LR Parsing. Parser Generators.

Lecture 7-8

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: int + (int) + (int)

The Idea

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

str $\leftarrow$ input string of terminals

repeat

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

until str = S

A Bottom-up Parse in Detail (1)

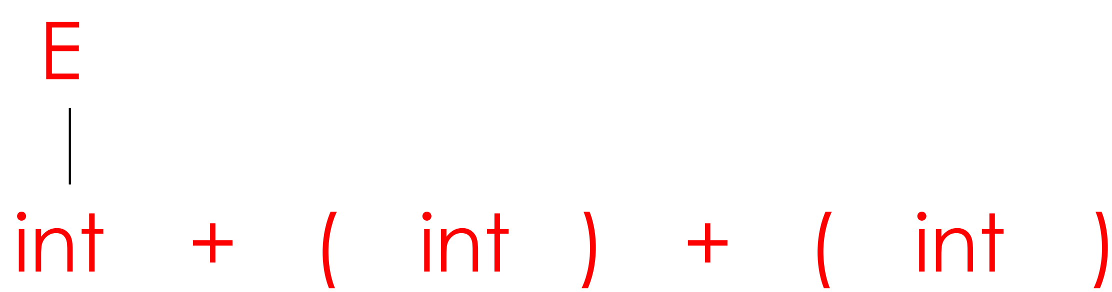
int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

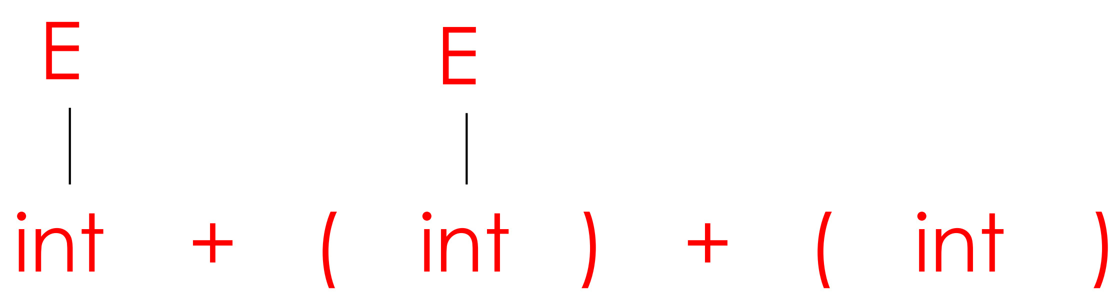


A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



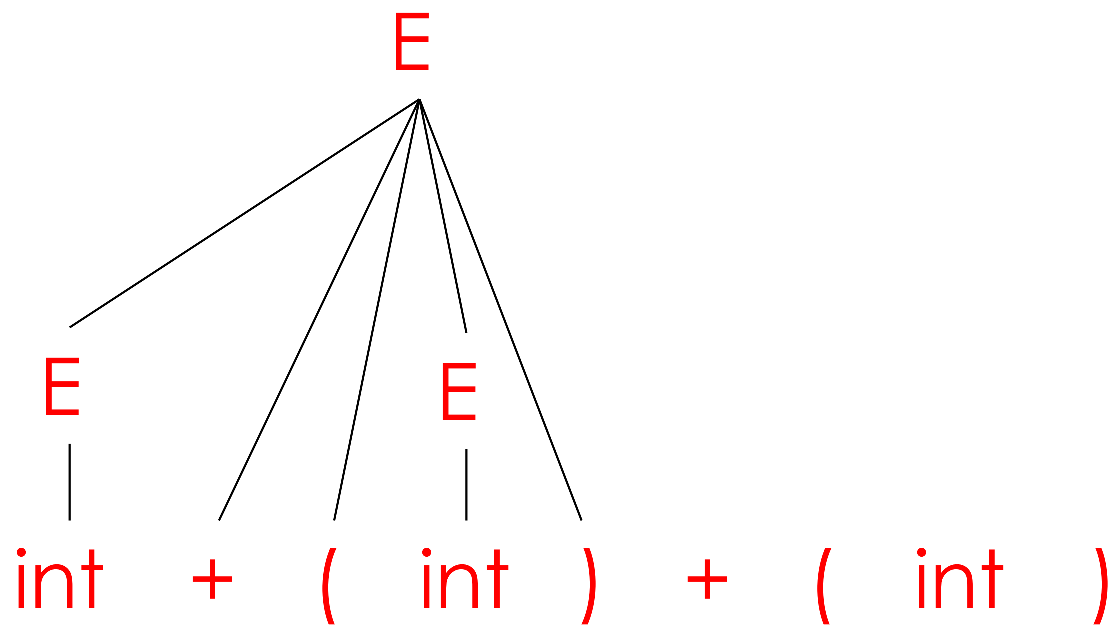
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

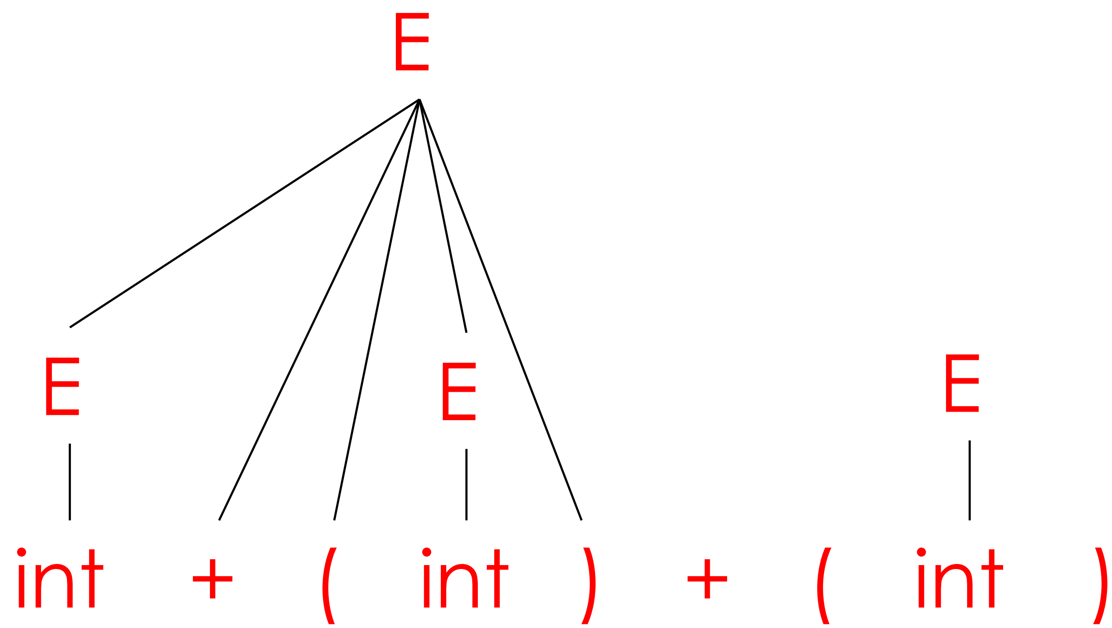
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

