第四章 语法分析

1. 语法分析的功能、基本任务
2. 自顶向下分析法
3. 自底向上分析法

4.1 语法分析概述

功能：根据文法规则，从源程序单词符号串中识别出语法成分，并进行语法检查。

基本任务：识别符号串S是否为某语法成分。

两大类分析方法：

自顶向下分析

自底向上分析

自顶向下分析算法的基本思想为:

若 $Z \overset{+}{\underset{G[Z]}{\Rightarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

? 存在主要问题:
- 左递归问题
- 回溯问题

■ 主要方法:
- 递归子程序法
- LL分析法

自底向上分析算法的基本思想为：

若 $Z \overset{+}{\underset{G[Z]}{\Leftarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

存在主要问题：

- 句柄的识别问题
- 若两个以上规则的右部有相同的符号串且构成句柄，选哪个候选式？

主要方法：

- 算符优先分析法
- LR分析法

4.2 自顶向下分析

4.2.1 自顶向下分析的一般过程

给定符号串S，若预测它是某一语法成分，那么可根据该语法成分的文法，设法为S构造一棵语法树。

若成功，则S最终被识别为某一语法成分。即

$S \in L(G[Z])$ 其中G[Z]为某语言成分的文法。

若不成功，则 $S \notin L(G[Z])$ 。

- 我们可以通过一例子来说明语法分析过程

例

2型文法！
上下文无关文法

S = c a d
G[Z]:
Z∷= c A d
A∷= a b | a
求解 S ∈ L(G[Z]) ?

分析过程是设法建立一棵语法树，使语法树的末端结点与给定符号串相匹配。

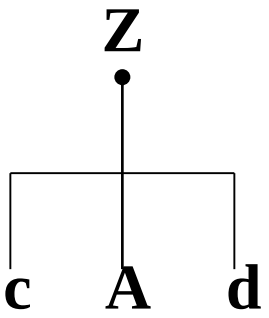
1.开始：令Z为根结点。

2.用Z的右部符号串去匹配输入串

完成一步推导 Z ⇒ c A d

检查 c - c匹配

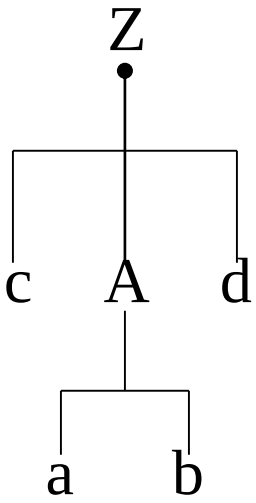
A是非终结符，将匹配任务交给A



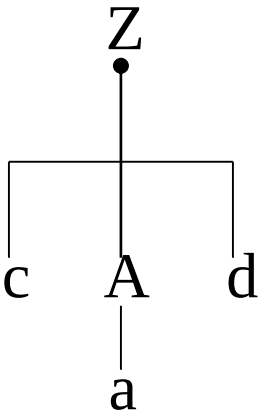
S = c a d　G[Z]: Z::= c A d
A::= a b | a

3.选用A的右部符号串匹配输入串
A有两个右部，选第一个。

完成进一步推导A ⇒ a b
检查：a - a匹配，b - d不匹配(失败)
但是还不能贸然宣布S∉L(G[Z])



4.回溯：即砍掉A的子树
改选A的第二个右部
A ⇒ a　检查 a - a匹配
d - d匹配



语法树末端结点为c a d与输入c a d相匹配，建立了推导序列 Z ⇒ c A d ⇒ c a d。
∴c a d ∈ L(G(Z))

自顶向下分析方法的特点

1. 分析过程是带有预测的。即要根据输入符号串中下一个单词，来预测之后的内容属于什么语法成分，然后用相应语法成分的文法建立语法树。

2. 分析过程是一种试探过程，是尽一切办法（选用不同规则）设法建立语法树的过程。由于是试探过程，故难免有失败，所以分析过程需进行回溯，因此我们也称这种方法是带回溯的自顶向下分析方法。

3. 最左推导可以编出程序来实现，但在实际上价值不大，效率低，代价高。

4.2.2 自顶向下分析存在的问题及解决方法

1、 左递归文法:

有如下文法:

令U是文法的任一非终结符，文法中有规则

$U ::= U\cdots$　　或者　　$U \overset{+}{\Rightarrow} U\cdots$

这个规则文法是左递归的。

该方法的基本缺点是不能处理具有左递归性的文法。

自顶向下分析为什么不能处理左递归文法?

如果我们在匹配输入串过程中，假定正好轮到要用非终结符U直接匹配输入串，即要用U的右部符号串U····去匹配，为了用U····去匹配，又得用U去匹配，这样无限地循环下去，过程将无法终止。

如果文法具有间接左递归，则也将发生上述问题，只不过兜的圈子更大。

要实行自顶向下分析，必须要消除文法的左递归，下面我们将介绍直接左递归的消除方法。在此基础上再介绍一般左递归的消除方法。

消除直接左递归

方法一：使用扩充的BNF表示来改写文法

例：(1) $E ::= E + T \mid T \quad \Rightarrow \quad E ::= T \{ + T \}$

(2) $T ::= T * F \mid T / F \mid F \quad \Rightarrow \quad T ::= F \{ * F \mid / F \}$

a. 改写以后的文法消除了左递归。

b. 可以证明，改写前后的文法是等价的，表现在

$L(G_{改前}) = L(G_{改后})$

如何改写文法能消除左递归，又前后等价？

现在我们可以给出两条规则。

规则一（提因子）

注意若是 y x | w x |....| z x
则x不是公因子！

若：U::= x y | x w |....| x z
则可改写为：U::= x (y | w |....| z)
其中再若：$y = y_1 y_2$，$w = y_1 w_2$
则　U::= x (y_1 (y_2 | w_2) |....| z)

若有规则：U::= x | x y
则可以改写为：U::= x (y |ε)
注意：不应写成U::= x (ε| y)

总把ε安置成最后的选择！

使用提因子法，不仅有助于消除直接左递归，而且有助于压缩文法的长度，使我们更加有效地分析句子。

规则二

若有文法规则：U::= x | y |...| z | U v

其特点是：具有一个直接左递归的右部并位于最后，这表明该语法类U是由 x 或 y...或 z打头，其后跟随零个或多个 v 组成。

U ⇒ Uv ⇒ Uvv ⇒ Uvvv ⇒ ……

∴ 可以改写为U::= (x | y |...| z) { v }

通过以上两条规则，就能消除文法的直接左递归，并且保证文法的等价性。

方法二：将左递归规则改为右递归规则

规则三

若：$P ::= P\alpha \mid \beta$

则可改写为：

$$P ::= \beta P'$$

$$P' ::= \alpha P' \mid \varepsilon$$

下面有两个例题：

若：P∷=Pα|β
则可改写为：P∷=βP’
P’∷=αP’|ε

例1　E∷=E+T|T

产生式右部无公因子，所以不能用规则一。

为了使用规则二，我们

令E∷=T|E+T

∴由规则二我们可以得到

E∷=T{+T}

右递归：

E ::= T E’

E’ ::= + T E’ | ε　　规则三

例2　T∷=T*F|T/F|F

T∷=F|T*F|T/F

T∷=F|T(*F|/F)　规则一

T∷=F{(*F|/F)}　规则二

即T∷=F{*F|/F}

右递归：

T ::= F T’

T’ ::= * F T’ | / F T’ | ε　规则三

消除一般左递归

一般左递归也可以通过改写法予以消除。

消除所有左递归的算法:

1.把G的非终结符整理成某种顺序A_1，A_2，……A_n，使得

$A_1 ::= \delta_1 | \delta_2 | \ldots\ldots | \delta_k$

$A_2 ::= A_1\, r\ldots\ldots$

$A_3 ::= A_2\, u \,|\, A_1\, v\ldots..$

……

2. for i:=1 to n do
   begin
     for j :=1 to i-1 do
       把每个形如$A_i ::= A_j r$的规则替换成
       $A_i ::= (\delta_1|\delta_2|\ldots\ldots|\delta_k)\, r$
       其中$A_j ::= \delta_1|\delta_2|\ldots\ldots|\delta_k$是当前$A_j$ 的全部规则
       消除A_i规则中的直接左递归
   end

3. 化简由 2 得到的文法，即去掉那些多余的规则。

例：文法G [S]为

S ::= Q c | c

Q ::= R b | b

R ::= S a | a

该文法无直接左递归，但有间接左递归

S ⇒ Q c ⇒ R b c ⇒ S a b c　　∴ S $\overset{+}{\Rightarrow}$ S a b c

非终结符顺序重新排列

R ::= S a | a

Q ::= R b | b

S ::= Q c | c

R::= S a | a
Q::= R b | b
S::= Q c | c

1.检查规则R是否存在直接左递归　　R::= S a | a

2.把R代入Q的有关选择，改写规则Q　　Q::= S a b | a b | b

3.检查Q是否直接左递归

4.把Q代入S的右部选择　　S::= S a b c | a b c | b c | c

5.消除S的直接左递归　　S::= (a b c | b c | c) { a b c }

最后得到文法为:

S::=(abc|bc|c){abc}
Q::=Sab|ab|b
R::=Sa|a

可以看出其中关于Q和R的规则是多余的规则
∴经过压缩后S::=(abc|bc|c){abc}
可以证明改写前后的文法是等价的

应该指出，由于对非终结符的排序不同，最后得到的文法在形式上可能是不一样的，但是不难证明它们的等价性。

2、 回溯问题

什么是回溯?

分析工作要部分地或全部地退回去重做叫回溯。

造成回溯的条件:

$U::= \alpha_1 | \alpha_2 | \alpha_3$

文法中，对于某个非终结符号的规则其右部有多个选择，并根据所面临的输入符号不能准确地确定所要的产生式，就可能出现回溯。

回溯带来的问题:

效率严重低下，只有在理论上的意义而无实际意义。

效率低的原因

1) 语法分析要重做

2) 语法处理工作要推倒重来

怎么样才能避免回溯?

[定义] 设文法G（不具左递归性），$U \in V_n$

$$U ::= \alpha_1 \mid \alpha_2 \mid \alpha_3$$

$$FIRST(\alpha_i) = \{a \mid \alpha_i \Rightarrow a\ldots, a \in V_t\}$$

为避免回溯，对文法的要求是:

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varphi \quad (i \neq j)$$

消除回溯的途径:

1.改写文法

对具有多个右部的规则反复提取左因子

例1　U::= x V | x W　　U, V, W∈V_n,　x∈V_t

改写为U::= x (V | W)

更清楚表示

U::= x Z

Z::= V | W

注意：问题到此并没有结束，还需要进一步检查V和W的首符号是否相交

若V::= a b | c d　　FIRST(V) = { a , c }

W::= d e | f g　　FIRST(W) = { d , f }

只要不相交就可以根据输入符号确定目标；若相交，则要代入，并再次提取左因子。如：V::= a b　W::= a c

则：Z::= a (b | c)

例2: 文法G[<程序>]

<程序> ::= <分程序> | <复合语句>

<分程序>::= begin<说明串>; <语句串> end

<复合语句>::= begin<语句串> end

FIRST(<分程序>) = {begin }

FIRST(<复合语句>) = {begin }

改写文法:

<程序> ::= begin （<说明串>; <语句串> end |
<语句串> end）

引入 <程序*>

<程序> ::= begin <程序*>

<程序*> ::= <说明串>; <语句串> end | <语句串> end

<程序> ::= begin <程序*>
<程序*> ::= <说明串>; <语句串> end | <语句串> end

对于：<程序*>
FIRST(<说明串>; <语句串> end)
= {real, integer, boolean, array, function, procedure }
FIRST(<语句串>end)
= {标识符， goto, begin, if, for}
不相交。

2. 超前扫描

当文法不满足避免回溯的条件时，即各选择的首符号相交时，可以采用超前扫描的方法，即向前侦察各输入符号串的第二个、第三个符号来确定要选择的目标。

这种方法是通过向前多看几个符号来确定所选择的目标，从本质上来讲也有回溯的味道，因此比第一种方法费时，但是读的仅仅是向前侦察情况，不作任何语义处理工作。

例

<程序> ::= <分程序> | <复合语句>

<分程序>::= begin <说明串>; <语句串> end

<复合语句>::= begin <语句串> end

这两个选择的首符号是相交，因此读到begin时并不能确定该用哪个选择。这时可采用向前假读进行侦察，此例题只需假读一次就可以确定目标。

因为<说明串>的首符集为{real，integer，……，procedure}；而<语句串>的首符集为{标识符，if，for，……，begin}。

∴只要超前假读得到的是“说明串”的首符，便是第一个选择。若是“语句串”的首符，就是第二个选择。

文法的两个条件

为了在不采取超前扫描的前提下实现不带回溯的自顶向下分析，对文法需要满足两个条件：

1、文法是非左递归的；

2、对文法的任一非终结符，若其规则右部有多个选择时，各选择所推出的终结符号串的首符号集合要两两不相交。

在上述条件下，就可以根据文法构造有效的、不带回溯的自顶向下分析器。

4.2.3 递归子程序法（递归下降分析法）

具体做法：对语法的每一个非终结符都编一个分析程序。当根据文法和当时的输入符号预测到要用某个非终结符去匹配输入串时，就调用该非终结符的分析程序。

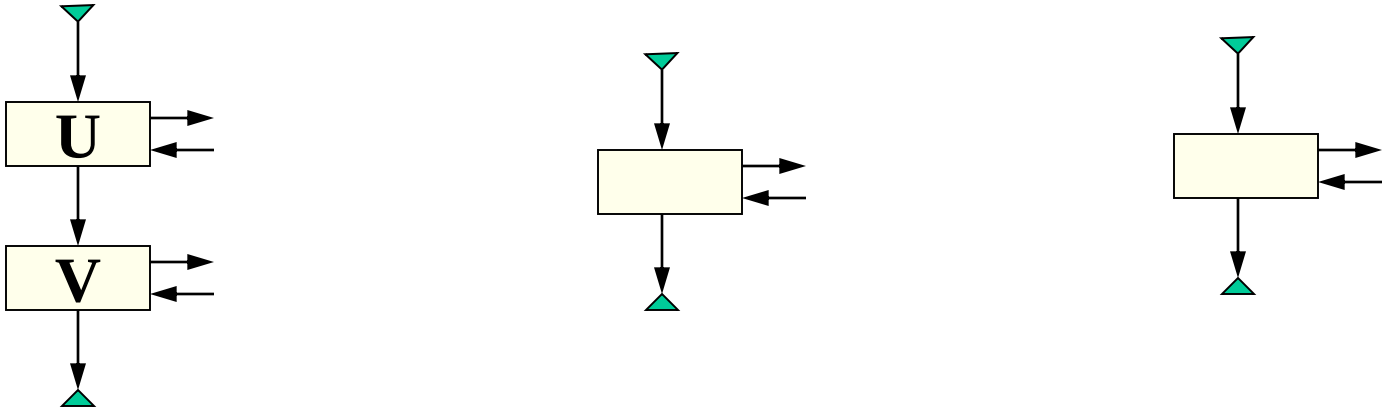
下面我们可以通过举例说明如何根据文法构造该文法的语法分析程序。

如文法G[E]: E ::= U V

U ::= ….

V ::= ….

E的分析程序 U的分析程序 V的分析程序



注：消除左递归后，可有其它递归（如右递归或自嵌入递归）：

U ::= …U… U ::= …W…

W::= …U…

例：文法G[Z]

Z::= (U) | a U b

U::= d Z | U d | e

注意：这两个式子中的圆括号不同！
Z中的括号是终结符号之一。
U中的括号则是元语言的括号。

1.检查并改写文法

Z::= (U) | a U b

U::= (d Z | e) { d }

改写后无左递归且首符集不相交:

{ (}∩{ a } =φ

{ d }∩{ e } =φ

2.检查文法的递归性

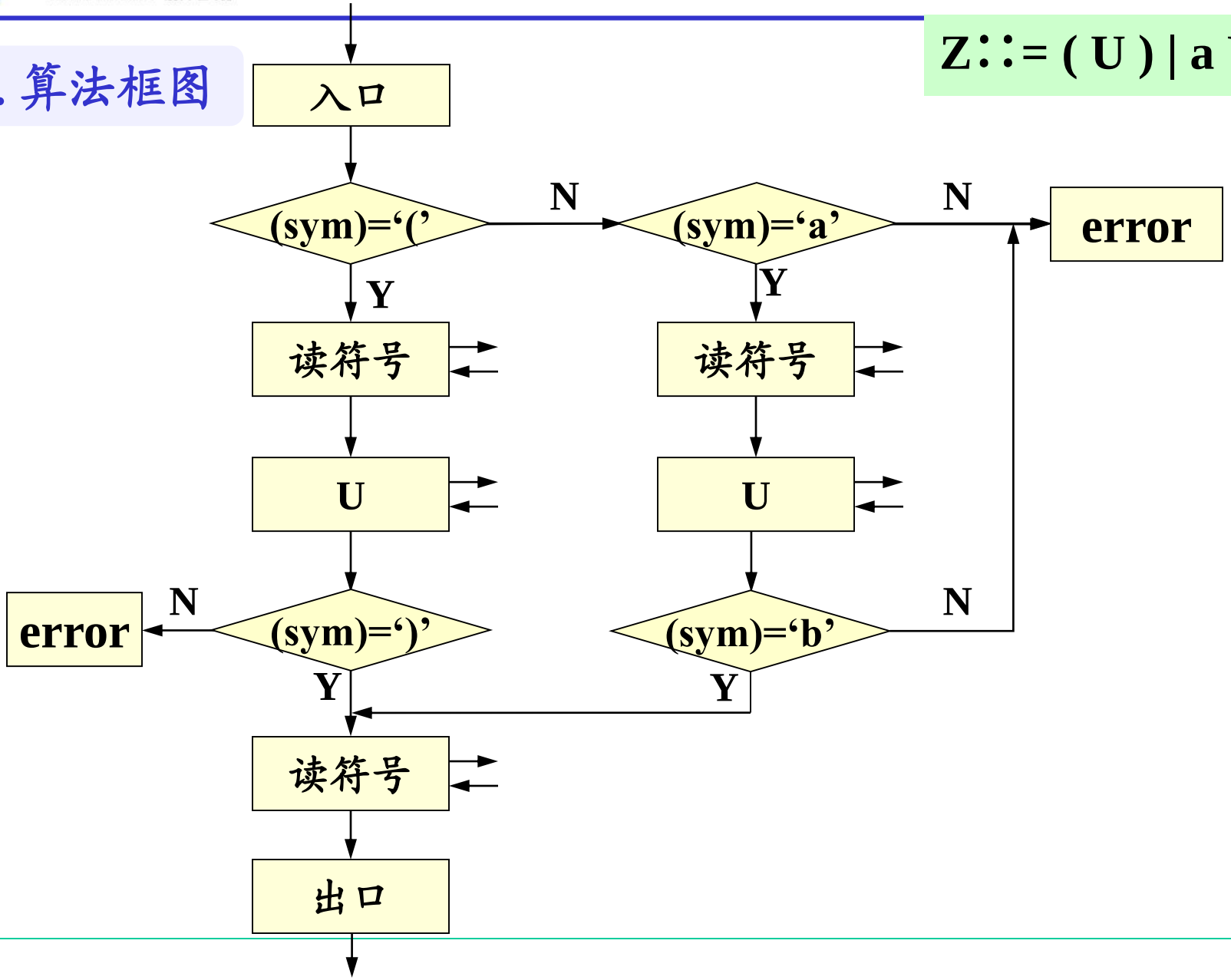
Z ⇒…U… ⇒ …Z…　　∴ Z $\overset{+}{\Rightarrow}$…Z…

