CMSC 430, Practice Problems 1 (Solutions)

1. Consider the following grammar: S → S and S | S or S | (S) | true | false
   a. Compute First sets for each production and nonterminal
      FIRST(true) = { "true" }
      FIRST(false) = { "false" }
      FIRST((S)) = { "(" }
      FIRST(S and S) = FIRST(S or S) =
      FIRST(S) = { "(", "true", "false" }
   b. Explain why the grammar cannot be parsed by a LL(1) parser
      First sets of productions intersect, grammar is left recursive

2. Consider the following grammar: S → abS | acS | c
   a. Compute First sets for each production and nonterminal
      FIRST(abS) = { a } **FIRST(acS) = { a }**
      FIRST(c) = { c } **FIRST(S) = { a, c }**
   b. Show why the grammar cannot be parsed by a LL(1) parser.
      First sets of productions overlap
      FIRST(abS) ∩ FIRST(acS) = { a } ∩ { a } = { a } ≠ ∅
   c. Rewrite the grammar so it can be parsed by a LL(1) parser.
      S → aL | c **L → bS | cS**

3. Consider the following grammar: S → Sa | Sc | c
   a. Show why the grammar cannot be parsed by a predictive parser.
      First sets of productions intersect, grammar is left recursive
   b. Rewrite the grammar so it can be parsed by a predictive parser.
      S → c L **L → aL | cL | ε**
   c. Using FIRST and FOLLOW , build the LL(1) table for the new grammar
      FIRST$^+$ (L → ε) = FOLLOW(L) = {$}

	a	c	$
S		**S → c L**	
L	**L → aL**	**L → cL**	**L → ε**

   d. Using the LL(1) table, show how a table-driven parser parses the string "caca"

Stack	Input	Action
[$, S]	caca$	S → c L, pop S + push L c
[$, L, c]	caca$	next input + pop c
[$, L]	aca$	L → aL, pop L + push L a
[$, L, a]	aca$	next input + pop a
[$, L]	ca$	L → cL, pop L + push L c
[$, L, c]	ca$	next input + pop c
[$, L]	a$	L → aL, pop L + push L a
[$, L, a]	a$	next input + pop a
[$, L]	$	L → ε, pop L
[$]	$	accept

4. Consider the following grammar

Goal	→	Expr
Expr	→	Term + Expr
	|	Term – Expr
	|	Term
Term	→	Factor * Term
	|	Factor / Term
	|	Factor
Factor	→	num
	|	id

a. Apply left factoring to create a LL(1) grammar
b. Compute FIRST and FOLLOW for the modified grammar

Left Factored Grammar			**FIRST**		**FOLLOW**
			Productions	**Nonterminals**	
Goal	**→**	**Expr**	**{ num, id }**	**Goal { num, id }**	**{ $ }**
Expr	**→**	**Term Expr'**	**{ num, id }**	**Expr { num, id }**	**{ $ }**
Expr'	**→**	**+ Expr'**	**{ + }**	**Expr' { +, -, ε }**	**{ $ }**
	|	**– Expr'**	**{ – }**	**Term { num, id }**	**{ +, -, $ }**
	|	**ε**	**{ ε }**	**Term' { *, /, ε }**	**{ +, -, $ }**
Term	**→**	**Factor Term'**	**{ num, id }**	**Factor { num, id }**	**{ *, /, +, -, $ }**
Term'	**→**	*** Term**	**{ * }**		
	|	**/ Term**	**{ / }**		
	|	**ε**	**{ ε }**		
Factor	**→**	**num**	**{ num }**		
	|	**id**	**{ id }**		

c. Using FIRST and FOLLOW , build the LL(1) table for the new grammar

	num	id	+	-	*	/	$
Goal	Goal → Expr	Goal → Expr					
Expr	Expr → Term Expr'	Expr → Term Expr'					
Expr'			Expr' → + Expr'	Expr' → - Expr'			Expr' → ε
Term	Term → Factor Term'	Term → Factor Term'					
Term'			Term' → ε	Term' → ε	Term' → * Term	Term' → / Term	Term' → ε
Factor	Factor → num	Factor → id					

5. Bottom up parsing
    a. Describe the difference between bottom-up and top-down parsing with respect to a grammar production A → β

        Bottom up parsers perform reductions of β to A, while top-down parsers perform expansions of A to β.
    b. Define handle

        The right-hand side of the next production to be reduced in the right-most derivation, and its position in the current sentential form.
    c. Explain why bottom-up parsers result in right-most derivations in reverse.

