Bottom-Up Parsing
LR Parsing. Parser Generators.

Lecture 6

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 = input string of terminals
repeat
    - Identify $\beta$ in $str$ such that $A \rightarrow \beta$ is a production
      (i.e., $str = \alpha \beta \gamma$)
    - Replace $\beta$ by $A$ in str (i.e., $str$ becomes $\alpha A \gamma$)
until str = $S$
```

A Bottom-up Parse in Detail (1)

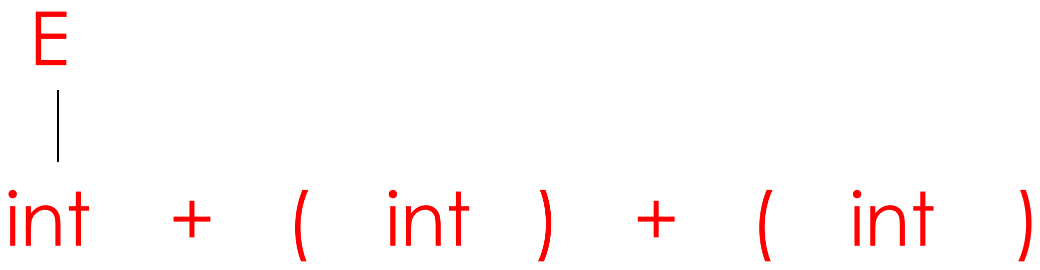
int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

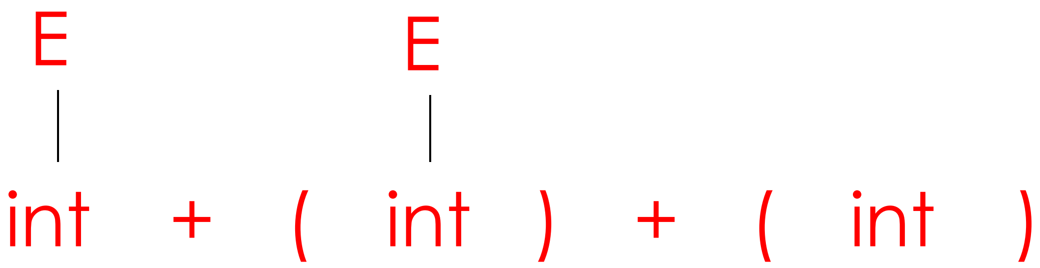


A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



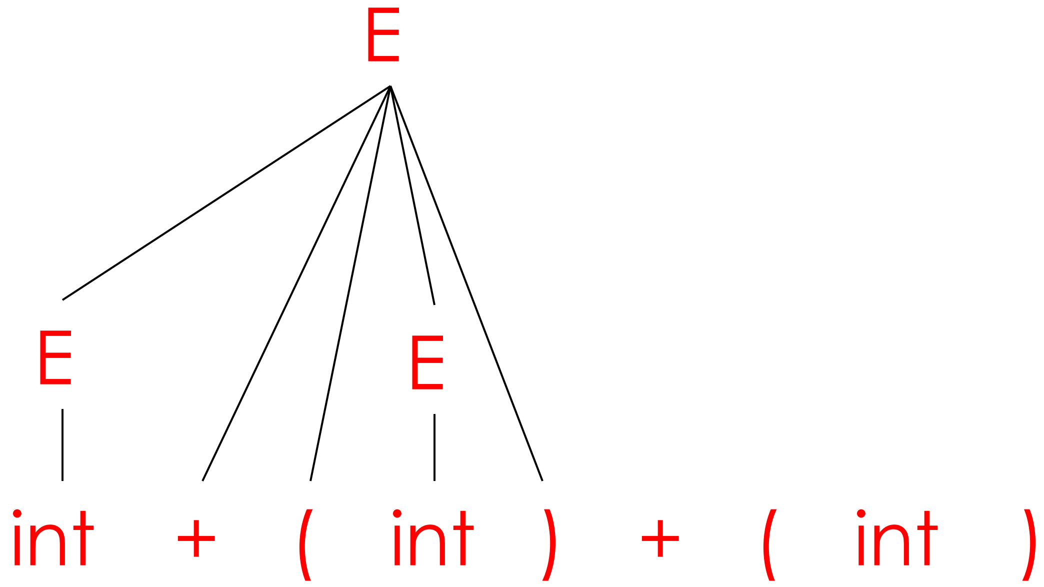
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

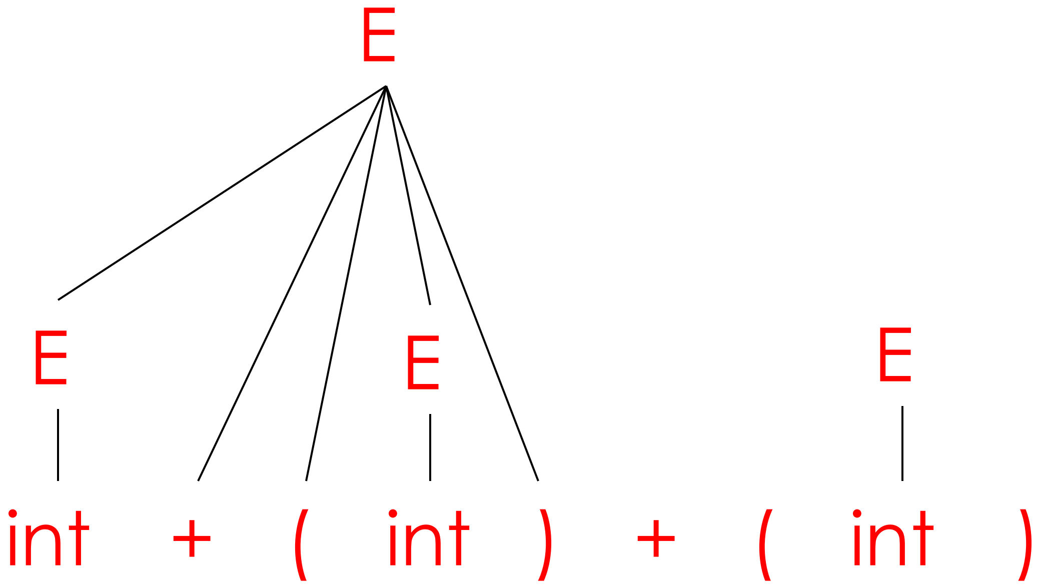
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)



A Bottom-up Parse in Detail (6)

int + (int) + (int)

E + (int) + (int)

