Bottom-Up Parsing

Lecture 8

(From slides by G. Necula & R. Bodik)

Administrivia

- **Test I** during class on 10 March.
- Notes updated (at last)

Bottom-Up Parsing

- We've been looking at general context-free parsing.
- It comes at a price, measured in overheads, so in practice, we design programming languages to be parsed by less general but faster means, like top-down recursive descent.
- Deterministic bottom-up parsing is more general than top-down parsing, and just as efficient.
- 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: int + (int) + (int)

The Idea

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

```
sent ← input string of terminals
while $sent \neq S$:
    – Identify first $\beta$ in sent such that $A \to \beta$ is a production and $S \to^* \alpha A \gamma \to \alpha \beta \gamma = sent$
    – Replace $\beta$ by $A$ in sent (so $\alpha A \gamma$ becomes new sent)
```

- Such $\alpha\ \beta$'s are called *handles*

A Bottom-up Parse in Detail (1)

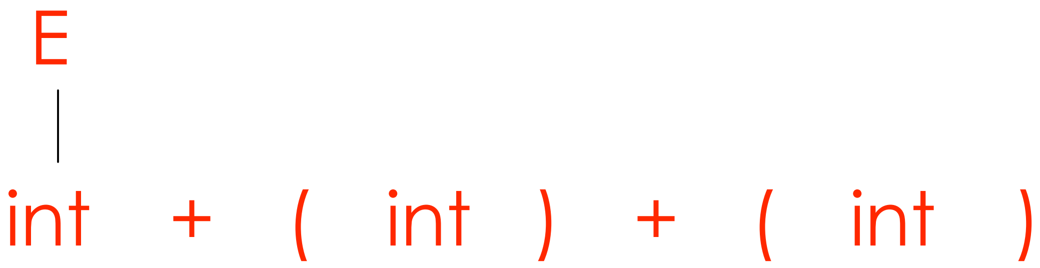
int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

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

(handles in red)

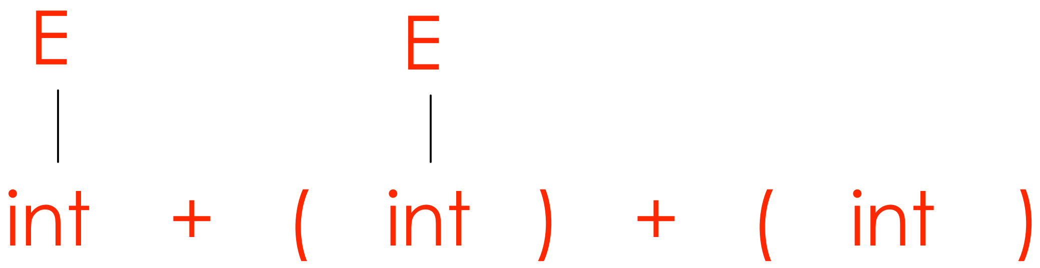


A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



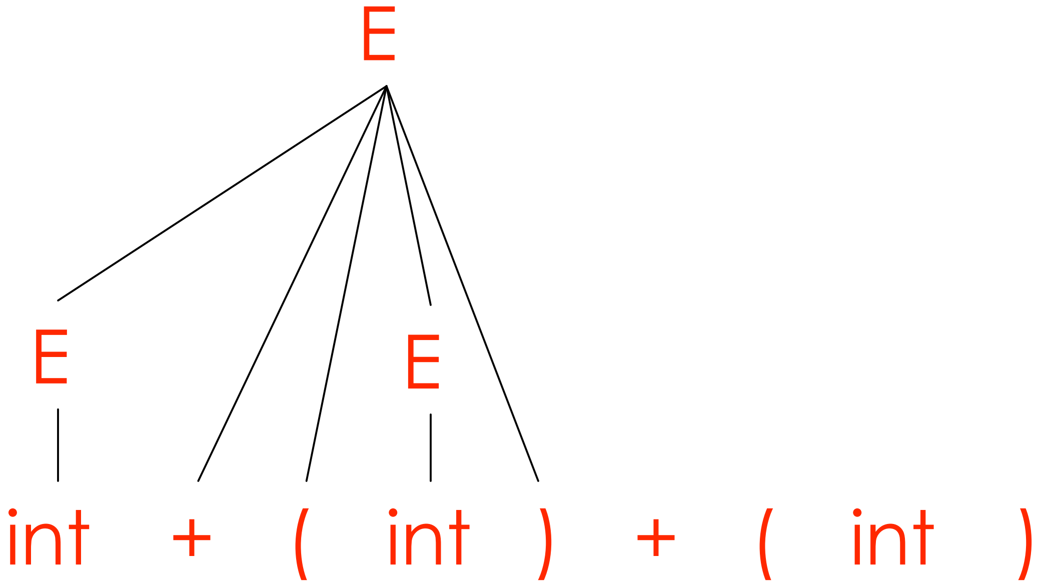
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

