Written Set 2 Solutions

Problem 1: LL(1)

(i) The FIRST sets are as follows:

FIRST(S) = {**`public`**, **`final`**, **`class`**}
FIRST(C) = {**`public`**, **`final`**, **`class`**}
FIRST(P) = {ε, **`public`**}
FIRST(F) = {ε, **`final`**}
FIRST(X) = {ε, **`extends`**}
FIRST(Y) = {ε, **`implements`**}
FIRST(I) = {**`identifier`**}
FIRST(J) = {ε, "**`,`**"}

(ii) The FOLLOW sets are as follows:

FOLLOW(S) = {**`$`**}
FOLLOW(C) = {**`$`**}
FOLLOW(P) = {**`final`**, **`class`**}
FOLLOW(F) = {**`class`**}
FOLLOW(X) = {**`implements`**, **`$`**}
FOLLOW(Y) = {**`$`**}
FOLLOW(I) = {**`$`**}
FOLLOW(J) = {**`$`**}

(iii) The LL(1) parse table for the grammar is given below:

	`public`	`final`	`class`	`extends`	`implements`	`identifier`	`,`	`$`
S	1	1	1					
C	2	2	2					
P	3	4	4					
F		5	6					
X				7	8			8
Y					9			10
I						11		
J							12	13

Problem 2: LL(*k*)

(i) A simple grammar that is not LL(1) but is LL(2) is this one below:

A → ab | ac

This cannot be LL(1), because there is a FIRST/FIRST conflict with the nonterminal A: given lookahead **a**, we cannot tell whether to use the production A → ab or A → ac. However, the grammar is LL(2); given two tokens of lookahead, it is trivial to see which of the two productions we should use.

(ii) The proof is by contradiction. Assume for the sake of contradiction that we have a grammar that is not LL(2) but is LL(1). Then there must be an LL(2) conflict in the grammar, meaning that in some case, given the current nonterminal (call it A) and knowledge of the next two tokens (call them **a** and **b**), we cannot determine which production of A to use. In particular, there must be at least two productions A → *u* and A → *v* for some strings of terminals and nonterminals *u* and *v* that we cannot decide between. But this is impossible – since the grammar is LL(1), we can use our knowledge of **a** and A to uniquely determine which production to use. We have reached a contradiction, so our assumption was wrong and there cannot be a grammar that is LL(1) but not LL(2).

Problem 3: LL(0)

Claim: A grammar is LL(0) if and only if every nonterminal has exactly one production.

Proof: (if) If every nonterminal has exactly one production, it's trivial to construct an LL(0) parser for the grammar – just set the production to use when encountering each nonterminal to be the one production for that nonterminal.

(only if) By contradiction. If the grammar has more than one production for some nonterminal A, then upon seeing nonterminal A an LL(0) parser could not uniquely determine which production to use. In particular, suppose that the parser has matched w so far and now must choose which production of A to use. Suppose the options are the productions u and v. Then there are two cases to consider:

1. If there is a string x where $u \rightarrow^* x$ and $v \rightarrow^* x$, then the grammar is ambiguous. This means that given as input a string with two parse trees, the LL(0) parser would not have a unique production to choose at this point, which is impossible because, by definition, LL(k) grammars can be parsed by uniquely predicting which production to apply based on the current nonterminal and k tokens of lookahead.
2. Otherwise, the set of strings derivable from u and v must be disjoint. Pick some (unequal) strings x and y such that $u \rightarrow^* x$ and $v \rightarrow^* y$. Then both wu and wv must be prefixes of some valid strings in the grammar which are different from one another. The grammar is therefore not LL(0) because if we were to give these two strings into the parser, it could recognize at most one of them correctly.

We've reached a contradiction, so our assumption was wrong and if a grammar is LL(0), it must have one unique production for each nonterminal.

Problem 4: Left Factoring

(i) Most grammars that are left-factorable are not LL(1) because they result in FIRST/FIRST conflicts where we cannot determine which of the left-factorable productions to apply given one token of lookahead. As an example, the grammar

$$A \rightarrow ab \mid ac$$

is left-factorable but not LL(1), because given just the lookahead token **a** we can't tell which production to use.

(ii) A simple grammar with this property is

$$A \rightarrow X \mid Xa$$
$$X \rightarrow \varepsilon$$

This grammar is LL(1) because it can be parsed with the following LL(1) parse table:

	a	**$**
A	A → Xa	A → X
X	X → ε	X → ε

(iii) The grammar from part (i) (and also from question 2... this grammar seems to be a great example!) is left-factorable because both productions of A start with **a**, but is (as mentioned earlier) LL(2).

Problem 5: SLR(1) Parsing

(i) The LR(0) configurating sets for this grammar are as follows.

