

LR Parsing. Parser Generators.

Lecture 8-9

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:

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

repeat

- Identify β in str such that $A \rightarrow \beta$ is a production (i.e., $\text{str} = \alpha \beta \gamma$)
- Replace β by A in str (i.e., str becomes $\alpha A \gamma$)

until $\text{str} = S$

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)

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)

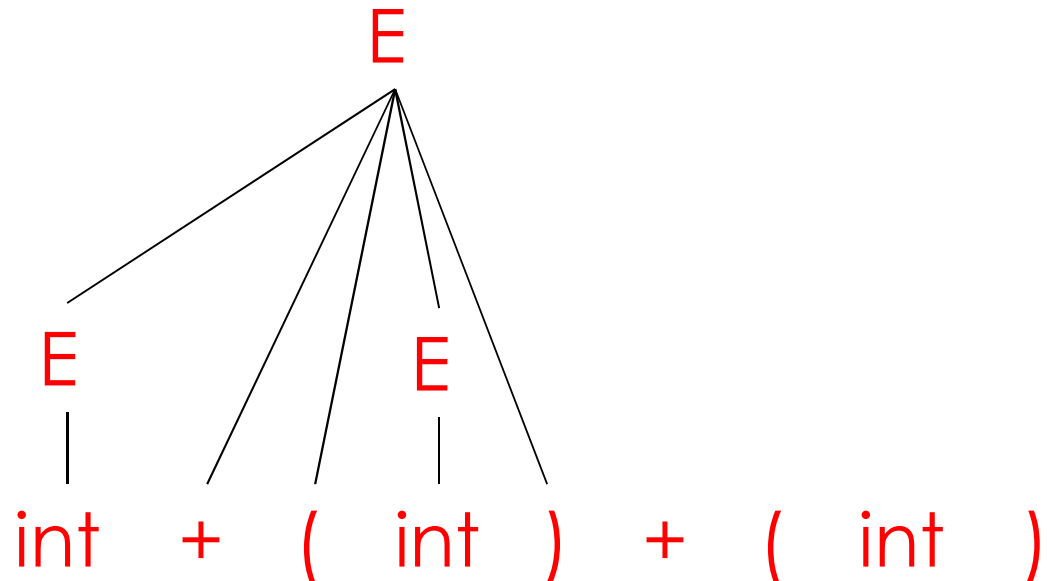
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

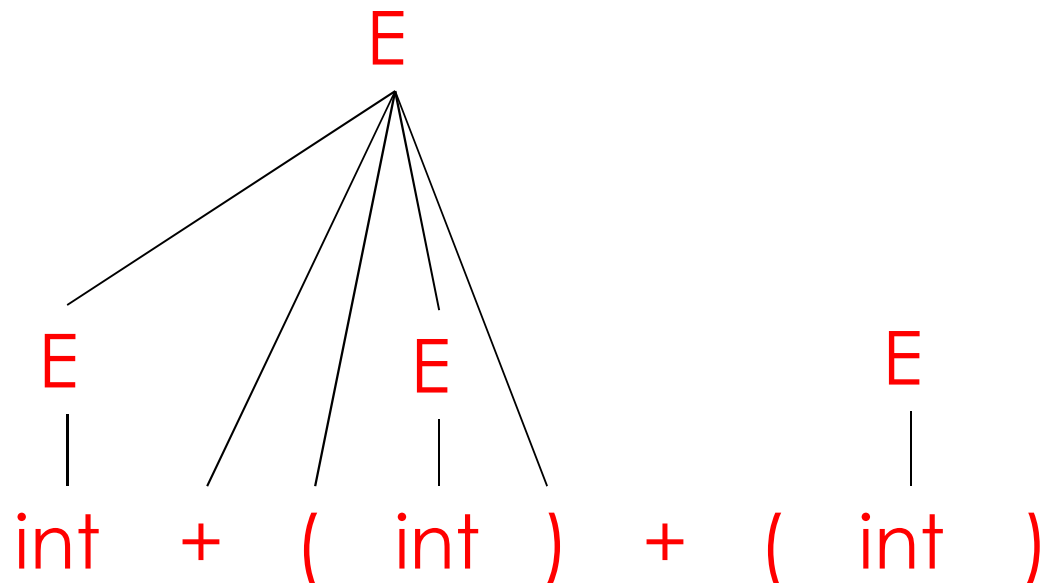
int + (int) + (int)

E + (int) + (int)

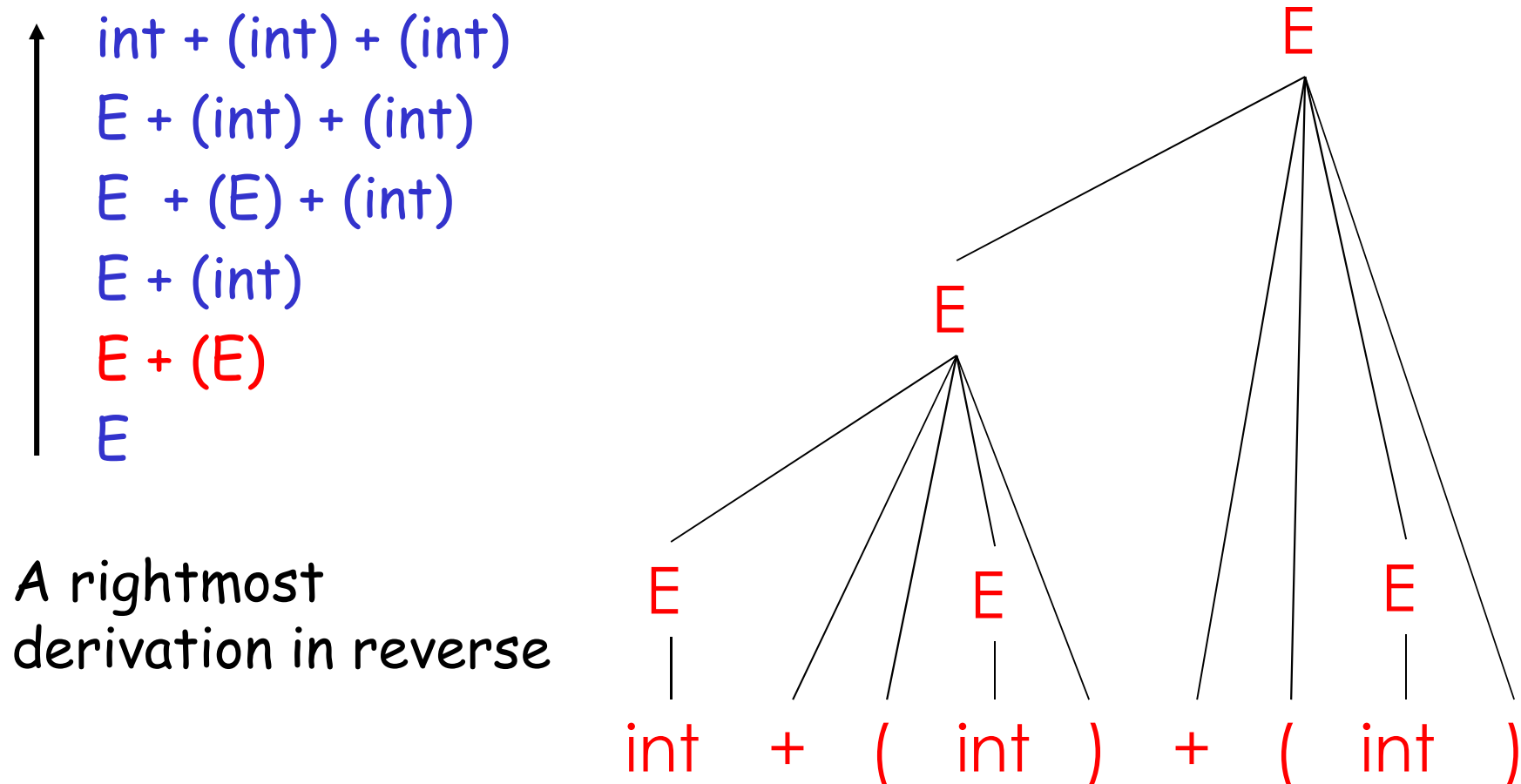
E + (E) + (int)

