第 7 章 LR 分析

第 1 题

已知文法

A→aAd|aAb| ε

判断该文法是否是 SLR(1)文法，若是构造相应分析表，并对输入串 ab#给出分析过程。

答案：

文法：

A→aAd|aAb| ε

拓广文法为 G′，增加产生式 S′→A

若产生式排序为：

0　S’→A

1　A →aAd

2　A →aAb

3　A → ε

由产生式知：

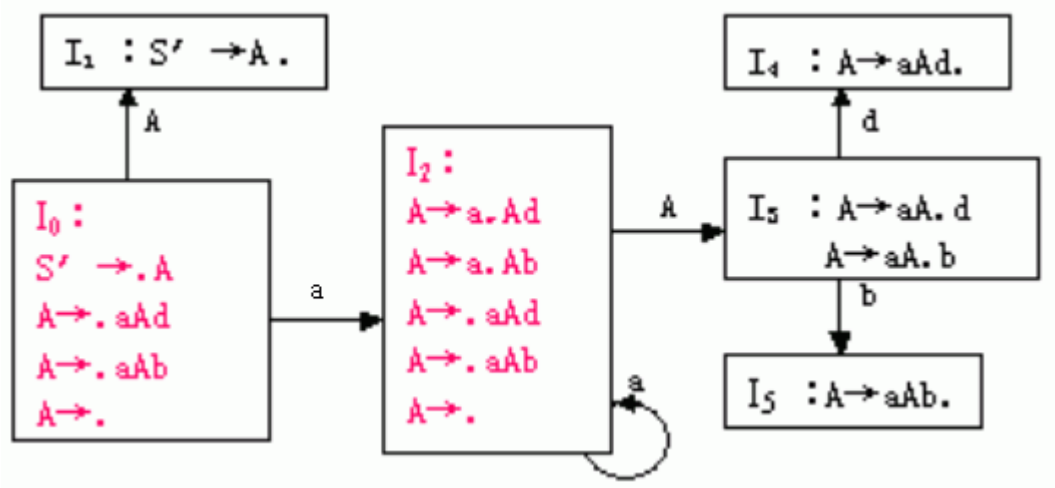
First (S’) = {ε,a}

First (A) = {ε,a}

Follow(S’) = {#}

Follow(A) = {d,b,#}

G′的 LR(0)项目集族及识别活前缀的 DFA 如下图所示：



在 I_0中：

A →.aAd 和 A →.aAb 为移进项目，A →.为归约项目，存在移进-归约冲突，因此所给文法不是 LR(0)文法。

在 I_0、I_2中：

Follow(A) ∩{a}= {d，b，#} ∩{a}=∅

所以在 I_0、I_2 中的移进-归约冲突可以由 Follow 集解决，所以 G 是 SLR(1)文法。
构造的 SLR(1)分析表如下：
题目 1 的 SLR(1)分析表

状态（State）	Action				Goto
	a	d	b	#	A
0	S2	r3	r3	r3	1
1				acc	
2	S2	r3	r3	r3	3
3		S4	S5		
4		r1	r1	r1	
5		r2	r2	r2	

题目 1 对输入串 ab#的分析过程

状态栈（state stack）	文法符号栈	剩余输入串（input left）	动作（action）
0	#	ab#....	Shift
0 2	#a	b#....	Reduce by :A → ε
0 2 3	#aA	b#....	Shift
0 2 3 5	#aAb	#....	Reduce by :A →aAb
0 1	#A	#....	

分析成功，说明输入串 ab 是文法的句子。

第 2 题

若有定义二进制数的文法如下：

S→L・L|L

L→LB|B

B→0|1

(1) 试为该文法构造 LR 分析表，并说明属哪类 LR 分析表。

(2) 给出输入串 101.110 的分析过程。

答案：

文法：

S→L.L|L

L→LB|B

B→0|1

拓广文法为 G′，增加产生式 S′→S

若产生式排序为：

0 S’→S

1 S →L.L

2 S →L

3 L →LB

4 L →B

5 B →0

6 B →1

由产生式知：

First (S’) = {0,1}

First (S) = {0,1}

First (L) = {0,1}

First (B) = {0,1}

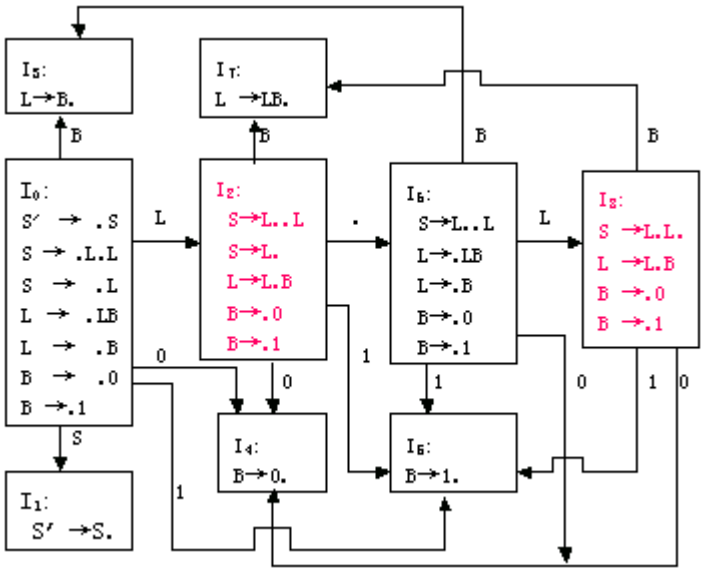
Follow(S’) = {#}

Follow(S) = {#}

Follow(L) = {.,0,1,#}

Follow(B) = {.,0,1,#}

G′ 的 LR(0)项目集族及识别活前缀的 DFA 如下图所示：



在 I_2 中：

B →.0 和 B →.1 为移进项目，S →L.为归约项目，存在移进-归约冲突，因此所给文法不是 LR(0)文法。

在 I_2、I_8 中：

Follow(s) ∩{0，1}= { #} ∩{0，1}=∅

所以在 I_2 、I_8 中的移进-归约冲突可以由 Follow 集解决，所以 G 是 SLR(1)文法。

构造的 SLR(1)分析表如下：

题目 2 的 SLR(1)分析表

状态（State）	Action				Goto		
	•	0	1	#	S	L	B
0		S4	S5		1	2	3
1				acc		.	
2	S6	S4	S5	r2			7
3	r4	r4	r4	r4		.	
4	r5	r5	r5	r5		.	
5	r6	r6	r6	r6		.	
6		S4	S5			8	3
7	r3	r3	r3	r3		.	
8		S4	S5	r1			7

题目 2 对输入串 101.110#的分析过程

状态栈（state stack）	文法符号栈	剩余输入串（input left）	动作（action）
0	#	101.110#....	Shift
0 5	#1	01.110#....	Reduce by :B →1
0 3	#B	01.110#....	Reduce by :S →LB
0 2	#L	01.110#....	Shift
0 2 4	#L0	1.110#....	Reduce by :B →0
0 2 7	#LB	1.110#....	Reduce by :S →LB
0 2	#L	1.110#....	Shift
0 2 5	#L1	.110#....	Reduce by :B →1
0 2 7	#LB	.110#....	Reduce by :S →LB
0 2	#L	.110#....	Shift
0 2 6	#L.	110#....	Shift
0 2 6 5	#L.1	10#....	Reduce by :B →1
0 2 6 3	#L.B	10#....	Reduce by :S →B
0 2 6 8	#L.L	10#....	Shift
0 2 6 8 5	#L.L1	0#....	Reduce by :B →1
0 2 6 8 7	#L.LB	0#....	Reduce by :S →LB
0 2 6 8	#L.L	0#....	Shift
0 2 6 8 4	#L.L0	#....	Reduce by :B →0
0 2 6 8 7	#L.LB	#....	Reduce by :S →L.L
0 1	#S	#....	