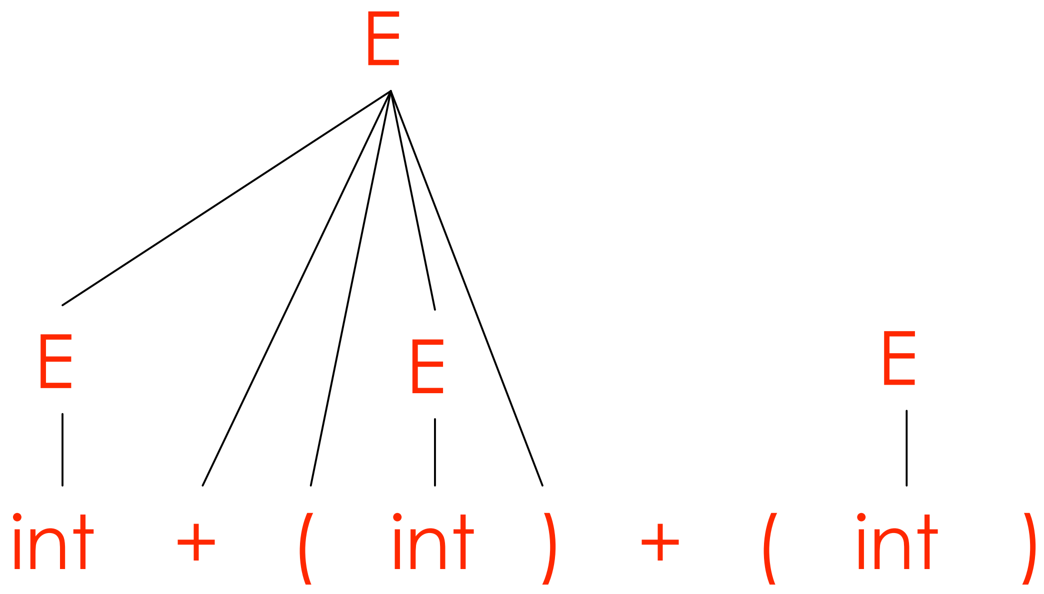
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

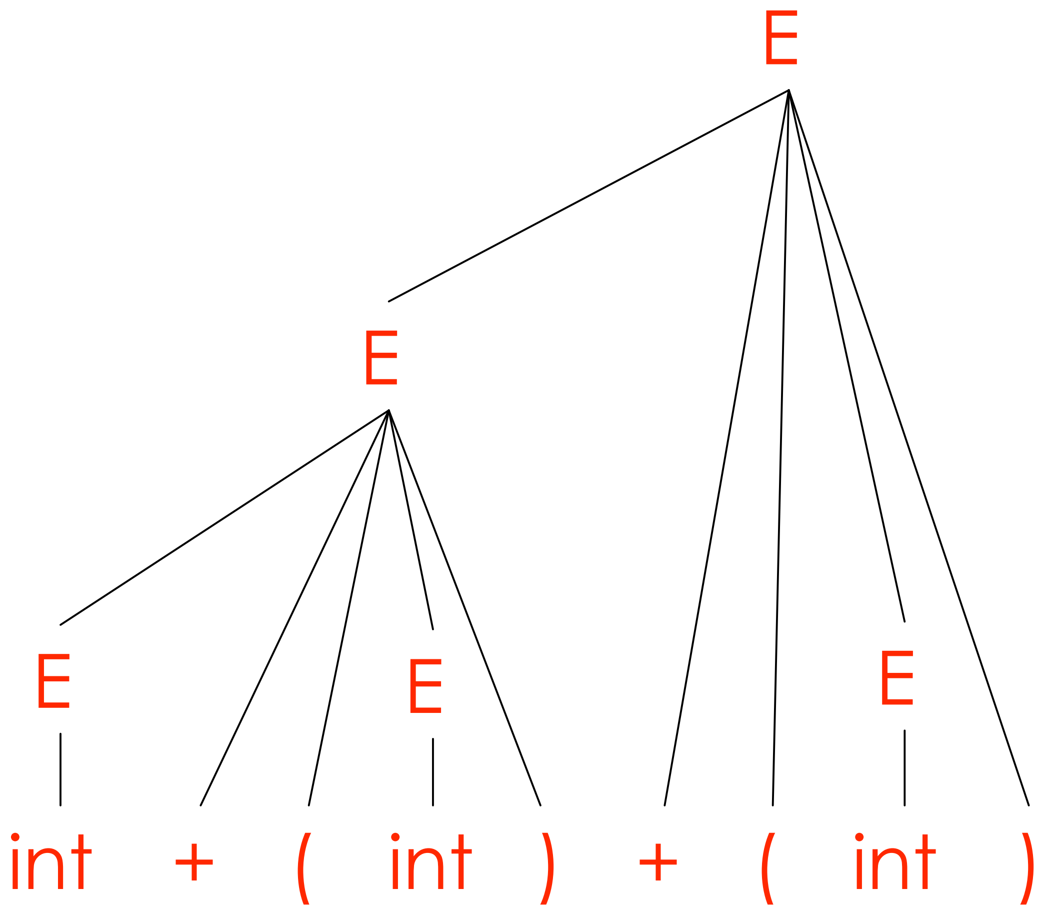
E + (E)



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 $\alpha\beta$
- 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

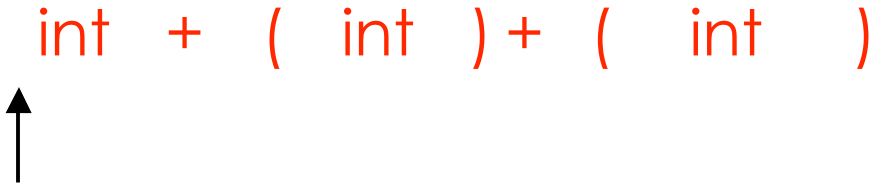
    $$E + (\mid \text{int}) \Rightarrow E + (\text{int} \mid)$$