E + (int)

E + (E)



A Bottom-up Parse in Detail (6)



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 γ 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 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: ► $x_1x_2 \dots x_n$

Shift-Reduce Parsing

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

Shift

Reduce

Shift

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

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

Reduce

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

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

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

Shift-Reduce Example

► `int + (int) + (int)$` *shift*

`int + (int) + (int)`

Shift-Reduce Example

► int + (int) + (int)\$ shift

int ► + (int) + (int)\$ red. E → int

int + (int) + (int)



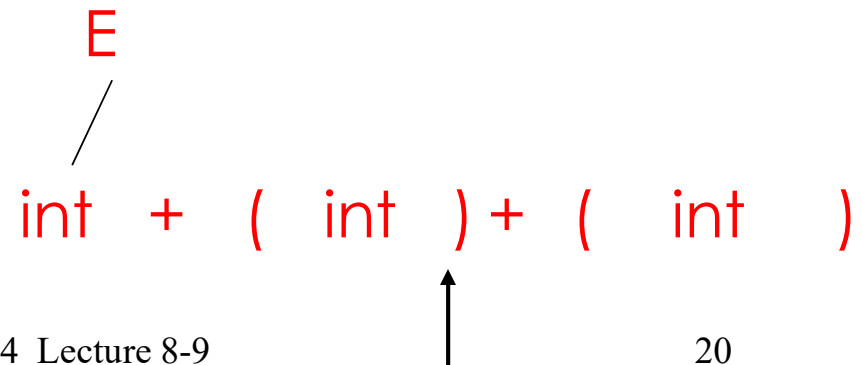
Shift-Reduce Example

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

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

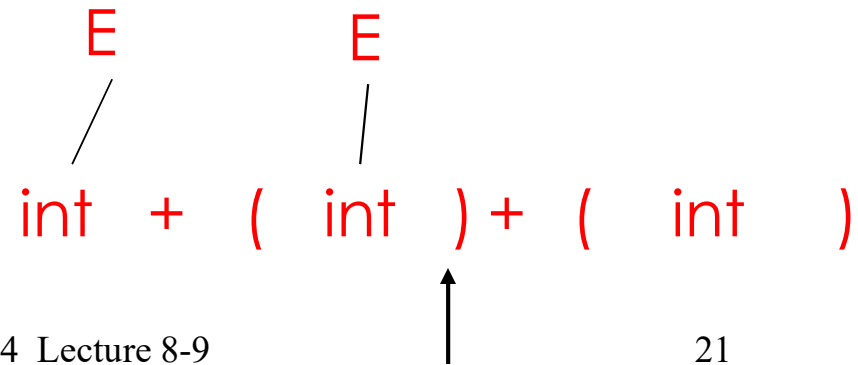
Shift-Reduce Example

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

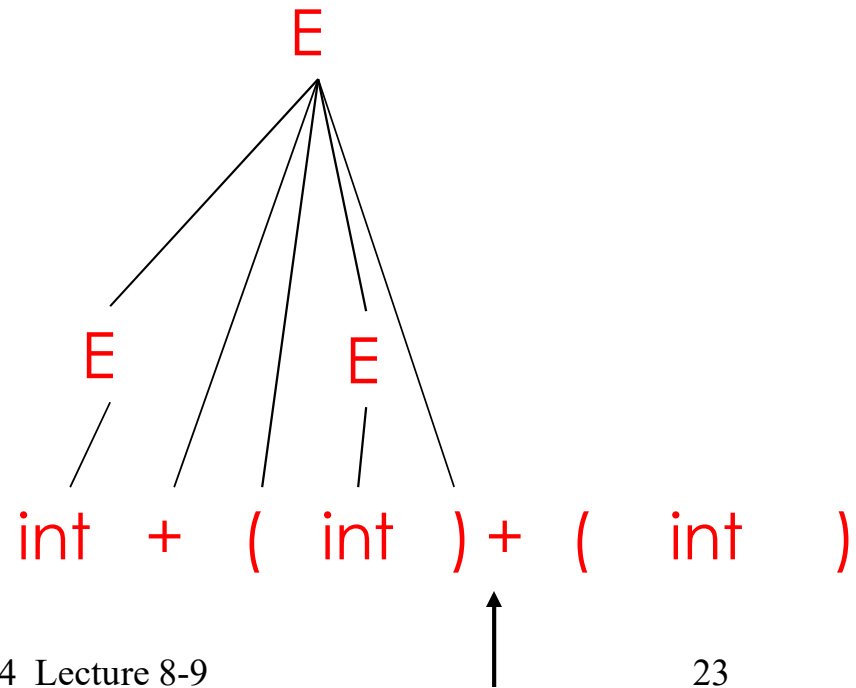


Shift-Reduce Example

▶ 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)	

