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

$$\alpha \mid \gamma$$

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

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

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

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

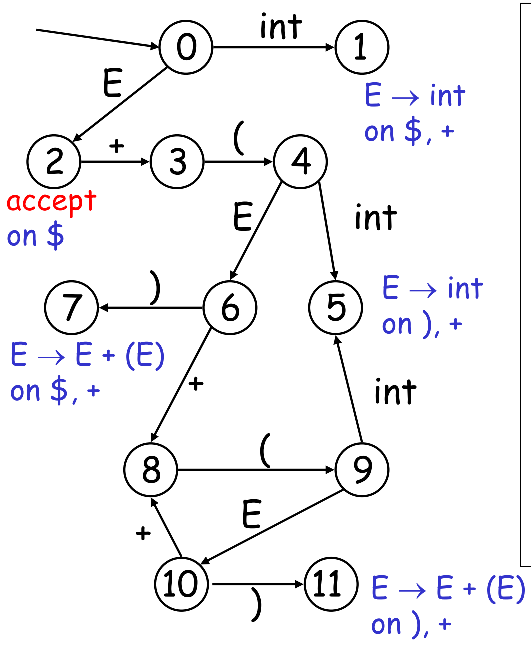
- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

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

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 |
  - If X has a transition labeled tok then <u>shift</u>
  - If X is labeled with "$A \to \beta$ on tok" then <u>reduce</u>

LR(1) Parsing: An Example



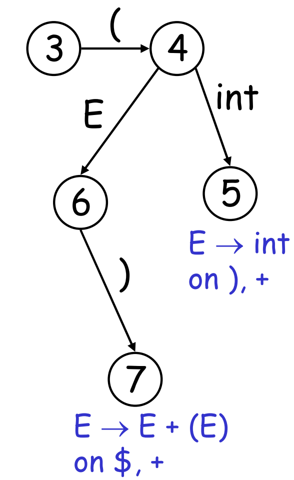
| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift (x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

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: 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		$r_{E \to int}$		$r_{E \to int}$		
6	s8		s7			
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
...						

sk is shift and goto state k
$r_{X \to \alpha}$ 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 <u>LR(0) item</u> is a production with a "ı" somewhere on the RHS
- The items for $T \to (E)$ are
  - $T \to\ ı\ (E)$
  - $T \to (\ ı\ E)$
  - $T \to (E\ ı\ )$
  - $T \to (E)\ ı$
- The only item for $X \to \varepsilon$ is $X \to\ ı$

LR(0) Items: Intuition

- An item $[X \to \alpha \mid \beta]$ 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:
  - $[X \to \alpha \mid a\beta]$ means that a should follow. Then we can shift it and still have a viable prefix
  - $[X \to \alpha \mid]$ means that we could reduce X
    - But this is not always a good idea !

LR(1) Items

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

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

  - $X \to \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- $[X \to \alpha \mid \beta, a]$ 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:

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

  - 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 $S \to E$
  - Where E is the old start symbol
- The initial parsing context contains:

$$S \to \mid E\ \ , \$$$

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

LR(1) Items (Cont.)

- In context containing

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

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

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

- In context containing

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

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

LR(1) Items (Cont.)

- Consider the item

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

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

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

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

  $E \to \mid \text{int} \quad ,)$

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

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)

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

- Construct the start context:

  Closure({$S \to \mid E, \$$})

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

- We abbreviate as:

  $S \to \mid E \quad , \$$
  $E \to \mid E+(E) \ , \$/+$
  $E \to \mid \text{int} \quad , \$/+$

Constructing the Parsing DFA (2)