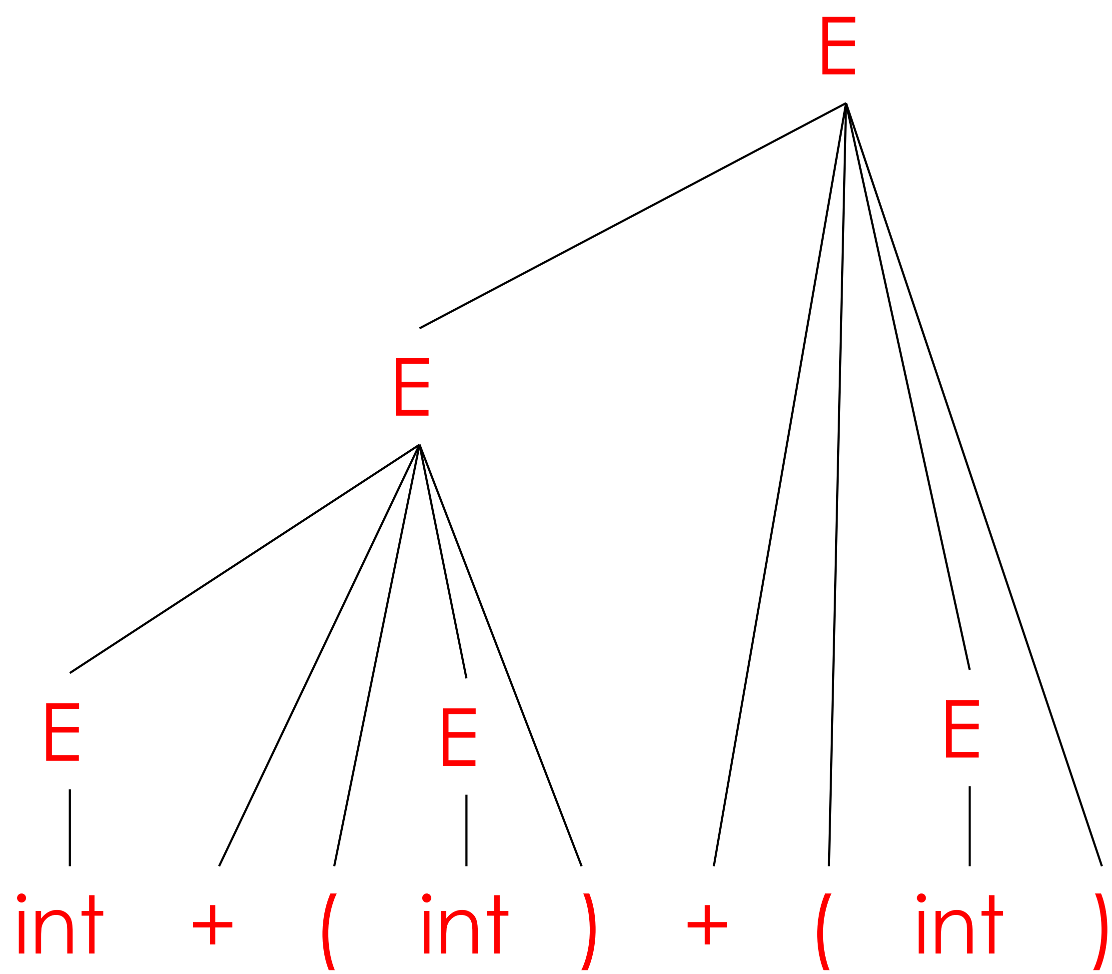
E + (E)



A Bottom-up Parse in Detail (6)

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

A rightmost derivation in reverse



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: ►$x_1 x_2 \ldots x_n$

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

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

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. $E \rightarrow int$

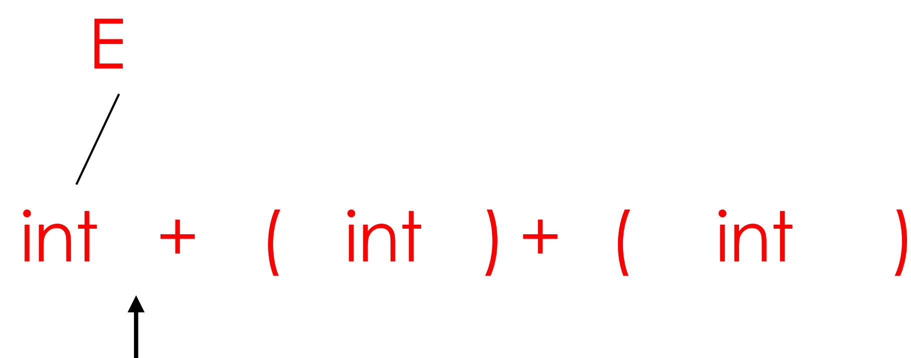
int + (int) + (int)

Shift-Reduce Example

▶ int + (int) + (int)\$ shift

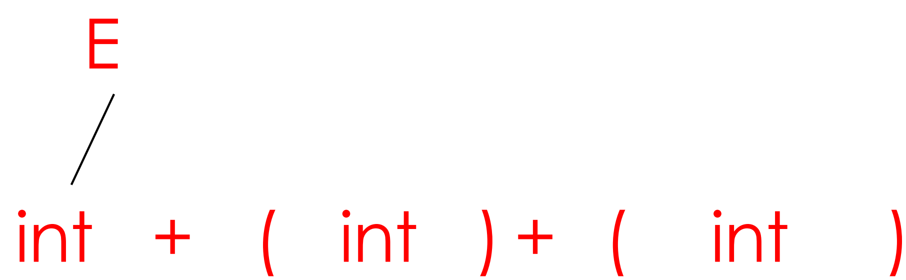
int ▶ + (int) + (int)\$ red. E $\rightarrow$ int

E ▶ + (int) + (int)\$ shift 3 times



Shift-Reduce Example

▶ int + (int) + (int)\$	shift
int ▶ + (int) + (int)\$	red. $E \rightarrow$ int
E ▶ + (int) + (int)\$	shift 3 times
E + (int ▶) + (int)\$	red. $E \rightarrow$ int



Shift-Reduce Example

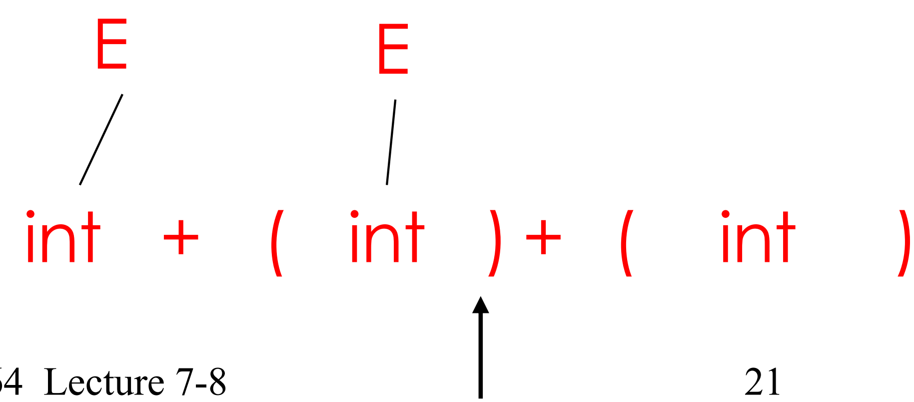
▶ int + (int) + (int)\$ shift

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

E ▶ + (int) + (int)\$ shift 3 times

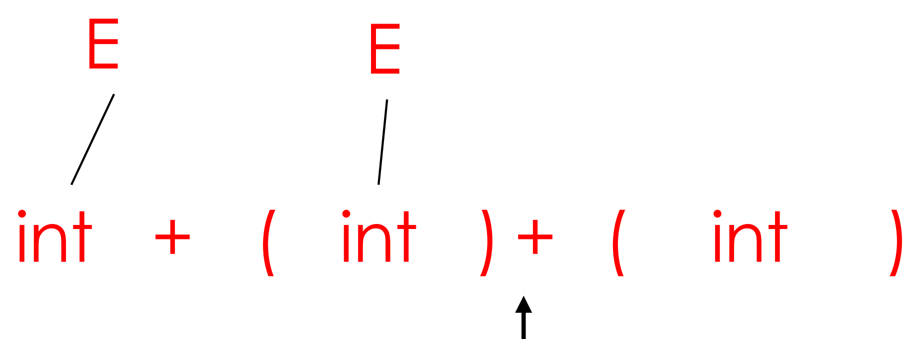
E + (int ▶) + (int)\$ red. $E \rightarrow$ int