    Reduce: Apply an inverse production at the handle. If $E \rightarrow E + (E)$ is a production, then

    $$E + (\underline{E + (E)} \mid) \Rightarrow E + (\underline{E} \mid)$$

Shift-Reduce Example

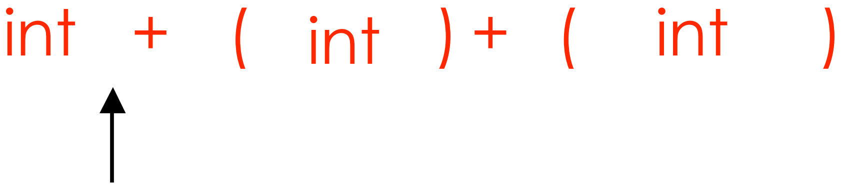
| int + (int) + (int)$ shift



Shift-Reduce Example

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

int | + (int) + (int)$ red. E → int

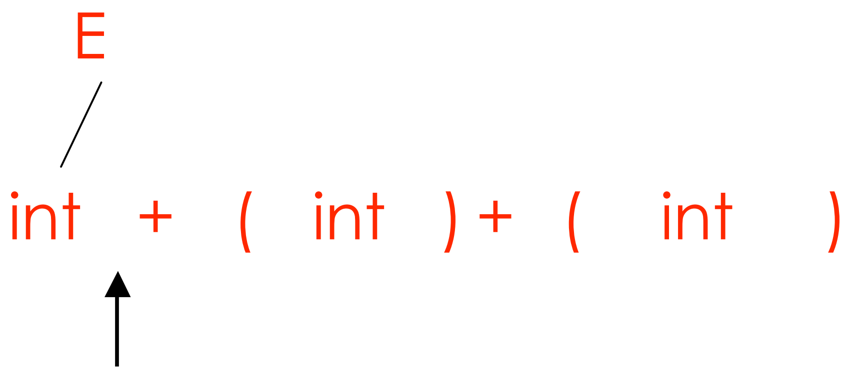


Shift-Reduce Example

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

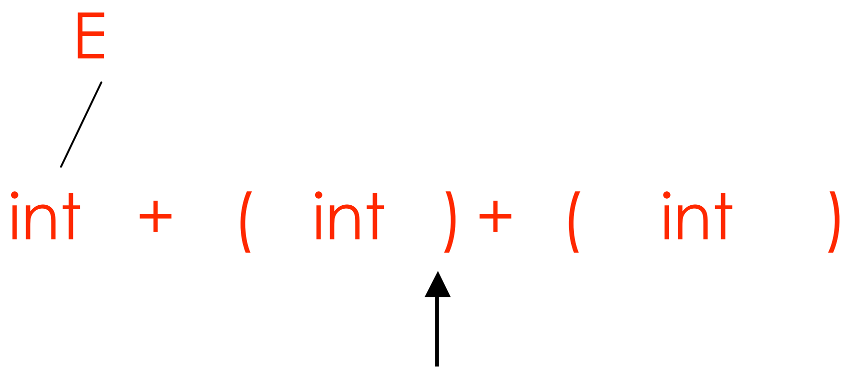
int | + (int) + (int)$ red. E → int

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



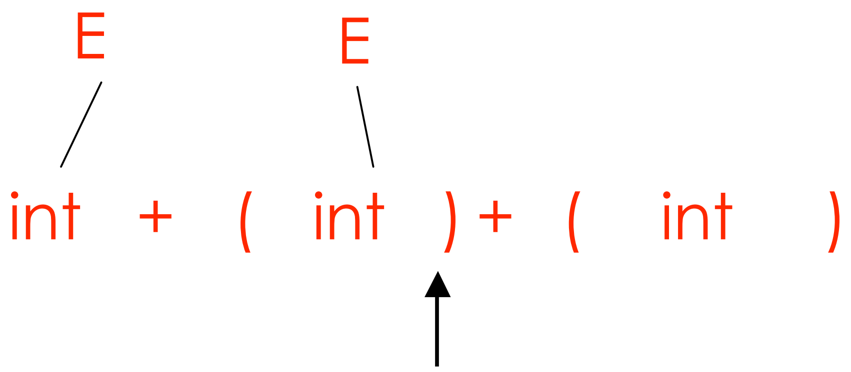
Shift-Reduce Example

I 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



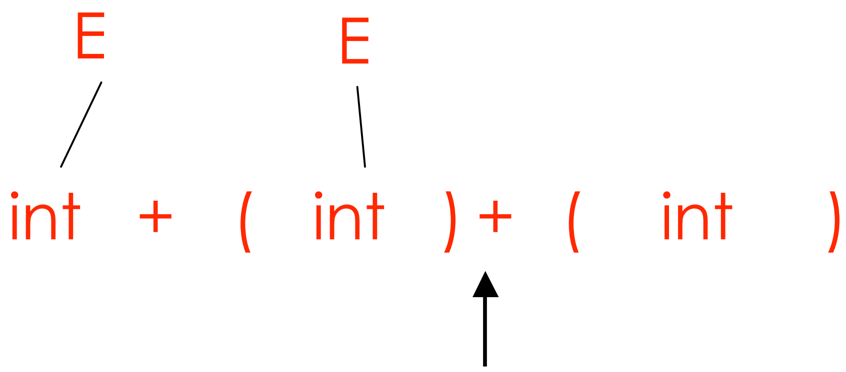
Shift-Reduce Example

| 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



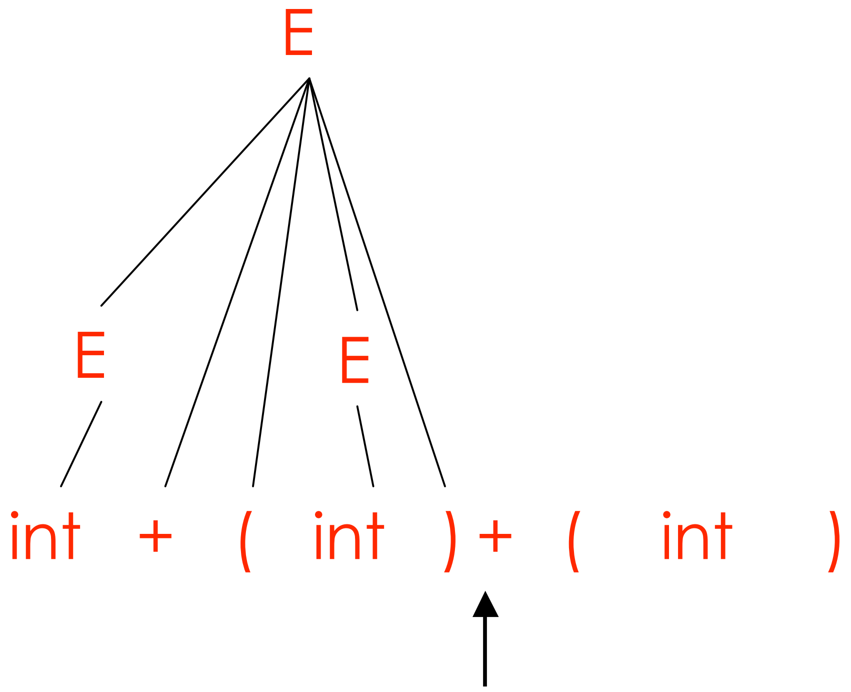
Shift-Reduce Example

| 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)