分析成功，说明输入串 101.110 是题目 2 文法的句子。

第 3 题

考虑文法 S → A S | b
A → S A | a

(1)构造文法的 LR(0)项目集规范族及相应的 DFA。
(2)如果把每一个 LR(0)项目看成一个状态，并从每一个形如 B → α·Xβ 的状态出发画一条标记为 X 的箭弧刀状态 B → αX·β，而且从每一个形如 B → α·Aβ 的状态出发画标记为 ε 的箭弧到所有形如 A → ·γ 的状态。这样就得到了一个 NFA。说明这个 NFA 与(a)中的 DFA 是等价的。
(3)构造文法的 SLR 分析表。
(4)对于输入串 bab，给出 SLR 分析器所作出的动作。
(5)构造文法的 LR(1)分析表和 LALR 分析表。

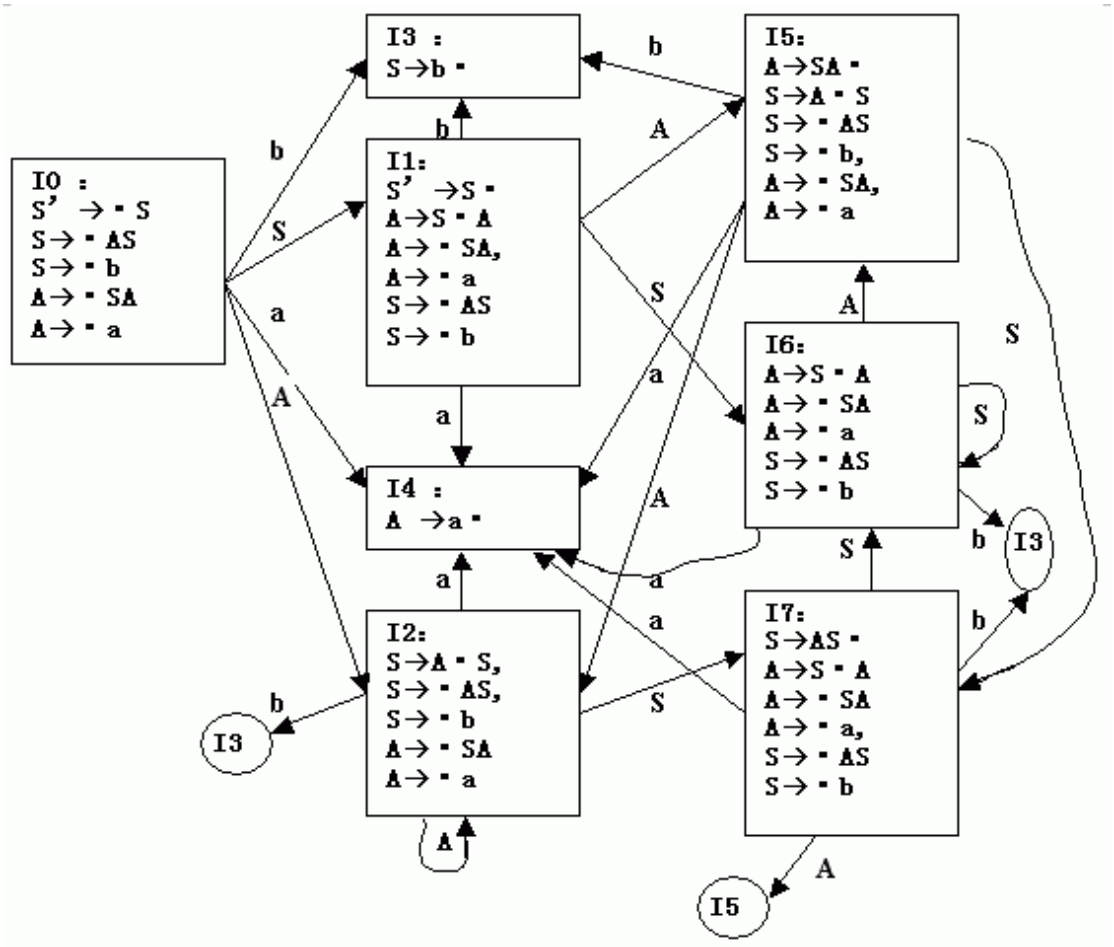
答案：

(1)令拓广文法 G’为
(0) S’ → S
(1)S → A S
(2)S → b
(3)A → S A
(4)A → a

其 LR(0)项目集规范族及识别该文法活前缀的 DFA 如下图所示：

FOLLOW(S)={#,a,b}
FOLLOW(A)={a,b}

LR(0)项目：



(2)显然，对所得的 NFA 求 ε 闭包，即得上面的 LR(0)项目集，即 DFA 中的状态。故此 NFA 与(a)中 DFA 是等价的。

(3)文法的 SLR 分析表如下：

状态	action			goto	
	a	b	#	S	A
0	S4	S3		1	2
1	S4	S3	acc	6	5
2	S4	S3		7	2
3	r2	r2	r2		
4	r4	r4			
5	S4/ r3	S3/r3		7	2
6	S4	S3		6	5
7	S4 / r1	S3 / r1	r1	6	5

因为 I5 中：FOLLOW（A）∩{a，b}≠Φ
I7 中：FOLLOW（S）∩{a，b}≠Φ
所以，该文法不是 SLR（1）文法。

或者：
从分析表中可看出存在歧义，所以不是该文法 SLR（1）文法。

注意：不是 SLR(1)文法就不能构造 SLR(1)分析表，也不能作分析过程。

(4)对于输入串 bab，SLR 分析器所作出的动作如下：

步骤	状态栈	符号栈	当前字符	剩余字符串	动作
(1)	0	#	b	ab#	移进
(2)	03	#b	a	b#	归约 S→b
(3)	01	#S	a	b#	移进
(4)	014	#Sa	b	#	归约 A→a
(5)	015	#SA	b	#	归约 A→SA
(6)	02	#A	b	#	移进
(7)	023	#Ab	#		归约 S→b
(8)	027	#AS	#		归约 S→AS
(9)	01	#S	#		接受

（在第 5 个动作产生歧义）

(5)LR(1)项目集族为：

I0 ：
S’ →·S, #
S →·AS, #
S →·b, #
S →·SA, a / b
A →·a, a / b

I1 ：S’ → S·,#
A → S·A, a / b
A →·a, a / b
A → ·SA, a / b
S →·AS, a / b
S →·b, a / b

I2 ：S →A·S, #
S →·b, #
S →·AS, #
A →·SA, a / b

A → • a, a / b

I3 : S → b • , #

I4 : A → a • , a / b

I5 : A → SA • , a / b
S → A • S, a / b
S → • AS, a / b
S → • b, a/ b
A → • SA, a / b
A → • a, a / b

I6 : A → S • A,a/b
A → • SA, a / b
A → • a, a / b
S → • AS, a / b
S → • b, a / b

I7: S → b • , a / b

I8 : S → AS • , #
A → S • A, a / b
A → • SA, a / b
A → • a, a / b
S → • AS, a / b
S → • b, a / b

I9 :
S →A • S, #
S → • AS, #
S → • b, #
S → • SA, a / b
A → • a, a / b

I10 :
S →AS • , a/b
A →S • A, a/b
A → • S A, a/b
A → • a, a / b
S → • b, a/b
S → • AS, a / b

I11 :

S →A·S, a/b
S →·b, a/b
S →·AS, a / b
A →·S A, a/b
A →·a, a / b

I12 :
S →SA·, a/b
S →A·S, a/b
S →·b, a/b
S →·AS, a / b
A →·S A, a/b
A →·a, a / b

∵I5 状态集中存在“归约－－移进”冲突，故无法构造 LR(1)分析表，因而也就无法构造 LALR 分析表。

注意：其实是可以构造的，这个题目出得不太严格。因为书上的定义是：根据这种文法构造的 LR（1）分析表不含多重定义时，称　这样的分析表为 LR（1）分析表，能用 LR（1）分析表的分析器称为 LR（1）分析器（规范的 LR 分析器），能构造的 LR（1）分析表的文法称为 LR（1）文法。

