Lectures 7 and 8. Parsing (syntax analysis)

Wei Le

2015.9
Bottom Up Parsing

- Recognize many programming languages, used in practice
- More general, more powerful parser: less restrictions on grammars, e.g., can handle left-recursive grammar
- No backtracking
- LR parsing: L – tokens are read from the left to right, R – rightmost derivation
- LR parsing reduces a string to the start symbol by inverting productions
LR Parsing

str ← input string of terminals
repeat
  - Identify β in str such that \( A \rightarrow \beta \) is a production (i.e., \( str = \alpha \beta \gamma \))
  - Replace \( \beta \) by \( A \) in str (i.e., \( str \) becomes \( \alpha A \gamma \))
until \( str = S \)
LR Parsing: An Example

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{int} + (\text{int} ) + (\text{int} ) \]
LR Parsing: An Example

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ E + (\text{int}) + (\text{int}) \]

\[
E \\
\mid \\
\text{int} + (\text{int}) + (\text{int})
\]
LR Parsing: An Example

\[
\begin{align*}
\text{int} &+ (\text{int}) + (\text{int}) \\
\text{E} &+ (\text{int}) + (\text{int}) \\
\text{E} &+ (\text{E}) + (\text{int})
\end{align*}
\]
LR Parsing: An Example

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
LR Parsing: An Example

\[ \text{int} + (\text{int}) + (\text{int}) \]
\[ E + (\text{int}) + (\text{int}) \]
\[ E + (E) + (\text{int}) \]
\[ E + (\text{int}) \]
\[ E + (E) \]
LR Parsing: An Example

A rightmost derivation in reverse
LR Parsing: Important Factor #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse
Important Fact #1 has an interesting consequence:
- Let $\alpha \beta \gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals!

Why? Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a right-most derivation
LR Parsing: Split Notation

• Idea: Split the string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals

• The dividing point is marked by a ►
  - The ► is not part of the string

• Initially, all input is unexamined: ►x₁x₂...xₙ
LR Parsing: Shift and Reduce

**Shift:** Move \( \rightarrow \) one place to the right
- Shifts a terminal to the left string

\[ E + (\rightarrow \text{ int } ) \Rightarrow E + ( \text{ int } \rightarrow ) \]

**Reduce:** Apply a production in reverse at the right end of the left string
- If \( E \rightarrow E + (E) \) is a production, then

\[ E + (E + (E) \rightarrow ) \Rightarrow E + (E \rightarrow ) \]
Shift and Reduce Example

- \texttt{int + (int) + (int)} \$ \texttt{shift}$

\[ \texttt{int + ( int ) + ( int )} \]
Shift and Reduce Example

\[
\begin{align*}
\text{int} & \Rightarrow (\text{int}) + (\text{int})$ & \text{shift} \\
\text{int} & \Rightarrow + (\text{int}) + (\text{int})$ & \text{red. E} \rightarrow \text{int}
\end{align*}
\]
Shift and Reduce Example

\[
\begin{align*}
\text{E} & \quad \text{shift 3 times} \\
\text{int} & \quad \text{shift} \\
\text{E} & \quad \text{red. } E \rightarrow \text{int} \\
\text{int} & \quad \text{red. } E \rightarrow \text{int}
\end{align*}
\]
Shift and Reduce Example

$\text{int } + (\text{int}) + (\text{int})$  \hspace{1em} \text{shift}

$\text{int } + (\text{int}) + (\text{int})$  \hspace{1em} \text{red. } E \rightarrow \text{int}

$E + (\text{int } ) + (\text{int})$  \hspace{1em} \text{shift 3 times}

$E + (\text{int } + (\text{int})) + (\text{int})$  \hspace{1em} \text{red. } E \rightarrow \text{int}
Shift and Reduce Example

\[\text{int} + (\text{int}) + (\text{int})\]  shift
\[\text{int} \triangleright + (\text{int}) + (\text{int})\]  red. \(E \rightarrow \text{int}\)
\[E \triangleright + (\text{int}) + (\text{int})\]  shift 3 times
\[E + (\text{int} \triangleright ) + (\text{int})\]  red. \(E \rightarrow \text{int}\)
\[E + (E \triangleright ) + (\text{int})\]  shift

\[\begin{array}{c}
E \\
\rightarrow \\
\text{int} + (\text{int}) + (\text{int})
\end{array}\]
Shift and Reduce Example

```
► int + (int) + (int)$  shift
int ► + (int) + (int)$  red. E → int
E ► + (int) + (int)$  shift 3 times
E + (int ► ) + (int)$  red. E → int
E + (E ► ) + (int)$  shift
E + (E) ► + (int)$  red. E → E + (E)
```

```
  E
  /   /
int + ( int ) + ( int )
  ^
E
```
Shift and Reduce Example

