Bottom-Up Parsing
LR Parsing. Parser Generators.

Lecture 6
Bottom-Up Parsing

• Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice

• Also called LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation!
An Introductory Example

• LR parsers don’t need left-factored grammars and can also handle left-recursive grammars

• Consider the following grammar:

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

- Why is this not LL(1)?

• Consider the string: \text{int} + ( \text{int} ) + ( \text{int} )
The Idea

- LR parsing reduces a string to the start symbol by inverting productions:

str = input string of terminals
repeat
  - Identify $\beta$ 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 = $S$
A Bottom-up Parse in Detail (1)

\[ \text{int} + (\text{int}) + (\text{int}) \]
A Bottom-up Parse in Detail (2)

\[ \text{int} + (\text{int}) + (\text{int}) \]
\[ E + (\text{int}) + (\text{int}) \]
A Bottom-up Parse in Detail (3)

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

\[
\begin{align*}
E & \quad E \\
\text{int} & + ( \text{int} ) + ( \text{int} )\end{align*}
\]
A Bottom-up Parse in Detail (4)

\[
\begin{align*}
\text{int} & + (\text{int}) + (\text{int}) \\
E & + (\text{int}) + (\text{int}) \\
E & + (E) + (\text{int}) \\
E & + (\text{int}) \\
\end{align*}
\]
A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} + (\text{int}) + (\text{int}) \\
E + (\text{int}) + (\text{int}) \\
E + (E) + (\text{int}) \\
E + (\text{int}) \\
E + (E)
\end{align*}
\]
A Bottom-up Parse in Detail (6)

A rightmost derivation in reverse

```
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
```

```
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
```
Important Fact #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse
Where Do Reductions Happen

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
Notation

• Idea: Split 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: $|x_1x_2 \ldots x_n$
Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

  \[ \text{Shift} \]

  \[ \text{Reduce} \]
**Shift**

*Shift: Move a one place to the right*
- Shifts a terminal to the left string

\[ E + (1 \text{ int}) \Rightarrow E + (\text{int } 1) \]
Reduce

Reduce: Apply an inverse production at the right end of the left string

- If $E \rightarrow E + (E)$ is a production, then

$$E + (E + (E)) \Rightarrow E + (E)$$
Shift-Reduce Example

int + (int) + (int)$ shift

int + (int) + (int)
Shift-Reduce Example

```
<table>
<thead>
<tr>
<th>int  +  (int) +  (int)$</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>int  l   +  (int) +  (int)$</td>
<td>red. E</td>
</tr>
<tr>
<td>→ int</td>
<td></td>
</tr>
</tbody>
</table>
```

```
int  +  ( int ) +  ( int )

↑
```
Shift-Reduce Example

\[
\begin{align*}
1 \text{ int } + (\text{int}) + (\text{int}) & \quad \text{shift} \\
\text{int } 1 + (\text{int}) + (\text{int}) & \quad \text{red. E} \\
& \quad \rightarrow \text{ int} \\
\text{E } 1 + (\text{int}) + (\text{int}) & \quad \text{shift} \\
& \quad \text{3 times}
\end{align*}
\]
Shift-Reduce Example

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

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

\[
\text{shift}
\]

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

\[
\text{red. E}
\]

\[
\rightarrow \text{int}
\]

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

\[
\text{shift}
\]

\[
3 \text{ times}
\]

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

\[
\text{red. E}
\]

\[
\rightarrow \text{int}
\]

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

\[
\text{shift}
\]

\[
\text{E} \quad \text{E}
\]

\[
\rightarrow \text{int} + (\text{int}) + (\text{int})
\]
Shift-Reduce Example

1. int + (int) + (int)$  
   shift

2. int 1 + (int) + (int)$  
   red. E
   → int

3. E 1 + (int) + (int)$  
   shift 3 times

4. E + (E 1 ) + (int)$  
   red. E
   → int

5. E + (E 1 ) + (int)$  
   shift
   / /  
   E E

6. E + (E ) 1 + (int)$  
   red. E int + ( int ) + ( int )
   → E + (E)
Shift-Reduce Example

1. \texttt{int + (int) + (int)$}\quad\text{shift}
2. \texttt{int \rightarrow int}$
3. \texttt{E \rightarrow E}$
4. \texttt{E + (E + (int)$}\quad\text{shift}
5. \texttt{E \rightarrow E + (E)$}
6. \texttt{E \rightarrow E}$
7. \texttt{E \rightarrow E + (E)$}\quad\text{shift 3
Shift-Reduce Example