        Because input is processed from left to right and unprocessed input is composed only of terminals, a right-most derivation is used so the sentential form to the right of the handle is composed entirely of terminals.
    d. Given the following ACTION/GOTO table, perform a parse of "a", assuming 0 is the start state. Show the contents of the stack, remaining input, and action performed at each step of the shift-reduce parse.

State	Action		Goto	
	a	$	A	B
0	shift 1	reduce $B \rightarrow \epsilon$	2	3
1	shift 3	reduce $A \rightarrow a$	0	
2		accept	3	0
3	shift 1	accept		1

Stack	Input	Action
$ 0	a $	shift 1
$ 0 a 1	$	reduce $A \rightarrow a$
$ 0 A	$	goto[0,A] = 2
$ 0 A 2	$	accept

6. LR parsing
    a. What is the canonical set of LR(k) items for a grammar?

        A deterministic finite automaton (DFA) for recognizing handles (right-hand sides of productions) for the grammar.
    b. What are ACTION/GOTO tables of an LR(k) parser for a grammar?

        Tables encoding the DFA for recognizing handles for the grammar.
    c. Given an LR(k) item $[A \rightarrow \alpha \bullet \gamma, \delta]$ from a production $A \rightarrow \beta$, describe the role of A, α, γ, δ

        A is the nonterminal that is potentially the result of a reduction. α is the portion of β that the parser has just seen. γ is the portion of β the parser will need to see before performing the reduction to A. δ is k terminals that must be the k lookaheads for the reduction to be performed.

d. Given an LR(k) item [$A \rightarrow \alpha \bullet \gamma,\ \delta$], what lookahead allows a shift to be performed (ignoring any other LR(k) items added to the state by closure)?

The terminal at the beginning of γ.

e. Given an LR(k) item [$A \rightarrow \alpha \bullet \gamma,\ \delta$], what LR(k) items must be added to the closure?

Any production for the nonterminal B at the beginning of γ, with • at the beginning of the right-hand side of the production, and FIRST(γ'δ) as the lookahead token, where γ' is what remains in γ after removing the first B.

f. If the parser is in a state containing an LR(k) item [$A \rightarrow \alpha \bullet \gamma,\ \delta$], what must be at the top of the stack, and why?

The top of the stack must be α, since the parser just saw α.

g. Describe what constitutes a shift/reduce conflict

A DFA state contains both LR items allowing shift and reduce for the same lookahead.

h. Give an example of two LR(1) items causing a reduce/reduce conflict.

[A → α•, δ] and [B → α•, δ]

i. Explain why there are no shift/shift conflicts

Because no decision needs to be made about which right-hand side of a production is the handle.

7. LALR parsing.

Consider the following sets of LR(1) items in the states of a LR(1) parser.

State 0:
[A → • a , b]
[A → a • , c]
[B → a • , b]

State 1:
[A → • a , a]
[A → • a , b]
[B → a • , b]

State 2:
[A → • a , c]
[A → a • , b]
[B → a • , a]

State 3:
[A → • a , b]
[B → • a , b]

a. Find all shift/reduce and reduce/reduce conflicts. List the LR(1) items and lookaheads causing conflicts.

Shift/reduce conflict is caused by [A → •a, c] and [B → a•, a] on lookahead a in state 2.

b. What states would be merged in a LALR(1) parser?

States 0 and 2.

c. Are any new reduce/reduce conflicts introduced in the LALR(1) parser?

Reduce/reduce conflict is caused by [A → a•, b] and [B → a•, b] on lookahead b in the merged state.

d. Explain why LALR(1) parsers will not introduce new shift/reduce conflicts.

Because the core of all merged states must be identical, the LR item for the shift must have been in all of the states prior to the merge. Any LR item causing a reduce must have come from some state, and the same shift/reduce conflict would have existed in that state.

e. What is the relationship between LALR(1) and SLR(1) parsers?

