

Bottom-Up Parsing

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

The Idea

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

$\text{sent} \leftarrow$ input string of terminals

while $\text{sent} \neq S$:

- Identify first β in sent such that $A \rightarrow \beta$ is a production and $S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = \text{sent}$
- Replace β by A in sent (so $\alpha A \gamma$ becomes new sent)
- Such β 's are called *handles*

A Bottom-up Parse in Detail (1)

int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

(handles in red)

E
|
int + (int) + (int)

A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

 E E
 | |
int + (int) + (int)

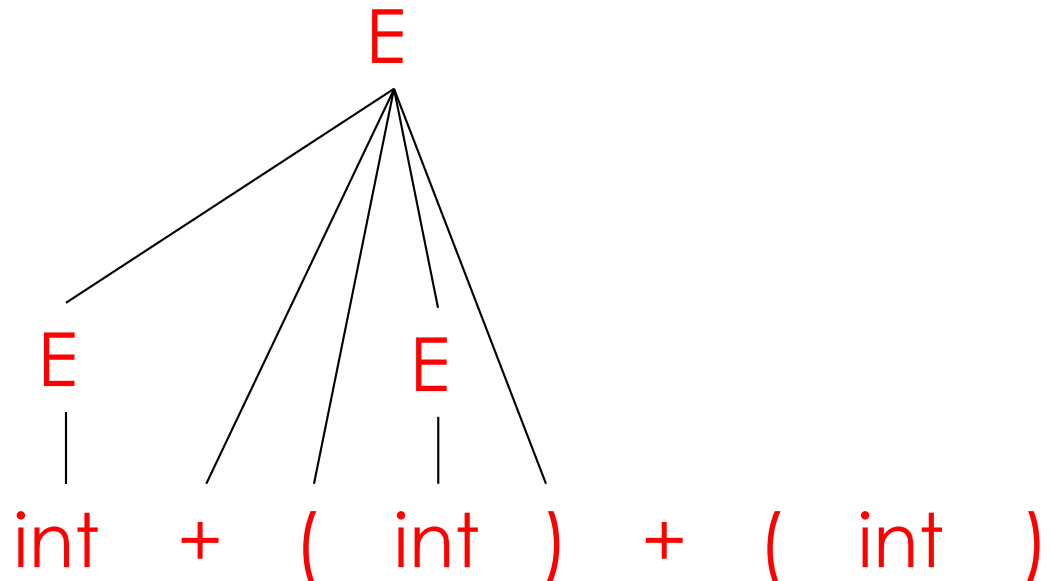
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

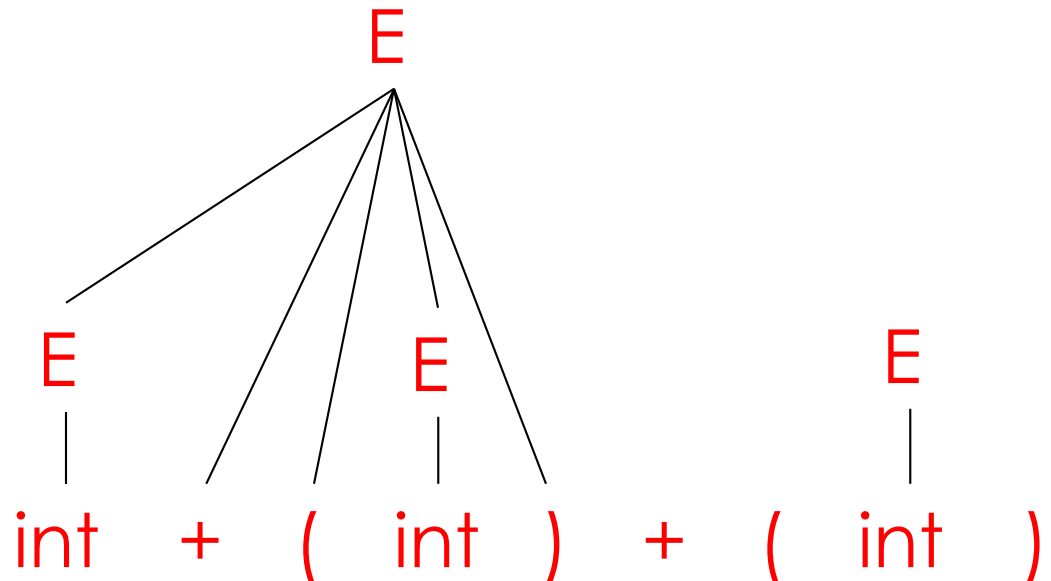
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