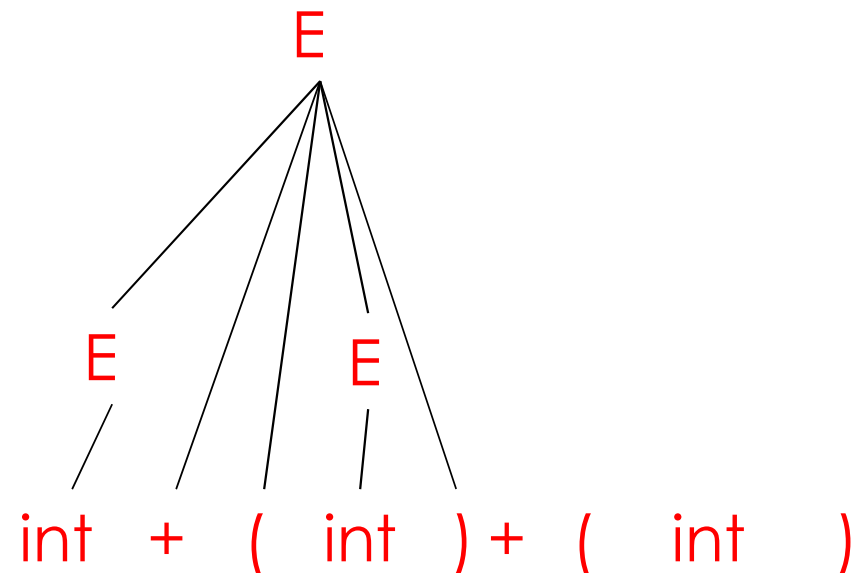
Shift-Reduce Example

▶ 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



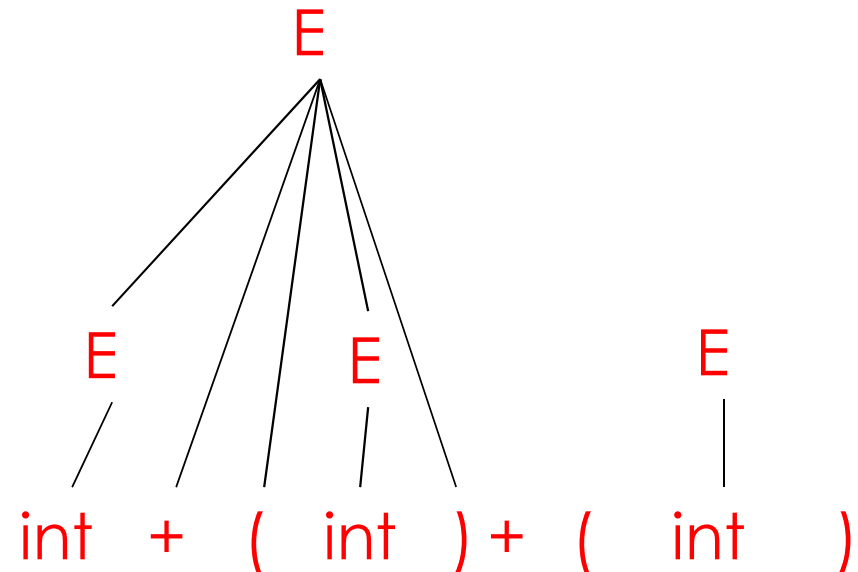
Shift-Reduce Example

► 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. $E \rightarrow \text{int}$