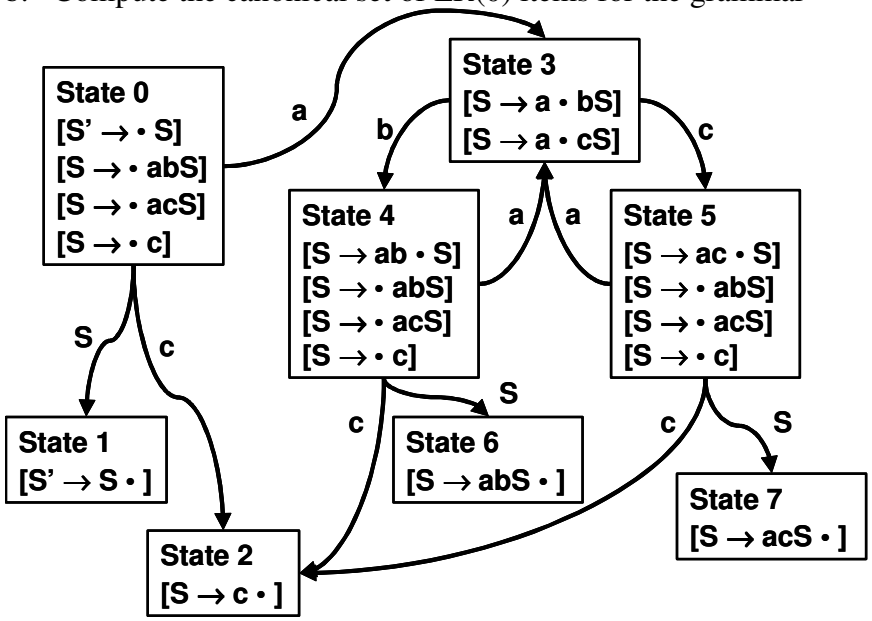
LALR(1) parsers are strictly more powerful than SLR(1) parsers, so any grammar that can be parsed by an SLR(1) parser may be parsed by a LALR(1) parser, but there are grammars that can be parsed by LALR(1) parsers that cannot be parsed by SLR(1) parsers.

8. Consider the following grammar: S → abS | acS | c

a. Compute FOLLOW(S)

FOLLOW(S) = { $ }

b. Compute the canonical set of LR(0) items for the grammar



c. Build the ACTION/GOTO table using LR(0) + FOLLOW

	Action				Goto
State	a	b	c	$	S
0	Shift 3		Shift 2		1
1				Accept	
2				Reduce S → c	
3		Shift 4	Shift 5		
4	Shift 3		Shift 2		6
5	Shift 3		Shift 2		7
6				Reduce S → abS	
7				Reduce S → acS	

d. Explain why the grammar is or is not SLR(1)

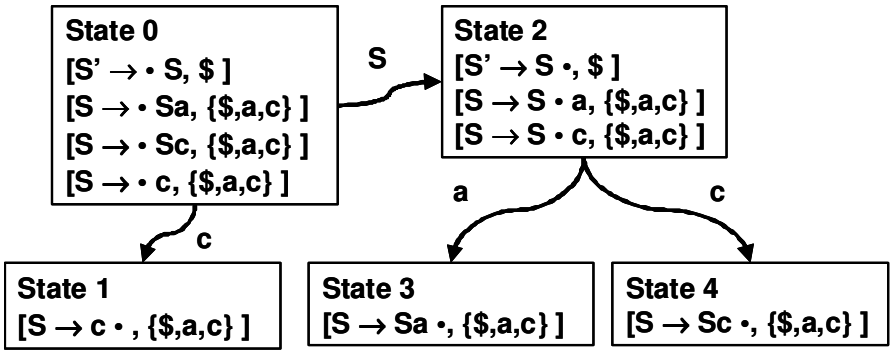
It is SLR(1) since the SLR(1) ACTION/GOTO table has no conflicts.

e. Use the ACTION/GOTO table to parse the string "abc"

Stack	Remaining Input	Action	Comment
$ 0	a b c $	Shift 3	
$ 0 a 3	b c $	Shift 4	
$ 0 a 3 b 4	c $	Shift 2	
$ 0 a 3 b 4 c 2	$	Reduce S → c	
$ 0 a 3 b 4 S	$	Goto 6	Intermediate Step
$ 0 a 3 b 4 S 6	$	Reduce S → abS	
$ 0 S	$	Goto 1	Intermediate Step
$ 0 S 1	$	Accept	

9. Consider the following grammar: S → Sa | Sc | c

a. Compute the canonical set of LR(1) items for the grammar



b. Build the ACTION/GOTO table for the grammar

	Action			Goto
State	a	c	$	S
0		Shift 1		2
1	Reduce S → c	Reduce S → c	Reduce S → c	
2	Shift 3	Shift 4	Accept	
3	Reduce S → Sa	Reduce S → Sa	Reduce S → Sa	
4	Reduce S → Sc	Reduce S → Sc	Reduce S → Sc	

c. Explain why the grammar is or is not LR(1)

