

Bottom up parsing

- General idea
- LR(0)
- SLR
- LR(1)
- LALR
- To best exploit JavaCUP, should understand the theoretical basis (LR parsing);

Top-down vs Bottom-up

- Bottom-up more powerful than top-down;
 - Can process more powerful grammar than LL, will explain later.
- Bottom-up parsers are too hard to write by hand
 - but JavaCUP (and yacc) generates parser from spec;
- Bottom up parser uses right most derivation
 - Top down uses left most derivation;
- Less grammar translation is required, hence the grammar looks more natural;
- Intuition: bottom-up parse postpones decisions about which production rule to apply until it has more data than was available to top-down.
 - Will explain later

Bottom up parsing

- Start with a string of terminals;
- Build up from leaves of parse tree;
- Apply productions backwards;
- When reach start symbol & exhausted input, done;
- Shift-reduce is common bottom-up technique.
 - Notice the blue d should not be reduced into B in step 2.
 $S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$
 - How to get the right reduction steps?

Example:

Grammar: $S \rightarrow aABe$
 $A \rightarrow Abc \mid b$
 $B \rightarrow d$

Reduce abbcde to S:

ab bcde
aA bcde
aAbc de
aAd e
aAd e
aAB e
aABe
S

Sentential form

- Sentential Form

- Any string that can be derived from non-terminals.
- Can consist of terminals and non terminals.
- Example: $E \Rightarrow E+T \Rightarrow E + id \Rightarrow T+id \Rightarrow id + id$
- Sentential forms: $E+id$, $T+id$, ...
- Right sentential form: obtained by right most derivation

- Sentence

- Sentential form with no non-terminals;
- `id+id` is a sentence.

Handles

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abcde$$
$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

- Informally, a handle of a sentential form is a substring that “can” be reduced.
 - Abc is a handle of the right sentential form $aAbcde$, because
 - $A \rightarrow Abc$, and
 - after Abc is replaced by A , the resulting string $aAde$ is still a right sentential form.
 - Is d a handle of $aAbcde$?
 - No. this is because $aAbcBe$ is not a right sentential form.
- Formally, a handle of a right sentential form γ is a production $A \rightarrow \beta$ and a position in γ where the string β may be found and replaced by A .
 - If $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha \beta w$, then $A \rightarrow \beta$ in the position after α is a handle of $\alpha \beta w$.
 - When the production $A \rightarrow \beta$ and the position are clear, we simply say “the substring β is a handle”.

Handles in expression example

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

- Consider the string: $\text{int} * \text{int} + \text{int}$
- The rightmost derivation

$$E \Rightarrow_{\text{rm}} T + E$$

$$\Rightarrow_{\text{rm}} T + T$$

$$\Rightarrow_{\text{rm}} T + \text{int}$$

$$\Rightarrow_{\text{rm}} \text{int} * T + \text{int}$$

$$\Rightarrow_{\text{rm}} \text{int} * \text{int} + \text{int}$$

- For unambiguous grammar, there is exactly one handle for each right-sentential form.
- The question is, how to find the handle?
- Observation: The substring to the right of a handle contains only terminal symbols.

Shift-reduce parsing

- Break the input string into two parts: un-digested part and semi-digested part
 - Left part of input partly processed;
 - Right part completely unprocessed.

```
int foo (double                n) { return (int) n+1 ; }
          ↑                      ↑
Shifted, partly reduced      So far unprocessed
```

- Use stack to keep track of tokens seen so far and the rules already applied backwards (reductions)
- Shift next input token onto stack
- When stack top contains a “good” right-hand-side of a production, reduce by a rule;
- Important fact: Handle is always at the top of the stack.

Shift-reduce main loop

- *Shift*: If can't perform a reduction and there are tokens remaining in the unprocessed input, then transfer a token from the input onto the stack.
- *Reduce*: If we can find a rule $A \rightarrow \alpha$, and the contents of the stack are $\beta\alpha$ for some β (β may be empty), then reduce the stack to βA . The α is called a *handle*. Recognizing handles is key!
- *Accept*: S is at the top of the stack and input now empty, done
- *Error*: other cases.

Example 1

Grammar:
 $S \rightarrow E$
 $E \rightarrow T \mid E + T$
 $T \rightarrow id \mid (E)$

Input string:
 (id)

Parse Stack	Remaining input	Parser Action
	(id)\$	Shift parenthesis onto stack
(id)\$	Shift id onto stack
(id)\$	Reduce: $T \rightarrow id$ (pop RHS of production, push LHS, input unchanged)
(T)\$	Reduce: $E \rightarrow T$
(E)\$	Shift right parenthesis
(E)	\$	Reduce: $T \rightarrow (E)$
T	\$	Reduce: $E \rightarrow T$
E	\$	Reduce: $S \rightarrow E$
S	\$	Done: Accept

$S \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} (E) \Rightarrow_{rm} (T) \Rightarrow_{rm} (id)$

Shift-Reduce Example 2

$S \rightarrow E$
 $E \rightarrow T \mid E + T$
 $T \rightarrow id \mid (E)$

Input: (id + id)

Parse Stack	Remaining Input	Action	
	(id + id) \$	Shift (
(id + id) \$	Shift id	
(id	+ id) \$	Reduce $T \rightarrow id$	(id • +id)
(T	+ id) \$	Reduce $E \rightarrow T$	(T • +id)
(E	+ id) \$	Shift +	
(E+	id) \$	Shift id	(E +id •)
(E+id) \$	Reduce $T \rightarrow id$	(E+T •)
(E+T) \$	Reduce $E \rightarrow E+T$; (Ignore: $E \rightarrow T$)	
(E) \$	Shift)	(E) •
(E)	\$	Reduce $T \rightarrow (E)$	T •
T	\$	Reduce $E \rightarrow T$	E •
E	\$	Reduce $S \rightarrow E$	S
S	\$	Accept	

Note that it is the reverse of the following rightmost derivation:

$$\begin{aligned}
 S &\Rightarrow_{rm} E \bullet \Rightarrow_{rm} T \bullet \Rightarrow_{rm} (E) \bullet \Rightarrow_{rm} (E+T \bullet) \Rightarrow_{rm} (E +id \bullet) \\
 &\Rightarrow_{rm} (T \bullet +id) \Rightarrow_{rm} (id \bullet +id)
 \end{aligned}$$

Conflicts during shift reduce parsing

- Reduce/reduce conflict

stack	input
... (E+T)

Which rule we should use, $E \rightarrow E+T$ or $E \rightarrow T$?

- Shift/reduce conflict

– ifStat \rightarrow if (E) S | if (E) S else S

stack	Input
... if (E) S	else ...

- Both reduce and shift are applicable.
- What we should do next, reduce or shift?

LR(K) parsing

- **Left-to-right, Rightmost derivation with k-token lookahead.**
 - L - Left-to-right scanning of the input
 - R - Constructing rightmost derivation in reverse
 - k - number of input symbols to select a parser action
- **Most general parsing technique for deterministic grammars.**
 - Efficient, Table-based parsing
 - Parses by shift-reduce
 - Can massage grammars less than LL(1)
 - Can handle almost all programming language structures
 - $LL \subset LR \subset CFG$
- **In general, not practical: tables too large (10^6 states for C++, Ada).**
- **Common subsets: SLR, LALR (1).**

LR Parsing continued

- Data structures:
 - Stack of states $\{s\}$
 - Action table $\text{Action}[s,a]$; $a \in T$
 - Goto table $\text{Goto}[s,X]$; $X \in N$
- In LR parsing, push whole *states* on stack
 - Stack of states keeps track of what we've seen so far (*left context*): what we've shifted & reduced & by what rules.
- Use Action tables to decide shift vs reduce
- Use Goto table to move to new state

Main loop of LR parser

- Initial state S_0 starts on top of stack;
- Given state St state on top of stack and the next input token a :
- If ($\text{Action}[St, a] == \text{shift } S_i$)
 - Push new state S_i onto stack
 - Call `yylex` to get next token
- If ($\text{Action}[St, a] == \text{reduce by } Y \rightarrow X_1 \dots X_n$)
 - Pop off n states to find S_u on top of stack
 - Push new state $S_v = \text{Goto}[S_u, Y]$ onto stack
- If ($\text{Action}[St, a] == \text{accept}$), done!
- If ($\text{Action}[St, a] == \text{error}$), can't continue to successful parse.

Example LR parse table

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow (E)$
- (4) $T \rightarrow id$

State on TOS	<u>Action</u>					<u>Goto</u>	
	id	+	()	\$	E	T
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

If ($\text{Action}[\text{St}, a] == \text{shift}$), Push new state $\text{Action}[\text{St}, a]$ onto stack, Call yylex to get next token

If ($\text{Action}[\text{St}, a] == \text{reduce by } Y \rightarrow X_1 \dots X_n$), Pop off n states to find S_u on top of stack, Push new state $S_v = \text{Goto}[S_u, Y]$ onto stack

We explain how to construct this table later.

State on TOS	Action					Goto	
	id	+	()	\$	E	T
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow (E)$
- (4) $T \rightarrow id$

State stack	Remaining Input	Parser action
S0	id + (id)\$	Shift S4 onto state stack, move ahead in input
S0S4	+ (id)\$	Reduce 4) $T \rightarrow id$, pop state stack, goto S2, input unchanged
S0S2	+ (id)\$	Reduce 2) $E \rightarrow T$, goto S1
S0S1	+ (id)\$	Shift S5
S0S1S5	(id)\$	Shift S3
S0S1S5S3	id)\$	Shift S4 (saw another id)
S0S1S5S3S4)\$	Reduce 4) $T \rightarrow id$, goto S2
S0S1S5S3S2)\$	Reduce 2) $E \rightarrow T$, goto S6
S0S1S5S3S6)\$	Shift S7
S0S1S5S3S6S7	\$	Reduce 3) $T \rightarrow (E)$, goto S8
S0S1S5S8	\$	Reduce 1) $E \rightarrow E + T$, goto S1 *
S0S1	\$	Accept

Types of LR parsers

- LR (k)
- SLR (k) -- Simple LR
- LALR (k) – LookAhead LR
- k = # symbols lookahead
 - 0 or 1 in this class
 - Dragon book has general cases
- Start with simplest: LR(0) parser

LR (0) parser

- Advantages:
 - Simplest to understand,
 - Smallest tables
- Disadvantages
 - No lookahead, so too simple-minded for real parsers
- Good case to see how to build tables, though.
- We'll use LR(0) constructions in other LR(k) parsers
- Key to LR parsing is recognizing handles
 - Handle: sequence of symbols encoded in top stack states representing a right-hand-side of a rule we want to reduce by.

LR Tables

- Given grammar G , identify possible states for parser.
 - States encapsulate what we've seen and shifted and what are reduced so far
- Steps to construct LR table:
 - Construct states using LR(0) *configurations* (or *items*);
 - Figure out transitions between states

Configuration

- A configuration (or item) is a rule of G with a dot in the right-hand side.
- If rule $A \rightarrow XYZ$ in grammar, then the configs are

$$\begin{array}{ll} A \rightarrow \bullet XYZ & A \rightarrow XY \bullet Z \\ A \rightarrow X \bullet YZ & A \rightarrow XYZ \bullet \end{array}$$

- Dot represents what parser has gotten in stack in recognizing the production.

$A \rightarrow XYZ \bullet$ means XYZ on stack. Reduce!

$A \rightarrow X \bullet YZ$ means X has been shifted. To continue parse, we must see a token that could begin a string derivable from Y.

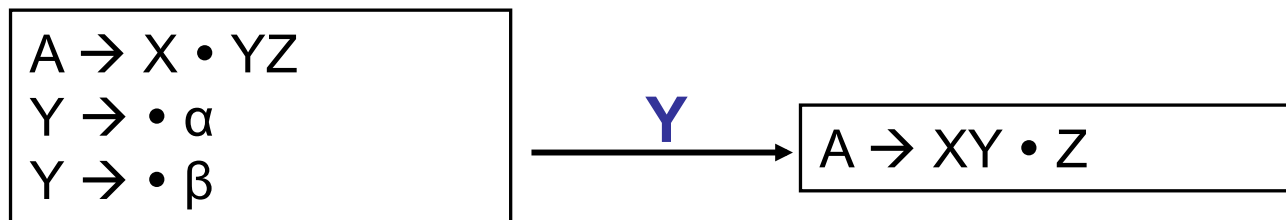
- **Notational convention:**
 - X, Y, Z: symbol, either terminal or non-terminal
 - a, b, c: terminal
 - α, β, γ : sequence of terminals or non-terminals

Set of configurations

- $A \rightarrow X \cdot YZ$ means X has been shifted. To continue parse, we must see a token that could begin a string derivable from Y .
- That is, we need to see a token in $\text{First}(Y)$ (or in $\text{Follow}(Y)$ if $Y \rightarrow \varepsilon$)
- Formally, need to see a token t such that
 $Y \Rightarrow^* t \beta$ for some β
- Suppose $Y \rightarrow \alpha \mid \beta$ also in G . Then these configs correspond to the same parse state:
 $A \rightarrow X \cdot YZ$
 $Y \rightarrow \cdot \alpha$
 $Y \rightarrow \cdot \beta$
- Since the above configurations represent the same state, we can:
 - Put them into a set together.
 - Add all other equivalent configurations to achieve closure. (algorithm later)
 - This set represents one parser state: the state the parser can be in while parsing a string.

Transitions between states

- Parser goes from one state to another based on symbols processed



- Model parse as a finite automaton!
- When a state (configuration set) has a dot at a end of an item, that is FA accept state
- Build LR(0) parser based on this FA

Constructing item sets & closure

- **Starting Configuration:**
 - Augment Grammar with symbol S'
 - Add production $S' \rightarrow S$ to grammar
 - Initial item set I_0 gets
 $S' \rightarrow \bullet S$
- **Perform Closure on $S' \rightarrow \bullet S$**
(That completes parser start state.)
- **Compute Successor function to make next state (next item set)**

Computing closure

- Closure(I)

1. Initially every item in I is added to closure(I)
2. If $A \rightarrow \alpha \bullet B \beta$ is in closure(I)
for all productions $B \rightarrow \gamma$, add $B \rightarrow \bullet \gamma$
3. Repeat step 2 until set gets no more additions.

- Example

Given the configuration set: $\{ E \rightarrow \bullet E + T \}$

What is the closure of $\{ E \rightarrow \bullet E + T \}$:

$E \rightarrow \bullet E + T$ by rule 1

$E \rightarrow \bullet T$ by rule 2

$T \rightarrow \bullet (E)$ by rule 2 and 3

$T \rightarrow \bullet id$ by rule 2 and 3

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow (E)$
(4) $T \rightarrow id$

Building state transitions

- LR Tables need to know what state to goto after shift or reduce.
- Given Set C & symbol X , we define the set $C' = \text{Successor}(C, X)$ as:
 - For each config in C of the form $Y \rightarrow \alpha \bullet X \beta$,
 - 1. Add $Y \rightarrow \alpha X \bullet \beta$ to C'
 - 2. Do closure on C'
- Informally, move by symbol X from one item set to another;
 - move \bullet to the right of X in all items where dot is before X ;
 - remove all other items;
 - compute closure.



