第十二章　语法分析(提高部分)

- 自顶向下分析法>
- 自底向上分析法>

复习：第一章 概述

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

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 $S \notin L(G[Z])$

? 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 $S \notin L(G[Z])$

? 主要问题：
- 句柄的识别问题

▪ 主要方法：
- 算符优先分析法
- LR分析法

12.1 自顶向下分析

自顶向下分析的一般过程

给定符号串S，若预测是某一语法成分，则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]为某语法成分的文法
若不成功, 则 $S \notin L(G[Z])$

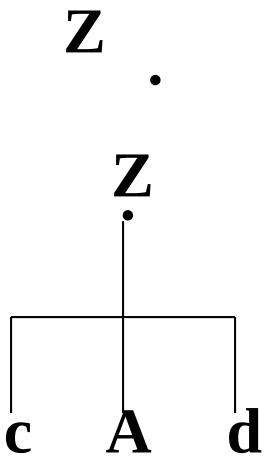
- 可以通过一例子来说明语法分析过程

例:

S = cad
G[Z]:
Z∷=cAd
A∷=ab|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



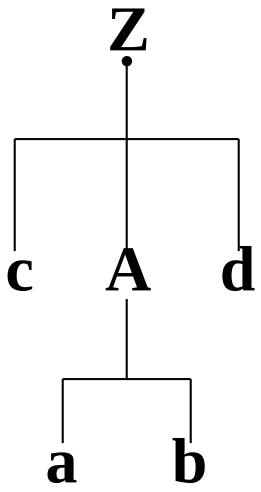
2. 用Z的右部符号串去匹配输入串

完成一步推导Z⇒cAd
检查, c-c匹配
A是非终结符,将匹配任务交给A

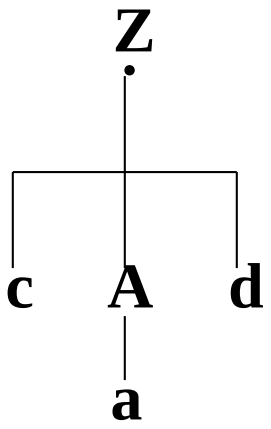
S=cad　G[Z]: Z∷=cAd
A∷=ab|a

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⇒ab
检查,a-a匹配,b-d不匹配(失败)
但是还不能冒然宣布S∉L(G[Z])



4. 回溯 即砍掉A的子树
改选A的第二右部
A⇒a　检查 a-a匹配
d-d匹配



建立语法树,末端结点为cad,与输入cad相匹配,
建立了推导序列 Z⇒cAd⇒cad
∴cad∈L(G(Z))

自顶向下分析方法特点：

1. 分析过程是带预测的，对输入符号串要预测属于什么语法成分，然后根据该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) 来建立语法树的过程, 由于是试探过程, 难免有失败, 所以分析过程需进行回溯, 因此也称这种方法是带回溯的自顶向下分析方法。

3. 最左推导可以编写程序来实现, 但带溯的自顶向下分析方法在实际上价值不大，效率低。

12.1 LL分析法

举例：

<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>‘)’

