第五章 自底向上的分析

自底向上分析算法功能更强大也更加复杂

最普通的自底向上算法称作LR(1)分析，`L`表示自左向右处理输入，`R`表示生成了最右推导，`1`表示使用一个先行符号.

自底向上分析的作用还表现在它使得LR(0)分析也具有了意义

自底向上分析概览

自底向上的分析程序使用了显式栈来完成分析，这与非递归的自顶向下的分析程序相类似。分析栈包括记号和非终结符，以及一些后面将讨论到的其他信息。自底向上的分析开始时栈是空的，在成功分析的末尾还包括了开始符号。

自底向上分析程序有时候也称作移进-归约分析程序

- 移进(shift)：将终结符由输入的开头移进栈的顶部
- 归约(reduce)：假设有BNF选择A→α,将栈顶部的串α归约为非终结符A

eg:

Example Bottom-Up Parsing

$S' \rightarrow S$
$S \rightarrow (S) S \mid \varepsilon$

String: ( ) **Rightmost derivation**
Derivation: S'=> S => (S) S=> (S) => ( )
The reverse of a Postorder numbering

	Parsing Stack	Input	Action
1	$	( )$	shift
2	$(	)$	reduce $S \rightarrow \varepsilon$
3	$(S	)$	shift
4	$ (S)	$	reduce $S \rightarrow \varepsilon$
5	$ (S) S	$	reduce $S \rightarrow (S) S$
6	$ S	$	reduce $S' \rightarrow S$
7	$ S'	$	accept

Example 5.2

$E' \rightarrow E$
$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$

String: **n + n**
Derivation: E' => E => E + **n** => **n** + **n**

	Parsing Stack	Input	Action
1	$	n + n$	shift
2	$n	+ n$	reduce E → n
3	$E	+ n$	shift
4	$E +	n$	shift
5	$E + n	$	reduce E → E + n
6	$E	$	reduce E' → E
7	$E'	$	accept

自底向上的分析程序可将输入符号移进到栈里直到它判断出要执行的是何种动作为止。

但是为了判断要完成的动作，自底向上的分析程序会需要在栈内看得更深，而不仅仅是顶部。

输入中的记号需要作为一个先行来考虑

移进-归约分析程序描绘出输入串的最右推导，但推导步骤的顺序却是颠倒的。

终结符和非终结符的每个中间串都称作右句型

- Example: E' => E => E + n => n + n

符号 || 指出每一时刻栈的顶部位于何处，在每一种情况下，分析栈的符号序列都被称作右句型的可行前缀

Viable Prefix

	Parsing Stack	Input	Action
1	$ ||	n + n$	shift
2	$n ||	+ n$	reduce E → n
3	$E ||	+ n$	shift
4	$E + ||	n$	shift
5	$E + n ||	$	reduce E → E + n
6	$E ||	$	reduce E' → E
7	$E' ||	$	accept

Derivation: E' => E => E + n => n + n

> The sequence of symbols on the parsing stack is called a viable prefix of the right sentential form.

移进-归约分析程序将终结符从输入移进到栈直到它能执行一个归约以得到下一个右句子格式。它发生在位于栈顶部的符号串匹配用于下一个归约的产生式的右边。这个串、它在右句子格式中发生的位置以及用来归约它的产生式被称作右句型的句柄 (handle)。