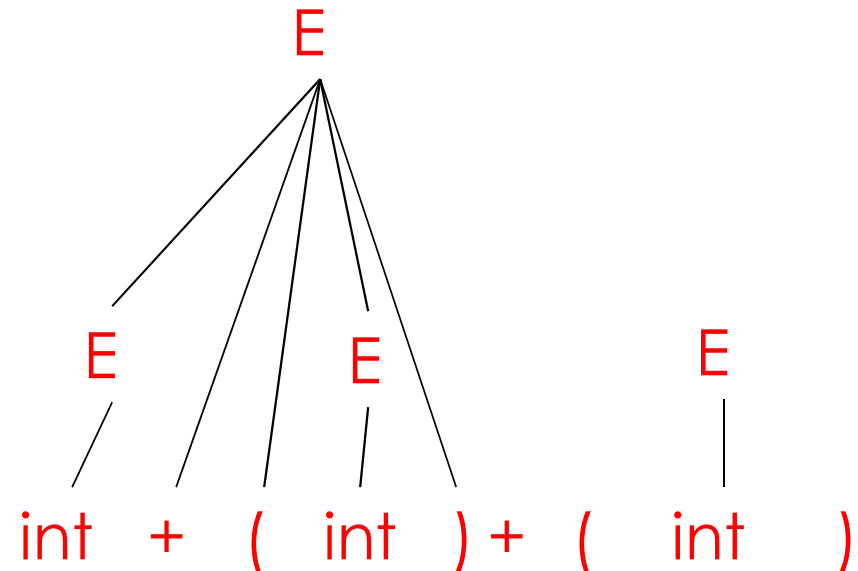
Shift-Reduce Example

► 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



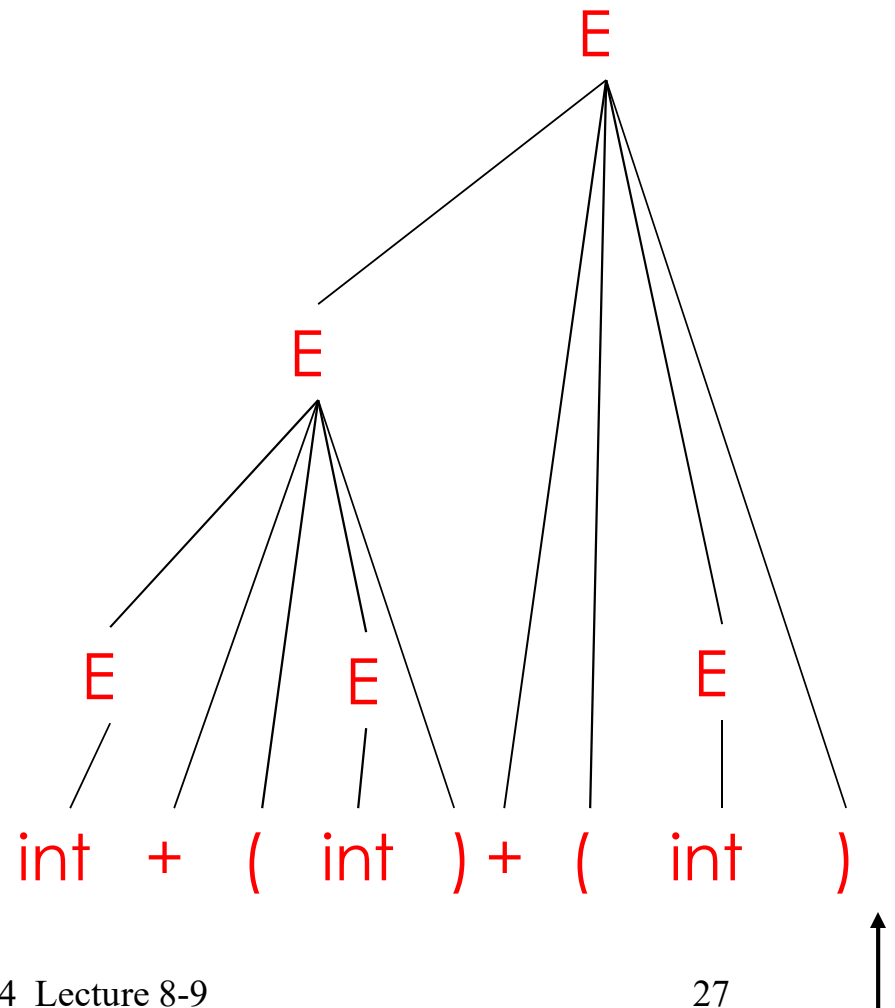
Shift-Reduce Example

► 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



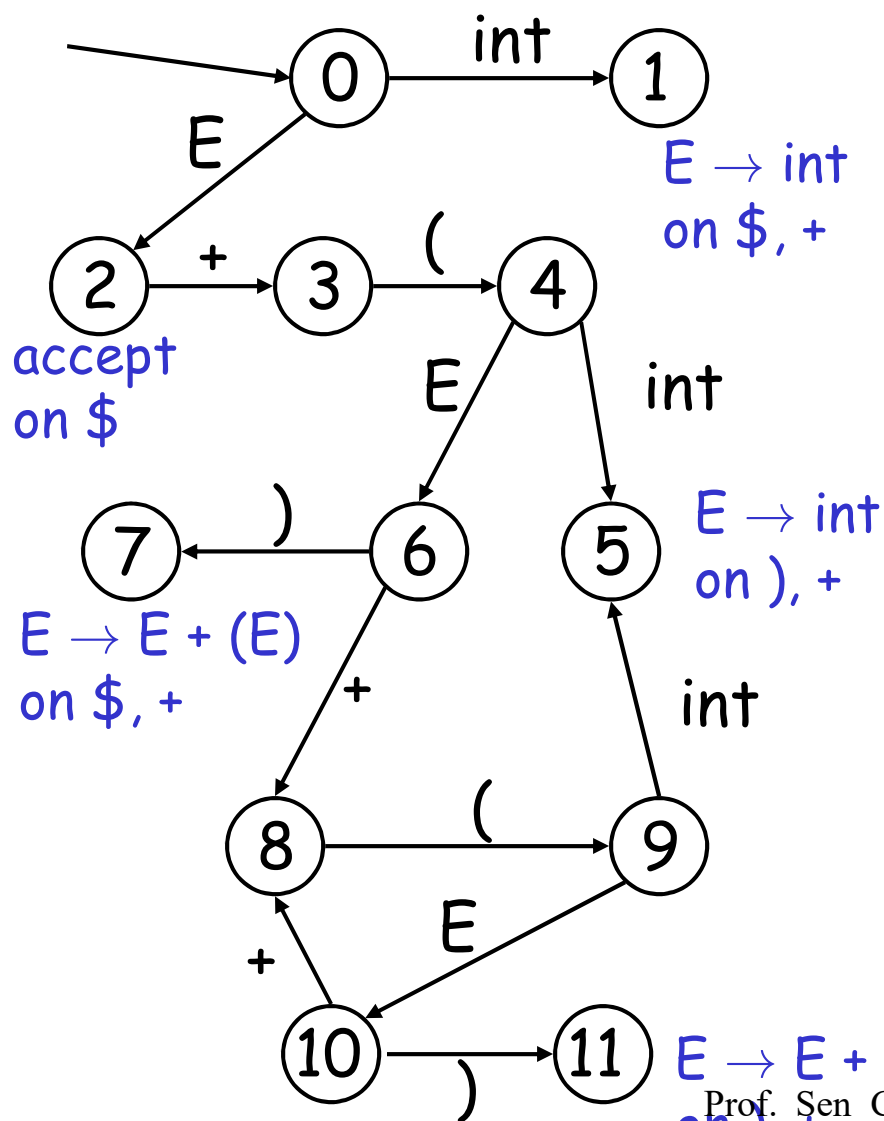
The Stack

- 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
 - DFA language consists of terminals and nonterminals
- 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



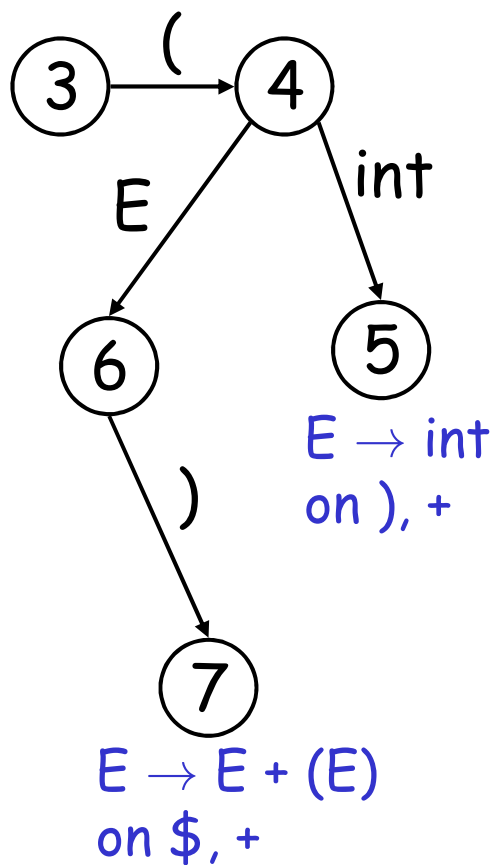
$\text{int} + (\text{int}) + (\text{int})\$$ shift
 $\text{int} \triangleright + (\text{int}) + (\text{int})\$$ $E \rightarrow \text{int}$
 $E \triangleright + (\text{int}) + (\text{int})\$$ shift(x3)
 $E + (\text{int} \triangleright) + (\text{int})\$$ $E \rightarrow \text{int}$
 $E + (E \triangleright) + (\text{int})\$$ shift
 $E + (E) \triangleright + (\text{int})\$$ $E \rightarrow$
 $E + (E)$
 $E \triangleright + (\text{int})\$$ shift
 $(\times 3)$
 $E + (\text{int} \triangleright)\$$ $E \rightarrow \text{int}$
 $E + (E \triangleright)\$$ shift
 $E + (E) \triangleright \$$ $E \rightarrow E + (E)$
 $E \triangleright \$$ accept

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:



	int	+	()	\$	E
...						
3			s4			
4	s5					g6
5		$r_{E \rightarrow int}$		$r_{E \rightarrow int}$		
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
- Remember for each stack element to which state it brings the DFA
- LR parser maintains a stack

$\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$

state_k is the final state of the DFA on $\text{sym}_1 \dots \text{sym}_k$

The LR Parsing Algorithm

Let $I = w\$$ be initial input

Let $j = 0$

Let DFA state 0 be the start state

Let $\text{stack} = \langle \text{dummy}, 0 \rangle$

repeat

case $\text{action}[\text{top_state}(\text{stack}), I[j]]$ of

shift k : push $\langle I[j++], k \rangle$

reduce $X \rightarrow \alpha$:

pop $|\alpha|$ pairs,

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

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 parser
 - What non-terminal we are looking for
 - What productions we are looking for
 - What we have seen so far from the rhs