E + (E ▶) + (int)\$ shift



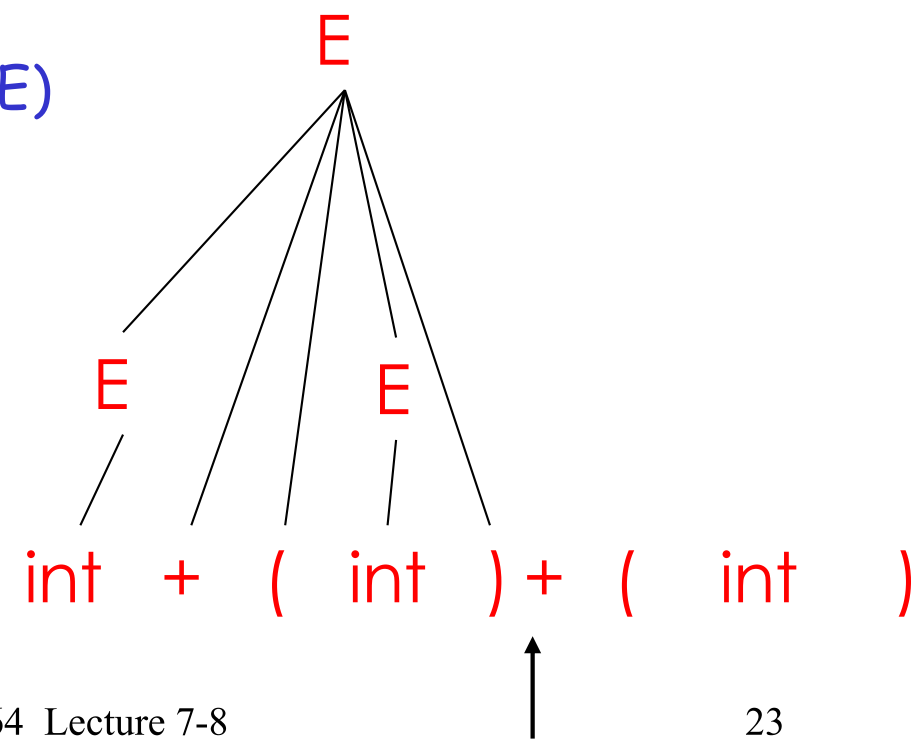
Shift-Reduce Example

▶ int + (int) + (int)\$	shift
int ▶ + (int) + (int)\$	red. E $\rightarrow$ int
E ▶ + (int) + (int)\$	shift 3 times
E + (int ▶) + (int)\$	red. E $\rightarrow$ int
E + (E ▶) + (int)\$	shift
E + (E) ▶ + (int)\$	red. E $\rightarrow$ E + (E)



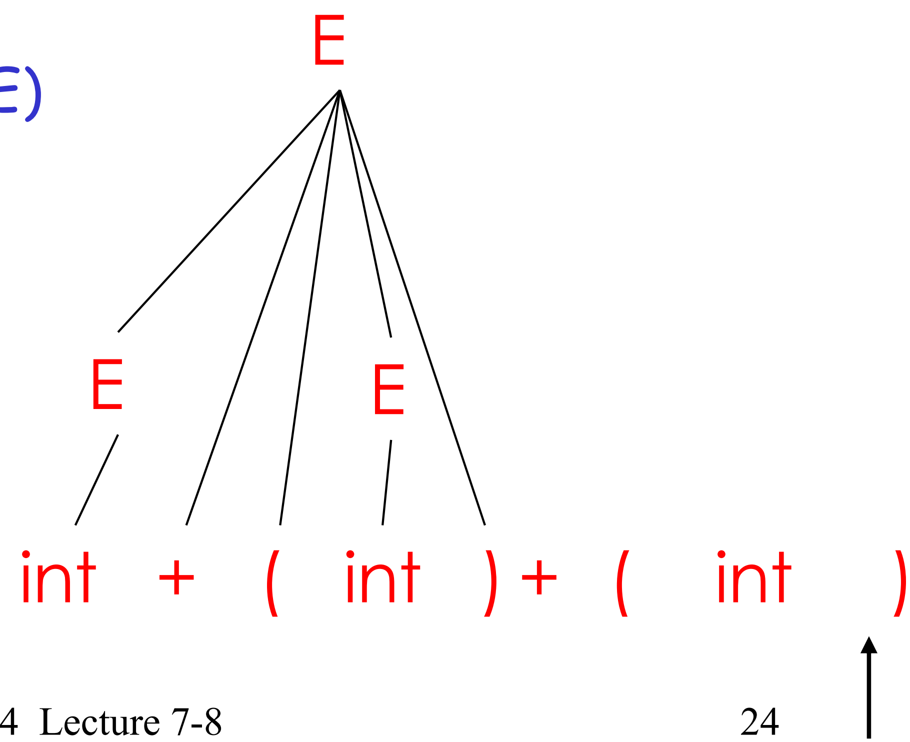
Shift-Reduce Example

▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E $\rightarrow$ int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E $\rightarrow$ int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	red. E $\rightarrow$ E + (E)
E ▶ + (int)$	shift 3 times



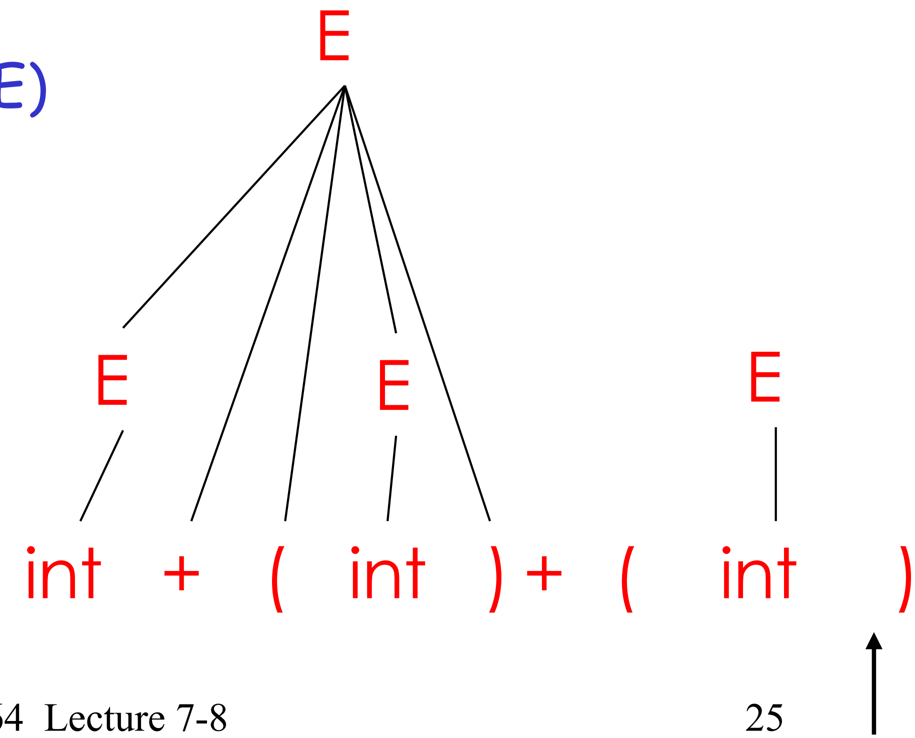
Shift-Reduce Example

▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E $\rightarrow$ int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E $\rightarrow$ int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	red. E $\rightarrow$ E + (E)
E ▶ + (int)$	shift 3 times
E + (int ▶)$	red. E $\rightarrow$ int