教材习题：

（1）列出这个文法的所有 LR(0)项目

（2）按（1）列出的项目构造识别这个文法活前缀的 NFA，把这个 NFA 确定化为 DFA，说明这个 DFA 的所有状态全体构成这个文法的 LR(0)规范族

（3）这个文法是 SLR 的吗？若是，构造出它的 SLR 分析表

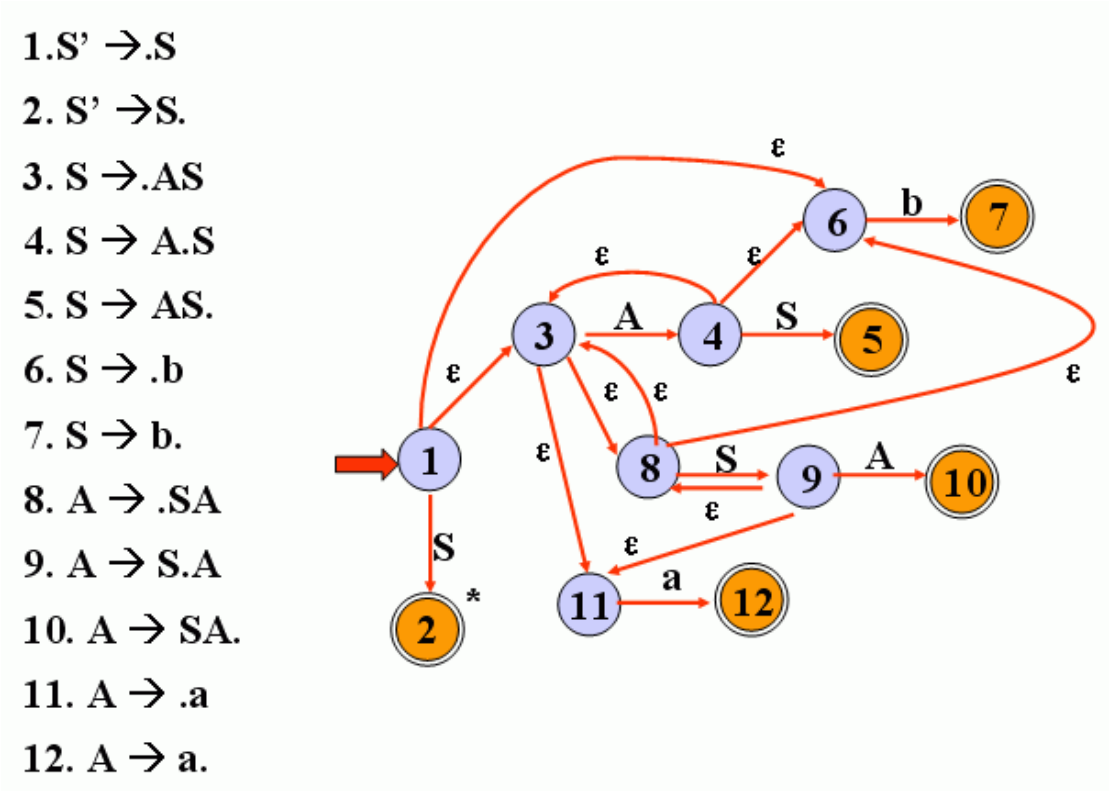
（4）这个文法是 LALR 或 LR(1)的吗？

答：

（1）令拓广文法 G’为
0　S’ → S
1　S → A S
2　S → b

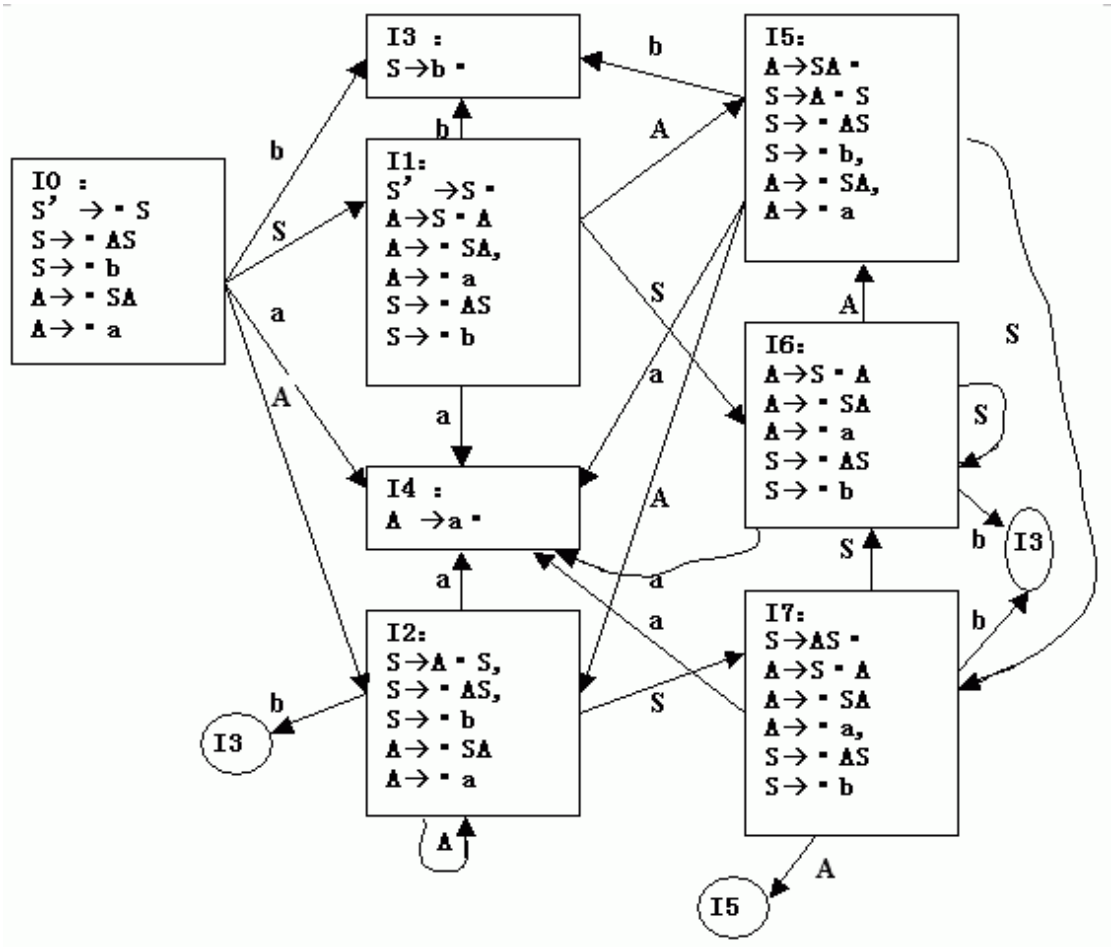
3 A → S A
4 A → a

其 LR(0)项目：



(2) 识别这个文法活前缀的 NFA 如上图所示：

确定化为 DFA 如下图所示：



（3）因为 I5 中：FOLLOW（A）∩{a，b}≠Φ
I7 中：FOLLOW（S）∩{a，b}≠Φ
所以，该文法不是 SLR（1）文法。

（4）LR(1)项目集族为：

I0 :
S' → · S, #
S → · AS, #
S → · b, #
S → · SA, a / b
A → · a, a / b

I1 : S' → S · ,#
A → S · A, a / b
A → · a, a / b
A → · SA, a / b
S → · AS, a / b
S → · b, a / b

I2 : S →A·S, #
S →·b, #
S →·AS, #
A →·SA, a / b
A →·a, a / b

I3 : S → b·, #

I4 : A → a·, a / b

I5 : A → SA·, a / b
S → A·S, a / b
S → ·AS, a / b
S →·b, a/ b
A →·SA, a / b
A →·a, a / b

I6 : A → S·A,a/b
A →·SA, a / b
A →·a, a / b
S →·AS, a / b
S →·b, a / b

I7: S → b·, a / b

I8 : S → AS·, #
A → S·A, a / b
A →·SA, a / b
A →·a, a / b
S →·AS, a / b
S →·b, a / b

I9 :
S →A·S, #
S →·AS, #
S →·b, #
S →·SA, a / b
A →·a, a / b

I10 :
S →AS·, a/b
A →S·A, a/b
A →·S A, a/b

A →・a, a / b
S →・b, a/b
S →・AS, a / b

I11 :
S →A・S, a/b
S →・b, a/b
S →・AS, a / b
A →・S A, a/b
A →・a, a / b

I12 :
S →SA・, a/b
S →A・S, a/b
S →・b, a/b
S →・AS, a / b
A →・S A, a/b
A →・a, a / b

因为 I5 状态集中存在“归约－－移进”冲突，所以不是 LR(1)文法，也不是 LALR 文法。

第 6 题

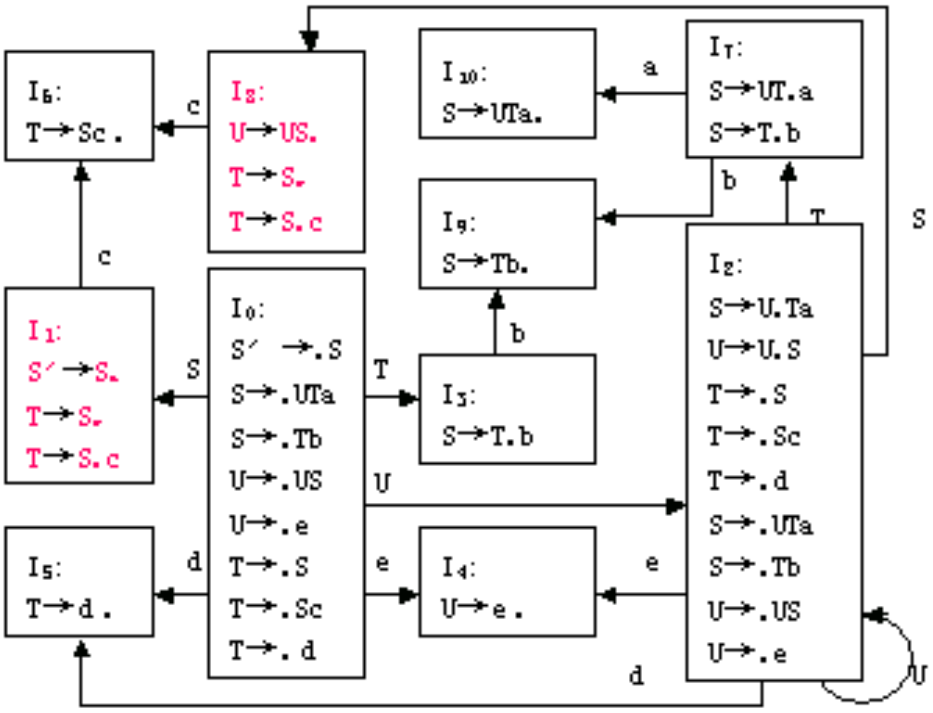
文法 G=({U,T,S},{a,b,c,d,e},P,S)
其中 P 为：
S→UTa|Tb
T→S|Sc|d
U→US|e
(1) 判断 G 是 LR(0)，SLR(1)，LALR(1)还是 LR(1)，说明理由。
(2) 构造相应的分析表。