U ⇒…Z… ⇒ …U…　　∴ U $\overset{+}{\Rightarrow}$…U…

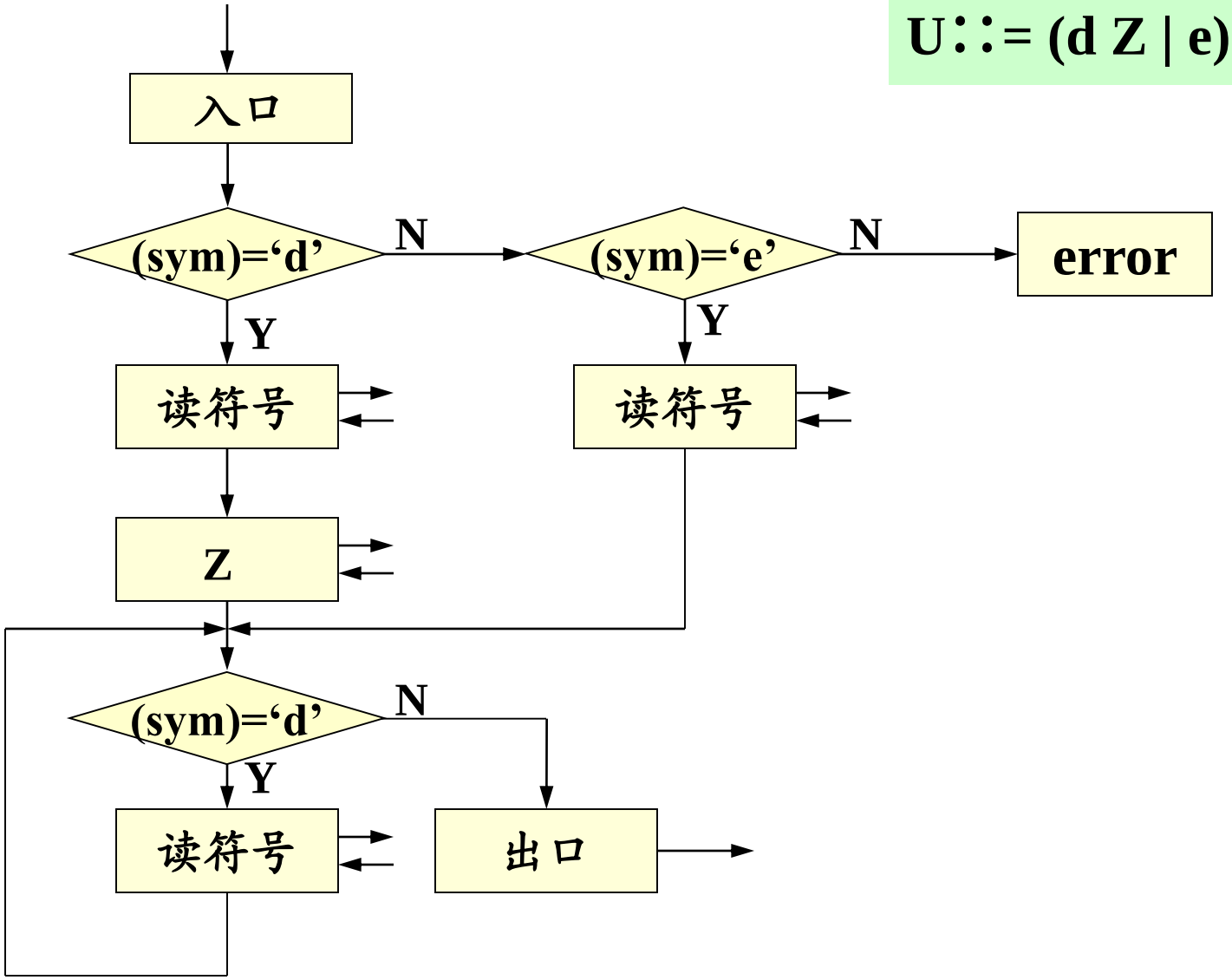
因此，Z和U的分析程序可以编成递归子程序

3. 算法框图

Z::= (U) | a U b



U∷= (d Z | e) { d }



说明：

- 非终结符号的分析子程序功能，是用规则右部符号串去匹配输入串。
- 要注意子程序之间的接口。在程序编制时，进入某个非终结符的分析程序前，其所要分析的语法成分的第一个符号已读入sym中了。
- 递归子程序法对应的是最左推导过程！
- 在上面的分析过程中，我们强调一定要消除左递归，但允许存在右递归或自嵌入递归。正是由于在子程序的调用过程中允许（直接或间接的）递归调用，这种方法才得名递归子程序法。

4.2.4 用递归子程序法构造语法分析程序的例子

文法：　<语句>::=<变量> := <表达式>
　　| IF <表达式> THEN <语句>
　　| IF <表达式> THEN <语句> ELSE <语句>
　　<变量>::= i | i ‘[‘<表达式>‘]‘
　　<表达式>::= <项> | <表达式> + <项>
　　<项>::= <因子> | <项> * <因子>
　　<因子>::= <变量> | ‘(‘<表达式>‘)‘

用单引号括起来的符号表示终结符号

改写文法：　<语句>::=<变量> := <表达式>
　　| IF <表达式> THEN <语句> [ELSE <语句>]
　　<变量>::= i [‘[‘<表达式>‘]‘]
　　<表达式>::= <项> { + <项> }
　　<项>::= <因子> { * <因子> }
　　<因子>::= <变量> | ‘(‘<表达式>‘)’

语法分析程序所要调用的子程序:

nextsym: 词法分析程序，每调用一次读进一个单词，单词的类别码放在sym中。

error: 出错处理程序。

<语句>::=<变量>：=<表达式>
| IF <表达式> THEN <语句> [ELSE <语句>]

```
PROCEDURE    state;                          /*语句*/
  IF  sym =‘IF’ THEN
     BEGIN   nextsym;  expr;
          IF  sym≠‘THEN’  THEN   error
          ELSE  BEGIN  nextsym; state; END

          IF  sym= ‘ELSE’ THEN
               BEGIN
                    nextsym;
                    state;
                END
     END
  ELSE  BEGIN   var;
               IF  sym≠‘：=’ THEN  error
               ELSE  BEGIN
                              nextsym;
                              expr;
                         END
          END
```

<变量>::= i ['['< 表达式 >']']

```
PROCEDURE     var;                                  /*变量*/
    IF  sym ≠'i'     THEN   error
        ELSE BEGIN  nextsym;
                    IF   sym = '['  THEN
                          BEGIN    nextsym;
                                   expr;
                                    IF  sym ≠']'  THEN   error
                                    ELSE    nextsym;
                           END
                END
```

<表达式>::= <项>{ + <项>}

```
PROCEDURE     expr;                                  /*表达式*/
    BEGIN    term;
        WHILE   sym = '+'   DO
                BEGIN  nextsym;
                       term;
                END
    END
```

<项>::=<因子>{*<因子>}
<因子>::=<变量>|‘(‘ <表达式> ‘)’

```
PROCEDURE     term;                          /*项*/
    BEGIN     factor;
        WHILE     sym = ‘*’  DO
                BEGIN  nextsym;
                       factor;
                END
    END
```

```
PROCEDURE     factor;                        /*因子*/
    BEGIN
        IF   sym=‘(’   THEN
             BEGIN  nextsym;  expr;
                 IF  sym ≠‘)’ THEN  error
                 ELSE   nextsym
             END
       ELSE var;
    END
```

举例分析

```
if ( i + i ) then i := i * i + i else
        i [ i ] := i + i [ i * i ] * ( i + i )
```

作业:

p85: 1

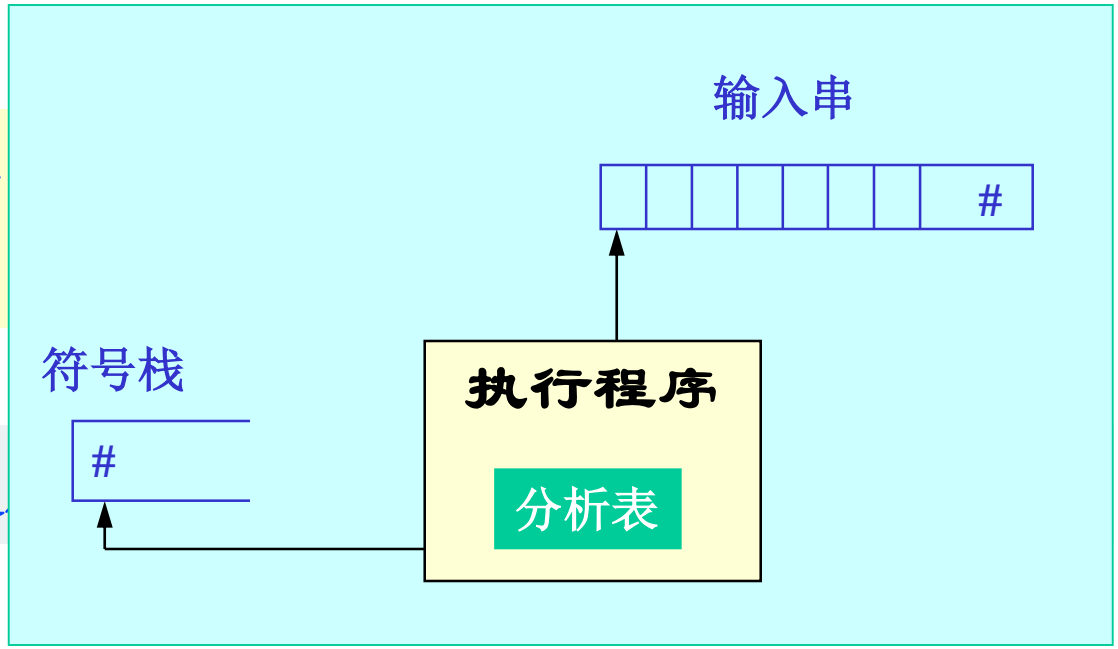
p90: 1-3

【注意：第3题第3个产生式应为B::=aB|a而不是B::=aA|a】

4.2.5 LL分析法

LL－自左向右扫描

∴ 分析过程表现为



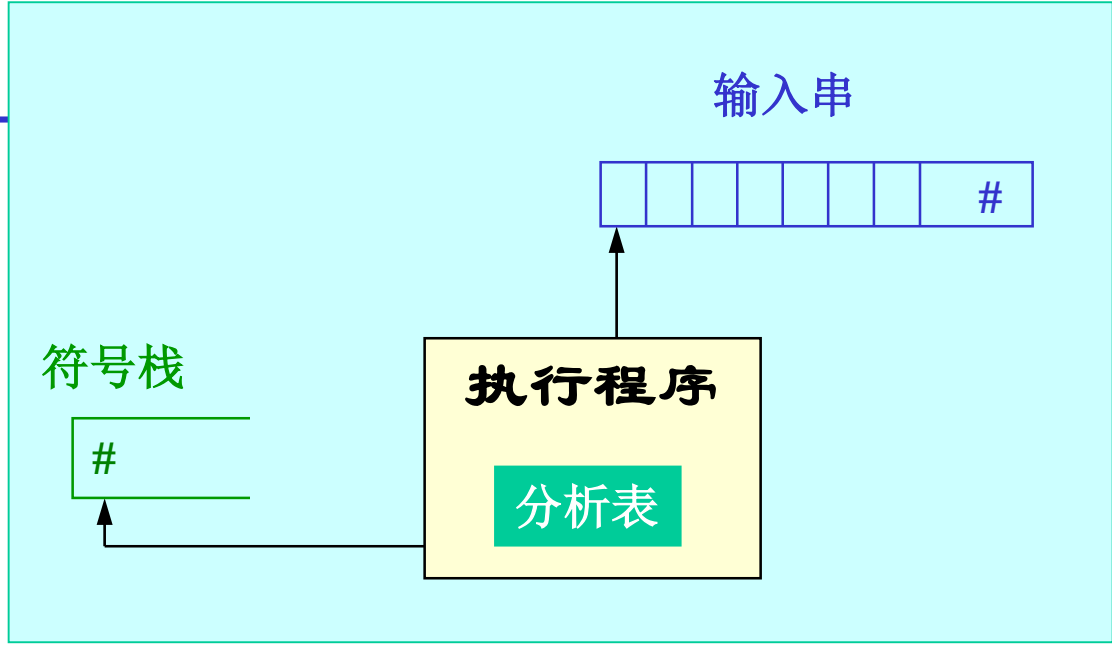
1、LL分析程序构造

此过程有三部分组成：分析表

执行程序 (总控程序）

符号栈（分析栈）

在实际语言中，每一种语法成分都有确定的左右界符，为了研究问题方便，统一用‘#’表示。



（1）分析表：二维矩阵M

$$M[A, a] = \begin{cases} A ::= \alpha_i \\ 或 \\ error \end{cases}$$

其中 $\alpha_i \in V^*$，$A \in V_n$，$a \in V_t$ or #

M[A, a]　A::=α_i

表示当要用A去匹配输入串时，且当前输入符号为 a 时，可用A的第 i 个选择去匹配。

即　当$\alpha_i \neq \varepsilon$时，有$\alpha_i \overset{*}{\Rightarrow} a\ldots$；

当$\alpha_i = \varepsilon$时，则 a 为A 的后继符号。

M[A, a]　error

表示当用A去匹配输入串时，若当前输入符号为 a，则不能匹配，表示无 $A \overset{*}{\Rightarrow} a\ldots$，或 a 不是A的后继符号。

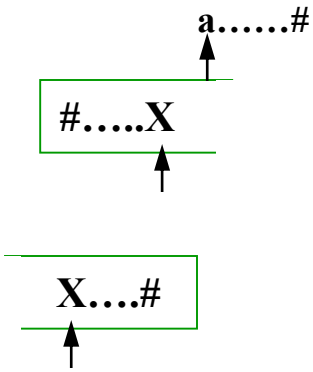
（2）符号栈： 有四种情况

- 开始状态

符号栈

- 工作状态

符号栈



查分析表得:

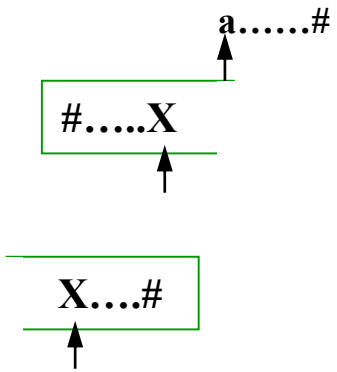
$X \in V_n$， $M[X, a] = X ::= \alpha_i$

$X \overset{+}{\Rightarrow} a\ldots$

$X \in V_t$， $X = a$

- 出错状态

符号栈



查分析表得:

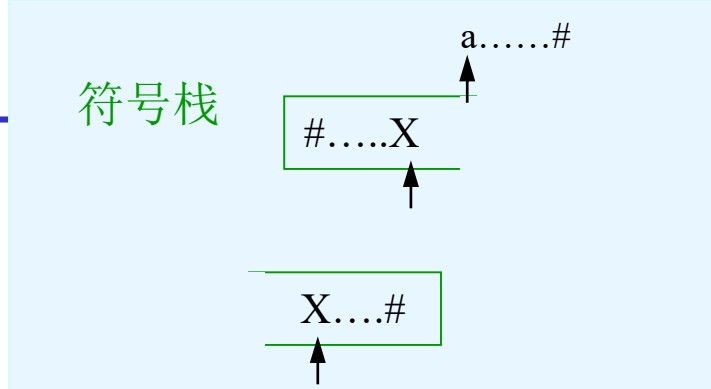
$X \in V_n$, $M[X, a] = error$

无 $X \overset{+}{\Rightarrow} a\ldots$

$X \in V_t$, $X \neq a$

- 结束状态

符号栈



（3）执行程序

执行程序主要实现如下操作：

1. 把 # 和文法识别符号 E 推进栈，并读入输入串的第一个符号 a，重复下述过程直到正常结束或出错。

2. 根据栈顶符号 X 和当前输入符号a，执行如下操作：

①若$X(\in V_t)=a=\#$，分析成功，停止。E匹配输入串成功。

②若$X(\in V_t)=a\neq\#$，把 X 退出栈，再读入下一个符号。

③若$X(\in V_t)\neq a$，转出错处理。

④若$X\in V_n$，查分析表 M。

④若X∈V_n，查分析表M。

a) M [X , a] = X∷= U V W

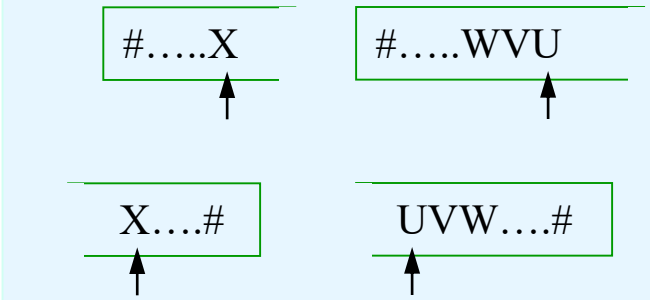
则将 X 弹出栈，将 U V W逆序入栈

注：U在栈顶（最左推导）

b) M [X , a] = error 转出错处理

c) M [X , a] = X:: =ε

a为X的后继符号，则将X弹出栈（不读下一符号）继续分析。



例：文法G [E]

E ::= E + T | T

T ::= T * F | F

F ::= (E) | i

消除左递归

E ::= T E’

E’ ::= + T E’ |ε

T ::= F T’

T’::= * F T’ |ε

F ::= (E) | i

分析表

	i	+	*	(	)	#
E	E::=TE'			E::=TE'		
E'		E'::=+TE'			E'::= ε	E'::= ε
T	T::=FT'			T::=FT'		
T'		T'::= ε	T'::=*FT'		T'::= ε	T'::= ε
F	F::= i			F::=(E)		

★ **注：矩阵元素空白表示Error**

	i	+	*	(	)	#
E	E ::= T E’			E ::= T E’		
E’		E’ ::= + T E’			E’ ::= ε	E’ ::= ε
T	T ::= F T’			T ::= F T’		
T’		T’ ::= ε	T’ ::= * F T’		T’ ::= ε	T’ ::= ε
F	F ::= i			F ::= (E)		

输入串为：i + i * i #

步骤	符号栈		读入符号	剩余符号串	使用规则
1.	# E	E #	i	+ i * i #	E ::= T E’
2.	# E’ T	T E’ #	i	+ i * i #	T ::= F T’
3.	# E’ T’ F	F T’ E’ #	i	+ i * i #	F ::= i
4.	# E’ T’ i	i T’ E’ #	i	+ i * i #	(出栈，读下一个符号)
5.	# E’ T’	T’ E’ #	+	i * i #	T’ ::= ε
6.	# E’	E’ #	+	i * i #	E’ ::= + T E’
7.	# E’ T +	+ T E’ #	+	i * i #	(出栈，读下一个符号)

	i	+	*	(	)	#
E	E ::= T E’			E ::= T E’		
E’		E’ ::= + T E’			E’ ::= ε	E’ ::= ε
T	T ::= F T’			T ::= F T’		
T’		T’ ::= ε	T’ ::= * F T’		T’ ::= ε	T’ ::= ε
F	F ::= i			F ::= (E)		

步骤	符号栈	读入符号	剩余符号串	使用规则
8.	# E’ T	i	* i #	T ::= F T’
9.	# E’ T’ F	i	* i #	F ::= i
10.	# E’ T’ i	i	* i #	(出栈，读下一个符号）
11.	# E’ T’	*	i #	T’ ::= * F T’
12.	# E’ T’ F *	*	i #	(出栈，读下一个符号）
13.	# E’ T’ F	i	#	F ::= i
14.	# E’ T’ i	i	#	(出栈，读下一个符号）
15.	# E’ T’	#		T’ ::= ε
16.	# E’	#		E’ ::= ε
17.	#	#		accept (分析成功）

推导过程:

$E \Rightarrow T\,E'$

$\Rightarrow F\,T'\,E'$

$\Rightarrow i\,T'\,E'$

$\Rightarrow i\,E'$

$\Rightarrow i + T\,E'$

$\Rightarrow i + F\,T'\,E'$

$\Rightarrow i + i\,T'\,E'$

$\Rightarrow i + i * F\,T'\,E'$

$\Rightarrow i + i * i\,T'\,E'$

$\Rightarrow i + i * i\,E'$

$\Rightarrow i + i * i$

最左推导!