\[
\begin{align*}
1 \text{ int } + (\text{int}) + (\text{int}) & \quad \text{shift} \\
\text{int } 1 + (\text{int}) + (\text{int}) & \quad \text{red. E} \\
& \quad \rightarrow \text{ int} \\
\text{E } 1 + (\text{int}) + (\text{int}) & \quad \text{shift 3} \\
& \quad \text{times} \\
\text{E } + (\text{int } 1) + (\text{int}) & \quad \text{red. E} \\
& \quad \rightarrow \text{ int} \\
\text{E } + (\text{E } 1) + (\text{int}) & \quad \text{shift} \\
\text{E } + (\text{E}) 1 + (\text{int}) & \quad \text{red. E} \\
& \quad \rightarrow \text{E } + (\text{E}) \\
\text{E } 1 + (\text{int}) & \quad \text{shift 3}
\end{align*}
\]
Shift-Reduce Example

\[
\begin{align*}
\text{int} + \text{(int)} + \text{(int)} & \quad \text{shift} \\
\text{int} \ 	ext{int} + \text{(int)} + \text{(int)} & \quad \text{red. E} \\
& \rightarrow \text{int} \\
E \ 	ext{int} + \text{(int)} + \text{(int)} & \quad \text{shift 3 times} \\
& \rightarrow \text{int} \\
E + \text{(int)} + \text{(int)} & \quad \text{red. E} \\
& \rightarrow \text{int} \\
E + \text{(E)} + \text{(int)} & \quad \text{shift} \\
E + \text{(E)} \ 	ext{int} + \text{(int)} & \quad \text{red. E} \\
& \rightarrow E + \text{(E)} \\
E \ 	ext{int} + \text{(int)} & \quad \text{shift 3}
\end{align*}
\]
Shift-Reduce Example

$1 \int + (\int) + (\int)\$

shift

$\int 1 + (\int) + (\int)\$

red. $E$

→ $\int$

$E 1 + (\int) + (\int)\$

shift 3 times

$E + (\int 1) + (\int)\$

red. $E$

→ $\int$

$E + (E 1) + (\int)\$

shift

$E + (E) 1 + (\int)\$

red. $E$

→ $E + (E)$

$E 1 + (\int)\$

shift 3
Shift-Reduce Example

```
\text{I \ int + (int) + (int)$ shift  
\int \ I + (int) + (int)$ red. E  
\rightarrow \ int  
E \ I + (int) + (int)$ shift 3 times  
E + (int I) + (int)$ red. E  
\rightarrow \ int  
E + (E I) + (int)$ shift  
E + (E) I + (int)$ red. E  
\rightarrow \ E + (E)  
E I + (int)$ shift 3  
```

The Stack

- Left string can be implemented by a stack
  - Top of the stack is the $I$
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
Key Issue: 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 we examine the resulting state $X$ and the token $tok$ after $i$
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on $tok$” then reduce
LR(1) Parsing. An Example

0 \rightarrow int

E \rightarrow int on $, +

1 \rightarrow int

E + (int) + (int)$ shift

2 \rightarrow E + (E)

on $, +

3 \rightarrow (on $)

4 \rightarrow int

E \rightarrow int on $, +

5 \rightarrow E + (int) + (int)$

6 \rightarrow int

E \rightarrow E + (E)

7 \rightarrow int

E \rightarrow int on $, +

8 \rightarrow (on $)

9 \rightarrow int

E \rightarrow E + (E)

10 \rightarrow E \rightarrow E + (E)

11 \rightarrow E \rightarrow E + (E)
Representing the DFA

- 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
Representing the DFA. Example

- The table for a fragment of our DFA:

<table>
<thead>
<tr>
<th>int</th>
<th>+</th>
<th>( )</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s5</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s7</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>r_E→E+</td>
<td></td>
<td>r_E→E+</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96</td>
<td></td>
</tr>
</tbody>
</table>

3 \rightarrow int
4 \rightarrow int
5 \rightarrow E \rightarrow int
6 \rightarrow E \rightarrow int
7 \rightarrow E \rightarrow int
8 \rightarrow E \rightarrow int
9 \rightarrow E \rightarrow int

The 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 on 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 is the final state of the DFA on \text{sym}_1 \ldots \text{sym}_k
The LR Parsing Algorithm

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

repeat
  case action[top_state(stack), $I[j]$] of
    shift $k$: push $\langle I[j++], k \rangle$
    reduce $X \rightarrow \alpha$:
      pop $|\alpha|$ pairs,
      push $\langle X, \text{Goto[top_state(stack), } X \rangle$
  accept: halt normally
  error: halt and report error
LR Parsing Notes

- Can be used to parse more grammars than LL
- Most programming languages grammars are LR
- Can be described as a simple table
- There are tools for building the table
- How is the table constructed?
Key Issue: How is the DFA Constructed?