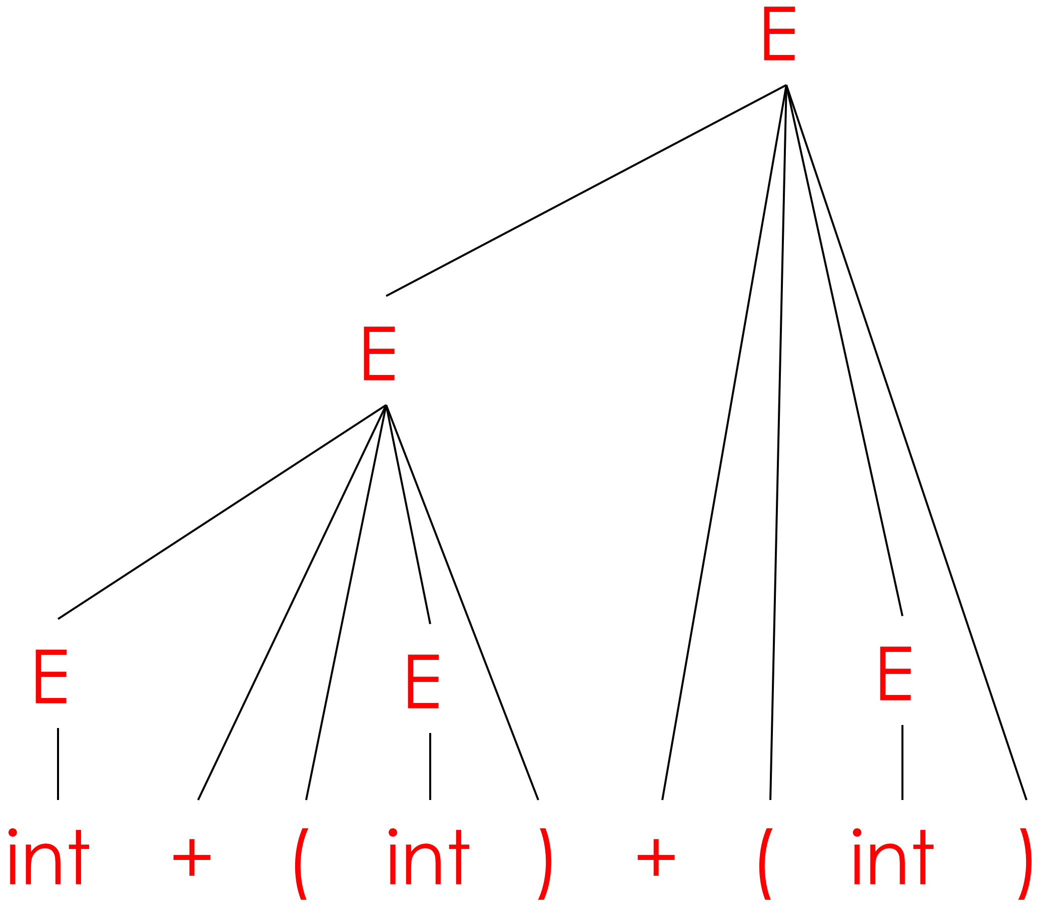
E + (E) + (int)

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 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

- Initially, all input is unexamined: I$x_1x_2 \ldots x_n$

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 + (| int) ⇒ E + (int |)

Reduce

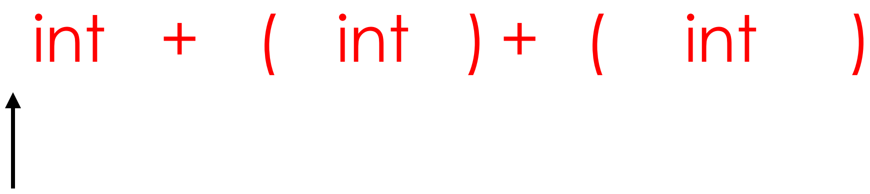
Reduce: Apply an inverse production at the right end of the left string

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

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

Shift-Reduce Example

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

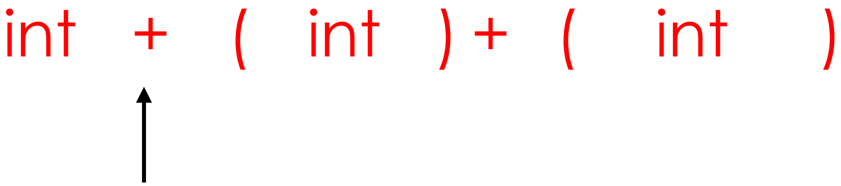


int + (int) + (int)

Shift-Reduce Example

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

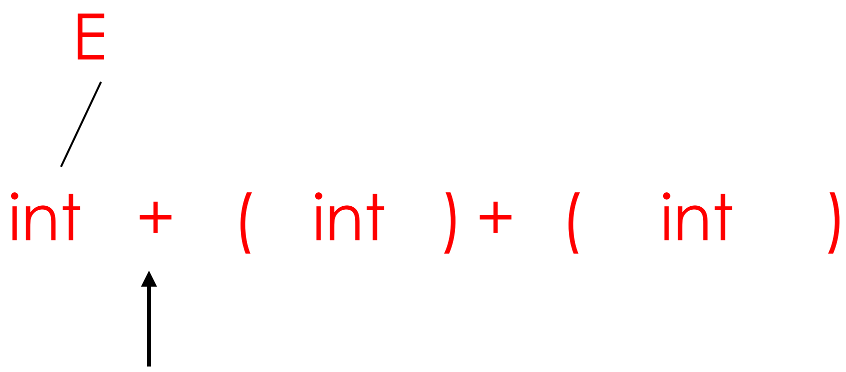
int | + (int) + (int)\$ red. E → int



int + (int) + (int)

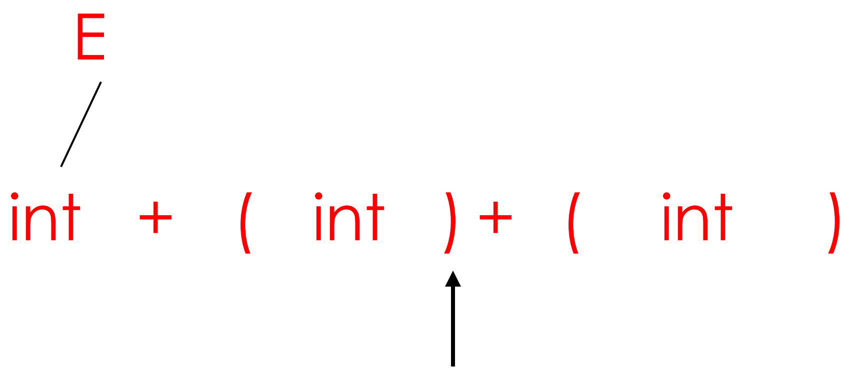