已识别出来的终结符号和符号栈内的现有符号排列（从栈顶到栈底），正好构成该推导过程!

2、分析表的构造

设有文法G [Z]：

定义：$FIRST(\alpha) = \{ a \mid \alpha \overset{*}{\Rightarrow} a\ldots,\ a \in V_t \}$

$\alpha \in V^*$, 若$\alpha \overset{*}{\Rightarrow} \varepsilon$，则$\varepsilon \in FIRST(\alpha)$

该集合称为α的头符号集合。

定义：$FOLLOW(\alpha) = \{ a \mid Z \overset{*}{\Rightarrow} \ldots\alpha a\ldots,\ a \in V_t \}$

$\alpha \in V_n$ ， Z为识别符号

该集合称为α的后继符号集合。

特殊地：若$Z \Rightarrow \ldots\alpha$ 则 $\# \in FOLLOW(\alpha)$

① 构造集合FIRST的算法

设$\alpha = X_1 X_2 \ldots X_n$，　　$X_i \in V_n \cup V_t$

求FIRST(α)=?

首先求出组成α的每一个符号X_i的FIRST集合。

(1) 若$X_i \in V_t$，则 $FIRST(X_i) = \{X_i\}$

(2) 若$X_i \in V_n$ 且 $X_i ::= a\ldots|\varepsilon$，　$a \in V_t$

则　$FIRST(X_i) = \{a, \varepsilon\}$

(3) 若$X_i \in V_n$且$X_i ::= y_1 y_2 \ldots y_k$，则按如下顺序计算FIRST($X_i$)

- 若$\varepsilon \notin$ FIRST(y_1) 则将FIRST(y_1) 加入 FIRST(X_i)；
- 若$\varepsilon \in$ FIRST(y_1) 则将FIRST(y_2) – {ε}加入FIRST(X_i)

且若$\varepsilon \in$ FIRST(y_2) 则将FIRST(y_3) – {ε}加入FIRST(X_i)

……

$\varepsilon \in$ FIRST(y_{k-1}) 则将FIRST(y_k) – {ε}加入FIRST(X_i)

★注意：要顺序往下做，一旦不满足条件，过程就要中断进行

若$\varepsilon \in$FIRST(y_1) ~ FIRST(y_k) 则将ε加入FIRST(X_i)

得到FIRST(X_i)，即可求出FIRST(α)。

② 构造集合FOLLOW的算法

设$S, A, B \in V_n$，

算法：连续使用以下规则，直至FOLLOW集合不再扩大。

(1) 若S为识别符号，则把“#”加入FOLLOW(S)中；

(2) 若$A::=\alpha B\beta(\beta\neq\varepsilon)$，则把FIRST(β)-{ε}加入FOLLOW(B);

(3) 若$A::=\alpha B$ 或$A::=\alpha B\beta$， 且$\beta \overset{*}{\Rightarrow} \varepsilon$则把FOLLOW(A)加入FOLLOW(B) 中去。

注意：FOLLOW集合中不能有ε!!!

例：G [E] 分析表的构造

E::= T E’

E’::= + T E’ |ε

T::= F T’

T’::= * F T’ |ε

F::= (E) | i

求FIRST集

FIRST(F) = { (, i }

FIRST(T’) = { * , ε}

FIRST(T) = FIRST(F) - {ε} = { (, i }

FIRST(E’) = { + ,ε}

FIRST(E) = FIRST(T) - {ε} = { (, i }

∴FIRST(TE’) = FIRST(T) - {ε}= { (, i }

FIRST(+TE’) = {+}　　FIRST(ε) = {ε}

FIRST(FT’) = FIRST(F) - {ε} = { (, i }

FIRST((E)) = { (}　　FIRST(i) = { i }

E::= T E’
E’::= + T E’ |ε
T::= F T’
T’::= * F T’ |ε
F::= (E) | i

FIRST(F) = { (, i }
FIRST(T’) = { * , ε}
FIRST(T) = { (, i }
FIRST(E’) = { + ,ε}
FIRST(E) = { (, i }

求FOLLOW 集

FOLLOW(E) = { # ,) }　∵ E 是识别符号　∴#∈FOLLOW(E)

又∵F::= (E)　∴)∈FOLLOW(E)

FOLLOW(E’) = { # ,) }　∵E::= TE’　∴FOLLOW(E)加入FOLLOW(E’)

FOLLOW(T) = { + ,) , # }　∵E’::= + T E’　∴FIRST(E’)-{ε}加入FOLLOW(T)

又∵E’⇒ε，　∴ FOLLOW(E’)加入FOLLOW(T)

FOLLOW(T’) = FOLLOW(T) = { + ,) , # }

∵T::= F T’　∴ FOLLOW(T)加入FOLLOW(T’)

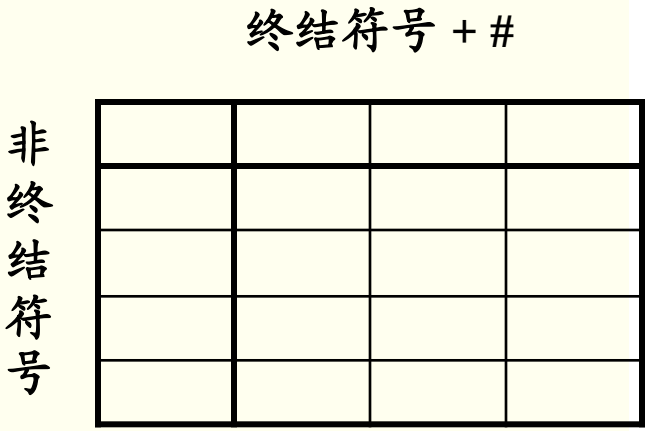
FOLLOW(F) = { * , + ,) , # }　∵T::= F T’　∴ FOLLOW(F) = FIRST(T’)-{ε}

又∵T’⇒ε ∴ FOLLOW(T)加入FOLLOW(F)

构造分析表

基本思想：

当文法中某一非终结符呈现在栈顶时，根据当前的输入符号，分析表应指示要用该非终结符里的哪一条规则去匹配输入串（即进行下一步最左推导）。



根据这个思想，我们不难把构造分析表算法构造出来！

算法:

设A::=α_i为文法中的任意一条规则，a为任一终结符或#。

1、若a∈FIRST(α_i)，则将 A ::=α_i 放入 M [A , a]。

表示：A在栈顶，输入符号是 a，应选择α_i 去匹配。

2、若α_i =ε或$\alpha_i \overset{+}{\Rightarrow} \varepsilon$，而且a∈FOLLOW ($\alpha_i$)，

则A::=α_i (即ε)放入M [A , a]，表示 A 已匹配输入串成功，

其后继符号终结符 a 由 A 后面的语法成分去匹配。

3、把所有无定义的M [A , a]都标上error。

构造分析表

E::= T E’ E’::= + T E’ |ε T::= F T’ T’::= * F T’ |ε F::= (E) | i	FIRST(TE’) = { (, i } FIRST(+TE’) = { + } FIRST(FT’) = { (, i } FIRST(*FT’) = { * } FIRST((E)) = { (}	FOLLOW(E) = { # ,) } FOLLOW(E’) = { # ,) } FOLLOW(T) = { + ,) , # } FOLLOW(T’) = { + ,) , # } FOLLOW(F) = { * , + ,) , # }

	i	+	*	(	)	#
E						
E’						
T						
T’						
F						

注意：用上述算法可以构造出任意给定文法的分析表，但不是所有文法都能构造出上述那种形状的分析表。即M [A , a] 只对应一条规则或Error。

对于能用上述算法构造分析表的文法我们称为LL(1)文法。

3、LL(1)文法

定义：一个文法G，其分析表M不含多重定义入口（即分析表中无两条以上规则），则称它是一个LL(1)文法。

定理：文法G是LL(1)文法的充分必要条件是：对于G的每个非终结符A的任意两条规则$A::=\alpha|\beta$,下列条件成立：

1、$FIRST(\alpha) \cap FIRST(\beta) = \Phi$

2、若$\beta \overset{*}{\Rightarrow} \varepsilon$，则$FIRST(\alpha) \cap FOLLOW(A) = \Phi$

用上述分析表的构造算法可以构造任何文法的分析表。但对于某些文法，有些M[A,a]中可能有若干条规则，这称为分析表的多重定义或者多重入口。

可以证明：如果G是左递归或二义的，那么M至少含有一个多重定义入口。因此，消除左递归和提取左因子将有助于获得无多重定义的分析表M。

反之，一个文法G的预测分析表M不含多重定义入口，当且仅当该文法为LL(1)的。

左递归：　U∷=U…| a …

则有：FIRST(U…)∩FIRST(a…)≠φ

∴ M[U,a] = { U∷=U…, U∷= a…}

二义文法：对文法所定义的某些句子存在着两(多)个最左推导，即在推导的某些步上存在多重定义，有两(多)条规则可用，所以分析表是多重定义的。

有些文法可以从非LL(1)文法改写为LL(1)文法。

但并不是所有的非LL(1)文法都能改写为LL(1)文法。

习题：p97页1，2，6（不需证明）

(注：第3题题目本身错误！)

选作题：有文法G[S]

S ::= i C t S S' | a

S'::= e S |ε

C ::= b

(1) 证明该文法是二义性文法。

(2) 求FIRST集和FOLLOW集

(3) 构造LL分析表，证明该文法是非LL(1)文法。

4、LL分析的错误恢复----补充

当符号栈顶的终结符和下一个输入符号不匹配，或栈顶是非终结符A，输入符号a，而M[A,a]为空白（即error）时，则分析发现错误。

错误恢复的基本思想是：跳过一些输入符号，直到期望的同步符号之一出现为止。

同步符号（可重新开始继续分析的输入符号）集合通常可按以下方法确定：

1) 把FOLLOW(A)的所有符号加入A的同步符号集。如果我们跳读一些输入符号直到出现FOLLOW(A)的符号，之后把A从栈中弹出，继续往下分析即可。

2) 只用FOLLOW(A)作为非终结符A的同步符号集是不够的(容易造成跳读过多，如输入串中缺少语句结束符分号时)。此时可将作为语句开头的关键字加入它的同步符号集，从而避免这种情况的发生。

3) 把FIRST(A)的符号加入非终结符A的同步符号集中。

4) 如果非终结符A可以产生空串，那么推导ε的产生式可以作为缺省的情况。这样做可以推迟某些错误检查，但不会漏过错误。

5) 如果终结符在栈顶而不能匹配，则可弹出该终结符并发出一条信息后继续分析。这好比把所有其他符号均作为该符号的同步集合元素。

4.3 自底向上分析

基本算法思想:

若采用自左向右地扫描和分析输入串，那么自底向上的基本算法是:

从输入符号串开始，通过反复查找当前句型的句柄（最左简单短语），并利用有关规则进行规约。若能规约为文法的识别符号，则表示分析成功，输入符号串是文法的合法句子；否则有语法错误。

分析过程是重复以下步骤：

1、找出当前句型的句柄 x（或句柄的变形）；

2、找出以 x 为右部的规则 X::= x；

3、把 x 规约为X，产生语法树的一枝。

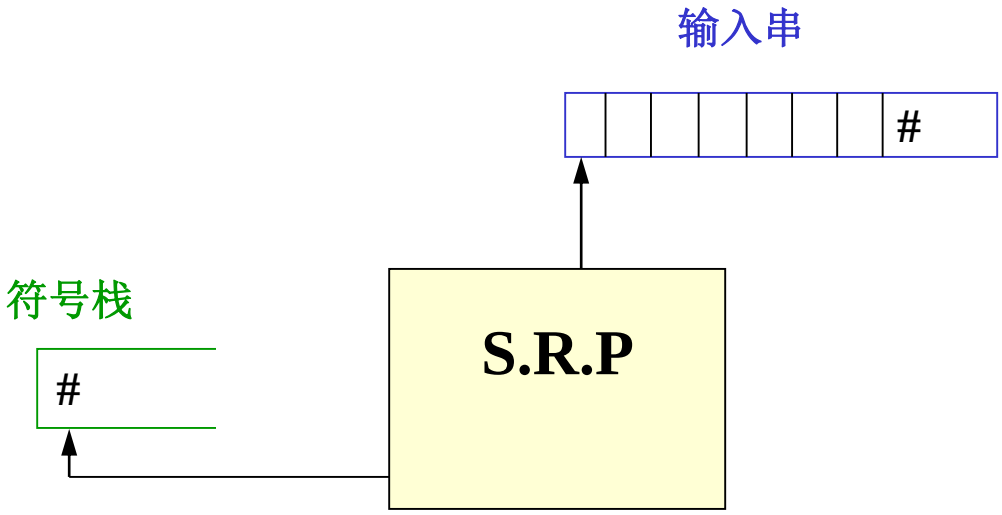
关键：找出当前句型的句柄 x (或其变形)， 这不是很容易的。

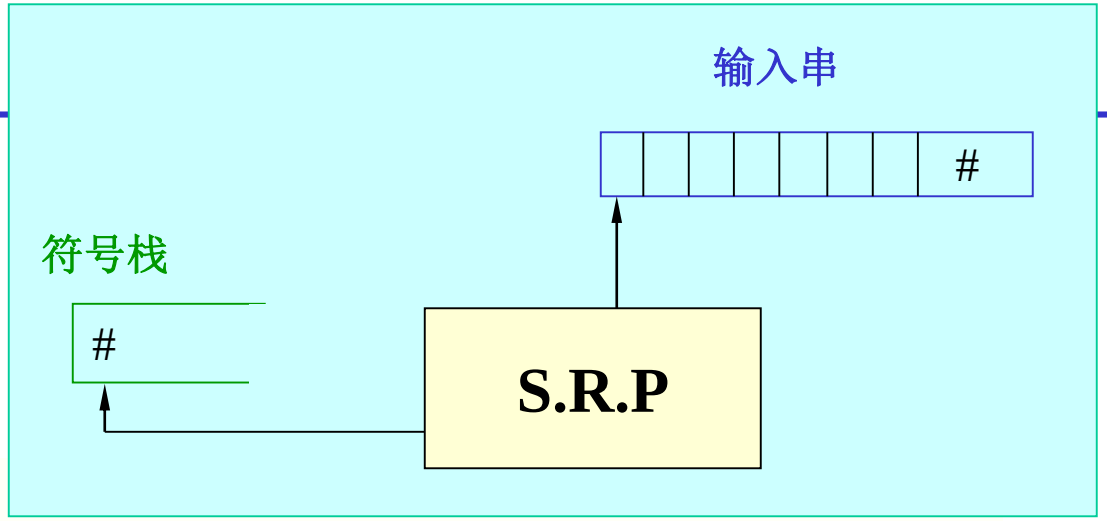
本节内容介绍（全书的重点！）

- 自底向上分析的一般过程（移进-归约分析）
- 算符优先分析法
- LR分析法
  - LR(0)分析法
  - SLR(1)分析法
  - 规范LR(1)分析法
  - LALR(1)分析法

4.3.1 移进—规约分析（Shift-reduce parsing）

要点：设置符号栈，用来纪录分析的历史和现状，并根据所面临的状态，确定下一步动作是移进还是规约。





分析过程:

1. 把输入符号串按顺序一个一个地移进符号栈（一次移一个）；
2. 检查栈中符号，当在栈顶的若干符号形成当前句型的句柄时，就根据规则进行规约——将句柄从符号栈中弹出，并将相应的非终结符号压入栈内（即规则的左部符号），然后再检查栈内符号串是否形成新的句柄，若有就再进行规约，否则移进符号；
3. 分析一直进行到读到输入串的右界符为止。最后，若栈中仅含有左界符号和识别符号，则表示分析成功，否则失败。

例：G[S]:　　输入串为：a b b c d e

S::= a A c B e

A::= b

A::= A b

B::= d

检查输入串abbcde是否是该文法的合法句子。

若采用自底向上分析，即能否一步步规约当前句型的句柄，最终规约到识别符号S。先设立一个符号栈，我们仍然统一用符号“#”作为待分析符号串的左右分界符。

作为初始状态，先将符号串的左分界符推进符号栈作为栈底符号。

分析过程如下表:

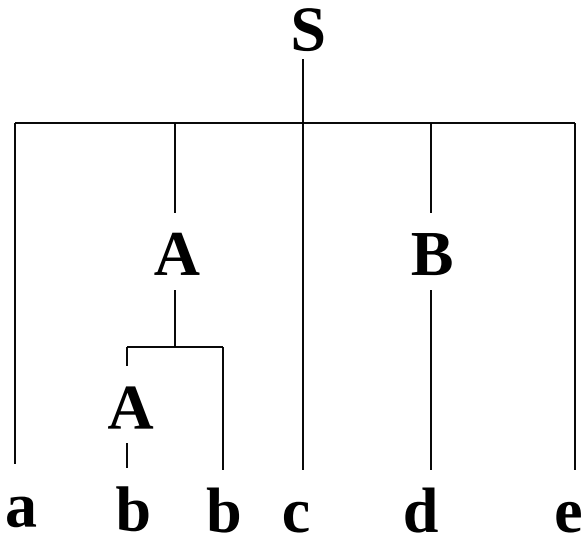
例：G[S]：　S∷= a A c B e　　A∷= b
　　A∷= A b　　B∷= d

步骤	符号栈	输入符号串	动作
1	#	abbcde#	准备。初始化
2	#a	bbcde#	移进
3	#ab	bcde#	移进
4	#aA	bcde#	规约(A∷=b)
5	#aAb	cde#	移进
6	#aA	cde#	规约(A∷=Ab)
7	#aAc	de#	移进
8	#aAcd	e#	移进
9	#aAcB	e#	规约(B∷=d)
10	#aAcBe	#	移进
11	#S	#	规约(S∷=aAcBe)
12	#S	#	成功