Successor example

- Given

$I = \{ E \rightarrow \bullet E + T,$
 $E \rightarrow \bullet T,$
 $T \rightarrow \bullet (E),$
 $T \rightarrow \bullet id$
 $\}$

- What is $\text{successor}(I, "(")$?
- move the \bullet after " $($ ": $T \rightarrow (\bullet E)$
- compute the closure:

$T \rightarrow (\bullet E)$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet (E)$
 $T \rightarrow \bullet id$

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow (E)$
(4) $T \rightarrow id$

Construct the LR(0) table

- Construct $F = \{I_0, I_1, I_2, \dots, I_n\}$
- State i is determined by li . The parsing actions for state i are:
 - if $A \rightarrow \alpha \bullet$ is in li ,
 - then set $\text{Action}[i, a]$ to reduce $A \rightarrow \alpha$ for all inputs (if A is not S')
 - If $S' \rightarrow S \bullet$ is in li ,
 - then set $\text{action}[i, \$]$ to accept.
 - if $A \rightarrow \alpha \bullet a \beta$ is in li and $\text{successor}(li, a) = lj$,
 - then set $\text{action}[i, j]$ to shift j . (a is a terminal)
- The goto transitions for state i are constructed for all non-terminals A using the rule:
 - if $\text{successor}(li, A) = lj$, then $\text{goto}[i, A] = j$.
- All entries not defined by above rules are errors.
- The initial state I_0 is the one constructed from $S' \rightarrow \bullet S$.

Steps of constructing LR(0) table

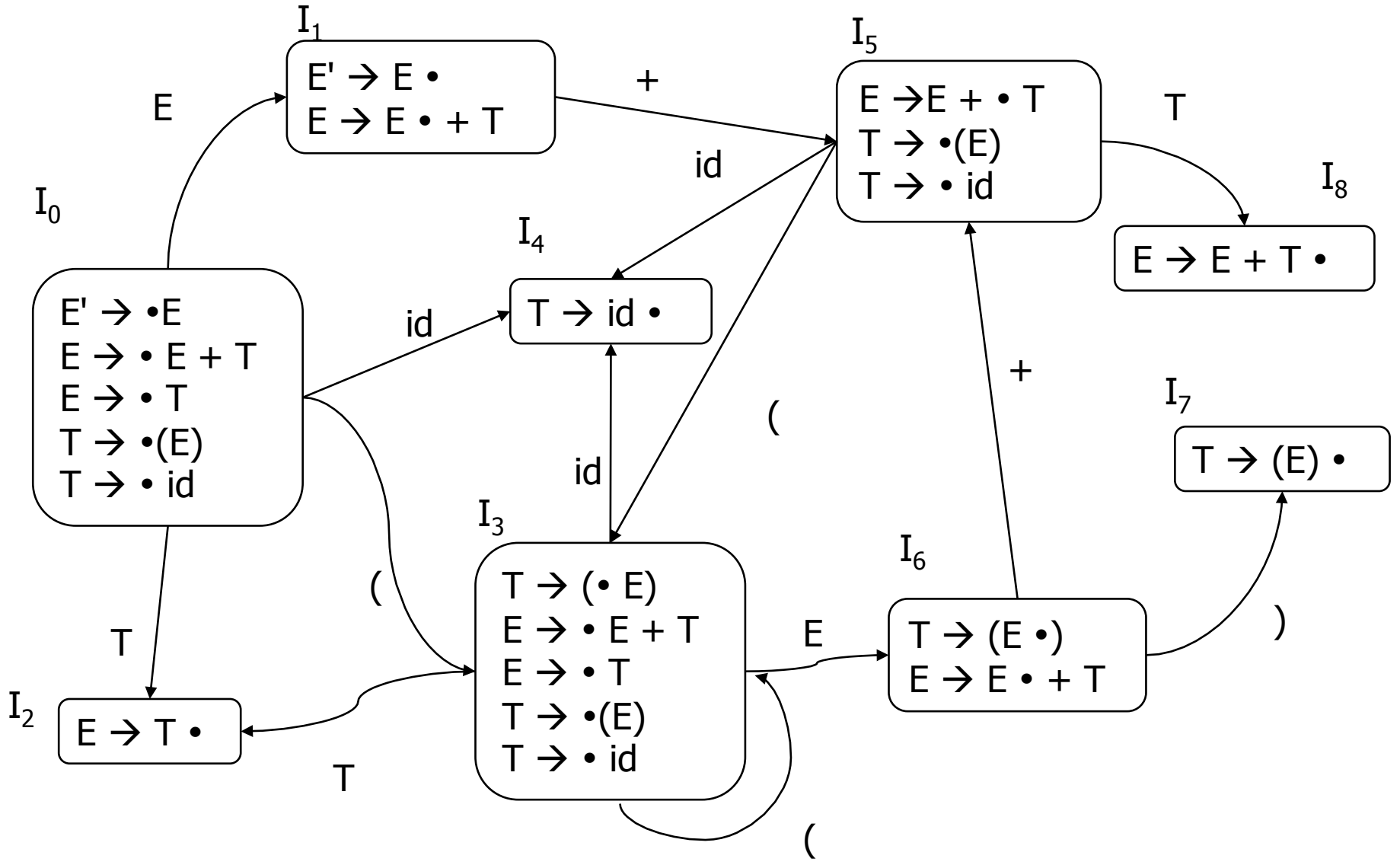
1. Augment the grammar;
2. Draw the transition diagram;
 1. Compute the configuration set (item set/state);
 2. Compute the successor;
3. Fill in the Action table and Goto table.

(0)	$E' \rightarrow E$
(1)	$E \rightarrow E + T$
(2)	$E \rightarrow T$
(3)	$T \rightarrow (E)$
(4)	$T \rightarrow id$

Item sets example

<u>Configuration set</u>	<u>Successor</u>
I0: $E' \rightarrow \bullet E$	I1
$E \rightarrow \bullet E+T$	I1
$E \rightarrow \bullet T$	I2
$T \rightarrow \bullet (E)$	I3
$T \rightarrow \bullet id$	I4
I1: $E' \rightarrow E \bullet$	Accept (dot at end of E' rule)
$E \rightarrow E \bullet +T$	I5
I2: $E \rightarrow T \bullet$	Reduce 2 (dot at end)
I3: $T \rightarrow (\bullet E)$	I6
$E \rightarrow \bullet E+T$	I6
$E \rightarrow \bullet T$	I2
$T \rightarrow \bullet (E)$	I3
$T \rightarrow \bullet id$	I4
I4: $T \rightarrow id \bullet$	Reduce 4 (dot at end)
I5: $E \rightarrow E+ \bullet T$	I8
$T \rightarrow \bullet (E)$	I3
$T \rightarrow \bullet id$	I4
I6: $T \rightarrow (E \bullet)$	I7
$E \rightarrow E \bullet +T$	I5
I7: $T \rightarrow (E) \bullet$	Reduce 3 (dot at end)
I8: $E \rightarrow E+T \bullet$	Reduce 1 (dot at end)

Transition diagram



The parsing table

State on TOS	<u>Action</u>					<u>Goto</u>	
	id	+	()	\$	E	T
0	S4		S3			1	2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			6	2
4	R4	R4	R4	R4	R4		
5	S4		S3				8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

Parsing an erroneous input

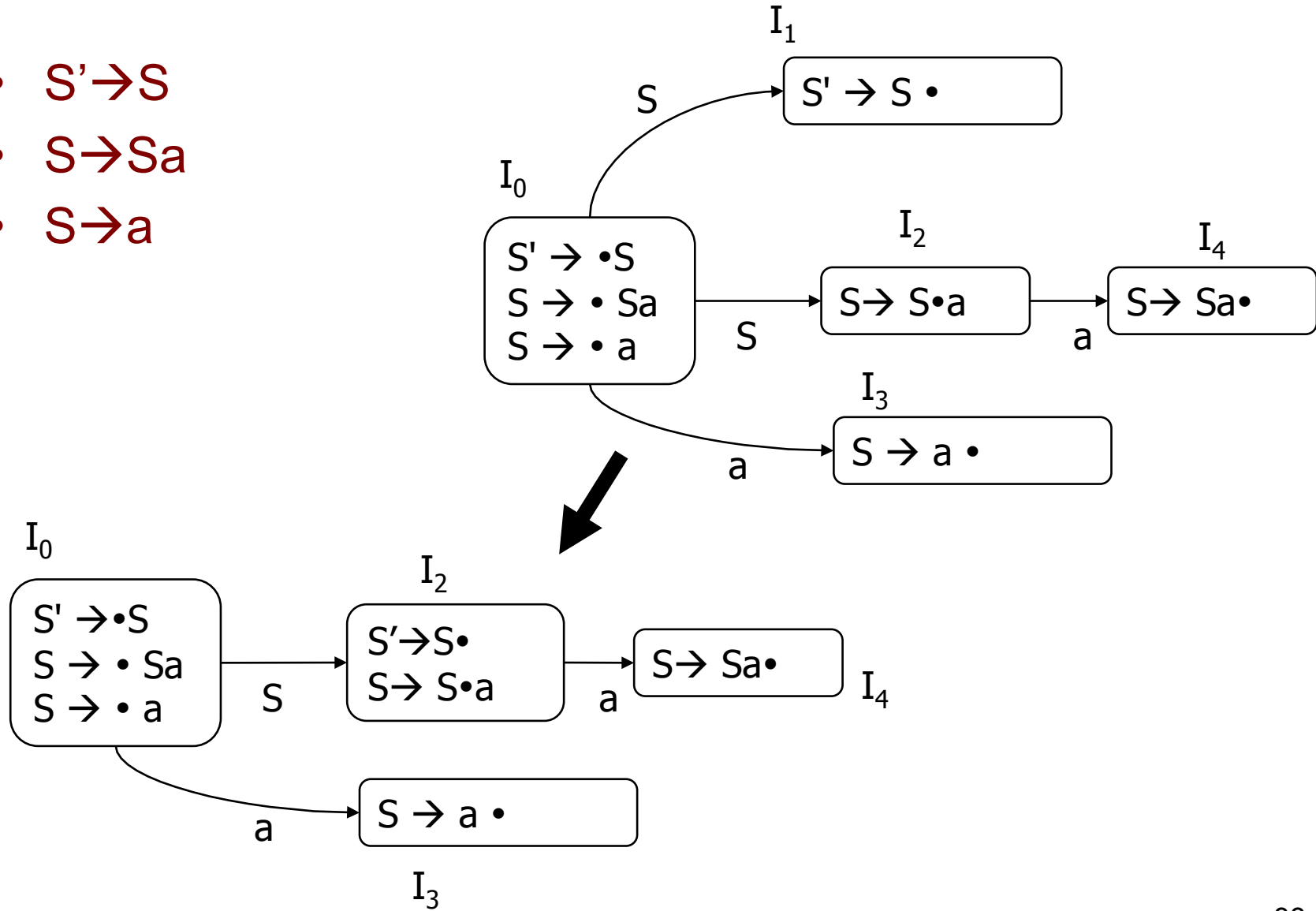
- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow (E)$
- (4) $T \rightarrow id$

State stack	Input	Parser action
S0	id + +\$	Shift S4
S0 S4	+ +\$	Reduce 4) $T \rightarrow id$, pop S4, Goto S2
S0 S2	+ +\$	Reduce 2) $E \rightarrow T$, pop S2, Goto S1
S0 S1	+ +\$	Push S5
S0 S1 S5	+\$	No Action [S5, +] Error!

State on TOS	Action					Goto	
	id	+	()	\$	E	T
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

Subset construction and closure

- $S' \rightarrow S$
- $S \rightarrow Sa$
- $S \rightarrow a$

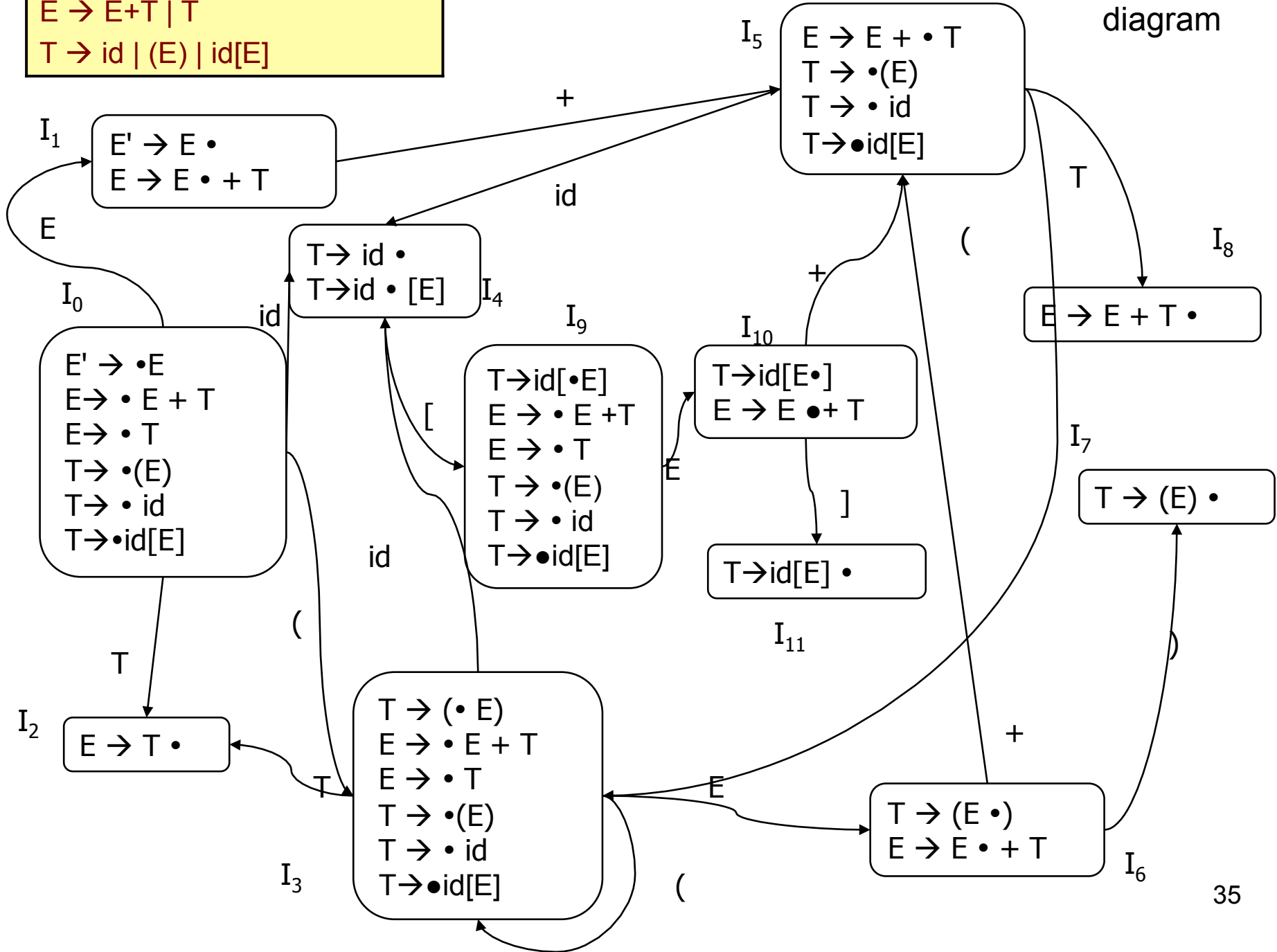


LR(0) grammar

- A grammar is LR(0) if the following two conditions hold:
 1. For any configuration set containing the item $A \rightarrow \alpha \bullet a \beta$, there is no complete item $B \rightarrow \gamma \bullet$ in that set.
 - No shift/reduce conflict in any state
 - in table, for each state, either shift or reduce
 2. There is at most one complete item $A \rightarrow \alpha \bullet$ in each configuration set.
 - No reduce/reduce conflict
 - in table, for each state, use same reduction rule for every input symbol.
- Very few grammars meet the requirements to be LR(0).

