

《编译原理》书面作业

截止日期：2025 年 11 月 11 日

若发现问题，或有任何想法（改进题目、调整任务量等等），请及时联系助教

Q1. 下面的文法 $G[S]$ 描述由命题变量 p, q ，联结词 $\wedge$ （合取）、 $\vee$ （析取）、 $\neg$ （否定）构成的命题公式集合：

$$S \rightarrow S \vee T \mid T$$

$$T \rightarrow T \wedge F \mid F$$

$$F \rightarrow \neg F \mid p \mid q$$

试分别指出以下句型的所有短语和直接短语。如果这些句型同时也是右句型，那么还要给出其句柄。

1. $\neg F \vee \neg q \wedge p$

2. $\neg F \vee p \wedge \neg F \vee T$

参考答案：

1. 短语： $\neg F \vee \neg q \wedge p, \neg F, \neg q \wedge p, \neg q, q, p$ ；

直接短语： $\neg F, p, q$ ；

句柄： $\neg F$ 。

2. 短语： $\neg F \vee p \wedge \neg F \vee T, \neg F \vee p \wedge \neg F, \neg F$ (第一个), $p \wedge \neg F, p, \neg F$ (第二个)；

直接短语： $\neg F$ (第一个), $p, \neg F$ (第二个)。(注意不同位置的短语不能合并)

Q2. 小明正在设计一款简单的编程语言。

1. 首先，小明要引入常表达式 a 和赋值语句 $a=b$ ，赋值语句的左侧 (L) 和右侧 (R) 都是一个标识符（终结符 i ），文法可以写作 $G_1[S]$ ：

$$S \rightarrow L = R \mid R$$

$$L \rightarrow i$$

$$R \rightarrow i$$

请问该文法是否是 LR(0) 文法？结合 LR(0) 自动机说明原因。（思考：如果不构造自动机，你能否直接判断？）

2. $G_1[S]$ 是一个 SLR(1) 文法，画出它的 SLR(1) 分析表。

3. 小明希望为自己的编程语言引入指针特性，比如允许 $*a = **b$ 这样的写法。于是，他将文

法修改为 $G_2[S]$:

$$S \rightarrow L = R \mid R$$

$$L \rightarrow *R \mid i$$

$$R \rightarrow L$$

现在, 它不再是 SLR(1) 文法了, 但仍是 LR(1) 文法, 请画出它的 LR(1) 自动机。

4. 基于 $G_2[S]$ 的 LR(1) 自动机进行分析, 若处于某个正常状态时所面临的输入符号是 $=$, 且当前栈顶已形成句柄, 那么这样的状态共有多少个?
5. 基于 $G_2[S]$ 的 LR(1) 自动机进行分析, 若处于某个出错状态时所面临的输入符号是 $=$, 那么这样的状态共有多少个?
6. 小明不想实现上面复杂的 LR(1) 自动机, 他希望将其简化。如果改用 LALR(1) 自动机, 那么原 LR(1) 自动机中哪些状态可以被合并?
7. 请问 $G_2[S]$ 是否是 LALR(1) 文法? 要求只判断一种冲突, 是哪种?
8. 画出 $G_2[S]$ 的 LALR(1) 分析表。
9. 基于你的 LALR(1) 分析表, 分析下面的输入串: $*i = **i$ 。
10. 小明希望为自己的编程语言引入连续赋值特性, 比如允许 $a=*b=c$ 这样的写法, 同时小明删去了对常表达式 a 的支持。于是, 他将文法修改为 $G_3[S]$:

$$S \rightarrow L = R$$

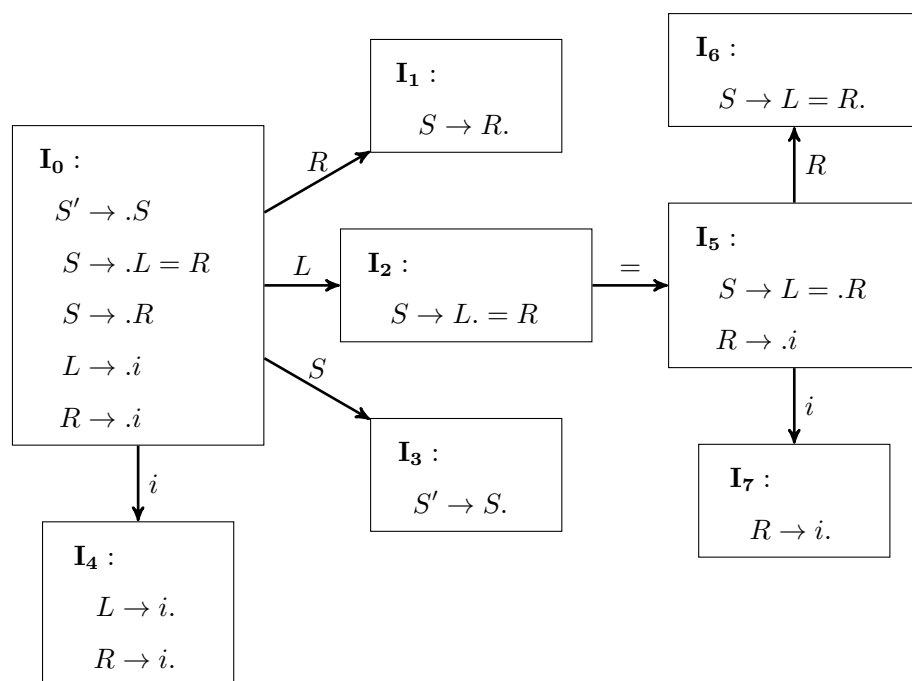
$$L \rightarrow *R \mid i$$

$$R \rightarrow L \mid L = R$$

小明发现, 这是一个二义文法。请给出一个合理的限定规则, 并说明该规则会引起 LR 分析表中何种变化 (描述到位即可, 无需构造出分析表)。

参考答案:

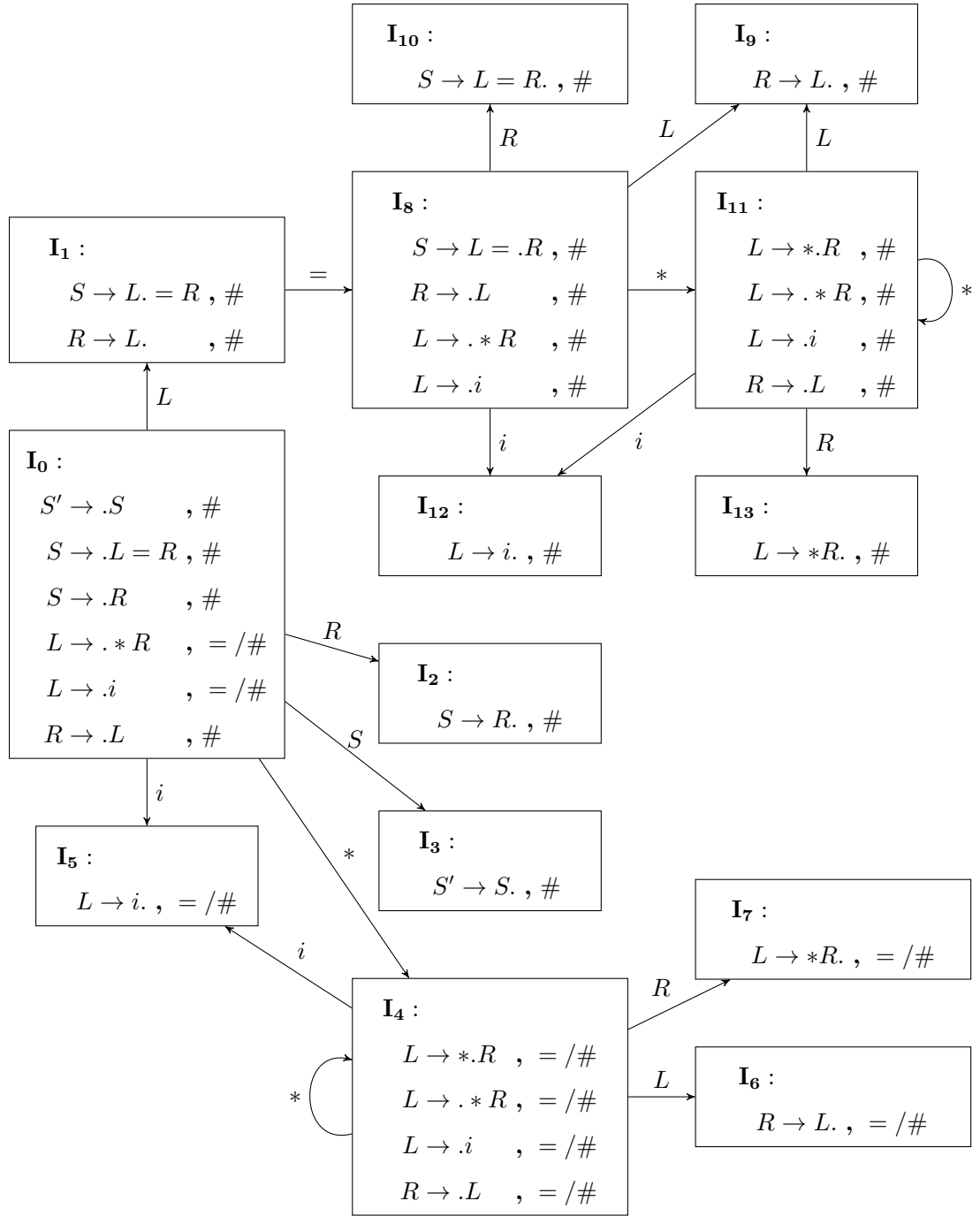
1. 增广文法 $G'_1[S']$ 的 LR(0) 自动机如下图所示。状态 I_4 存在归约-归约冲突, 因此 $G_1[S]$ 不是 LR(0) 文法。(思考: 对于一个合法字符串, 第一步一定是移进 i , 此后会陷入归约/归约冲突 ($L \rightarrow i$ 和 $R \rightarrow i$), 故不是 LR(0) 文法。)



2. 增广文法 $G'_1[S']$ 的 SLR(1) 分析表如下表所示。

栈顶状态	ACTION	GOTO		
	i $=$ $\$$	S	L	R
0	s4	3	2	1
1				
2	s5			
3				
4	r3 r4			
5	s7			6
6				
7				

3. 增广文法 $G'_2[S']$ 的 LR(1) 自动机如下图所示。



- 共 3 个状态: I_5, I_6, I_7 。栈顶已经形成句柄的含义是该状态将进行归约。
- 共 5 个状态: $I_0, I_4, I_8, I_{11}, I_{12}$ 。注意这样的状态不能通过归约得到, 因为即将归约时会检查下一个符号, 如不匹配会直接报错。
- 共 4 组: 状态 I_4, I_{11} 、状态 I_5, I_{12} 、状态 I_6, I_9 以及状态 I_7, I_{13} 。
- 合并过程没有引入归约-归约冲突, 因此 $G_2[S]$ 是 LALR(1) 文法。
- 增广文法 $G'_2[S']$ 的 LALR(1) 分析表如下表所示。

栈顶状态	ACTION				GOTO		
	i	*	=	#	S	L	R
0	s5-12	s4-11			3	1	2
1			s8	r5			
2				r2			
3				acc			
4-11	s5-12	s4-11				6-9	7-13
5-12			r4	r4			
6-9			r5	r5			
7-13			r3	r3			
8	s5-12	s4-11				6-9	10
10				r1			