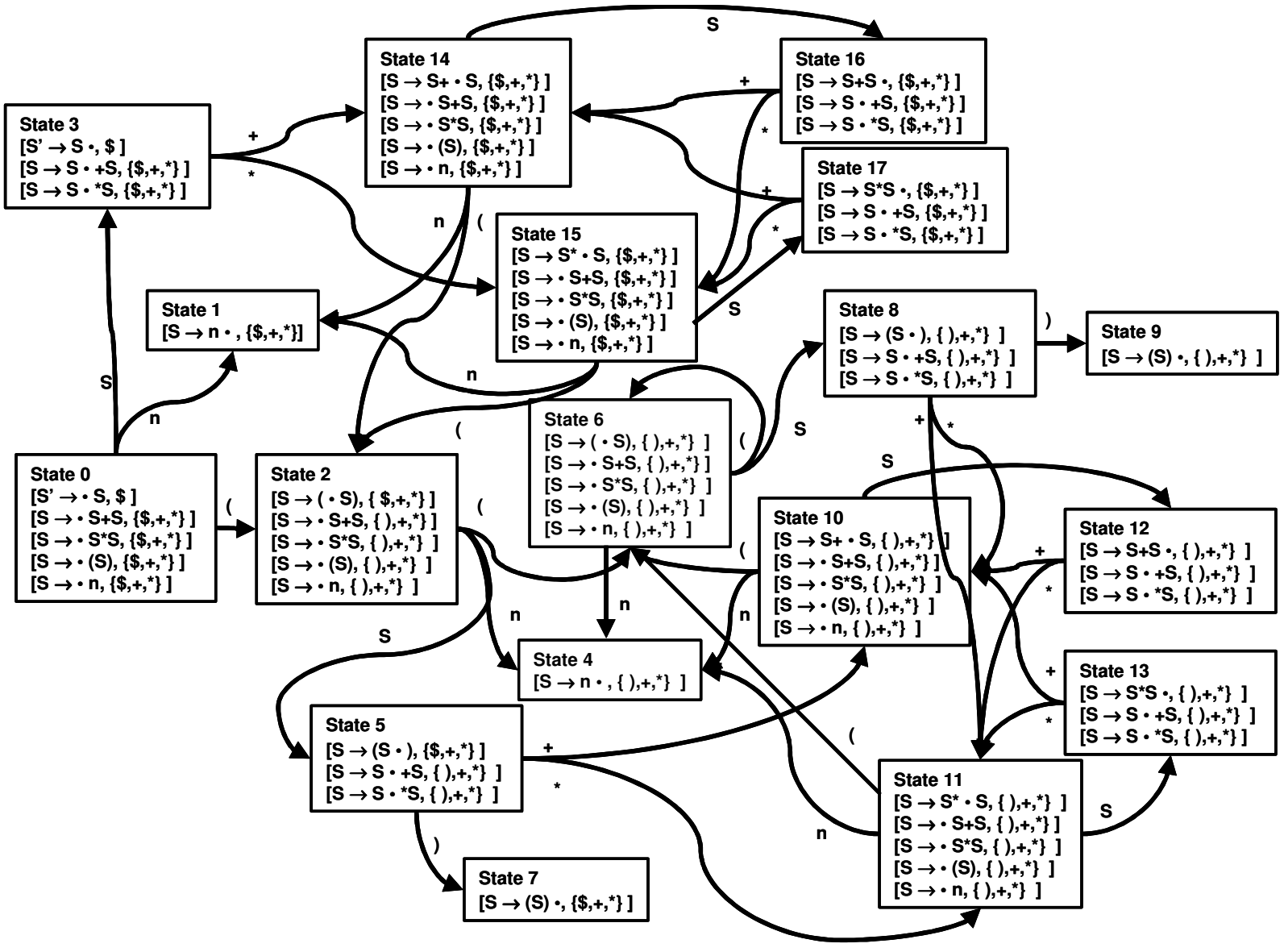
It is LR(1) since the LR(1) ACTION/GOTO table has no conflicts.

d. Use the ACTION/GOTO table to parse the string "caca"