答案：

文法：
S→UTa|Tb
T→S|Sc|d
U→US|e
拓广文法为 G'，增加产生式 S'→S
若产生式排序为：
0 S' →S
1 S →UTa
2 S →Tb
3 T →S
4 T →Sc
5 T →d
6 U →US
7 U →e
由产生式知：
First (S') = {d,e}
First (S) = {d,e}
First (U) = {e}
First (T) = {d,e}

Follow(S') = {#}
Follow(S) = {a,b,c,d,e,#}
Follow(U) = {d,e}
Follow(T) = {a,b}

G′ 的 LR(0)项目集族及识别活前缀的 DFA 如下图所示：



在 I_1 中：

S’ →S. 为接受项目，T →S. 为归约项目，T →S.c 为移进项目，存在接受-归约和移进-归约冲突，因此所给文法不是 LR(0)文法。

在 I1 中：

Follow(S’) ∩ Follow(T)= { #} ∩{a ， b}=∅

Follow(T) ∩{ c}= { a ， b} ∩{c}=∅

在 I_8 中：

Follow(U) ∩ Follow(T) ∩{ c}={d,e}∩{ a ， b} ∩{c}=∅

所以在 I_1 中的接受-归约和移进-归约冲突与 I_8 中的移进-归约和归约-归约冲突可以由 Follow 集解决，所以 G 是 SLR(1)文法。

构造的 SLR(1)分析表如下：

状态（State）	Action						Goto		
	a	b	c	d	e	#	S	U	T
0				S5	S4		1	2	3
1	r3	r3	S6			Acc			
2				S5	S4		8	2	7
3		S9							
4				r7	r7				
5	r5	r5							
6	r4	r4							
7	S10	S9							
8	r3	r3	S6	r6	r6				
9	r2	r2	r2	r2	r2	r2			
10	r1	r1	r1	r1	r1	r1			

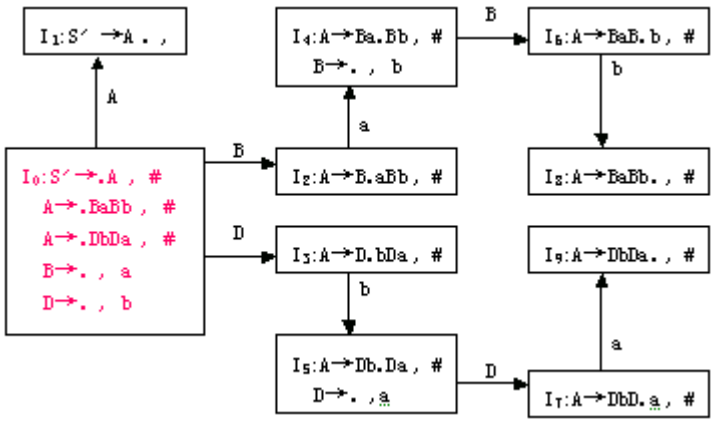
第 8 题

证明文法：
S→A$
A→BaBb|DbDa
B→ ε
D→ ε
是 LR(1)但不是 SLR(1)。(其中'$'相当于'#')

答案：

文法：
A→BaBb|DbDa
B→ ε
D→ ε
拓广文法为 G′，增加产生式 S′ →A
若产生式排序为：
0 S'→A
1 A →BaBb
2 A →DbDa
3 B → ε
4 D → ε
由产生式知：
First (S') = {a,b}

First (A) = {a,b}
First (B) = { ε }
First (D) = { ε }
Follow(S') = {#}
Follow(A) = {#}
Follow(B) = {a,b}
Follow(D) = {a,b}
G′ 的 LR(1)项目集族及识别活前缀的 DFA 如下图所示：



在 I_0 中：
B →. , a 和 T →. , b 为归约项目，但它们的搜索符不同，若当前输入符为 a 时用产生式 B → ε 归约，为 b 时用产生式 D → ε 归约，所以该文法是 LR(1)文法。
若不看搜索符，在 I_0 中项目 B →. 和 T →. 为归约-归约冲突，而
Follow(B) ∩Follow(D) = {a,b} ∩{a,b}≠∅，冲突不能用 Follow 集解决，所以该文法不是 SLR(1)文法。
构造的 LR(1)分析表如下：
题目 4 的 LR(1)分析表

State	Action				Goto		
	a	.	b	.	#	A	B	D
0	r3		r4.....			1	2	3
1					Acc		.	
2	S4.....						.	
3			S5				.	
4			r3				6	
5	r4.....							7
6			S8					
7	S9.....							
8					r1			
9					r2			

第 16 题

给定文法：

S→do S or S|do S|S;S|act

(1) 构造识别该文法活前缀的 DFA。

(2) 该文法是 LR(0)吗?是 SLR(1)吗?说明理由。

(3) 若对一些终结符的优先级以及算符的结合规则规定如下：

a) or 优先性大于 do;

b) ;服从左结合;

c) ;优先性大于 do ;

d) ;优先性大于 or ;

请构造该文法的 LR 分析表,并说明 LR(0)项目集中是否存在冲突和冲突如何解决的。

答案：

首先化简文法,用 d 代替 do;用 o 代替 or;用 a 代替 act;文法可写成：

S→dSoS|dS|S;S|a

拓广文法为 G’, 增加产生式 S′ →S

若产生式排序为：

0 S’ →S

1 S →dSoS

2 S →dS

3 S →S;S

4 S →a

由产生式知：

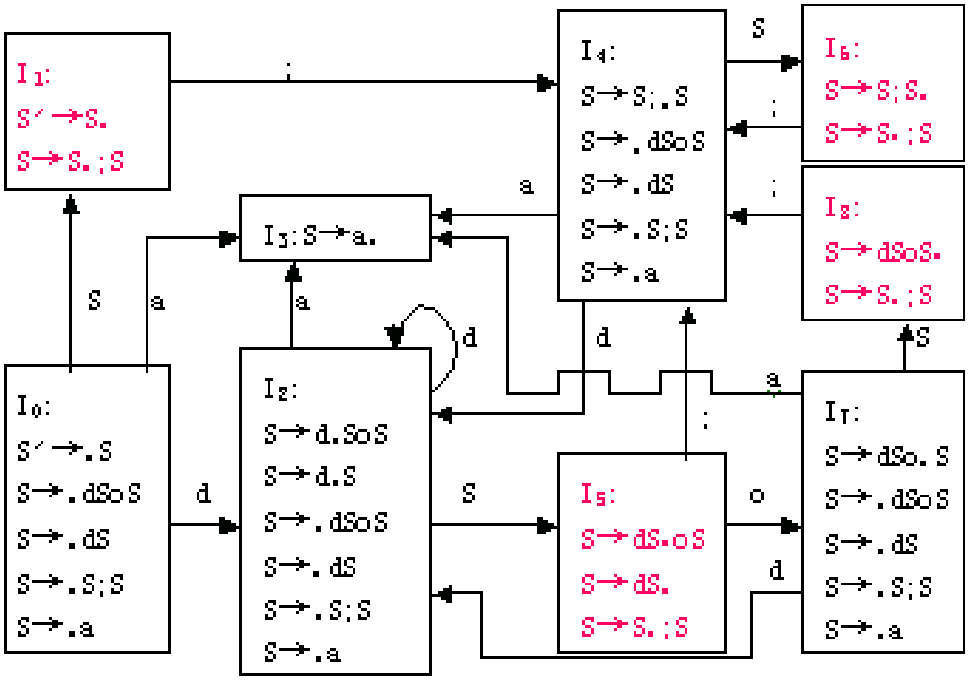
First (S’) = {d,a}

First (S) = {d,a}

Follow(S’) = {#}

Follow(S) = {o,;,#}

(1)识别该文法活前缀的 DFA 如下图。



(2) 该文法不是 LR(0)也不是 SLR(1)因为：
在 I_5、I_6、和 I_8中存在移进-归约冲突，因此所给文法不是 LR(0)文法。
又由于 Follow(S) = {o, ; ,#}
在 I_6、和 I_8中：
Follow(S) ∩{ ; }={o, ; ,#} ∩{ ; }={;}≠∅
在 I_5中：
Follow(S) ∩{ ; , o }={o , ; ,#} ∩{ ; , o }={; , o }≠∅
所以该文法也不是 SLR(1) 文法。