这一方法简单明了，不断地进行移进规约。关键是确定当前句型的句柄。

说明： 1) 例子的分析过程是一步步地规约当前句型的句柄。

该句子的唯一语法树为：



注意两点：

① 栈内符号串 + 未处理输入符号串 = 当前句型

② 句柄都在栈顶

实际上，以上分析过程并未真正解决句柄的识别问题。

2) 未真正解决句柄的识别问题

上述分析过程中识别句柄的过程，主要看栈顶符号串是否形成规则的右部。

这种做法形式上是正确的，但在实际上不一定正确。举例的分析过程可以说是一种巧合。

因为不能认为：对句型 **xuy** 而言，若有**U::= u**，即 **U ⇒ u** 就断定**u**是简单短语，**u** 就是句柄，而是要同时满足 **Z** $\overset{*}{\Rightarrow}$ **xUy**。

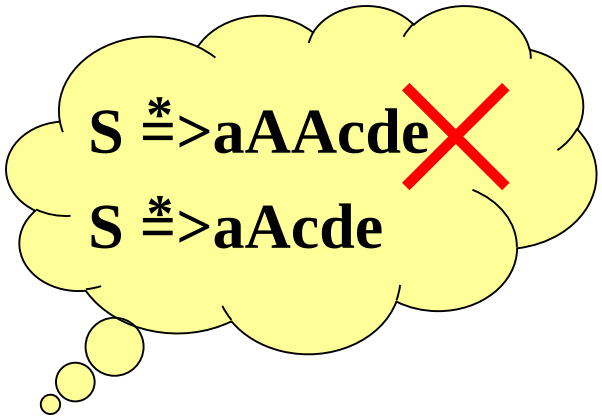
步骤	符号栈	输入符号串	当前句型
5	#aAb	cde#	aAbcde

A::= b　　∴ A => b

A::= Ab　　∴ A => Ab

若用A::= b 归约，　得aAAcde

若用A::= Ab归约，得aAcde



4.3.2 算符优先分析(Operator-Precedence Parsing)

1) 这是一种经典的自底向上分析法，简单直观，并被广泛使用。开始主要是对表达式的分析，现在已不限于此，可以用于一大类上下文无关文法。

2) 称为算符优先分析是因为这种方法是仿效算术式的四则运算而建立起来的。做算术式的四则运算时，为了保证计算结果和过程的唯一性，规定了一个统一的四则运算法则，确定运算符之间的优先关系。

运算法则：

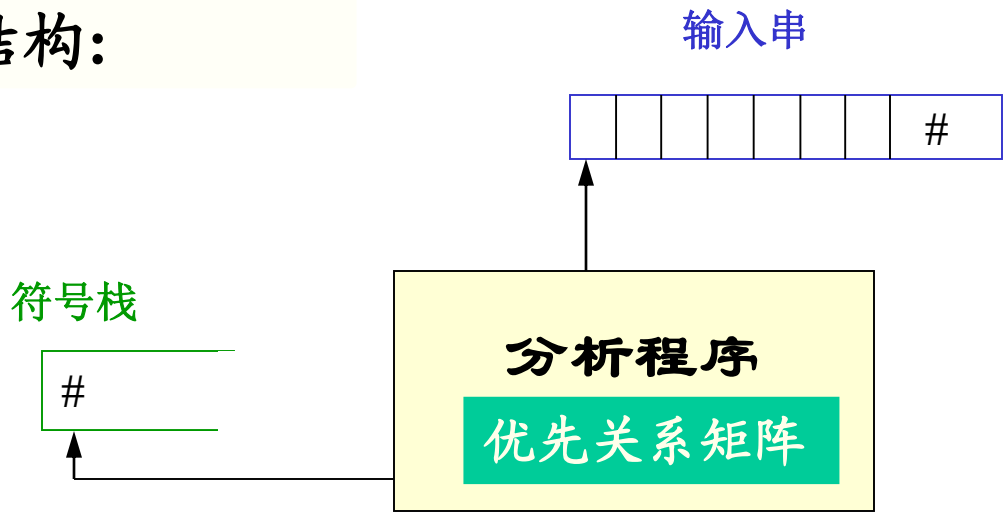
1.乘除的优先级大于加减；
2.同优先级的运算符左大于右；
3.括号内的优先级大于括号外。

于是： 4 + 8 – 6 / 2 * 3 运算过程和结果唯一。

3) 算符优先分析的特点：

仿效四则运算过程，预先规定相邻终结符之间的优先关系，然后利用这种优先关系来确定句型的“句柄”，并进行规约。

4) 分析器结构：



例： G[E]

E::= E+E|E*E|(E)|i

V_t={+,*,(,),i}

这是一个二义文法，要用算符优先法分析由该文法所确定的语言句子。 如：i+i*i

(1) 先确定终结符之间的优先关系

1) 优先关系的定义:

设 a，b 为可能相邻的终结符

定义： a ≐ b — a的优先级等于b

a ⋖ b — a的优先级小于b

a ⋗ b — a的优先级大于b

优先关系矩阵

a \ b	+	*	i	(	)	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(	<	<	<	<	=	
)	>	>			>	>
#	<	<	<	<		

2) 矩阵元素空白处表示这两个终结符不能相邻，故没有优先关系。

(2) 分析过程　　i + i * i

算法：

当栈顶项（或次栈顶项）终结符的优先级大于栈外的终结符的优先级则进行规约，否则移进。

$E ::= E + E \mid E * E \mid (E) \mid i$

a \ b	+	*	i	(	)	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(	<	<	<	<	=	
)	>	>			>	>
#	<	<	<	<		

步骤	符号栈	输入串	优先关系	动作
1	#	i + i * i #	# < i	移进
2	# i	+ i * i #	i > +	规约
3	# E	+ i * i #	# < +	移进
4	# E +	i * i #	+ < i	移进
5	# E + i	* i #	i > *	规约
6	# E + E	* i #	+ < *	移进
7	# E + E *	i #	* < i	移进
8	# E + E * i	#	i > #	规约
9	# E + E * E	#	* > #	规约
10	# E + E	#	+ > #	规约
11	# E	#		接受

分析过程是从符号串开始，根据相邻终结符之间的优先关系确定句型的“句柄”并进行规约，直到识别符号E。最后分析成功：　i + i * i ∈L(G[E])

出错情况:

1. 相邻终结符之间无优先关系。
2. 对双目运算符进行规约时，符号栈中无足够项。
3. 非正常结束状态。

重要说明

(1) 上述分析过程不一定是严格的最左规约（即不一定是规范规约）。也就是每次规约不一定是规约当前句型的句柄，而是句柄的变形，但也是短语。

(2) 实际应用中，文法终结符间优先关系一般不用矩阵表示，而是采用两个优先函数来表示：

f — 栈内优先函数

g — 栈外优先函数

若 $a \lessdot b$ 则令 $f(a) < g(b)$

$a \doteq b$ $f(a) = g(b)$

$a \gtrdot b$ $f(a) > g(b)$

根据这些原则，构造出上述文法的优先函数：

算符优先函数值的确定方法

① 由定义直接构造优先函数

② 用关系图构造优先函数

具体方法不要求！

	+	*	(	)	i	#
f (栈内)	2	4	0	5	5	0
g (栈外)	1	3	6	0	6	0

	+	*	(	)	i	#
f (栈内)	2	4	0	5	5	0
g (栈外)	1	3	6	0	6	0

f (+) > g (+)　　左结合

f (+) < g (*)　　先乘后加

f (+) < g (( )　　先括号内后括号外

⋮

特点：

① 优先函数值不唯一。

② 优点：

- 节省内存空间。

  若文法有n个终结符，则关系矩阵为n^2，而优先函数为2n。

- 易于比较：算法上容易实现。数与数比，不必查矩阵。

③ 缺点：可能掩盖错误。

(3) 可以设立两个栈来代替一个栈

运算对象栈 (OPND)　　运算符栈 (OPTR)

好处是：便于比较，只需将输入符号与运算符栈的栈顶符号相比较。

(4) 使用算符优先分析方法可以分析二义性文法所产生的语言

二义性文法若按规范分析，其句柄不唯一。

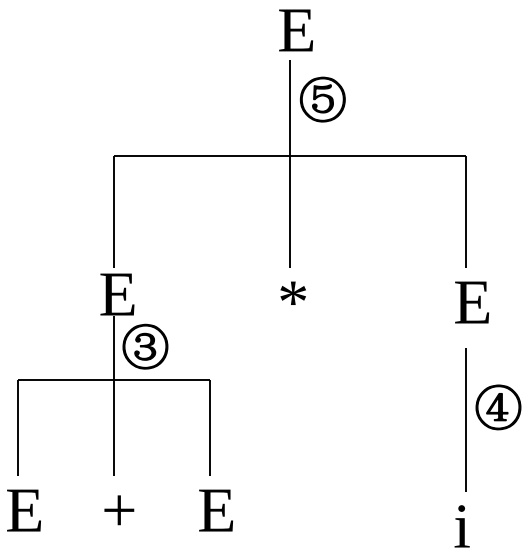
例：G[E]

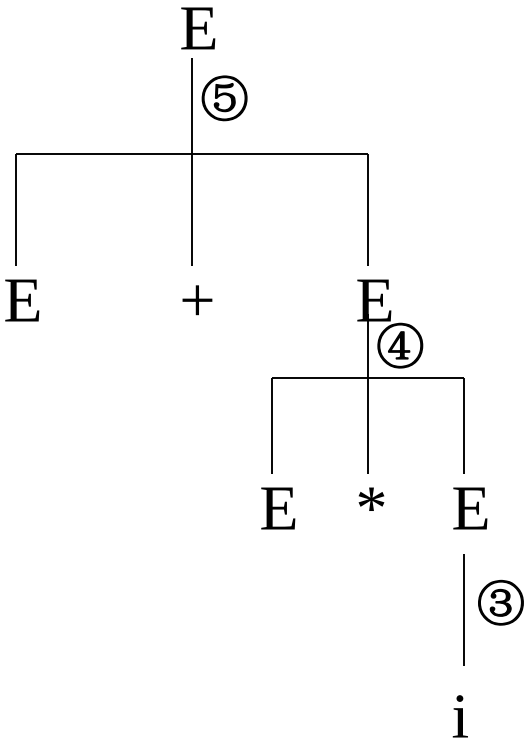
E::= E + E | E * E | (E) | i

V_t = { + , * , (,) , i }

这是一个二义性文法，

i + i * i 有两棵语法树。





如对句型E + E * i，规范规约时句柄不唯一，所以整个规约过程就不唯一，编译所得的结果也将不唯一。

4.3.3 算符优先分析法的进一步讨论

三个问题:

(1) 算符优先文法(OPG)

(2) 构造优先关系矩阵

(3) 算符优先分析算法的设计

（1）算符优先文法（OPG—Operator Precedence Grammar）

算符文法（OG）的定义

若文法中无形如U::= …VW…的规则，这里V, W∈V_n

则称G为OG文法，也就是算符文法。

算符文法不允许两个非终结符相邻！

优先关系的定义

若G是一个OG文法，$a, b \in V_t$，　$U, V, W \in V_n$

分别有以下三种情况:

1) $a \doteq b$　iff 文法中有形如 U::=…ab…或U::=…aVb…的规则。

2) $a \lessdot b$　iff 文法中有形如 U::=…aW…的规则，其中 $W \overset{+}{\Rightarrow} b\ldots$或$W \overset{+}{\Rightarrow} Vb\ldots$ 。

3) $a \gtrdot b$　iff 文法中有形如 U::=…Wb…的规则，其中 $W \overset{+}{\Rightarrow} \ldots a$或$W \overset{+}{\Rightarrow} \ldots aV$ 。

例： 文法G [E]

E ::= E + T | T

T ::= T * F | F

F ::= (E) | i

1) $a \doteq b$ iff 文法中有形如 U::=…ab…或U::=…aVb…的规则。

2) $a \lessdot b$ iff 文法中有形如 U::=…aW…的规则，其中 $W \overset{+}{\Rightarrow} b$…或$W \overset{+}{\Rightarrow} Vb$… 。

3) $a \gtrdot b$ iff 文法中有形如 U::=…Wb…的规则，其中 $W \overset{+}{\Rightarrow}$…a或$W \overset{+}{\Rightarrow}$…aV 。

对于E::= E + T

∵ E => E + T ∴ + ⋗ +

∵ T => T * F ∴ + ⋖ *

∵ T => F => (E) ∴ + ⋖ (

∵ T => F => i ∴ + ⋖ i

对于F::= (E)

∵ E => E + T ∴ + ⋗)

∴ (≐)

∴ (⋖ +

算符优先文法（OPG—Operator Precedence Grammar）的定义

设有一OG文法，如果在任意两个终结符之间，至多只有上述关系中的一种，则称该文法为算符优先文法(OPG)。

对于OG算法的几点说明:

① 运算是以中缀形式出现的。

② 可以证明，若文法为OG文法，则不会出现两个非终结符相邻的句型。

③ 算法语言中的表达式以及大部分语言成分的文法均是OG文法。

（2）构造优先关系矩阵

- 求“ $\doteq$ ”检查每一条规则，若有U::=…ab… 或 U::=…aVb…, 则 a$\doteq$b。

- 求“ $<\cdot$ ”、“ $\cdot>$ ”复杂一些，需定义两个集合:

$$FIRSTVT(U) = \{ b \mid U \overset{+}{\Rightarrow} b\ldots \text{或} U \overset{+}{\Rightarrow} Vb\ldots, b \in V_t, V \in V_n \}$$

$$LASTVT(U) = \{ a \mid U \overset{+}{\Rightarrow} \ldots a \text{ 或} U \overset{+}{\Rightarrow} \ldots aV, a \in V_t, V \in V_n \}$$

- 求 “ $\lessdot$ ”、 “ $\gtrdot$ ”:

若文法有规则

W∷= ...a U... ，对任何b，b∈FIRSTVT(U)

则有： a $\lessdot$ b

若文法有规则

W∷=...U b... ，对任何a，a∈LASTVT(U)

则有： a $\gtrdot$ b

构造FIRSTVT(U)的算法

1)若有规则U::= b…或U::= V b… 则b∈FIRSTVT(U)
（FIRSTVT的定义中一步推导）

2)若有规则U::= V…且 b∈FIRSTVT(V), 则b∈FIRSTVT(U)
（FIRSTVT的定义中多步推导）

说明：因为$V \overset{+}{\Rightarrow} b\ldots$ 或 $V \overset{+}{\Rightarrow} W\ b\ldots$,

所以有$U \Rightarrow V\ldots \overset{+}{\Rightarrow} b\ldots$ 或 $U \Rightarrow V\ldots \overset{+}{\Rightarrow} W\ b\ldots$

具体程序如下：

设一个栈S和一个二维布尔数组F

F[U, b] = TRUE　　iff　b∈FIRSTVT(U)

```
PROCEDURE    INSERT(U, b)
        IF  NOT  F[U, b]   THEN
                BEGIN
                    F[U, b] := TRUE;
                    把(U, b)推进S栈          /* b∈FIRSTVT(U) */
                END
```

```
main( )
BEGIN
    FOR 每个非终结符号U和终结符b DO
        F[U, b] := FALSE;          /*赋初值*/

    FOR 每个形如 U ::= b…或U ::= Vb… 的规则 DO
        INSERT(U, b);              /*完成算法的第1条*/

    WHILE S栈非空 DO
    BEGIN
        把S栈的顶项弹出，记为 (V, b)        /* b∈FIRSTVT(V)*/
        FOR 每个形如U ::= V…的规则 DO
            INSERT (U, b);                  /* b∈FIRSTVT(U)*/
    END OF WHILE                   /*完成算法的第2条*/
END
```

上述算法的工作结果是得到一个二维的布尔数组F，

从F可以得到任何非终结符号 U 的FIRSTVT:

FIRSTVT(U) = { b | F[U, b] = TRUE }

构造LASTVT(U)的算法

1.若有规则U::=…a 或 U::=…aV，则a∈LASTVT(U)

2.若有规则U::=…V，且a∈LASTVT(V) 则 a∈LASTVT(U)

设一个栈 ST 和一个布尔数组 B

```
PROCEDURE    INSERT(U, a)
        IF  NOT  B[U, a]    THEN
            BEGIN
                B[U, a] ::= TRUE;
                把(U, a)推进ST栈;
            END
```

主程序:

```
BEGIN
    FOR 每个非终结符号U和终结符号a  DO
            B[U, a] := FALSE;

    FOR  每个形如U::=…a 或 U::=…aV 的规则  DO
            INSERT (U, a);

    WHILE  ST栈非空  DO
        BEGIN
          把ST栈的栈顶弹出，记为(V, a);
          FOR 每条形如U::=…V的规则  DO
              INSERT(U, a);
        END  OF  WHILE
END
```

构造优先关系矩阵的算法

FOR 每条规则U::= $x_1 x_2 \ldots x_n$ DO
  FOR i:=1 TO n-1 DO
    BEGIN
      IF x_i和x_{i+1}均为终结符, THEN 置

      IF $i \leq n-2$，且x_i和x_{i+2}都为终结符号但
      x_{i+1}为非终结符号 THEN 置

      IF x_i为终结符号x_{i+1}为非终结符号 THEN
      FOR FIRSTVT(x_{i+1})中的每个b DO
        置

      IF x_i为非终结符号x_{i+1}为终结符号 THEN
      FOR LASTVT(x_i)中的每个a DO
        置
    END

例：文法G [E]　　E ::= E + T | T　　T ::= T * F | F　　F ::= (E) | i

FIRSTVT

	+	*	i	(	)
E	✓	✓	✓	✓	
T		✓	✓	✓	
F			✓	✓	

LASTVT

	+	*	i	(	)
E	✓	✓	✓		✓
T		✓	✓		✓
F			✓		✓

算符优先矩阵

	+	*	i	(	)
+	>	<	<	<	>
*	>	>	<	<	>
i	>	>			>
(	<	<	<	<	=
)	>	>			>

例：文法G [E]　　E∷= E + E　　E∷= E * E　　E∷= (E)　　E∷= i

FIRSTVT

	+	*	i	(	)
E	✓	✓	✓	✓	

LASTVT

	+	*	i	(	)
E	✓	✓	✓		✓

冲突

算符优先矩阵

	+	*	i	(	)
+	< >	< >	<	<	>
*	< >	< >	<	<	>
i	>	>			>
(	<	<	<	<	=
)	>	>			>

（3）算符优先分析算法的实现

先定义优先级，在分析过程中通过比较相邻运算符之间的优先级来确定句型的“句柄”并进行归约。

？—— 最左素短语

[定义] 素短语：文法G的句型的素短语是一个短语，它至少包含有一个终结符号，并且除它自身以外不再包含其它素短语。

例：文法G[E]

