and \( E \rightarrow T \)
  - Either \( E + T \) or \( T \) is expected
  - The items: \( E \rightarrow \bullet E + T \) and \( E \rightarrow \bullet T \) are added to the initial state

- Since \( T \) can be expected and there are two productions for \( T \)
  - Either \( \text{ID} \) or \( (E) \) can be expected
  - The items: \( T \rightarrow \bullet \text{ID} \) and \( T \rightarrow \bullet (E) \) are added to the initial state

- The initial state (0) is identified by the following set of items:

\[
\begin{align*}
S & \rightarrow \bullet E \$ \\
E & \rightarrow \bullet E + T \\
E & \rightarrow \bullet T \\
T & \rightarrow \bullet \text{ID} \\
T & \rightarrow \bullet (E) \quad \circled{0}
\end{align*}
\]
Shift Actions

- In state 0, we can shift either an **ID** or a left parenthesis
  - If we shift an **ID**, we shift the dot past the **ID**
  - We obtain a new item \( T \rightarrow \text{ID} \bullet \) and a new state (state 1)
  - If we shift a left parenthesis, we obtain \( T \rightarrow ( \bullet E ) \)
  - Since the dot appears before \( E \), an \( E \) is expected
  - We add the items \( E \rightarrow \bullet E + T \) and \( E \rightarrow \bullet T \)
  - Since the dot appears before \( T \), we add \( T \rightarrow \bullet \text{ID} \) and \( T \rightarrow \bullet ( E ) \)
  - The new set of items forms a new state (state 2)

- In State 2, we can also shift an **ID** or a left parenthesis as shown
Reduce and Goto Actions

- In state 1, the dot appears at the end of item $T \rightarrow \text{ID} \bullet$
  - This means that \text{ID} appears on top of stack and can be reduced to $T$
  - When \textbullet appears at end of an item, the parser can perform a reduce action

- If \text{ID} is reduced to $T$, what is the next state of the parser?
  - \text{ID} is popped from the stack; Previous state appears on top of stack
  - $T$ is pushed on the stack
  - A new item $E \rightarrow T \bullet$ and a new state (state 3) are obtained
  - If top of stack is state 0 and we push a $T$, we go to state 3
  - Similarly, if top of stack is state 2 and we push a $T$, we go also to state 3
**DFA of LR(0) States**

- We complete the state diagram to obtain the DFA of LR(0) states
- In state 4, if next token is $\$, the parser **accepts** (successful parse)

---

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ID</td>
<td>$S \rightarrow E \cdot E \cdot$</td>
</tr>
<tr>
<td>1</td>
<td>ID</td>
<td>$E \rightarrow E \cdot$</td>
</tr>
<tr>
<td>2</td>
<td>ID</td>
<td>$T \rightarrow (E)$</td>
</tr>
<tr>
<td>3</td>
<td>$T \rightarrow E \cdot$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$+$</td>
<td>$S \rightarrow E \cdot E \cdot$</td>
</tr>
<tr>
<td>5</td>
<td>$+$</td>
<td>$E \rightarrow E \cdot T \cdot$</td>
</tr>
<tr>
<td>6</td>
<td>$+$</td>
<td>$T \rightarrow (E \cdot$</td>
</tr>
<tr>
<td>7</td>
<td>$T \rightarrow E \cdot$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$)$</td>
<td>$T \rightarrow (E \cdot$</td>
</tr>
</tbody>
</table>
LR(0) Parsing Table

- The LR(0) parsing table is obtained from the LR(0) state diagram
- The rows of the parsing table correspond to the LR(0) states
- The columns correspond to tokens and non-terminals
- For each state transition $i \rightarrow j$ caused by a token $x$ …
  - Put **Shift** $j$ at position $[i, x]$ of the table
- For each transition $i \rightarrow j$ caused by a nonterminal $A$ …
  - Put **Goto** $j$ at position $[i, A]$ of the table
- For each state containing an item $A \rightarrow \alpha \bullet$ of rule $n$ …
  - Put **Reduce** $n$ at position $[i, y]$ for every token $y$
- For each transition $i \rightarrow Accept$ …
  - Put **Accept** at position $[i, \$]$ of the table
LR(0) Parsing Table – cont'd

- The LR(0) table of grammar $G1$ is shown below
  - For a shift, the token to be shifted determines the next state
  - For a reduce, the state on top of stack specifies the production to be used

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>$E$</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>G4</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>G3</td>
</tr>
<tr>
<td>1</td>
<td>R3</td>
<td>G4</td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>G3</td>
</tr>
<tr>
<td>2</td>
<td>S1</td>
<td>G6</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>G3</td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S1</td>
<td>G7</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S8</td>
</tr>
<tr>
<td>7</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>R4</td>
<td></td>
</tr>
</tbody>
</table>