9. 分析过程如下表所示。

状态栈	余留符号串	下一步动作
0	$*i = **i\#$	ACTION[0, *] = s4-11
0 4-11	$i = **i\#$	ACTION[4-11, i] = s5-12
0 4-11 5-12	$= **i\#$	ACTION[5-12, =] = r4, GOTO[4-11, L] = 6-9
0 4-11 6-9	$= **i\#$	ACTION[6-9, =] = r5, GOTO[4-11, R] = 7-13
0 4-11 7-13	$= **i\#$	ACTION[7-13, =] = r3, GOTO[0, L] = 1
0 1	$= **i\#$	ACTION[1, =] = s8
0 1 8	$**i\#$	ACTION[8, *] = s4-11
0 1 8 4-11	$*i\#$	ACTION[4-11, *] = s4-11
0 1 8 4-11 4-11	$i\#$	ACTION[4-11, i] = s5-12
0 1 8 4-11 4-11 5-12	$\#$	ACTION[5-12, $\#$] = r4, GOTO[4-11, L] = 6-9
0 1 8 4-11 4-11 6-9	$\#$	ACTION[6-9, $\#$] = r5, GOTO[4-11, R] = 7-13
0 1 8 4-11 4-11 7-13	$\#$	ACTION[7-13, $\#$] = r3, GOTO[4-11, L] = 6-9
0 1 8 4-11 6-9	$\#$	ACTION[6-9, $\#$] = r5, GOTO[4-11, R] = 7-13
0 1 8 4-11 7-13	$\#$	ACTION[7-13, $\#$] = r3, GOTO[8, L] = 6-9
0 1 8 6-9	$\#$	ACTION[6-9, $\#$] = r5, GOTO[8, R] = 10
0 1 8 10	$\#$	ACTION[10, $\#$] = r1, GOTO[0, S] = 3
0 3	$\#$	ACTION[3, $\#$] = acc

10. 可以规定 * 运算符优先级高于 = 运算符。原 LR 分析表中会存在移进-归约冲突（冲突表项对应的输入符号为 =），前述规则下我们强制归约（不允许移进），该表项不再冲突。

Q3. 在课堂上，我们介绍了 LR(0) 分析器，却不曾提到 LL(0) 分析器。下面我们来探讨一下 LL 分析和 LR 分析的一些区别。

1. LR(0) 分析器使用 LR 分析表，会根据下一个输入符号来判断当前需要采取的行动，为什么被认为是“不需要向前查看符号”？（可以考虑 LR(0) 分析中有哪些不同的行动，在 LR(0) 分析表中决定采取哪种行动需要向前查看符号吗？）
2. 请分析为什么我们不曾提到 LL(0) 分析器？它相比 LR(0) 分析器差在哪里？

3. 语法分析过程是对语法分析树的遍历过程，对每一个内部节点（即非终结符），对其的“访问”过程是确定其使用何种产生式的过程。从这个角度考虑，LL 分析相当于 _____（先序/后序）遍历，LR 分析相当于 _____（先序/后序）遍历。

参考答案：

1. 在 LR(0) 分析中，移进和归约是不同的行动，按不同产生式归约也是不同的行动，而移进不同的符号是相同的行动（不存在“移进—移进冲突”）。LR(0) 分析表中一行或者只有移进，或者全部是相同的归约，因此不需要向前查看符号，只根据栈顶状态就能决定下一步行动。移进后转移到不同的状态可以认为是已经读入了下一个符号后发生的行为，不属于向前查看。
2. LL(0) 分析器只能根据当前最左侧非终结符选择产生式，这就要求每个非终结符的产生式唯一，即一个文法要么产生唯一字符串，要么陷入死循环，这不具备应用价值。相比之下，虽然 LR(0) 分析器也不能向前查看符号，但它维护了一个状态栈，可以根据栈顶状态选择移进或归约，因此分析能力强于 LL(0)。
3. LL 分析相当于先序遍历，先选择产生式再匹配符号；LR 分析相当于后序遍历，先移进符号再匹配产生式。（因此在选择产生式时，LR 比 LL 有更多的已知信息。）

Principles and Practice of Compiler Construction HW

Due: 11/11/2025

If you have any suggestions, feel free to contact the TAs.