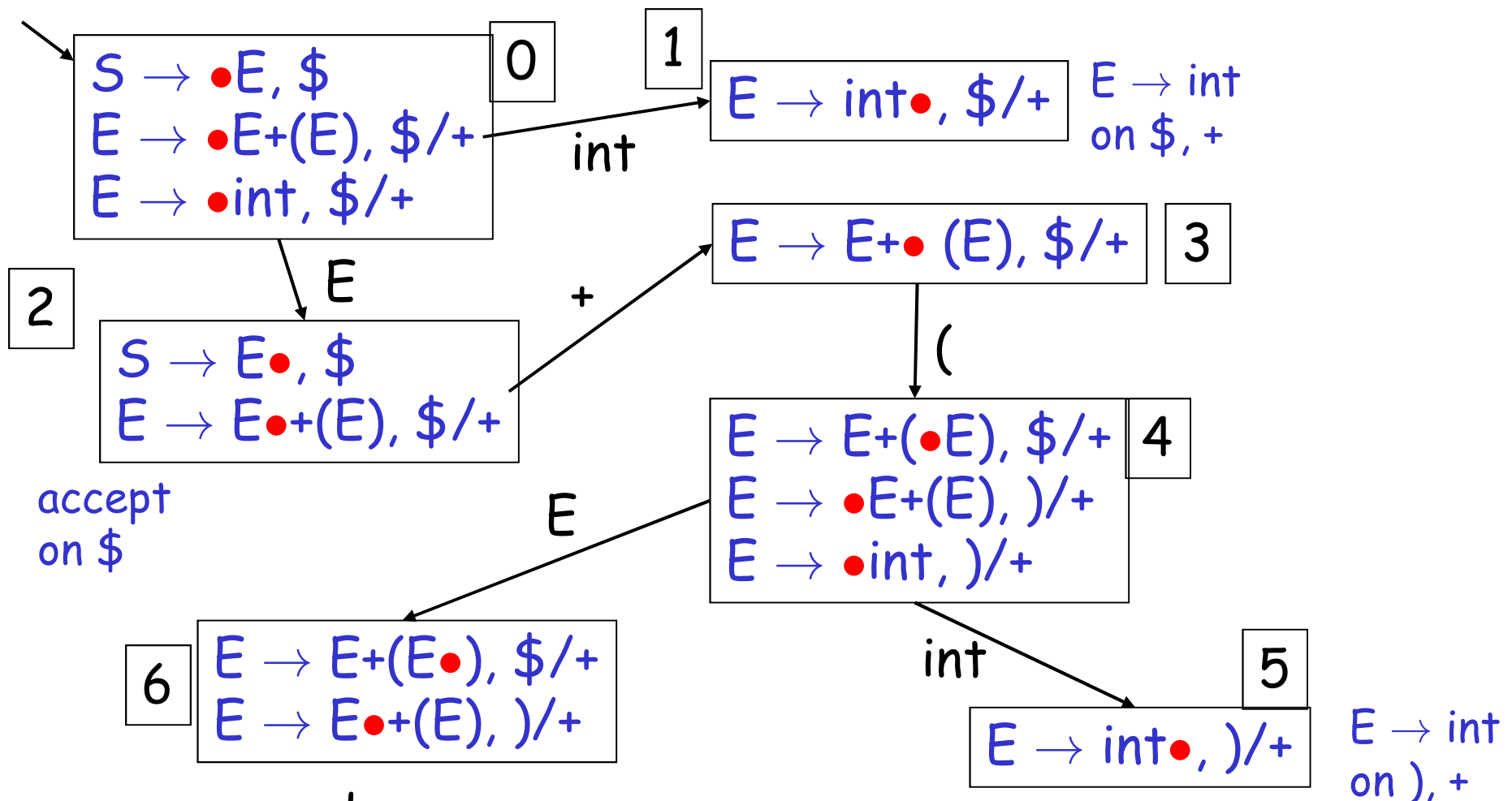
LR(1) Items

- 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

Constructing the Parsing DFA. Example.



and so on...

Note

- The symbol \blacktriangleright was used before to separate the stack from the rest of input
 - $\alpha \blacktriangleright \gamma$, where α is the stack and γ is the remaining string of terminals
- In LR(1) items \bullet is used to mark a prefix of a production rhs:
$$X \rightarrow \alpha \bullet \beta, a$$
 - Here β 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:
 $S \rightarrow \bullet E, \$$
 - Trying to find an S as a string derived from $E\$$
 - The stack is empty

LR(1) Items (Cont.)

- In context containing

$$E \rightarrow E + \bullet (E), +$$

- If (follows then we can perform a shift to context containing

$$E \rightarrow E + (\bullet E), +$$

- In 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 (Cont.)

- Consider a context with the item

$$E \rightarrow E + (\bullet E) , +$$

- 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:

$$E \rightarrow \bullet \text{int},)$$

$$E \rightarrow \bullet E + (E) ,)$$

The Closure Operation

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

Closure(Items) =

repeat

for each $[X \rightarrow \alpha \bullet Y \beta, a]$ in Items

for each production $Y \rightarrow \gamma$

for each $b \in \text{First}(\beta a)$

add $[Y \rightarrow \bullet \gamma, b]$ to Items

until Items is unchanged

Constructing the Parsing DFA (1)

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

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

- We abbreviate as:

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

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 $\text{Closure}([S \rightarrow \bullet E, \$])$
- A state that contains $[X \rightarrow \alpha \bullet, b]$ is labeled with “reduce with $X \rightarrow \alpha$ on b ”
- And now the transitions ...

The DFA Transitions

- A state “State” that contains $[X \rightarrow \alpha \bullet y \beta, b]$ 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)