E ::= E + T | T

T ::= T * F | F

F ::= (E) | i

求句型 T + T * F + i 的最左素短语。

从文法的语法树中可以找出以下短语：

① T + T * F + i ×　包含其它短语

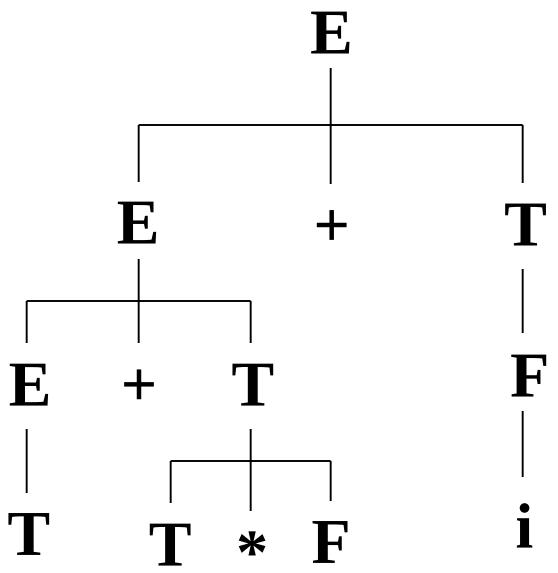
② T + T * F ×　包含其它短语

③ T ×　不包含终结符

④ T * F √　是素短语

⑤ i √　是素短语

文法的语法树



T * F 为最左素短语！

而该句型的句柄为T。

更进一步，对句型：T + T + i

短语：T + T + i

T + T

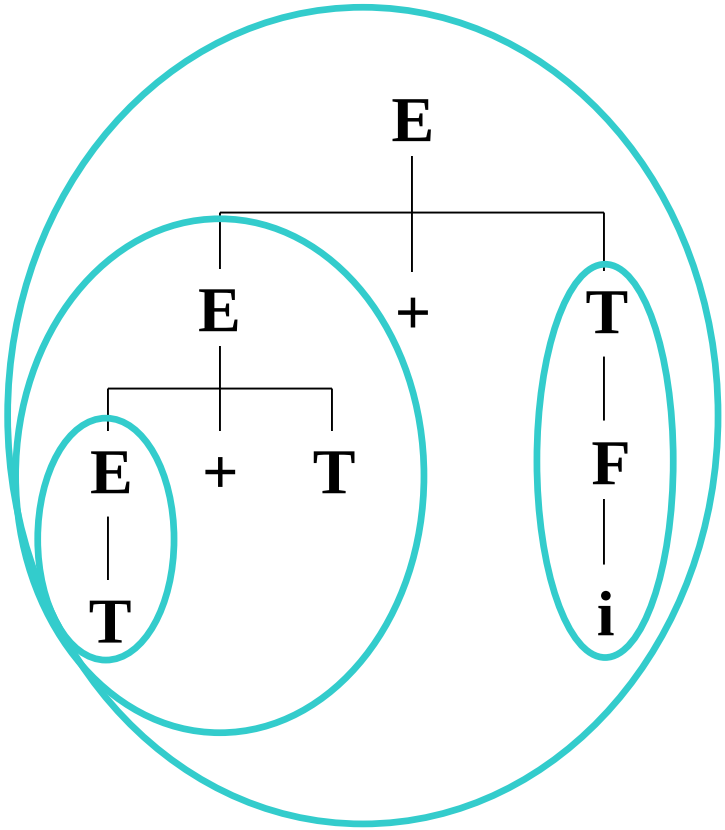
T

i

句柄：T

素短语：T + T，i

最左素短语：T + T



对于算符优先分析法：如何确定当前句型的最左素短语？

设有OPG文法句型为：

$$\#\, N_1\, a_1\, N_2\, a_2 \ldots N_n\, a_n\, N_{n+1}\, \#$$

其中N_i为非终结符(可以为空)，a_i为终结符。

定理：一个OPG句型的最左素短语是满足下列条件的最左子串：

$$a_{j-1}\, N_j\, a_j \ldots N_i\, a_i\, N_{i+1}\, a_{i+1}$$

其中 $a_{j-1} \lessdot a_j$，$a_j \doteq a_{j+1}$，$a_{j+1} \doteq a_{j+2}$，…，$a_{i-2} \doteq a_{i-1}$，$a_{i-1} \doteq a_i$，$a_i \gtrdot a_{i+1}$

根据该定理，要找句型的最左素短语就是要找满足上述条件的最左子串。

$N_j a_j \dots N_i a_i N_{i+1}$

★ 注意：出现在 a_j 左端和 a_i 右端的非终结符号一定属于这个素短语，因为我们的运算是中缀形式给出的（OPG文法的特点）

$$N\,a\,N\,a\,N\,a\,N \Rightarrow N\,a\,W\,a\,N$$

例：文法G [E]

E ::= E + T | T

T ::= T * F | F

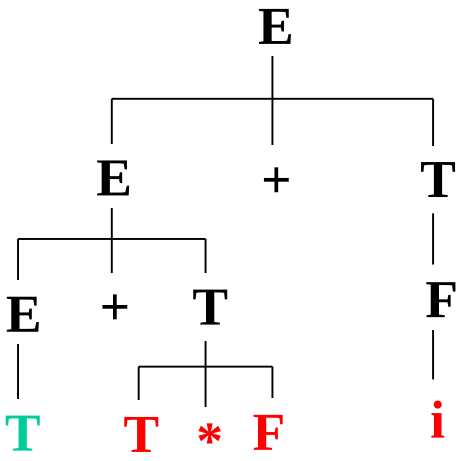
F ::= (E) | i

分析文法的句型 T + T * F + i

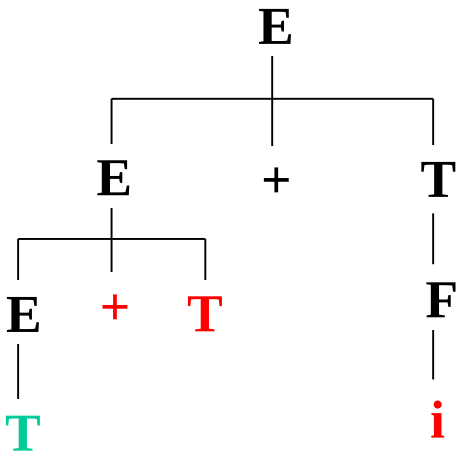
步骤	句型	关系	最左子串	规约符号
1	#T+T*F+i#	#<+<*>+<i>#	T*F	T
2	#T+T+i#	#<+>+<i>#	T+T	E
3	#E+i#	#<+<i>#	i	F
4	#E+F#	#<+>#	E+F	E

可以看出:

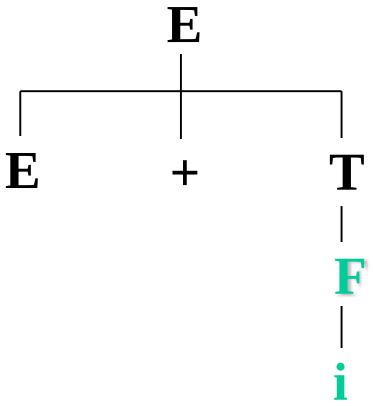
1. 每次规约的最左子串，确实是当前句型的最左素短语(从语法树可以看出来)
2. 规约的不都是真正的句柄(仅 i 规约为 F 是句柄，但它同时也是最左素短语)
3. 没有完全按规则进行规约，因为素短语不一定是简单短语



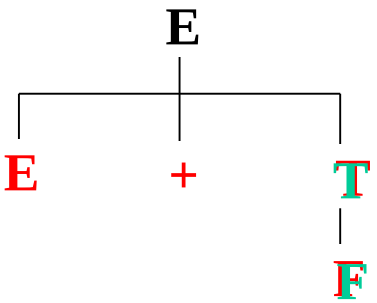
⋖ + ⋖ * ⋗ + ⋖ i ⋗



⋖ + ⋗ + ⋖ i ⋗



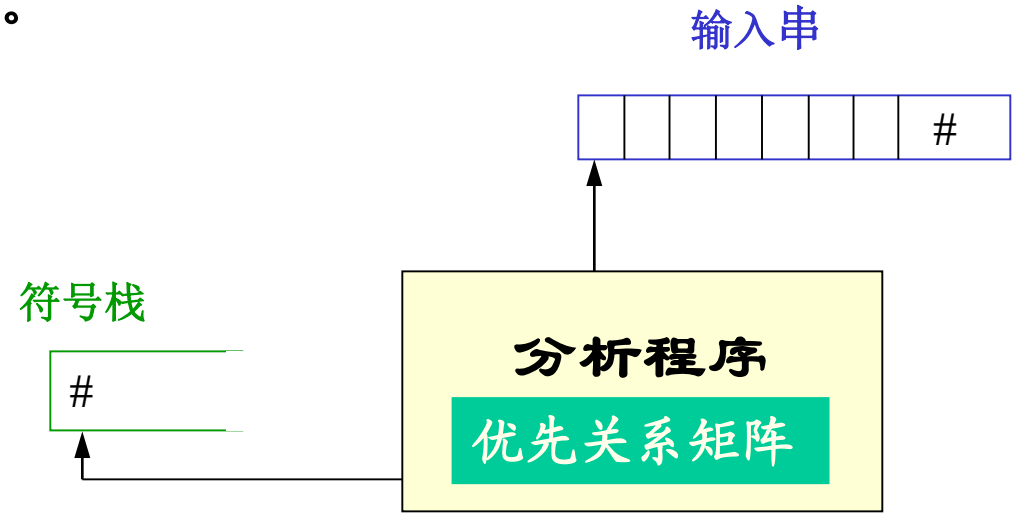
⋖ + ⋖ i ⋗



⋖ + ⋗

算符优先分析法的实现:

基本部分是找句型的最左子串（最左素短语）并进行规约。



当栈内终结符的优先级＜或＝栈外终结符的优先级时，移进;
当栈内终结符的优先级＞栈外终结符的优先级时，表明找到了素短语的尾，再往前找其头，并进行规约。

作业

P110 2(2)、4、5(i + i, i * (i * i))

补充题：有如下文法G[E]:

- **E→E + T | T**
- **T→E | (E) | i**

1. 求每个非终结符的FIRSTVT和LASTVT集合。

2. 构造算符优先关系矩阵。

3. 判断该文法是否为算符优先文法。

作业

编程实现算术表达式文法的算符优先文法（OPG）。

文法G [E]

E ::= E + T | T

T ::= T * F | F

F ::= (E) | i

4.3.4 LR分析法

1. 什么是LR分析：从左到右扫描(L)自底向上进行规约(R)

(是规范规约，也即最右推导）

是自底向上分析方法的高度概括和集中

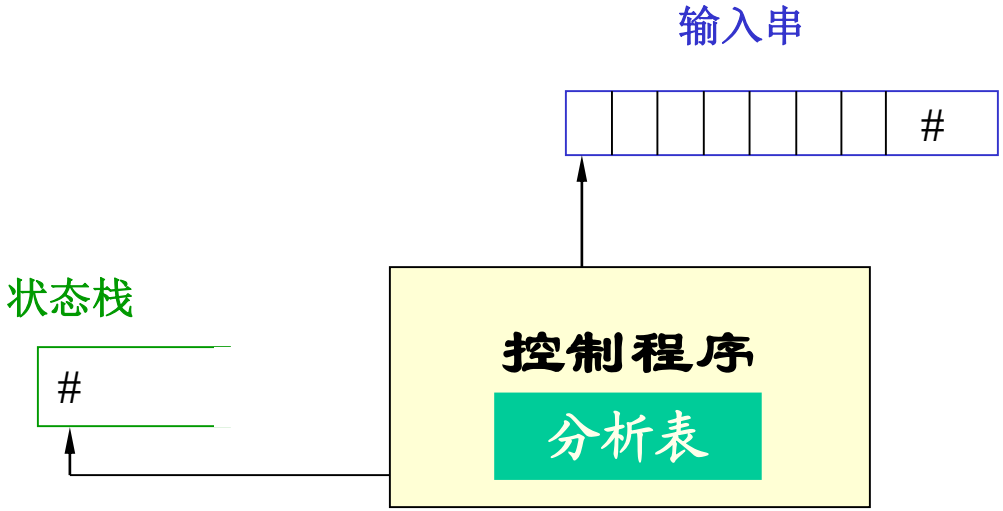
历史 + 展望 + 现状 => 句柄

(1) LR分析法的优缺点：

① 适合文法类足够大，适用于所有上下文无关文法
② 分析效率高
③ 报错及时
④ 可以自动生成
⑤ 但手工实现工作量大

(2) LR分析器有三部分：状态栈、分析表、控制程序

状态栈：放置分析器状态和文法符号。



分析表：由两个矩阵组成，其功能是指示分析器的动作，是移进还是规约，根据不同的文法类要采用不同的构造方法。

控制程序：执行分析表所规定的动作，对栈进行操作。

(3) 分析表的种类

① SLR分析表（简单LR分析表）

构造简单，最易实现，大多数上下文无关文法都可以构造出SLR分析表，所以具有较高的实用价值。使用SLR分析表进行语法分析的分析器叫SLR分析器。

② LR分析表（规范LR分析表）

适用文法类最大，所有上下文无关文法都能构造出LR分析表，但其分析表体积太大，实用价值不大。

③ LALR分析表（超前LR分析表）

这种表适用的文法类及其实现上难易在上面两种之间，在实用上很吸引人。

使用**LALR**分析表进行语法分析的分析器叫**LALR**分析器。

例：**UNIX——YACC**

(4) 几点说明

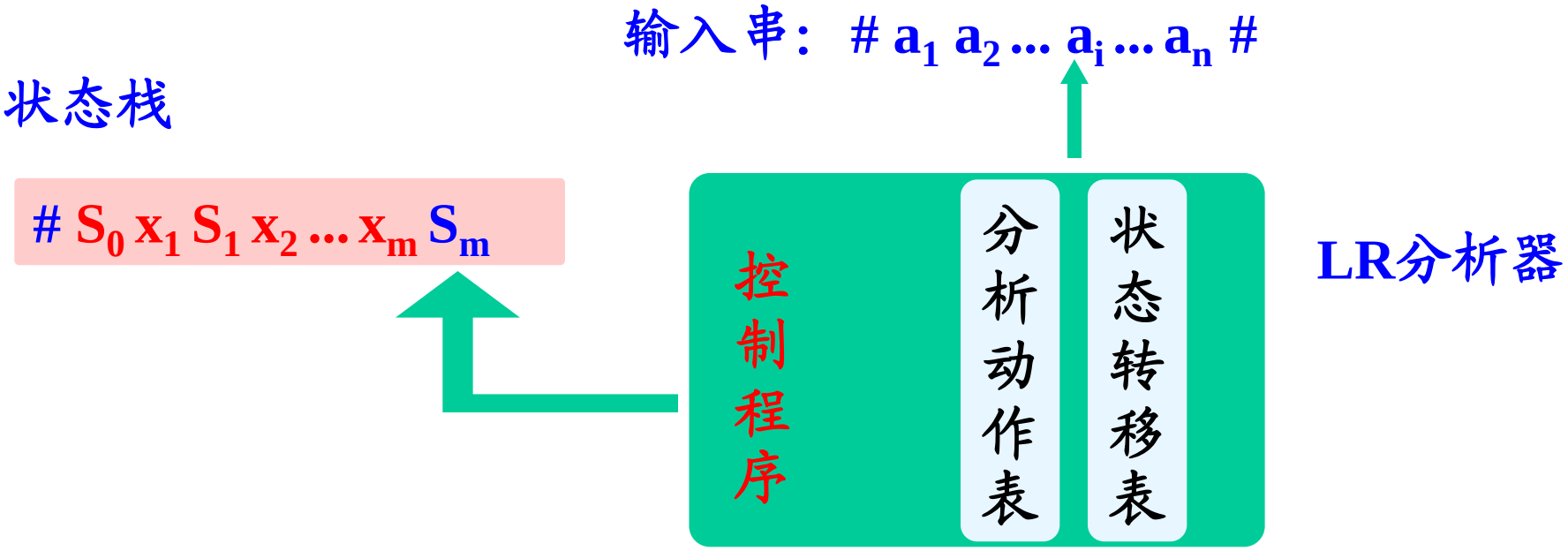
① 三种分析表对应三类文法

② 一个SLR文法必定是LALR文法和LR文法

③ 仅讨论SLR分析表的构造方法

2、LR分析器

（1）逻辑结构



$\# S_0 x_1 S_1 x_2 \ldots x_m S_m$ ⇒ $S_0 S_1 \ldots S_m$ / $\# x_1 x_2 \ldots x_m$

状态栈: $S_0, S_1, \ldots, S_m$ 状态

S_0—初始状态　　S_m—栈顶状态

栈顶状态概括了从分析开始到该状态的全部分析历史和展望信息。

符号串: $x_1 x_2 \ldots x_m$　　其中: $x_i \in V_n \cup V_t$

为从开始状态(S_0)到当前状态(S_m)所识别出的规范句型的活前缀。

规范句型： 通过规范规约得到的句型。

规范句型前缀： 将输入串的剩余部分与其连结起来就构成了规范句型。

如：$x_1 x_2 \cdots x_m a_i \cdots a_n$为规范句型。

活前缀： 若分析过程能够保证栈中符号均是规范句型的前缀，则表示输入串已分析过的部分没有语法错误，所以称为规范句型的活前缀。

规范句型的活前缀:

对于句型αβt，β表示句柄，如果αβ= $u_1 u_2 \dots u_r$，
那么符号串$u_1 u_2 \dots u_i (1 \le i \le r)$即是句型αβt的活前缀。

例: 文法G: E→T | E + T | E - T

T→ i | (E)

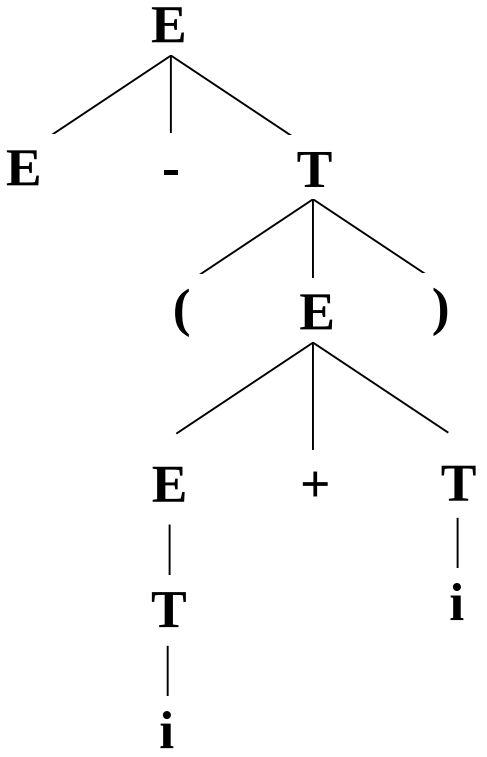
拓广文法G′: S→E #

E→T | E + T | E - T

T→ i | (E)

对于句型 E - (i + i) # 而言，

E、E -、E - (、E - (i 是其所有活前缀。



- 分析表

a. 状态转移表 (GOTO表)

一个矩阵:

行—分析器的状态

列—文法符号

（包括V_n和V_t）

GOTO表

状态＼符号	E	T	F	i	+	*	(	)	#
S_0									
S_1									
S_2									
:									
S_n									

GOTO表

状态 \ 符号	E	T	F	i	+	*	(	)	#
S_0									
S_1									
S_2									
:									
S_n									

$GOTO[S_{i-1}, x_i] = S_i$

S_{i-1}—当前状态（栈顶状态）

x_i—新的栈顶符号

S_i—新的栈顶状态（转移状态）

$\# S_0 x_1 S_1 x_2 \cdots x_{i-1} S_{i-1} x_i S_i$

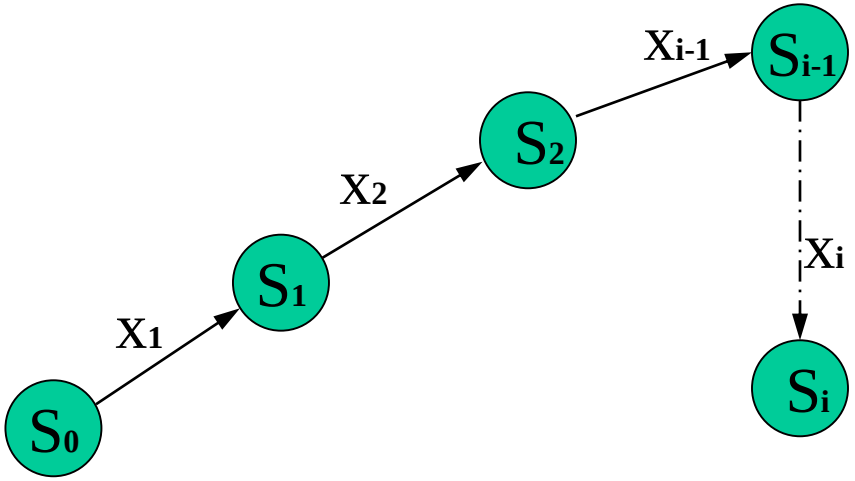
S_i需要满足条件是:

若$x_1 x_2 \ldots x_{i-1}$是由 S_0 到 S_{i-1} 所识别的规范句型的活前缀，则 $x_1 x_2 \ldots x_i$ 是由 S_0 到 S_i 所识别的规范句型的活前缀。

通过之前对有穷自动机的了解，我们可以看出：

状态转移函数GOTO是定义了一个以文法符号集为字母表的有穷自动机，该自动机识别文法所有规范句型的活前缀。

$$M = (S, V, GOTO, S_0, Z)$$



b. 分析动作表（ACTION表）

ACTION表

输入符号a / 状态S	+	*	i	(	)	#
S_0						
S_1						
S_2						
:						
S_n						

$ACTION[S_i , a]$ =分析动作　　$a \in V_t$

分析动作:

① 移进 (shift)

$ACTION[S_i, a] = s$

动作：将 a 推进栈，并设置新的栈顶

$S_j = GOTO[S_i, a]$，并将指针

② 规约 (reduce)

$ACTION[S_i, a] = r_d$

d: 文法规则编号　　(d) $A \rightarrow \beta$

动作：将符号串β(假定长度为n) 连同状态从栈内弹出，再把 A 推进栈，并设置新的栈顶状态为S_j。

$S_j = GOTO[S_{i-n}, A]$

例：文法G [E]

(1) E ::= E + T

(2) E ::= T

(3) T ::= T * F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

③ 接受(accept)

$ACTION[S_i , \#] = accept$

④ 出错 (error)

$ACTION[S_i , a] = error \qquad a \in V_t$

控制程序：(Driver Routine)

1、 根据栈顶状态和现行输入符号，查分析动作表(ACTION表)，执行由分析表所规定的操作；

2、并根据GOTO表设置新的栈顶状态（即实现状态转移）。

（2）LR分析过程

例：文法G [E]

(1) E ::= E + T

(2) E ::= T

(3) T ::= T * F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

该文法是SLR文法，故可以构造出SLR分析表（ACTION表和 GOTO表）

GOTO表

文法符号 / 状态	i	+	*	(	)	E	T	F
0(S_0)	5			4		1	2	3
1(S_1)		6						
2(S_2)			7					
3(S_3)								
4(S_4)	5			4		8	2	3
5(S_5)								
6(S_6)	5			4			9	3
7(S_7)	5			4				10
8(S_8)		6			11			
9(S_9)			7					
10(S_{10})								
11(S_{11})								

ACTION表

状态 \ 文法符号	i	+	*	(	)	#
0(S_0)	s			s		
1(S_1)		s				acc
2(S_2)		r_2	s		r_2	r_2
3(S_3)		r_4	r_4		r_4	r_4
4(S_4)	s			s		
5(S_5)		r_6	r_6		r_6	r_6
6(S_6)	s			s		
7(S_7)	s			s		
8(S_8)		s			s	
9(S_9)		r_1	s		r_1	r_1
10(S_{10})		r_3	r_3		r_3	r_3
11(S_{11})		r_5	r_5		r_5	r_5

ACTION 表　　GOTO 表

状态＼输入符号	i	+	*	(	)	#	E	T	F
0	s_5			s_4			1	2	3
1		s_6				ACCEPT			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

分析过程 i * i + i

ACTION 表 / GOTO 表

	i	+	*	(	)	#	E	T	F
0	s_5			s_4			1	2	3
1		s_6				ACCEPT			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

文法G [E]

(1) E ::= E + T

(2) E ::= T

(3) T ::= T * F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

步骤	状态栈	符号	输入串	动作
1	# 0	#	i * i + i #	初始化
2	# 0 i 5	# i	* i + i #	S
3	# 0 F 3	# F	* i + i #	R_6
4	# 0 T 2	# T	* i + i #	R_4
5	# 0 T 2 * 7	# T *	i + i #	S
6	# 0 T 2 * 7 i 5	# T * i	+ i #	S
7	# 0 T 2 * 7 F 10	# T * F	+ i #	R_6

ACTION 表 / GOTO 表

	i	+	*	(	)	#	E	T	F
0	s_5			s_4			1	2	3
1		s_6				ACCEPT			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

文法G [E]

(1) E ::= E + T

(2) E ::= T

(3) T ::= T * F

(4) T ::= F

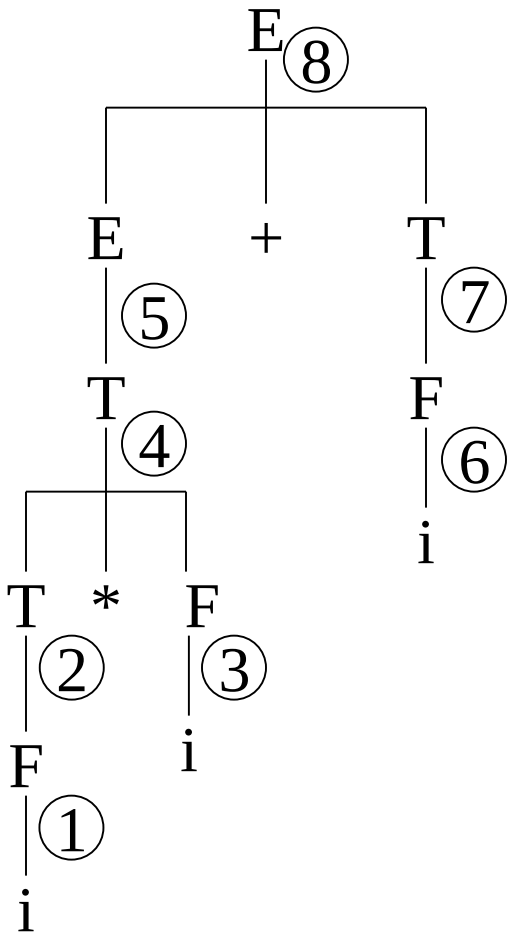
(5) F ::= (E)

(6) F ::= i

8	# 0 T 2	# T	+ i #	R_3
9	# 0 E 1	# E	+ i #	R_2
10	# 0 E 1 + 6	# E +	i #	S
11	# 0 E 1 + 6 i 5	# E + i	#	S
12	# 0 E 1 + 6 F 3	# E + F	#	R_6
13	# 0 E 1 + 6 T 9	# E + T	#	R_4
14	# 0 E 1	# E	#	R_1
15		# E		Accept

将上述过程每一步的活前缀与剩余输入串连接起来——每一步的句型

i * i + i		E
F * i + i		=> E + T
T * i + i		=> E + F
T * F + i		=> E + i
T + i	颠倒	=> T + i
E + i		=> T * F + i
E + F		=> T * i + i
E + T		=> F * i + i
E		=> i * i + i



最左规约（规范规约）
最右推导（规范推导）

由分析过程可以看到:

(1) 每次规约总是规约当前句型的句柄，是规范规约!

（而算符优先是规约最左素短语）

(2) 分析的每一步栈内符号串均是规范句型的活前缀，且与输入串的剩余部分构成规范句型。

3、构造LR分析表

构造LR分析器的关键是构造其分析表！

构造LR分析表的方法是：

（1）根据文法构造识别规范句型活前缀的有穷自动机 DFA。

（2）由DFA构造LR分析表。

(1) 构造DFA

① DFA 是一个五元式

$$M = (S,\ V,\ GOTO,\ S_0,\ Z)$$

S: 有穷状态集

在此具体情况下，S = LR(0)项目集规范族。

项目集规范族：其元素是由项目所构成的集合。

V: 文法字汇表

S_0: 初始状态

Z: 终态集合 $Z = S - \{S_0\}$

除 S_0 以外，其余全部是终态。

GOTO：状态转移函数

$GOTO[S_i, x] = S_j$

$S_i, S_j \in S$　　S_i, S_j为项目集合

$x \in V_n \cup V_t$

表示当前状态 S_i 面临文法符号 x 时，应将状态转移到 S_j。

构造DFA:

一、 确定 S 集合，即LR(0)项目集规范族，同时确定S_0

二、确定状态转移函数GOTO

② 构造LR(0)的方法

LR(0)是DFA的状态集，其中每个状态又都是项目的集合。

项目：文法 G 的每个产生式（规则）的右部添加一个圆点就构成一个项目。

例：产生式：A→XYZ

项　目：A→.XYZ

A→X.YZ

A→XY.Z

A→XYZ.

产生式：A→ε

项　目：A→.

项目的直观意义：指明在分析过程中的某一时刻的已经规约部分和等待规约部分。

构造LR(0)文法的方法(三步)

1. 将文法扩充

目的：使构造出来的分析表只有一个接受状态，这是为了实现上的方便。

方法：修改文法，使识别符号（开始符号）的规则只有一条。

G [E]
E → …
E → …
…

拓广 ⇒

G' [E']
E' →E
E → …
E → …
…

L (G (E)) = L (G' [E])

2、 根据文法列出所有的项目

3、将有关项目组合成集合，即DFA中的状态;
所有状态构成了LR(0) 项目集规范族。

我们通过一个具体例子来说明LR(0)的构造以及DFA的构造方法。

例： G [E]

E→E + T | T

T→T * F | F

F→(E) | i

例：G [E]
E→E + T | T
T→T * F | F
F→(E) | i

① 将文法拓广为G' [E']

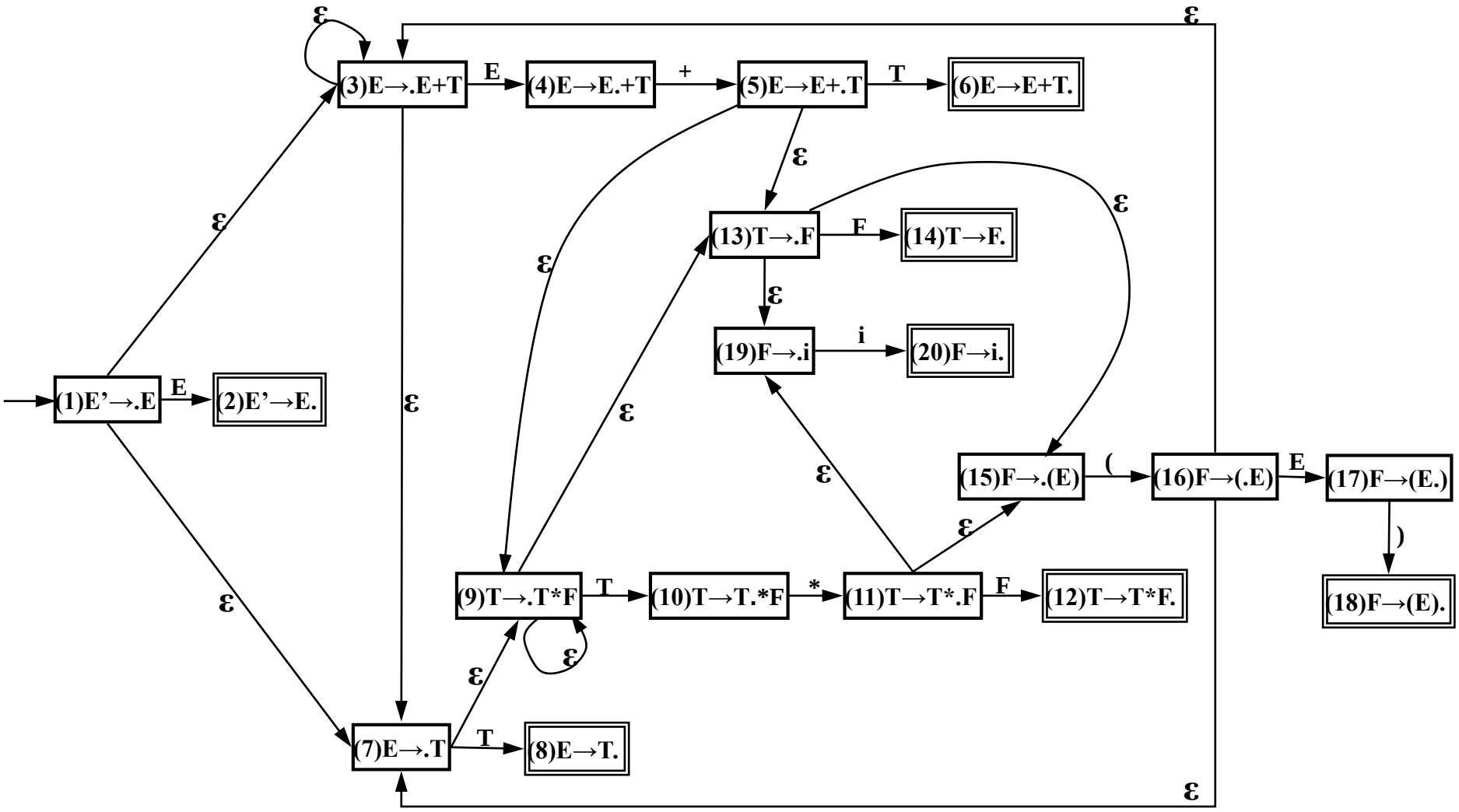
(0) E'→E
(1) E→E + T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i

②列出文法的所有项目

(1) E'→. E
(2) E'→E .
(3) E→. E + T
(4) E→E . + T
(5) E→E + .T
(6) E→E + T .
(7) E→. T
(8) E→T .
(9) T→. T * F
(10) T→T . * F
(11) T→T * . F
(12) T→T * F .
(13) T→. F
(14) T→F .
(15) F→. (E)
(16) F→(. E)
(17) F→(E .)
(18) F→(E) .
(19) F→. i
(20) F→i .

识别活前缀的NFA

(1) E'→. E
(2) E'→E .
(3) E→. E + T
(4) E→E . + T
(5) E→E + .T
(6) E→E + T .
(7) E→. T
(8) E→T .
(9) T→. T * F
(10) T→T . * F
(11) T→T * . F
(12) T→T * F .
(13) T→. F
(14) T→F .
(15) F→. (E)
(16) F→(. E)
(17) F→(E .)
(18) F→(E) .
(19) F→. i
(20) F→i .



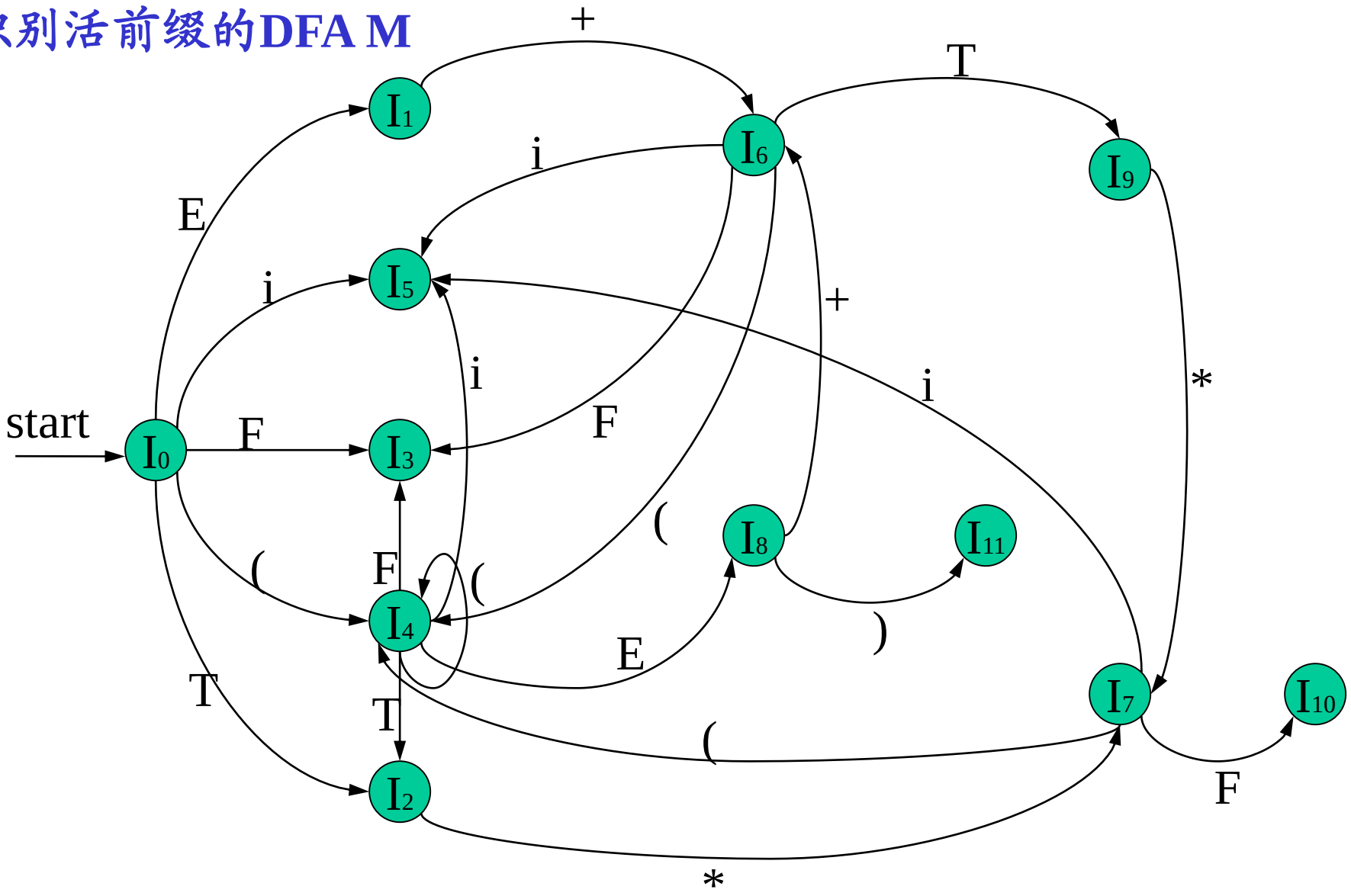
NFA的确定化

	E	T	F	+	*	(	)	i
{ 1, 3, 7, 9, 13, 15, 19 }	{ 2, 4 }	{ 8, 10 }	{ 14 }			{ 3, 7, 9, 13, 15, 16, 19 }		{ 20 }
{ 2, 4 }				{ 5, 9, 13, 15, 19 }				
{ 8, 10 }					{ 11, 15, 19 }			
{ 14 }								
{ 3, 7, 9, 13, 15, 16, 19 }	{ 17, 4 }	{ 8, 10 }	{ 14 }			{ 3, 7, 9, 13, 15, 16, 19 }		{ 20 }
{ 20 }								
{ 5, 9, 13, 15, 19 }		{ 6, 10 }	{ 14 }			{ 3, 7, 9, 13, 15, 16, 19 }		{ 20 }
{ 11, 15, 19 }			{ 12 }			{ 3, 7, 9, 13, 15, 16, 19 }		{ 20 }
{ 17, 4 }				{ 5, 9, 13, 15, 19 }				{ 18 }
{ 14 }								
{ 6, 10 }					{ 11, 15, 19 }			
{ 12 }								
{ 18 }								