此外很容易证明所给文法是二义性的，

S`→S→dSoS→ddSoS

S`→S→dS→ddSoS

因此该文法不可能满足 LR 文法。

(3) 若对一些终结符的优先级以及算符的结合规则规定如下：
a) or 优先性大于 do;
b) ;服从左结合；
c) ;优先性大于 do ；
d) ;优先性大于 or ；

则：

在 I5 中： “or 和;优先性都大于 do”，所以遇输入符 o 和；移进；遇#号归约。

在 I6 中： “；号服从左结合”，所以遇输入符属于 Follow(S)的都应归约。

在 I8 中： “；号优先性大于 do ”， 所以遇输入符为;号 移进；遇 o 和#号归约。

此外，在 I_1 中：接受和移进可以不看成冲突，因为接受只有遇#号。

由以上分析，所有存在的移进-归约冲突可用规定的终结符优先级以及算符的结合规则解决，所构造的 LR 分析表如下：

状态（State）	Action					Goto
	d	o	;	a	#	S
0	S2			S3		1
1			S4		Acc	
2	S2			S3		5
3		r4	r4		r4	
4	S2			S3		6
5		S7	S4		r2	
6		r3	r3		r3	
7	S2			S3		8
8		r1	r4		r1	

附加题

问题 1：

试判别如下文法是否 LR(0)或 SLR(1)文法：

a) 文法 G[E]： E → E + T | T

T → (E) | id | id [E]

其中 E,T 为非终结符,其余符号为终结符

b) 文法 G[S]： S → Ab | ABc

A → aA | a

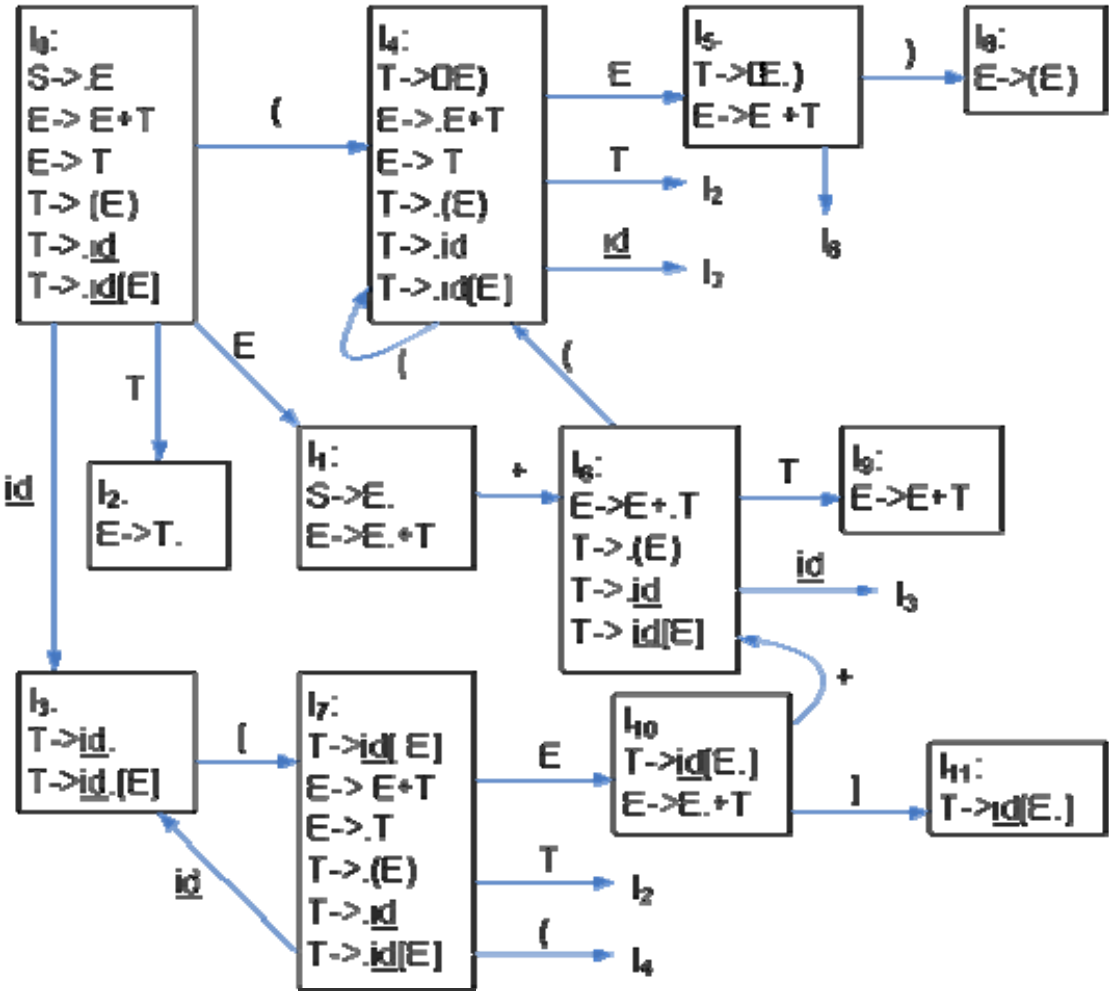
B → b

其中 S,A,B 为非终结符,其余符号为终结符

答案：

a) 增加产生式 S→E，得增广文法 G’[S]

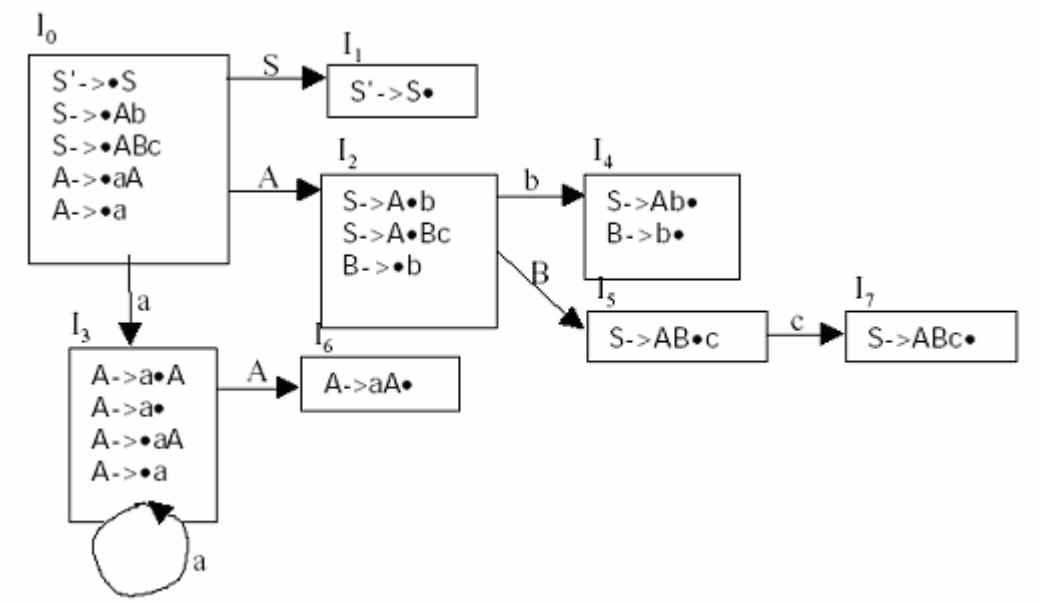
构造识别活前缀的有限状态机如下：



可验证：状态 I_3 有移进-归约冲突，所以 G[E]不是 LR(0)文法；进一步，因 Follow(T)={+,},},#}，不含[，所以 G[E]是 SLR(1)文法。

b) 增加产生式 S'→S，得增广文法 G'[S']

构造识别活前缀的有限状态机如下：



I4 存在归约/归约冲突， I3存在归约/移进冲突.因此不是LR(0)文法。
考察能否使用SLR(1)方法解决冲突：I4 中，因为 Follow(S) = {#} 而 Follow(B) = {c}. 所以可以解决。 I3中，因Follow(A) = {b}，不含 a，因此该移进/归约冲突也可解决. 文法是SLR(1)文法

参考解答：

为下列文法构造 LR（1）分析表，并给出串 baab# 的分析过程:

增广文法文法 G[S']:

(0) S' → S
(1) S → XX
(2) X → aX
(3) X → b

其中 S',S,X 为非终结符,其余符号为终结符

参考解答：

参见教材 P146 的例子, 自行补充对输入串 baab#的分析过程

问题 2：

（1）构造下列文法 G（P’）的 LR（1）FSM，验证它是 LR（1）文法：

(0) P’→ P

(1) P → P(P)