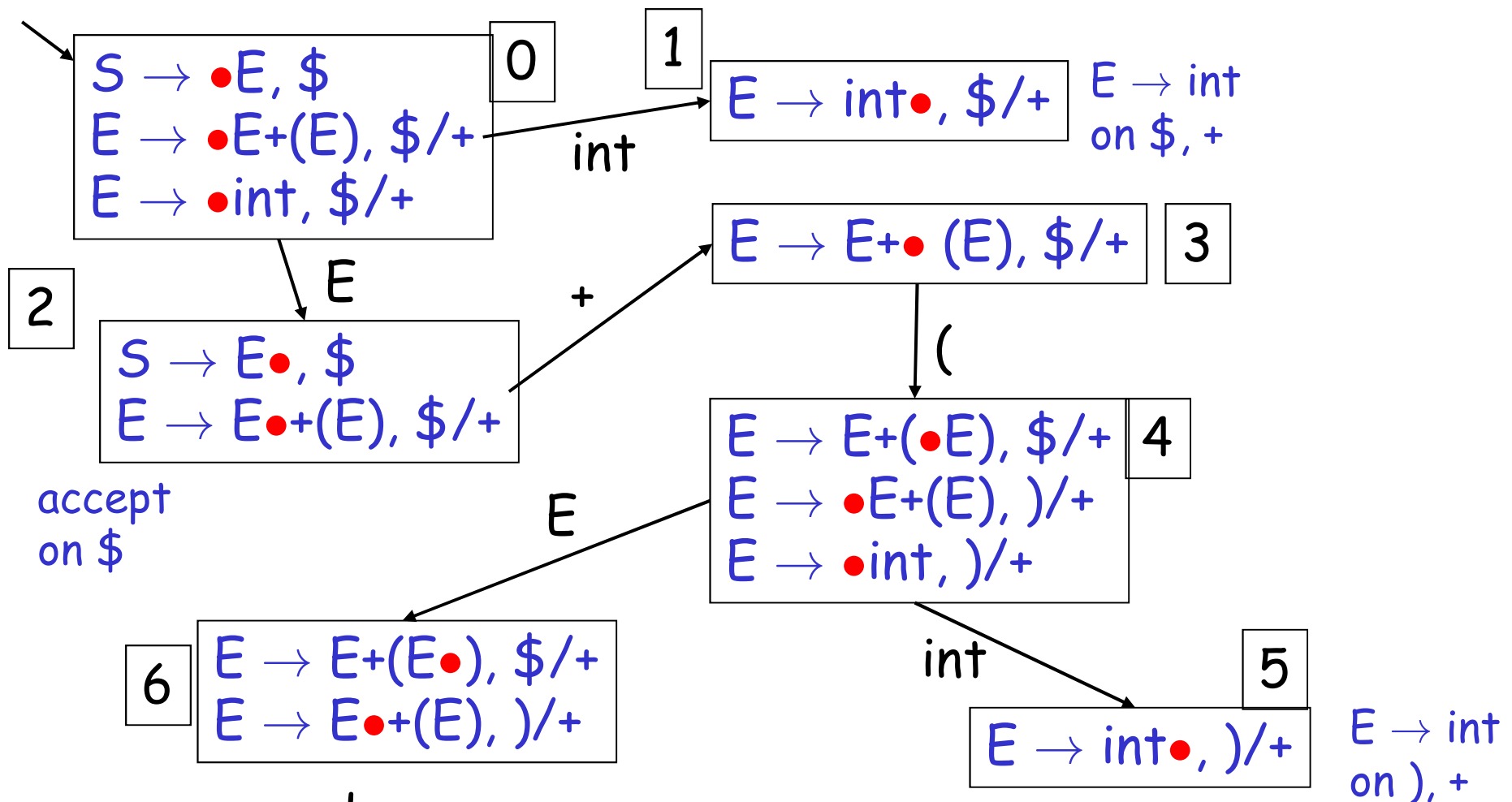
Items $\leftarrow \emptyset$

for each $[X \rightarrow \alpha \bullet y \beta, b] \in \text{State}$

add $[X \rightarrow \alpha y \bullet \beta, b]$ to Items

return Closure(Items)

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
- Default (bison, CUP, etc.) is to shift
 - Default behavior is as needed in this case

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

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

- We will have the states containing

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

...

...

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

More Shift/Reduce Conflicts

- In bison/CUP declare precedence and associativity:

```
%left +  
%left *
```

- Precedence of a rule = that of its last terminal
 - See bison/CUP 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:

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

- 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*
 - 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
$$[X \rightarrow \alpha \bullet, a] \text{ and } [Y \rightarrow \beta \bullet, a]$$
 - 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

$[S' \rightarrow \bullet S, \$]$		$[S \rightarrow id \bullet, \$]$
$[S \rightarrow \bullet, \$]$	\Rightarrow^{id}	$[S \rightarrow id \bullet S, \$]$
$[S \rightarrow \bullet id, \$]$		$[S \rightarrow \bullet, \$]$
$[S \rightarrow \bullet id S, \$]$		$[S \rightarrow \bullet id, \$]$
		$[S \rightarrow \bullet id S, \$]$
- Reduce/reduce conflict on input \$

$S' \rightarrow S \rightarrow id$
 $S' \rightarrow S \rightarrow id S \rightarrow id$
- Better rewrite the grammar: $S \rightarrow \epsilon \mid id S$

Using Parser Generators

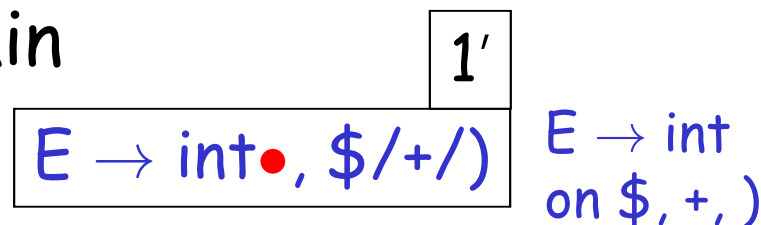
- Parser generators construct the parsing DFA given a CFG
 - Use precedence declarations and default conventions to resolve conflicts
 - The parser algorithm is the same for all grammars (and is provided as a library function)
- But most parser generators do not construct the DFA as described before
 - 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



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

$$\{ [X \rightarrow \alpha \bullet \beta, b], [Y \rightarrow \gamma \bullet \delta, d] \}$$

is

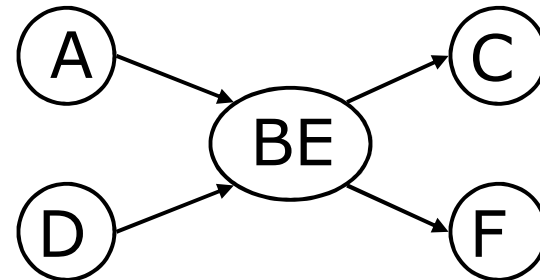
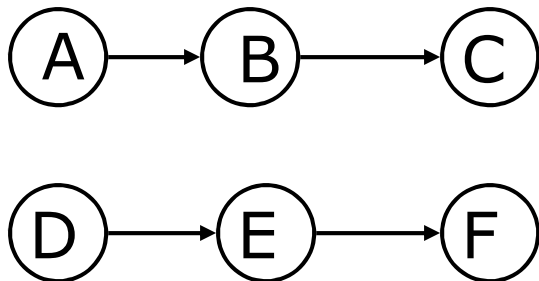
$$\{ X \rightarrow \alpha \bullet \beta, Y \rightarrow \gamma \bullet \delta \}$$