• 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 $\text{int}$ or a $E + (E)$ rhs
LR(1) Items

• An LR(1) item is a pair:

  \[ X \rightarrow \alpha \cdot \beta, \ a \]

  - \( X \rightarrow \alpha \cdot \beta \) 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 \( \beta a \)
Note

• The symbol $I$ was used before to separate the stack from the rest of input
  - $\alpha I \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals
• In items . is used to mark a prefix of a production rhs:
  \[ X \rightarrow \alpha \beta, \alpha \]
  - Here $\beta$ might contain non-terminals as well
• In both case the stack is on the left
Convention

- 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:
  $$S \rightarrow .E, \$$$
  - Trying to find an $S$ as a string derived from $E\$
  - The stack is empty
LR(1) Items (Cont.)

• In context containing
  \[ E \to E + . ( E ), + \]
  - If ( follows then we can perform a shift to context containing
    \[ E \to E + (. E ), + \]

• In context containing
  \[ E \to E + ( E ) ., + \]
  - We can perform a reduction with \[ E \to E + ( E ) \]
  - But only if a + follows
LR(1) Items (Cont.)

- Consider the item
  \[ E \rightarrow E + (. E ) , + \]
- We expect 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:
  \[ E \rightarrow . \text{int} , ) \]
  \[ E \rightarrow . E + ( E ) , ) \]
The Closure Operation

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

\[ \text{Closure(Items)} = \]
\[
\text{repeat }
\text{for each } [X \rightarrow \alpha.Y\beta, a] \text{ in Items }
\text{for each production } Y \rightarrow \gamma
\text{for each } b \in \text{First}(\beta a)
\text{add } [Y \rightarrow \gamma, b] \text{ to Items }
\text{until Items is unchanged } \]
Constructing the Parsing DFA (1)

• Construct the start context: Closure({S → .E, $})

\[
\begin{align*}
S & \rightarrow .E, $ \\
E & \rightarrow .E+(E), $ \\
E & \rightarrow .int, $ \\
E & \rightarrow .E+(E), + \\
E & \rightarrow .int, +
\end{align*}
\]

• We abbreviate as:

\[
\begin{align*}
S & \rightarrow .E, $ \\
E & \rightarrow .E+(E), $/+ \\
E & \rightarrow .int, $/+ 
\end{align*}
\]
Constructing the Parsing DFA (2)

- A DFA state is a **closed** set of LR(1) items
- The start state contains $[S \rightarrow .E, \$
- A state that contains $[X \rightarrow \alpha., b]$ is labeled with “reduce with $X \rightarrow \alpha$ on b”
- And now the transitions ...
The 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 \(\tilde{A} \not\subset \emptyset\)
for each \([X \rightarrow \alpha \cdot y\beta, b]\) 2 State
  add \([X \rightarrow \alpha y\cdot \beta, b]\) to Items
return Closure(Items)
Constructing the Parsing DFA. Example.

\[
S \rightarrow .E, \$
E \rightarrow .E+(E), $/+\nE \rightarrow .int, $/+\n\]

\[
S \rightarrow E, \$
E \rightarrow E.+ (E), $/+\n\]

\[
E \rightarrow \text{int ., } $/+\nE \rightarrow \text{int on } $, +\n\]

\[
E \rightarrow E+. (E), $/+\n\]

\[
E \rightarrow E+. (E), $/+\nE \rightarrow .E+(E), )/+\nE \rightarrow .int, )/+\n\]

\[
E \rightarrow E+(.E), $/+\nE \rightarrow .E+(E), )/+\nE \rightarrow .int, )/+\n\]

\[
E \rightarrow \text{int ., )/+}\nE \rightarrow \text{int on ), +}\n\]

and so on...
LR Parsing Tables. Notes

• 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/Reduce Conflicts

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

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

• This is called a shift-reduce conflict
Shift/Reduce Conflicts

• 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
  \[ [S \rightarrow \text{if } E \text{ then } S, \quad \text{else}] \]
  \[ [S \rightarrow \text{if } E \text{ then } S, \text{ else } S, \quad x] \]

• If else follows then we can shift or reduce

• Default (bison, CUP, etc.) is to shift
  - Default behavior is as needed in this case
More Shift/Reduce Conflicts

• Consider the ambiguous grammar
  \[ E \rightarrow E + E | E * E | \text{int} \]

• We will have the states containing

  ...

• Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Recall solution: declare the precedence of *
More Shift/Reduce Conflicts

• In bison declare precedence and associativity:
  \[ \text{%left } + \]
  \[ \text{%left } * \]

• Precedence of a rule = that of its last terminal
  - See bison manual for ways to override this default

• Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative
Using Precedence to Solve S/R Conflicts

• Back to our example:

  \[ E \rightarrow E \cdot E, + \]  \[ E \rightarrow E \cdot E, + \]  
  \[ E \rightarrow E \cdot E + E, + \] \Rightarrow^E \[ E \rightarrow E \cdot E, + \] 

  ...  