(2) P → Aa

(3) P → ε

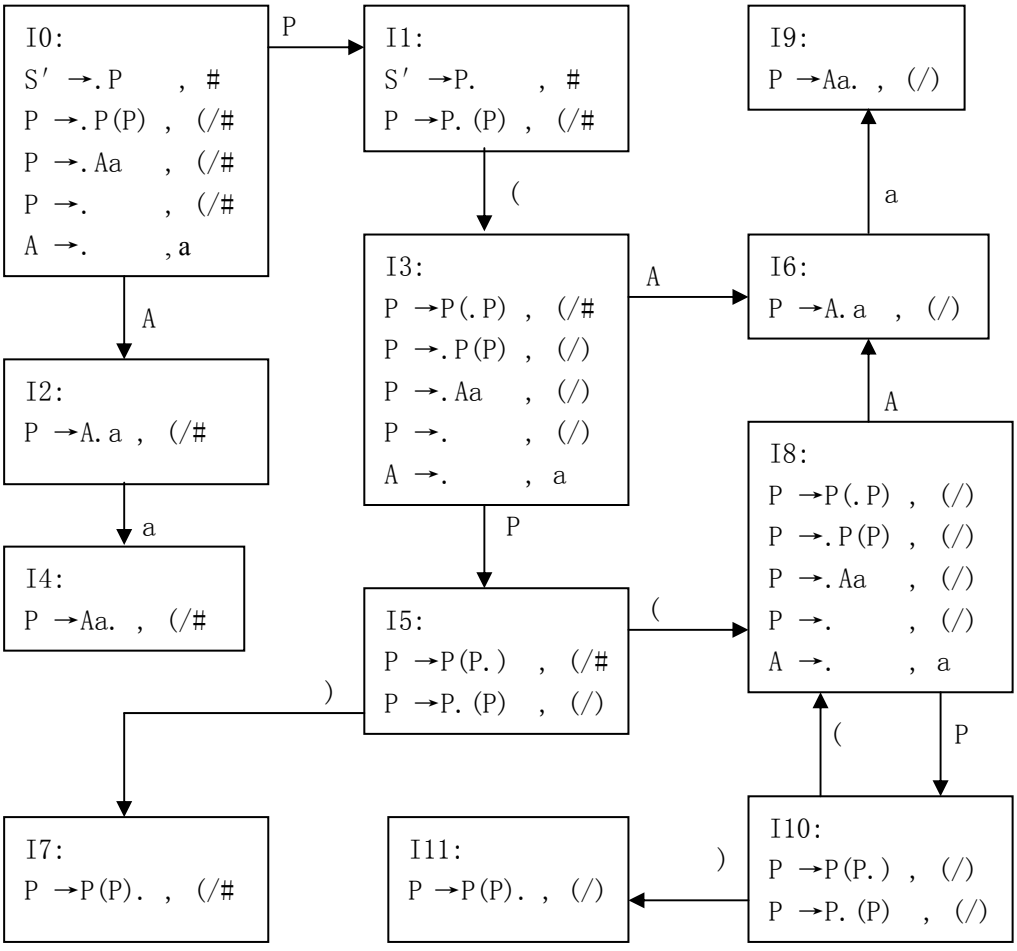
(4) A → ε

其中 P’,P,A 为非终结符

（2）通过合并同芯集（状态）的方法构造相应于上述 LR（1）FSM 的 LALR（1）FSM，并判断 G（P’）是否 LALR（1）文法？

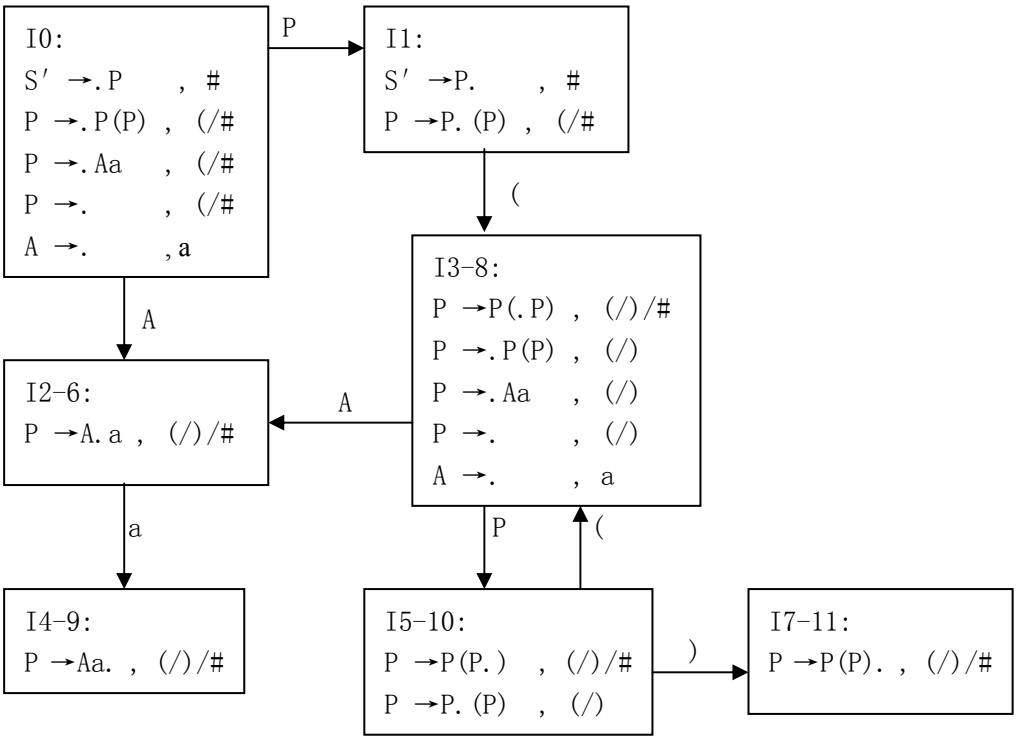
答案：

（1） 构造文法 G（P’）的 LR（1）项目集族和转换函数如下：



其中的每个状态均无宏冲突，所以 G（P’）是 LR（1）文法。

（2） 可以看出：I2 和 I6，I3 和 I8，I4 和 I9，I5 和 I10，I7 和 I11 是同芯状态. 通过合并同芯状态的方法构造相应于上述 LR（1）转换图的 LALR（1）转换图 如下：



可以看出：所有状态都不存在移进-归约和归约-归约冲突.所以，*G*（*P*’）是 LALR（1）文法。

问题 3：

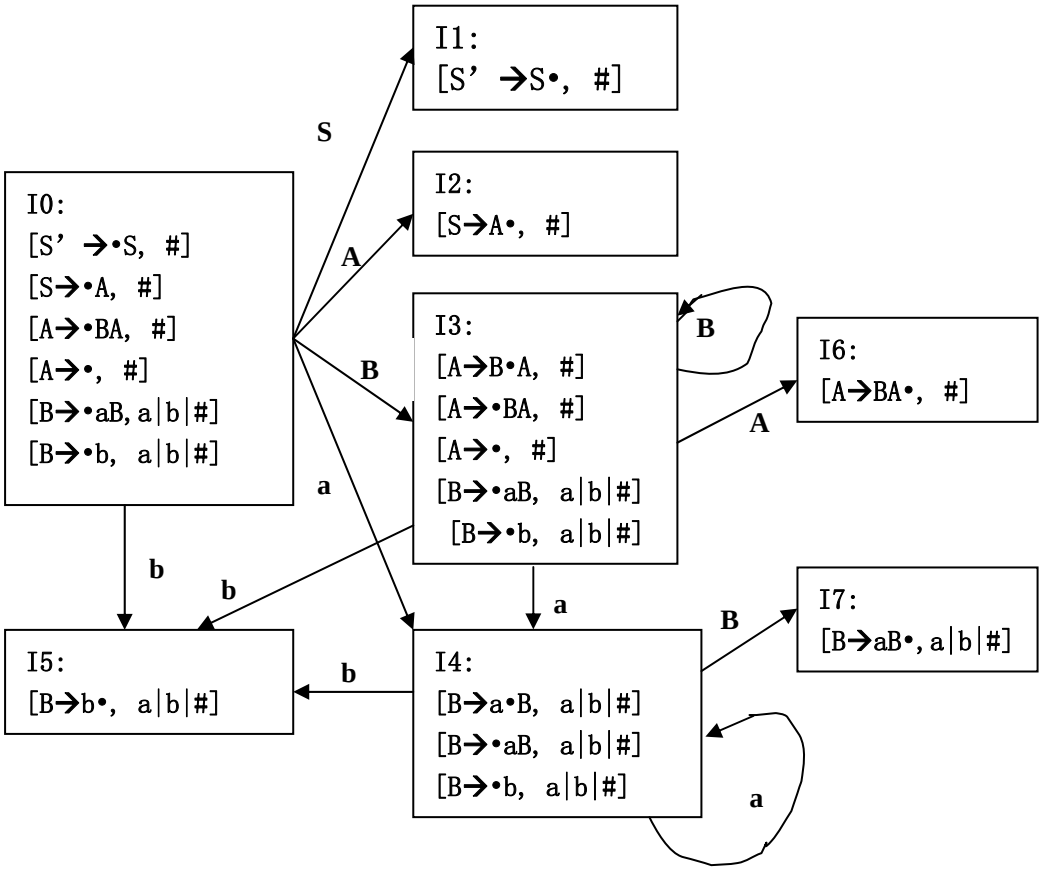
设文法 G 为：
S → A
A → BA | ε
B → aB | b
(1)证明它是 LR(1)文法；
(2)构造它的 LR(1)分析表；
(3)给出输入符号串 abab 的分析过程。