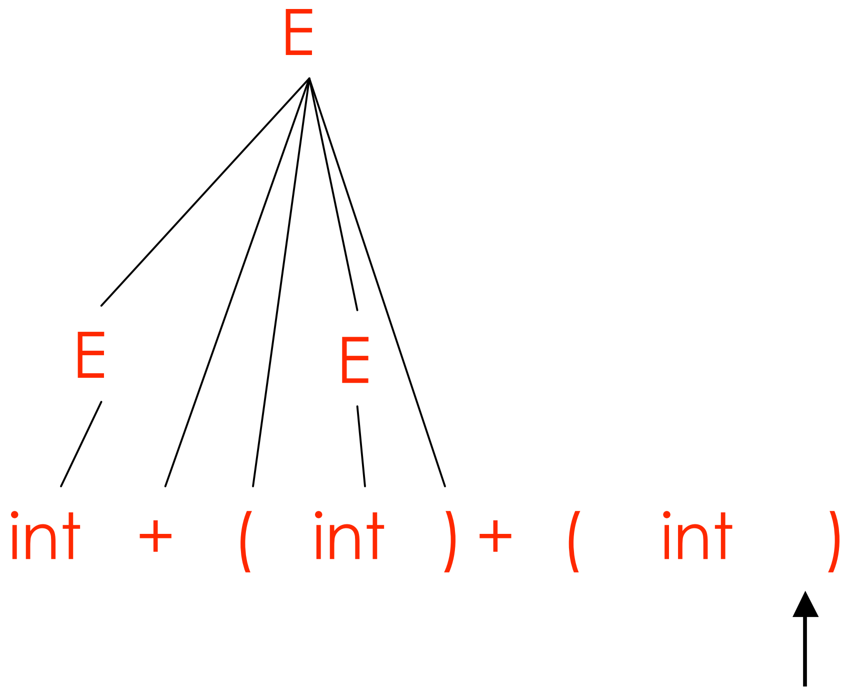
Shift-Reduce Example

| 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



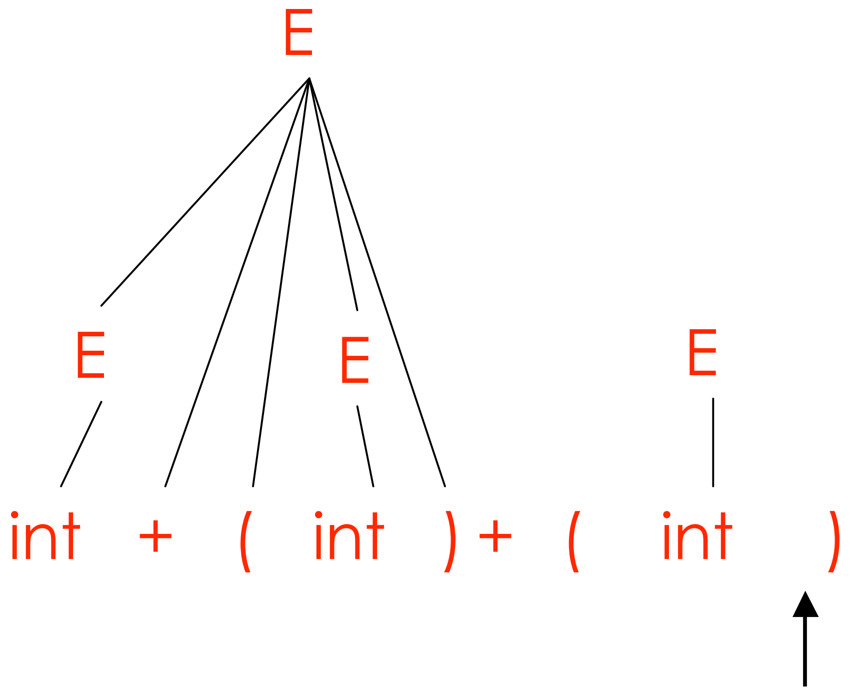
Shift-Reduce Example

∣ 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



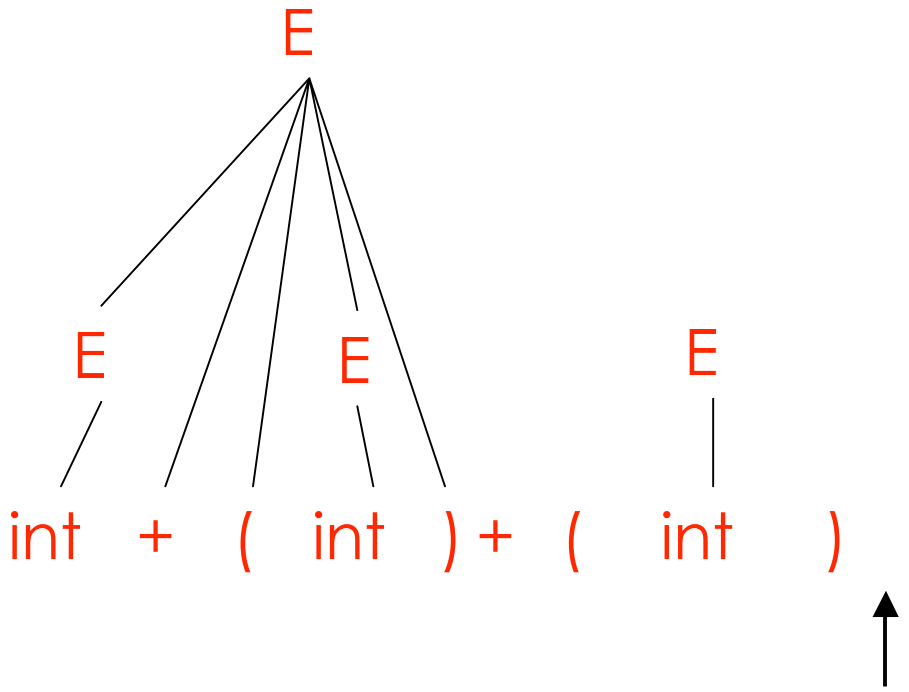