Stack	Remaining Input	Action	Comment
$ 0	c a c a $	Shift 1	
$ 0 c 1	a c a $	Reduce S → c	
$ 0 S	a c a $	Goto 2	Intermediate Step
$ 0 S 2	a c a $	Shift 3	
$ 0 S 2 a 3	c a $	Reduce S → Sa	
$ 0 S	c a $	Goto 2	Intermediate Step
$ 0 S 2	c a $	Shift 4	
$ 0 S 2 c 4	a $	Reduce S → Sc	
$ 0 S	a $	Goto 2	Intermediate Step
$ 0 S 2	a $	Shift 3	
$ 0 S 2 a 3	$	Reduce S → Sa	
$ 0 S	$	Goto 2	Intermediate Step
$ 0 S 2	$	Accept	

10. Consider the following grammar: S → S + S | S * S | (S) | num

a. Compute the canonical set of LR(1) items for the grammar



Build the ACTION/GOTO table for the grammar

	Action						Goto
State	**+**	*****	**(**	**)**	**num**	**$**	**S**
0			**Shift 2**		**Shift 1**		**3**
1	**Reduce S → num**	**Reduce S → num**				**Reduce S → num**	
2			**Shift 6**		**Shift 4**		**5**
3	**Shift 14**	**Shift 15**				**Accept**	
4	**Reduce S → num**	**Reduce S → num**		**Reduce S → num**			
5	**Shift 10**	**Shift 11**		**Shift 7**			
6			**Shift 6**		**Shift 4**		**8**
7	**Reduce S → (S)**	**Reduce S → (S)**				**Reduce S → (S)**	
8	**Shift 10**	**Shift 11**		**Shift 9**			
9	**Reduce S → (S)**	**Reduce S → (S)**		**Reduce S → (S)**			
10			**Shift 6**		**Shift 4**		**12**
11			**Shift 6**		**Shift 4**		**13**
12	**Shift 10 Reduce S → S+S**	**Shift 11 Reduce S → S+S**		**Reduce S → S+S**			
13	**Shift 10 Reduce S → S*S**	**Shift 11 Reduce S → S*S**		**Reduce S → S*S**			
14			**Shift 2**		**Shift 1**		**16**
15			**Shift 2**		**Shift 1**		**17**
16	**Shift 14 Reduce S → S+S**	**Shift 15 Reduce S → S+S**				**Reduce S → S+S**	
17	**Shift 14 Reduce S → S*S**	**Shift 15 Reduce S → S*S**				**Reduce S → S*S**	

b. Explain why the grammar is or is not LR(1)

It is not LR(1) because there are shift/reduce conflicts in states 16 & 17 (and 12 & 13) for lookahead + and *.

c. Explain how to use the ACTION/GOTO table so that

	Lookahead	
State	**+**	*****
16 **S+S on stack**	**Reduce + on left or** **shift + on right onto stack?** **Depends on associativity of +** **(shift if right assoc, reduce if left)**	**Reduce + on left or** **shift * on right onto stack?** **Depends on precedence of + vs *** **(shift if * > +, reduce if * < +)**
17 **S*S on stack**	**Reduce * on left or** **shift + on right onto stack?** **Depends on precedence of * vs +** **(shift if + > *, reduce if + < *)**	**Reduce * on left or** **shift * on right onto stack?** **Depends on associativity of *** **(shift if right assoc, reduce if left)**

i. * has higher precedence than + and *, + are both left associative
Resolve the shift/reduce conflicts as follows (same for 12 & 13)..

	Lookahead	
State	**+**	*****
16	**Reduce S → S+S**	**Shift 15**
17	**Reduce S → S*S**	**Reduce S → S*S**

ii. * has higher precedence than +, * is left associative, and + is right associative
Resolve the shift/reduce conflicts as follows (same for 12 & 13)..

	Lookahead	
State	**+**	*****
16	**Shift 14**	**Shift 15**
17	**Reduce S → S*S**	**Reduce S → S*S**

iii. + has higher precedence than * and *, + are both right associative
Resolve the shift/reduce conflicts as follows (same for 12 & 13).

