Conflicts

Shift-Reduce Conflict

- Problem 1: What if the state looks like this?

$$\boxed{\begin{array}{l} A \to \alpha \bullet c\beta \\ B \to \gamma \bullet \end{array}}$$

- Question: Do we …
  - *shift* the next character c (as suggested by $A \to \alpha \bullet c\beta$) or
  - *reduce* γ to B (as suggested by $B \to \gamma \bullet$)?
- This is known as a *shift-reduce conflict*.

Conflicts

Reduce-Reduce Conflict

- Problem 1: What if the state looks like this?

$$\boxed{\begin{array}{l} A \to \alpha\bullet \\ B \to \beta\bullet \end{array}}$$

- Question: Do we …
  - *reduce* α to A (as suggested by $A \to \alpha\bullet$) or
  - *reduce* β to B (as suggested by $B \to \beta\bullet$)?
- This is known as a *reduce-reduce conflict*.

Causes of Conflicts

- If any item $A \to \alpha\bullet$ occurs in a state in which *it is not alone* then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0).

Conflicts

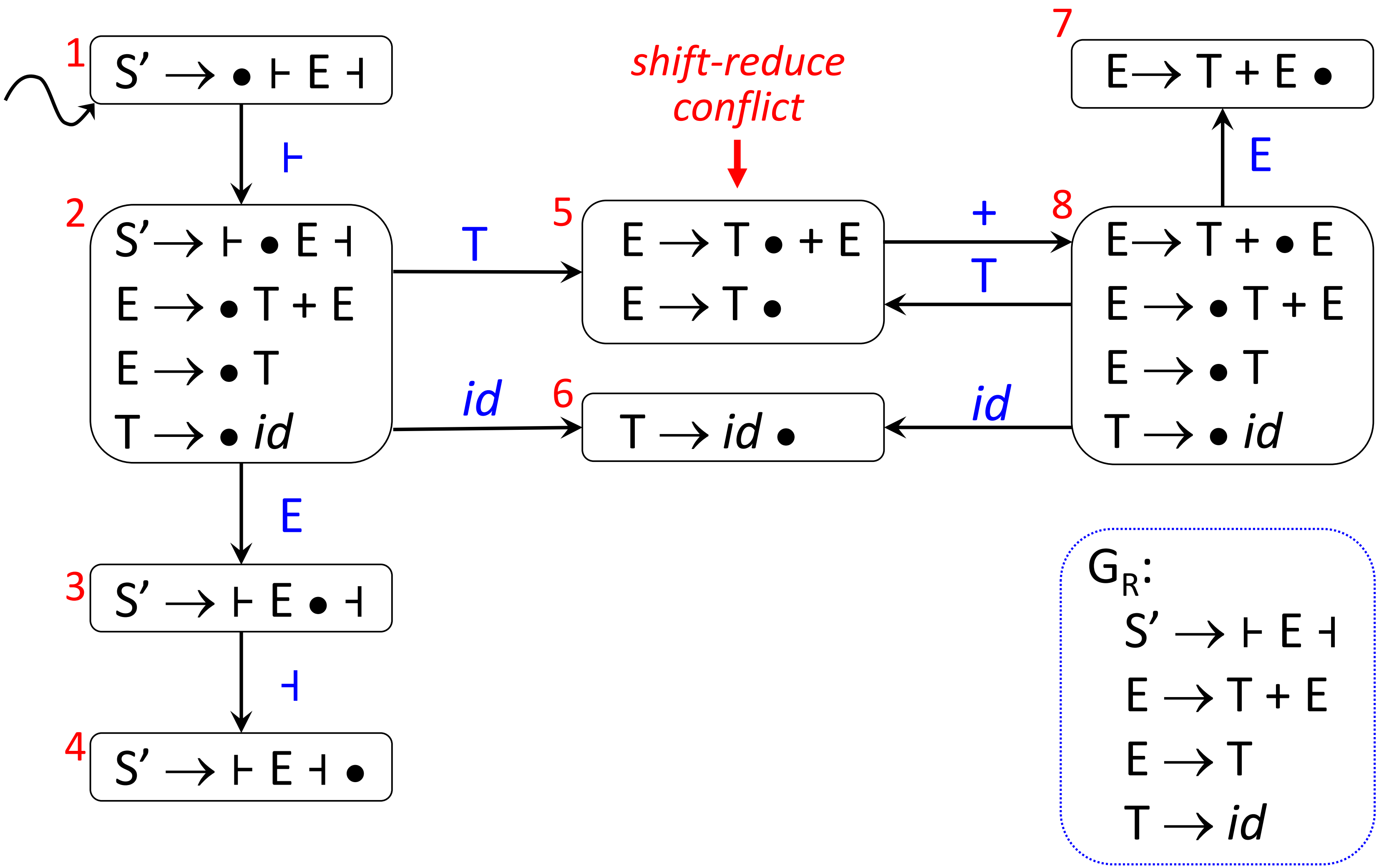
Sample Grammar with Conflict

- Consider right-associative expressions. Modify our previous grammar slightly (i.e. reverse RHS of second rule):

G_R: 1. $S' \rightarrow \vdash E \dashv$
2. $E \rightarrow T + E$ (was $E \rightarrow E + T$)
3. $E \rightarrow T$
4. $T \rightarrow id$

- Now build an automaton based on this modified grammar.

Conflicts: New LR(0) automaton



Conflicts

Sample Conflict

- Input starts with ⊢ *id* ...
- Consider the stack (initially empty)

$$\boxed{} \xrightarrow{\text{shift } \vdash} \boxed{\vdash} \xrightarrow{\text{shift } id} \boxed{\vdash id} \xrightarrow{\text{reduce } id} \boxed{\vdash T}$$

- Should we now reduce T to E (i.e. use rule $E \to T$)?
- Answer: it depends
  - If the input is $\vdash id \dashv$ then YES.
  - If the input is $\vdash id + \ldots \dashv$ then NO. Keep shifting to get T + E, and then reduce using rule $E \to T + E$ instead

G_R:

$S' \to \vdash E \dashv$

$E \to T + E$

$E \to T$

$T \to id$

Resolving Conflicts

Sample Conflict

- Solution: *add a lookahead token* to the automaton to resolve the conflict
- For each $A \rightarrow \alpha$, attach Follow(A), e.g.
  - Follow(E) = { ⊣ }
  - Follow(T) = { +, ⊣ }

G_R:

$S' \rightarrow \vdash E \dashv$

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id$