E' => E => E + n => n + n

Viable prefix

Right sentential form

Production

	Parsing Stack	Input	Action
1	$	n + n$	shift
2	$n	+ n$	reduce E → n

LR(0)项的有穷自动机

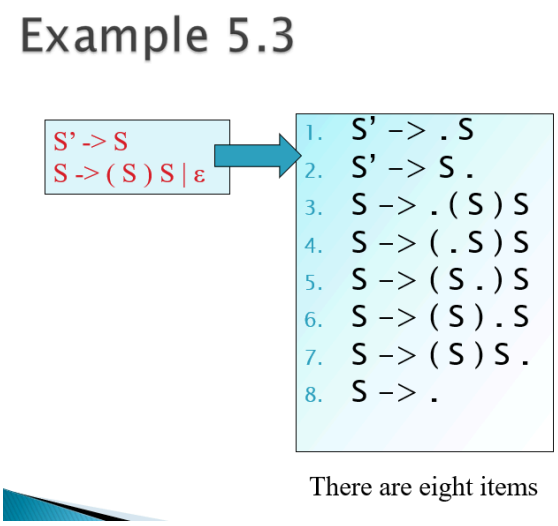
LR(0)项

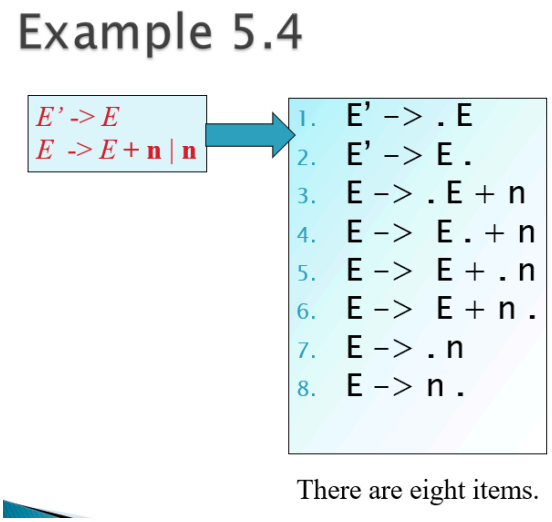
上下文无关文法的LR(0)项(LR(0) item)(或简写为项(item))是在其右边带有区分位置的产生式选择。

我们可用一个句点 (当然它就变成了元符号，而不会与真正的记号相混淆)来指出这个区分的位置。

若 A→a是产生式选择，且若β和γ是符号的任何两个串（包括空串 ）， 且存在着βγ = α，那么A -> β.γ就是LR (0)项。

eg:



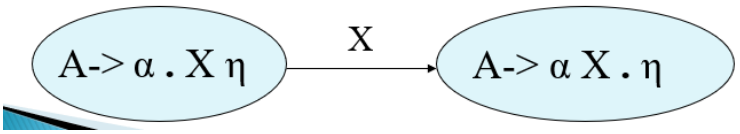


项目概念的思想就是指项目记录了特定文法规则右边识别中的中间步骤。特别地，项目 $A \rightarrow \beta.\gamma$ 是由文法规则选择$A \rightarrow \alpha$构成（其中$\alpha=\beta\gamma$），这一点意味着早已看到了β，且可能从下一个输入记号中获取γ。从分析栈的观点来看，这就意味着 β必须出现在栈的顶部。项目 $A \rightarrow .\alpha$意味着将要利用文法规则选择$A \rightarrow \alpha$识别A(将这样的项目称作初始项(initial item)。项目$A \rightarrow \alpha.$意味着α现在位于分析栈的顶部，而且若$A \rightarrow \alpha$在下一个归约中使用的话，它有可能就是句柄（将这样的项目称作完整项（complete item））。

项目的有穷自动机

NFA

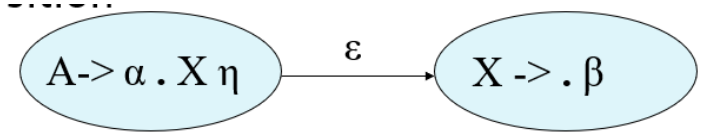
若有项目 `A -> α . γ`，且假设 `γ` 以符号 `X` 开始，其中 `X` 可以是 记号或非终结符，所以该项目就可写作 `A -> α . X η`



若 X 是一个记号，那么该转换与 X 的一个在分析中从输入到栈顶部的移进相对应。

若X是一个非终结符，X永远不会作为输入符号出现,实际上,这样的转换仍与在分析时将X压入到栈中相对应，但是它只发生在由产生式 `X -> β` 形成的归约时.

由于这样的归约前面必须有一个 `β` 的识别，而且由初始项 `X→.β` 给出的状态代表了这个处理的开始(句点指出将要识别一个 `β`)，则对于每个项目 `A -> α. X η` ，必须为X的每个产生式 `X→β` 添 加一个 `ε` 产生式,