	Lookahead	
State	**+**	*****
16	**Shift 14**	**Reduce S → S+S**
17	**Shift 14**	**Shift 15**

11. Consider the following grammar: S → AS | b A → SA | a
a. Compute the canonical set of LR(0) items for the grammar
b. Build the ACTION/GOTO table for the grammar
c. Compute the canonical set of LR(1) items for the grammar
d. Build the ACTION/GOTO table for the grammar
See solution for problem 8 at
http://www.cs.umd.edu/class/spring2006/cmsc430/p1-f03-soln.txt

12. Consider the following grammar: S → S ; a | a

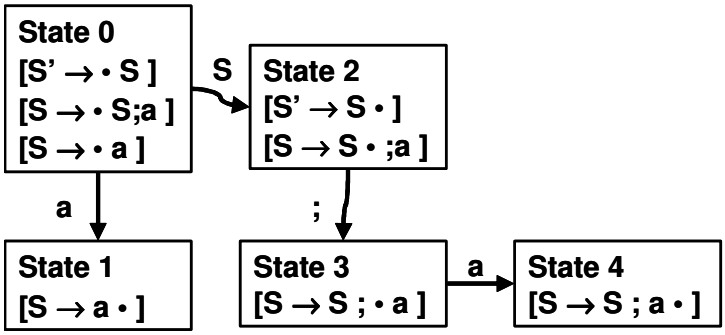
a. Show the grammar is LR(1) but not SLR(1)

SLR(1) parser?

FOLLOW(S') = { $ }

FOLLOW(S) = { $, ; }

Canonical set of LR(0) items =

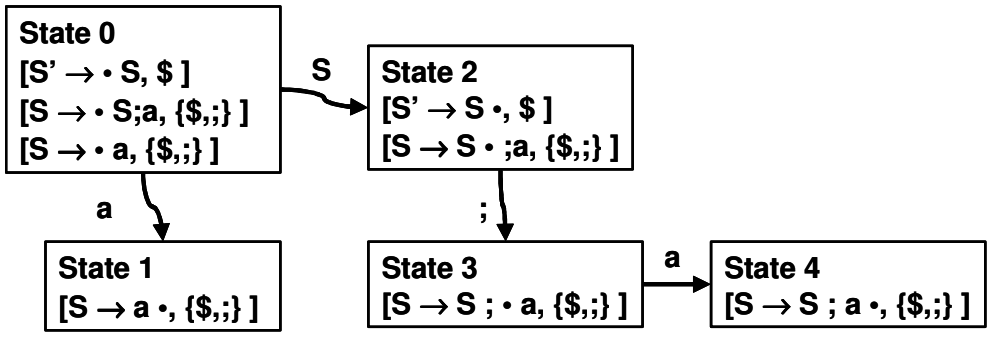


	Action			Goto
State	a	;	$	S
0	Shift 1			2
1		Reduce S → a	Reduce S → a	
2		Shift 3	Accept	
3	Shift 4			
4		Reduce S → S;a	Reduce S → S;a	

No conflicts in any state, grammar is SLR(1)

LR(1) parser?

Canonical set of LR(1) items =



No conflicts in any state, grammar is LR(1)

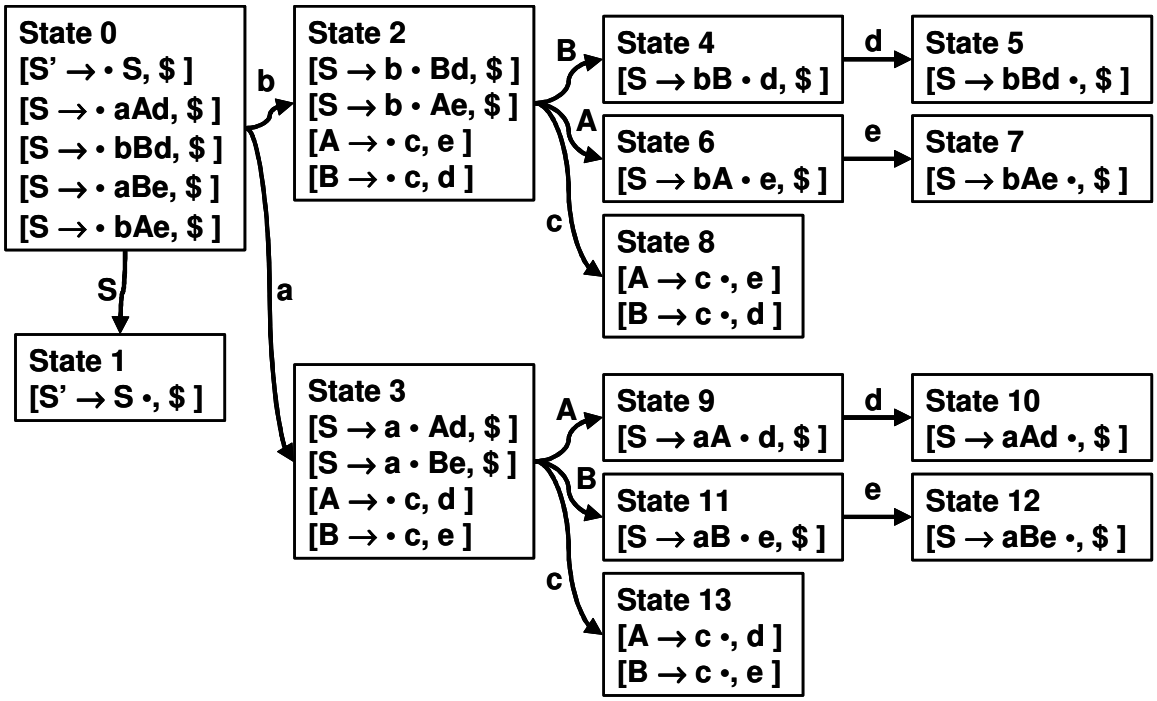
Alternative proof: Since grammar is SLR(1), it must be also LR(1) since LR(1) is a superset of SLR(1).