$S' \rightarrow E$
 $E \rightarrow E+T \mid T$
 $T \rightarrow id \mid (E) \mid id[E]$

Incomplete diagram



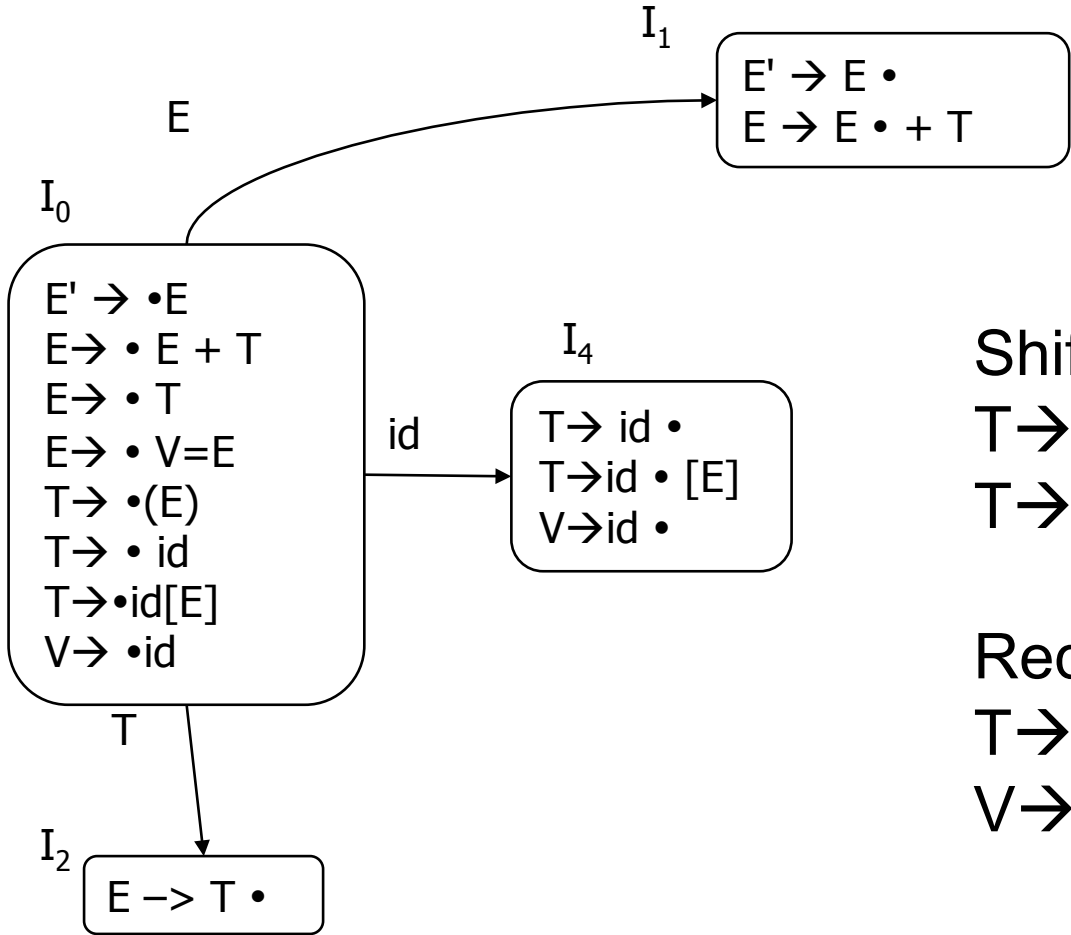
SLR Parse table (incomplete)

State on TOS	<u>Action</u>							<u>Goto</u>	
	id	+	()	\$	[]	E	T
0	S4		S3					1	2
1		S5			accept				
2		R2		R2	R2		R2		
3	S4		S3					6	2
4		R3		R3	R3	S9	R3		
5	S4		S3						8
6		S5		S7					
7		R5		R5	R5		R5		
8		R1		R1	R1		R1		
9	S4		S3					10	2
10		S5					S11		
11		R5		R5	R5		R5		

- (0) $S' \rightarrow E$
- (1) $E \rightarrow E+T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow id$
- (4) $T \rightarrow (E)$
- (5) $T \rightarrow id[E]$

<u>State stack</u>	<u>Input</u>	<u>Parser action</u>
S0	id [id] + id \$	Shift S4
S0 S4 id	[id] + id \$	S9
S0 S4id S9[id] + id \$	shift S4
S0 S4id S9[S4id] +id \$	Reduce T→id
S0 S4id S9[S2T] + id \$	Reduce E→T
S0 S4id S9[S10E] + id \$	S11
S0 S4id S9[S10E S11]	+ id \$	Reduce T→id[E]
S0 S2T	+ id \$	Reduce E→T
S0 S1E	+ id \$	S5
S0 S1E S5+	id \$	S4
S0 S1E S5+ S4id	\$	Reduce T→id
S0 S1E S5+ S8T	\$	Reduce E→E+T
S0 S1E	\$	Accept

$S' \rightarrow E$
 $E \rightarrow E+T \mid T \mid V=E$
 $T \rightarrow id \mid (E) \mid id[E]$
 $V \rightarrow id$



Shift/reduce conflict:

$T \rightarrow id \cdot$

$T \rightarrow id \cdot [E]$

Reduce/reduce conflict:

$T \rightarrow id \cdot$

$V \rightarrow id \cdot$

SLR Parse table (incomplete)

State on TOS	<u>Action</u>							<u>Goto</u>	
	id	+	()	\$	[]	E	T
0	S4		S3					1	2
1		S5			accept				
2		R2		R2	R2		R2		
3	S4		S3					6	2
4		R4		R4	R4	S9	R4		
5	S4		S3						8
6		S5		S7					
7		R5		R5	R5		R5		
8		R1		R1	R1		R1		
9									
10									
11									

- (0) $S' \rightarrow E$
- (1) $E \rightarrow E+T$
- (2) $E \rightarrow T$
- (3) $E \rightarrow V=E$
- (4) $T \rightarrow id$
- (5) $T \rightarrow (E)$
- (6) $T \rightarrow id[E]$
- (7) $V \rightarrow id$

LR(0) key points

- Construct LR table
 - Start with augmented grammar ($S' \rightarrow S$)
 - Generate *items* from productions.
 - Insert the Dot into all positions
 - Generate *item sets* (or configuration sets) from items; they are our parser states.
 - Generate state *transitions* from function successor (state, symbol).
 - Build *Action* and *Goto* tables from states and transitions.
- Tables implement shift-reduce parser.
- View [states and transitions] as a finite automaton.
- An Item represents how far a parser is in recognizing part of one rule's RHS.
- An Item set combines various paths the parser might have taken so far, to diverge as more input is parsed.
- LR(0) grammars are easiest LR to understand, but too simple to use in real life parsing.

Simple LR(1) parsing: SLR

- LR(0)
 - One LR(0) state mustn't have both shift and reduce items, or two reduce items.
 - So any complete item (dot at end) must be in its own state; parser will always reduce when in this state.
- SLR
 - Peek ahead at input to see if reduction is appropriate.
 - Before reducing by rule $A \rightarrow XYZ$, see if the next token is in Follow (A). Reduce *only* in that case. Otherwise, shift.

Construction for SLR tables

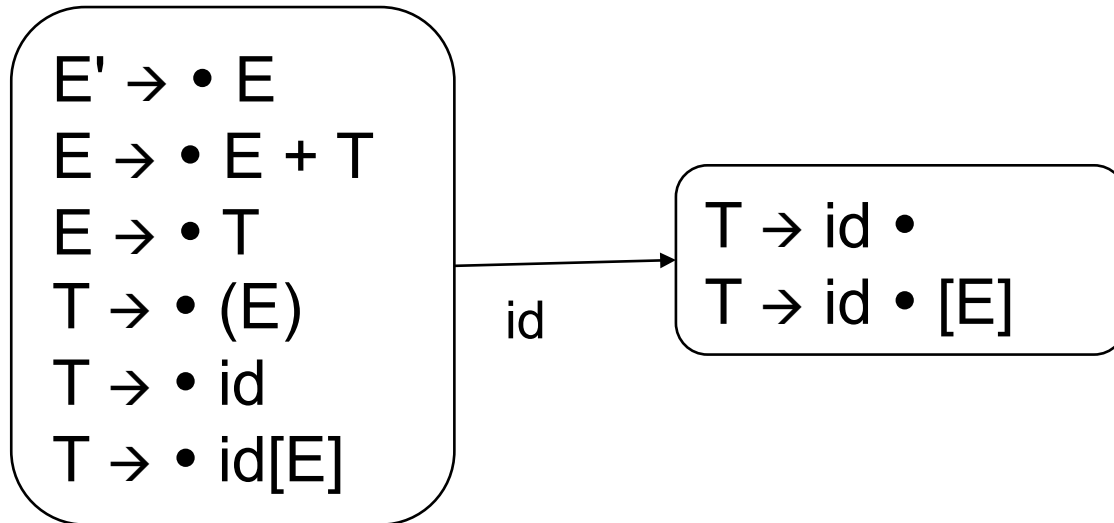
1. Construct $F = \{I_0, I_1, \dots, I_n\}$, the LR(0) item sets.
2. State i is I_i . The parsing actions for the state are:
 - a) If $A \rightarrow \alpha \cdot$ is in I_i
then set $\text{Action}[i, a]$ to reduce $A \rightarrow \alpha$ for all a in $\text{Follow}(A)$ (A is not S').
 - b) If $S' \rightarrow S \cdot$ is in I_i
then set $\text{Action}[i, \$]$ to accept.
 - c) If $A \rightarrow \alpha \cdot a \beta$ is in I_i and $\text{successor}(I_i, a) = I_j$,
then set $\text{Action}[i, a]$ to shift j (a must be a terminal).
3. The goto transitions for state i are constructed for all non-terminals A using the rule:
If $\text{successor}(I_i, A) = I_j$, then $\text{Goto}[i, A] = j$.
4. All entries not defined by rules 2 and 3 are errors.
5. The initial state is closure of set with item $S' \rightarrow \cdot S$.

Properties of SLR

- Pickier rule about setting Action table is the only difference from LR(0) tables;
- If G is SLR it is unambiguous, but not vice versa;
- State can have both shift and reduce items, if Follow sets are disjoint.

SLR Example

Item sets I_0 and successor (I_0, id):



$E' \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow (E) \mid id$
 $\mid id[E]$

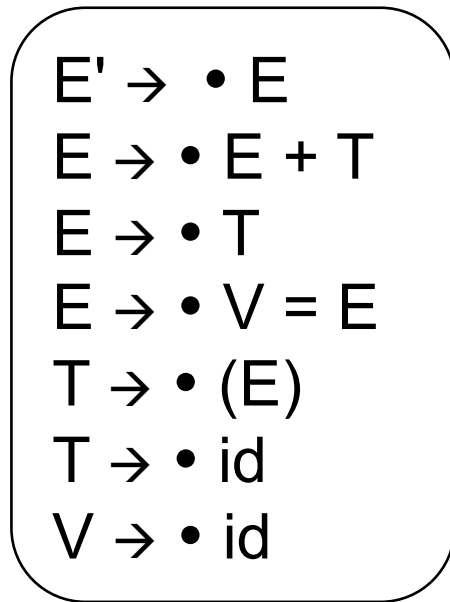
LR(0) parser sees both shift and reduce, but SLR parser consults Follow set:

$Follow(T) = \{ +,),], \$ \}$ so

$T \rightarrow id \bullet$ means reduce on `+` or `)` or `]` or `$`

$T \rightarrow id \bullet [E]$ means shift otherwise (e.g. on `[`)

SLR Example 2



$E' \rightarrow E$
 $E \rightarrow E + T \mid T \mid V = E$
 $T \rightarrow (E) \mid id$
 $V \rightarrow id$

Two complete LR(0) items, so reduce-reduce conflict in LR(0) grammar, but:

$\text{Follow}(T) = \{ +,), \$ \}$

$\text{Follow}(V) = \{ = \}$

Disjoint, so no conflict. Separate Action entries in table.

