LR Parsing
LALR Parser Generators

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

  α | γ

  - α is a stack of terminals and non-terminals
  - γ is the string of terminals not yet examined
- Initially: | $x_1 x_2 \ldots x_n$

The Shift and Reduce Actions (Review)

- Recall the CFG: $E \rightarrow$ int | E + (E)
- A bottom-up parser uses two kinds of actions:
- Shift pushes a terminal from input on the stack

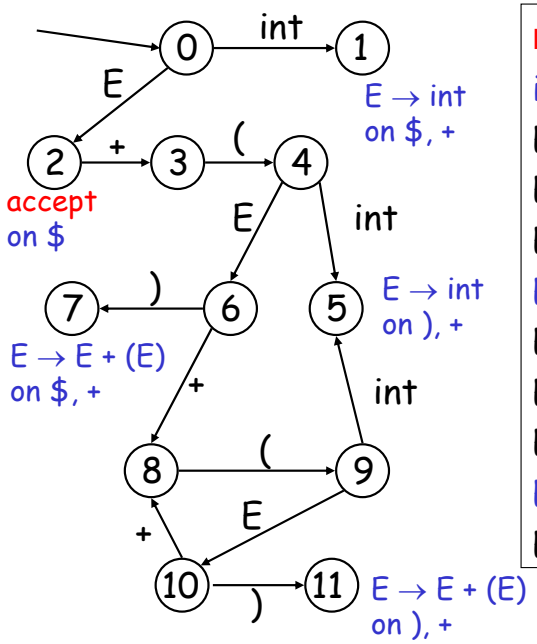
  E + (| int) $\Rightarrow$ E + (int |)
- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

  E + (E + (E) |) $\Rightarrow$ E + (E |)

Key Issue: When to Shift or Reduce?

- Idea: use a deterministic 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 I
  - 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



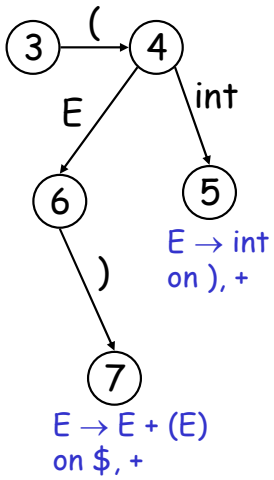
I int + (int) + (int)$	shift
int I + (int) + (int)$	E → int
E I + (int) + (int)$	shift (x3)
E + (int I) + (int)$	E → int
E + (E I) + (int)$	shift
E + (E) I + (int)$	E → E+(E)
E I + (int)$	shift (x3)
E + (int I)$	E → int
E + (E I)$	shift
E + (E) I $	E → E+(E)
E I $	accept

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: the action table
  - Those for non-terminals: the 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		$r_{E \rightarrow E+(E)}$			$r_{E \rightarrow E+(E)}$	
...						

sk is shift and goto state k
$r_{X \rightarrow \alpha}$ is reduce
gk is goto state k

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

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

$state_k$ 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 → A:
                     pop |A| pairs,
                     push ⟨ X, goto[top_state(stack), X] ⟩
                accept: halt normally
                error: halt and report error
```


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 an E + (E) RHS

LR(0) Items

- An LR(0) item is a production with a "•" somewhere on the RHS
- The items for $T \rightarrow (E)$ are

  $T \rightarrow \bullet (E)$

  $T \rightarrow (\bullet E)$

  $T \rightarrow (E \bullet)$

  $T \rightarrow (E) \bullet$
- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow \bullet$

LR(0) Items: Intuition

- An item $[X \rightarrow \alpha \mid \beta]$ says that
  - the parser is looking for an X
  - it has an α on top of the stack
  - Expects to find a string derived from β next in the input
- Notes:
  - $[X \rightarrow \alpha \mid a\beta]$ means that a should follow. Then we can shift it and still have a viable prefix
  - $[X \rightarrow \alpha \mid]$ means that we could reduce X
    - But this is not always a good idea !

LR(1) Items

- An LR(1) item is a pair:

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

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

Note

- The symbol $\mid$ was used before to separate the stack from the rest of input
  - $\alpha \mid \gamma$, where α is the stack and γ is the remaining string of terminals
- In items $\mid$ is used to mark a prefix of a production RHS:

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

  - Here β might contain terminals as well
- In both case the stack is on the left of $\mid$

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
- The initial parsing context contains:

$$S \rightarrow \mid E \ , \$$$

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

LR(1) Items (Cont.)

- In context containing

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

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

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

- In context containing

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

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

LR(1) Items (Cont.)

- Consider the item

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

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

  $E \rightarrow int$ and $E \rightarrow E + (E)$

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

  $E \rightarrow \cdot int$,)

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

The Closure Operation

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