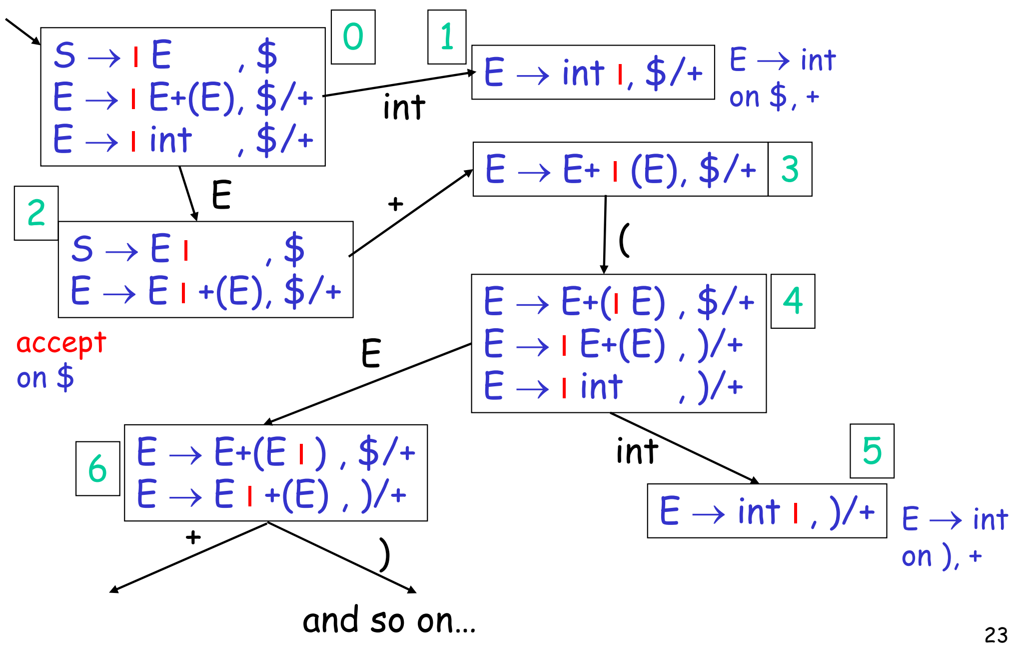
- A DFA state is a closed set of LR(1) items
- The start state contains [S → • E , $]
- A state that contains [X → α •, b] is labelled with "reduce with X → α on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains [X → α • 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 • yβ, b] in State
      add [X → αy • 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
  $[X \to \alpha \cdot a\beta,\ b]$ and $[Y \to \gamma \cdot,\ a]$
- Then on input "a" we could either
  - Shift into state $[X \to \alpha a \cdot \beta,\ b]$, or
  - Reduce with $Y \to \gamma$
- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  S → if E then S | if E then S else S | OTHER
- Will have DFA state containing
  [S → if E then S •, else]
  [S → if E then S • else S, 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 → E + E | E * E | int
- We will have the states containing
  [E → E * • E, +] [E → E * E •, +]
  [E → • E + E, +] $\Rightarrow^E$ [E → E • + 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 <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 → E * | E, +] [E → E * E |, +]

  [E → | E + E, +] ⇒^{E} [E → E | + E, +]

  … …

- Will choose reduce because precedence of rule E → E * E is higher than of terminal +

Using Precedence to Solve S/R Conflicts

- Same grammar as before

  E → E + E | E * E | int

- We will also have the states

  [E → E + | E, +] [E → E + E |, +]

  [E → | E + E, +] ⇒^{E} [E → E | + E, +]

  … …

- Now we also have a shift/reduce on input +
  - We choose reduce because E → 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 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

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

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

- Reduce/reduce conflict on input \$

  $S' \to S \to id$

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

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

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.

State 1: $E \to int \bullet, \$/+$ ($E \to int$ on \$, +)

and

State 5: $E \to int \bullet,)/+$ ($E \to int$ 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

State 1′: $E \to int \bullet, \$/+/)$ ($E \to int$ 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

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

is

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

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:

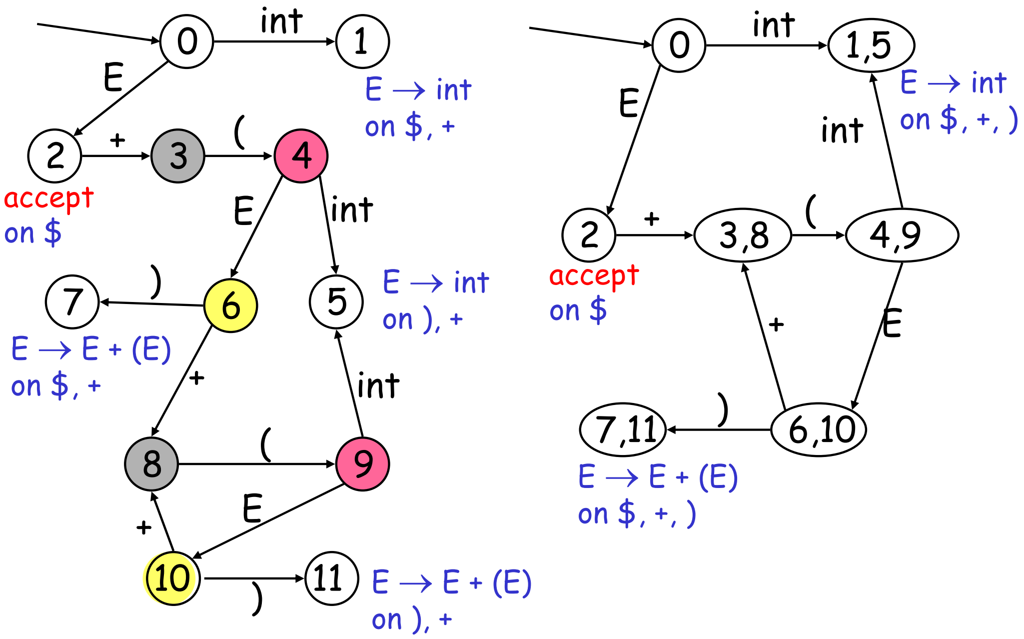
$$\{[X \to \alpha \bullet, a/b], [Y \to \beta \bullet, c/d]\}$$