SLR grammar

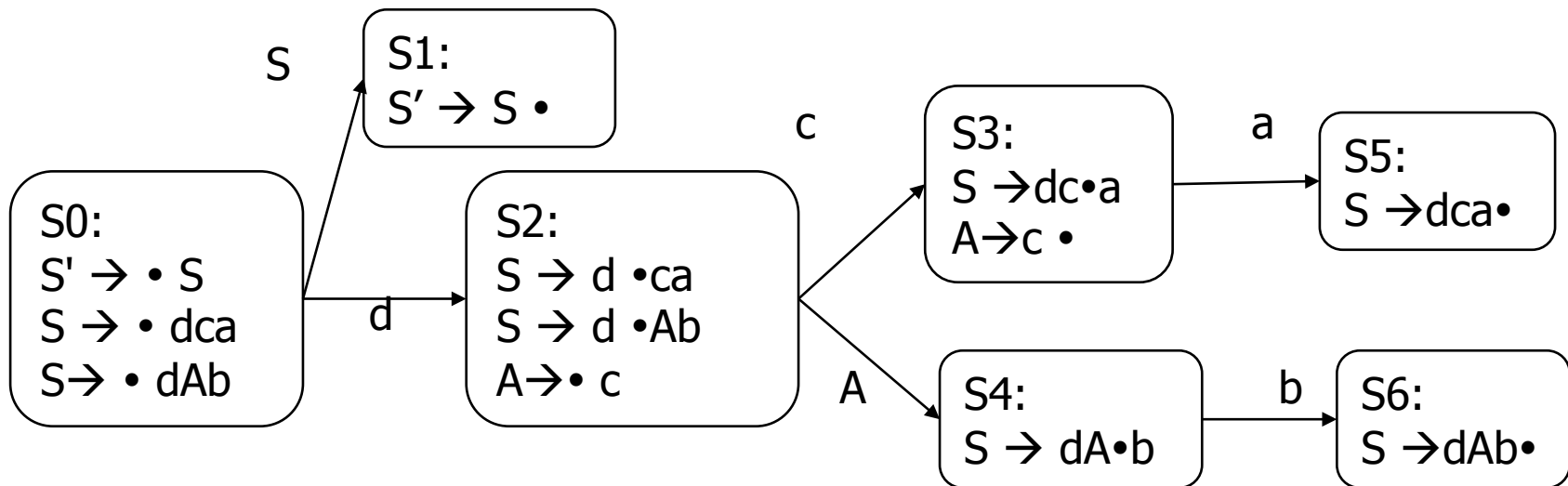
- A grammar is SLR if the following two conditions hold:
 - If items $A \rightarrow \alpha \bullet a \beta$ and $B \rightarrow \gamma \bullet$ are in a state, then terminal $a \notin \text{Follow}(B)$.
 - no shift-reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both.
 - For any two complete items $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ in a state, the Follow sets must be disjoint. ($\text{Follow}(A) \cap \text{Follow}(B)$ is empty.)
 - no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead.
- Compare with LR(0) grammar:
 1. For any configuration set containing the item $A \rightarrow \alpha \bullet a \beta$, there is no complete item $B \rightarrow \gamma \bullet$ in that set.
 2. There is at most one complete item $A \rightarrow \alpha \bullet$ in each configuration set.
- Note that $\text{LR}(0) \subset \text{SLR}$

SLR

1. $S' \rightarrow S$
2. $S \rightarrow dca$
3. $S \rightarrow dAb$
4. $A \rightarrow c$

In S3 there is reduce/shift conflict: It can be R4 or shift. By looking at the Follow set of A, the conflict is removed.

	Action					Goto	
	a	b	c	d	\$	S	A
S0				S2		1	
S1					A		
S2			S3				4
S3	S5	R4					
S4		S6					
S5					R2		
S6					R3		



Parse trace

<u>State stack</u>	<u>Input</u>	<u>Parser action</u>
S0	dca\$	Shift S2
S0 S2d	ca\$	Shift S3
S0 S2d S3c	a\$	shift S5
S0 S2d S3c S5a	\$	Reduce 2
S0 S1S	\$	Accept

Non-SLR example

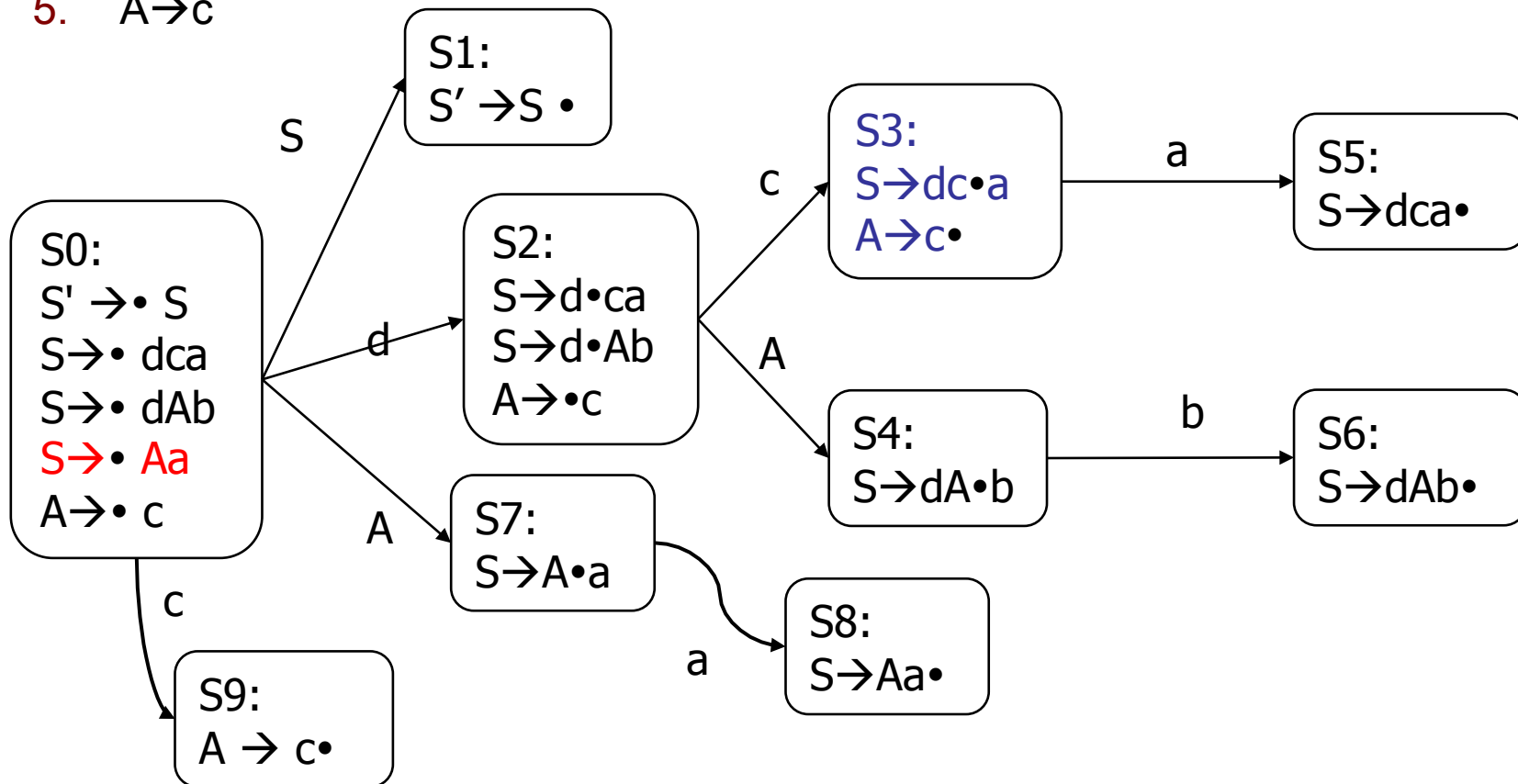
1. $S' \rightarrow S$
2. $S \rightarrow dca$
3. $S \rightarrow dAb$
4. $S \rightarrow Aa$
5. $A \rightarrow c$

S3 has shift/reduce conflict.

By looking at Follow(A),

both a and b are in the follow set.

So under column a we still don't know whether to reduce or shift.



The conflict SLR parsing table

	Action					Goto	
	a	b	c	d	\$	S	A
S0			S9	S2		1	7
S1					A		
S2			S3				4
S3	S5/R5	R5					
S4		S6					
S5					R2		
S6					R3		
S7	S8						
S8					R4		
S9	R5	R5					

Follow(A) = {a, b}

LR(1) parsing

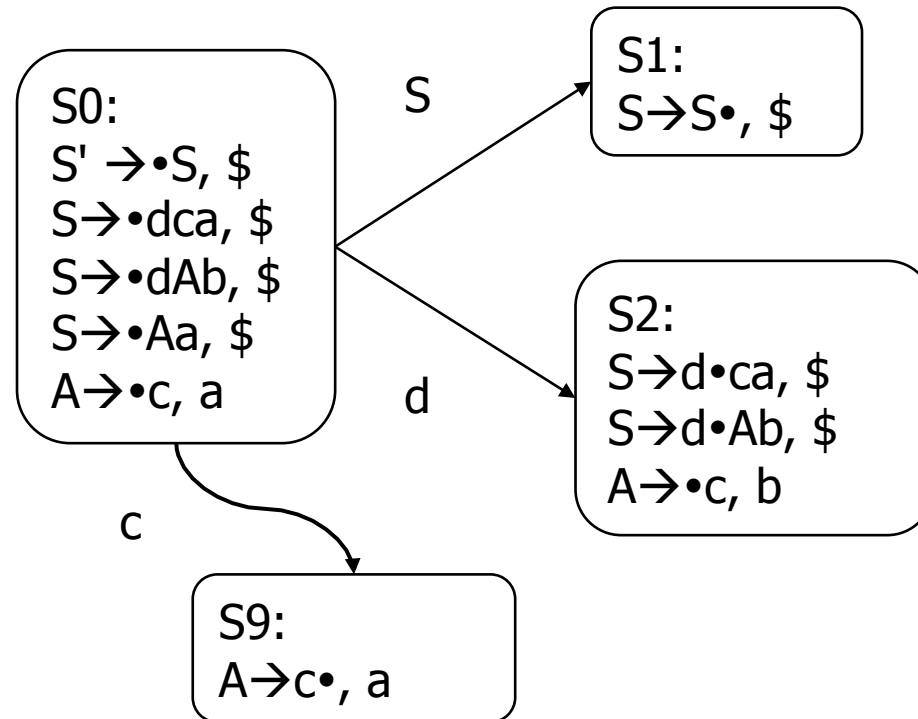
- Make items carry more information.
- LR(1) item:
 $A \rightarrow X_1 \dots X_i \bullet X_{i+1} \dots X_j$, tok
- Terminal **tok** is the *lookahead*.
- Meaning:
 - have states for $X_1 \dots X_i$ on stack already
 - expect to put states for $X_{i+1} \dots X_j$ onto stack and then reduce, but
 - *only* if token following X_j is **tok**
 - tok can be \$
- Split $\text{Follow}(A)$ into separate cases
- Can cluster items notationally:
 $[A \rightarrow \alpha \bullet, a/b/c]$ means the three items: $[A \rightarrow \alpha \bullet, a]$ $[A \rightarrow \alpha \bullet, b]$ $[A \rightarrow \alpha \bullet, c]$
“Reduce α to A if next token is a or b or c ”
- $\{ a, b, c \} \subseteq \text{Follow}(A)$

LR(1) item sets

- More items and more item sets than SLR
- Closure: For each item $[A \rightarrow \alpha \bullet B \beta, a]$ in I ,
 - for each production $B \rightarrow \gamma$ in G ,
 - and for each terminal b in $\text{First}(\beta a)$, add $[B \rightarrow \bullet \gamma, b]$ to I
(Add only items with the correct lookahead)
- Once we have a closed item set, use LR(1) successor function to compute transitions and next items.
- Example:
 - Initial item: $[S' \rightarrow \bullet S, \$]$
 - What is the closure?
 - $S' \rightarrow S$
 - $S \rightarrow dca \mid dAb \mid Aa$
 - $A \rightarrow c$
 - $[S \rightarrow \bullet dca, \$]$
 - $[S \rightarrow \bullet dAb, \$]$
 - $[S \rightarrow \bullet Aa, \$]$
 - $[A \rightarrow \bullet c, a]$

LR(1) successor function

- Given I an item set with $[A \rightarrow \alpha \bullet X \beta, a]$,
 - Add $[A \rightarrow \alpha X \bullet \beta, a]$ to item set J.
 - $\text{successor}(I, X)$ is the closure of set J.
 - Similar to successor function to LR(0), but we propagate the lookahead token for each item.
- Example



LR(1) tables

Action table entries:

- If $[A \rightarrow \alpha \bullet, a] \in I_i$, then set $\text{Action}[i, a]$ to reduce by rule $A \rightarrow \alpha$ (A is not S').
- If $[S' \rightarrow S \bullet, \$] \in I_i$ then set $\text{Action}[i, \$]$ to accept.
- If $[A \rightarrow \alpha \bullet a \beta, b]$ is in I_i and $\text{succ}(I_i, a) = I_j$, then set $\text{Action}[i, a]$ to shift j . Here a is a terminal.

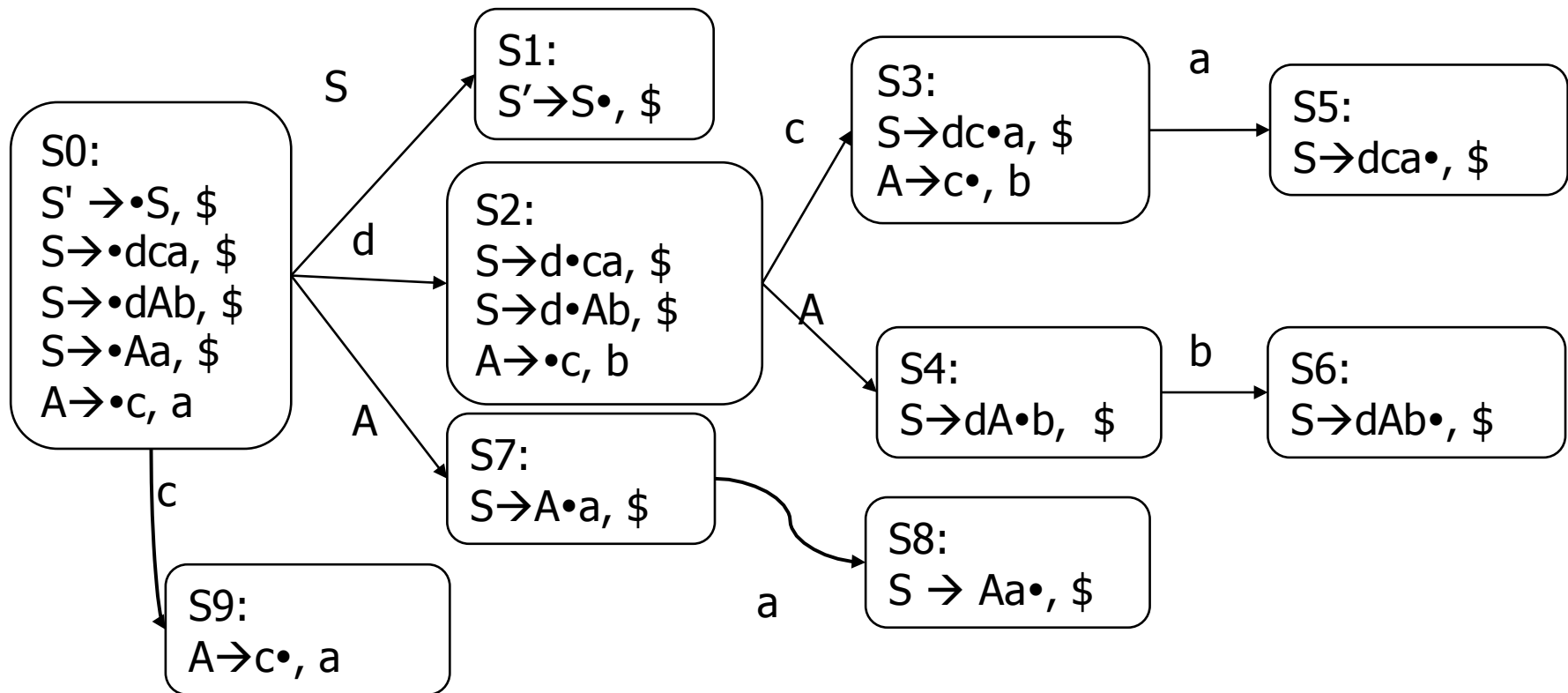
Goto entries:

For each state I & each non-terminal A :

- If $\text{succ}(I_i, A) = I_j$, then $\text{Goto}[i, A] = j$.