Entries are labeled with …

- **Sn**: Shift token and goto state $n$ (call scanner for next token)
- **Rn**: Reduce using production $n$
- **Gn**: Goto state $n$ (after reduce)
- **A**: Accept parse (terminate successfully)
- **blank**: Syntax error
Limitations of the LR(0) Parsing Method

- Consider grammar $G_2$ for matched parentheses
  
  $0: S' \rightarrow S \, \$ 
  $1: S \rightarrow (\, S\,) \, S 
  $2: S \rightarrow \varepsilon$

- The LR(0) DFA of grammar $G_2$ is shown below

- In states: 0, 2, and 4, parser can shift ( and reduce $\varepsilon$ to $S$
Conflicts

- In state 0 parser encounters a conflict ...
  - It can shift state 2 on stack when next token is (
  - It can reduce production 2: $S \rightarrow \varepsilon$
  - This is a called a shift-reduce conflict
  - This conflict also appears in states 2 and 4

- Two kinds of conflicts may arise
  - **Shift-reduce** and **reduce-reduce**
    - **Shift-reduce** conflict
      Parser can shift and can reduce
    - **Reduce-reduce** conflict
      Two (or more) productions can be reduced
LR(0) Grammars

- The shift-reduce conflict in state 0 indicates that $G_2$ is not LR(0)
- A grammar is LR(0) if and only if each state is either …
  - A **shift state**, containing only shift items
  - A **reduce state**, containing only a single reduce item
- If a state contains $A \rightarrow \alpha \bullet x \gamma$ then it cannot contain $B \rightarrow \beta \bullet$
  - Otherwise, parser can shift $x$ and reduce $B \rightarrow \beta \bullet$ (shift-reduce conflict)
- If a state contains $A \rightarrow \alpha \bullet$ then it cannot contain $B \rightarrow \beta \bullet$
  - Otherwise, parser can reduce $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ (reduce-reduce conflict)
- LR(0) lacks the power to parse programming language grammars
  - Because they do not use the lookahead token in making parsing decisions
SLR(1) Parsing

- SLR(1), or simple LR(1), improves LR(0) by ...
  - Making use of the lookahead token to eliminate conflicts

- SLR(1) works as follows ...
  - It uses the same DFA obtained by the LR(0) parsing method
  - It puts reduce actions only where indicated by the FOLLOW set

- To reduce $\alpha$ to $A$ in $A \rightarrow \alpha \bullet$ we must ensure that ...
  - Next token may follow $A$ (belongs to FOLLOW($A$))

- We should not reduce $A \rightarrow \alpha \bullet$ when next token $\notin$ FOLLOW($A$)

- In grammar $G2$ ...
  - 0: $S' \rightarrow S \, \$$  
    1: $S \rightarrow ( \, S \, ) \, S$  
    2: $S \rightarrow \varepsilon$
  - FOLLOW($S$) = \{$\$$, $\);$\}
  - Productions 1 and 2 are reduced when next token is $\$$ or $\,$ only

---

**LR Parsing Techniques – 22**

**Compiler Design – © Muhammed Mudawwar**
SLR(1) Parsing Table

- The SLR(1) parsing table of grammar $G_2$ is shown below

- The shift-reduce conflicts are now eliminated
  - The R2 action is removed from [0, ( ), [2, ( ), and [4, ( )]
  - Because ( does not follow $S$
  - $S_2$ remains under [0, ( ), [2, ( ), and [4, ( )]
  - R1 action is also removed from [5, ( )]

- Grammar $G_2$ is SLR(1)
  - No conflicts in parsing table
  - R1 and R2 for ) and $\$ only
  - Follow set indicates when to reduce

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S2</td>
<td>R2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>R2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S2</td>
<td>R2</td>
</tr>
<tr>
<td>5</td>
<td>R1</td>
<td>R1</td>
</tr>
</tbody>
</table>
SLR(1) Grammars

- SLR(1) parsing increases the power of LR(0) significantly
  - Lookahead token is used to make parsing decisions
  - Reduce action is applied more selectively according to FOLLOW set

- A grammar is SLR(1) if two conditions are met in every state …
  - If \( A \rightarrow \alpha \cdot x \gamma \) and \( B \rightarrow \beta \cdot \) then token \( x \notin \text{FOLLOW}(B) \)
  - If \( A \rightarrow \alpha \cdot \) and \( B \rightarrow \beta \cdot \) then \( \text{FOLLOW}(A) \cap \text{FOLLOW}(B) = \emptyset \)