if (i+i) then i:=i*i+i else

12.1 LL分析法

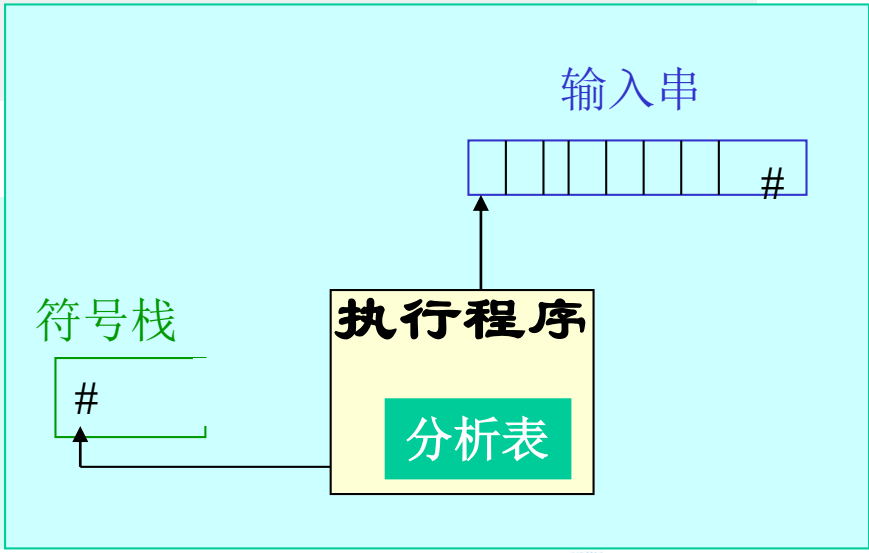
LL－自左向右扫描、自左向右地分析和匹配输入串。

∴ 分析过程表现为最左推导的性质。

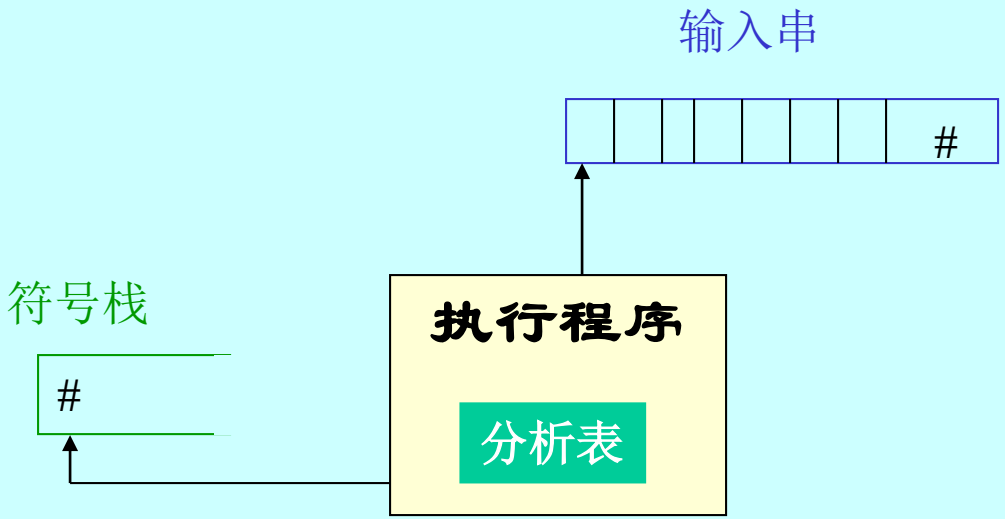
1、LL分析程序构造及分析过程

由三部分组成:

- 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，统一以‘#’表示。



（1）、分析表：二维矩阵M

$$M[A,a]=\begin{cases} A::=\alpha_i & \alpha_i \in V^* \\ \text{或} & A \in V_n \\ \text{error} & a \in V_t \text{ or } \# \end{cases}$$

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$\alpha_i \Rightarrow^{*} a\ldots$;

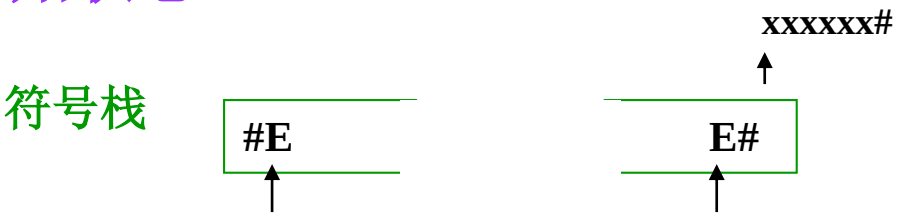
当$\alpha_i = \varepsilon$时，则a为A的后继符号。

M[A,a] = error

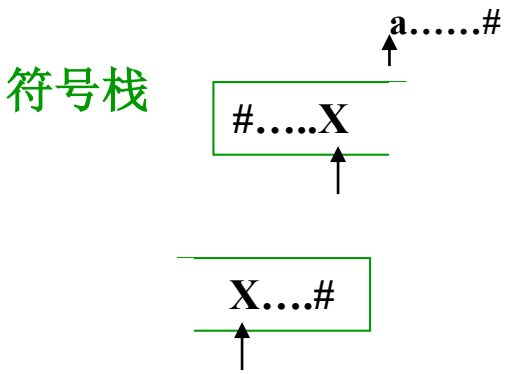
表示当用A去匹配输入串时，若当前输入符号为a，则不能匹配，表示无$A \not\Rightarrow^{*} a\ldots$,或a不是A的后继符号。