LR(1) diagram

1. $S' \rightarrow S$
2. $S \rightarrow dca$
3. $S \rightarrow dAb$
4. $S \rightarrow Aa$
5. $A \rightarrow c$



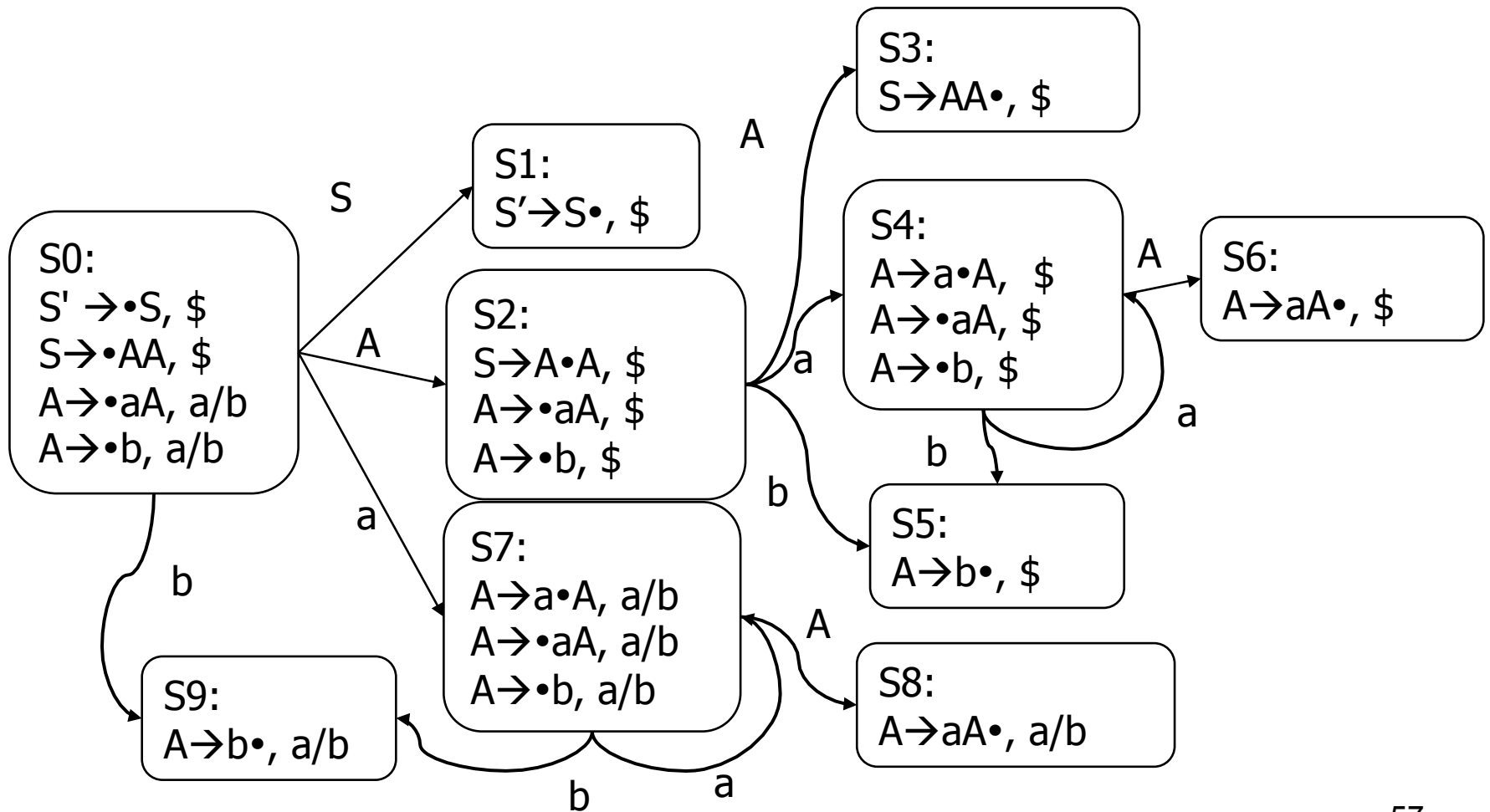
Create the LR(1) parse table

	Action					Goto	
	a	b	c	d	\$	S	A
S0			S9	S2		1	7
S1					A		
S2			S3				4
S3	S5	R5					
S4		S6					
S5					R2		
S6					R3		
S7	S8						
S8					R4		
S9	R5						

Another LR(1) example

- Create the transition diagram

- | | |
|----|--------------------|
| 0) | $S' \rightarrow S$ |
| 1) | $S \rightarrow AA$ |
| 2) | $A \rightarrow aA$ |
| 3) | $A \rightarrow b$ |



Parse table

state	Action			Goto	
	a	b	\$	S	A
S0	S7	S9		1	2
S1			Accept		
S2	S4	S5			3
S3			R1		
S4	S4	S5			6
S5			R3		
S6			R2		
S7	S7	S9			8
S8	R2	R2			
S9	R3	R3			

Parse trace

stack	remaining input	parse action
S0	baab\$	S9
S0S9	aab\$	R3 $A \rightarrow b$
S0S2	aab	S4
S0S2S4	ab	S4
S0S2S4S4	b	S5
S0S2S4S4S5	\$	R3 $A \rightarrow b$
S0S2S4S4S6	\$	R2 $A \rightarrow aA$
S0S2S4S6	\$	R2 $A \rightarrow aA$
S0S2S3	\$	R1 $S \rightarrow AA$
S0S1	\$	Accept

LR(1) grammar

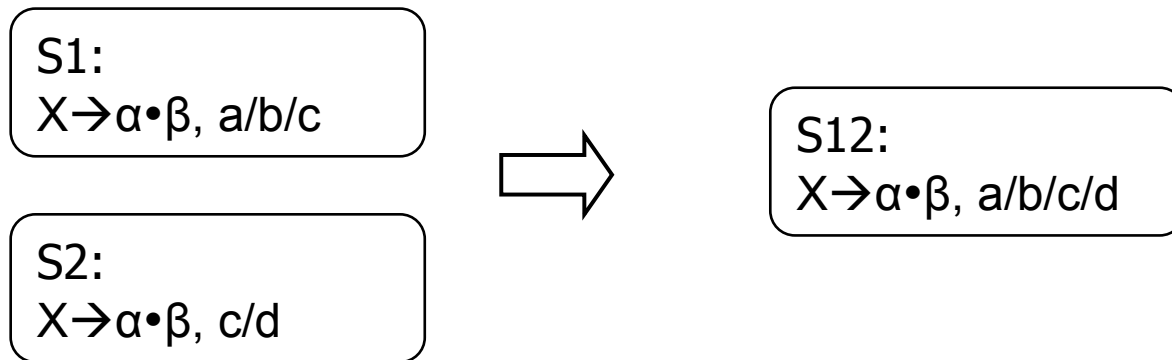
- A grammar is LR(1) if the following 2 conditions are satisfied for each configuration set:
 - For each item $[A \rightarrow \alpha \bullet a \beta, b]$ in the set, there is no item in the set of the form $[B \rightarrow \gamma \bullet, a]$
 - In the action table, this translates to no shift/reduce conflict.
 - If there are two complete items $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \beta \bullet, b]$ in the set, then a and b should be different.
 - In the action table, this translates to no reduce/reduce conflict
- **Compare with the SLR grammar**
 - For any item $A \rightarrow \alpha \bullet a \beta$ in the set, with terminal a , there is no complete item $B \rightarrow \gamma \bullet$ in that set with a in $\text{Follow}(B)$.
 - For any two complete items $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ in the set, the Follow sets must be disjoint.
- Note that $\text{SLR}(1) \subset \text{LR}(1)$
- $\text{LR}(0) \subset \text{SLR}(1) \subset \text{LR}(1)$

LR(1) tables continued

- LR(1) tables can get big – exponential in size of rules
- Can we keep the additional power we got from going SLR \rightarrow LR without table explosion?
 - LALR!
- We split SLR(1) states to get LR(1) states, maybe too aggressively.
- Try to merge item sets that are *almost* identical.
- Tricky bit: Don't introduce shift-reduce or reduce-reduce conflicts.

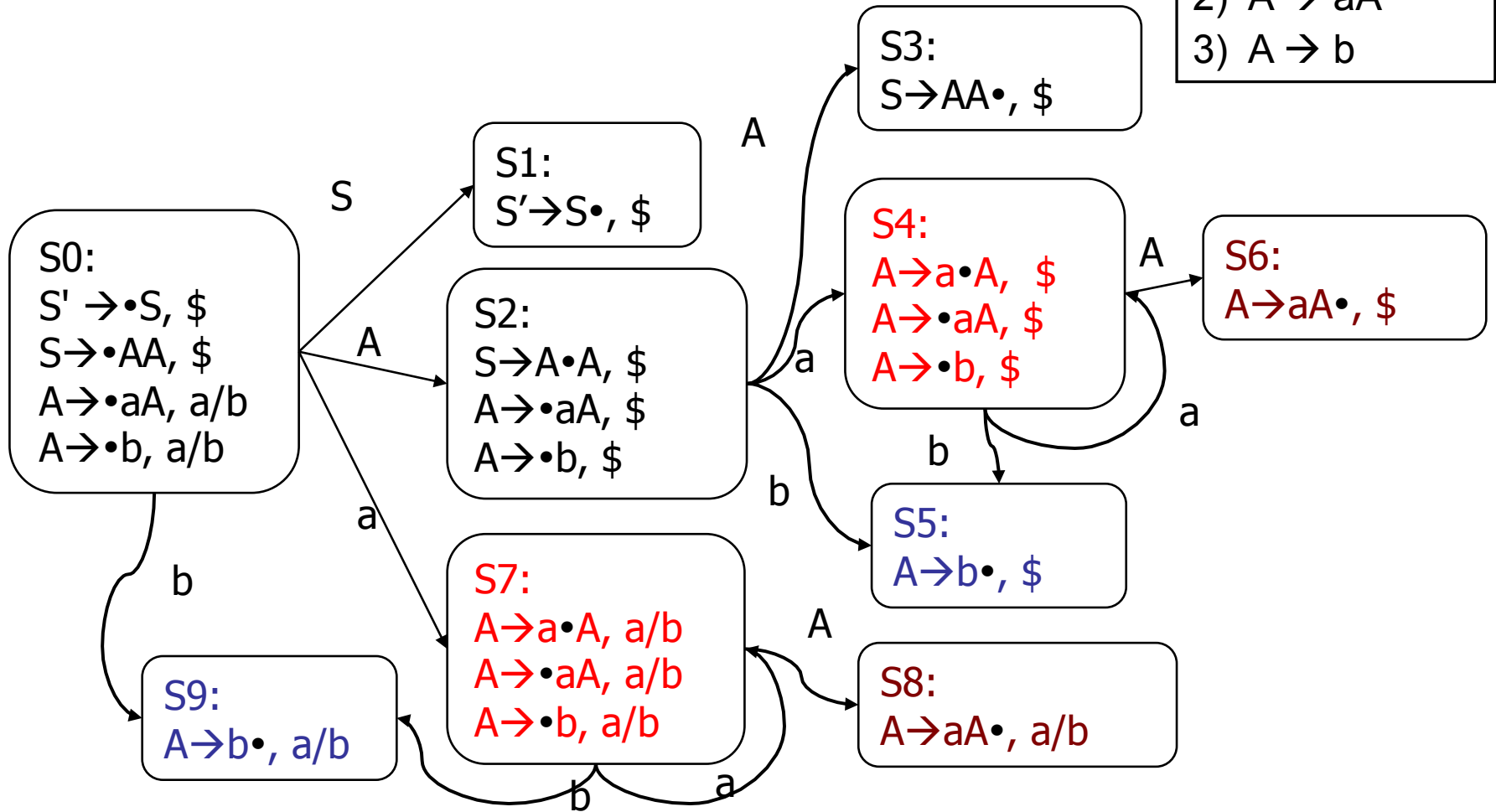
LALR approach

- Just say LALR (it's always 1 in practice)
- Given the numerous LR(1) states for grammar G , consider merging similar states, to get fewer states.
- Candidates for merging:
 - same core (LR(0) item)
 - only differences in lookaheads
- Example:



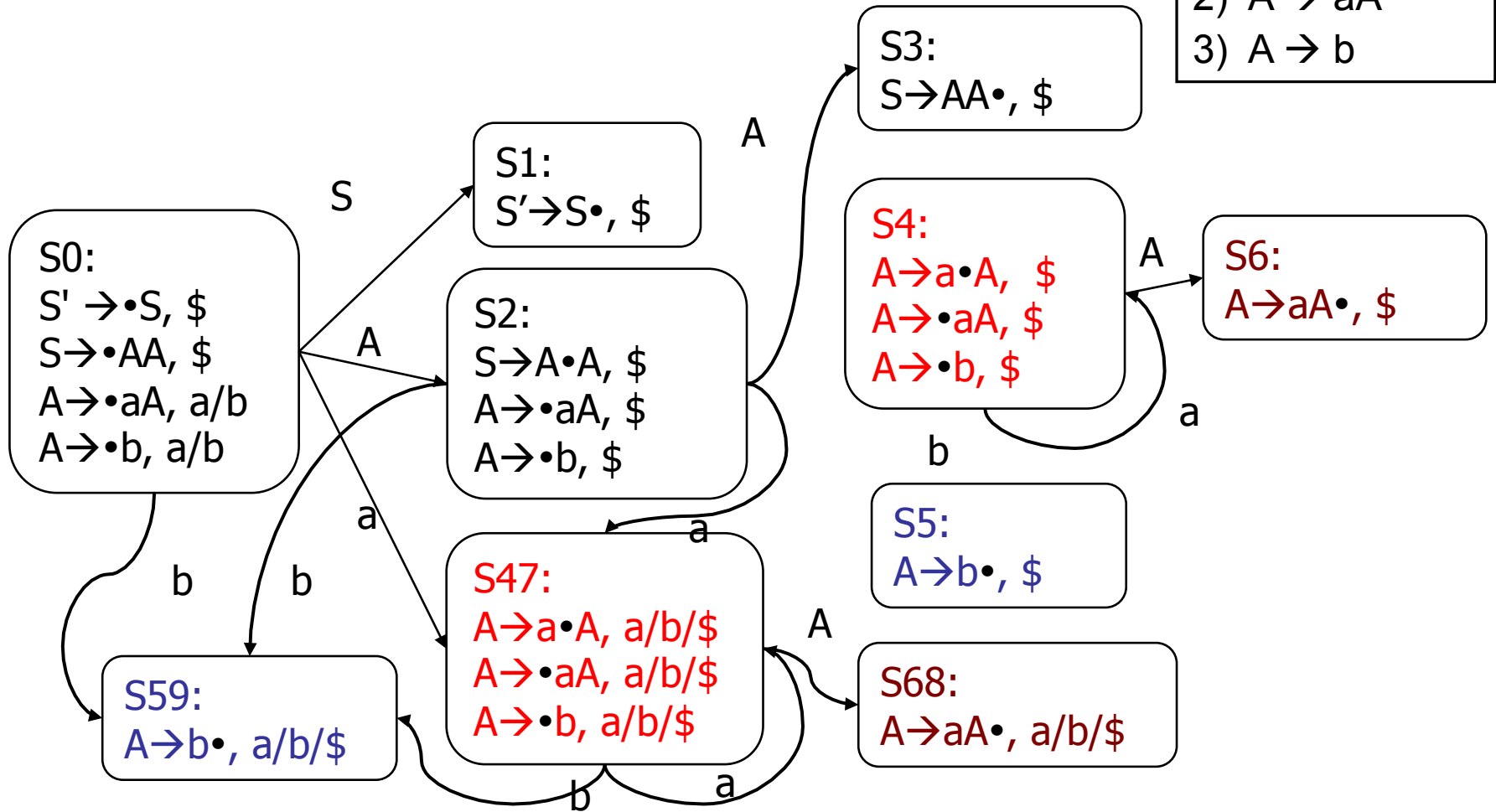
States with same core items

- 0) $S' \rightarrow S$
- 1) $S \rightarrow AA$
- 2) $A \rightarrow aA$
- 3) $A \rightarrow b$



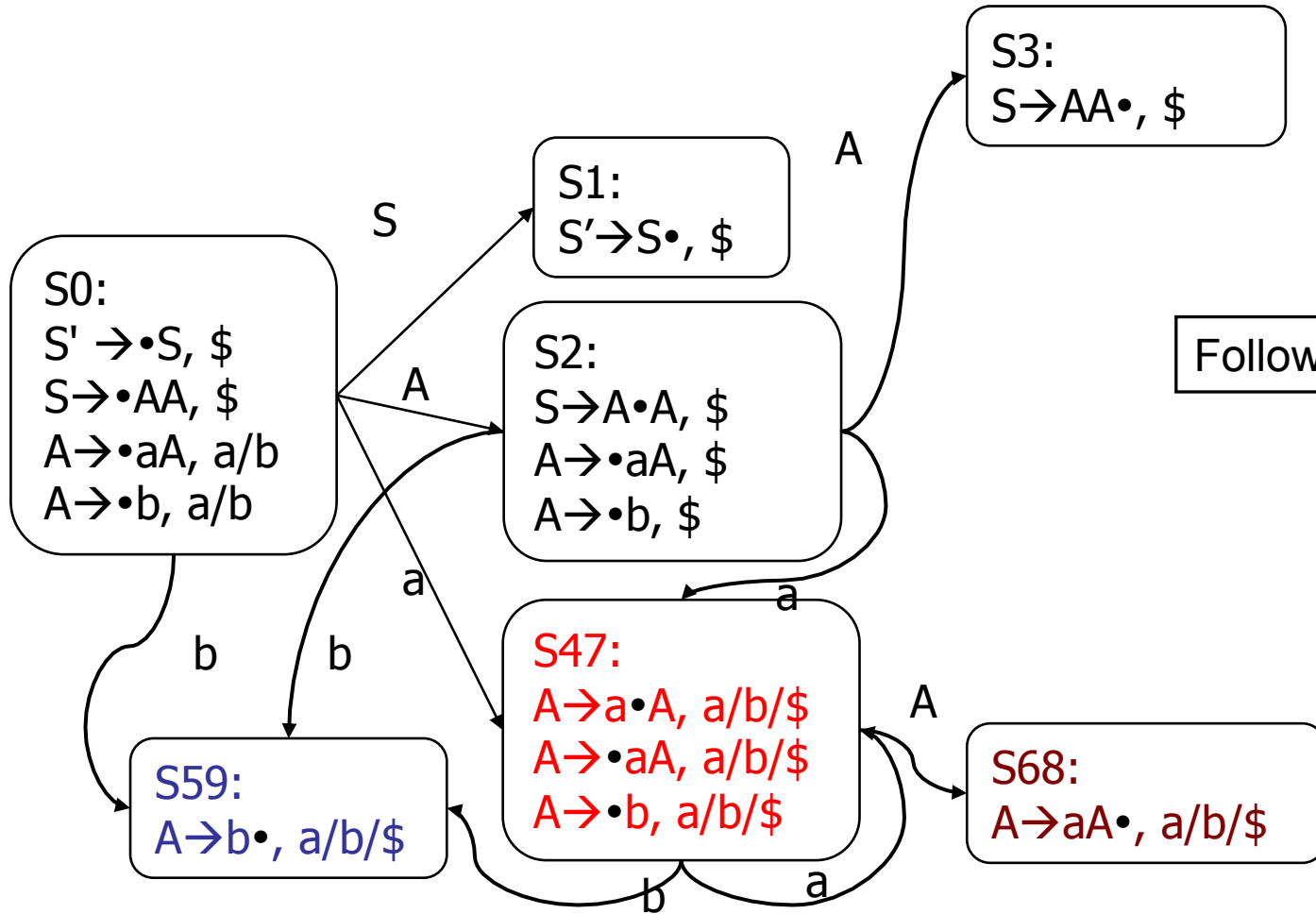
Merge the states

- 0) $S' \rightarrow S$
- 1) $S \rightarrow AA$
- 2) $A \rightarrow aA$
- 3) $A \rightarrow b$



Merge the states

- 0) $S' \rightarrow S$
- 1) $S \rightarrow AA$
- 2) $A \rightarrow aA$
- 3) $A \rightarrow b$



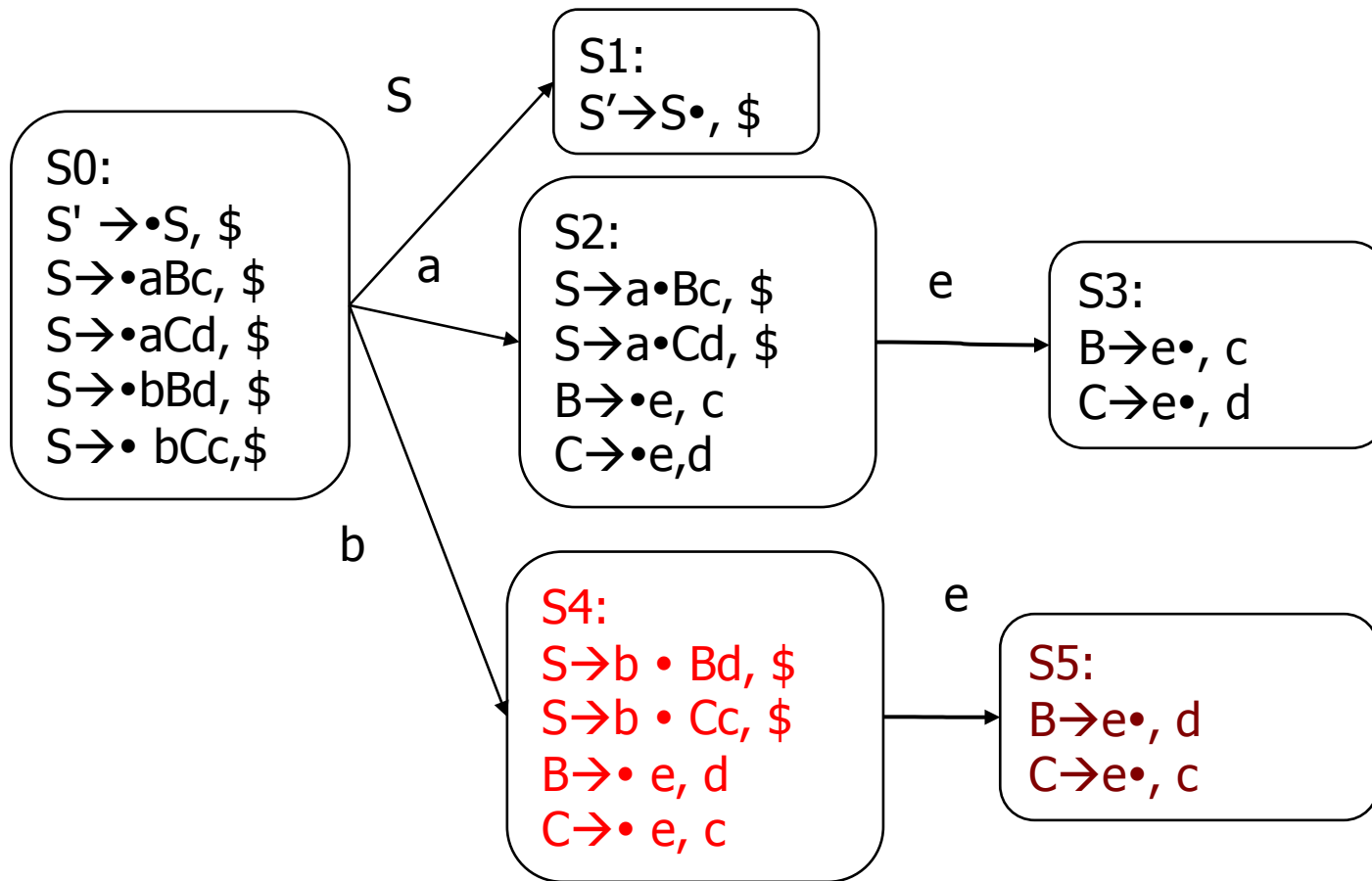
Follow(A)={a b \$}

After the merge

- What happened when we merged?
 - Three fewer states
- Lookahead on items merged.
- In this case, lookahead in merged sets constitutes entire Follow set.
- So, we made SLR(1) grammar by merging.
- Result of merge usually not SLR(1).

conflict after merging

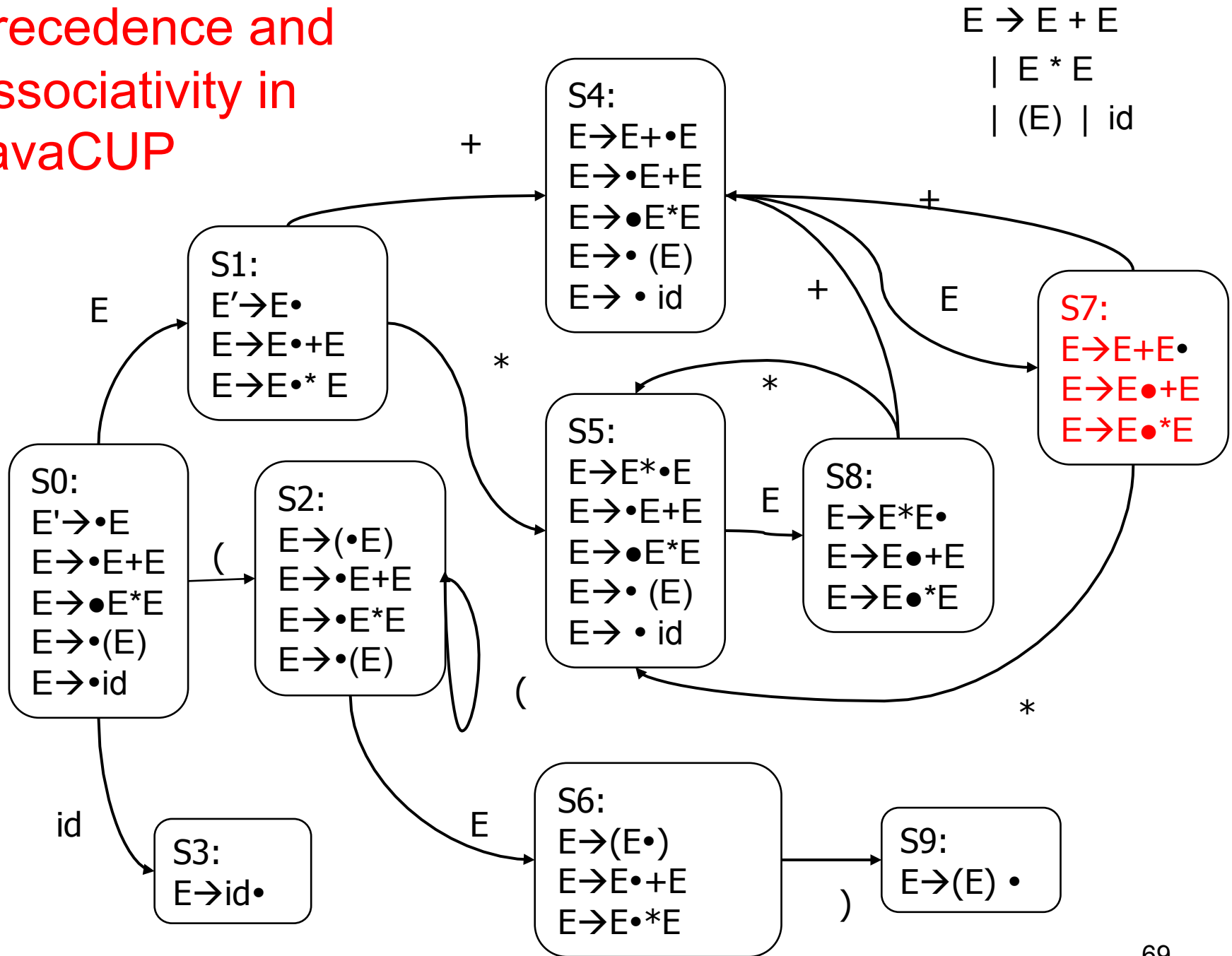
- 1) $S \rightarrow aBc|aCd|bBd|bCc$
- 2) $B \rightarrow e$
- 3) $C \rightarrow e$