- Violation of first condition results in **shift-reduce conflict**
  - \( A \rightarrow \alpha \cdot x \gamma \) and \( B \rightarrow \beta \cdot \) and \( x \in \text{FOLLOW}(B) \) then …
  - Parser can shift \( x \) and reduce \( B \rightarrow \beta \)

- Violation of second condition results in **reduce-reduce conflict**
  - \( A \rightarrow \alpha \cdot \) and \( B \rightarrow \beta \cdot \) and \( x \in \text{FOLLOW}(A) \cap \text{FOLLOW}(B) \)
  - Parser can reduce \( A \rightarrow \alpha \) and \( B \rightarrow \beta \)

- SLR(1) grammars are a superset of LR(0) grammars
Limits of the SLR(1) Parsing Method

- Consider the following grammar $G3$ ...
  
  0: $S' \rightarrow S \ \$ \quad 1: S \rightarrow id \quad 2: S \rightarrow V := E \quad 3: V \rightarrow id \quad 4: E \rightarrow V \quad 5: E \rightarrow n$

- The initial state consists of 4 items as shown below
  
  * When $id$ is shifted in state 0, we obtain 2 items: $S \rightarrow id \bullet$ and $V \rightarrow id \bullet$

- $FOLLOW(S) = \{\$\}$ and $FOLLOW(V) = \{:=, \$\}$

- **Reduce-reduce conflict** in state 1 when lookahead token is $\$$
  
  * Therefore, grammar $G3$ is **not** SLR(1)
  
  * The reduce-reduce conflict is caused by the weakness of SLR(1) method
  
  * $V \rightarrow id$ should be reduced only when lookahead token is $:= (but not $)$
General LR(1) Parsing – Items and States

- Even more powerful than SLR(1) is the LR(1) parsing method
- LR(1) generalizes LR(0) by including a lookahead token in items
- An LR(1) item consists of …
  - Grammar production rule
  - Right-hand position represented by the dot, and
  - Lookahead token

$$A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_n, \ l$$ where \( l \) is a lookahead token

- The \( \bullet \) represents how much of the right-hand side has been seen
  - \( X_1 \ldots X_i \) appear on top of the stack
  - \( X_{i+1} \ldots X_n \) are expected to appear
- The lookahead token \( l \) is expected after \( X_1 \ldots X_n \) appear on stack
- An LR(1) state is a set of LR(1) items
LR(1) Parser Generation – Initial State

- Consider again grammar $G3$ ...
  
  0: $S' \to S$ $\$  
  1: $S \to id$  
  2: $S \to V := E$  
  3: $V \to id$  
  4: $E \to V$  
  5: $E \to n$

- The initial state contains the LR(1) item: $S' \to \bullet S$, $\$
  
  - $S' \to \bullet S$, $\$ means that $S$ is expected and to be followed by $\$

- The closure of $(S' \to \bullet S$, $\$)$ produces the initial state items
  
  - Since the dot appears before $S$, an $S$ is expected
  - There are two productions of $S$: $S \to id$ and $S \to V := E$
  - The LR(1) items $(S \to \bullet id$, $\$)$ and $(S \to \bullet V := E$, $\$)$ are obtained
    - The lookahead token is $\$$(end-of-file token)$
  - Since the $\bullet$ appears before $V$ in $(S \to \bullet V := E$, $\$)$, a $V$ is expected
  - The LR(1) item $(V \to \bullet id$, $\$)$ is obtained
    - The lookahead token is $\$ because it appears after $V$ in $(S \to \bullet V := E$, $\$)$
Shift Action

- The initial state (state 0) consists of 4 items
- In state 0, we can shift an **id**
  - The token **id** can be shifted in two items
  - When shifting **id**, we shift the dot past the **id**
  - We obtain \(S \rightarrow \text{id }\cdot, \$
  \) and \(V \rightarrow \text{id }\cdot, :=\)
  - The two LR(1) items form a new state (state 1)
  - The two items are **reduce items**
  - No additional item can be added to state 1

\[
\begin{align*}
S' & \rightarrow \cdot S, \\
S & \rightarrow \cdot \text{id }, \\
S & \rightarrow \cdot V := E, \\
V & \rightarrow \cdot \text{id } , := \\
\text{id} & \rightarrow \cdot \\
S & \rightarrow \text{id }\cdot, \\
V & \rightarrow \text{id }\cdot, :=
\end{align*}
\]
Reduce and Goto Actions

