Bottom-Up Parsing

Lecture 11-12
(From slides by G. Necula & R. Bodik)
Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Most common form is 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: \texttt{int + ( int ) + ( int )}
The Idea

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

\[ \text{str} \leftarrow \text{input string of terminals} \]

\(\text{while } \text{str} \neq S:\)

- Identify first \(\beta\) in \(\text{str}\) such that \(A \rightarrow \beta\) is a production and \(S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = \text{str}\)
- Replace \(\beta\) by \(A\) in \(\text{str}\) (so \(\alpha A \gamma\) becomes new \(\text{str}\))

• Such \(\alpha \beta\)'s are called handles
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)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
A Bottom-up Parse in Detail (4)

\[
\text{int} + (\text{int}) + (\text{int}) \\
E + (\text{int}) + (\text{int}) \\
E + (E) + (\text{int}) \\
E + (\text{int})
\]
A Bottom-up Parse in Detail (5)

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

A reverse rightmost derivation
Where Do Reductions Happen

Because an LR parser produces a reverse rightmost derivation:

- If $\alpha\beta\gamma$ is step of a bottom-up parse with handle $\alpha\beta$
- And the next reduction is by $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals!

... Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a right-most derivation

Intuition: We make decisions about what reduction to use after seeing all symbols in handle, rather than before (as for LL(1))
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 I
  - The I is not part of the string
  - Marks end of next potential handle

• Initially, all input is unexamined: Ix_1x_2 \ldots x_n
Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:
  
  **Shift**: Move one place to the right, shifting a terminal to the left string
  
  \[ E + ( \text{int} ) \Rightarrow E + ( \text{int} \ 1) \]
  
  **Reduce**: Apply an inverse production at the handle.
  
  If \( E \rightarrow E + (E) \) is a production, then
  
  \[ E + (E + (E \ 1)) \Rightarrow E + (E \ 1) \]
Shift-Reduce Example

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

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

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

$\text{int } l + (\text{int}) + (\text{int})$  red. $E \rightarrow \text{int}$
Shift-Reduce Example

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

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

1. `int + (int) + (int)$` shift
2. `int I + (int) + (int)$` red. `E → int`
3. `E I + (int) + (int)$` shift 3 times
4. `E + (int I ) + (int)$` red. `E → int`
Shift-Reduce Example

1. `int + (int) + (int)$` shift
2. `int I + (int) + (int)$` red. $E \rightarrow int$
3. `E I + (int) + (int)$` shift 3 times
4. `E + (int I ) + (int)$` red. $E \rightarrow int$
5. `E + (E I ) + (int)$` shift
Shift-Reduce Example

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

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

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

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

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

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

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

1 int + (int) + (int)$ shift
int I + (int) + (int)$ red. E → int
E I + (int) + (int)$ shift 3 times
E + (int I ) + (int)$ red. E → int
E + (E I ) + (int)$ shift
E + (E) I + (int)$ red. E → E + (E)
E I + (int)$ shift 3 times
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 times
E + (int 1)$  red. E → int

Diagram:

```
  E
 /\  \
E  E
/\  /\nint + ( int ) + ( int )
```
Shift-Reduce Example

1. `int + (int) + (int)$`  shift
2. `int $I + (int) + (int)$`  red. $E \rightarrow \text{int}$
3. `E $I + (int) + (int)$`  shift 3 times
4. `E + (int $I + (int)$`  red. $E \rightarrow \text{int}$
5. `E + (E $I + (int)$`  shift
6. `E + (E) $I + (int)$`  red. $E \rightarrow E + (E)$
7. `E $I + (int)$`  shift 3 times
8. `E + (int $I + (int)$`  red. $E \rightarrow \text{int}$
9. `E + (E $I + (int)$`  shift
Shift-Reduce Example