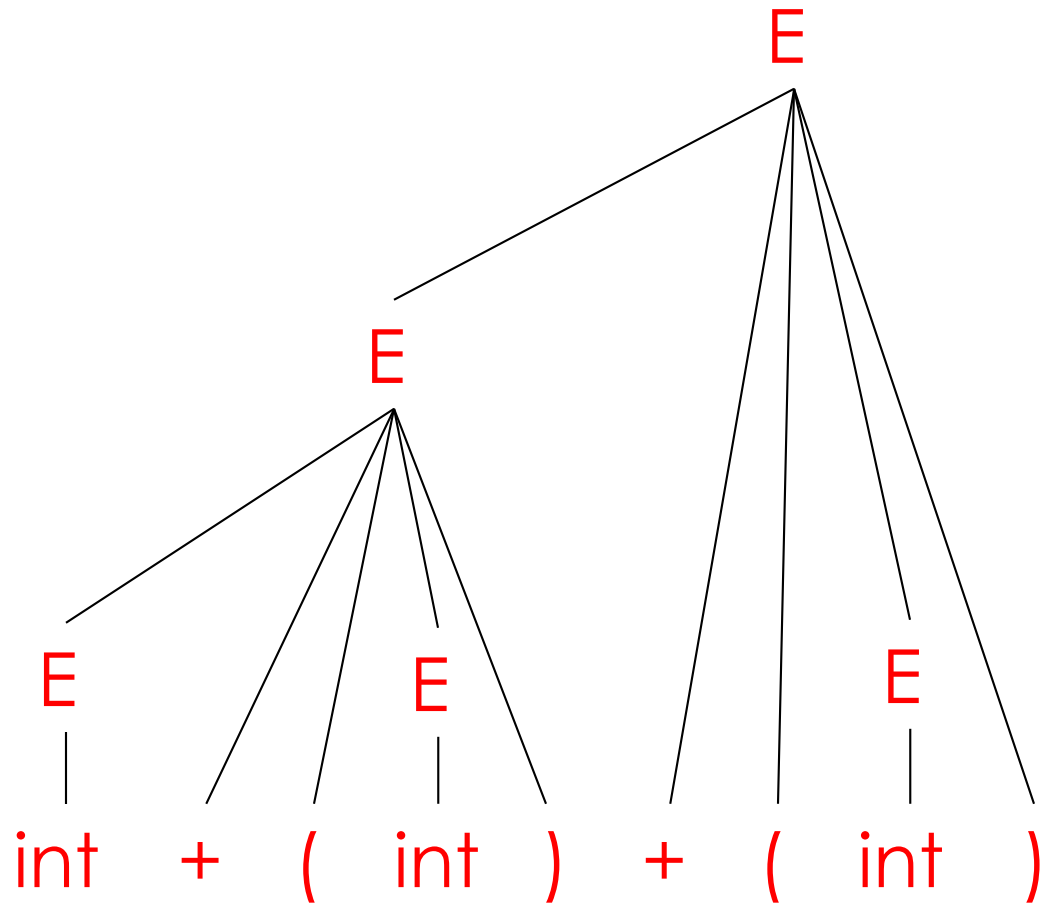
E + (E)



A Bottom-up Parse in Detail (6)

↑
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

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 β
- And the next reduction is by $A \rightarrow \beta$
- Then γ 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 **|**
 - The **|** is not part of the string
 - Marks end of next potential handle
- Initially, all input is unexamined: **|** $x_1x_2 \dots x_n$

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

$$E + (\textcolor{red}{|} \text{int}) \Rightarrow E + (\text{int} \textcolor{red}{|})$$

Reduce: Apply an inverse production at the handle.

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

$$E + (\underline{E + (E)} \textcolor{red}{|}) \Rightarrow E + (\underline{E} \textcolor{red}{|})$$

Shift-Reduce Example

| int + (int) + (int)\$ shift

int + (int) + (int)



Shift-Reduce Example

| int + (int) + (int)\$ shift

int | + (int) + (int)\$ red. $E \rightarrow int$

int + (int) + (int)

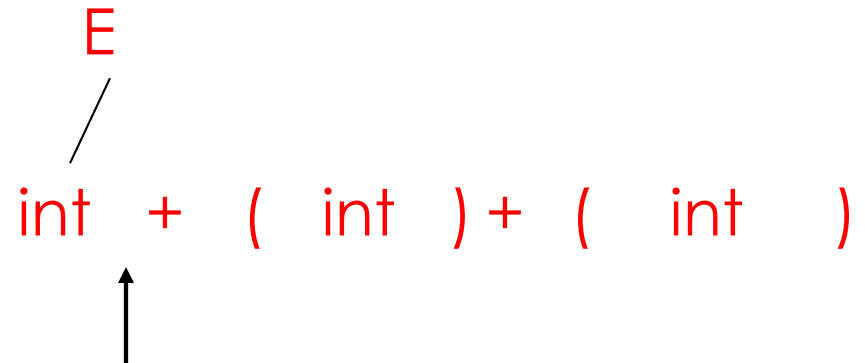


Shift-Reduce Example

| int + (int) + (int)\$ shift

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

E | + (int) + (int)\$ shift 3 times



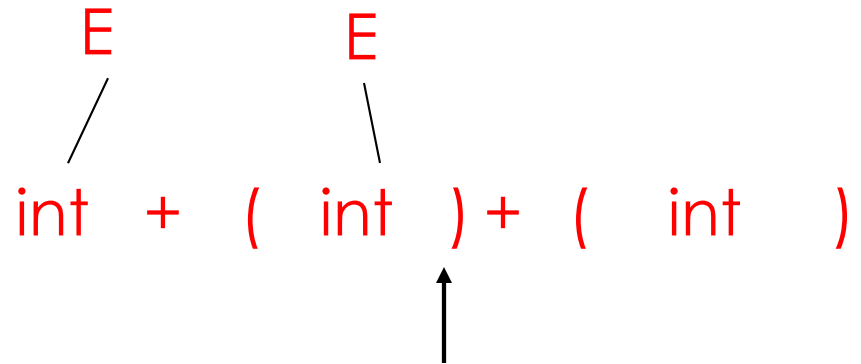
Shift-Reduce Example

| int + (int) + (int)\$ shift
int | + (int) + (int)\$ red. $E \rightarrow \text{int}$
E | + (int) + (int)\$ shift 3 times
E + (int |) + (int)\$ red. $E \rightarrow \text{int}$