13. Consider the following grammar: S → aAd | bBd | aBe | bAe A → c B → c

a. Show the grammar is LR(1) but not LALR(1)

LR(1) parser?

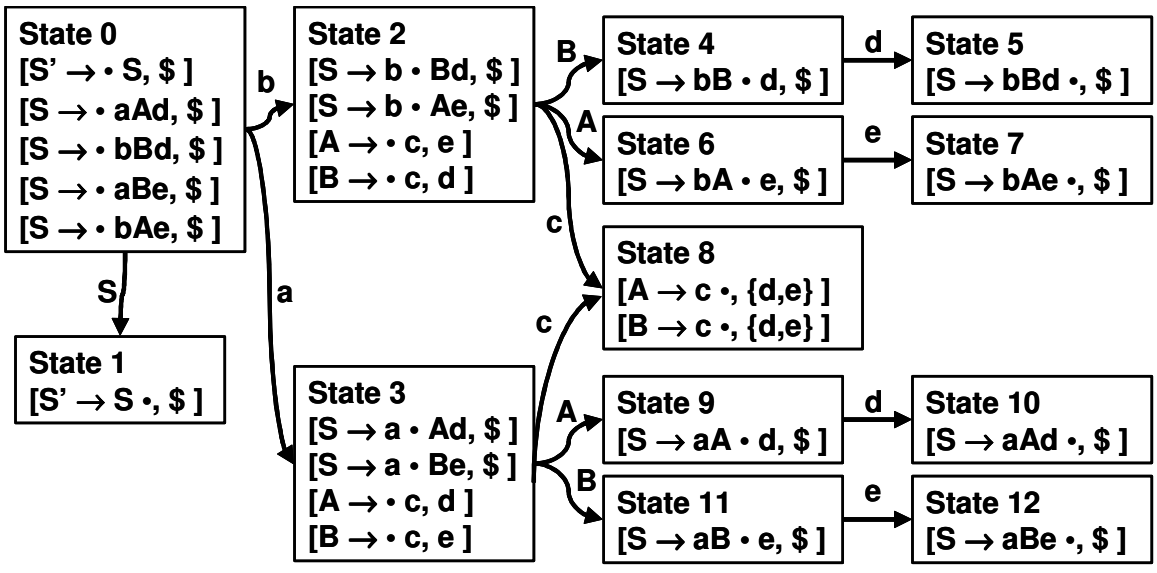
Canonical set of LR(1) items =



No conflicts in any state, grammar is LR(1)

LALR(1) parser?