1. int + (int) + (int)$  shift
2. int I + (int) + (int)$  red. E → int
3. E I + (int) + (int)$  shift 3 times
4. E + (int I) + (int)$  red. E → int
5. E + (E I) + (int)$  shift
6. E + (E) I + (int)$  red. E → E + (E)
7. E I + (int)$  shift 3 times
8. E + (int I)$  red. E → int
9. E + (E I)$  shift
10. E + (E)$  red. E → E + (E)
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 times
E + (int 1)$    red. E → int
E + (E 1)$    shift
E + (E) 1$    red. E → E + (E)
E 1$    accept
```

The Stack

• Left string can be implemented as a stack
  - Top of the stack is the 1

• Shift pushes a terminal on the stack

• Reduce pops 0 or more symbols from 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 up to potential handle
  - DFA alphabet consists of terminals and nonterminals
  - DFA recognizes complete handles

- We run the DFA on the stack and we examine the resulting state $X$ and the token $tok$ after $\mathbf{1}$
  - 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

**States:**

0. **E** → **int**

1. **E** → **int** + **(int)** + **(int)**$
   $ shift

2. accept on $**

3. **E** + **(int)** + **(int)**$
   $ shift(x3)

4. **E** → **int**

5. **E** + **(E)** + **(int)**$
   $ shift

6. **E** → **E** + **(E)**

7. accept on $

8. **E** + **(int)**$

9. **E** + **(int)**$

10. **E** + **(E)**$

11. **E** + **(E)**$

**Actions:**

- **shift**
- **accept**
- **reduce**
Representing the DFA

• Parsers represent the DFA as a 2D table
  - As for table-driven lexical analysis
• Lines correspond to DFA states
• Columns correspond to terminals and non-terminals
• In classical treatments, 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:

```plaintext
<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></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 -&gt; int</td>
<td></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 -&gt; E+(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:

```
3  ( 4
   |   int
   |   |
6  5  |
   |   E -> int
   |   |
   |   )
   |   |
   |   E -> E + (E) on $, +
   |   |
7  |
```
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

• So record, for each stack element, state of the DFA after that state

• 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 = \( w_1 \ldots w_n \) be initial input
Let \( j = 1 \)
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 \( \langle I[j], k \rangle \); j += 1
    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?
To Be Done

• Review of bottom-up parsing

• Computing the parsing DFA

• Using parser generators
Bottom-up Parsing (Review)

• A bottom-up parser rewrites the input string to the start symbol
• The state of the parser is described as
  \( \alpha \mid \gamma \)
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined

• Initially: \( \text{\textbackslash } x_1x_2 \ldots x_n \)
The Shift and Reduce Actions (Review)

- Recall the CFG: $E \rightarrow \text{int} \mid E + (E)$
- A bottom-up parser uses two kinds of actions:
  - **Shift** pushes a terminal from input on the stack
    $$E + (\text{int}) \Rightarrow E + (\text{int})$$
  - **Reduce** pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
    $$E + (E + (E) \text{int}) \Rightarrow E + (E \text{int})$$
Key Issue: 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 $I$
  - 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

```
E → int

E + (int) + (int)$ shift
int E + (int) + (int)$ E → int
E + (E) + (int)$ shift
E + (E) + (int)$ shift(x3)
E + (E) + (int)$ shift
E + (E) + (int)$ shift
E $ accept
```

- E → int
- E + (int) + (int)$ shift
- E $ accept
Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What productions we are looking for
  - What we have seen so far from the rhs
Parsing Contexts

- Consider the state:

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

- The stack is

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

- Context:
  - We are looking for an $E \rightarrow E + (\cdot E)$
    - Have have seen $E + (\cdot)$ from the right-hand side
  - We are also looking for $E \rightarrow \cdot int$ or $E \rightarrow \cdot E + (E)$
    - Have seen nothing from the right-hand side

- One DFA state describes several contexts
LR(1) Items

- 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 \( \beta a \)
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 LR(1) items • is used to mark a prefix of a production rhs:
  \[ X \rightarrow \alpha \cdot \beta, \ a \]
  - 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
  - No need to do this if $E$ had only one production

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

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

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

• 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:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + ( E ), ) \]
The Closure Operation

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

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

• Construct the start context: \( \text{Closure}({S \rightarrow \cdot E, \$}) \)

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

• We abbreviate as:

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

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

- The start state is $\text{Closure}([S \rightarrow \cdot E, \$])$

- A state that contains $[X \rightarrow \alpha \cdot, b]$ 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)

\[
\text{Items} \leftarrow \emptyset
\]

for each \([X \rightarrow \alpha \cdot y\beta, b] \in \text{State}\)

add \([X \rightarrow \alpha y\cdot \beta, b]\) to Items

return Closure(Items)
Constructing the Parsing DFA. Example.