- In state 1, • appears at end of (\( S \rightarrow \text{id} \cdot, \$ \)) and (\( V \rightarrow \text{id} \cdot, := \))
  - This means that \text{id} appears on top of stack and can be reduced
  - Two productions can be reduced: \( S \rightarrow \text{id} \) and \( V \rightarrow \text{id} \)

- The lookahead token eliminates the conflict of the reduce items
  - If lookahead token is \$ then \text{id} is reduced to \( S \)
  - If lookahead token is := then \text{id} is reduced to \( V \)

- When in state 0 after a reduce action …
  - If \( S \) is pushed, we obtain item (\( S' \rightarrow S \cdot, \$ \)) and go to state 2
  - If \( V \) is pushed, we obtain item (\( S \rightarrow V \cdot := E, \$ \)) and go to state 3
LR(1) State Diagram

- The LR(1) state diagram of grammar $G_3$ is shown below.
- Grammar $G_3$, which was not SLR(1), is now LR(1).
- The reduce-reduce conflict that existed in state 1 is now removed.
- The lookahead token in LR(1) items eliminated the conflict.

```
S' → S . , $
S → id . , $
S → V := E , $
V → id . , :=

S → $, $,

V → id . , :=
S → id . , $,
V → id . , $,
S → id . , $,
V → id . , $,
V → id . , $,
S → V := E , $,
V → V := E , $,
E → V . , $,
E → V . , $,
E → n . , $
E → n . , $
V → id . , $,
V → id . , $,
V → id . , $,
S → V := E , $,
```

Accept
LR(1) Grammars

A grammar is LR(1) if the following two conditions are met …

- If a state contains \((A \rightarrow \alpha \bullet x \gamma, a)\) and \((B \rightarrow \beta \bullet, b)\) then \(b \neq x\)
- If a state contains \((A \rightarrow \alpha \bullet, a)\) and \((B \rightarrow \beta \bullet, b)\) then \(a \neq b\)

Violation of first condition results in a **shift-reduce conflict**

- If a state contains \((A \rightarrow \alpha \bullet x \gamma, a)\) and \((B \rightarrow \beta \bullet, x)\) then …
  - It can shift \(x\) and can reduce \(B \rightarrow \beta\) when lookahead token is \(x\)

Violation of second condition results in **reduce-reduce conflict**

- If a state contains \((A \rightarrow \alpha \bullet, a)\) and \((B \rightarrow \beta \bullet, a)\) then …
  - It can reduce \(A \rightarrow \alpha\) and \(B \rightarrow \beta\) when lookahead token is \(a\)

LR(1) grammars are a superset of SLR(1) grammars
Drawback of LR(1)

- LR(1) can generate very large parsing tables
- For a typical programming language grammar …
  - The number of states is around several hundred for LR(0) and SLR(1)
  - The number of states can be several thousand for LR(1)
- This is why parser generators do not adopt the general LR(1)
- Consider again grammar G2 for matched parentheses
  
  0: $S' \rightarrow S \$ 
  1: $S \rightarrow ( S ) S$
  2: $S \rightarrow \varepsilon$

- The LR(1) DFA has 10 states, while the LR(0) DFA has 6
LR(1) DFA of Grammar G2

$$
\begin{align*}
S' &\rightarrow \bullet S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow (S) S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow \bullet \\
S &\rightarrow (S) S \\
S' &\rightarrow S \bullet \\
S &\rightarrow \bullet S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow (S) S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow \bullet \\
S' &\rightarrow \bullet S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow (S) S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow \bullet \\
S' &\rightarrow S \bullet \\
S &\rightarrow \bullet S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow (S) S \\
S &\rightarrow \bullet (S) S \\
S &\rightarrow \bullet \\
S &\rightarrow \bullet \\
\end{align*}
$$
LALR(1) : Look-Ahead LR(1)

- Preferred parsing technique in many parser generators
- Close in power to LR(1), but with less number of states
- Increased number of states in LR(1) is because
  - Different lookahead tokens are associated with same LR(0) items
- Number of states in LALR(1) = states in LR(0)
- LALR(1) is based on the observation that
  - Some LR(1) states have same LR(0) items
  - Differ only in lookahead tokens
- LALR(1) can be obtained from LR(1) by
  - Merging LR(1) states that have same LR(0) items
  - Obtaining the union of the LR(1) lookahead tokens
LALR(1) DFA of Grammar G2

\[
S' \rightarrow S \cdot S \\
S \rightarrow ( S \cdot ) S \\
S' \rightarrow ( S ) S \\
S \rightarrow ( S \cdot ) S \\
S \rightarrow S \cdot \\
S \rightarrow ( S ) S \\
S \rightarrow S \cdot S \\
S \rightarrow ( S ) S \\
\]

Accept