States 8 & 13 merged since they have identical cores, yielding



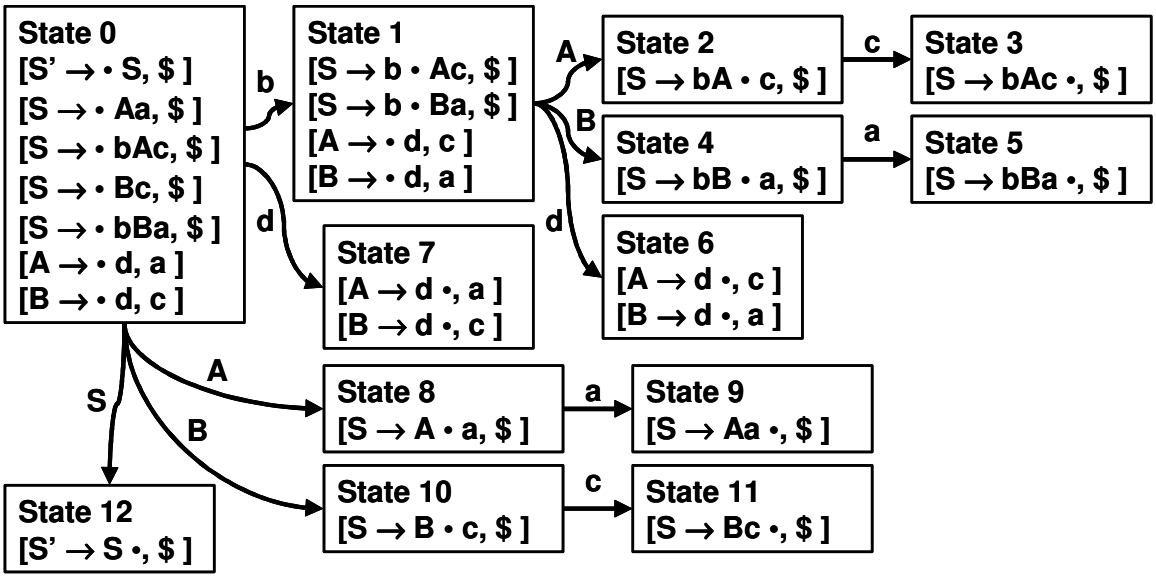
Not LALR(1), reduce/reduce error in State 8 on lookaheads d,e

14. Consider the following grammar: S → Aa | bAc | Bc | bBa A → d B → d

a. Show the grammar is LR(1) but not LALR(1)

LR(1) parser?

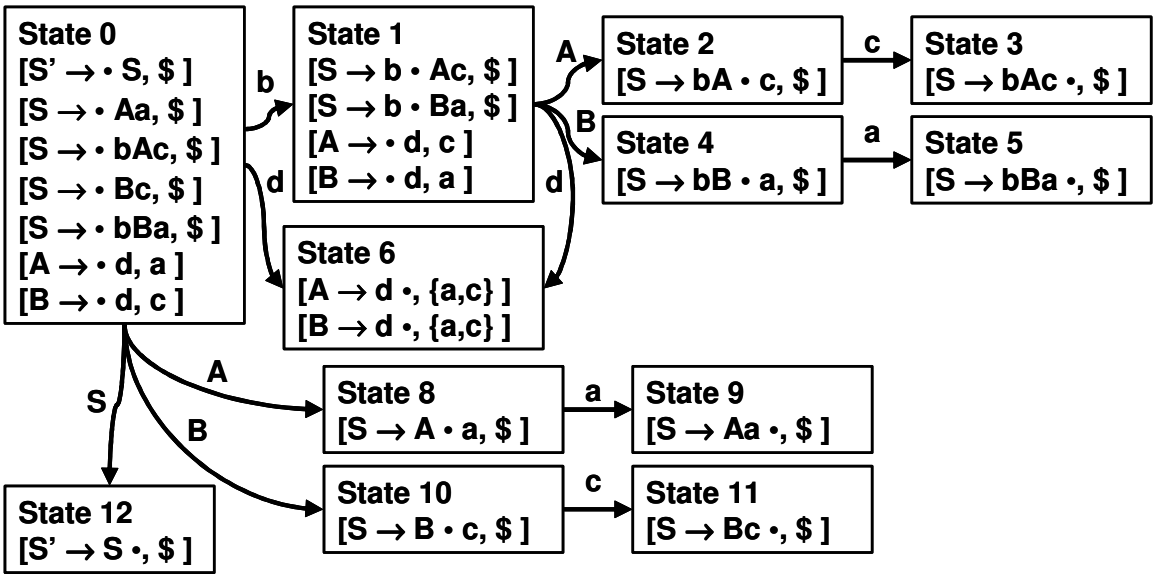
Canonical set of LR(1) items =



No conflicts in any state, grammar is LR(1)

LALR(1) parser?

States 6 & 7 merged since they have identical cores, yielding



Not LALR(1), reduce/reduce error in State 6 on lookaheads a,c