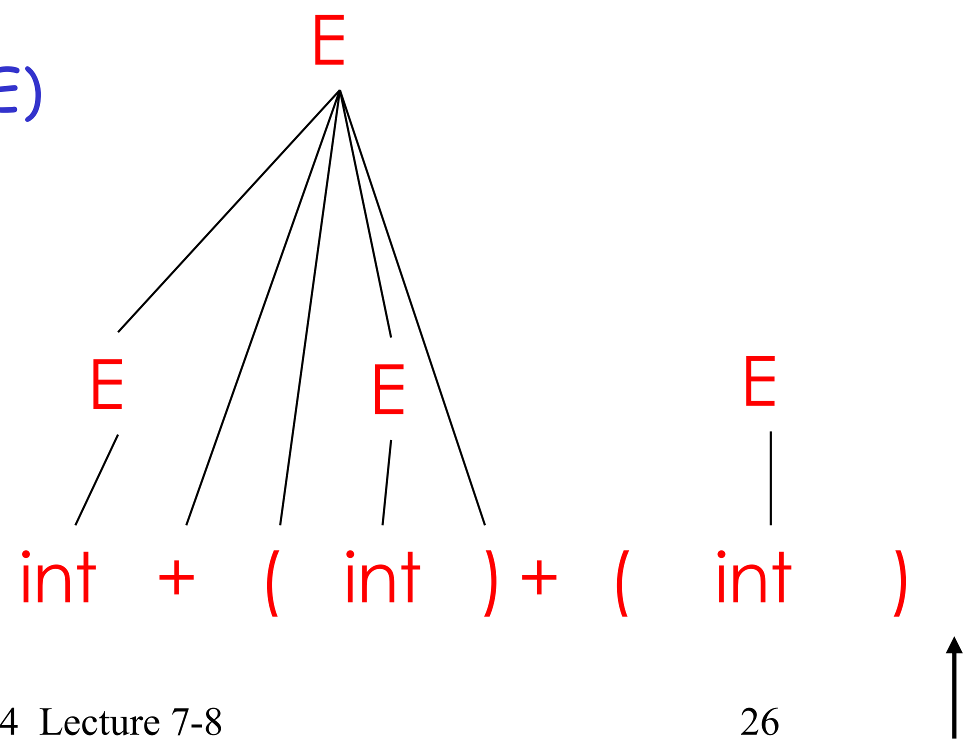
Shift-Reduce Example

▸ int + (int) + (int)$	shift
int ▸ + (int) + (int)$	red. E $\rightarrow$ int
E ▸ + (int) + (int)$	shift 3 times
E + (int ▸) + (int)$	red. E $\rightarrow$ int
E + (E ▸) + (int)$	shift
E + (E) ▸ + (int)$	red. E $\rightarrow$ E + (E)
E ▸ + (int)$	shift 3 times
E + (int ▸)$	red. E $\rightarrow$ int
E + (E ▸)$	shift



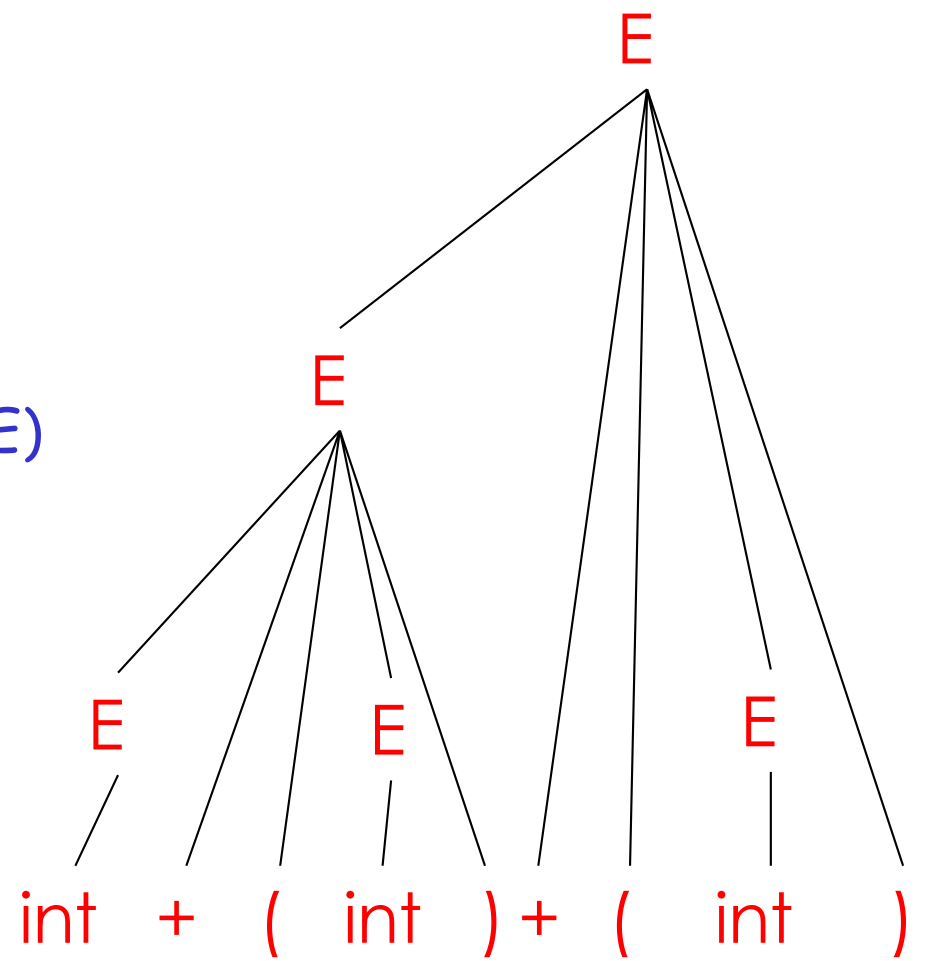
Shift-Reduce Example

▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E $\rightarrow$ int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E $\rightarrow$ int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	red. E $\rightarrow$ E + (E)
E ▶ + (int)$	shift 3 times
E + (int ▶)$	red. E $\rightarrow$ int
E + (E ▶)$	shift
E + (E) ▶ $	red. E $\rightarrow$ E + (E)



Shift-Reduce Example

▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	red. E $\rightarrow$ int
E ▶ + (int) + (int)$	shift 3 times
E + (int ▶) + (int)$	red. E $\rightarrow$ int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	red. E $\rightarrow$ E + (E)
E ▶ + (int)$	shift 3 times
E + (int ▶)$	red. E $\rightarrow$ 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

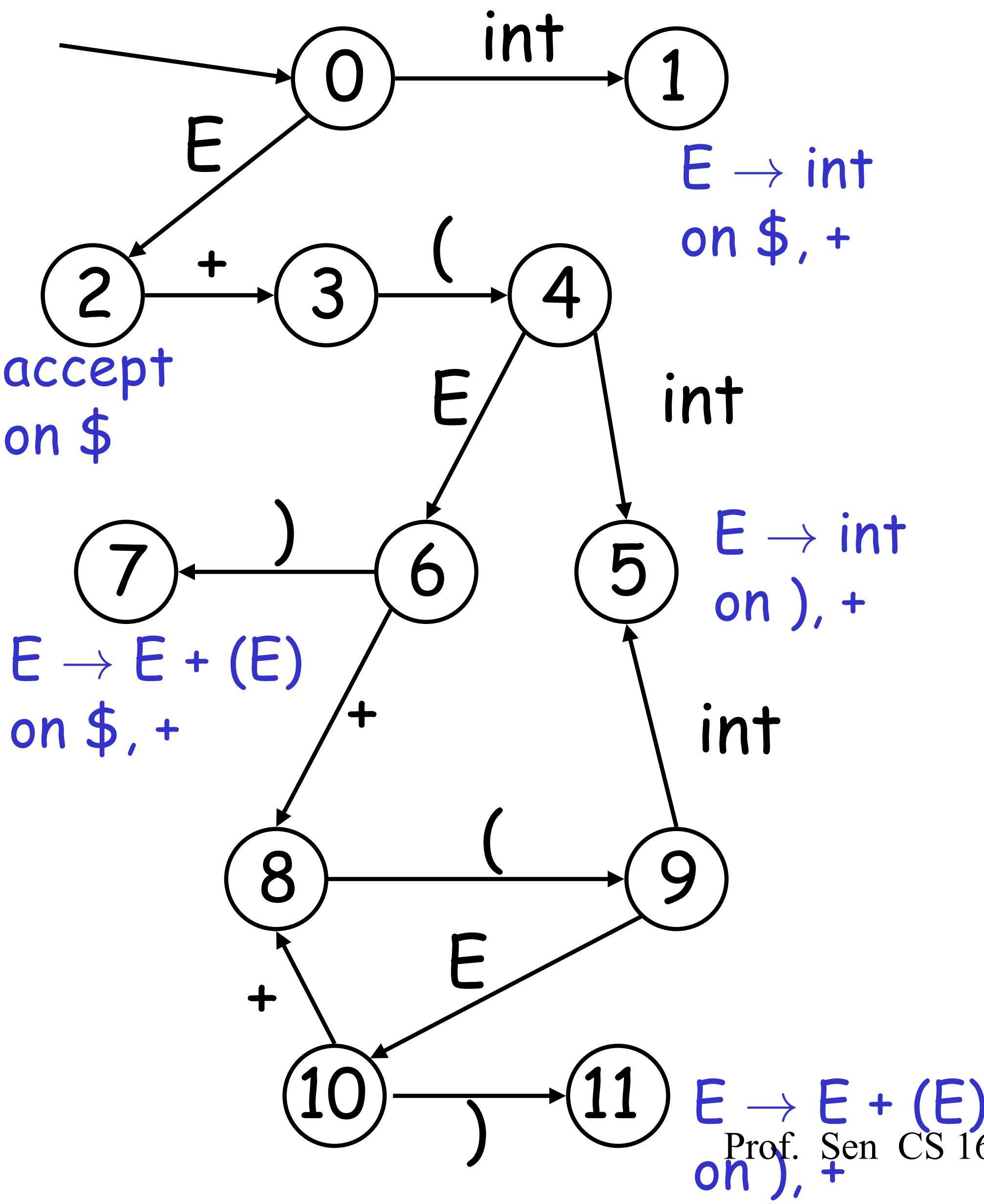
- Reduce pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