Shift-Reduce Example

| int + (int) + (int)$	shift
int | + (int) + (int)$	red. E → int
E | + (int) + (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

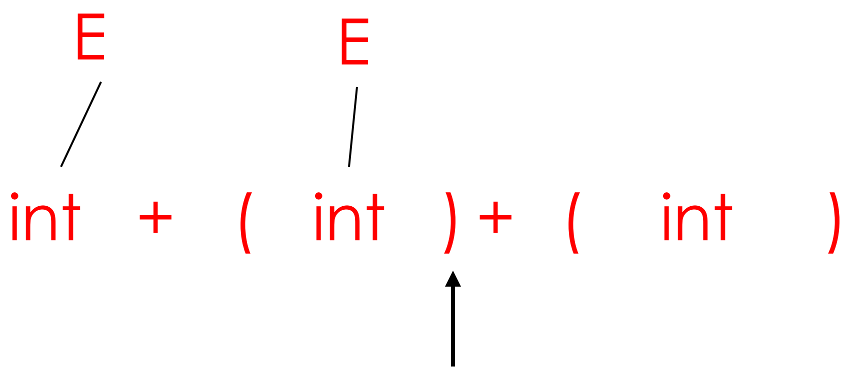


E

int + (int) + (int)

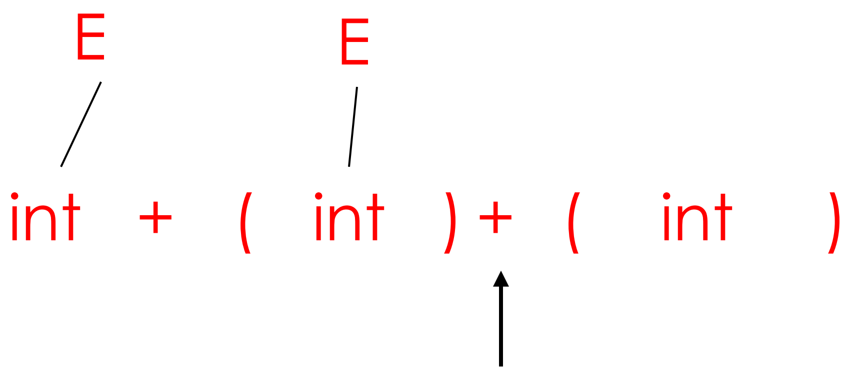
Shift-Reduce Example

| 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



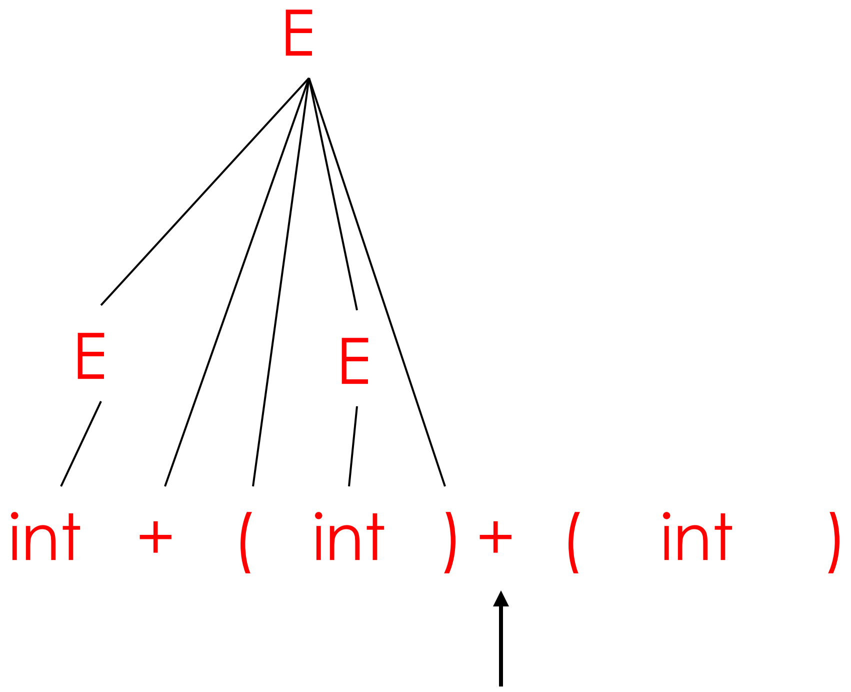
Shift-Reduce Example

| 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)



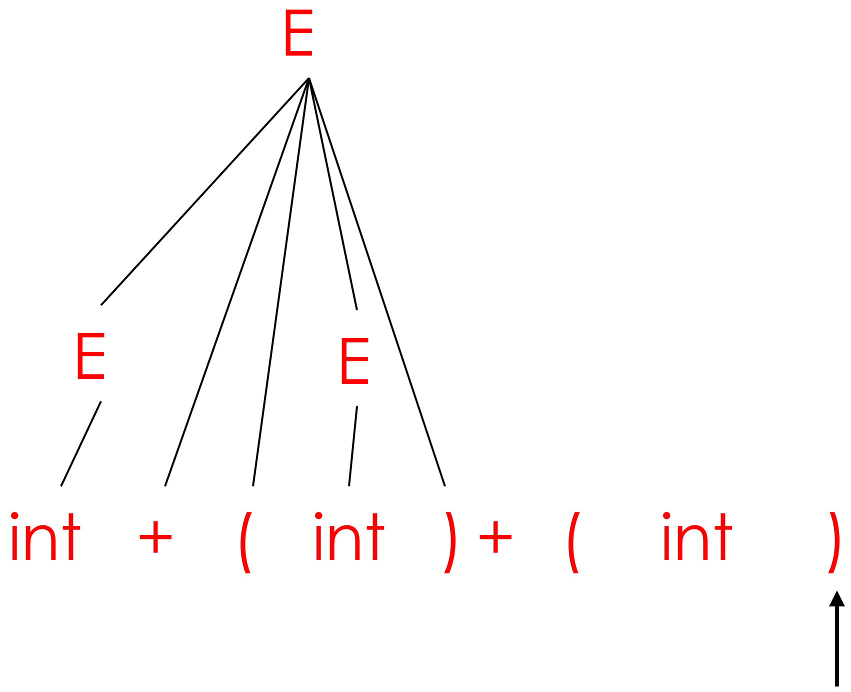
Shift-Reduce Example

| 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



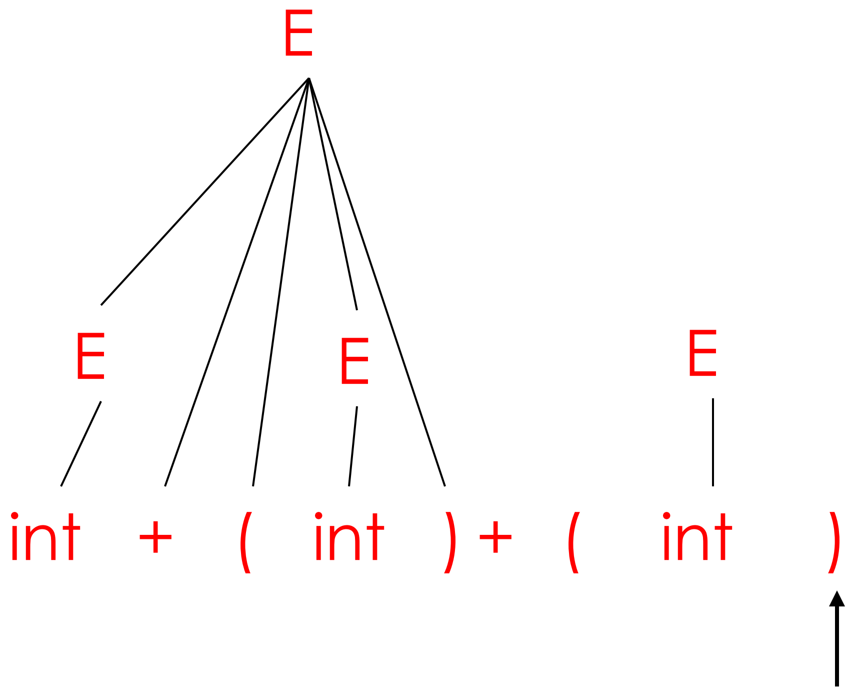
Shift-Reduce Example

| 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



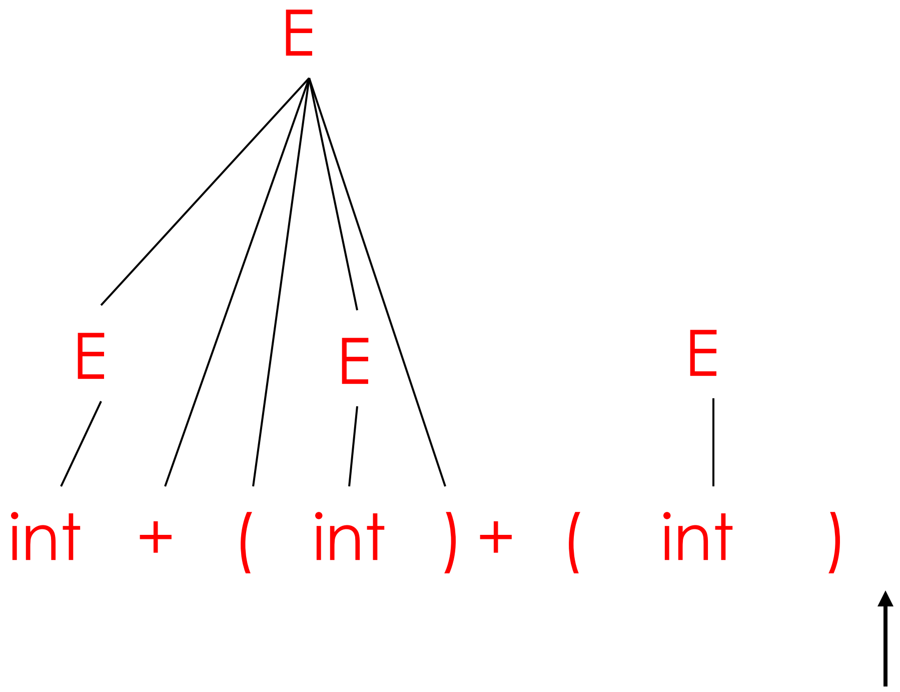
Shift-Reduce Example

❙ 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



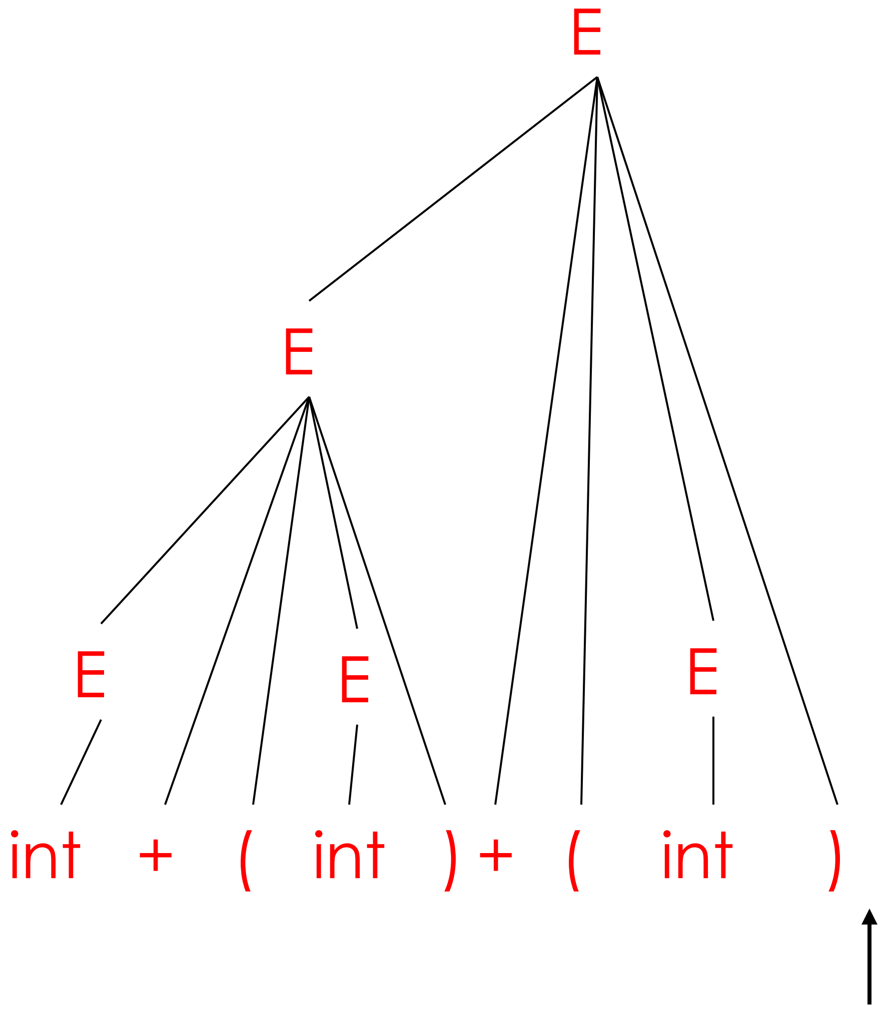
Shift-Reduce Example