```
Closure(Items) =
  repeat
    for each [X → α · Yβ, a] in Items
      for each production Y → γ
        for each b in First(βa)
          add [Y → · γ, b] to Items
  until Items is unchanged
```


Constructing the Parsing DFA (1)

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

  $S \rightarrow \cdot E$, \$
  $E \rightarrow \cdot E+(E)$, \$
  $E \rightarrow \cdot int$, \$
  $E \rightarrow \cdot E+(E)$, +
  $E \rightarrow \cdot int$, +

- We abbreviate as:

  $S \rightarrow \cdot E$, \$
  $E \rightarrow \cdot E+(E)$, \$/+
  $E \rightarrow \cdot int$, \$/+

Constructing the Parsing DFA (2)

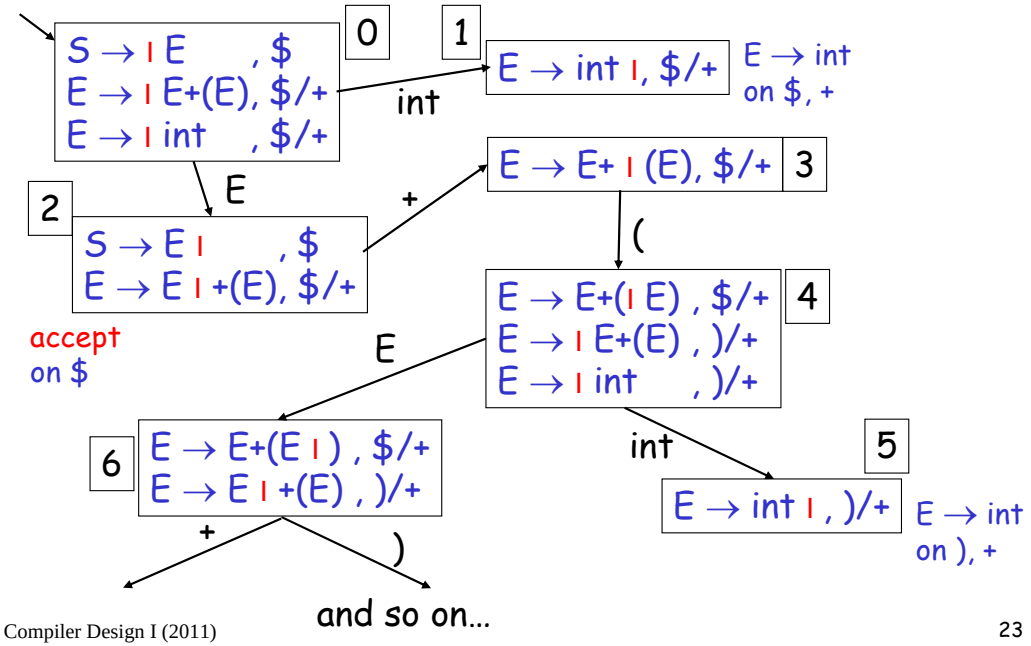
- A DFA state is a closed set of LR(1) items
- The start state contains [S → ı E , $]
- A state that contains [X → α ı, b] is labelled with "reduce with X → α on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains [X → α ı yβ, 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 = ∅
  for each [X → α ı yβ, b]  in State
      add [X → αy ı β, 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
  $[X \to \alpha \mid a\beta, b]$ and $[Y \to \gamma \mid, a]$
- Then on input "a" we could either
  - Shift into state $[X \to \alpha a \mid \beta, b]$, 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
  $S \to$ 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 (yacc, ML-yacc, etc.) is to shift
  - Default behavior is as needed in this case

More Shift/Reduce Conflicts

- Consider the ambiguous grammar
  $E \to E + E \mid E * E \mid$ int
- We will have the states containing
  $[E \to E * \mid E, +]$ $[E \to E * E \mid, +]$
  $[E \to \mid E + E, +]$ $\Rightarrow^E$ $[E \to E \mid + E, +]$
  ... ...
- 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 yacc declare precedence and associativity:
  ```
  %left +
  %left *
  ```
- Precedence of a rule = that of its last terminal
  See yacc 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:

  $[E \to E * \mid E,\ +]$ $\quad$ $[E \to E * E \mid,\ +]$

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

  ... $\quad$ ...

- 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

  $[E \to E + \mid E,\ +]$ $\quad$ $[E \to E + E \mid,\ +]$

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

  ... $\quad$ ...

- 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

  $[S \to \text{if } E \text{ then } S \mid,$ $\quad$ else$]$

  $[S \to \text{if } E \text{ then } S \mid \text{else } S,$ $\quad x]$

- Can eliminate conflict by declaring else having higher precedence than then
- But this starts to look like "hacking the tables"
- Best to avoid overuse of precedence declarations or we will end with unexpected parse trees

Precedence Declarations Revisited

The term "precedence declaration" is misleading!

These declarations do not define precedence: they define conflict resolutions

I.e., they instruct shift-reduce parsers to resolve conflicts in certain ways

The two are not quite the same thing!

Reduce/Reduce Conflicts

- If a DFA state contains both

  $[X \to \alpha \,\mathrm{I}, a]$ and $[Y \to \beta \,\mathrm{I}, a]$

  - 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

  $S \to \varepsilon \mid id \mid id\ S$

- There are two parse trees for the string id

  $S \to id$

  $S \to id\ S \to id$

- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states

  $[S' \to \mathrm{I}\ S, \$]$ $\Rightarrow^{id}$ $[S \to id\ \mathrm{I}, \$]$

  $[S \to \mathrm{I}, \$]$ $\Rightarrow^{id}$ $[S \to id\ \mathrm{I}\ S, \$]$