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



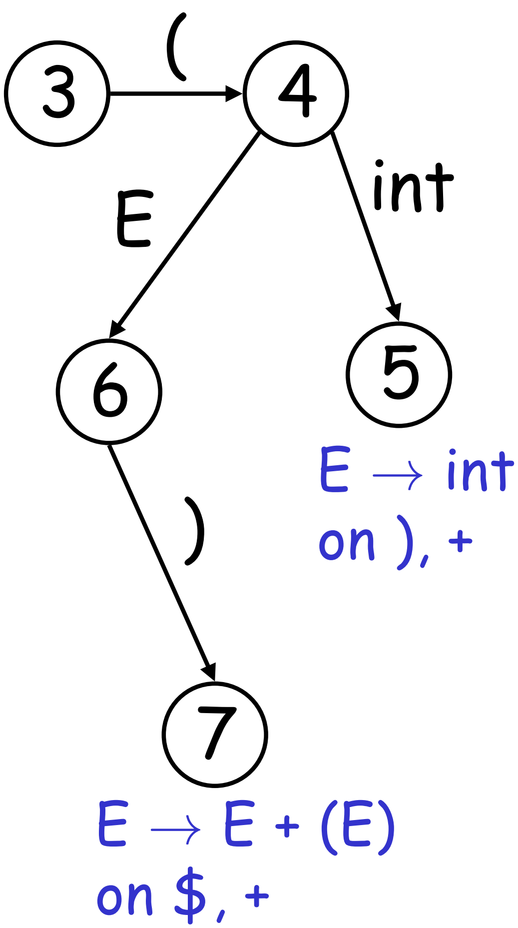
▸ int + (int) + (int)$	shift
int ▸ + (int) + (int)$	E → int
E ▸ + (int) + (int)$	shift(x3)
E + (int ▸) + (int)$	E → int
E + (E ▸) + (int)$	shift
E + (E) ▸ + (int)$	E → E+(E)
E ▸ + (int)$ (x3)	shift
E + (int ▸)$	E → int
E + (E ▸)$	shift
E + (E) ▸ $	E → E+(E)
E ▸ $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - 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		$r_{E \rightarrow \text{int}}$		$r_{E \rightarrow \text{int}}$		
6	s8		s7			
7		$r_{E \rightarrow E+(E)}$			$r_{E \rightarrow E+(E)}$	
…						

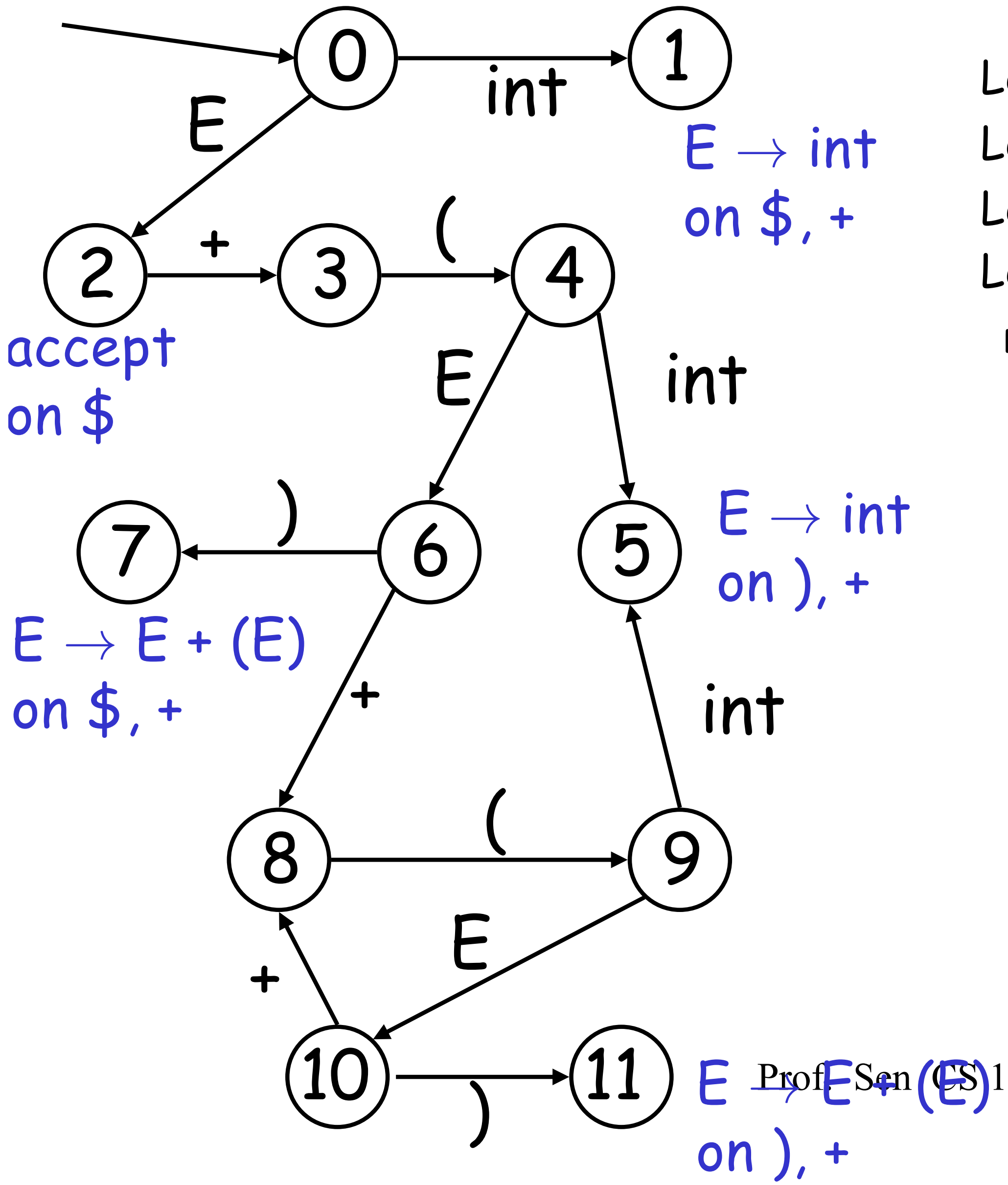
The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- Remember for each stack element to which state it brings the DFA
- LR parser maintains a stack

$$\langle\, sym_1, state_1 \,\rangle \ldots \langle\, sym_n, state_n \,\rangle$$

$state_k$ is the final state of the DFA on $sym_1 \ldots sym_k$

The LR Parsing Algorithm



Let I = w$ be initial input

Let j = 0

Let DFA state 0 be the start state

Let stack = ⟨ dummy, 0 ⟩

repeat

case action[top_state(stack), I[j]] of

shift k: push ⟨ I[j++], k ⟩

reduce X → α:

pop |α| pairs,

push ⟨X, Goto[top_state(stack), X]⟩

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?