E
/
int + (int) + (int)
 ↑

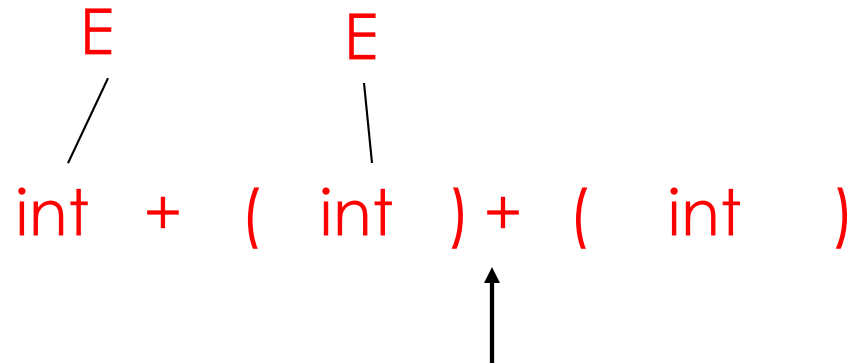
Shift-Reduce Example

| int + (int) + (int)\$ shift
int | + (int) + (int)\$ red. $E \rightarrow \text{int}$
E | + (int) + (int)\$ shift 3 times
E + (int |) + (int)\$ red. $E \rightarrow \text{int}$
E + (E |) + (int)\$ shift