| 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

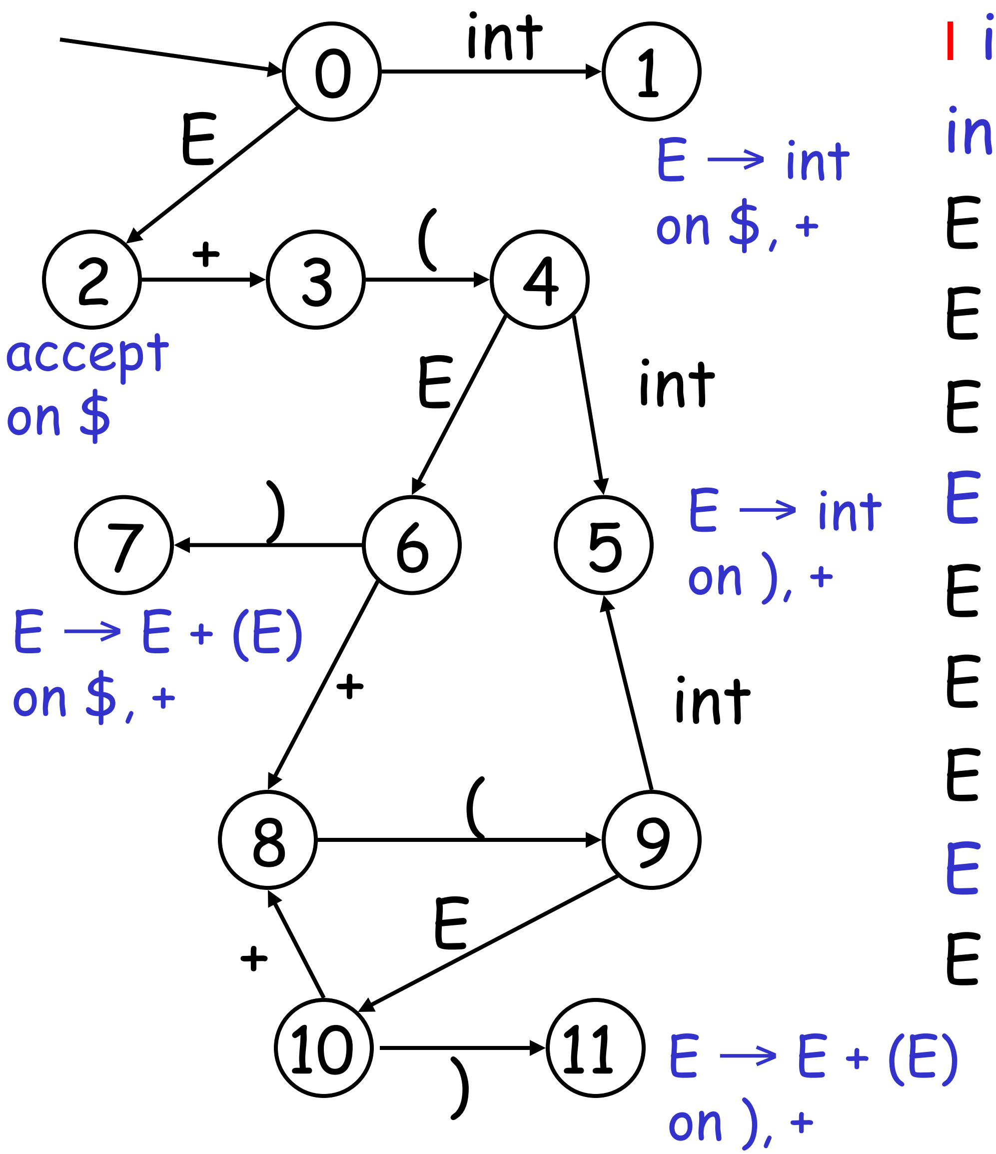
- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

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
  - 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 "$A \rightarrow \beta$ on tok" then reduce

LR(1) Parsing. An Example



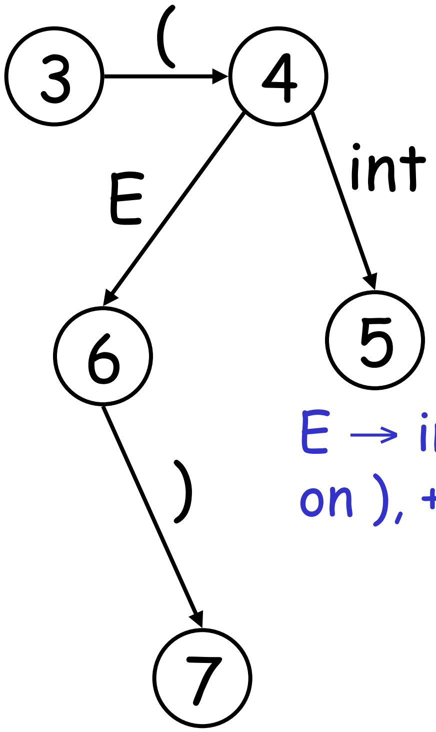
| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift(x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - 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:



$E \rightarrow int$
on), +

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

	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 on which state it brings the DFA
- LR parser maintains a stack

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

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

The LR Parsing Algorithm