Practical consideration

- Ambiguity in LR grammars G : G produces multiple rightmost derivations. (i.e. can build two different parse trees for one input string.)
- Remember:
 $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- We added terms and factors to force unambiguous parse with correct precedence and associativity
- What if we threw the grammar into an LR-grammar table-construction machine anyway?
- Conflicts = multiple action entries for one cell
 - We choose which entry to keep, toss others

Precedence and Associativity in JavaCUP



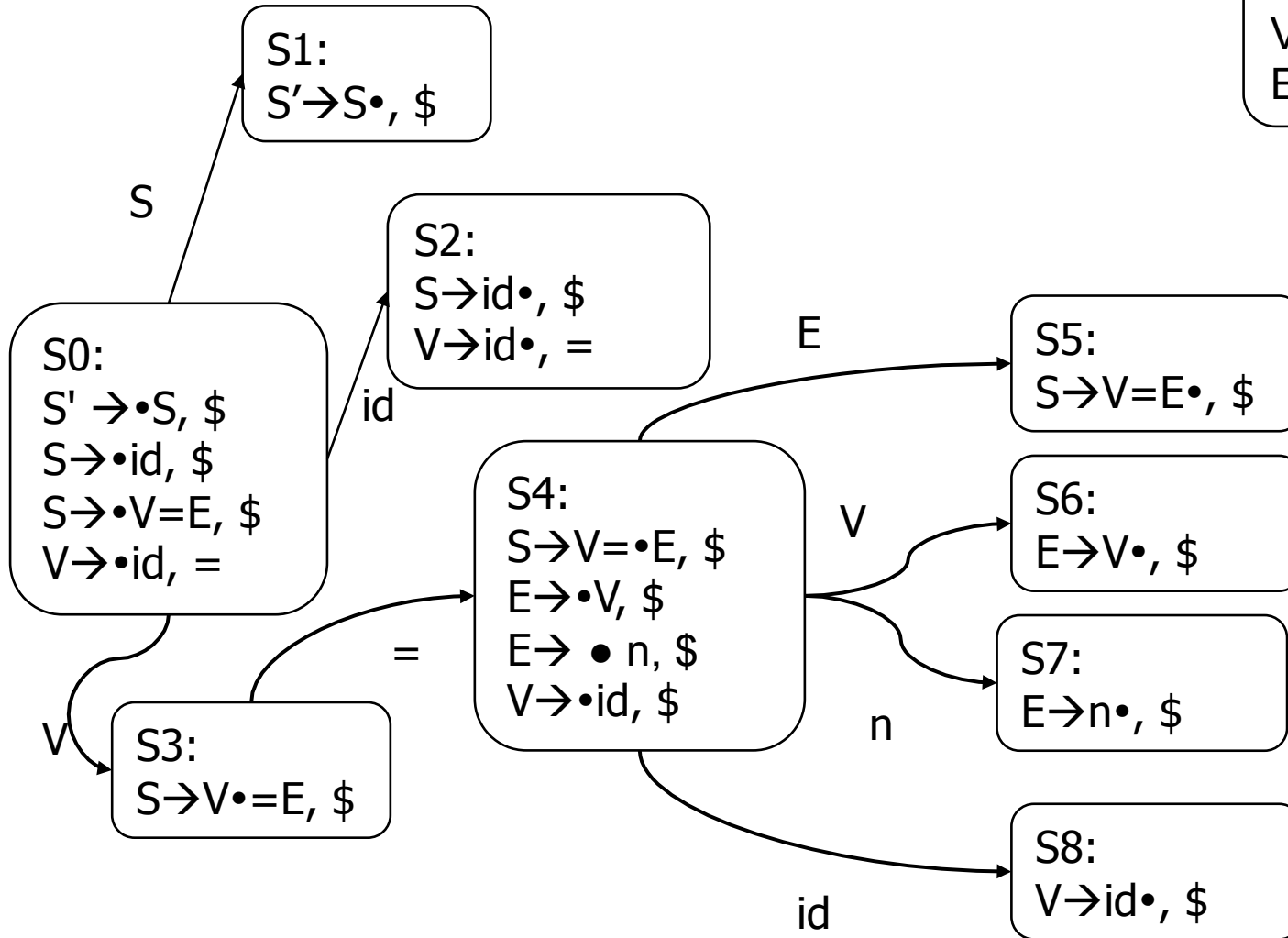
JavaCup grammar

```
terminal PLUS, TIMES;  
precedence left PLUS;  
precedence left TIME;  
E ::= E PLUS E | E TIMES E | ID
```

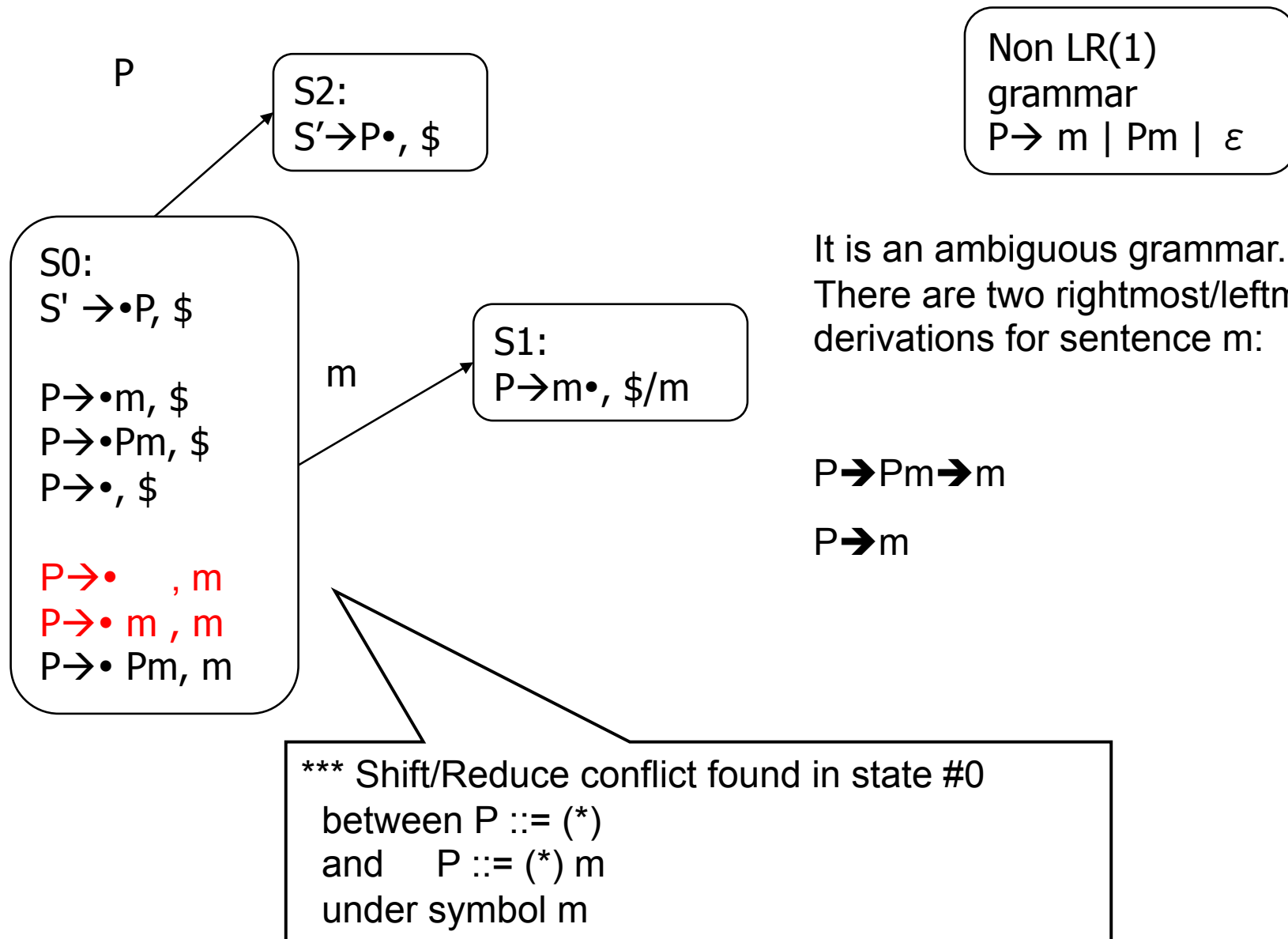
- What if the input is $x+y+z$?
 - When shifting $+$ conflicts with reducing a production containing $+$, choose reduce
- What if the input is $x+y*z$?
- What if the input is $x*y+z$?

Transition diagram for assignment expr

$S \rightarrow id \mid V = E$
 $V \rightarrow id$
 $E \rightarrow V \mid n$

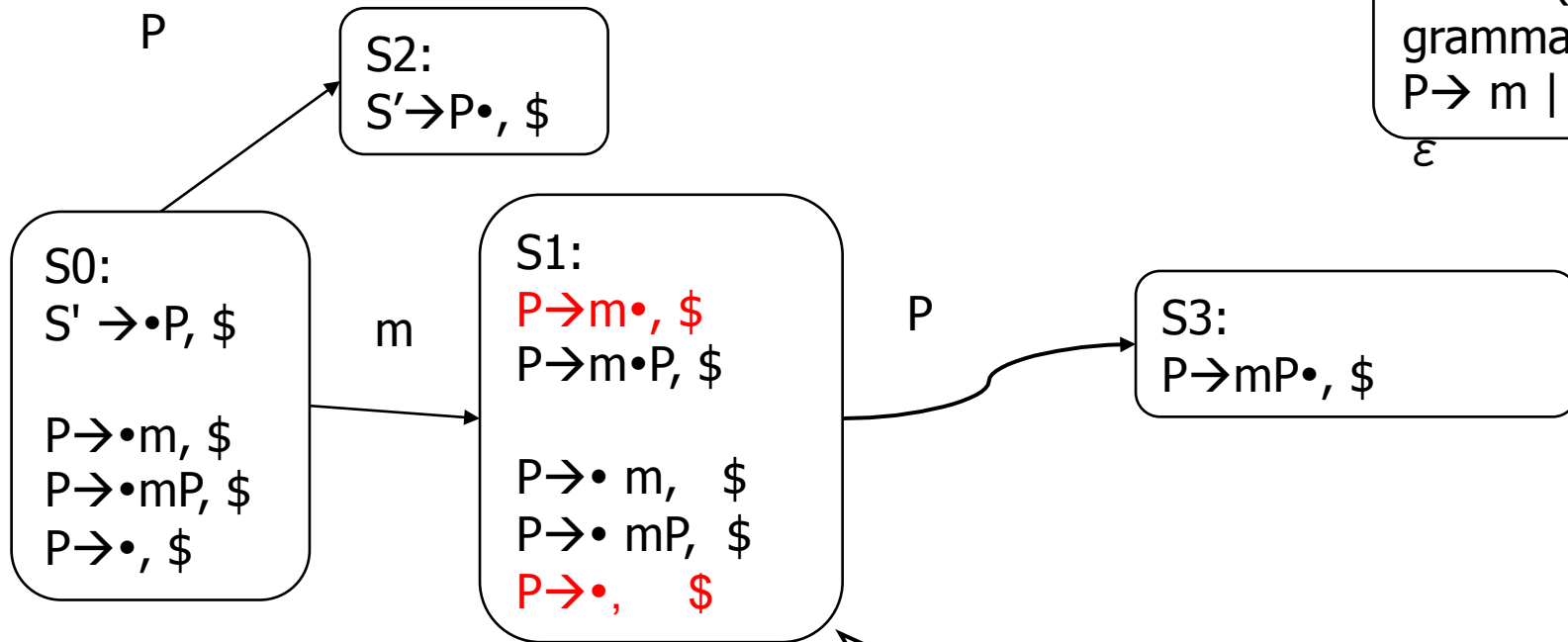


Why are there conflicts in some rules in assignments?



a slightly changed grammar, still not LR

Non LR(1)
grammar
 $P \rightarrow m \mid m P \mid \epsilon$



It is an ambiguous grammar. There are two parse trees for sentence m :

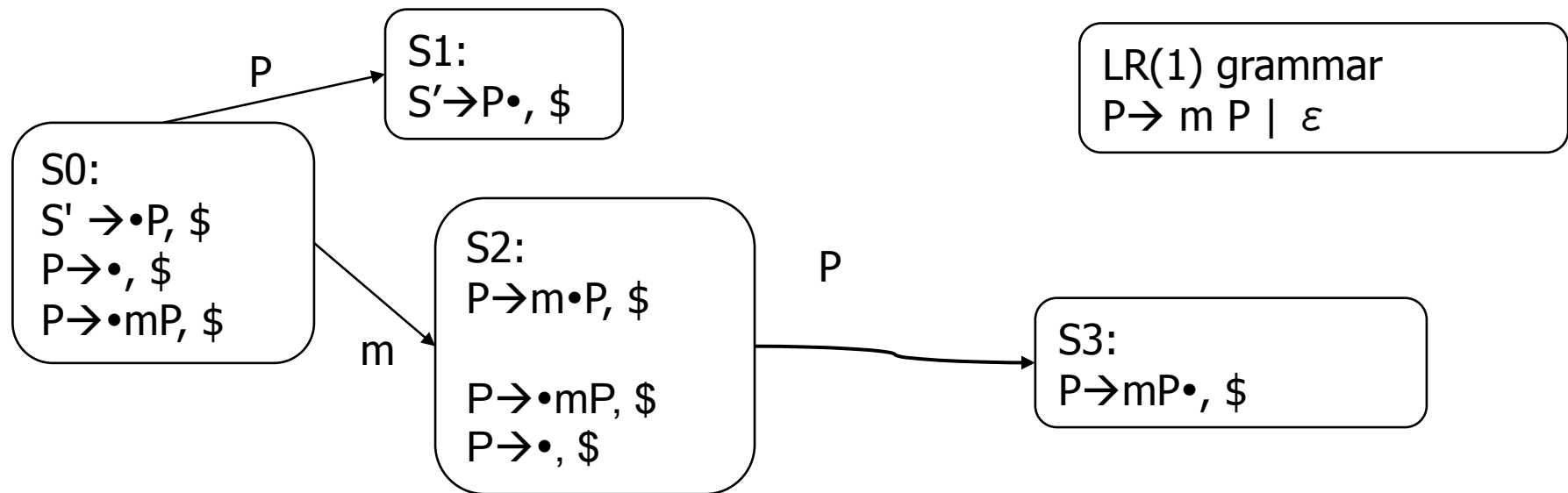
$P \rightarrow mP \rightarrow m$

$P \rightarrow m$

Reduce/Reduce conflict found in state #1
between $P ::= m (*)$
and $P ::= (*)$
under symbols: {EOF}