- 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

  $\{[X \rightarrow \alpha\,\mathsf{I},\ a],\ [Y \rightarrow \beta\,\mathsf{I},\ b]\}$

  $\{[X \rightarrow \alpha\,\mathsf{I},\ b],\ [Y \rightarrow \beta\,\mathsf{I},\ a]\}$

- And the merged LALR(1) state

  $\{[X \rightarrow \alpha\,\mathsf{I},\ a/b],\ [Y \rightarrow \beta\,\mathsf{I},\ a/b]\}$

- Has a new reduce/reduce conflict

- In practice such cases are rare

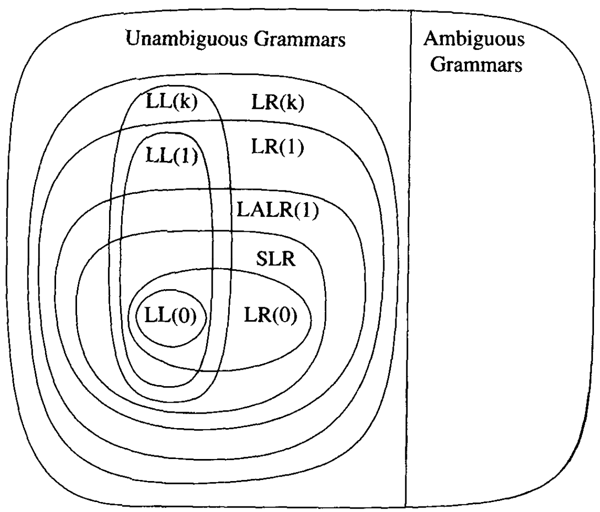
LALR vs. LR Parsing: Things to keep in mind

- 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 $X \to \alpha$
  - pop attributes for α
  - compute attribute for X
  - and push it on the stack

Performing Semantic Actions: Example

Recall the example

$E \to T + E_1$	{ $E.val = T.val + E_1.val$ }
$\mid T$	{ $E.val = T.val$ }
$T \to int * T_1$	{ $T.val = int.val * T_1.val$ }
$\mid int$	{ $T.val = int.val$ }

Consider the parsing of the string: **4 * 9 + 6**

Performing Semantic Actions: Example

4 * 9 + 6

| int * int + int	shift
int_4 | * int + int	shift
int_4 * | int + int	shift
int_4 * int_9 | + int	reduce $T \to int$
int_4 * T_9 | + int	reduce $T \to int * T$
T_{36} | + int	shift
T_{36} + | int	shift
T_{36} + int_6 |	reduce $T \to int$
T_{36} + T_6 |	reduce $E \to T$
T_{36} + E_6 |	reduce $E \to T + E$
E_{42} |	accept

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

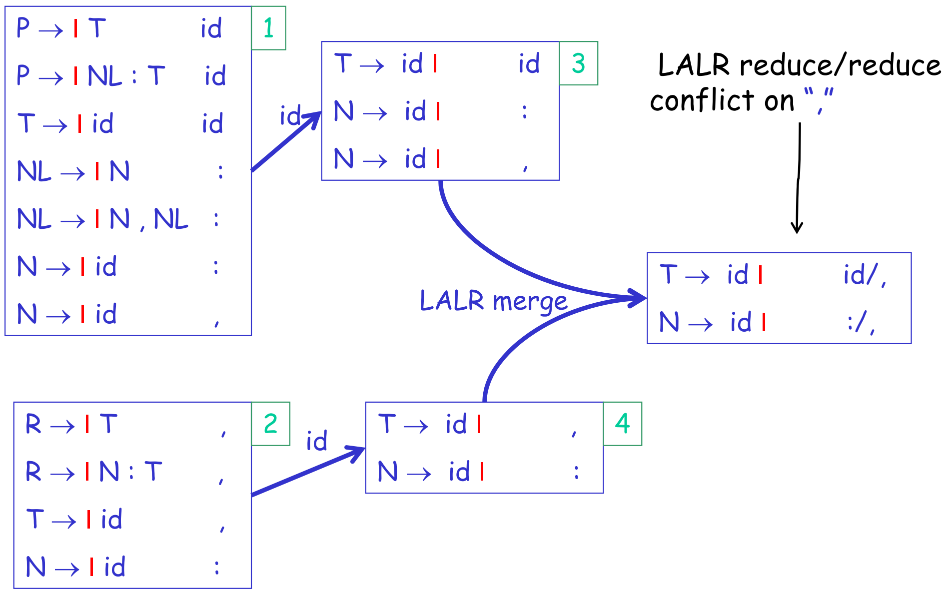
$$S \rightarrow P\ R\ , \qquad NL \rightarrow N \mid N\ ,\ NL$$
$$P \rightarrow T \mid NL : T \qquad R \rightarrow T \mid N : T$$
$$N \rightarrow id \qquad T \rightarrow id$$

- 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 → 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