Shift-Reduce Example

| 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



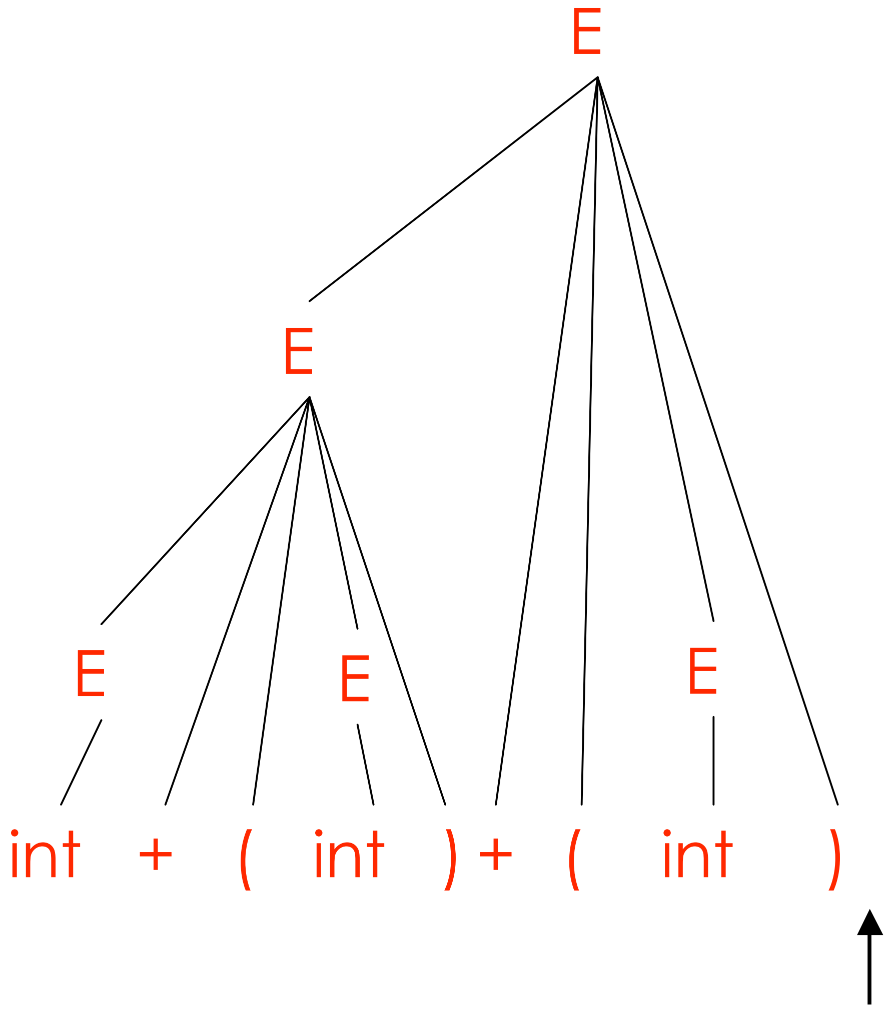
Shift-Reduce Example

| 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-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