$S \rightarrow \cdot E, \$\nE \rightarrow \cdot E+(E), \$/+\nE \rightarrow \cdot \text{int}, \$/+$

$S \rightarrow E\cdot, \$
E \rightarrow E\cdot+(E), \$/+$

$E \rightarrow E, \$
E \rightarrow E\cdot+(E), \$/+$

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
  
  \[ X \rightarrow \alpha \cdot a \beta, b \] 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/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
  \[ \{ S \rightarrow \text{if } E \text{ then } S \cdot, \text{ else} \}\]
  \[ \{ S \rightarrow \text{if } E \text{ then } S \cdot \text{ else } S, \$\}\]
• If \text{else} follows then we can shift or reduce
More Shift/Reduce Conflicts

• Consider the ambiguous grammar

\[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

• We will have the states containing

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

... ...

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

• In bison declare precedence and associativity of terminal symbols:
  \%
  \%left +
  \%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:
  - 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:

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

... ... ...

• Will choose reduce because precedence of rule \( E \rightarrow E \ast E \) 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
  \[
  [E \rightarrow E \ast E, +] \quad [E \rightarrow E + E, +] \\
  [E \rightarrow \ast E + E, +] \quad \Rightarrow_{E} \quad [E \rightarrow E + E, +]
  \]

• 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-associative
Using Precedence to Solve S/R Conflicts

- Back to our dangling else example
  \[ S \rightarrow \text{if E then } S^\bullet, \quad \text{else} \]
  \[ S \rightarrow \text{if E then } S^\bullet \text{ else } S, \quad x \]
- Can eliminate conflict by declaring \textit{else} with higher precedence than \textit{then}
- However, best to avoid overuse of precedence declarations or you’ll end with unexpected parse trees
Reduce/Reduce Conflicts

• If a DFA state contains both

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

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

• There are two parse trees for the string \( id \)
  \[
  S \rightarrow id \\
  S \rightarrow id \ S \rightarrow id
  \]

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

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

- Reduce/reduce conflict on input $\$

  \[
  S' \rightarrow S \rightarrow \text{id} \\
  S' \rightarrow S \rightarrow \text{id } S \rightarrow \text{id}
  \]

- Better rewrite the grammar:

  \[
  S \rightarrow \varepsilon \mid \text{id } S
  \]
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.
  - Idea: merge the DFA states whose items differ only in the lookahead tokens
    - We say that such states have the same core
  - We obtain

\[
\begin{align*}
E &\rightarrow \text{int}\cdot, \$, + \\
E &\rightarrow \text{int} \\
E &\rightarrow \text{int}\cdot, \), + \\
E &\rightarrow \text{int} \\
\end{align*}
\]
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
  \[ \{ [X \to \alpha \cdot \beta, b], [Y \to \gamma \cdot \delta, d] \} \]
  is
  \[ \{ X \to \alpha \cdot \beta, Y \to \gamma \cdot \delta \} \]
LALR States

- Consider for example the LR(1) states
  $$\{[X \rightarrow \alpha\bullet, a], [Y \rightarrow \beta\bullet, c]\}$$
  $$\{[X \rightarrow \alpha\bullet, b], [Y \rightarrow \beta\bullet, d]\}$$
- They have the same core and can be merged
- And the merged state contains:
  $$\{[X \rightarrow \alpha\bullet, a/b], [Y \rightarrow \beta\bullet, c/d]\}$$
- 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.

```
int E → int
on $, +

E → E + (E)
on $, +

E → int
on $, +

E → int
on $, +, )

E → E + (E)
on $, +, )
```

Diagram:

```
0 → int → 1

E → int
on $, +

E → E + (E)
on $, +

accept on $ 

2 + 3 ( 4 int

E → int
on $, +

E → int
on $, +, )

6 7 5 4

E → E + (E)
on $, +

0 → int → 1,5

E → int
on $, +, )

2 + 3,8 ( 4,9 int

E → int
on $, +

E → E + (E)
on $, +, )

7,11 6,10

E → E + (E)
on $, +, )
```
The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \[
  \{[X \rightarrow \alpha\cdot, a], [Y \rightarrow \beta\cdot, b]\}
  \{[X \rightarrow \alpha\cdot, b], [Y \rightarrow \beta\cdot, a]\}
  \]

• And the merged LALR(1) state
  \[
  \{[X \rightarrow \alpha\cdot, a/b], [Y \rightarrow \beta\cdot, a/b]\}
  \]

• 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

• But 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)
  - We use LALR(1) parser generators