替换

	E	T	F	+	*	(	)	i
I_0	I_1	I_2	I_3			I_4		I_5
I_1				I_6				
I_2					I_7			
I_3								
I_4	I_8	I_2	I_3			I_4		I_5
I_5								
I_6		I_9	I_3			I_4		I_5
I_7			I_{10}			I_4		I_5
I_8				I_6			I_{11}	
I_3								
I_9					I_7			
I_{10}								
I_{11}								

识别活前缀的DFA M



上述方法计算复杂！容易出错，下面给出另外一种方法：

③ 将有关项目组成项目集，所有项目集构成的集合即为 **LR(0)**。

为实现这一步，需先定义：

- 项目集闭包 **closure**
- 状态转移函数**GOTO**

(0) E'→E
(1) E→E + T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i

例: G'[E']

令I = { E'→. E }

closure(I) = {E'→. E , E→. E + T, E→. T , T→. T * F , T→.F , F→. (E) , F→. i }

Procedure closure(I);

begin

将属于 I 的项目加入 closure(I);

repeat

for closure(I)中的每个项目A→α.Bβ(B∈V_n) do

将B→. r (r∈V*)加入closure(I)

until closure(I)不再增大

end

B. 状态转移函数GOTO的定义:

GOTO (I, X) = closure (J)

I: 项目集合

X: 文法符号，X∈V

J: 项目集合

J = { 任何形如A→αX.β的项目 | A→α.Xβ∈I }

closure (J): 项目集 J 的闭包仍是项目集合。

所以，GOTO(I, X) = closure(J) 的直观意义是:

它规定了识别文法规范句型活前缀的DFA从状态 I (项目集)出发，经过X弧所应该到达的状态(项目集) 。

(0) E'→E
(1) E→E + T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i

例:

I = { E'→E . , E→E . + T }　求GOTO(I , +) = ?

GOTO(I , +) = closure(J)

∵ J = { E→E + .T }

∴ GOTO(I , +) = { E→E + .T , T→.T * F , T→. F ,
F→. (E) , F→. i }

LR (0) 项目集规范族的构造算法:

G'→LR (0)

```
procedure   ITEMSETS (G');
      begin
         LR(0):={closure({E'→.E})};
         repeat
            for  LR (0)中的每个项目集 I 和 G' 的每个符号X  do
               if  GOTO (I , X) 非空且不属于LR(0)
                  then   把GOTO (I , X) 放入LR(0)中
            until   LR(0)不再增大
end
```

(0) E'→E
(1) E→E + T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i

例：求G'[E']的LR (0)　　V = { E, T, F, i, +, *, (,) }

G'[E']共有20个项目

项目集如下：

I_0：
E'→. E
E→. E + T
E→. T
T→. T * F
T→. F
F→. (E)
F→. i

closure({E'→. E}) = I_0

I_1：
E'→E .
E→E . + T

GOTO(I_0, E) = closure({E'→E., E→E.+T})
=I_1

I_2: $E \rightarrow T .$ $\quad GOTO(I_0, T) = closure(\{E \rightarrow T., T \rightarrow T.*F\}) = I_2$
$T \rightarrow T . * F$

I_3: $T \rightarrow F .$ $\quad GOTO(I_0, F) = closure(\{T \rightarrow F.\}) = I_3$

I_4: $F \rightarrow (. E)$ $\quad GOTO(I_0, ( ) = closure(\{F \rightarrow (.E)\}) = I_4$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . i$

I_0: $E' \rightarrow . E$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . i$

I_5: $F \rightarrow i .$ $\quad GOTO(I_0, i) = closure(\{F \rightarrow i.\}) = I_5$

$GOTO(I_0, *) = \varphi$

$GOTO(I_0, +) = \varphi$

$GOTO(I_0,)) = \varphi$

I_1: E'→E .
E→E . + T

I_6:
- E→E + . T
- T→. T * F
- T→. F
- F→. (E)
- F→. i

GOTO (I_1 , +) = closure({E→E +.T}) = I_6

GOTO (I_1 , 其他符号)为空

I_2: E→T .
T→T . * F

I_7:
- T→T * . F
- F→. (E)
- F→. i

GOTO (I_2 , *) = closure({T→T*.F}) = I_7

GOTO (I_2 , 其他符号)为空

GOTO (I_3 , 其他符号)为空

I_3: T→F .

I_4: $F \rightarrow (\ .\ E\ )$ $E \rightarrow .\ E + T$
$E \rightarrow .T$ $T \rightarrow .\ T * F$
$T \rightarrow .F$ $F \rightarrow .\ (\ E\ )$ $F \rightarrow .\ i$

I_8: $F \rightarrow (\ E\ .\ )$ $E \rightarrow E\ .+\ T$

$GOTO(I_4, E) = closure(\{F \rightarrow (E.), E \rightarrow E.+T\}) = I_8$
$GOTO(I_4, T) = I_2 \in LR(0)$
$GOTO(I_4, F) = I_3 \in LR(0)$
$GOTO(I_4, () = I_4 \in LR(0)$
$GOTO(I_4, i) = I_5 \in LR(0)$
$GOTO(I_4, +) = \varphi$
$GOTO(I_4, *) = \varphi$
$GOTO(I_4,)) = \varphi$
$GOTO(I_5, 其他符号) = \varphi$

I_6: $E \rightarrow E + .\ T$ $F \rightarrow .\ (\ E\ )$
$T \rightarrow .\ T * F$ $F \rightarrow .\ i$
$T \rightarrow .\ F$

I_9: $E \rightarrow E + T\ .$ $E \rightarrow T.\ *\ F$

$GOTO(I_6, T) = closure(\{E \rightarrow E+T., T \rightarrow T.*F\}) = I_9$
$GOTO(I_6, F) = I_3$
$GOTO(I_6, () = I_4$
$GOTO(I_6, i) = I_5$

I_{10}: $T \rightarrow T * F .$ $GOTO(I_7, F) = closure(\{T \rightarrow T * F.\}) = I_{10}$

$GOTO(I_7, ( ) = I_4$

$GOTO(I_7, i) = I_5$

I_{11}: $F \rightarrow (E) .$ $GOTO(I_8,)) = closure(\{F \rightarrow (E) .\}) = I_{11}$

$GOTO(I_8, +) = I_6$

继续求完所有 I_i 的后继

$GOTO(I_9, *) = I_7$

$GOTO(I_{10}$, 所有符号) =φ, $GOTO(I_{11}$, 所有符号) =φ

$LR(0) = \{ I_0, I_1, I_2, \ldots I_{11}\}$ 共由12个项目集组成

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot i$

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot +T$

I_2: $E \rightarrow T \cdot$
$T \rightarrow T \cdot *F$

I_3: $T \rightarrow F \cdot$

I_4: $F \rightarrow (\cdot E)$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot i$

I_5: $F \rightarrow i \cdot$

I_6: $E \rightarrow E+ \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot i$

I_7: $T \rightarrow T* \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot i$

I_8: $F \rightarrow (E \cdot)$
$E \rightarrow E \cdot +T$

I_9: $E \rightarrow E+T \cdot$
$T \rightarrow T \cdot *F$

I_{10}: $T \rightarrow T*F \cdot$

I_{11}: $F \rightarrow (E) \cdot$

状态转换函数

	E	T	F	i	+	*	(	)
I_0	I_1	I_2	I_3	I_5			I_4	
I_1					I_6			
I_2						I_7		
I_3								
I_4	I_8	I_2	I_3	I_5			I_4	
I_5								
I_6		I_9	I_3	I_5			I_4	
I_7			I_{10}	I_5			I_4	
I_8					I_6			I_{11}
I_9						I_7		
I_{10}								
I_{11}								

④ 构造DFA

$M = (S, V, GOTO, S_0, Z)$

其中：$S = \{I_0, I_1, I_2, \cdots, I_{11}\} = LR(0)$

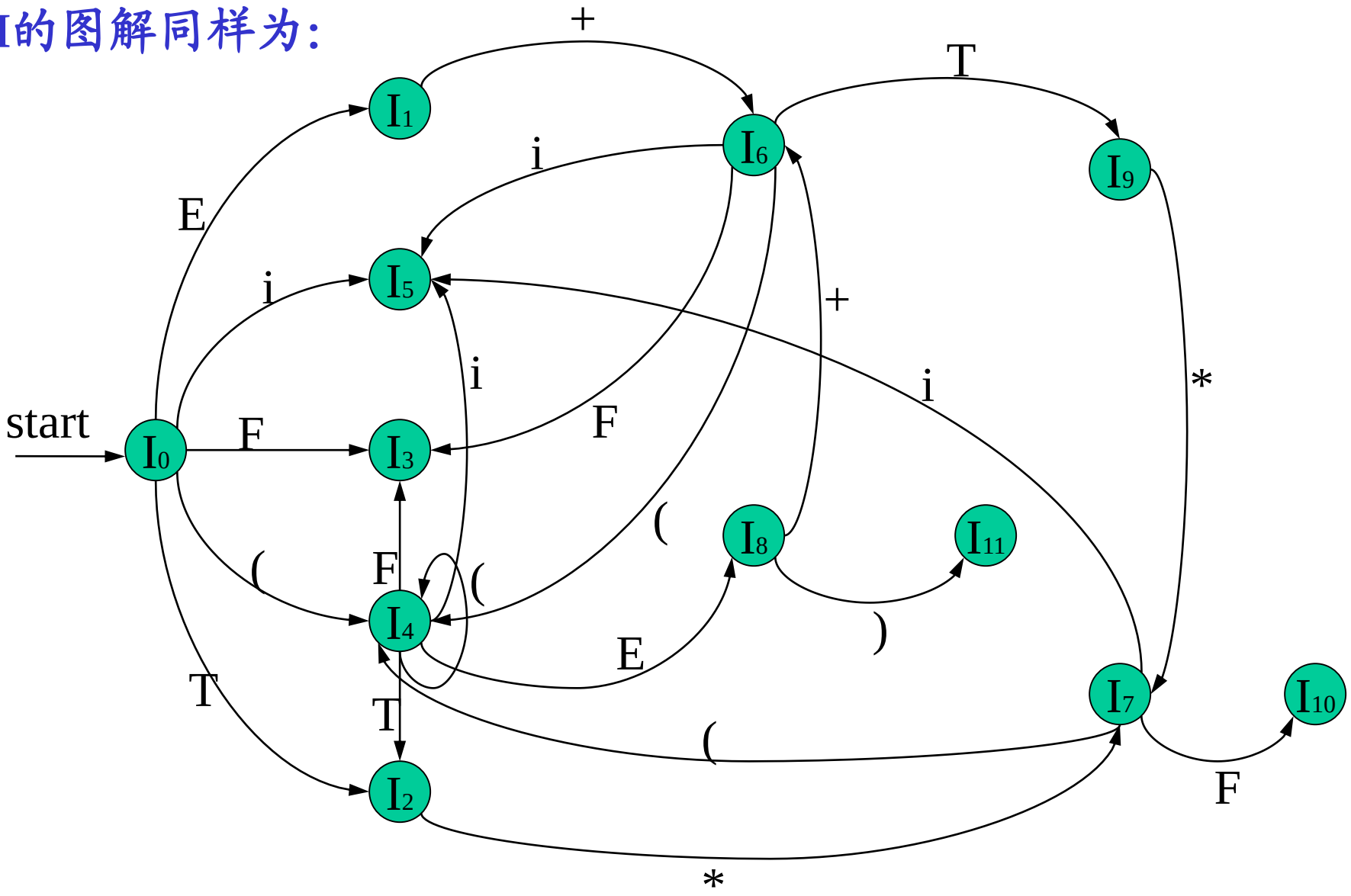
$V = \{+, *, i, (,), E, T, F\}$

$GOTO(I_m, X) = I_n$

$S_0 = I_0$

$Z = S - \{I_0\} = \{I_1, I_2, \cdots, I_{11}\}$

M的图解同样为:



关于自动机的说明:

① 从 I_0 到每一状态（除 I_0 以外）的每条路径都识别和接受一个规范句型的活前缀。

如对文法句子 i + i * i 进行规范规约所得到的规范句型的活前缀都可以由该自动机识别。如:

$I_0 \sim I_1$ 识别规范句型E + i * i 的活前缀 E

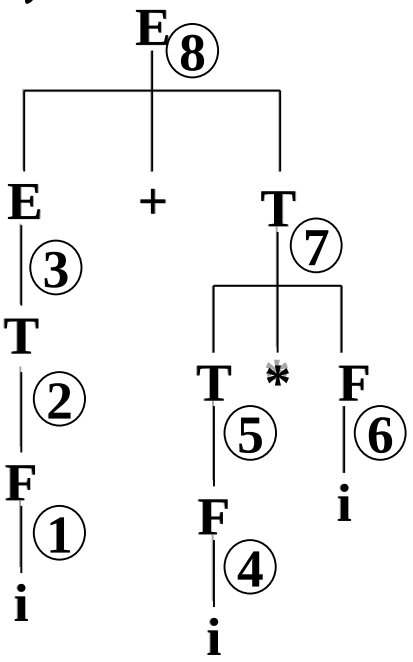
$I_0 \sim I_3$ 识别规范句型E + F * i 的活前缀 E + F

$I_0 \sim I_6$ 识别规范句型E + i * i的活前缀 E +

$I_0 \sim I_7$ 识别规范句型E + T * i的活前缀 E + T *

$I_0 \sim I_9$ 识别规范句型E + T * i的活前缀 E + T

$I_0 \sim I_{10}$ 识别规范句型E + T * F的活前缀 E + T * F



② 要求状态中每个项目对该状态能识别的活前缀都有效。

有效项目定义：若项目A→α.β，对活前缀φα有效，则其条件是存在规范推导

$E' \Rightarrow \phi A w \Rightarrow \phi \alpha \beta w$。

注意：项目中圆点前的符号串称为活前缀的后缀。

③ 有效项目能预测分析的下一步动作：

E→E+T．表示已将输入串规约为E+T，下一步应该将E+T规约为E

$E' \Rightarrow E \Rightarrow E+T$

T→T.*F　表示已将输入串规约为T，下一步动作是移进输入符号*

注意：经移进或规约后，栈内仍是规范句型的活前缀。

④ DFA中的状态既代表了分析历史又提供了展望信息。

每条规范句型的活前缀都代表了一个确定的规范规约过程，故有效状态可以代表分析历史。

由于状态中的项目都是有效项目，所以提供了下一步可能采取的动作。

历史 + 展望 + 实现 ⇒ 句柄

LR文法

- 对于一个文法，如果能够构造一张分析表，使得它的每个入口均是唯一确定的，则我们将把这个文法称为LR文法。

- 并非所有的上下文无关文法都是LR文法。但对于多数程序语言来说，一般都可用LR文法描述。

（2）构造LR分析表

* GOTO表可由DFA直接求出

GOTO表

状态 \ 文法符号	E	T	F	i	+	*	(	)
0(S_0)	1	2	3	5			4	
1(S_1)					6			
2(S_2)						7		
3(S_3)								
4(S_4)	8	2	3	5			4	
5(S_5)								
6(S_6)		9	3	5			4	
7(S_7)			10	5			4	
8(S_8)					6			11
9(S_9)						7		
10(S_{10})								
11(S_{11})								

* ACTION表由项目集规范族求出

根据圆点所在的位置和圆点后是终结符还是非终结符，把项目集规范族中的项目分为以下四种：

项目	种类	分析动作
$A\rightarrow\alpha.$	规约项目	规约
$E'\rightarrow\alpha.$ (E'为开始符号)	接受项目	接受
$A\rightarrow\alpha.a\beta$ $(a\in V_t)$	移进项目	移进
$A\rightarrow\alpha.B\beta$ $(B\in V_n)$	待约项目	无

LR(0)文法

假若一个文法G的拓广文法G′的活前缀识别自动机中的每个状态(项目集)不存在下述情况:

1) 既含移进项目又含归约项目

2) 含有多个归约项目

则称G是一个**LR(0)文法**。

若:

a) 移进和归约项目同时存在。移进-归约冲突

b) 归约和归约项目同时存在。归约-归约冲突

LR(0)文法

LR(0)分析器的特点是不需要向前查看任何输入符号就能归约。即当栈顶形成句柄，不管下一个输入符号是什么，都可以立即进行归约而不会发生错误。

LR(0)文法过于简单。即使是定义算术表达式这样的简单文法也不是LR(0)文法，所以没有实用价值!