Q1. The following grammar $G[S]$ describes the propositional formulas containing variables p , q , connectives $\wedge$ (logic and), $\vee$ (logic or), and $\neg$ (logic not):

$$S \rightarrow S \vee T \mid T$$

$$T \rightarrow T \wedge F \mid F$$

$$F \rightarrow \neg F \mid p \mid q$$

Please point out all the phrases and direct phrases of the following sentential forms. If the form is a right-sentential form, please also point out its handle.

1. $\neg F \vee \neg q \wedge p$

2. $\neg F \vee p \wedge \neg F \vee T$

Solution:

1. Phrases: $\neg F \vee \neg q \wedge p$, $\neg F$, $\neg q \wedge p$, $\neg q$, q , p ;

Direct phrases: $\neg F$, p , q ;

Handle: $\neg F$.

2. Phrases: $\neg F \vee p \wedge \neg F \vee T$, $\neg F \vee p \wedge \neg F$, $\neg F$ (first), $p \wedge \neg F$, p , $\neg F$ (second);

Direct phrases: $\neg F$ (first), p , $\neg F$ (second).

Notice: the same phrase occurring multiple times should be counted separately.

Q2. A student is designing a simple programming language.

1. First, he wants to include constant expression $\mathbf{a}$ and assignment statement $\mathbf{a=b}$. Both left side (L) and right side (R) of an assignment are identifiers (terminal i). The grammar can be written as $G_1[S]$:

$$S \rightarrow L = R \mid R$$

$$L \rightarrow i$$

$$R \rightarrow i$$

Is this grammar an LR(0) grammar? Explain why with its LR(0) automaton.

Thought-provoking: Can you decide this problem without drawing the automaton?

2. $G_1[S]$ is an SLR(1) grammar. Draw its SLR(1) parsing table.

3. He wants to include pointer into his language, for example, $*a**b$. Therefore, he changes the grammar into $G_2[S]$:

$$S \rightarrow L = R \mid R$$

$$L \rightarrow *R \mid i$$

$$R \rightarrow L$$

Now, it is not an SLR(1) grammar, but still LR(1). Draw its LR(1) automaton.

4. Given $G_2[S]$'s LR(1) automaton, if in some legal case we face an input symbol $=$, and the current stack top forms a handle, how many LR(1) states can we be in?
5. Given $G_2[S]$'s LR(1) automaton, if we face an input symbol $=$ when an error occurs, how many LR(1) states can we be in?
6. The student doesn't want to implement the complex LR(1) automaton above, he wants to simplify it. If we use LALR(1) automaton, which states can be merged in the original LR(1) automaton?
7. Is $G_2[S]$ an LALR(1) grammar? We only need to consider one kind of conflicts here, which one is it? (*Shift-shift or shift-reduce?*)
8. Draw the LALR(1) parsing table of $G_2[S]$.
9. Parse the input string $*i = **i$ based on your LALR(1) parsing table.
10. The student wants to include chained assignment such as $a=b=c$ into his language. Meanwhile, he removes constant expression a from it. So he changes the grammar into $G_3[S]$:

$$S \rightarrow L = R$$

$$L \rightarrow *R \mid i$$

$$R \rightarrow L \mid L = R$$

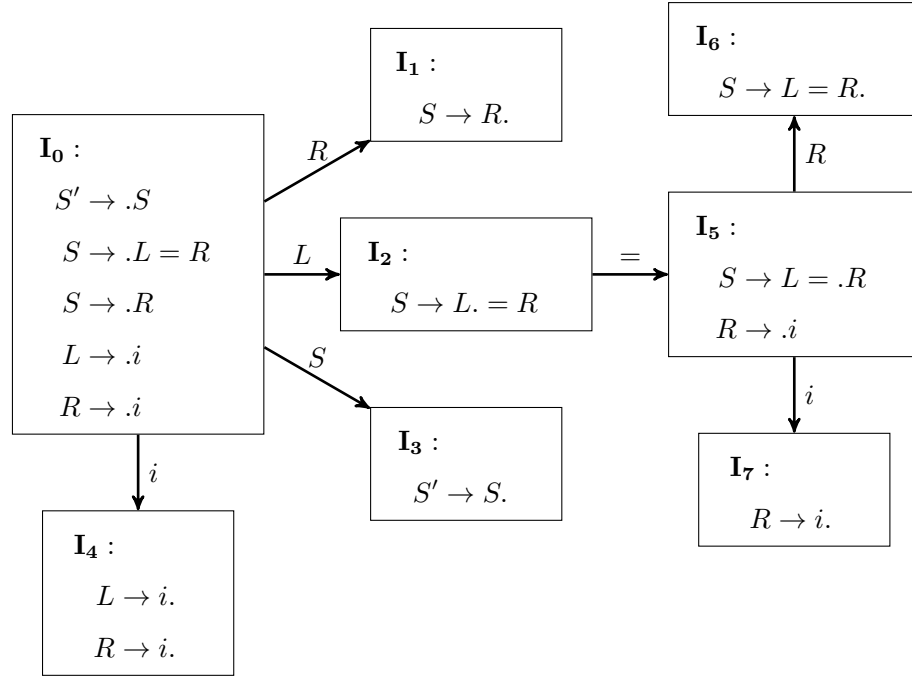
He finds out that this is an ambiguous grammar. Please give a reasonable disambiguating rule and explain what kind of changes this rule would cause in the LR parsing table.

Describe it appropriately; there is no need to construct the actual parsing table.

Solution:

1. The LR(0) automaton of augmented grammar $G'_1[S']$ is as follows. State I_4 has a reduce-reduce conflict, so $G_1[S]$ is not LR(0).