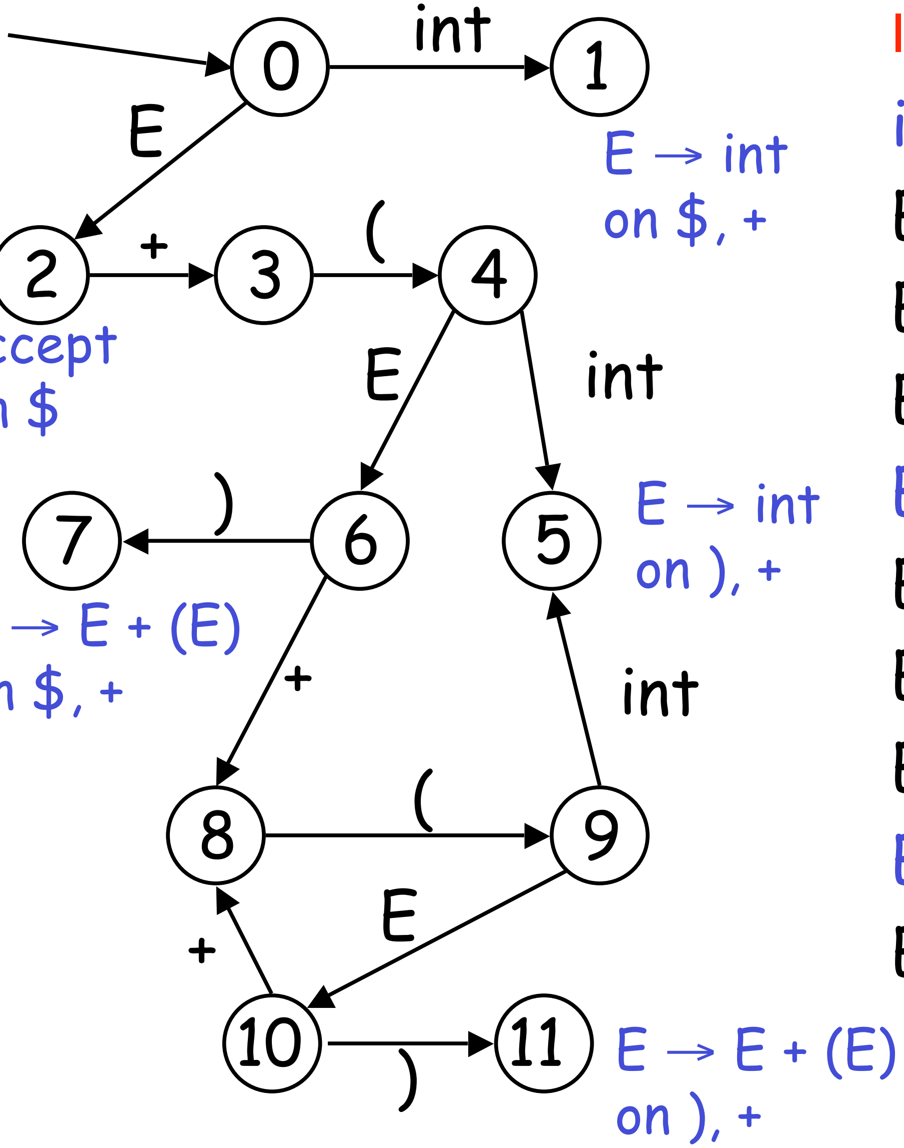
- Left string can be implemented as a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

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



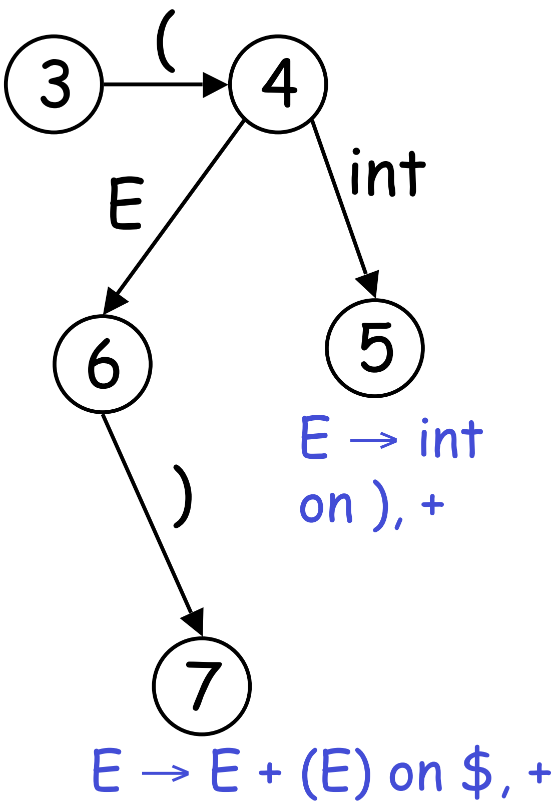
| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift(x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