$\text{int} + (\text{int}) + (\text{int})$

- shift

$\text{int} \rightarrow + (\text{int}) + (\text{int})$

- red. $E \rightarrow \text{int}$

$E \rightarrow + (\text{int}) + (\text{int})$

- shift 3 times

$E \rightarrow (\text{int} \rightarrow ) + (\text{int})$

- red. $E \rightarrow \text{int}$

$E \rightarrow (\text{E} \rightarrow ) + (\text{int})$

- shift

$E \rightarrow + (\text{int})$

- red. $E \rightarrow E + (E)$

$E \rightarrow + (\text{int})$

- shift 3 times
Shift and Reduce Example

\[ \text{int } + (\text{int}) + (\text{int})$ shift
\]

\[ \text{int } \rightarrow (\text{int}) + (\text{int})$ red. $E \rightarrow \text{int}$
\]

\[ E \rightarrow (\text{int}) + (\text{int})$ shift 3 times
\]

\[ E + (E \rightarrow ) + (\text{int})$ red. $E \rightarrow \text{int}$
\]

\[ E + (E \rightarrow ) + (\text{int})$ shift
\]

\[ E + (E \rightarrow ) + (\text{int})$ red. $E \rightarrow E + (E)$
\]

\[ E \rightarrow + (\text{int})$ shift 3 times
\]

\[ E + (\text{int } \rightarrow )$ red. $E \rightarrow \text{int}$
\]
Shift and Reduce Example

\[
\text{\texttt{int} + (\texttt{int}) + (\texttt{int})$ shift}
\]

\[
\text{\texttt{int} $\rightarrow$ + (\texttt{int}) + (\texttt{int})$ red. E } \rightarrow \texttt{int}
\]

\[
\text{E } \rightarrow \texttt{+ (int) + (int)$ shift 3 times}
\]

\[
\text{E + (int $\rightarrow$ ) + (int)$ red. E } \rightarrow \texttt{int}
\]

\[
\text{E + (E $\rightarrow$ ) + (int)$ shift}
\]

\[
\text{E + (E $\rightarrow$ ) + (int)$ red. E } \rightarrow \texttt{E + (E)}
\]

\[
\text{E + (int)$ shift 3 times}
\]

\[
\text{E + (int $\rightarrow$ )$ red. E } \rightarrow \texttt{int}
\]

\[
\text{E + (E $\rightarrow$ )$ shift}
\]
Shift and Reduce Example

- int + (int) + (int)
  - shift
- int + (int) + (int)
  - red. E → int
- E + (int) + (int)
  - shift 3 times
- E + (int) + (int)
  - red. E → int
- E + (E) + (int)
  - shift
- E + (E) + (int)
  - red. E → E + (E)
- E + (int)
  - shift 3 times
- E + (int)
  - red. E → int
- E + (E)
  - shift
- E + (E)
  - red. E → E + (E)
Shift and Reduce Example

- `int + (int) + (int)$` shift
- `int ▶ + (int) + (int)$` red. `E → int`
- `E ▶ + (int) + (int)$` shift 3 times
- `E + (int ▶ ) + (int)$` red. `E → int`
- `E + (E ▶ ) + (int)$` shift
- `E + (E) ▶ + (int)$` red. `E → E + (E)`
- `E ▶ + (int)$` shift 3 times
- `E + (int ▶ )$` red. `E → int`
- `E + (E ▶ )$` shift
- `E + (E) ▶ $` red. `E → E + (E)`
- `E ▶ $` accept
The stack

- Left string can be implemented as a stack
  - Top of the stack is the ➤

- Shift pushes a terminal on the stack

- Reduce
  - Pops 0 or more symbols off the stack: production rhs
  - Pushes a non-terminal on the stack: production lhs
Key issues: when to shift or reduce

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and examine the resulting state \( X \) and the token \( \text{tok} \) after
  - If \( X \) has a transition labeled \( \text{tok} \) then \text{shift}
  - If \( X \) is labeled with “\( A \rightarrow \beta \) on \( \text{tok} \)” then \text{reduce}
Key issues: when to shift or reduce
LR Parsing Algorithm