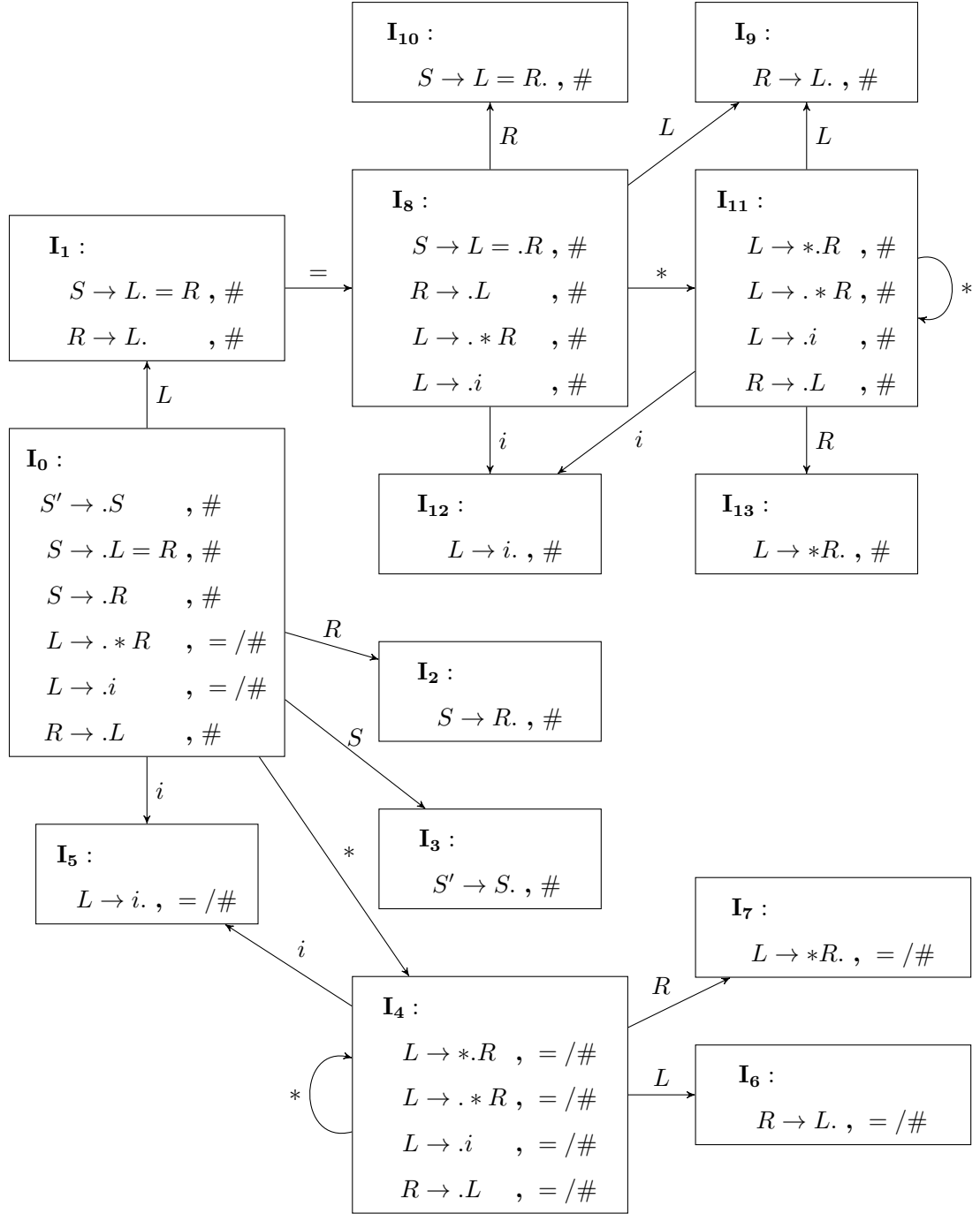
Thought-provoking: For any legal string, the first step must be shifting an i , then reduce-reduce conflict ($L \rightarrow i$ vs $R \rightarrow i$) will be faced, not LR(0).



2. SLR(1) table of augmented grammar $G'_1[S']$ is as follows.

Stack top	ACTION	GOTO		
	i $=$ $\$$	S	L	R
0	s4	3	2	1
1	r2			
2	s5			
3	acc			
4	r3 r4			
5	s7			6
6	r1			
7	r4			

3. LR(1) automaton of augmented grammar $G'_1[S']$ is as follows.



4. There are 3 such states: I_5, I_6, I_7 . *Stack top forms a handle means a reduction will occur.*
5. There are 5 such states: $I_0, I_4, I_8, I_{11}, I_{12}$. *Notice: Such states cannot be reached by reduction, because when about to reduce, the next symbol will be checked, and an error will be thrown immediately if mismatch.*
6. There are 4 groups: (I_4, I_{11}) , (I_5, I_{12}) , (I_6, I_9) as well as (I_7, I_{13}) .
7. No reduce-reduce conflict involved in the merging process, so it is an LALR(1) grammar.
8. LALR(1) table of augmented grammar $G'_2[S']$ is as follows.