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					s6
5		$r_{E \rightarrow int}$		$r_{E \rightarrow int}$		
6				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

$$\langle\ \mathrm{sym}_1,\ \mathrm{state}_1\ \rangle\ .\ .\ .\ \langle\ \mathrm{sym}_n,\ \mathrm{state}_n\ \rangle$$

state_k is the final state of the DFA on $\mathrm{sym}_1 \ldots \mathrm{sym}_k$

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 = ⟨ dummy, 0 ⟩
   repeat
        case table[top_state(stack), I[j]] of
                shift k:  push ⟨ I[j], k ⟩; j += 1
                reduce X → α:
                    pop |α| pairs,
                    push ⟨X, table[top_state(stack), X]⟩
                accept: halt normally
                error: halt and report error
```

Parsing Contexts

- Consider the state describing the situation at the | in the stack E + (| int) + (int)
- Context:
  - We are looking for an E → E + (• E)
    - Have have seen E + (from the right-hand side
  - We are also looking for E → • int *or* E → • E + (E)
    - Have seen nothing from the right-hand side
- One DFA state describes a set of such contexts
- (Traditionally, use • 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

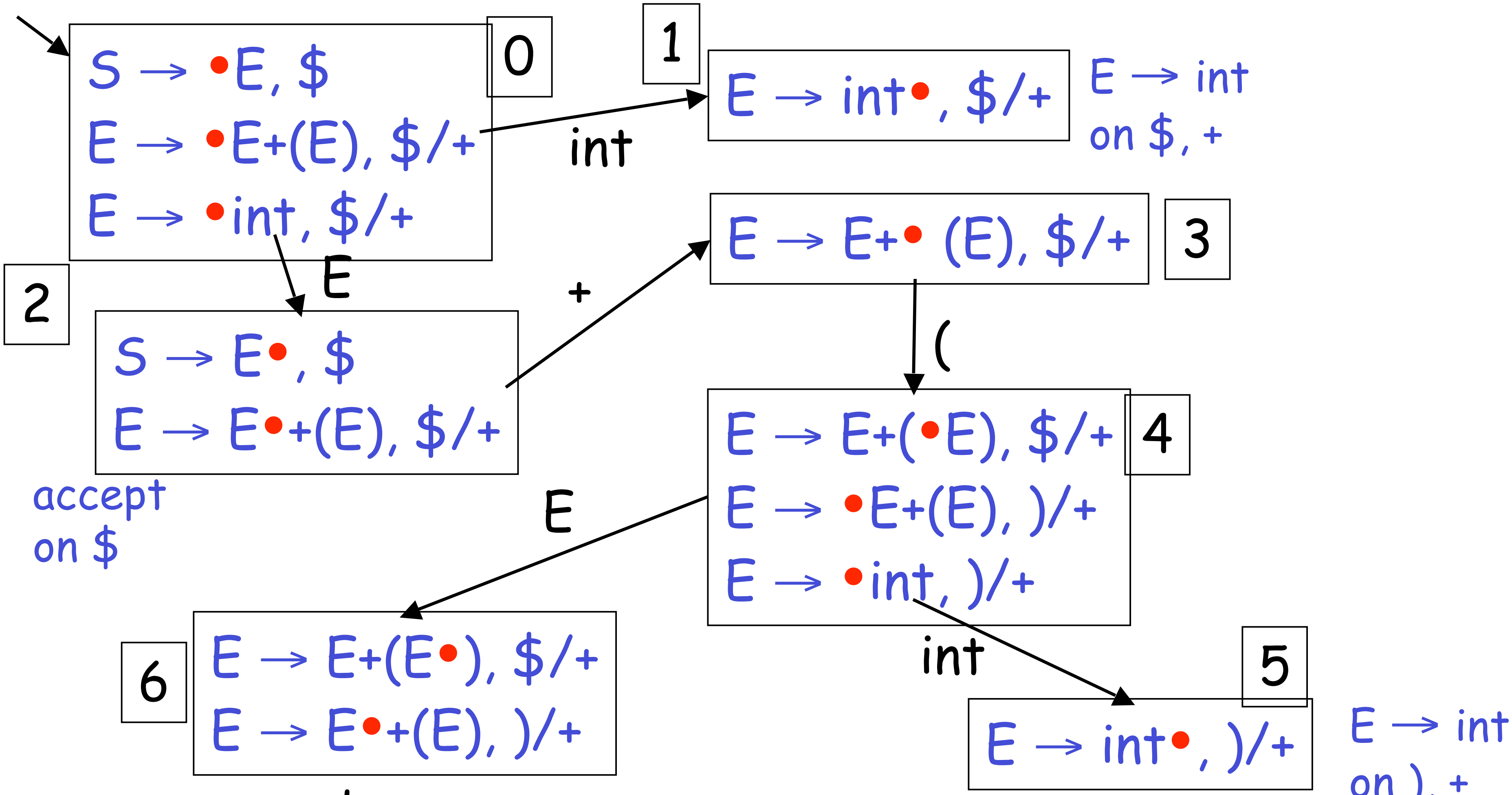
- We add to our grammar a fresh new start symbol S and a production S → E
  - Where E is the old start symbol
  - No need to do this if E had only one production

- 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.



0

S → •E, $

E → •E+(E), $/+

E → •int, $/+

int

1

E → int•, $/+

E → int

on $, +

E

2

S → E•, $

E → E•+(E), $/+

accept

on $

\+

3

E → E+• (E), $/+

(

4

E → E+(•E), $/+

E → •E+(E),)/+

E → •int,)/+

E

6

E → E+(E•), $/+

E → E•+(E),)/+

int

5

E → int•,)/+

E → int

on), +

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

  $[X \to \alpha \bullet a\beta, b]$ and $[Y \to \gamma\bullet, a]$

- Then on input "a" we could either
  - Shift into state $[X \to \alpha a \bullet \beta, b]$, or
  - Reduce with $Y \to \gamma$

- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else

  S → if E then S | if E then S else S | OTHER

- Will have DFA state containing

  [S → if E then S•, else]

  [S → if E then S• else S, $]