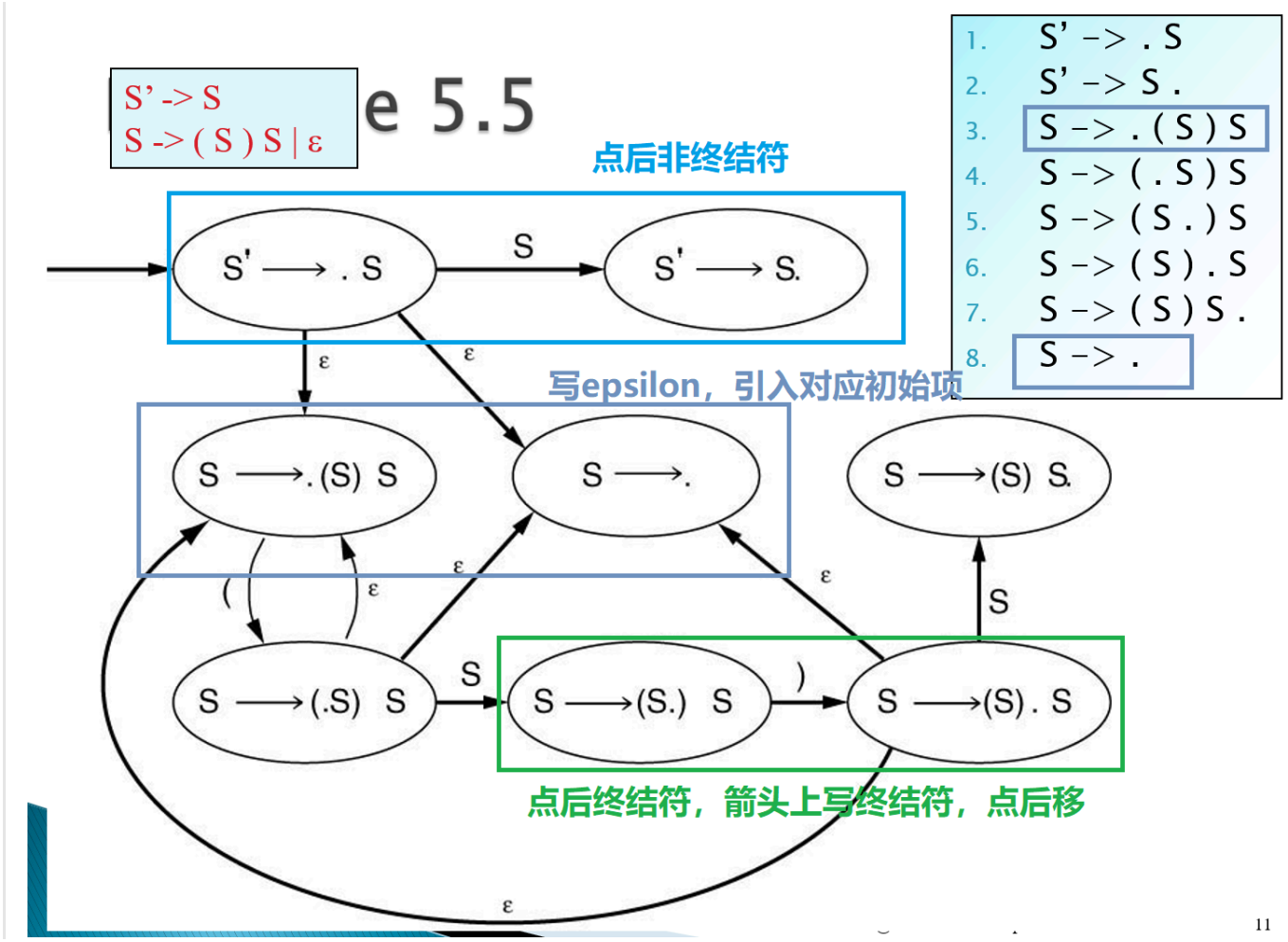
NFA的初始状态应与分析程序的初始状态相对应：栈是空的，而且将要识别一个 S

任何由 S的产生式选择构造的初始项 `S -> . α` 都可作为开始状态,但S可能有许多这样的产生式,通过产生式 `S'→S` 扩充(augment)文法，其中 `S'` 是一个新的非终结符。接着，S'成为扩充文法(augmented grammar)的开始状态