• After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated

• Remember for each stack element to which state it brings the DFA

• LR parser maintains a stack

\[ \langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle \]

\text{state}_k \text{ is the final state of the DFA on } \text{sym}_1 \ldots \text{sym}_k
Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
Lines correspond to DFA states
Columns correspond to terminals and non-terminals
Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table
The table for a fragment of our DFA

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r_{E \rightarrow int}</td>
<td>r_{E \rightarrow int}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r_{E \rightarrow E+(E)}</td>
<td></td>
<td></td>
<td></td>
<td>r_{E \rightarrow E+(E)}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table-Driven Bottom Up Parsing

Parsing Stack

Token stream

TOP → $ → Parser Driver

parse states

<table>
<thead>
<tr>
<th>(</th>
<th>id</th>
<th>)</th>
<th>*</th>
<th>$</th>
<th>T</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table[state,terminal] =
- shift token and state onto stack.
- reduce by production A → β
  pop rhs from stack; push A; push next state
given by Goto[exposed state, A]
- accept
- error
LR Parsing Algorithm

Let $I = w\$\$ be initial input
Let $j = 0$
Let DFA state 0 be the start state
Let stack = \langle dummy, 0 \rangle

repeat

    case action[top_state(stack), I[j]] of
        shift $k$: push \langle I[j++], k \rangle
        reduce $X \rightarrow \alpha$:
            - pop |$\alpha$| pairs off the stack
            - push \langle $X$, Goto[top_state(stack), $X$] \rangle
        accept: halt normally
        error: halt and report error
Review: Key Ideas of LR Parsing

- A bottom-up parser rewrites the input string to the start symbol.
- The state of the parser is described as:
  \[ \alpha \rightarrow \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined
- Initially: \( \rightarrow x_1x_2 \ldots x_n \)
Review: Key Ideas of LR Parsing

- Recall the CFG: \( E \rightarrow \text{int} | E + (E) \)
- A bottom-up parser uses two kinds of actions
  - **Shift** pushes a terminal from input on the stack
    \[ E + (\text{\textleftarrow int \textrightarrow}) \Rightarrow E + (\text{\textleftarrow int \textrightarrow}) \]
  - **Reduce** pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
    \[ E + (E + (E) \text{\textleftarrow}) \Rightarrow E + (E \text{\textrightarrow}) \]
How to Decide When to Shift or Reduce

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state \( X \) and the token \( tok \) after
  - If \( X \) has a transition labeled \( tok \) then shift
  - If \( X \) is labeled with “\( A \rightarrow \beta \) on \( tok \)” then reduce
How to Construct the Parsing Table

• **The stack describes the context of the parse**
  - What non-terminal we are looking for
  - What production rhs we are looking for
  - What we have seen so far from the rhs

• **Each DFA state describes several such contexts**
  - E.g., when we are looking for non-terminal E, we might be looking either for an \texttt{int} or an \texttt{E + (E)} rhs

Run stack on DFA: search which context matches the stack?
LR(1) Item Defines the State (or Context)

- An LR(1) item is a pair
  \[ X \rightarrow \alpha \cdot \beta, a \]
  - \( X \rightarrow \alpha \beta \) is a production
  - \( a \) is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

- \([X \rightarrow \alpha \cdot \beta, a]\) describes a context of the parser
  - We are trying to find an \( X \) followed by an \( a \), and
  - We have \( \alpha \) already on top of the stack
  - Thus we need to see next a prefix derived from \( \beta a \)
Notation Clarification

- The symbol $\alpha \triangleright \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals.

- In LR(1) items $\bullet$ is used to mark a prefix of a production rhs:
  $$X \rightarrow \alpha \bullet \beta, \alpha$$
  - Here $\beta$ might contain non-terminals as well.

- In both cases the stack is on the left.
LR(1) Items Are Used to Define DFA

- A DFA state is a **closed** set of LR(1) items
  - This means that we performed Closure

- The start state contains \([S \rightarrow \bullet E, \$]\)

- A state that contains \([X \rightarrow \alpha\bullet, b]\) is labeled with “reduce with \(X \rightarrow \alpha\) on \(b\)”

- A state “State” that contains \([X \rightarrow \alpha\bullet y\beta, b]\) has a transition labeled \(y\)
  - \(y\) can be a terminal or a non-terminal
LR(1) Items Are Used to Define DFA: An Example

- In context containing
  \[ E \rightarrow E + \bullet ( E ), + \]
  - If ( follows then we can perform a shift to context containing
    \[ E \rightarrow E + ( \bullet E ), + \]
- In a context containing
  \[ E \rightarrow E + ( E ) \bullet, + \]
  - We can perform a reduction with \[ E \rightarrow E + ( E ) \]
  - But only if a + follows
LR(1) Items Are Used to Define DFA: An Example

and so on...
How To Determine a set of LR(1) Items for the State?