Outline

- 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

$$\alpha \blacktriangleright \gamma$$

  - α is a stack of terminals and non-terminals
  - γ is the string of terminals not yet examined

- Initially: $\blacktriangleright x_1 x_2 \ldots x_n$

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

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

- Reduce pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

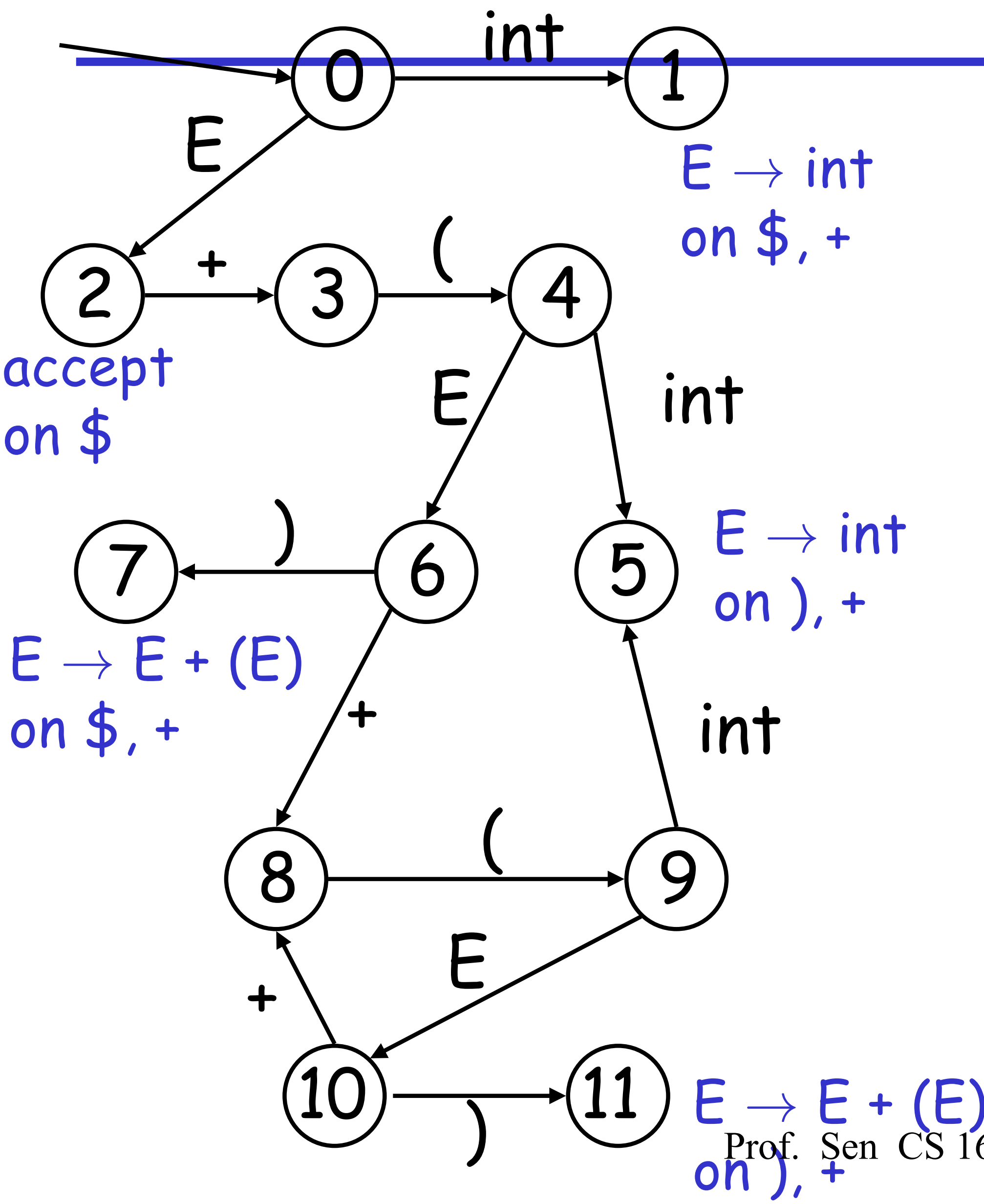
$$E + (\underline{E + (E)} \blacktriangleright) \Rightarrow E + (\underline{E} \blacktriangleright)$$

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 ►
  - If X has a transition labeled tok then shift
  - If X is labeled with "$A \to \beta$ 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

End of review

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 of a production

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

LR(1) items

- $A \to \bullet X\ Y\ Z, a$
  - we hope to see a string derivable from X Y Z a
- $A \to X \bullet Y\ Z, a$
  - we have seen a string derivable from X
  - we hope to see a string derivable from Y Z a
- $A \to X\ Y\ Z \bullet, a$
  - we have seen a string derivable from X Y Z
  - time to reduce X Y Z to A

Note

- The symbol ► was used before to separate the stack from the rest of input
  - α ► γ, where α is the stack and γ is the remaining string of terminals
- In LR(1) items • is used to mark a prefix of a production rhs:

$$X \to \alpha \bullet \beta, a$$

  - Here β might contain non-terminals as well
- In both cases 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

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

  - If (follows then we can perform a <u>shift</u> to context containing

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

- In context containing

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

  - We can perform a reduction with $E \to E + (E)$
  - But only if a + follows

LR(1) Items (Cont.)

- Consider a context with the item

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

- We expect next a string derived from $E\,)\,+$
- There are two productions for E

$$E \to int \quad \text{and} \quad E \to E + (\,E)$$

- We describe this by <u>extending</u> the context with two more items:

$$E \to \bullet\, int,\,)$$

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

The Closure Operation

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

Closure(Items) =

    repeat
        for each $[X \to \alpha \bullet Y\beta, a]$ in Items
            for each production $Y \to \gamma$
                for each $b \in$ First(βa)
                    add $[Y \to \bullet\gamma, b]$ to Items
    until Items is unchanged

Constructing the Parsing DFA (1)

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

$$
\begin{aligned}
S &\rightarrow \bullet E,\ \$ \\
E &\rightarrow \bullet E+(E),\ \$ \\
E &\rightarrow \bullet \mathrm{int},\ \$ \\
E &\rightarrow \bullet E+(E),\ + \\
E &\rightarrow \bullet \mathrm{int},\ +
\end{aligned}
$$

- We abbreviate as:

$$
\begin{aligned}
S &\rightarrow \bullet E,\ \$ \\
E &\rightarrow \bullet E+(E),\ \$/+ \\
E &\rightarrow \bullet \mathrm{int},\ \$/+
\end{aligned}
$$

Constructing the Parsing DFA (2)

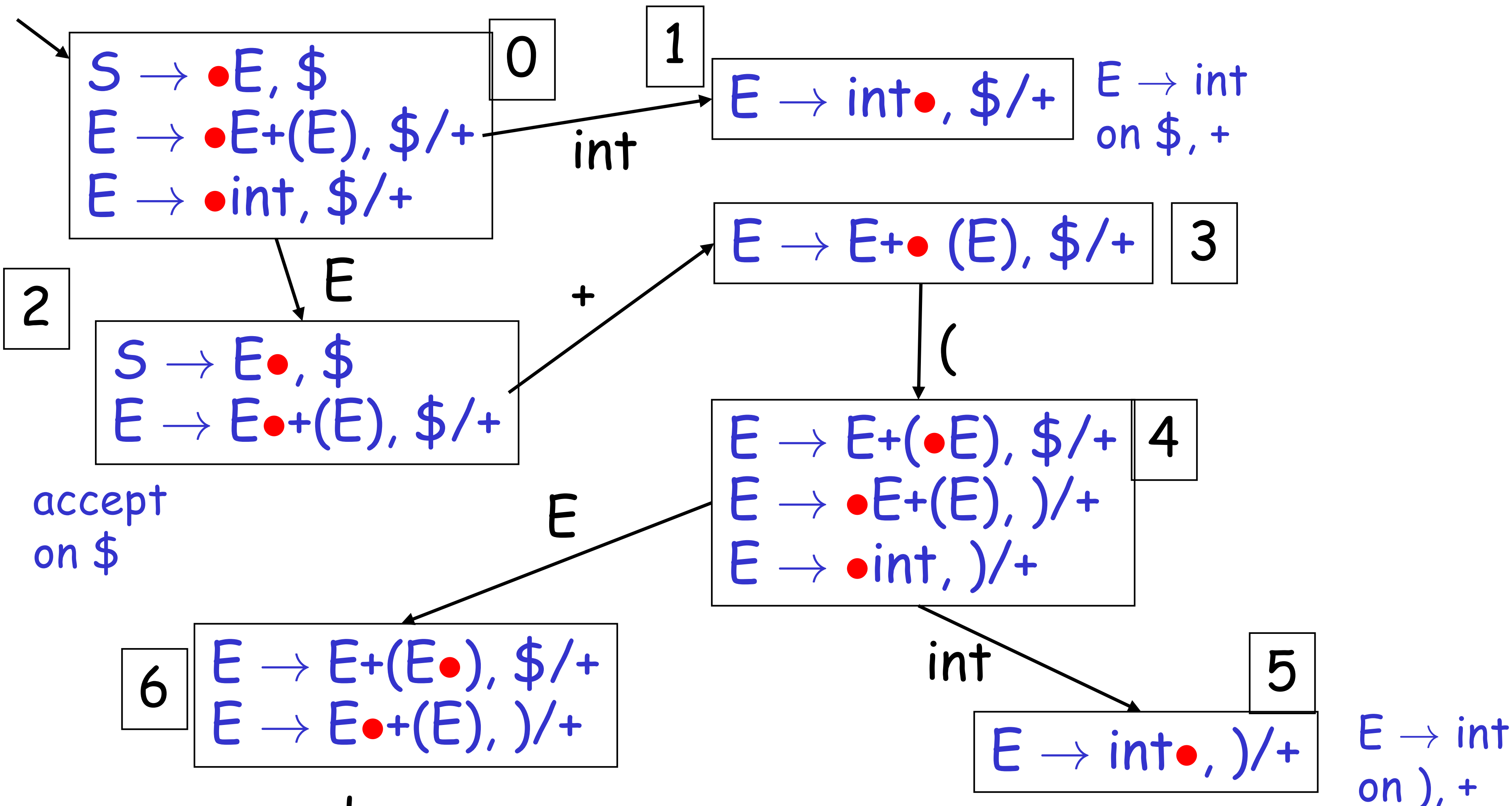
- A DFA state is a <u>closed</u> set of LR(1) items
  - This means that we performed Closure
- The start state is Closure([$S \rightarrow \bullet E$, \$])
- A state that contains [$X \rightarrow \alpha\bullet$, b] is labeled with "reduce with $X \rightarrow \alpha$ on b"
- And now the transitions …