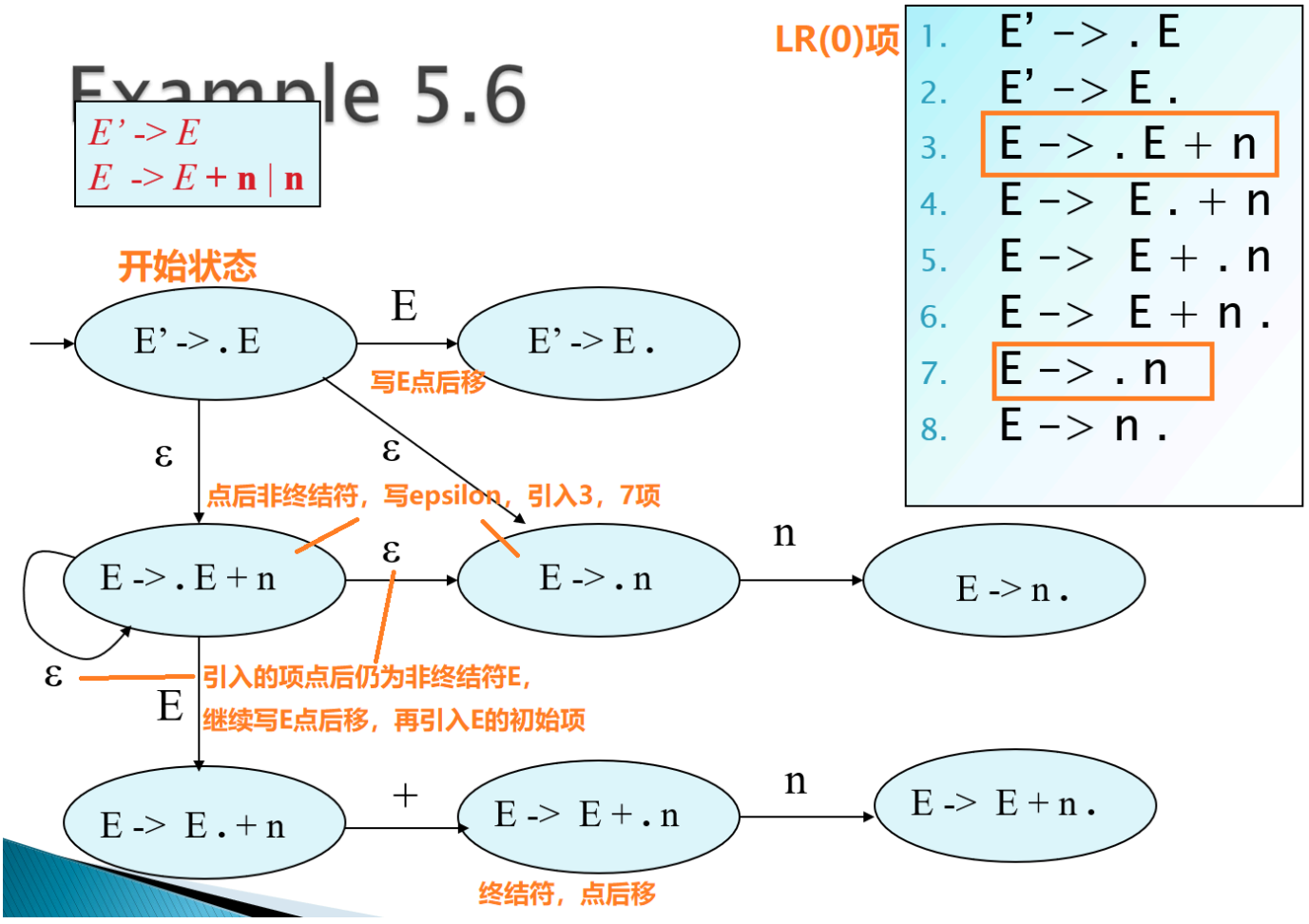
分析程序本身将决定何时接受，而NFA则无需包含这一信息，所以NFA实际上根本就没有接受状态

eg:

- step1:写LR(0)项
- step2：从开始状态开始
  - 如果点后是终结符，箭头上写该终结符，指向点后移之后的LR(0)项
  - 如果点后是非终结符，同上操作，然后箭头上写 `ε` 将该非终结符开头的初始项写进来（点紧挨着箭头的）如果引进来的点后还是非终结符，则仍要按此操作


S' -> S
S -> (S) S | ε
e 5.5
点后非终结符
1. S' -> . S
2. S' -> S .
3. S -> . (S) S
4. S -> (. S) S
5. S -> (S .) S
6. S -> (S) . S
7. S -> (S) S .
8. S -> .
写epsilon，引入对应初始项
S' ⟶ . S
S
S' ⟶ S.
ε
S ⟶ .(S) S
S ⟶ .
S ⟶ (S) S.
S ⟶ (.S) S
S ⟶ (S.) S
S ⟶ (S) . S
点后终结符，箭头上写终结符，点后移

eg:


Example 5.6
E' -> E
E -> E + n | n
LR(0)项
1. E' -> . E
2. E' -> E .
3. E -> . E + n
4. E -> E . + n
5. E -> E + . n
6. E -> E + n .
7. E -> . n
8. E -> n .
开始状态
E' -> . E
E
E' -> E .
写E点后移
ε
ε
点后非终结符，写epsilon，引入3，7项
E -> . E + n
ε
E -> . n
n
E -> n .
ε
引入的项点后仍为非终结符E，
E
继续写E点后移，再引入E的初始项
E -> E . + n
+
E -> E + . n
n
E -> E + n .
终结符，点后移