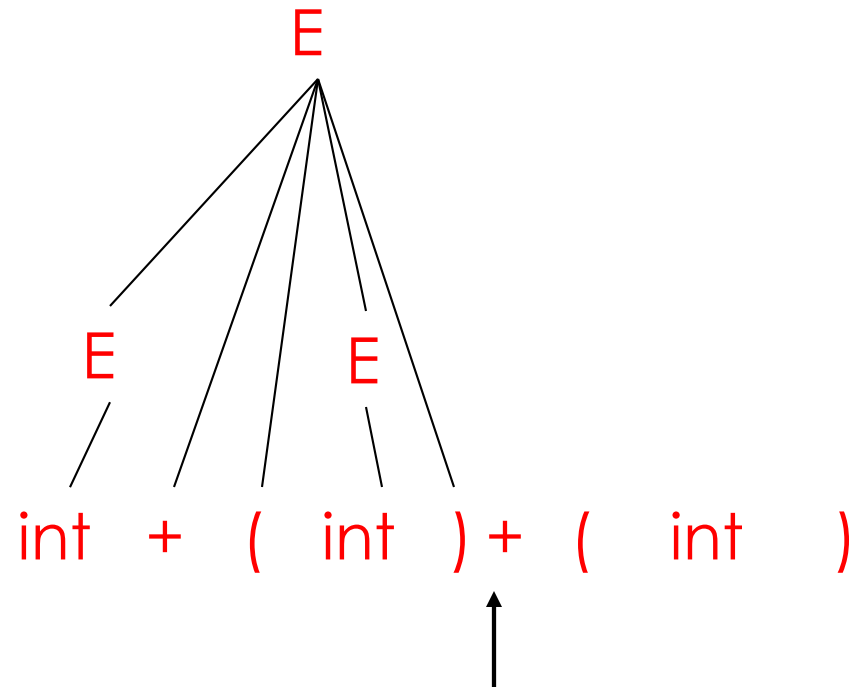
Shift-Reduce Example

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



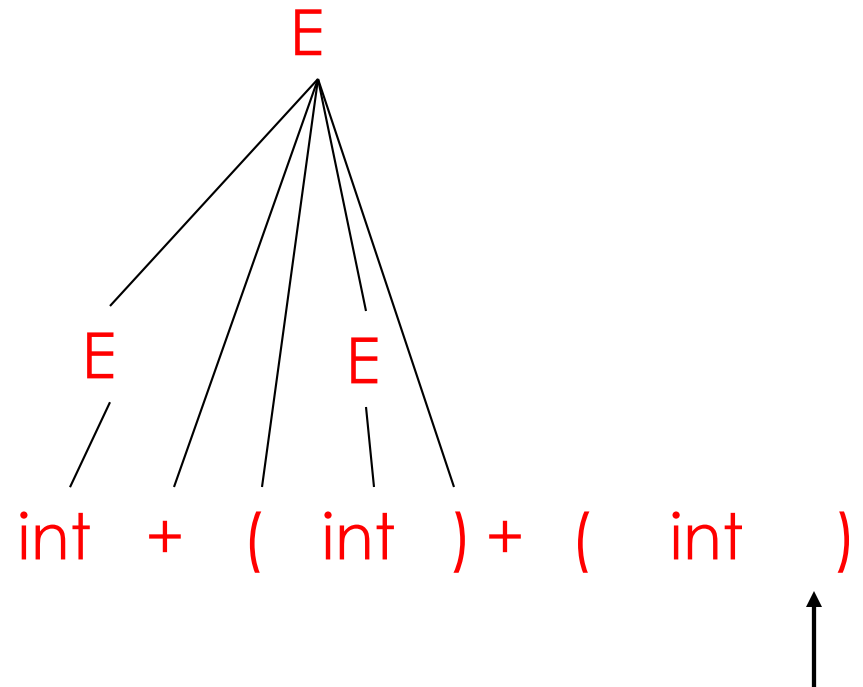
Shift-Reduce Example

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



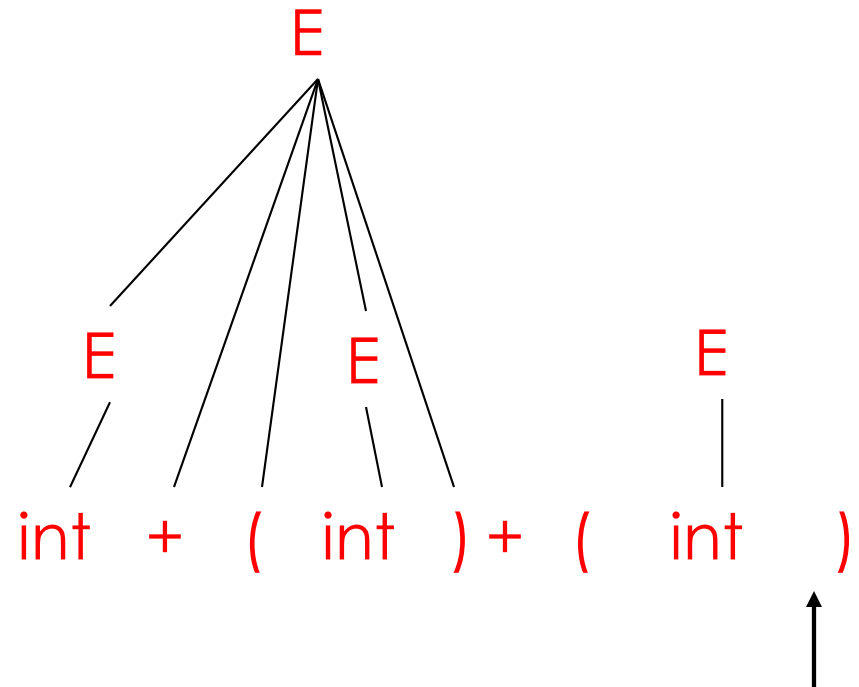
Shift-Reduce Example

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



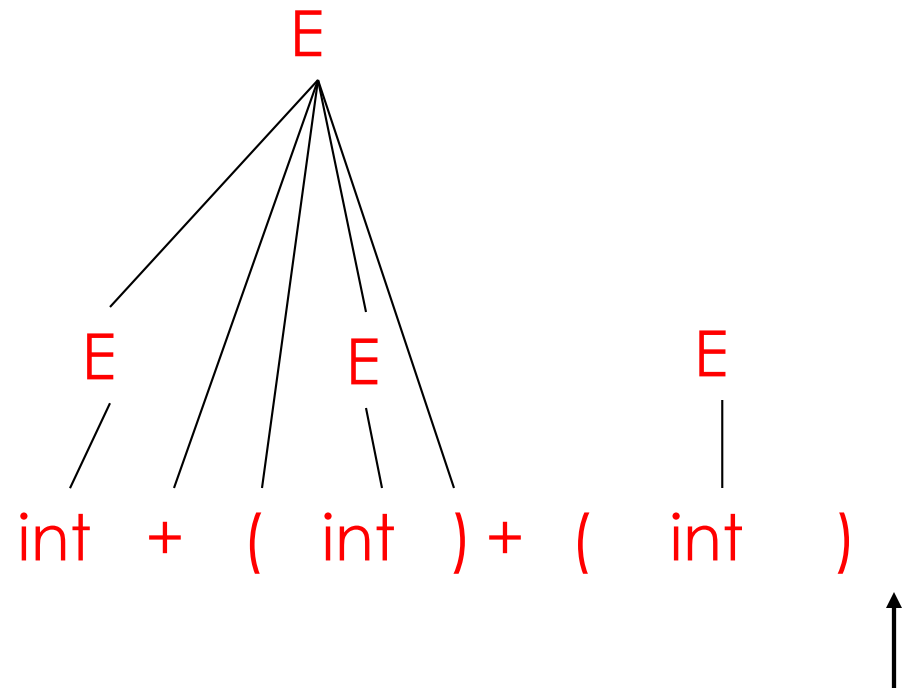
Shift-Reduce Example

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



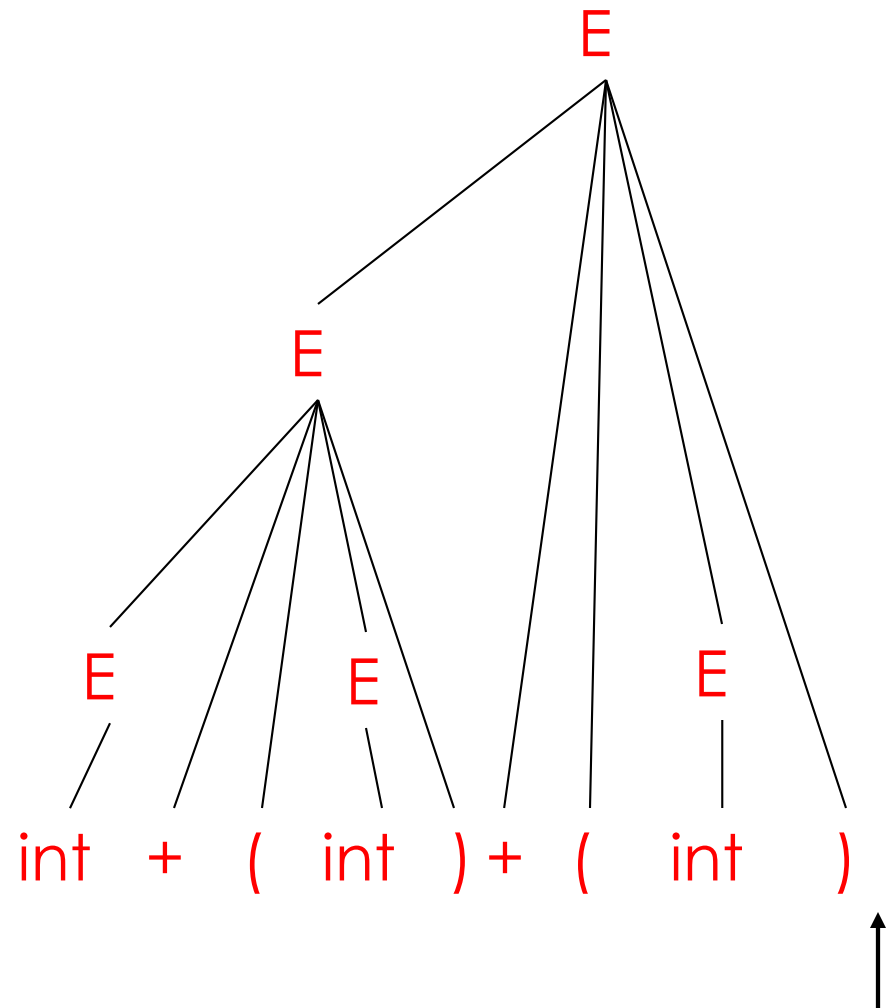
Shift-Reduce Example

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



Shift-Reduce Example

int + (int) + (int)\$	shift
int + (int) + (int)\$	red. $E \rightarrow \text{int}$
E + (int) + (int)\$	shift 3 times
E + (int) + (int)\$	red. $E \rightarrow \text{int}$
E + (E) + (int)\$	shift
E + (E) + (int)\$	red. $E \rightarrow E + (E)$
E + (int)\$	shift 3 times
E + (int)\$	red. $E \rightarrow \text{int}$
E + (E)\$	shift
E + (E) \$	red. $E \rightarrow E + (E)$
E \$	accept