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 $\leftarrow \varnothing$
  for each $[X \rightarrow \alpha \bullet y\beta, b] \in$ State
    add $[X \rightarrow \alpha y \bullet \beta, b]$ to Items
  return Closure(Items)

Constructing the Parsing DFA. Example.



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 <u>shift-reduce conflict</u>

Shift/Reduce Conflicts

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

  $S \rightarrow$ if E then S | if E then S else S | OTHER

- Will have DFA state containing

  [$S \rightarrow$ if E then S•, else]

  [$S \rightarrow$ if E then S• else S, $]

- 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

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

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

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

- 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 <u>shift</u> 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:

$$[E \to E * \bullet E, +] \qquad [E \to E * E\bullet, +]$$
$$[E \to \bullet E + E, +] \Rightarrow^{E} [E \to E \bullet + E, +]$$
$$\ldots \qquad \ldots$$

- 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

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

- We will also have the states

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

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

… $\qquad$ …

- 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 → if E then S•, else]

  [S → if E then S• else S, 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

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

  - 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

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

- Reduce/reduce conflict on input $\$$

$$S' \to S \to id$$

$$S' \to S \to id\ S \to id$$

- Better rewrite the grammar: $S \to \varepsilon \mid id\ S$

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.

1			5	
$E \to int\bullet$, \$/+	$E \to int$ on \$, +	and	$E \to int\bullet$,)/+	$E \to int$ 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

1'	
$E \to int\bullet$, \$/+/)	$E \to int$ on \$, +,)

The Core of a Set of LR Items

- Definition: The <u>core</u> 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

$$\{[X \to \alpha\bullet, a], [Y \to \beta\bullet, c]\}$$

$$\{[X \to \alpha\bullet, b], [Y \to \beta\bullet, d]\}$$

- They have the same core and can be merged
- And the merged state contains:

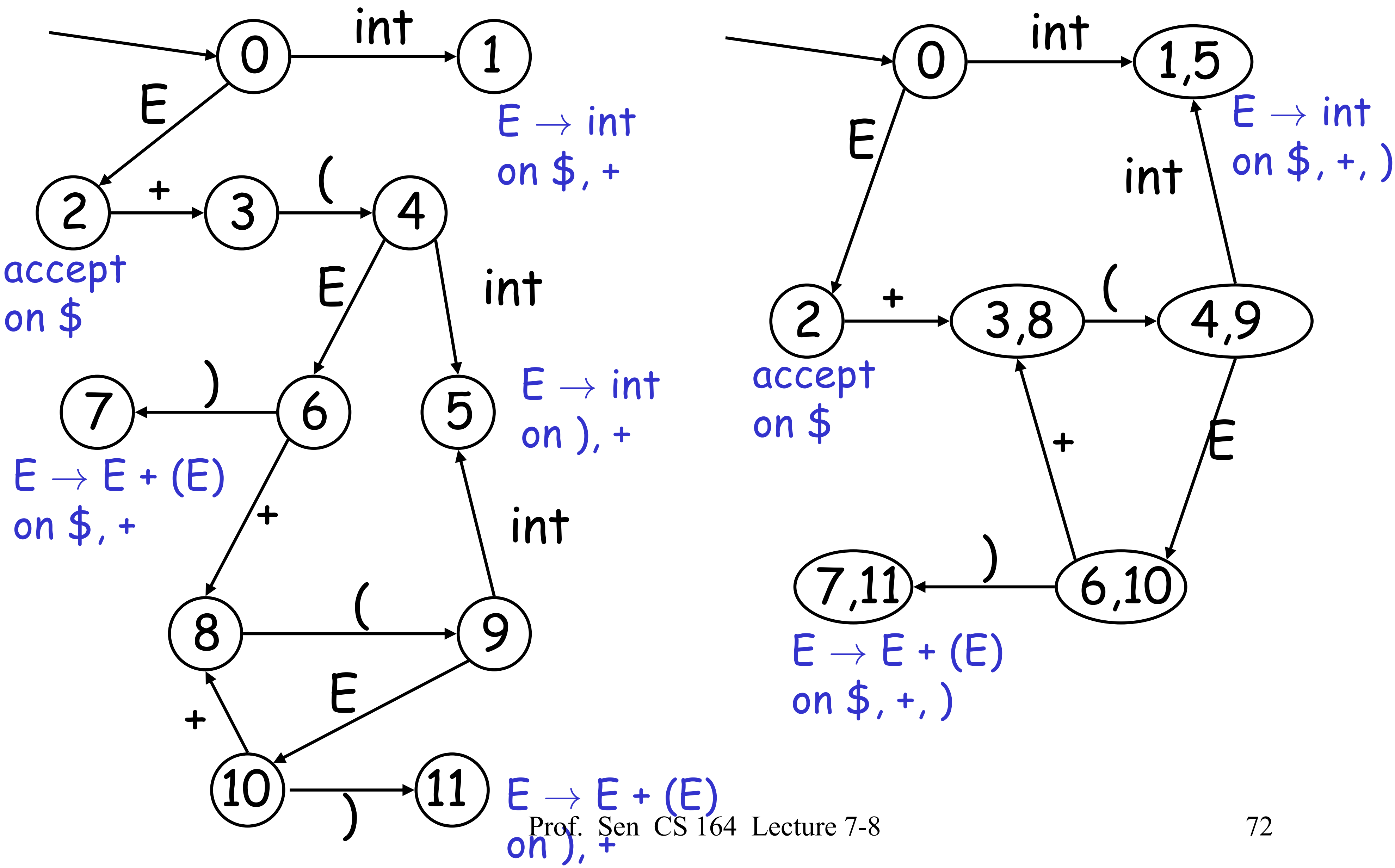
$$\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, c/d]\}$$