答案：

(1)拓广文法 G’：
(0) S’ → S　　(1) S → A　　(2) A → BA
(3) A → ε　　(4) B → aB　　(5) B → b
FIRST(A) = { ε , a, b}

FIRST(B) = {a, b}

构造的 DFA 如下:



由项目集规范族看出，不存在冲突动作。
∴该文法是 LR(1)文法。

(2)

LR(1)分析表如下：

状态	Action			Goto		
	a	B	#	S	A	B
0	S4	S5	r3	1	2	3
1			acc			
2			r1			
3	S4	S5	r3		6	3
4	S4	S5				7
5	r5	r5	r5			

6			r2			
7	r4	r4	r4			

(3)输入串 abab 的分析过程为：

步骤	状态栈	符号栈	当前字符	剩余字符串	动作
(1)	0	#	a	bab#	移进
(2)	04	#a	b	ab#	移进
(3)	045	#ab	a	b#	归约 B→b
(4)	047	#aB	a	b#	归约 B→aB
(5)	03	#B	a	b#	移进
(6)	034	#Ba	b	#	移进
(7)	0345	#Bab	#		归约 B→b
(8)	0347	#BaB	#		归约 B→aB
(9)	033	#BB	#		归约 A→ ε
(10)	0336	#BBA	#		归约 A→BA
(11)	036	#BA	#		归约 A→BA
(12)	02	# A	#		归约 S→A
(13)	01	#S	#		acc

问题4：

给定文法G'(S')：

S' → S
S → (L) | a
L → L , S| S

试为该文法配上属性计算的语义规则（或动作）集合（即设计一个属性文法），它输出配对括号的个数。如对于句子(a,(a))，输出是2。

答案：

产生式	语义规则
S′ →S	{print (S.num)}
S→(L)	{S.num:=L.num+1}
S→a	{S.num:=0}
L→L_1,S	{L.num:= L_1.num+S.num}
L→S	{L.num:=S.num}

问题5：

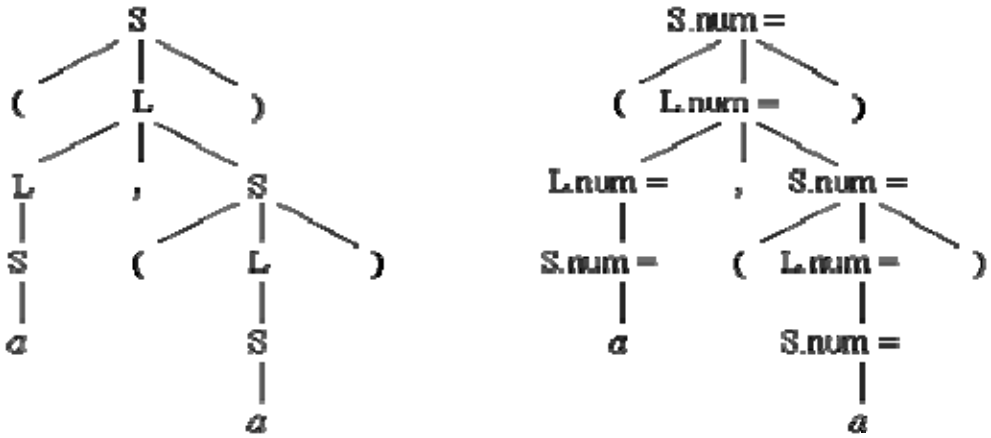
给定文法G[S]：

S → (L) | a
L → L , S | S

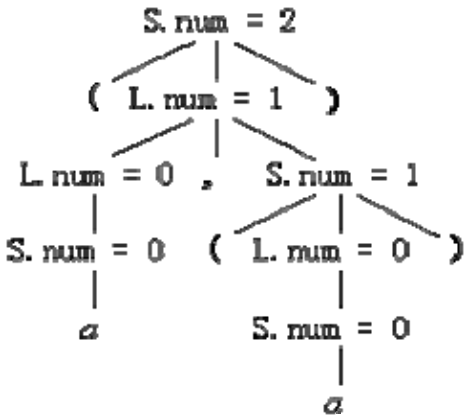
如下是相应于G[S]的一个属性文法：

(1) S → (L) { S.num := L.num +1; }
(2) S → a { S.num := 0; }
(3) L → L_1 , S { L.num := L_1.num + S.num; }
(4) L → S { L.num := S.num; }

下图分别是输入串（a,（a））的语法分析树和对应的带标注语法树，但其属性值没有标出，试将其标出（即填写右下图中符号 “=” 右边的值）。



答案：



问题 6：

对上题中所给的 G[S]的属性文法是一个 S-属性文法，故可以在自下而上分析的过程中，增加一个语义栈来计算属性值。下图（a）是 G[S]的一个 LR 分析表，图(b) 描述了输入串（ *a*，（ *a* ））的分析和计值过程（语义栈中的值对应 S.num 或 L.num ），其中，第 14)，15）行没有给出，试补齐之。

状态	ACTION					GOTO	
	a	,	(	)	#	S	L
0	s_3		s_2			1	
1					acc		
2	s_3		s_2			5	4
3		r_2		r_2	r_2		
4		s_7		s_6			
5		r_4		r_4			
6		r_1		r_1	r_1		
7	s_3		s_2			8	
8		r_3		r_3			

题 3 图（a）

步骤	状态栈	语义栈	符号栈	余留符号串
1)	0	-	#	(a,(a))#
2)	02	- -	#(	a,(a))#
3)	023	- - -	#(a	,(a))#
4)	025	- - 0	#(S	,(a))#
5)	024	- - 0	#(L	,(a))#
6)	0247	- - 0 -	#(L,	(a))#
7)	02472	- - 0 - -	#(L,(	a))#
8)	024723	- - 0 - - -	#(L,(a	))#
9)	024725	- - 0 - - 0	#(L,(S	))#
10)	024724	- - 0 - - 0	#(L,(L	))#
11)	0247246	- - 0 - - 0 -	#(L,(L)	)#
12)	02478	- - 0 - 1	#(L,S	)#
13)	024	- - 1	#(L	)#
14)				
15)				
16)	接受			

题 3 图（b）

答案：

14）	0246	- - 1 -	# (L)	#
15）	01	- 2	# S	#

问题 7：

下面的属性文法 G[N]可以将一个二进制小数转换为十进制小数，令 N.val 为 G[N]生成的二进制数的值，例如对输入串 101.101，N.val=5.625.

N → S^1 '•' S^2 { N.val := S^1.val + $2^{-S^2.len}$ *S^2 .val ; }

S → S^1 B { S.val := 2 * S^1.val + B.val; S.len := S^1.len + 1 }

S → B { S.val = B.val; S.len:= 1 }

B → '0' {B.val :=0}

B → ‘1’ {B.val:= 1}

（1） 试消除该属性文法（翻译模式）中的左递归，以便可以得到一个可以进行自上而下进行语义处理（翻译）的翻译模式；

（2） 对变换后的翻译模式，构造一个自上而下预测翻译程序。

答案：

（1）消去原文法中的左递归：

N → S ‘•’ S
S → BR
R → BR
R → ε
B → ‘0’
B → ‘1’

可验证该文法是 LL（1）文法。

再考虑语义规则，得到如下翻译模式：

N → S^1 ‘•’ S^2 { N.val := S^1.val + $2^{-S^2.len}$ *S^2 .val ; }

S → B {R.ival := B.val; R.ilen := 1} R { S.val := R.sval; S.len := R.slen }

R → B {R^1.ival:=2*R.ival+B.val; R^1.ilen:=R.ilen+1} R^1 {R.sval:=R^1.sval; R.slen:=R^1.slen}