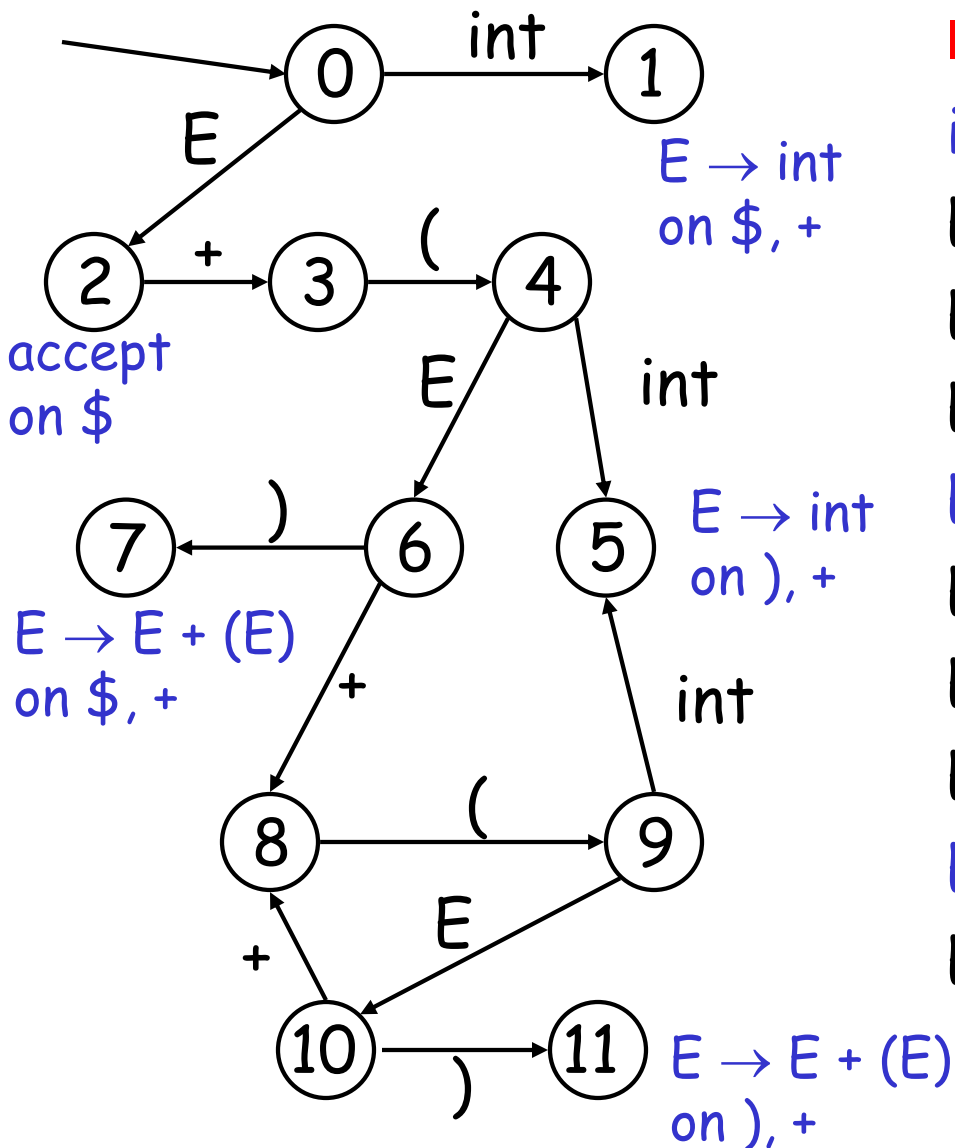
The Stack

- 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 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 $|$
 - 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



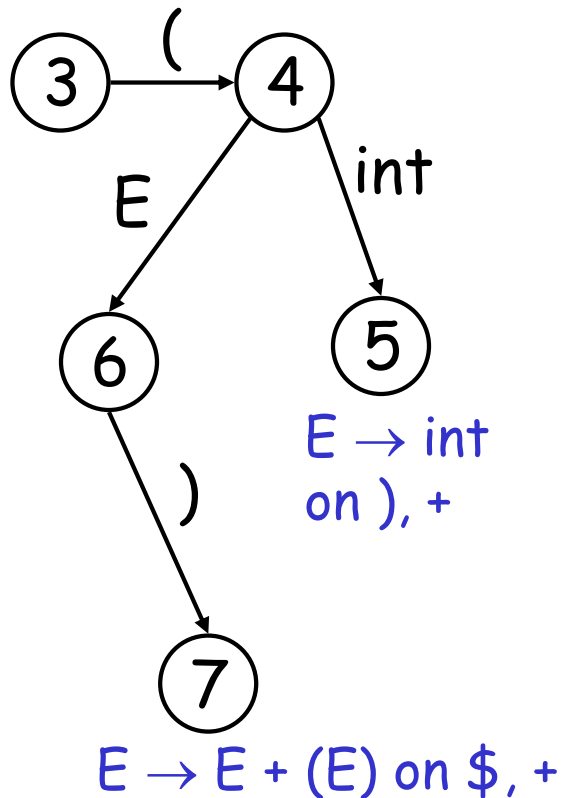
int + (int) + (int)\$	shift
int + (int) + (int)\$	$E \rightarrow \text{int}$
E + (int) + (int)\$	shift(x3)
E + (int) + (int)\$	$E \rightarrow \text{int}$
E + (E) + (int)\$	shift
E + (E) + (int)\$	$E \rightarrow E + (E)$
E + (int)\$	shift (x3)
E + (int)\$	$E \rightarrow \text{int}$
E + (E)\$	shift
E + (E) \$	$E \rightarrow E + (E)$
E \$	accept

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:



	int	+	()	\$	E
...						
3			s4			
4	s5					g6
5		$r_{E \rightarrow \text{int}}$		$r_{E \rightarrow \text{int}}$		
6		s8		s7		
7		$r_{E \rightarrow E+(E)}$			$r_{E \rightarrow E+(E)}$	
...						

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
$$\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$$
$$\text{state}_k \text{ is the final state of the DFA on } \text{sym}_1 \dots \text{sym}_k$$

The LR Parsing Algorithm

Let $I = w_1w_2...w_n\$$ be initial input

Let $j = 1$

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 $\langle I[j], k \rangle$; $j += 1$

reduce $X \rightarrow \alpha$:

pop $|\alpha|$ pairs,

push $\langle X, Goto[top_state(stack), X] \rangle$

accept: halt normally

error: halt and report error

Parsing Contexts

- Consider the state describing the situation at the **|** in the stack
$$E \quad + \quad (\textcolor{red}{|} \text{int} \) \ + \ (\quad \text{int} \quad)$$
- Context:
 - We are looking for an $E \rightarrow E + (\bullet E)$
 - Have seen $E + ($ from the right-hand side
 - We are also looking for $E \rightarrow \bullet \text{int}$ or $E \rightarrow \bullet E + (E)$
 - Have seen nothing from the right-hand side
- One DFA state describes a set of such contexts
- (Traditionally, use \bullet to show where the **|** is.)

