Written Set 2 Solutions

Problem 1: LL(1)

(i) The FIRST sets are as follows:

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(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**	**,**	**$**
C	1	1	1					
P	2	3	3					
F		4	5					
X				6	7			7
Y					8			9
I						10		
J							11	12

Problem 2: LL(1) Conflicts

i. This grammar is not LL(1). Identify the conflicts in the grammar that make it not LL(1) and explain each.

The three productions for S all conflict with one another, because they all start (directly or indirectly) with the terminal symbol **noun**. This gives a three-way FIRST/FIRST conflict. The two productions for M also give a FIRST/FIRST conflict, because the grammar is left-recursive.

ii. Rewrite the grammar so that it is LL(1). To prove that your grammar is LL(1), construct an LL(1) parsing table for it. You do not need to explicitly show the FIRST or FOLLOW sets, thought it might be useful to compute them.

One way that we might rewrite the grammar is as follows:

S → **noun** Z
Z → ε | **and noun** | **,** **noun,** M
M → **and noun** | **noun,** M

	noun	**,**	**and**	**$**
S	**noun Z**			
Z		**, noun, M**	**and noun**	ε
M	**noun, M**		**and noun**	

Problem 3: LL(1) and LR(0)

i. Give an example of a grammar that is LL(1) but not LR(0), and explain why.

One simple observation is that LR(0) grammars cannot contain ε-productions, because they will force a shift/reduce conflict to occur. However, LL(1) grammars can handle this.

Consider the grammar A → ε | **x**. This is clearly LL(1) because it has this parsing table:

	x	**$**
A	**x**	ε

However, it is not LR(0). After augmenting the grammar to get S → A and A → ε | **x**, we get these configurating sets:

S → · A
A → ·
A → · **x**

S → A ·

A → **x** ·

Here, there is a shift/reduce conflict in the first state – do we shift **x** or reduce A → ε?

ii. Give an example of a grammar that is LR(0) but not LL(1), and explain why.

LL(1) cannot handle most left-factorable grammars. The grammar A → ab | ac is not LL(1), because we have a FIRST/FIRST conflict between the two productions. However, the grammar is LR(0), with these configurating sets:

S → · A
A → ·ab
A → ·ac

S → A ·

A → a · b
A → a · c

A → ab ·

A → ac ·

Problem 4: 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 $P \rightarrow \cdot(P)P$, we would try to shift into state 3, and because of the item $P \rightarrow \cdot$, 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.

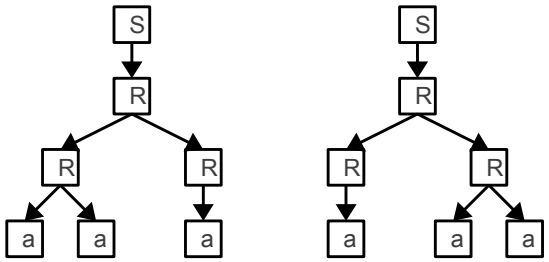
Problem 5: Manual Conflict Resolution

(i) 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 here:

	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	

(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:



Problem 6: 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).

(vi) Since the grammar is LALR(1), it is also LR(1).