• We add to our grammar a fresh new start symbol $S$ and a production $S \rightarrow E$
  - Where $E$ is the old start symbol

• The initial parsing context contains:
  
  $S \rightarrow \cdot E, \$

  - Trying to find an $S$ as a string derived from $E\$
  - The stack is empty
Closure Operation: An Example

- Consider a context with the item
  \[ E \rightarrow E + ( \cdot E ), + \]
- We expect next a string derived from \( E ) + \)
- There are two productions for \( E \)
  \[ E \rightarrow \text{int} \quad \text{and} \quad E \rightarrow E + ( E ) \]
- We describe this by extending the context with two more items:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + ( E ), ) \]
Closure Operation: An Example

- The operation of extending the context with items is called the closure operation

\[
\text{Closure(Items)} =
\]

repeat
for each \([X \rightarrow \alpha \cdot Y \beta, a]\) in Items
for each production \(Y \rightarrow \gamma\)
for each \(b \in \text{First}(\beta a)\)
add \([Y \rightarrow \cdot \gamma, b]\) to Items
until Items is unchanged
Closure Operation: An Example

- Construct the start context: Closure({$S \rightarrow \bullet E, \$})

  \[
  S \rightarrow \bullet E, \$
  
  E \rightarrow \bullet E+(E), \$
  
  E \rightarrow \bullet \text{int}, \$
  
  E \rightarrow \bullet E+(E), +
  
  E \rightarrow \bullet \text{int}, +
  \]

- We abbreviate as

  \[
  S \rightarrow \bullet E, \$
  
  E \rightarrow \bullet E+(E), \$/+
  
  E \rightarrow \bullet \text{int}, \$/+
  \]
LR(1) Items Are Used to Define DFA: An Example

```
S → •E, $  
E → •E+(E), $/+  
E → •int, $/+  

E → int•, $/+  
E → int  
E → int on $, +  

E → E• (E), $/+  
E → E+(E), $/+  
E → E+(E•), $/+  
E → •E+(E), )/+  
E → •int, )/+  
E → int•, )/+  
E → int on ), +
```

and so on...
LR(1) Items Are Used to Define DFA: Transitions

- A state “State” that contains $[X \rightarrow \alpha \cdot y \beta, b]$ has a transition labeled $y$ to a state that contains the items “Transition(State, $y$)”
  - $y$ can be a terminal or a non-terminal

Transition(State, $y$)

Items $\leftarrow \emptyset$
for each $[X \rightarrow \alpha \cdot y \beta, b] \in$ State
  add $[X \rightarrow \alpha y \cdot \beta, b]$ to Items
return Closure(Items)
Automatically Generating Parsing Tables

• Parsing tables (i.e. the DFA) can be constructed automatically for a CFG

• But we still need to understand the construction to work with parser generators
  - E.g., they report errors in terms of sets of items

• What kind of errors can we expect?
Shift and Reduce Conflict

- If a DFA state contains both 
  \[ X \rightarrow \alpha \cdot a \beta, b \] and \[ Y \rightarrow \gamma \cdot, a \]

- Then on input “a” we could either
  - Shift into state \[ X \rightarrow \alpha a \cdot \beta, b \], or
  - Reduce with \[ Y \rightarrow \gamma \]

- This is called a shift-reduce conflict
Shift and Reduce Conflict

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]
- Will have DFA state containing
  \[
  [S \rightarrow \text{if } E \text{ then } S, \text{else}],
  [S \rightarrow \text{if } E \text{ then } S \text{ else } S, \text{x}]
  \]
- If \textbf{else} follows then we can shift or reduce
- Default (bison, CUP, etc.) is to shift
  - Default behavior is as needed in this case
Shift and Reduce Conflict

- Back to our dangling else example
  
  \[ S \rightarrow \text{if } E \text{ then } S_1, \text{ else } \]  
  
  \[ S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S, \ x]  

- Can eliminate conflict by declaring \text{else} with higher precedence than \text{then}
  