(1)	(2)	(3)	(4)	(5)	(6)
S → ·P P → ·(P)P P → ·	S → P·	P → (·P)P P → ·(P)P P → ·	P → (P·)P	P → (P)·P P → ·(P)P P → ·	P → (P)P·

(ii) The FOLLOW sets for the nonterminals are

FOLLOW(S) = {**$**}
FOLLOW(P) = {**)** , **$**}

(iii) The SLR(1) parsing table is as follows:

	(	**)**	**$**	P
(1)	s3	r3	r3	s2
(2)			accept	
(3)	s3	r3	r3	s4
(4)		s5		
(5)	s3	r3	r3	s6
(6)		r2	r2	

An important detail to note is that in state 1, even though there is no legal string that could end in **)**, we still must put a reduce action in on seeing a close parenthesis because that terminal is in FOLLOW(P). For this reason, SLR(1) parsers sometimes apply spurious reductions when encountering invalid input. There is a similar table entry for **$** in state 3.

(iv) The entry for **(** in state one would be a shift/reduce conflict in an LR(0) parser. Because of the item P → ·(P)P, we would try to shift into state 3, and because of the item P → ·, we would try to reduce production 3.

(v) No entry would contain a reduce/reduce conflict in the LR(0) parse table, since each LR(0) configurating set contains at most one reduce item.

Problem 6: The Power of LR Parsers

(i) By contradiction. Assume that some grammar is LR(0) but not SLR(1). This means that in the SLR(1) parsing table there would have to be a shift/reduce or reduce/reduce conflict. But since SLR(1) states correspond identically to LR(0) states (except for lookahead), this would mean that the corresponding LR(0) state either has two reduce items in it (if there's a reduce/reduce conflict) or both a shift and a reduce item (if there's a shift/reduce conflict). Either of these would make the grammar not LR(0), since we could not uniquely determine an action for each state (we would either be doing one of two reductions or both shifting and reducing). But this contradicts our initial assumption, so any LR(0) grammar is SLR(1).

(ii) By contradiction. Assume that some grammar is SLR(1) but not LALR(1). Then there must be a shift/reduce or reduce/reduce conflict in some LALR(1) state. LALR(1) states and SLR(1) states are identical (except for the lookaheads).

If the conflict is a shift/reduce conflict in some LALR(1) state on some terminal **t**, then consider the corresponding SLR(1) state. LALR(1) lookaheads are always a subset of SLR(1) lookaheads, so whatever reduce items were present for the reduce item for the LALR(1) parser are also present for the SLR(1) reduce item. Since LALR(1) and SLR(1) states are the same except for lookaheads, the shift item that shifts on **t** must also be in the SLR(1) state, and by the logic of the previous sentence there will be a reduce item here that wants to reduce on **t**, leading to a shift/reduce conflict in the SLR(1) parser.

If the conflict is a reduce/reduce conflict in some LALR(1) state on some terminal **t**, then in the corresponding SLR(1) state we will still have the same reduce items, both of which must still have **t** as a lookahead (since LALR(1) lookaheads are a subset of SLR(1) lookaheads) and we have the same reduce/reduce conflict in the SLR(1) parser.

In both cases, we have a conflict in the SLR(1) parser, which means that the grammar cannot be SLR(1), contradicting our assumption. Thus our assumption was wrong and any SLR(1) grammar is also LALR(1).

(iii) By contradiction. Assume that there is some grammar that is LALR(1) but not LR(1). LALR(1) states are constructed by merging LR(1) states with identical cores. Since our grammar is not LR(1), there must be a shift/reduce conflict or a reduce/reduce conflict somewhere.

If we have a reduce/reduce conflict on some lookahead **t**, then when we consider the LALR(1) state corresponding to the set of LR(1) states with the same core as the

deficient state, it must contain the same reduce items, both of which must have lookahead **t** because the original deficient LR(1) state has lookahead **t**. This causes a reduce/reduce conflict in the LALR(1) automaton.

If we have a shift/reduce conflict on some lookahead **t**, then when we consider the LALR(1) state corresponding to the set of LR(1) states with the same core as the deficient state, then it must have a shift item on **t** and the same reduce item containing lookahead **t**. This causes a shift/reduce conflict in the LALR(1) automaton.

In both cases we contradict that the grammar is LALR(1), so our assumption was wrong and any LALR(1) grammar is also LR(1).