LALR States

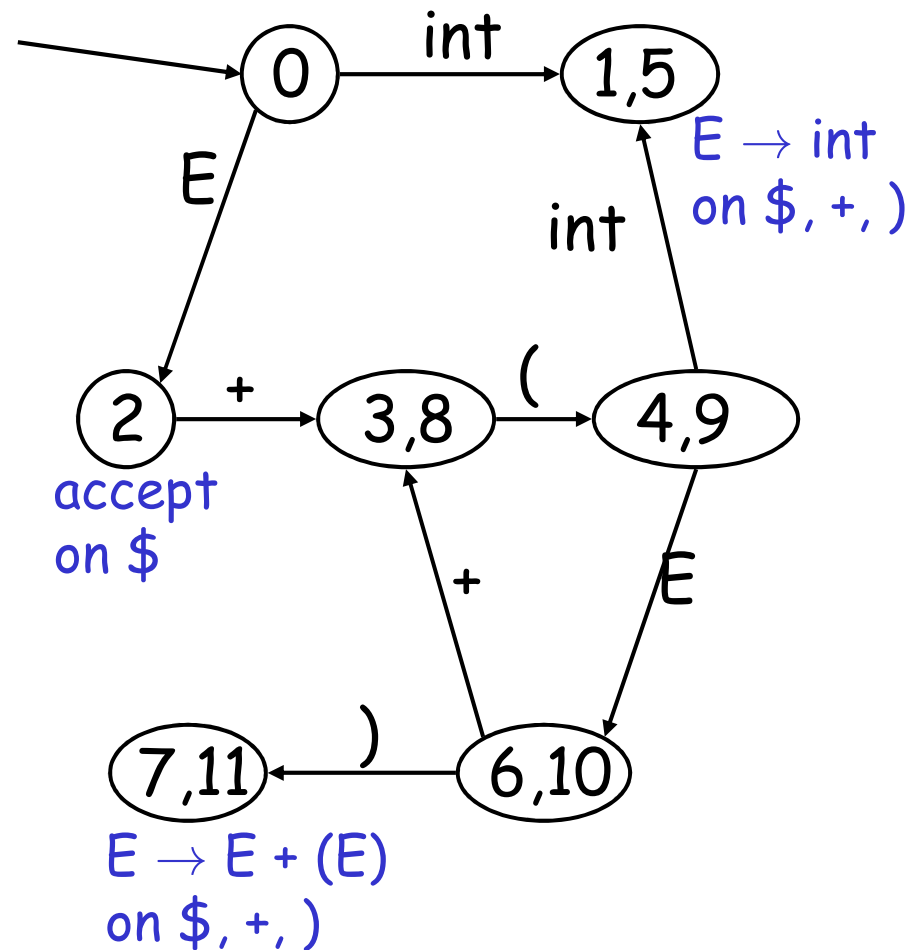
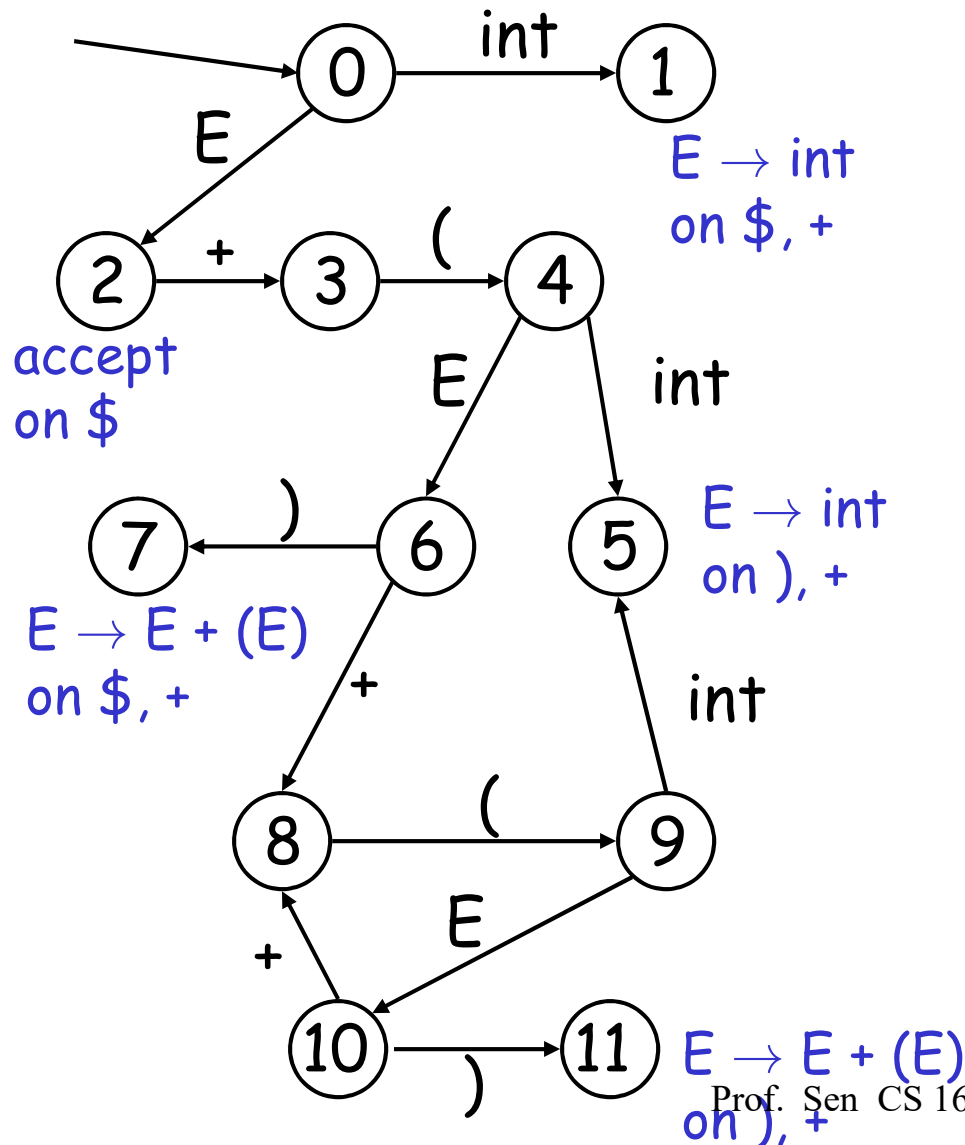
- Consider for example the LR(1) states
$$\{[X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c]\}$$
$$\{[X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d]\}$$
- They have the same core and can be merged
- And the merged state contains:
$$\{[X \rightarrow \alpha \bullet,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