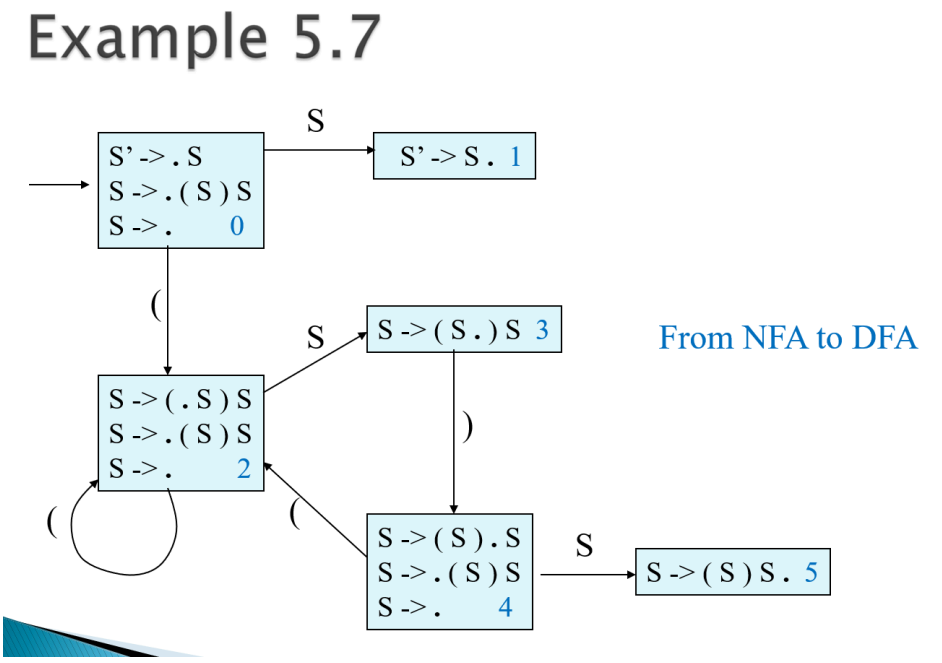
NFA-DFA

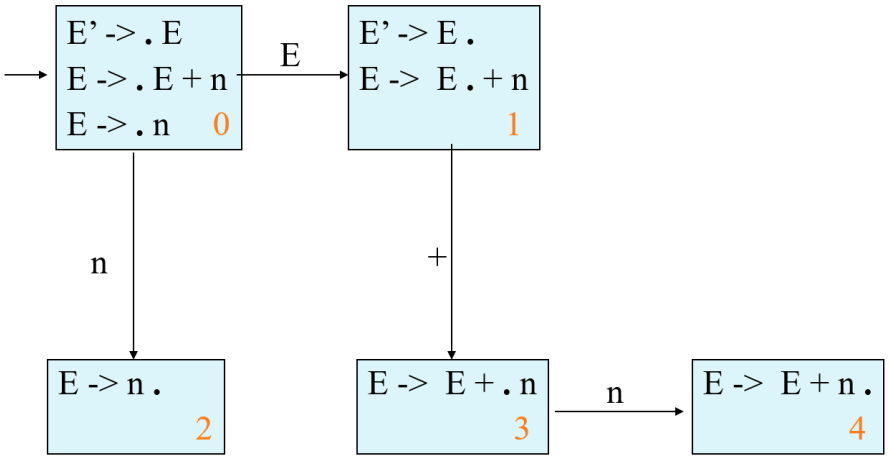
讲解视频：【编译原理- LR（0）】 https://www.bilibili.com/video/BV11C4y1e7uo/?share_source=copy_web&vd_source=d80f309439bdd49df0b0808ccb80749a

- 归约项：点在最后
- 移进项：点后终结符
- 待约项：点后非终结符
- 接受：扩充项

遇到待约项，将点后非终结符的式子全写进来，点在最左

eg:

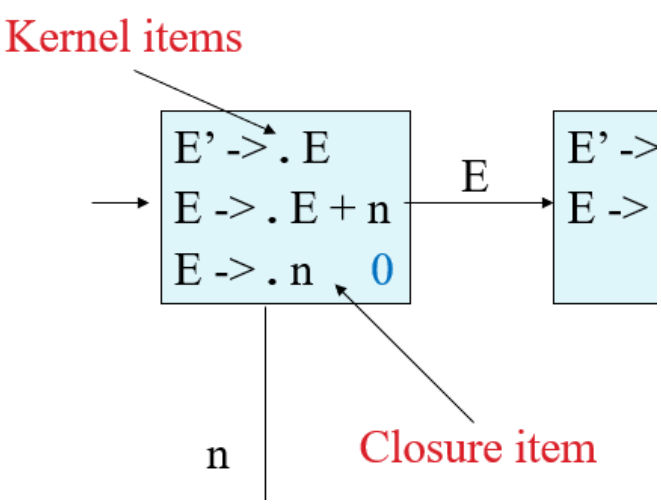


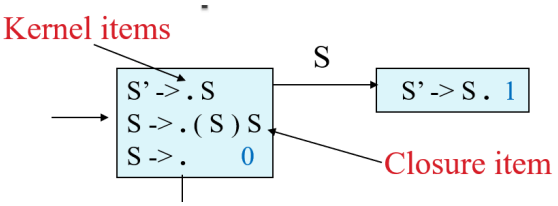


在ε闭包步骤中添加到状态中的项目被称为闭包项，所有的闭包项都是初始项

引起状态作为非ε-转换的目标的那些项目称为核心项，核心项唯一地判断出状态以及它的转换，只需要指出核心项就可以完整的表示出项目集合的DFA来。项目集合的DFA可以被直接计算。

eg:





LR(0)分析算法

于要了解项目集合的DFA的当前状态，须修改分析栈以使不但能存储符号而且还能存储状态数。这是通过在压入一个符号之后再将新的状态数压入到分析栈中完成的。

初始化：将底标记$和开始状态0放入栈中

Parsing Stack	Input
$ 0	*Input String* $

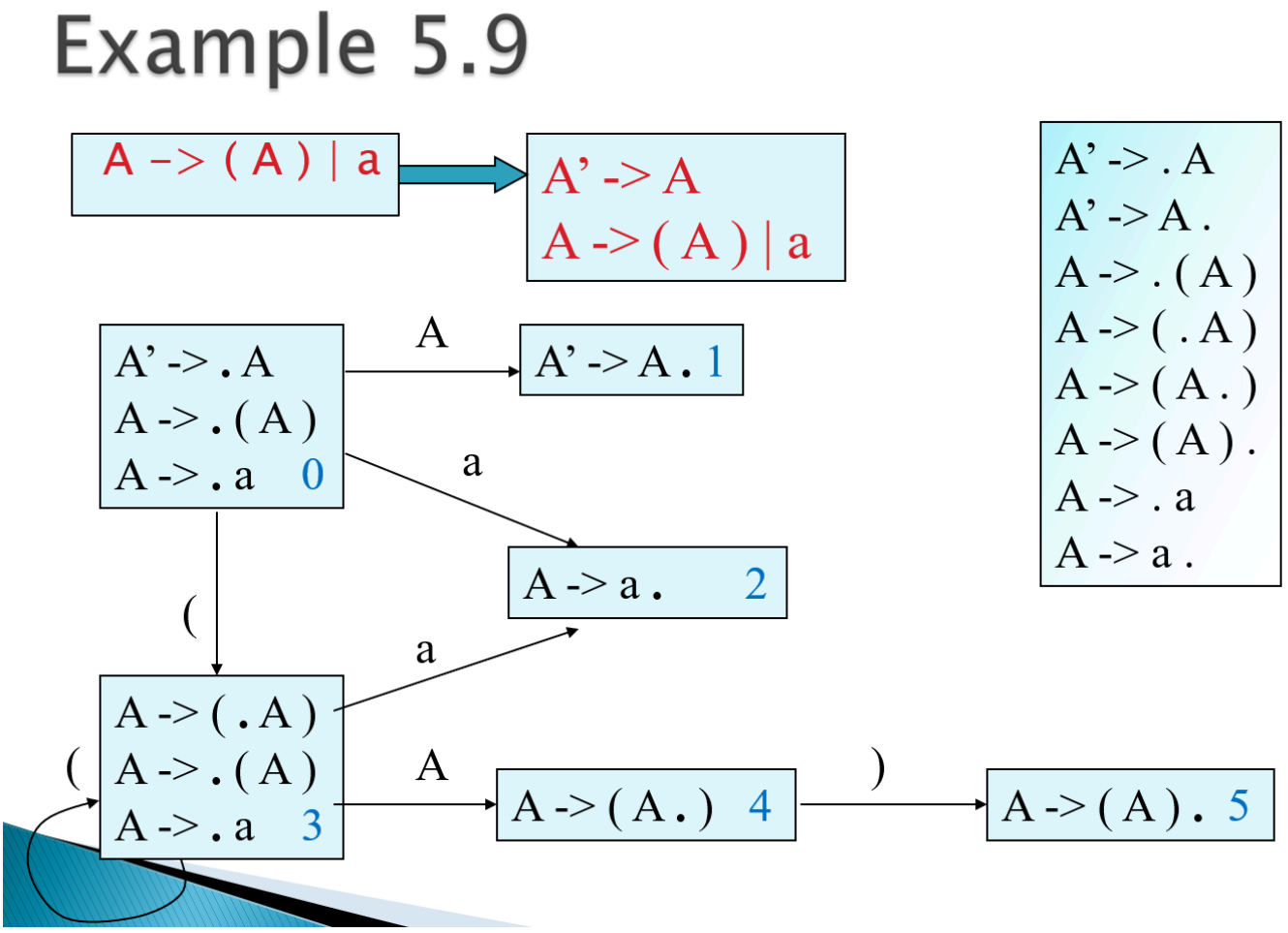
LR（0）分析算法根据当前的DFA状态选择一个动作，这个状态总是出现在栈的顶部

定义： LR(0)分析算法(LR(0) parsing algorithm)。令s 为当前的状态(位于分析栈的顶部)。则动作定义如下：

1. 若状态s 包含了格式$A\rightarrow\alpha.X\beta$的任何项目，其中$X$是一个终结符，则动作就是将当前的输入记号移进到栈中。若这个记号是 X，且状态s 包含了项目$A\rightarrow\alpha.X\beta$，则压入到栈中的新状态就是包含了项目 $A\rightarrow\alpha X.\beta$的状态。若由于位于刚才所描述的格式的状态$s$ 中的某个项目，这个记号不是X，则声明一个错误。
2. 若状态s 包含了任何完整的项目(格式$A\rightarrow\alpha.$ 的一个项目)，则动作是用规则$A\rightarrow\alpha$归约。假设输入为空，用规则$S'\rightarrow S$归约(其中S是开始状态)与接受相等价；若输入不为空，则出现错误。在所有的其他情况中，新状态的计算如下：将串α及它的所有对应状态从分析栈中删去(根据DFA的构造方式，串α必须位于栈的顶部)。相应地，在DFA中返回到由α开始构造的状态中（这须是由α的删除所揭示的状态)。由于构造DFA，这个状态就还须包含格式$B\rightarrow\alpha.A\beta$的一个项目。将$A$压入到栈中，并压入包含了项目$B\rightarrow\alpha A.\beta$的状态（作为新状态)。(请注意，由于正将$A$压入到栈中，所以这与在DFA中跟随$A$的转换相对应(这实际上是合理的)。