- 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

$$\{[X \to \alpha\bullet, a], [Y \to \beta\bullet, b]\}$$

$$\{[X \to \alpha\bullet, b], [Y \to \beta\bullet, a]\}$$

- And the merged LALR(1) state

$$\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, a/b]\}$$

- Has a new reduce-reduce conflict

- In practice such cases are rare

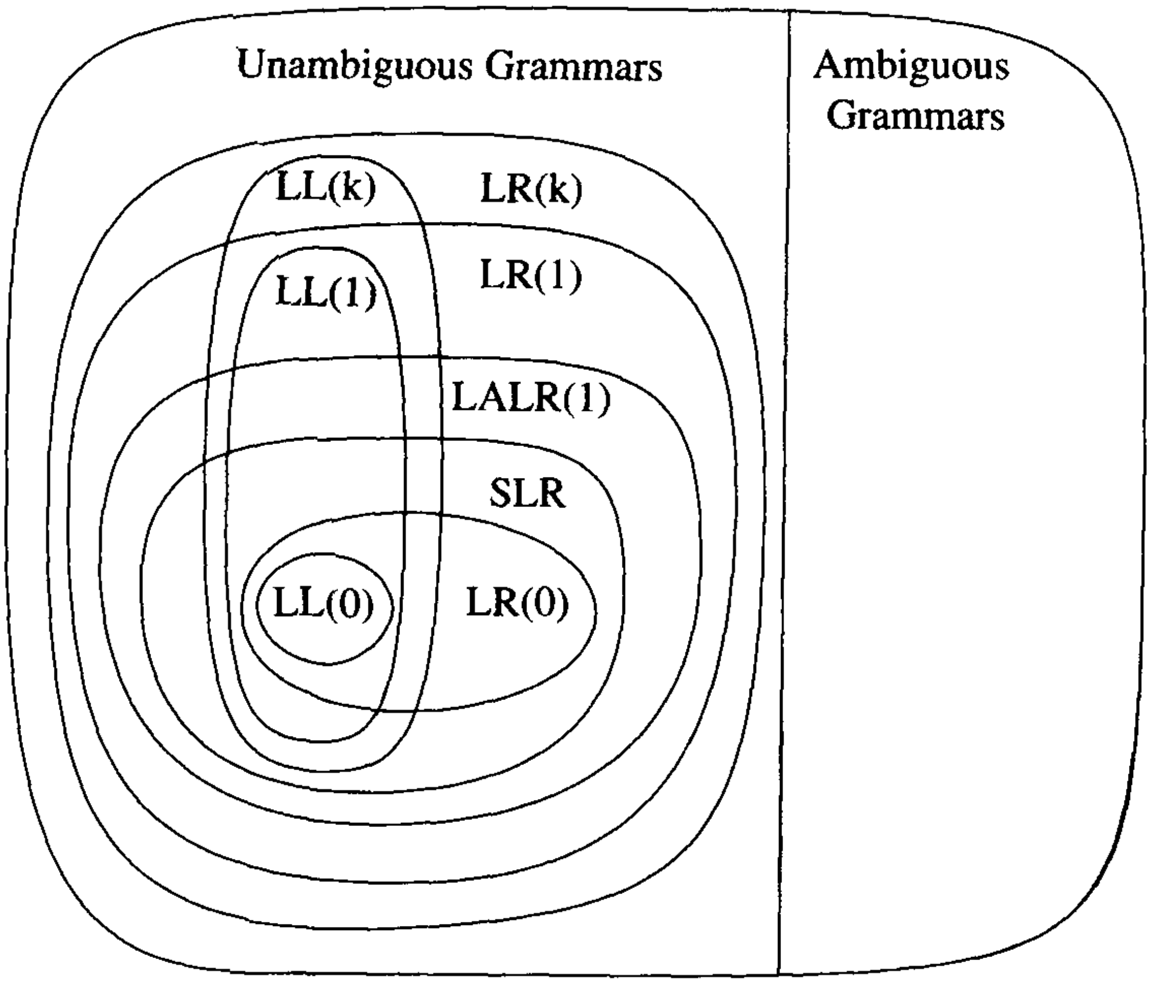
LALR vs. LR Parsing

- 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

$S \rightarrow P\ R\ ,$ $\qquad NL \rightarrow N \mid N\ ,\ NL$

$P \rightarrow T \mid NL : T \qquad R \rightarrow T \mid N : T$

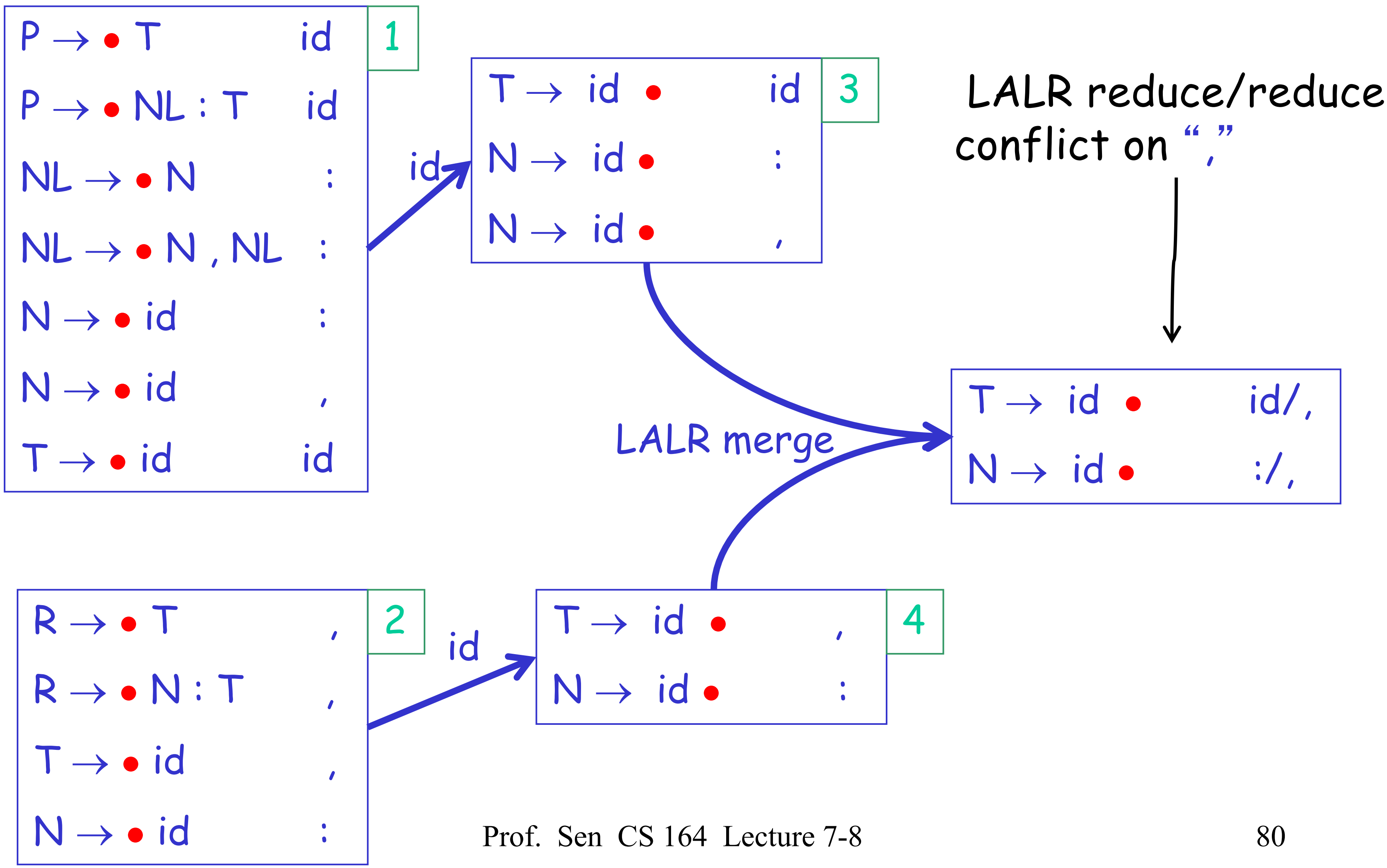
$N \rightarrow id \qquad T \rightarrow id$

- 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 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

$$R \rightarrow \text{id bogus}$$

  - bogus 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

State 1

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

$1 \xrightarrow{id} 3$

State 3

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

Different cores $\Rightarrow$ no LALR merging

State 2

$R \to .\ T$	,
$R \to .\ N : T$	,
$R \to .\ id\ bogus$	,
$T \to .\ id$	,
$N \to .\ id$	:

$2 \xrightarrow{id} 4$

State 4

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