  $[S \to \mathrm{I}\ id, \$]$ $\Rightarrow^{id}$ $[S \to \mathrm{I}, \$]$

  $[S \to \mathrm{I}\ id\ S, \$]$ $\Rightarrow^{id}$ $[S \to \mathrm{I}\ id, \$]$

  $[S \to \mathrm{I}\ id\ S, \$]$

- Reduce/reduce conflict on input $

  $S' \to S \to id$

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

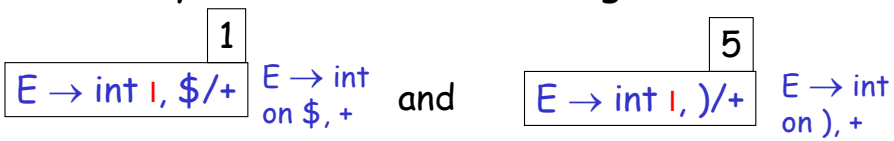
- Better rewrite the grammar: $S \to \varepsilon \mid id\ S$

Using Parser Generators

- Parser generators automatically 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

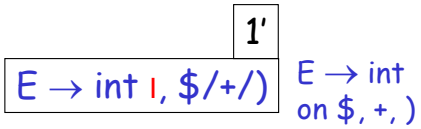
LR(1) Parsing Tables are Big

- 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

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

is

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

LALR States

- Consider for example the LR(1) states

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

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

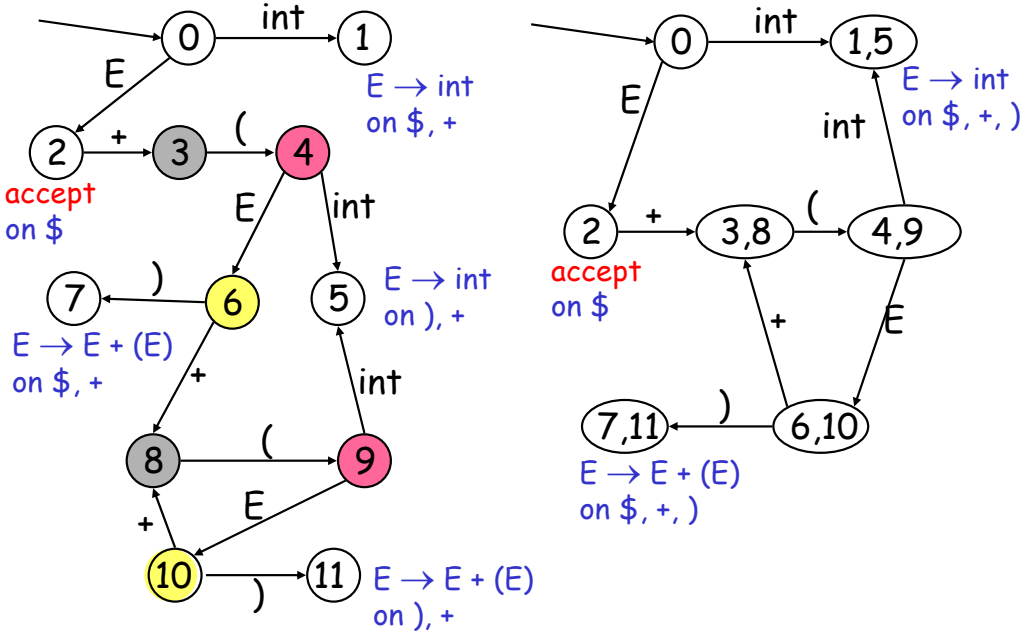
$$\{[X \to \alpha \cdot, a/b], [Y \to \beta \cdot, c/d]\}$$

- 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.



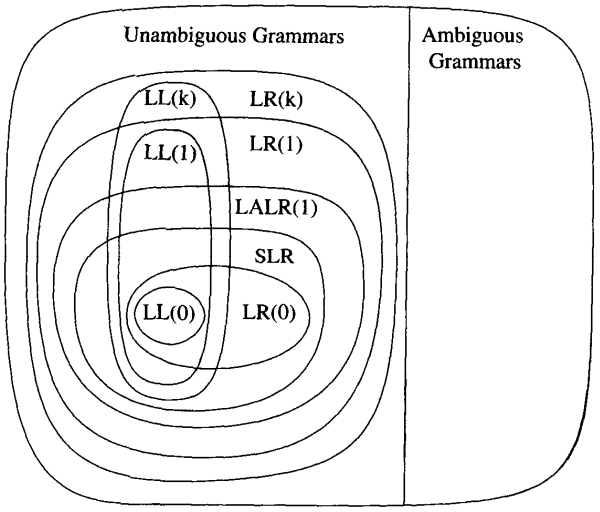
The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states
  {[X → α I, a], [Y → β I, b]}
  {[X → α I, b], [Y → β I, a]}
- And the merged LALR(1) state
  {[X → α I, a/b], [Y → β I, a/b]}
- Has a <u>new</u> reduce/reduce conflict
- In practice such cases are rare

LALR vs. LR Parsing: Things to keep in mind

- LALR languages are not natural
  - They are an efficiency hack on LR languages
- Any reasonable programming language has a LALR(1) grammar
- LALR(1) parsing has become a standard for programming languages and for parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in ML"