R → ε {R.sval:=R.ival; R.slen:=R.ilen}

B → ‘0’ {B.val :=0}

B → ‘1’ {B.val :=1}

(2) 对变换后的翻译模式，可构造一个如下的自上而下预测翻译程序：

```
real ParseN()
{
    (S1val,S1len) := ParseS();     //变量 S1val,S1len 分别对应属性 S1.val, S1.len
    MatchToken( ‘•’ );             //匹配 ‘•’ , 函数 MatchToken 参见第 4 讲，本例省略
    (S2val,S2len) := ParseS();     //变量 S2val,S2len 分别对应属性 S2.val, S2.len
    Nval := S1val + 2 ^ (-S2len) * S2val ; /变量 Nval 对应属性 N.val
    return Nval;
}
(int, int) ParseS()                //（int, int）代表一个记录/结构类型
{
```

```
    Bval := ParseB();
    Rival := Bval;
    Rilen := 1;
    (Rsval, Rslen) := ParseR(Rival, Rilen);
    Sval := Rsval;
    Slen := Rslen ;
    return (Sval,Slen);
}
(int, int) ParseR(int Rival, int Rilen)
{
    switch (lookahead)  {              // lookahead 为下一个输入符号
         case ' 0' , ' 1' :
              Bval := ParseB();
              R1ival := 2*Rival+Bval;
              R1ilen := Rilen+1;
              (R1sval, R1slen) := ParseR(R1ival, R1ilen);
              Rsval := R1sval;
              Rslen := R1slen ;
             break;
         case ' •' , ' #' :
             Rsval:=Rival;
             Rslen:=Rilen;
             break;
         default:
             printf("syntax error \n")
             exit(0);
    }
    return (Rsval,Rslen);
}

int ParseB()
{
    switch (lookahead)  {
         case ' 0' :
             MatchToken( '0' );
             Bval := 0;
             break;
         case ' 1' :
             MatchToken( '1' );
             Bval := 1;
             break;
         default:
             printf("syntax error \n")
             exit(0);
```

```
        }
        return Bval;
    }
```

问题 8：

对于上题（1）所得到的翻译模式（结果应满足 L-属性的条件），在进行自下而上的语义处理时，语义栈中的值有两个分量，分别对应文法符号的综合属性 val 和 len。

（1） 若该翻译模式中，嵌在产生式中间的语义规则集中含有除复写规则之外的语义规则，则变换该翻译模式，使嵌在产生式中间的语义规则集中仅含复写规则；

（2） 根据（1）所得到的新翻译模式，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的综合属性进行访问。试写出在按每个产生式归约时语义处理的代码片断（设语义栈由向量 v 表示，归约前栈顶位置为 top，语义值 v[i]的两个分量分别用 v[i].val 和 v[i].len 表示）。

答案：

(1) 将如下翻译模式

N → S^1 ‘•’ S^2 { N.val := S^1.val + $2^{-S^2.len}$ *S^2 .val ; }

S → B {R.ival := B.val; R.ilen := 1} R { S.val := R.sval; S.len := R.slen }

R → B {R^1.ival:=2*R.ival+B.val; R^1.ilen:=R.ilen+1} R^1 {R.sval:=R^1.sval; R.slen:=R^1.slen}

R → ε {R.sval:=R.ival; R.slen:=R.ilen}

B → ‘0’ {B.val :=0}

B → ‘1’ {B.val :=1}

变换为：

N → S^1 ‘•’ S^2 { N.val := S^1.val + $2^{-S^2.len}$ *S^2 .val ; }

S → B { M.bval := B.val } M {R.ival := M.sval; R.ilen := M.val} R { S.val := R.sval; S.len := R.slen }

R → B {P.rival:= R.ival; P.rilen:= R.ilen; P.bval:=B.val; } P {R^1.ival:= P.sval; R^1.ilen:= P.slen } R^1 {R.sval:=R^1.sval; R.slen:=R^1.slen}

R → ε {R.sval:=R.ival; R.slen:=R.ilen}

B → ‘0’ {B.val :=0}

B → ‘1’ {B.val :=1}

M → ε { M.sval := M.bval; M.val := 1 }

P → ε {P.sval = 2*P.rival + P.bval; P.sleP:= P.rilen +1}

(2) 按每个产生式归约时语义处理的代码片断：

```
N → S¹ '•' S²  {v[top-2].val := v[top-2].val + 2^(-v[top].len) * v[top].val; }
S → B M R      {v[top-2].val := v[top].val; v[top-2].len := v[top].len }
R → B P R¹     {v[top-2].val := v[top].val; v[top-2].len := v[top].len }
R → ε          {v[top+1].val := v[top].val; v[top+1].len := v[top].len }
```

```
B → '0'        {v[top].val :=0}
B → '1'        {v[top].val :=1}
M → ε        { v[top+1].val := v[top].val; v[top+1].len := 1}
P → ε         { v[top+1].val := 2*v[top-1].val+v[top].val; v[top+1].len :=
v[top-1].len+1}
```

问题 9：

证明 LR 分析过程正确性的一个重要引理：由构造 LR(0)项目集规范族得到的 DFA，它可以也只能读进所分析文法的活前缀。

需要证明两个方面：

命题 1　所有活前缀一定都可由 DFA 读进，即不会错过合法的归约。

命题 2　DFA 只能读活前缀。

证明思路：

首先要了解活前缀是如何产生的。活前缀的集合 *Prefix* 可归纳定义如下：

（1） 设 S'是增广文法的开始符号，既有产生式 $S' \to S$（S 是原文法的开始符号），令 $S \in Prefix$。

（2） 若 $v \in Prefix$，则 v 的任一前缀 u 都是活前缀，即 $u \in Prefix$。

（3） 若 $v \in Prefix$，且 v 中至少包含一个非终结符，即可以将 v 写成 $\alpha B\gamma$，其中 B 为非终结符。若有产生式 $B \to \beta$，则$\alpha\beta$ 的任一前缀 u 都是活前缀，即 $u \in Prefix$。

（4） *Prefix* 中的元素只能通过上述步骤产生。

命题 1 可以根据 *Prefix* 的定义进行归纳证明。

基础：对于由规则（1）产生的活前缀 $S \in Prefix$，由于在 DFA 的初始项目集（状态）I_0 中含有项目 $S' \to \cdot S$，显然 S 是可以被 DFA 读进的。

归纳：若 $v \in Prefix$，且 v 可以被 DFA 读进，则 v 的前缀可以被 DFA 读进也是显然的。

若 $v \in Prefix$，且可以将 v 写成 $\alpha B\gamma$，其中 B 为非终结符并有产生式 $B \to \beta$。设 DFA 在从初态 I_0 读进 α 后进入状态 I。因为 $v=\alpha B\gamma$ 可以被 DFA 读进，所以在状态 I 可以读进 B，根据 DFA 的构造过程，一定存在项目 $C \to \alpha \cdot B\gamma' \in I$。由闭包的计算过程，可知一定有 $B \to \cdot\beta \in I$，因此 β 是从 I 开始后续的可读串。所以，从初态 I_0 开始，$\alpha\beta$ 的任一前缀 u 都是可读进的。

命题 1 证毕。

命题 2 可归纳于 DFA 可读序列的长度 n 来证明。

基础：n=0。显然空序列 ε 是活前缀。

归纳：设 αX 是 DFA 可读序列，且 $|\alpha X| = n+1$，其中 X 是某个文法符号。在读过 α 后，DFA 一定处于状态 I，在此状态下 X 是可读的。根据 DFA 的构造过程，一定存在项目 $C \to \beta \cdot X\gamma \in I$。该项目或者是基本项目，或者是通过闭包计算得到的项目。下面分这两种情形来讨论。

1） $C \to \beta \cdot X\gamma$ 是基本项目。

若 $|\beta| = 0$，则该项目只能是 $S' \to \cdot S$，此时 $\alpha X = S$ 显然是活前缀。

若 $|\beta| = m \neq 0$，则 DFA 从状态 I_0 读进 $n-m$ 个符号的序列 μ 后进入状态 I'，且必定有 $C \to \cdot \beta X\gamma \in I'$。根据 DFA 的构造过程，一定存在项目 $B \to \cdot C\gamma' \in I'$。这样在状态 I'下，可以读进 B。因为 $|\mu B| \leq n$，由归纳假设，可知 μB 是活前缀。因此，由活前缀集合的归纳定义，得知 $\mu\beta X = \alpha X$ 是活前缀。

2） $C \to \beta \cdot X\gamma$ 是通过闭包计算得到的项目。

此时，β 一定为空序列，而项目 $C \to \cdot X\gamma$ 一定是由 I 中的某个基本项目 $B \to \mu \cdot C_0\nu$ 直接或间接地通过闭包计算序列得到的：$C_0 \to \cdot C_1\gamma_1$，$C_1 \to \cdot C_2\gamma_2$，…，$C_k \to \cdot C\gamma_{k+1}$。由 1）的讨论结果，可知 αC_0 是活前缀。从而 αC_1，αC_2，…，αC_k 都是或前缀，αC 也是活前缀。所以，αX 是活前缀。

命题 2 证毕。