Produced from javacup

Modified LR(1) grammar



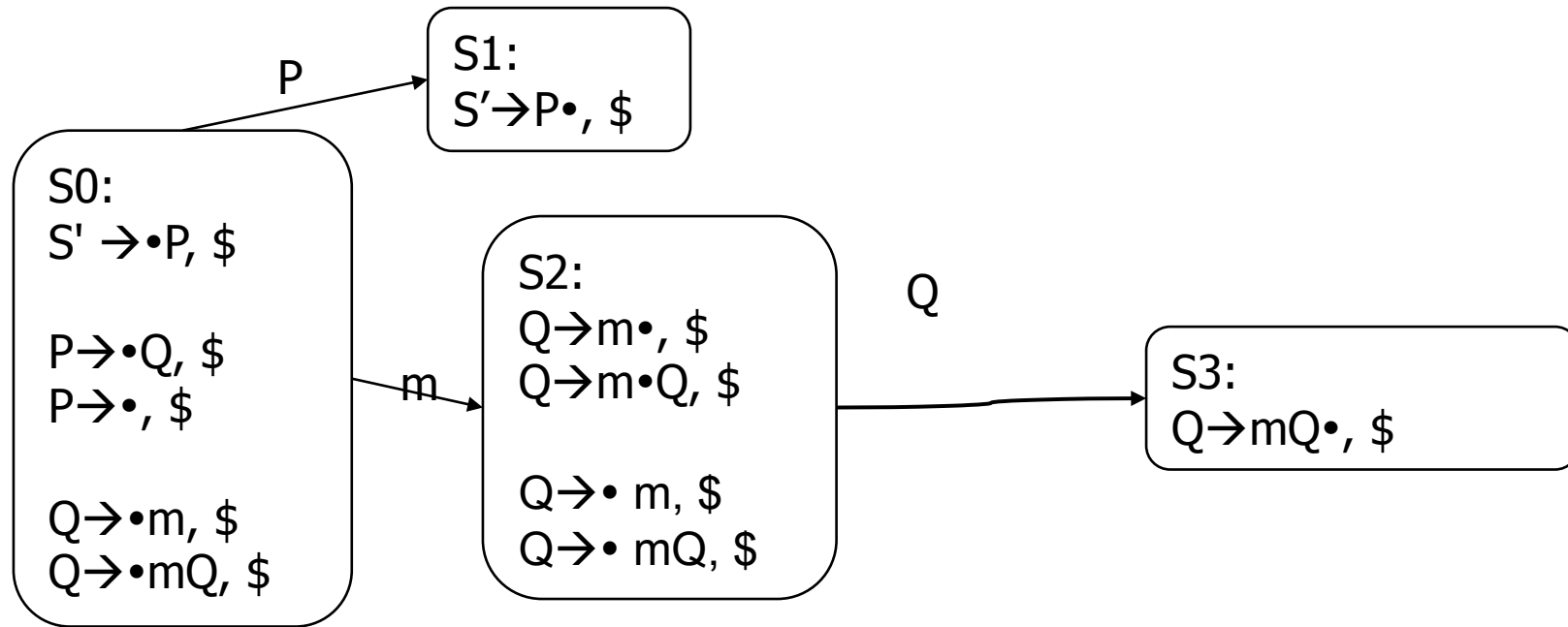
- Note that there are no conflicts
- The derivation:
 - $P \rightarrow mP \rightarrow mmP \rightarrow mm$

Another way of changing to LR(1) grammar

LR(1) grammar

$P \rightarrow Q \mid \varepsilon$

$Q \rightarrow m \mid mQ$

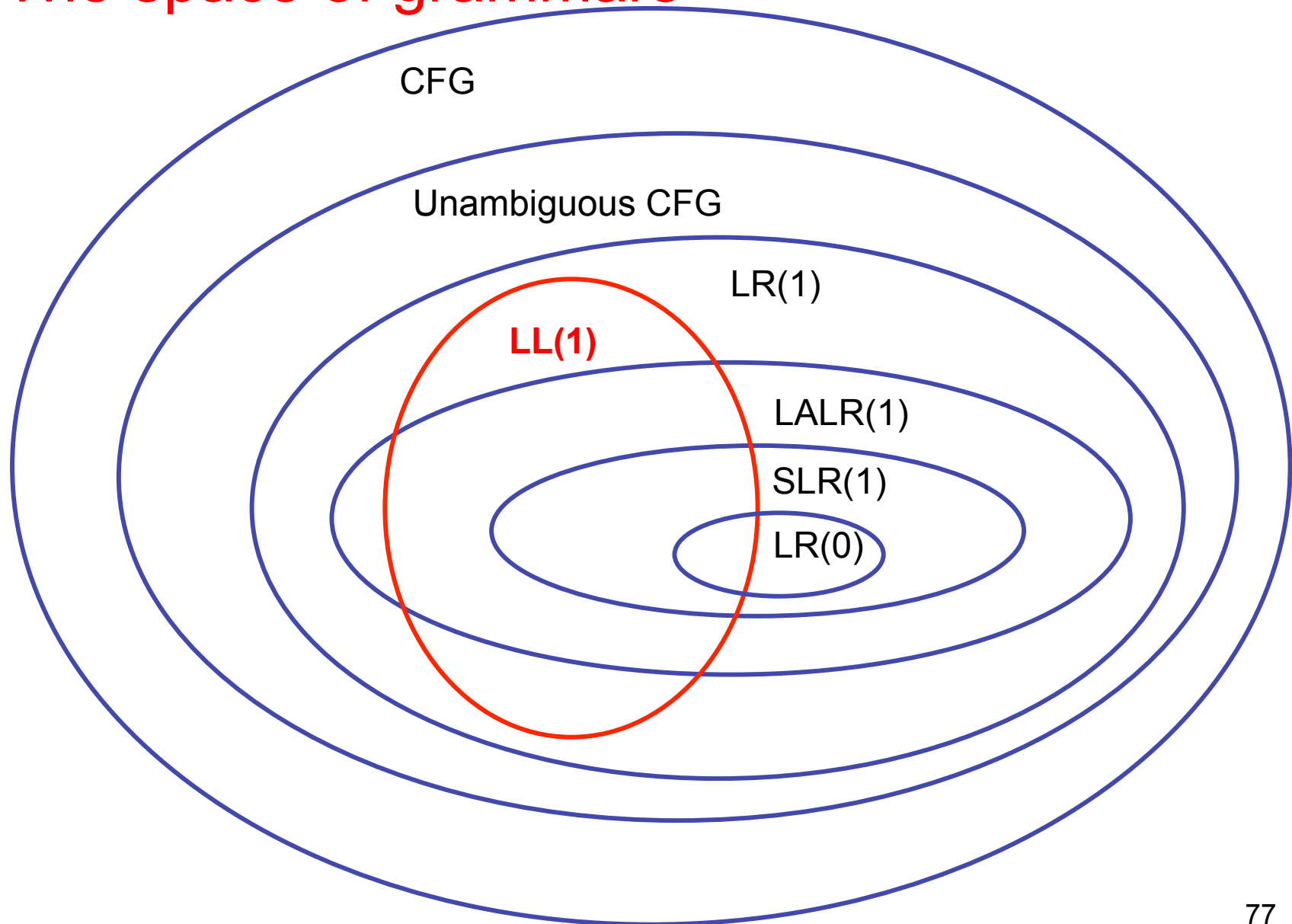


LR grammars: comparison

$LR(0) \subset SLR(1) \subset LALR \subset LR(1) \subset CFG$

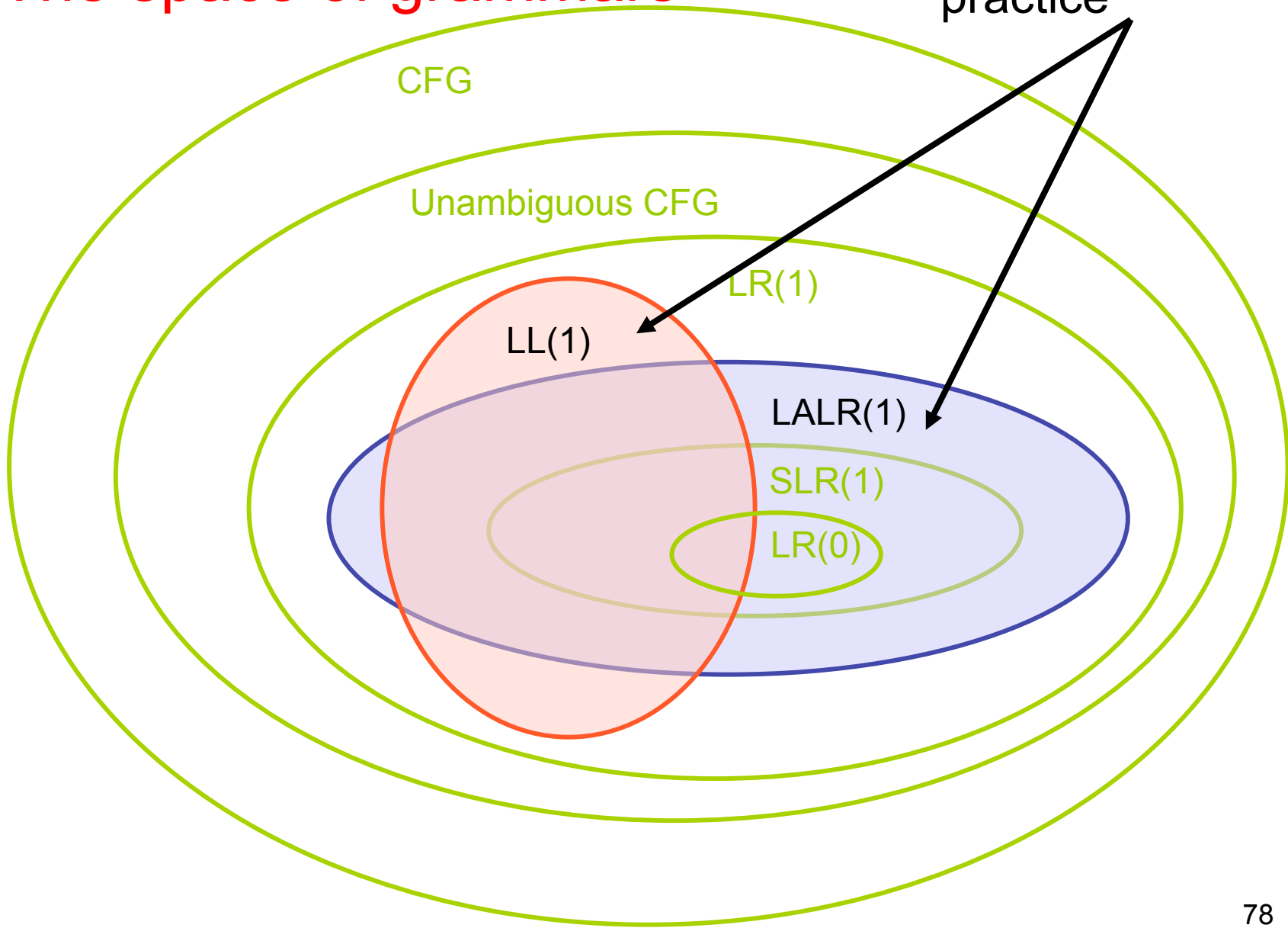
	Advantages	Disadvantages
LR(0)	Smallest tables, easiest to build	Inadequate for many PL structures
SLR(1)	More inclusive, more information than LR(0)	Many useful grammars are not SLR(1)
LALR(1)	Same size tables as SLR, more langs, efficient to build	empirical, not mathematical
LR(1)	Most precise use of lookahead, most PL structures we want	Tables order of magnitude > SLR(1)

The space of grammars



The space of grammars

What are used in practice

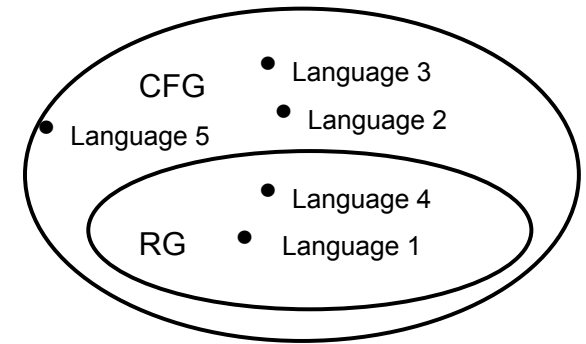
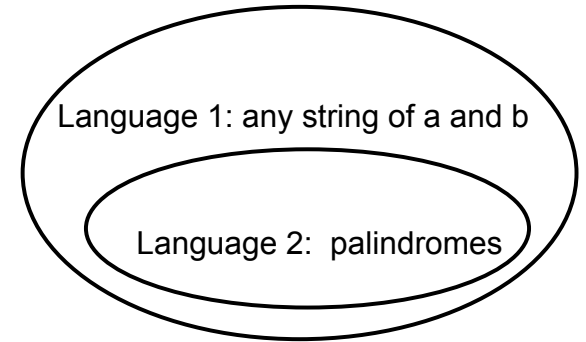


Verifying the language generated by a grammar

- To verify a grammar:
 - every string generated by G is in L
 - every string in L can be generated by G
- Example: $S \rightarrow (S)S | \epsilon$
 - the language is all the strings of balanced parenthesis, such as (), (()), ()
((()))
- Proof part 1: every sentence derived from S is balanced.
 - basis: empty string is balanced.
 - induction: suppose that all derivations fewer than n steps produce balanced sentences, and consider a leftmost derivation of n steps. such a derivation must be of the form:
$$S \Rightarrow (S)S \Rightarrow^*(x)S \Rightarrow^*(x)y$$
- Proof part 2: every balanced string can be derived from S
 - Basis: the empty string can be derived from S.
 - Induction: suppose that every balanced string of length less than 2n can be derived from S. Consider a balanced string w of length 2n. w must start with (. w can be written as (x)y, where x, y are balanced.

Hierarchy of grammars

- CFG is more powerful than RE
- “Type n grammar is more powerful than type n+1 grammar”
- Example: $\Sigma = \{a, b\}$
- The language of any string consists of a and b
 - $A \rightarrow aA | bA | \epsilon$
 - Can be describe by RE
- The language of palindromes consist of a and b
 - $A \rightarrow aAa | bAb | a | b | \epsilon$
 - Can be described by CFG, but not RE
- When a grammar is more powerful, it is not that it can describe a larger language. Instead, the ‘power’ means the ability to restrict the set.
- More powerful grammar can define
 - more complicated boundary between correct and incorrect sentences.
 - Therefore, more different languages



Metaphoric comparison of grammars

- RE draw the rose use straight lines (ruler and T-square suffice)
- CFG approximate the outline by straight lines and circle segments (ruler and compasses)