• Will choose reduce because precedence of rule \( E \rightarrow E \cdot E \) is higher than of terminal +
Using Precedence to Solve S/R Conflicts

- Same grammar as before
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]
- We will also have the states
  \[
  \begin{align*}
  [E \rightarrow E + . E, +] & \quad [E \rightarrow E + E., +] \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  \cdots & \quad \cdots \\
  \end{align*}
  \]

- Now we also have a shift/reduce on input +
  - We choose reduce because \( E \rightarrow E + E \) and + have the same precedence and + is left-
Using Precedence to Solve S/R Conflicts

• Back to our dangling else example
  \[ S \rightarrow \text{if } E \text{ then } S. \quad \text{else} \]
  \[ S \rightarrow \text{if } E \text{ then } S. \text{ else } S, \quad x \]

• Can eliminate conflict by declaring else with higher precedence than then
  – Or just rely on the default shift action

• But this starts to look like “hacking the parser”

• Best to avoid overuse of precedence declarations or you’ll end with unexpected parse trees
Reduce/Reduce Conflicts

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

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

• 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}

\[
\begin{align*}
S & \rightarrow \text{id} \\
S & \rightarrow \text{id} S \rightarrow \text{id}
\end{align*}
\]

• How does this confuse the parser?
More on Reduce/Reduce Conflicts

• Consider the states
  
  \[
  [S \rightarrow \text{id}, \$
  
  [S' \rightarrow . S, \$, 
  \]
  
  [S \rightarrow ., \$, 
  \]
  \]
  [S \rightarrow ., \$, 
  \]
  \]
  [S \rightarrow . \text{id}, \$, 
  \]
  
  [S \rightarrow . \text{id} S, \$, 
  \]
  \]
  [S \rightarrow . \text{id} S, \$, 
  \]
  \]

• Reduce/reduce conflict on input $
Using Parser Generators

- 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
LR(1) Parsing Tables are Big

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

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

- We obtain
  \[
  E \rightarrow \text{int.}, $/+/
  \]
  \[
  E \rightarrow \text{int.} \text{ on } $, +, )
  \]
The Core of a Set of LR Items

• 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 States

• Consider for example the LR(1) states
  \{\[X \rightarrow \alpha, a\], \[Y \rightarrow \beta, c\]\}
  \{\[X \rightarrow \alpha, b\], \[Y \rightarrow \beta, d\]\}
• 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)
A LALR(1) DFA

• 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
Conversion LR(1) to LALR(1). Example.

Conversion diagrams for LR(1) and LALR(1) parsing.
The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}
  \{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}
• And the merged LALR(1) state
  \{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}
• Has a new reduce-reduce conflict

• In practice such cases are rare
LALR vs. LR Parsing

- 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
A Hierarchy of Grammar Classes

From Andrew Appel, "Modern Compiler Implementation in Java"
Notes on Parsing

• Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

• Now we move on to semantic analysis
Supplement to LR Parsing

Strange Reduce/Reduce Conflicts
Due to LALR Conversion
(from the bison manual)
Strange Reduce/Reduce Conflicts

• Consider the grammar

  \[ S \rightarrow P \, R \, , \quad NL \rightarrow N \mid N , NL \]
  \[ P \rightarrow T \mid NL : T \quad R \rightarrow T \mid N : T \]
  \[ N \rightarrow id \quad T \rightarrow id \]

• \( P \) - parameters specification
• \( R \) - result specification
• \( N \) - a parameter or result name
• \( T \) - a type name
• \( NL \) - a list of names
Strange Reduce/Reduce Conflicts

• In $P$ an \texttt{id} is a
  - $N$ when followed by , or : 
  - $T$ when followed by \texttt{id}

• In $R$ an \texttt{id} is a
  - $N$ when followed by :
  - $T$ when followed by ,

• This is an LR(1) grammar.

• But it is not LALR(1). Why?
  - For obscure reasons
A Few LR(1) States

1. \( P \rightarrow . T \quad \text{id} \)
2. \( R \rightarrow . T \quad , \)
   \( R \rightarrow . N : T \quad , \)
   \( T \rightarrow . \text{id} \quad , \)
   \( N \rightarrow . \text{id} \quad : \)
3. \( T \rightarrow . \text{id} \quad \text{id} \)
   \( N \rightarrow . \text{id} \quad : \)
   \( N \rightarrow . \text{id} \quad , \)
4. \( T \rightarrow . \text{id} \quad \text{id} \quad , \)
   \( N \rightarrow . \text{id} \quad : \)

LALR reduce/reduce conflict on ","
What Happened?

• Two distinct states were confused because they have the same core
• Fix: add dummy productions to distinguish the two confused states
• E.g., add
  \[ R \rightarrow \text{id bogus} \]
  - \text{bogus} is a terminal not used by the lexer
  - This production will never be used during parsing
  - But it distinguishes \text{R} from \text{P}
A Few LR(1) States After Fix

Different cores ⇒ no LALR merging