$$\boxed{\begin{array}{l} E \rightarrow T \bullet \\ E \rightarrow T \bullet + E \end{array}} \quad \text{becomes} \quad \boxed{\begin{array}{l} E \rightarrow T \bullet \quad \dashv \\ E \rightarrow T \bullet + E \end{array}}$$

- Interpretation: a reduce action $\boxed{A \rightarrow \alpha \bullet \quad X}$, where X=Follow(A), *applies only if the next token is X*
- $E \rightarrow T \bullet \quad \dashv$ only applies when the next token is "⊣"
- $E \rightarrow T \bullet + E$ only applies when the next token is "+"

SLR(1) Parser

LR Parsing

- When we add this one character of lookahead, we have an *SLR(1)* (Simple LR with 1 character lookahead) parser
- SLR(1) resolves many, but not all, conflicts.
- Can create increasing more sophisticated automatons
  - e.g. LALR(1) and LR(1) parsers
  - each is more complex
  - each can parse more grammars
  - the parsing algorithm is the same, it is the automaton that changes
  - common parsing tools like Yacc and Bison use LALR(1)

LR Parsing

Reducing the Time Complexity

- Problem: the time complexity is O(n^2)
  - for each of the n input chars, move through the stack and automaton up to n times, e.g. lines 7, 8 of our LR(0) table

	Symbol Stack	States Stack	Input Read	Unread Input	Action
7	⊢ E + *id*	1 2 3 7 6	⊢ *id*+*id*	+*id* ⊣	reduce *id*
8	⊢ E + T	1 2 3 7 8	⊢ *id*+*id*	+*id* ⊣	reduce E + T

- *Idea:* store the automaton state in a States Stack
  - if you pop elements off the top of the stack, the States Stack tells you what state to go to next (rather than have to start at the beginning)
- *Result:* complexity now O(n)

LR Parsing

Outputting a Derivation

- *Idea:* each time a reduction is done, output the rule that was used.
- *Modification:* since LR parsing is bottom up, list the rules in reverse order.
- For our ⊢ abywz ⊣ derivations, it would be rules 1, 2, 6, 3

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$

Derivation

S'	$\Rightarrow \vdash S \dashv$	(1)
	$\Rightarrow \vdash AyB \dashv$	(2)
	$\Rightarrow \vdash Aywz \dashv$	(6)
	$\Rightarrow \vdash abywz \dashv$	(3)

LR Parsing

Outputting a Derivation

- In the table we are expanding the leftmost terminal.
- When we output the rules in reverse, the list now expands on the rightmost terminal first.