Stack top	ACTION				GOTO		
	<i>i</i>	*	=	#	<i>S</i>	<i>L</i>	<i>R</i>
0	s5-12	s4-11			3	1	2
1			s8	r5			
2				r2			
3				acc			
4-11	s5-12	s4-11				6-9	7-13
5-12			r4	r4			
6-9			r5	r5			
7-13			r3	r3			
8	s5-12	s4-11				6-9	10
10				r1			

9. Parsing process as follows.

State stack	Remaining string	Next action
0	<i>*i = **i#</i>	ACTION[0, *] = s4-11
0 4-11	<i>i = **i#</i>	ACTION[4-11, i] = s5-12
0 4-11 5-12	<i>= **i#</i>	ACTION[5-12, =] = r4, GOTO[4-11, L] = 6-9
0 4-11 6-9	<i>= **i#</i>	ACTION[6-9, =] = r5, GOTO[4-11, R] = 7-13
0 4-11 7-13	<i>= **i#</i>	ACTION[7-13, =] = r3, GOTO[0, L] = 1
0 1	<i>= **i#</i>	ACTION[1, =] = s8
0 1 8	<i>**i#</i>	ACTION[8, *] = s4-11
0 1 8 4-11	<i>*i#</i>	ACTION[4-11, *] = s4-11
0 1 8 4-11 4-11	<i>i#</i>	ACTION[4-11, i] = s5-12
0 1 8 4-11 4-11 5-12	<i>#</i>	ACTION[5-12, #] = r4, GOTO[4-11, L] = 6-9
0 1 8 4-11 4-11 6-9	<i>#</i>	ACTION[6-9, #] = r5, GOTO[4-11, R] = 7-13
0 1 8 4-11 4-11 7-13	<i>#</i>	ACTION[7-13, #] = r3, GOTO[4-11, L] = 6-9
0 1 8 4-11 6-9	<i>#</i>	ACTION[6-9, #] = r5, GOTO[4-11, R] = 7-13
0 1 8 4-11 7-13	<i>#</i>	ACTION[7-13, #] = r3, GOTO[8, L] = 6-9
0 1 8 6-9	<i>#</i>	ACTION[6-9, #] = r5, GOTO[8, R] = 10
0 1 8 10	<i>#</i>	ACTION[10, #] = r1, GOTO[0, S] = 3
0 3	<i>#</i>	ACTION[3, #] = acc

10. We can require that operator $*$ has a higher priority than operator $=$. The original LR table contains shift-reduce conflict when the input symbol is $=$. By adding this rule, we only choose to reduce in such cases, and the conflict will be resolved.

Q3. In class, we introduced the LR(0) parser, but without introducing the LL(0) parser. Now let's discuss the differences between LL and LR.

1. LR(0) parser uses an LR parsing table, and it will choose next action according to the next symbol. If so, why do we say LR(0) "does not need to look ahead symbols"?
Consider what different actions are there in LR(0) parsing. Do we really need looking ahead to decide which action to take?

2. Explain why we did not mention LL(0) parser. Why is it worse than LR(0) parser?
3. The parsing process is a traversing process of the syntactic tree. For an internal node (a nonterminal), we define "to visit it" means "to decide which production we use on this nonterminal". Given this idea, LL parsing is a _____ (preorder/postorder) traversing, LR parsing is a _____ (preorder/postorder) traversing.

Solution:

1. In LR(0) parsing, shift and reduction are seen as different actions, so are reductions of different productions. However, shifts of different symbols are the same action (there is NO shift-shift conflict). As the rows in LR(0) parsing table consist of either only shifts, either the same reduction, we don't need to look ahead symbols to decide which action to take, we can do this simply relying on the stack top state. We do jump to different states after different shifts, but this happens after we read in the symbol, so it's not a look ahead.
2. LL(0) parser decides the production only based on the leftmost nonterminal, so for any given grammar, we either get a fixed string, either get into a dead loop. In comparison, although LR(0) parser cannot look ahead, it maintains a state stack, and we can decide the action based on the stack top state. Therefore, LR(0) is stronger than LL(0).
3. LL parsing is a preorder traversing, we choose production and then match tokens. LR parsing is a postorder traversing, we shift in tokens and then match production.
Therefore, when deciding which production to use, LR has more information than LL.