```
Let I = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let 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?

Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What production rhs we are looking for
  - What we have seen so far from the rhs
- Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal E, we might be looking either for an int or a E + (E) rhs

LR(1) Items

- An <u>LR(1) item</u> is a pair:

  $$X \rightarrow \alpha.\beta,\ a$$

  - $X \rightarrow \alpha.\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

- $[X \rightarrow \alpha.\beta,\ a]$ 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

Note

- The symbol I was used before to separate the stack from the rest of input
  - α I γ, where α is the stack and γ is the remaining string of terminals
- In items . is used to mark a prefix of a production rhs:

$$X \rightarrow \alpha.\beta, a$$

  - 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 → E
  - Where E is the old start symbol

- The initial parsing context contains:

$$S \to .E, \$$$

  - Trying to find an S as a string derived from E$
  - The stack is empty

LR(1) Items (Cont.)

- In context containing

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

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

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

- In context containing

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

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

LR(1) Items (Cont.)

- Consider the item

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

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

  $E \rightarrow \text{int}$ and $E \rightarrow E + (E)$

- We describe this by extending the context with two more items:

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

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

The Closure Operation

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

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

Constructing the Parsing DFA (1)

- Construct the start context: Closure({S → .E, $})

S → .E, $
E → .E+(E), $
E → .int, $
E → .E+(E), +
E → .int, +

- We abbreviate as:

S → .E, $
E → .E+(E), $/+
E → .int, $/+

Constructing the Parsing DFA (2)

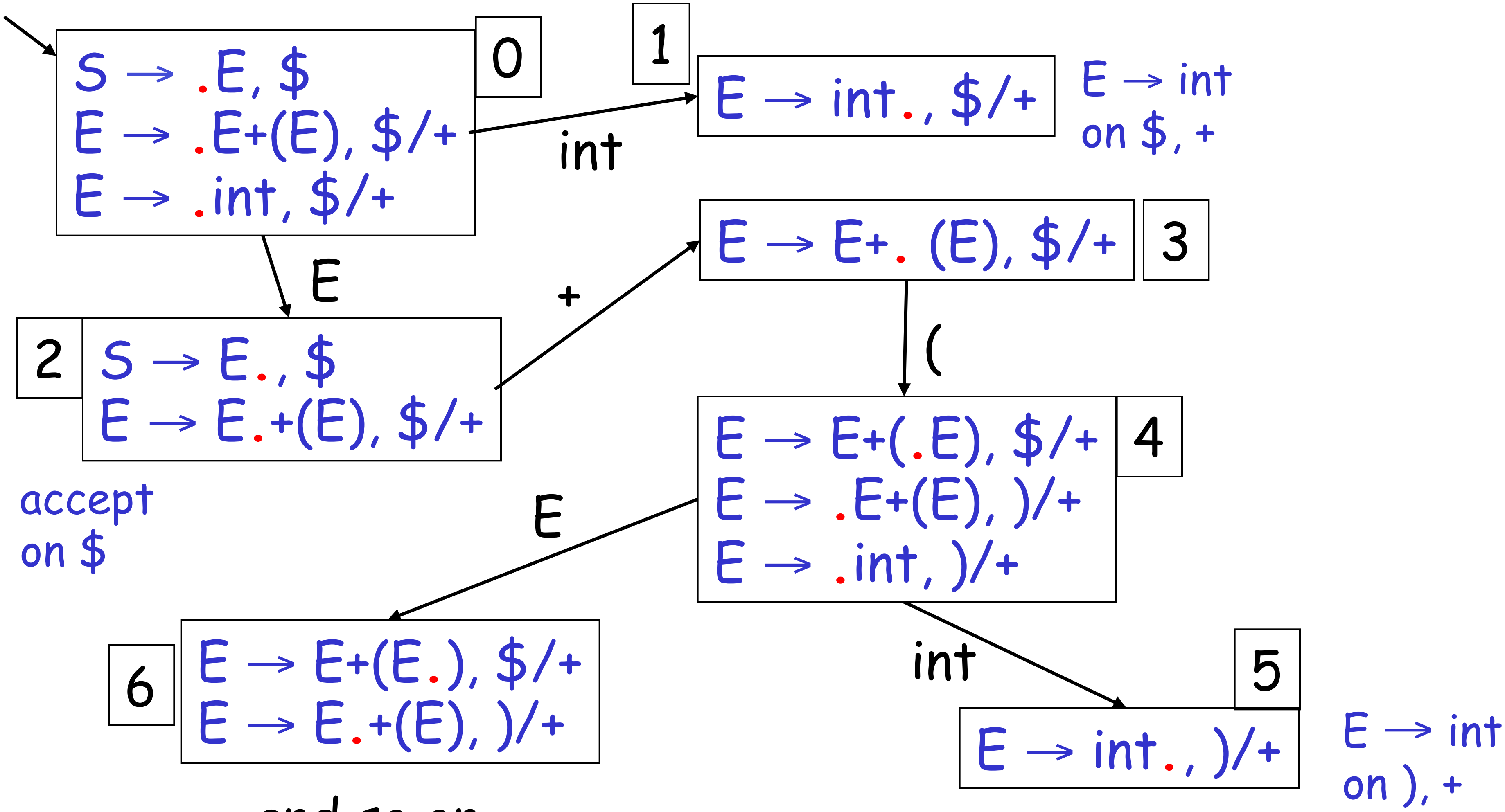
- A DFA state is a **closed** set of LR(1) items
- The start state contains $[S \rightarrow .E, \$]$
- A state that contains $[X \rightarrow \alpha., b]$ is labeled with "reduce with $X \rightarrow \alpha$ on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains $[X \rightarrow \alpha.y\beta, b]$ 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 = $\emptyset$
  for each $[X \rightarrow \alpha.y\beta, b] \in$ State
      add $[X \rightarrow \alpha y.\beta, b]$ to Items
  return Closure(Items)
```

Constructing the Parsing DFA. Example.



S → .E, $
E → .E+(E), $/+
E → .int, $/+

E → int., $/+

E → int
on $, +

S → E., $
E → E.+(E), $/+

accept
on $

E → E+. (E), $/+

E → E+(.E), $/+
E → .E+(E),)/+
E → .int,)/+

E → E+(E.), $/+
E → E.+(E),)/+

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

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

- Then on input "a" we could either
  - Shift into state $[X \to \alpha a.\beta, b]$, 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 → 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, x]
- 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

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

$$[E \rightarrow . E + E, \; +] \quad \xRightarrow{E} \quad [E \rightarrow E . + E, \; +]$$

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

- 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 declare precedence and associativity:

```
%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 <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:

$$
\begin{array}{lcl}
[E \to E * .\, E, +] & & [E \to E * E., +] \\
[E \to .\, E + E, +] & \Rightarrow^{E} & [E \to E \,.\, + E, +] \\
\ldots & & \ldots
\end{array}
$$

- 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

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

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

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

- 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

  $[X \rightarrow \alpha., a]$ and $[Y \rightarrow \beta., a]$

  - 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

$$\begin{array}{lcl} & & {[S \to id\,.,\quad \$]} \\ {[S' \to .\,S,\quad \$]} & & {[S \to id\,.\,S,\ \$]} \\ {[S \to .,\qquad \$]} & \overset{id}{\Rightarrow} & {[S \to .,\qquad \$]} \\ {[S \to .\,id,\quad \$]} & & {[S \to .\,id,\quad \$]} \\ {[S \to .\,id\ S,\ \$]} & & {[S \to .\,id\ S,\ \$]} \end{array}$$

- 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

E → int., $/+

E → int
on $, +

and

5

E → int.,)/+

E → 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 → int., $/+/)

E → int
on $, +,)

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

$$\{ [X \to \alpha.\beta, b], [Y \to \gamma.\delta, d]\}$$

is

$$\{X \to \alpha.\beta,\ Y \to \gamma.\delta\}$$

LALR States

- Consider for example the LR(1) states

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

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

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

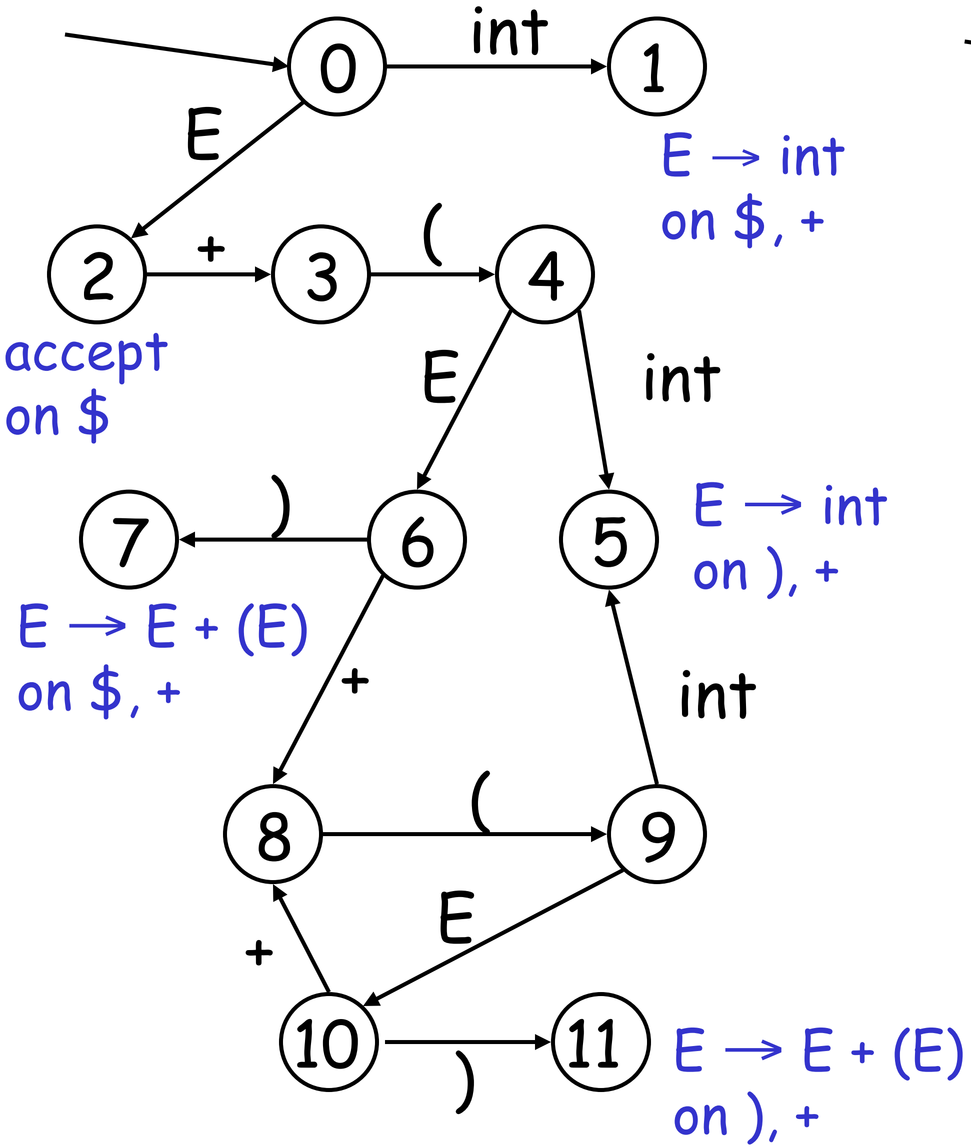
  $\{[X \to \alpha., a/b], [Y \to \beta., c/d]\}$