- 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

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

$$[E \rightarrow \bullet E + E, \; +] \quad \xrightarrow{E} \quad [E \rightarrow E\bullet + E, \; +]$$

$$\ldots \qquad\qquad\qquad\qquad \ldots$$

- 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:

$[E \to E * \bullet E, +]$ $[E \to E * E \bullet, +]$

$[E \to \bullet E + E, +] \overset{E}{\Rightarrow} [E \to E \bullet + E, +]$

… …

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

Using Precedence to Solve S/R Conflicts

- Same grammar as before

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

- We will also have the states

$$[E \to E + \bullet E, +] \qquad\qquad [E \to E + E\bullet, +]$$

$$[E \to \bullet E + E, +] \quad \Rightarrow^{E} \quad [E \to E \bullet + E, +]$$

$$\ldots \qquad\qquad\qquad\qquad \ldots$$

- Now we also have a shift/reduce on input $+$
  - We choose reduce because $E \to E + E$ and $+$ have the same precedence and $+$ is left-associative

Using Precedence to Solve S/R Conflicts

- Back to our dangling else example

  [S → if E then S•,   else]

  [S → if E then S• else S,   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

$$\begin{array}{lcl}
 & & [S \rightarrow \mathrm{id} \bullet, \quad \$] \\
[S' \rightarrow \bullet\, S, \quad \$] & & [S \rightarrow \mathrm{id} \bullet S, \quad \$] \\
[S \rightarrow \bullet, \quad \$] & \Rightarrow^{\mathrm{id}} & [S \rightarrow \bullet, \quad \$] \\
[S \rightarrow \bullet\, \mathrm{id}, \quad \$] & & [S \rightarrow \bullet\, \mathrm{id}, \quad \$] \\
[S \rightarrow \bullet\, \mathrm{id}\ S, \quad \$] & & [S \rightarrow \bullet\, \mathrm{id}\ S, \quad \$]
\end{array}$$

- Reduce/reduce conflict on input $\$$

$$S' \rightarrow S \rightarrow \mathrm{id}$$

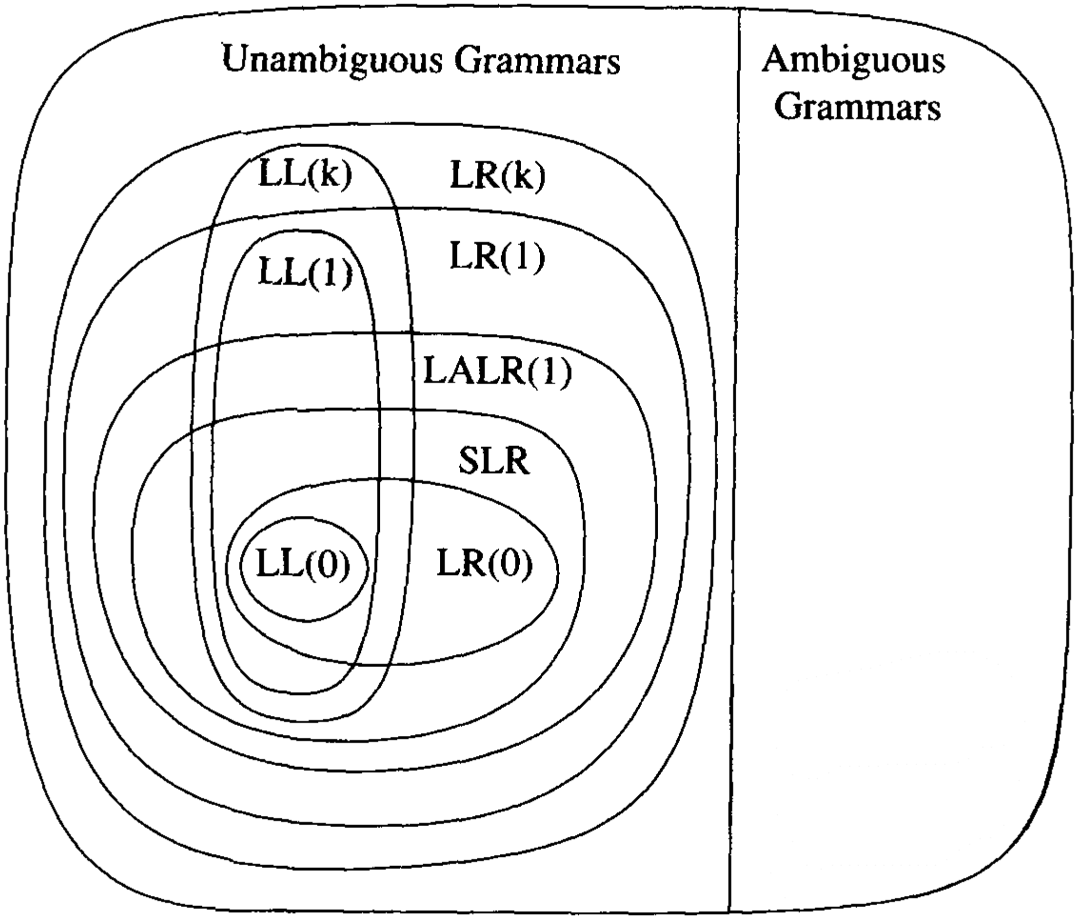
$$S' \rightarrow S \rightarrow \mathrm{id}\ S \rightarrow \mathrm{id}$$

- Better rewrite the grammar: $S \rightarrow \varepsilon \mid \mathrm{id}\ S$

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 simple parser: LL(1), recursive descent
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - We use LALR(1) parser generators
  - Earley's algorithm provides a complete algorithm for parsing context-free languages.