Problem 7: Manual Conflict Resolution

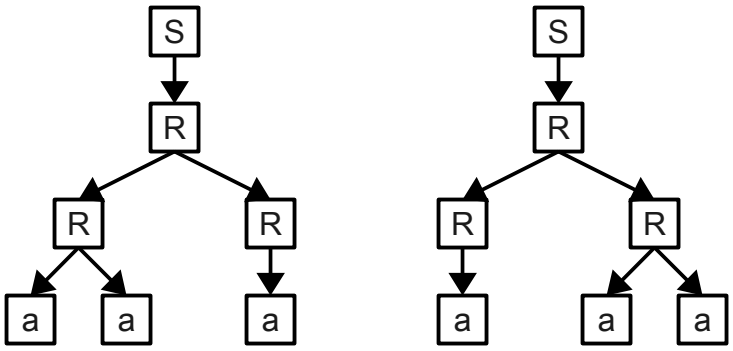
(i) First, we need to compute the FOLLOW sets for all of the nonterminals in the grammar. The result is shown here:

FOLLOW(S) = {**$**}
FOLLOW(R) = {**a**, **ε**, **(**, **)**, **|**, ***, **$**}

Given this, the resulting SLR(1) parsing table (with conflicts) is as follows

	a	**ε**	**(**	**)**	**|**	*****	**$**	R
(1)	s10	s9	s4					s2
(2)	s10	s9	s4		s6	s8	acc	s3
(3)	r2/s10	r2/s9	r2/s4	r2	r2/s6	r2/s8	r2	s3
(4)	s10	s9	s4					s5
(5)	s10	s9	s4	s11	s6	s8		s3
(6)	s10	s9	s4					s7
(7)	r3/s10	r3/s9	r3/s4	r3	r3/s6	r3/s8	r3	s3
(8)	r4	r4	r4	r4	r4	r4	r4	
(9)	r5	r5	r5	r5	r5	r5	r5	
(10)	r6	r6	r6	r6	r6	r6	r6	
(11)	r7	r7	r7	r7	r7	r7	r7	

(ii) No, upgrading this parser to LR(1) will not fix this problem. LR(1) only works on unambiguous grammars, but this grammar is ambiguous. For example, we can derive **aaa** in two ways:



(iii) In state (3), we are trying to reduce a concatenation, which is left-associative. We should therefore reduce anything that could start an "or" (since concatenation have higher precedence) or anything else that would be a concatenation (since concatenation is left-associative). However, we should shift the * symbol because star has higher precedence than concatenation.

In state (7), we are trying to reduce an "or," which is left-associative. Since "or" has lowest precedence, this means we should always shift when possible, unless we find another vertical bar signaling the start of another "or" (because "or" is left-associative).

The resulting table is shown on the next page.

	a	ε	(	)	|	*	$	R
(1)	s10	s9	s4					s2
(2)	s10	s9	s4		s6	s8	acc	s3
(3)	r2	r2	r2	r2	r2	s8	r2	s3
(4)	s10	s9	s4					s5
(5)	s10	s9	s4	s11	s6	s8		s3
(6)	s10	s9	s4					s7
(7)	s10	s9	s4	r3	r3	s8	r3	s3
(8)	r4	r4	r4	r4	r4	r4	r4	
(9)	r5	r5	r5	r5	r5	r5	r5	
(10)	r6	r6	r6	r6	r6	r6	r6	
(11)	r7	r7	r7	r7	r7	r7	r7	

Problem 8: LALR(1)-by-SLR(1)

(i) The FOLLOW set for Y contains **a** because of the production Y → bYa. Consequently, in state (3) we have a shift/reduce conflict, because on seeing an **a** we can't tell whether to shift it (from X → a·a) or to reduce it (because of Y → a·).

(ii) We augment the grammar by looking at every production of the form A → ·w for some string w and replacing the nonterminals in it by the appropriately augmented nonterminals. Here, this gives us

$S_1 \rightarrow X_{1\text{-}2}$
$X_{1\text{-}2} \rightarrow Y_{1\text{-}5}\ b$
$X_{1\text{-}2} \rightarrow aa$
$Y_{1\text{-}5} \rightarrow bY_{7\text{-}9}\ a$
$Y_{1\text{-}5} \rightarrow a$
$Y_{7\text{-}9} \rightarrow bY_{7\text{-}9}\ a$
$Y_{7\text{-}9} \rightarrow a$

(iii) The FOLLOW sets for these nonterminals are

FOLLOW(S_1) = {**$**}
FOLLOW($X_{1\text{-}2}$) = {**$**}
FOLLOW($Y_{1\text{-}5}$) = {**b**}
FOLLOW($Y_{7\text{-}9}$) = {**a**}

(iv) The updated lookahead sets are

LA(2, S → X·) = {**$**}
LA(3, Y → a·) = {**b**}
LA(4, X → aa·) = {**$**}
LA(6, X → Yb·) = {**$**}
LA(8, Y → a·) = {**a**}
LA(10, Y → bYa·) = {**a**, **b**}

(v) The only way this grammar could not be LALR(1) is if we have a shift/reduce conflict in state 3, since it's the only state containing a reduce item and any other item. However, the lookahead here for Y → a· is **b**, which does not overlap with the shift item **a** as before. The grammar is thus LALR(1).