若以上的规则都是无歧义的，则文法就是 LR(0)文法(LR(0) grammar)。这就意味着若一个状态包含了完整项目 $A\rightarrow\alpha.$，那么它就不能再包含其他项目了。实际上，若这样的状态还包含了一个“移进的”项目 $A\rightarrow\alpha.X\beta$（$X$是一个终结符），就会出现一个到底是执行动作 (1)还是执行动作(2)的二义性。这种情况称作移进 - 归约冲突 (shift-reduce conflict)。类似地，如果这样的状态包含了另一个完整项目 $B\rightarrow\beta.$，那么也会出现一个关于为该归约使用哪个产生式 ($A\rightarrow\alpha$ 或$B\rightarrow\beta$）二义性。这种情况称作归约 - 归约冲突 (reduce-reduce conflict)。所以，当仅当每个状态都是移进状态(仅包含了“移进”项目的状态)或包含了单个完整项目的归约时，该文法才是LR(0)。

eg:



State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'->A				
2	reduce	A->a				
3	shift		3	2		4
4	shift				5	
5	reduce	A->(A)				

	Parsing Stack	Input	Action
1	$ 0	((a)) $	shift
2	$ 0 (3	(a)) $	shift
3	$ 0 (3 (3	a)) $	shift
4	$ 0 (3 (3 a 2	)) $	reduce A->a
5	$ 0 (3 (3 A 4	)) $	shift
6	$ 0 (3 (3 A 4) 5	) $	reduce A->(A)
7	$ 0 (3 A 4	) $	shift
8	$ 0 (3 A 4)	$	reduce A->(A)
9	$ 0 A 1	$	accept

SLR(1)分析

简单LR(1)分析，或SLR(1)分析，也如上一节中一样使用了 LR(0)项目集合的DFA。但是，通过使用输入串中下一个记号来指导它的动作，它大大地提高了 LR(0)分析的能力。