Semantic Actions in LR Parsing

- We can now illustrate how semantic actions are implemented for LR parsing
- Keep attributes on the stack

- On shifting a, push attribute for a on stack
- On reduce $X \rightarrow \alpha$
  - pop attributes for α
  - compute attribute for X
  - and push it on the stack

Performing Semantic Actions: Example

- Recall the example

$E \rightarrow T + E_1$ { E.val = T.val + E_1.val }
| T { E.val = T.val }
$T \rightarrow int * T_1$ { T.val = int.val * T_1.val }
| int { T.val = int.val }

- Consider the parsing of the string **3 * 5 + 8**

Performing Semantic Actions: Example

| int * int + int	shift
int_3 | * int + int	shift
int_3 * | int + int	shift
int_3 * int_5 | + int	reduce $T \rightarrow int$
int_3 * T_5 | + int	reduce $T \rightarrow int * T$
T_{15} | + int	shift
T_{15} + | int	shift
T_{15} + int_8 |	reduce $T \rightarrow int$
T_{15} + T_8 |	reduce $E \rightarrow T$
T_{15} + E_8 |	reduce $E \rightarrow T + E$
E_{23} |	accept

3 * 5 + 8

Notes

- The previous example shows how synthesized attributes are computed by LR parsers

- It is also possible to compute inherited attributes in an LR parser

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
- Next time we move on to semantic analysis

Supplement to LR Parsing

Strange Reduce/Reduce Conflicts
due to LALR Conversion
(and how to handle them)

Strange Reduce/Reduce Conflicts

- Consider the grammar

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

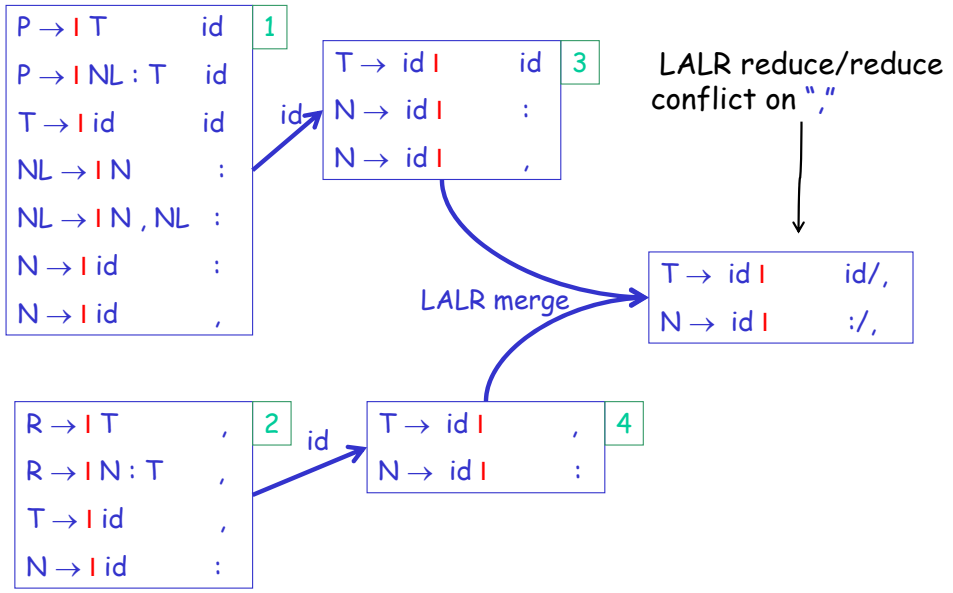
  $P \rightarrow T \mid NL\ :\ T$ $\quad R \rightarrow T \mid N\ :\ T$

  $N \rightarrow id$ $\quad T \rightarrow id$
- 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

  $R \rightarrow$ 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