Table

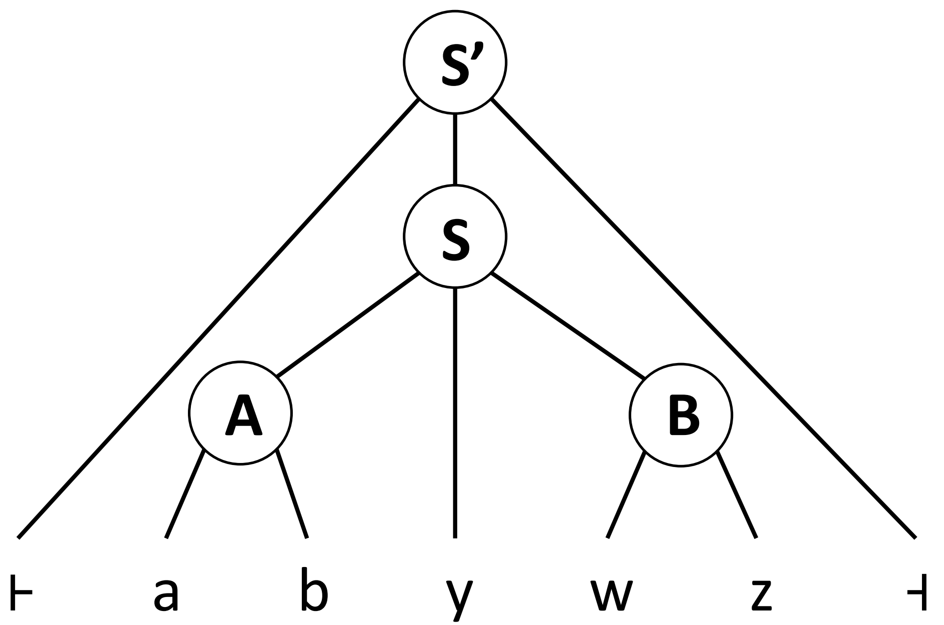
	⊢ abywz ⊣	
⇒	⊢ Aywz ⊣	(3)
⇒	⊢ AyB ⊣	(6)
⇒	⊢ S ⊣	(2)
⇒	S’	(1)

Derivation

S’ ⇒	⊢ S ⊣	(1)
⇒	⊢ AyB ⊣	(2)
⇒	⊢ Aywz ⊣	(6)
⇒	⊢ abywz ⊣	(3)

LR Parsing

The Parse Tree



Derivation

S' ⇒ ⊢ S ⊣ (1)
⇒ ⊢ AyB ⊣ (2)
⇒ ⊢ Aywz ⊣ (6)
⇒ ⊢ abywz ⊣ (3)

Non- LL(1) Grammars

Use LR to Parse our Non-LL(1) Grammar

G: $L = \{a^n b^m \mid n \geq m \geq 0\}$ is not in LL(k) for any k

1: $S' \rightarrow \vdash A \dashv$
2: $A \rightarrow a\,A$
3: $A \rightarrow B$
4: $B \rightarrow aBb$
5: $B \rightarrow \varepsilon$

if A on the stack
  and input = a, shift
  and input = b, reduce 5

if B on the stack
  and input = b, reduce 4
  and input = ⊣, reduce 3

	⊢ aaabb ⊣	shift ⊢
⊢	aaabb ⊣	shift a
⊢ a	aabb ⊣	shift a
⊢ aa	abb ⊣	shift a
⊢ aaa	bb ⊣	reduce 5
⊢ aaaB	bb ⊣	reduce 4
⊢ aaB	b⊣	reduce 4
⊢ aB	⊣	reduce 3
⊢ aA	⊣	reduce 2
⊢ A	⊣	reduce 1
S’		accept