前述项目集规范族

I_0: E′→·E
E→·E+T
E→·T
T→·T*F
T→·F
F→· (E)
F→·i

I_1: E′→E·
E→E·+T

I_2: E→T·
T→T·*F

I_3: T→F·

I_4: F→(·E)
E→·E+T
E→·T
T→·T*F
T→·F
F→· (E)
F→·i

I_5: F→i·

I_6: E→E+·T
T→·T*F
T→·F
F→·(E)
F→·i

I_7: T→T*·F
F→·(E)
F→·i

I_8: F→(E·)
E→E·+T

I_9: E→E+T·
T→T·*F

I_{10}: T→T*F·

I_{11}: F→(E)·

(0) E'→E
(1) E→E + T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i

其中I_1、I_2和I_9都可能含有“移进－归约”冲突。

I_1: E′→E·
E→E·+T

I_2: E→T·
T→T·*F

I_9: E→E+T·
T→T·*F

结论：前述G(E′)文法不是LR(0)文法！

解决方法——看FOLLOW集

FOLLOW(E′) = { # }

FOLLOW(E) = { # , + ,) }

FOLLOW(T) = FOLLOW(F) = { # , * , + ,) }

其分析表为:

ACTION表

状态	i	+	*	(	)	#
0	s			s		
1		s				acc
2		r2	s		r2	r2
3		r4	r4		r4	r4
4	s			s		
5		r6	r6		r6	r6
6	s			s		
7	s			s		
8		s			s11	
9		r1	s		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

* 求SLR文法ACTION表的一般方法

归约-归约冲突

假定一个LR(0)规范族中含有如下的一个项目集（状态）

I = { X→α·bβ，A→α·，B→α·}

FOLLOW(A)和FOLLOW(B)的交集为空，且不包含b，那么，当状态I面临任何输入符号a 时:

1. 若a = b，则移进;
2. 若a∈FOLLOW(A)，用产生式A→α进行归约;
3. 若a∈FOLLOW(B)，用产生式B→α进行归约;
4. 此外，报错。

一般地，假定LR(0)规范族的一个项目集

$I = \{A_1 \rightarrow \alpha \cdot a_1\beta_1，A_2 \rightarrow \alpha \cdot a_2\beta_2，\ldots，A_m \rightarrow \alpha \cdot a_m\beta_m，B_1 \rightarrow \alpha \cdot，B_2 \rightarrow \alpha \cdot，\ldots，B_n \rightarrow \alpha \cdot\}$

如果集合$\{a_1, \ldots, a_m\}$, $FOLLOW(B_1), \ldots, FOLLOW(B_n)$两两不相交（包括不得有两个FOLLOW集合有#），则：

1. 若输入a是某个a_i，$i = 1, 2, \ldots, m$，则移进；
2. 若$a \in FOLLOW(B_i)$，$i = 1, 2, \ldots, n$，则用产生式$B_i \rightarrow \alpha$进行归约；
3. 此外，报错。

——冲突的SLR(1)解决办法。

说明:

- 按上述方法构造出的ACTION与GOTO表如果不含多重入口，则称该文法为SLR(1)文法（向前看了一步）。
- 使用SLR表的分析器叫做一个SLR分析器。
- 每个SLR(1)文法都是无二义的。但也存在许多无二义文法不是SLR(1)的。

非SLR文法示例：考虑如下文法：

(0) S′→S

(1) S→L=R

(2) S→R

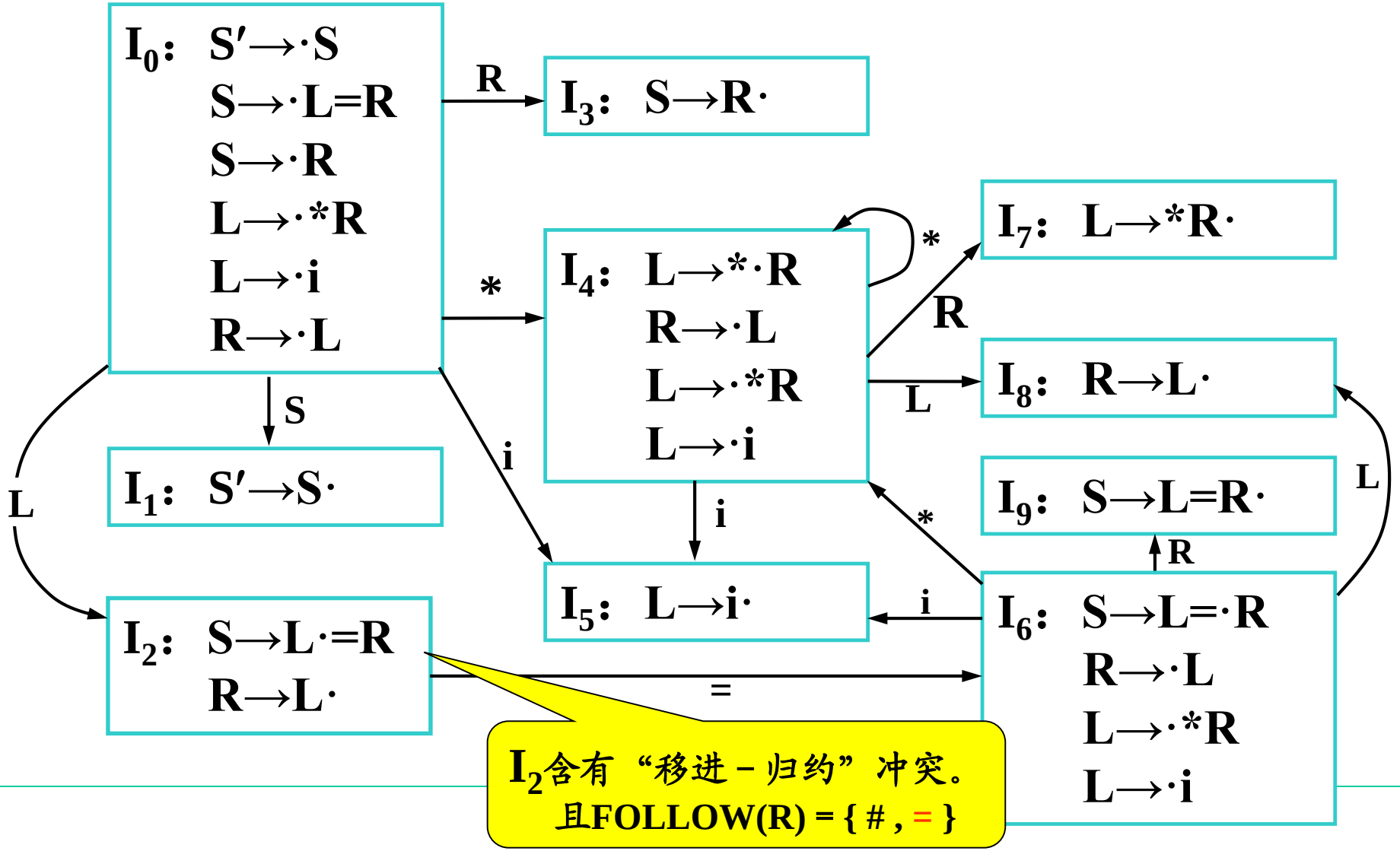
(3) L→*R

(4) L→i

(5) R→L

(0) S′→S
(1) S→L=R
(2) S→R
(3) L→*R
(4) L→i
(5) R→L

这个文法的LR(0)项目集规范族为：



- SLR文法中，如果项目集 I_i 含项目 $A\rightarrow\alpha.$ 而下一输入符号 $a\in FOLLOW(A)$，则状态 i 面临 a 时，可选用“$A\rightarrow\alpha$”进行归约（但是有前提的）。
- 但在有些情况下（非SLR文法），当状态 i 显现于栈顶时，栈里的活前缀未必允许把 α 归约为A，因为可能根本就不存在一个形如“βAa”的规范句型。因此，在这种情况下用“$A\rightarrow\alpha$”归约不一定合适。

FOLLOW集合提供的信息太泛！

(规范) LR(1)分析法

若项目集[A→α•B β]属于 I 时，则[B→• γ] 也属于I。

把FIRST(β)作为用产生式归约的搜索符（称为向前搜索符），即用产生式B→γ归约时查看的符号集合（用以代替SLR(1)分析中的FOLLOW集），并把此搜索符号的集合也放在相应项目的后面，这种处理方法即为LR(1)分析方法。

LR(1)项目集族的构造：针对初始项目S'→•S, # 求闭包后再用转换函数逐步求出整个文法的LR(1)项目集族。

1）构造LR(1) 项目集的闭包函数

a) I 的项目都在closure(I)中；

b) 若A→α•Bβ, a属于closure(I)，B→γ是文法的产生式，β∈V*，b∈first(β)，则B→• γ, b也属于closure(I)；

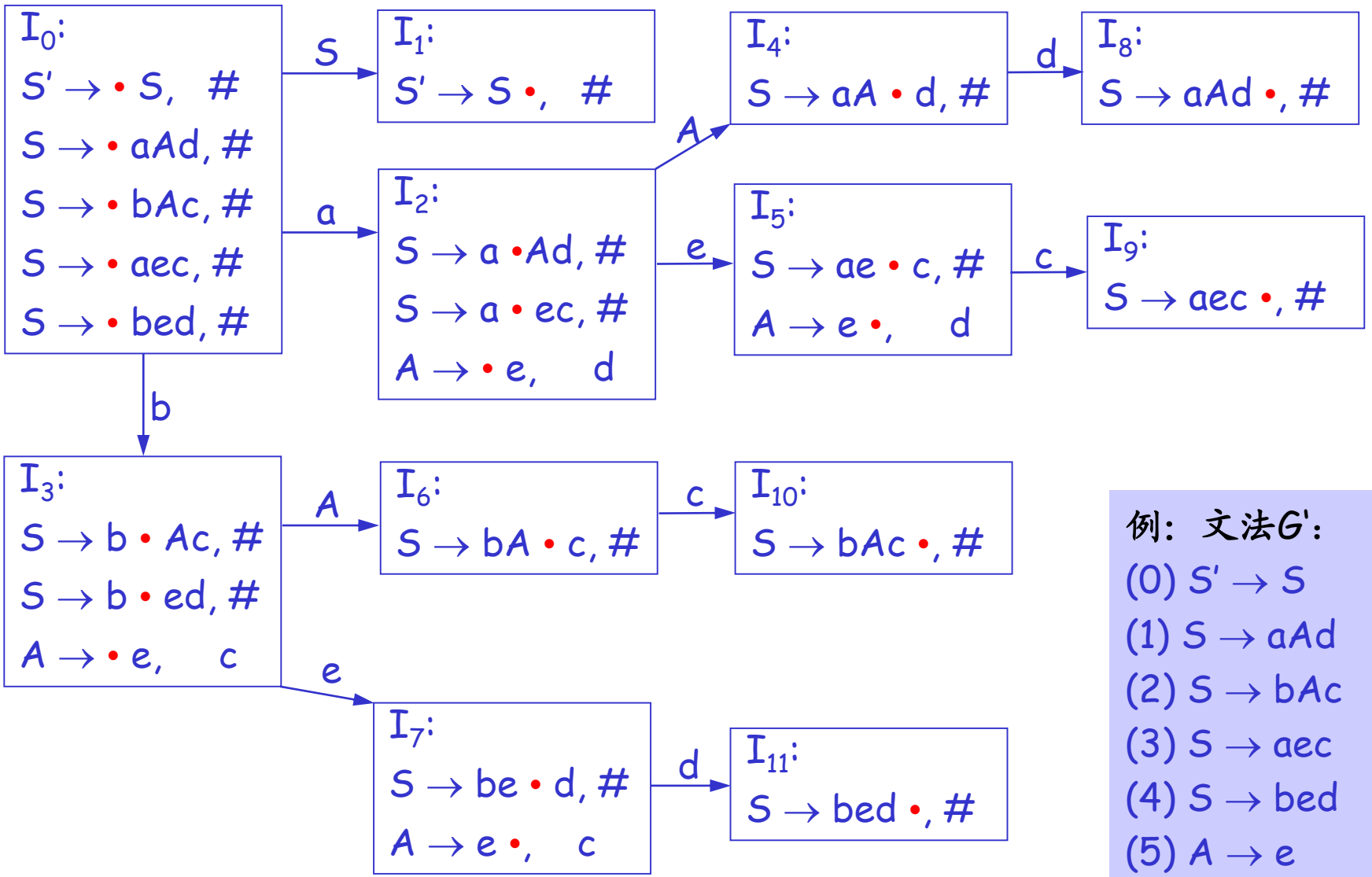
c) 重复b)直到closure(I)不再扩大为止。

2) 转换函数的构造

GOTO(I, X) = closure(J)

其中：I为LR(1)的项目集，X为一文法符号

J = {任何形如A→αX•β, a的项目 | A→α•X β, a属于I }



例：文法G'：

(0) S' → S
(1) S → aAd
(2) S → bAc
(3) S → aec
(4) S → bed
(5) A → e

LR(1)分析表的构造算法

1) 若项目[A→α•a β, b]属于I_k，且转换函数GOTO(I_k, a) = I_j。当a为终结符时，则置ACTION[k, a]为S_j。（b没有用处！）

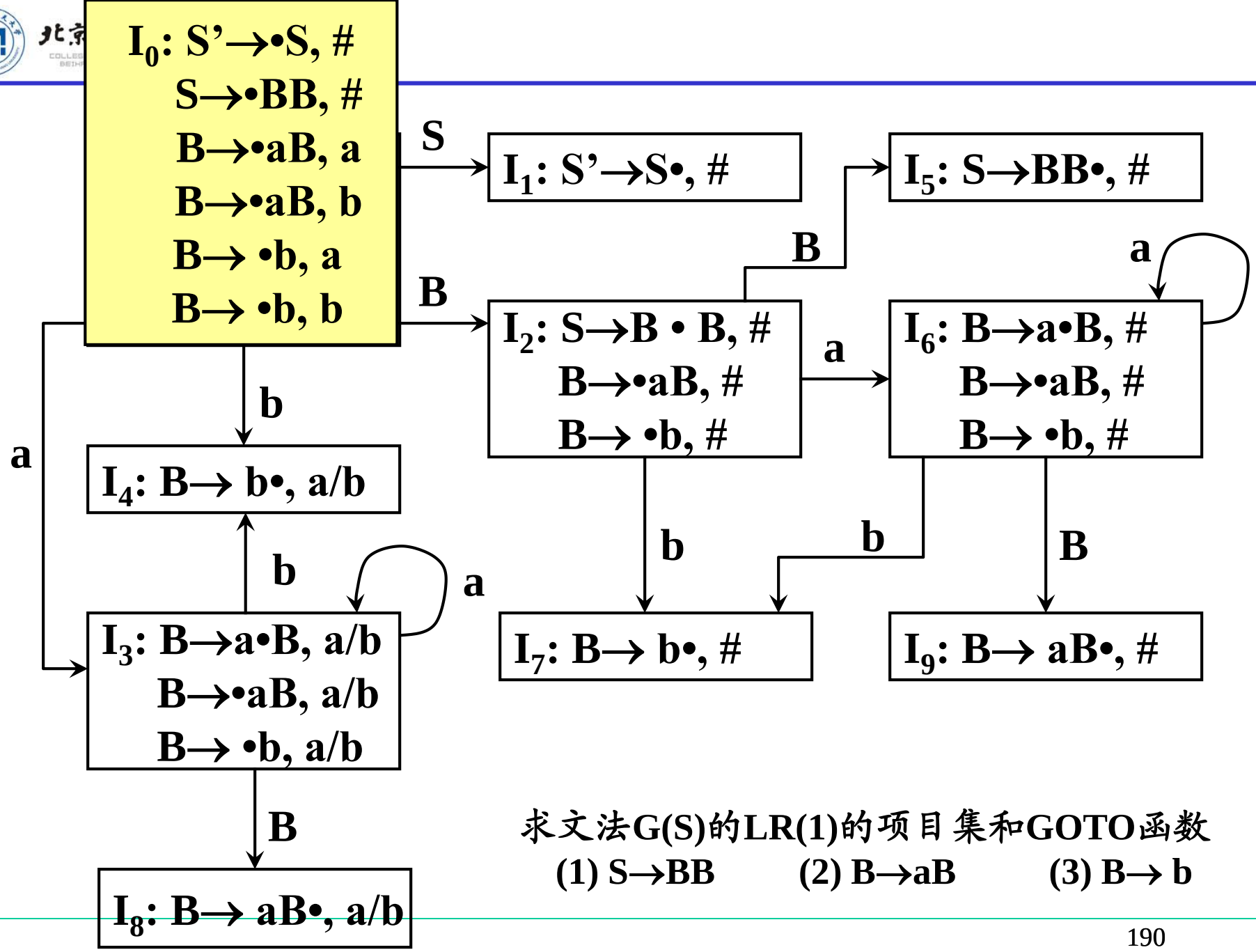
2) 若项目[A→α•, a]属于I_k，则对a为任何终结符或“#”，置ACTION[k, a] = r_j，j为产生式在文法G’中的编号。

（SLR文法中, 是将FOLLOW(A)所对应的位置全置为规约）

3) 若GOTO(I_k, A) = I_j，这里则置GOTO[k, A] = j。其中A为非终结符，j为某一状态号。

4) 若项目[S’ →S •, #]属于I_k，则置ACTION[k, #] = acc

5) 其它填上“报错标志”。



求文法G(S)的LR(1)的项目集和GOTO函数

(1) S→BB　　(2) B→aB　　(3) B→ b

LR(1)分析表为:

状态	ACTION			GOTO	
	a	b	#	*S*	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

例: 按上表对aabab进行分析

步骤	状态	符号	输入串	
0	0	#	aabab#	
1	03	#*a*	abab#	
2	033	#aa	bab#	
3	0334	#aab	ab#	
4	0338	#aaB	ab#	
5	038	#aB	ab#	
6	02	#B	ab#	
7	026	#Ba	b#	
8	0267	#Bab	#	
9	0269	#BaB	#	
10	025	#BB	#	
11	01	#*S*	#	acc

状态	ACTION			GOTO	
	a	b	#	*S*	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

0. S'→S
1. S→BB
2. B→aB
3. B→ b

又例：按上表对abab进行分析

步骤	状态	符号	输入串	
0	0	#	abab#	
1	03	#*a*	bab#	
2	034	#ab	ab#	
3	038	#aB	ab#	
4	02	#B	ab#	
5	026	#Ba	b#	
6	0267	#Bab	#	
7	0269	#BaB	#	
8	025	#BB	#	
9	01	#*S*	#	acc

状态	ACTION			GOTO	
	a	b	#	*S*	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1)分析法的特点:

- **可适用的文法范围最大。**
- **每个SLR(1)文法都是LR(1)文法，但反之不成立!**
- **LR(1)项目集的构造对某些项目集的分裂可能使状态数目剧烈地增长。**

  例如对Pascal这样的语言，规范LR(1)表将有几千个状态!

LALR(1)分析

- 对LR(1)项目集规范族合并同心集，若合并同心集后不产生新的冲突，则为LALR(1)项目集。
- 相应的分析方法即为LALR(1)分析法。

分析可发现上例I_3和I_6， I_4和I_7， I_8和I_9分别为同心集

I_3:
S → a•B, a/b
B → •aB, a/b
B → •b, a/b

I_6:
S → a•B, #
B → •aB, #
B → •b, #

合并为 →

$I_{3,6}$:
S → a•B, a/b/#
B → •aB, a/b/#
B → •b, a/b/#

I_4:
B → b•, a/b

I_7:
B → b•, #

合并为 →

$I_{4,7}$:
B → b•, a/b/#

I_8:
B → aB•, a/b

I_9:
B → aB•, #

合并为 →

$I_{8,9}$:
B → aB•, a/b/#

合并同心集的几点说明

- 同心集合并后心仍相同，只是超前搜索符集合为各同心集超前搜索符的和集;
- 合并同心集后转换函数自动合并;
- LR(1)文法合并同心集后只可能出现归约-归约冲突，而没有移进-归约冲突;
- 合并同心集后可能会推迟发现错误的时间，但错误出现的位置仍是准确的。

状态	ACTION			GOTO	
	a	b	#	S	B
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

合并同心集后

状态	ACTION			GOTO	
	a	b	#	S	B
0	$S_{3,6}$	$S_{4,7}$		1	2
1			acc		
2	$S_{3,6}$	S_7			5
3,6	$S_{3,6}$	S_4			8,9
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

LR分析法总结

1. 适用文法范围:

   LR(0)⊂SLR(1) ⊂ LR(1) ⊂无二义文法

2. 分析表大小:

   LR(0) 和SLR(1) 较小，规范 LR(1)最大，LALR(1)适中。

3. 报错效率:

   LALR(1)会迟报（但不会漏报）

习题：P112 1，2 （规范句型活前缀）

P117 1 （有效项目集）

P122 1，2 （LR(0)文法）

P124 1，2，5 （SLR(1)文法）

P130 1，2 （选作 LR(1)文法）