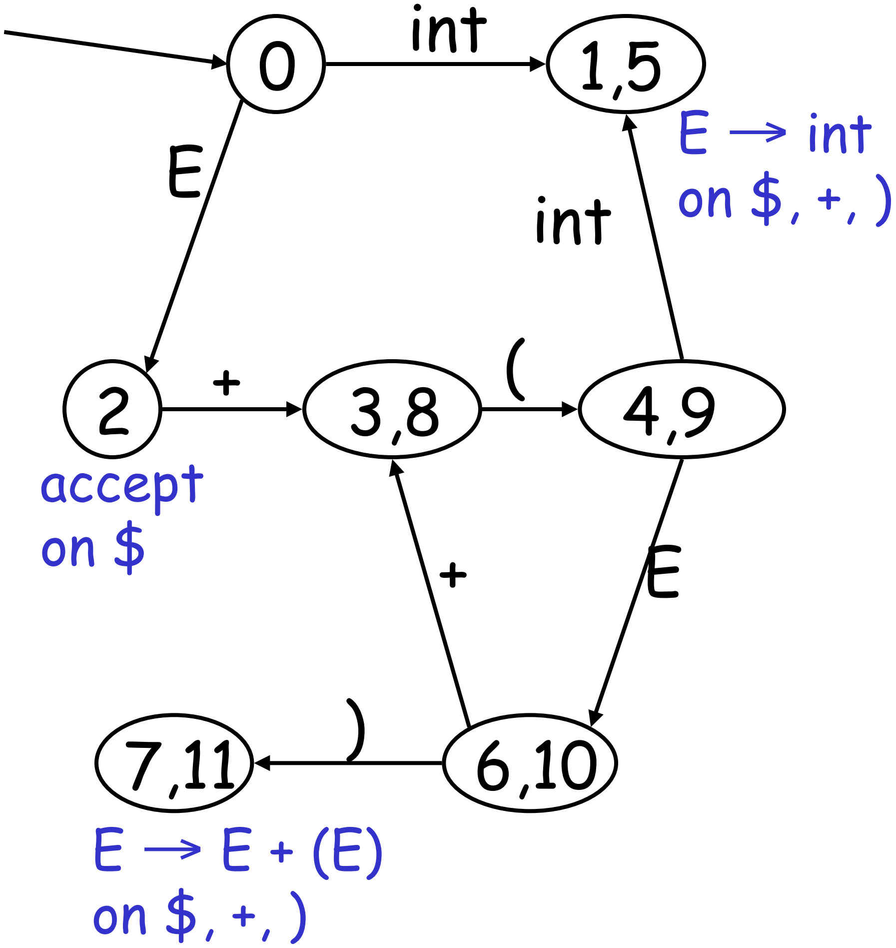
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

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

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

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

- And the merged LALR(1) state

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

- Has a new reduce-reduce conflict

- In practice such cases are rare

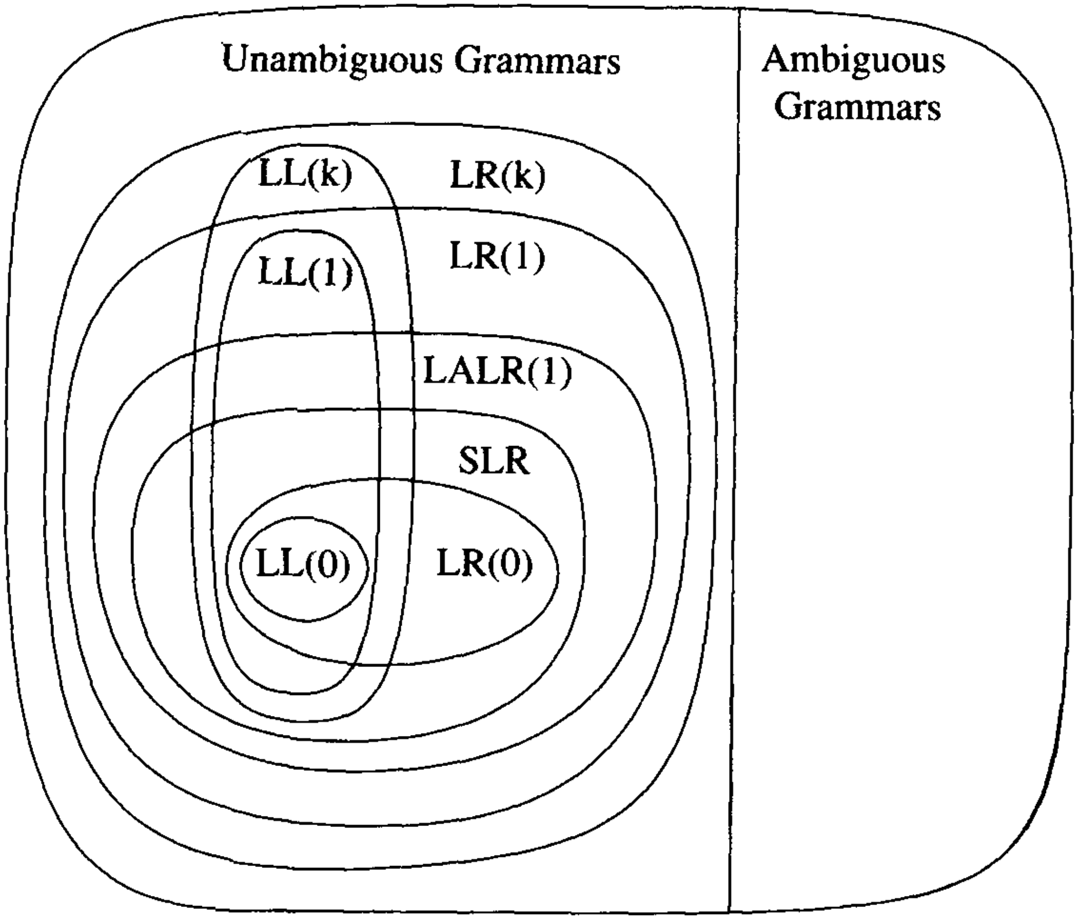
LALR vs. LR Parsing

- 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)
  - 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