  - Or just rely on the default shift action
Shift and Reduce Conflict

- Consider the ambiguous grammar
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]
- We will have the states containing
  \[
  [E \rightarrow E \ast \cdot E, +] \quad [E \rightarrow E \ast E \cdot, +] \\
  [E \rightarrow \cdot E + E, +] \quad \Rightarrow^E \quad [E \rightarrow E \cdot + E, +] \\
  \ldots \quad \ldots 
  \]
- Again we have a shift/reduce on input +
  - We need to reduce (\ast binds more tightly than +)
  - Recall solution: declare the precedence of \ast and +
Reduce Reduce Conflict

- If a DFA state contains both
  \[ X \rightarrow \alpha \cdot, a \] and \[ Y \rightarrow \beta \cdot, a \]
  - Then on input “a” we don’t know which production to reduce

- This is called a reduce/reduce conflict
Reduce Reduce Conflict

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers
  \[ S \rightarrow \varepsilon \mid \text{id} \mid \text{id} \ S \]

- There are two parse trees for the string \text{id}
  \[ S \rightarrow \text{id} \]
  \[ S \rightarrow \text{id} \ S \rightarrow \text{id} \]
- How does this confuse the parser?
Reduce Reduce Conflict

• **Consider the states**
  
  \[
  \begin{align*}
  [S' \rightarrow \bullet S, & \quad $] \\
  [S \rightarrow \bullet, & \quad $] \quad \Rightarrow^{id} \quad [S \rightarrow \bullet, & \quad $] \\
  [S \rightarrow \bullet \text{id}, & \quad $] \\
  [S \rightarrow \bullet \text{id} S, & \quad $]
  \end{align*}
  \]

  \[\begin{align*}
  [S \rightarrow \text{id} \bullet, & \quad $] \\
  [S \rightarrow \text{id} \bullet S, & \quad $]
  \end{align*}\]

• **Reduce/reduce conflict on input $**
  
  \[
  \begin{align*}
  S' \rightarrow S \rightarrow \text{id} \\
  S' \rightarrow S \rightarrow \text{id} \text{ S} \rightarrow \text{id}
  \end{align*}
  \]

• **Better rewrite the grammar:**
  
  \[S \rightarrow \varepsilon \mid \text{id} S\]
Motivation of LALR

- Parser generators construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)
- But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language
Motivation of LALR

- But many states are similar, e.g.
  \[ E \rightarrow \text{int, $/, +} \]
  \[ E \rightarrow \text{int on $, +} \]
  and
  \[ E \rightarrow \text{int, $/, +} \]
  \[ E \rightarrow \text{int on $, +} \]

- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

- We obtain
  \[ E \rightarrow \text{int, $/, +/)} \]
  \[ E \rightarrow \text{int on $, +, )} \]
Motivation of LALR

• Definition: The core of a set of LR items is the set of first components
  - Without the lookahead terminals

• Example: the core of
  \[
  \{ [X \rightarrow \alpha \cdot \beta, b], [Y \rightarrow \gamma \cdot \delta, d]\}
  \]
  is
  \[
  \{ X \rightarrow \alpha \cdot \beta, \ Y \rightarrow \gamma \cdot \delta \} 
  \]
LALR

- Consider for example the LR(1) states
  \[
  \begin{align*}
  &\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., c]\} \\
  &\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., d]\}
  \end{align*}
  \]
- They have the same core and can be merged
- And the merged state contains:
  \[
  \{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., c/d]\}
  \]
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)
• Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors
LALR: Example
LALR

• Consider for example the LR(1) states
  \[ ([X \rightarrow \alpha\bullet, a], [Y \rightarrow \beta\bullet, b]) \]
  \[ ([X \rightarrow \alpha\bullet, b], [Y \rightarrow \beta\bullet, a]) \]
• And the merged LALR(1) state
  \[ ([X \rightarrow \alpha\bullet, a/b], [Y \rightarrow \beta\bullet, a/b]) \]
• Has a new reduce-reduce conflict
• In practice such cases are rare

• However, no new shift/reduce conflicts. Why?
• LALR languages are not natural
  - They are an efficiency hack on LR languages

• Any reasonable programming language has a LALR(1) grammar

• LALR(1) has become a standard for programming languages and for parser generators
Parsing: A Summary

- Parse tree: how to derive a string from the grammar
- Context free grammars
- Algorithms: Recursive Decendent Parsers, LL (1), LR (1), LALR (1)
- General approach: table-driven parsing (the stack, streams of tokens, general parsing algorithm), using table to guide each step of building parsing trees
- Challenge: Building parsing table