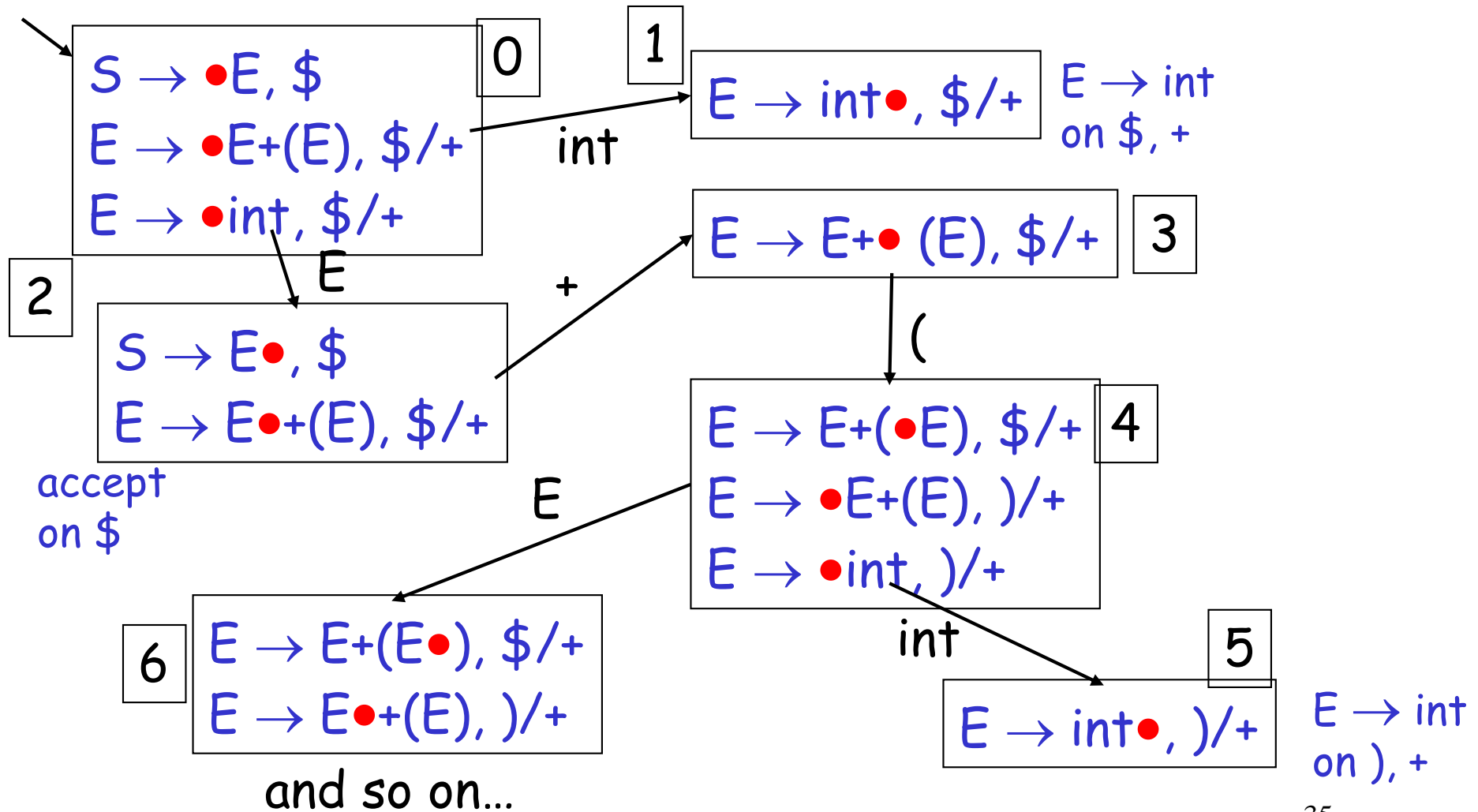
LR(1) Items

- An $LR(1)$ item is a pair:
$$X \rightarrow \alpha \bullet \beta, a$$
 - $X \rightarrow \alpha \beta$ is a production
 - a is a terminal (the lookahead terminal)
 - $LR(1)$ means 1 lookahead terminal
- $[X \rightarrow \alpha \bullet \beta, a]$ describes a context of the parser
 - We are trying to find an X followed by an a , and
 - We have α already on top of the stack
 - Thus we need to see next a prefix derived from βa

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:
 - $S \rightarrow \bullet E, \$$
 - Trying to find an S as a string derived from $E\$$
 - The stack is empty

Constructing the Parsing DFA. Example.



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
 $[X \rightarrow \alpha \bullet a \beta, b]$ and $[Y \rightarrow \gamma \bullet, a]$
- Then on input "a" we could either
 - Shift into state $[X \rightarrow \alpha a \bullet \beta, b]$, 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
$$[S \rightarrow \text{if } E \text{ then } S \bullet, \quad \text{else}]$$
$$[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, \quad \$]$$
- If **else** follows then we can shift or reduce

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

$$E \rightarrow E + E \mid E * E \mid \text{int}$$

- We will have the states containing

$$[E \rightarrow E * \bullet E, +] \qquad [E \rightarrow E * E \bullet, +]$$

$$[E \rightarrow \bullet E + E, +] \Rightarrow^E [E \rightarrow E \bullet + E, +]$$

...

...