- 它在一个移进之前先考虑输入记号以确保存在着一个恰当的DFA
- 它使用非终结符的 Follow集合来决定是否应执行一个归约

先行的这个简单应用的能力强大得足以分析几乎所有的一般的语言构造。

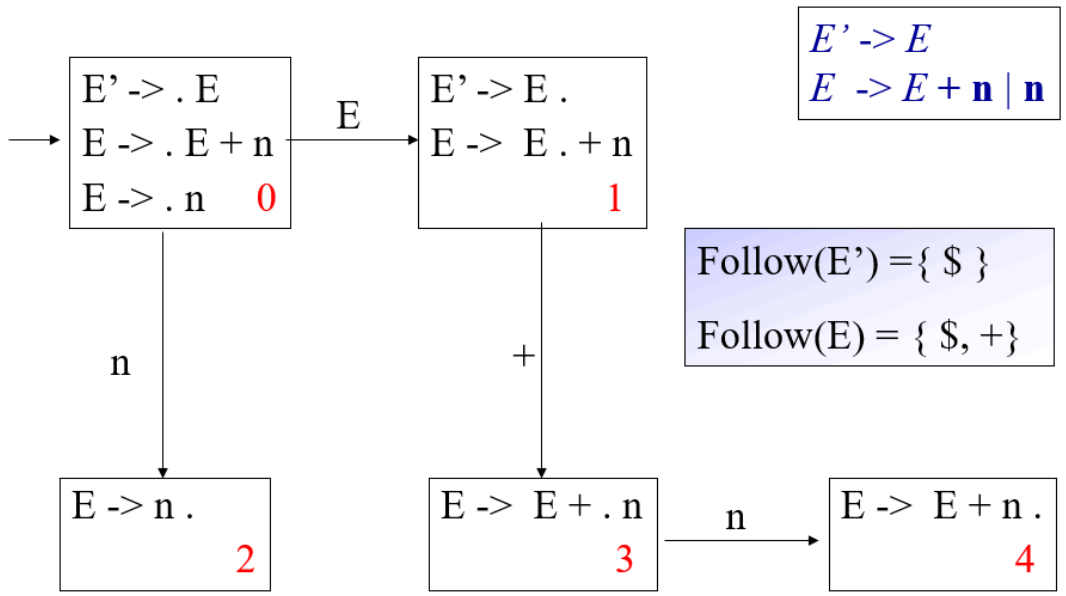
> **定义：** SLR(1)分析算法(SLR(1) parsing algorithm)。令s 为当前状态(位于分析栈的顶部)。则动作可定义如下：
>
> 1. 若状态s 包含了格式$A\to\alpha.X\beta$的任意项目，其中X是一个终结符，且X是输入串中的下一个记号，则动作将当前的输入记号移进到栈中，且被压入到栈中的新状态是包含了项目$A\to\alpha X.\beta$的状态。
> 2. 若状态s 包含了完整项目$A\to\gamma.$，则输入串中的下一个记号是在Follow(A)中 所以动作是用规则$A\to\gamma$ 归约。用规则$S'\to S$归约与接受等价，其中S是开始状态；只有当下一个输入记号是$时，这才会发生㊀。在所有的其他情况中，新状态都是如下计算的：删除串 α和所有它的来自分析栈中的对应状态。相对应地，DFA回到α开始构造的状态。通过构造，这个状态必须包括格式 $B\to\gamma.\ A\beta$的一个项目。将A压入到栈中，并将包含了项目$B\to\alpha A.\beta$的状态压入。
> 3. 若下一个输入记号都不是上面两种情况所提到的，则声明一个错误。
>
> 若上述的SLR(1)分析规则并不导致二义性，则文法为 **SLR(1)**文法(SLR(1) grammar)。特别地，当且仅当对于任何状态s，以下的两个条件：
>
> 否则对于A，可以移进，同时对于B可以归约
>
> 1) 对于在s 中的任何项目$A\to\alpha.X\beta$，当X是一个终结符，且X在Follow (B)中时，s 中没有完整的项目$B\to\gamma.$。
>
> 否则归约时不知道该归到A还是B
>
> 2) 对于在s 中的任何两个完整项目$A\to\alpha.$和$B\to\beta.$，Follow(A) ∩ Follow(B)为空。
>
> 均满足时，文法为SLR(1)。
>
> 若第1个条件不满足，就表示这是一个移进 - 归约冲突 (shift-reduce conflict)。若第2个条件不满足，就表示这是一个归约 - 归约冲突(reduce-reduce conflict)。

SLR(1)分析表：这里求follow集不用消除左递归，原来是什么样就怎么算

> SLR(1)分析的分析表也可以用与前一节所述的 LR(0)分析的分析表的类似方式构造。两者的差别如下：由于状态在SLR(1)分析程序中可以具有移进和归约(取决于先行)，输入部分中的每项现在必须要有一个“ 移进 ”或“ 归约 ”的标签，而且文法规则选择也必须被放在标有“ 归约 ”的项中。这还使得动作和规则列成为多余。由于输入结束符号 $也可成为一个合法的先行，所以必须为这个符号在输入部分建立一个新的列。我们将 SLR(1)分析表的构造放在SLR(1)分析的第1个示例中。

eg:

For any item A -> α . X β in s with X a terminal, there is no complete item B -> γ . in s with X in Follow(B)



Example 5.10 (cont)

State	Input			Goto
	n	+	$	E
0	s 2			1
1		s 3	accept	
2		r (E -> n)	r (E -> n)	
3	s 4			
4		r (E -> E+n)	r (E -> E+n)	

	Parsing Stack	Input	Action
1	$ 0	n + n + n $	shift 2
2	$ 0 n 2	+ n + n $	reduce E -> n
3	$ 0 E 1	+ n + n $	shift 3
4	$ 0 E 1 + 3	n + n $	shift 4
5	$ 0 E 1 + 3 n 4	+ n $	reduce E->E + n
6	$ 0 E 1	+ n $	shift 3
7	$ 0 E 1 + 3	n $	shift 4
8	$ 0 E 1 + 3 n 4	$	reduce E->E + n
9	$ 0 E 1	$	accept