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

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

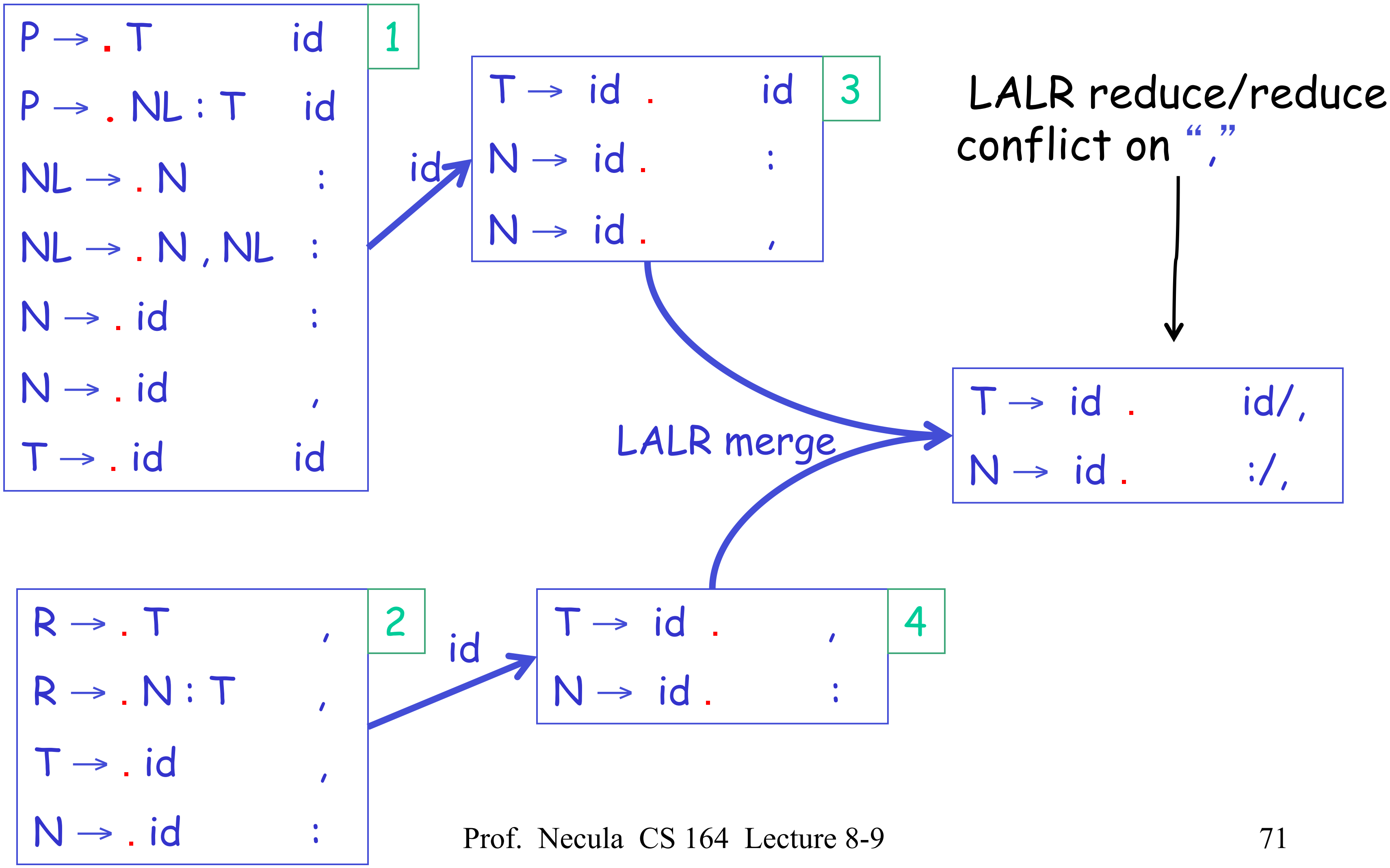
$N \to id$ $\qquad T \to id$

- 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



P → . T	id	1
P → . NL : T	id	
NL → . N	:	
NL → . N , NL	:	
N → . id	:	
N → . id	,	
T → . id	id	

T → id .	id	3
N → id .	:	
N → id .	,	

LALR reduce/reduce conflict on “,”

T → id .	id/,
N → id .	:/,

R → . T	,	2
R → . N : T	,	
T → . id	,	
N → . id	:	

T → id .	,	4
N → id .	:	

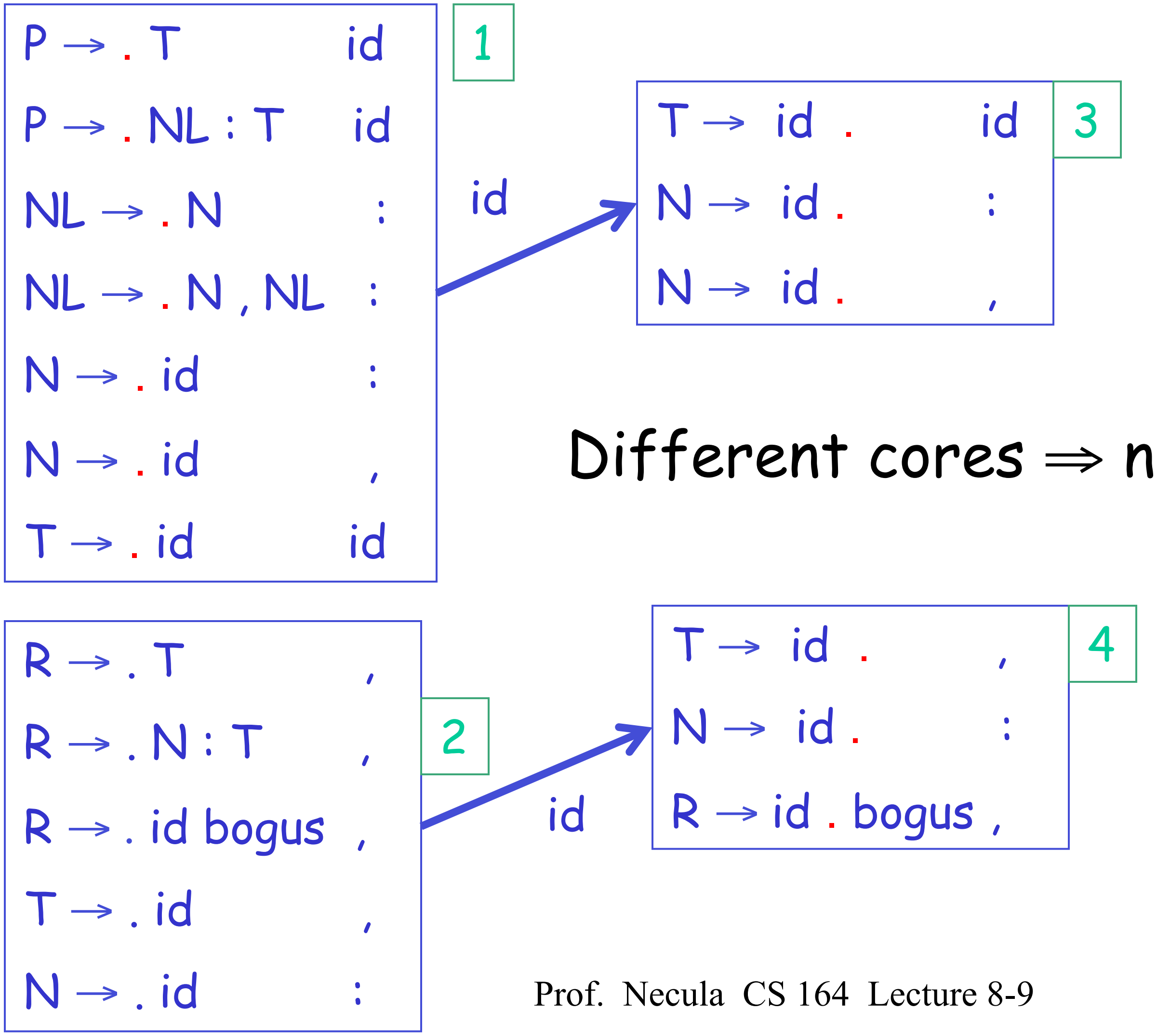
What Happened?

- Two distinct states were confused because they have the same core
- Fix: add dummy productions to distinguish the two confused states
- E.g., add

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

  - 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



1

P → . T	id
P → . NL : T	id
NL → . N	:
NL → . N , NL	:
N → . id	:
N → . id	,
T → . id	id

id

3

T → id .	id
N → id .	:
N → id .	,

2

R → . T	,
R → . N : T	,
R → . id bogus	,
T → . id	,
N → . id	:

id

4

T → id .	,
N → id .	:
R → id . bogus	,

Different cores ⇒ no LALR merging