- Again we have a shift/reduce on input +
 - We need to reduce ($*$ binds more tightly than $+$)
 - Solution: declare the precedence of $*$ 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:

$$\begin{array}{cc} [E \rightarrow E * \bullet E, +] & [E \rightarrow E * E \bullet, +] \\ [E \rightarrow \bullet E + E, +] \Rightarrow^E & [E \rightarrow E \bullet + E, +] \\ \dots & \dots \end{array}$$

- Will choose reduce because precedence of rule $E \rightarrow E * E$ is higher than of terminal $+$

Using Precedence to Solve S/R Conflicts

- Same grammar as before

$$E \rightarrow E + E \mid E * E \mid \text{int}$$

- We will also have the states

$$\begin{array}{ccc} [E \rightarrow E + \bullet E, +] & & [E \rightarrow E + E \bullet, +] \\ [E \rightarrow \bullet E + E, +] & \Rightarrow^E & [E \rightarrow E \bullet + E, +] \\ \dots & & \dots \end{array}$$

- 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
 $[S \rightarrow \text{if } E \text{ then } S \bullet, \quad \text{else}]$
 $[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, \quad x]$
- Can eliminate conflict by declaring *else* with higher precedence than *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
 $[X \rightarrow \alpha \bullet, a]$ and $[Y \rightarrow \beta \bullet, a]$
 - 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?

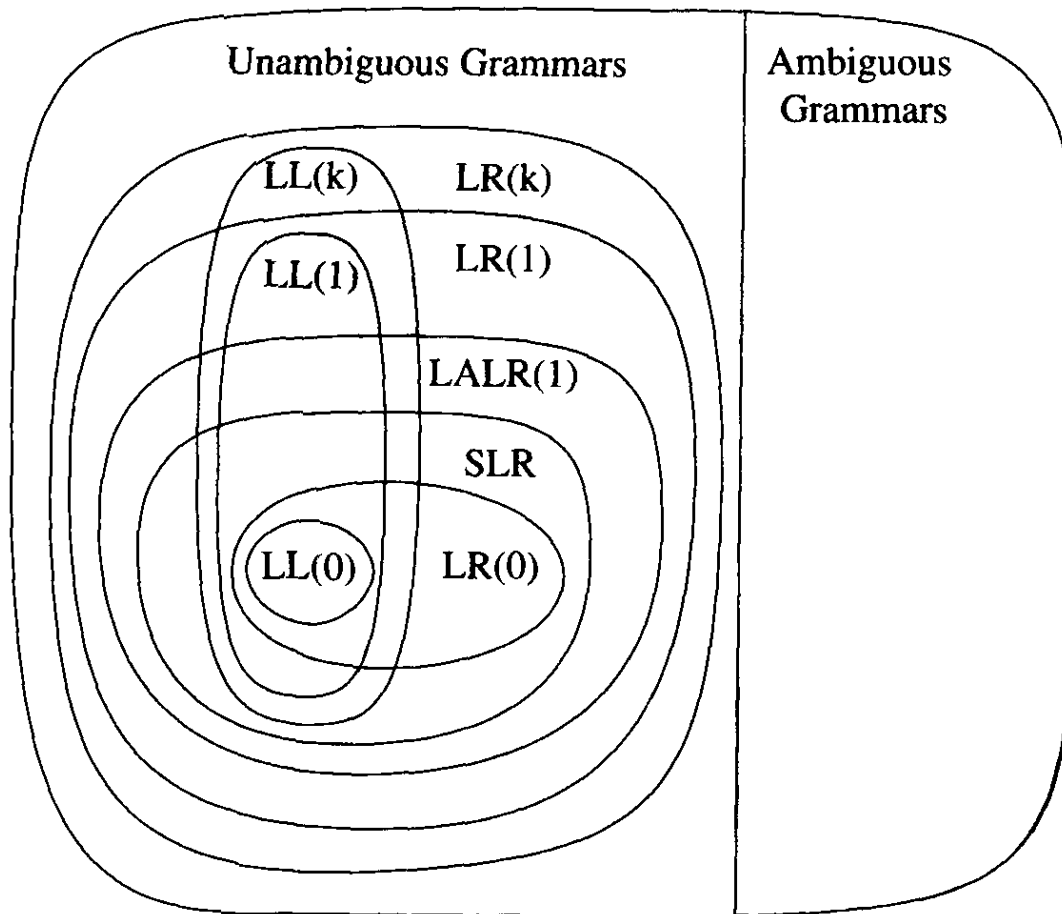
More on Reduce/Reduce Conflicts

- Consider the states
 - $[S \rightarrow id \bullet, \$]$
 - $[S' \rightarrow \bullet S, \$]$
 - $[S \rightarrow id \bullet S, \$]$
 - $[S \rightarrow \bullet, \$] \Rightarrow_{id} [S \rightarrow \bullet, \$]$
 - $[S \rightarrow \bullet id, \$]$
 - $[S \rightarrow \bullet id S, \$]$
- Reduce/reduce conflict on input $\$$
 - $S' \rightarrow S \rightarrow id$
 - $S' \rightarrow S \rightarrow id S \rightarrow id$
- Better rewrite the grammar: $S \rightarrow \epsilon \mid id S$

Relation to Bison

- Bison builds this kind of machine.
- However, for efficiency concerns, collapses many of the states together.
- Causes some additional conflicts, but not many.
- The machines discussed here are LR(1) engines. Bison's optimized versions are LALR(1) engines.

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
 - But let's also see a more powerful engine.

Assignment

$G(S):$

$S \rightarrow [B$

$A \rightarrow \text{int} \mid [B$

$B \rightarrow] \mid C$

$C \rightarrow A] \mid A, C$