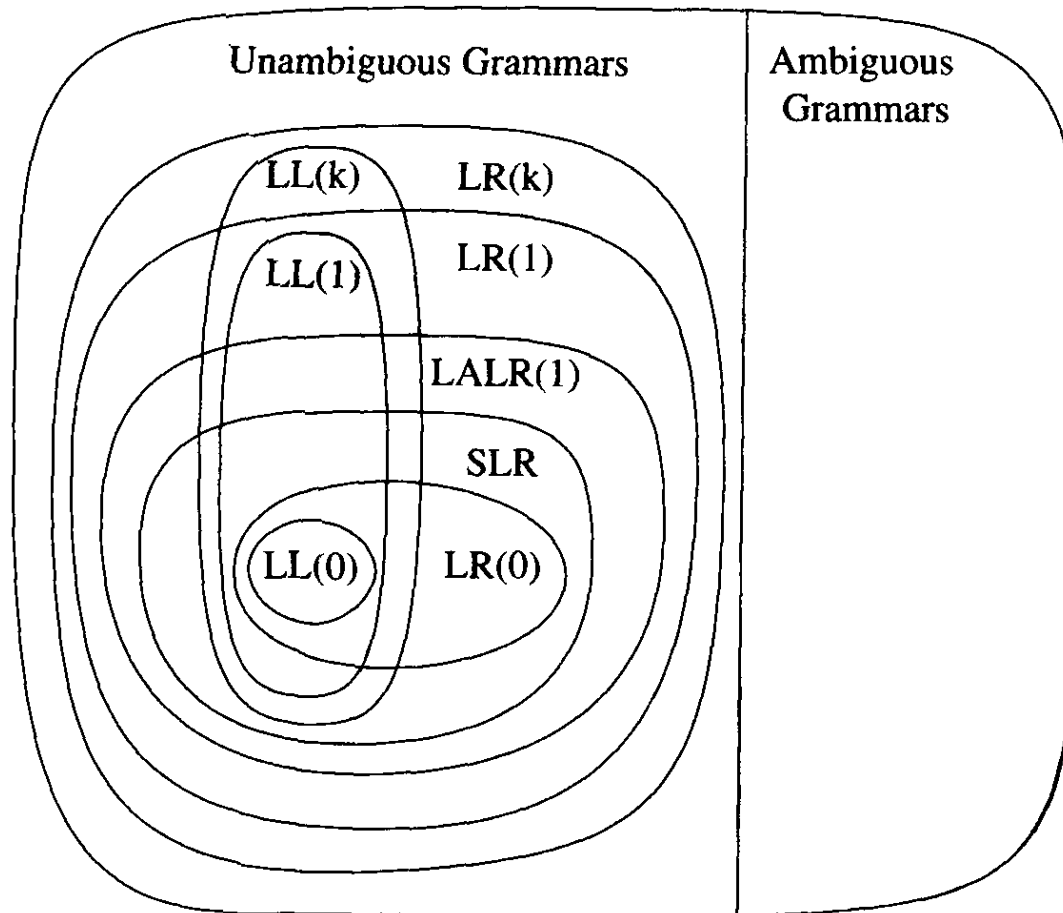
The LALR Parser Can Have Conflicts

- 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

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)
 - We use 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

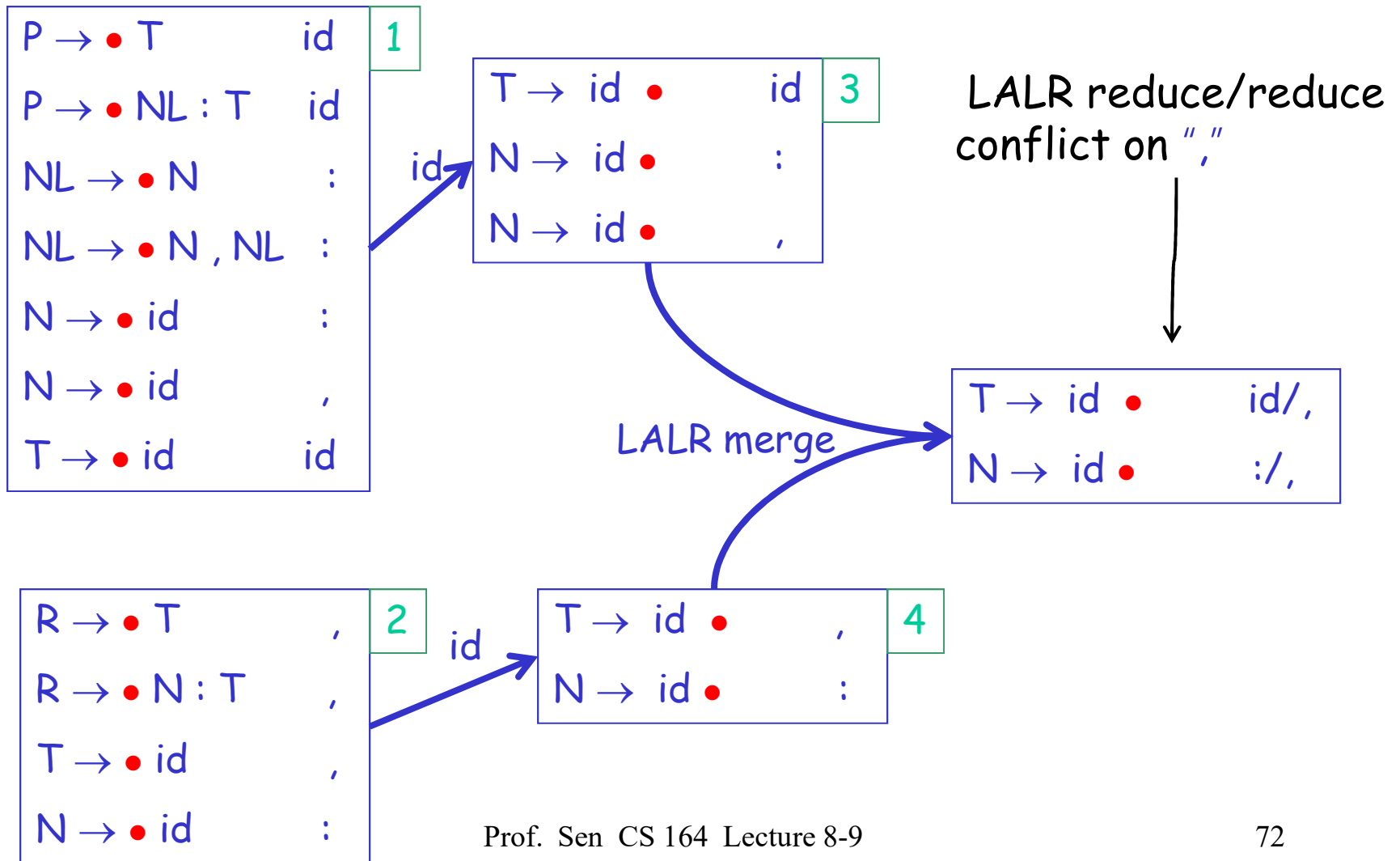
$$\begin{array}{ll} S \rightarrow P R , & NL \rightarrow N \mid N , NL \\ P \rightarrow T \mid NL : T & R \rightarrow T \mid N : T \\ N \rightarrow id & T \rightarrow id \end{array}$$

- 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 id is a
 - N when followed by $,$ or $:$
 - T when followed by id
- In R an 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



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

$R \rightarrow \text{id bogus}$

- **bugus** is a terminal not used by the lexer
- This production will never be used during parsing
- But it distinguishes **R** from **P**

A Few LR(1) States After Fix

$P \rightarrow \bullet T$ id
 $P \rightarrow \bullet NL : T$ id
 $NL \rightarrow \bullet N$:
 $NL \rightarrow \bullet N , NL$:
 $N \rightarrow \bullet id$:
 $N \rightarrow \bullet id$,
 $T \rightarrow \bullet id$ id

1

id

$T \rightarrow id \bullet$ id
 $N \rightarrow id \bullet$:
 $N \rightarrow id \bullet$,

3

Different cores \Rightarrow no LALR merging

$R \rightarrow \cdot T$,
 $R \rightarrow \cdot N : T$,
 $R \rightarrow \cdot id$ bogus ,
 $T \rightarrow \cdot id$,
 $N \rightarrow \cdot id$:

2

id

$T \rightarrow id \bullet$,
 $N \rightarrow id \bullet$:
 $R \rightarrow id \bullet$ bogus ,

4