（2）符号栈： 有四种情况

- 开始状态



- 工作状态



查分析表得：

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

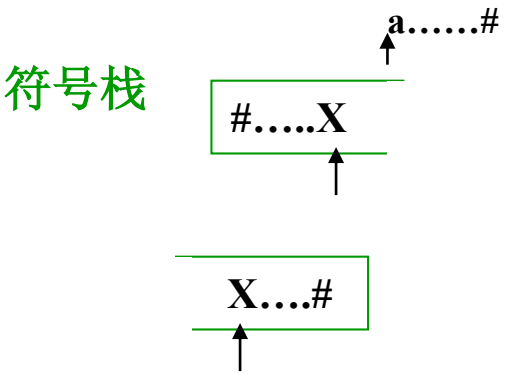
$X \overset{+}{\Rightarrow} a\ldots$

$X \in V_t$, $X = a$

	a	
X	$X::=\alpha_i$	

	a	
X	error	

- 出错状态



查分析表得：

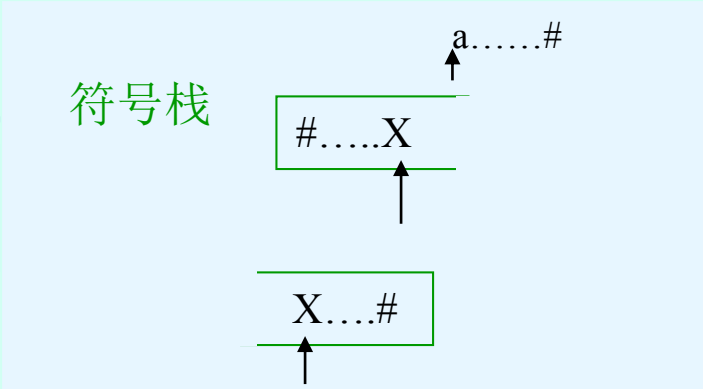
$X \in V_n$, $M[X,a] = error$

无$X \overset{+}{\Rightarrow} a\ldots$

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

- 结束状态

（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，读入下一个符号，重复下述过程直到正常结束或出错。

2. 测定栈顶符号X和当前输入符号a,执行如下操作：

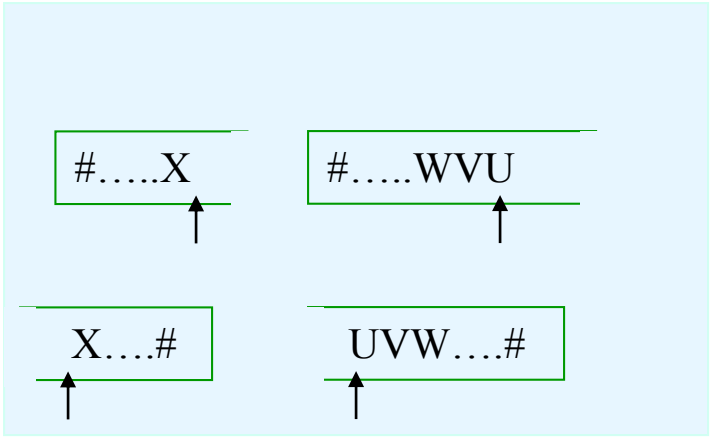
(1) 若X=a=#，分析成功，停止。E匹配输入串成功。
(2) 若X=a≠#，把X推出栈，再读入下一个符号。
(3) 若X∈V_n，查分析表M。

(3) 若X∈V_n，查分析表M

a) M[X,a] = X∷= UVW
则将X弹出栈，将UVW压入
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。



	a	
X	X::=UVW	

例：文法G[E]

E ::= E +T | T

T ::= T * F | F

F ::= (E)|i

E ::= TE’

E’ ::= +TE’|ε

T ::= FT’

T’ ::= *FT’|ε

F ::= (E)|i

- 消除左递归
- **P∷=Pα| β**
- 改写为：
- **P ∷= β P’。 P’ ∷= αP’| ε**

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::=TE'			E::= TE'		
E'	F::=i	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)		

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	i	+i*i#	
2.	#E’T	i	+i*i#	E::=TE’
3.	#E’T’F	i	+i*i#	T::=FT’
4.	#E’T’ i	i	+i*i#	F::= i
5.	#E’T’	+	i*i#	(出栈，读下一个符号)
6.	#E’	+	i*i#	T ’ ::= ε
7.	#E’T+	+	i*i#	E’::=+TE’

	i	+	*	(	)	#
E	E::=TE'			E::= TE'		
E'	F::=i	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)		

步骤

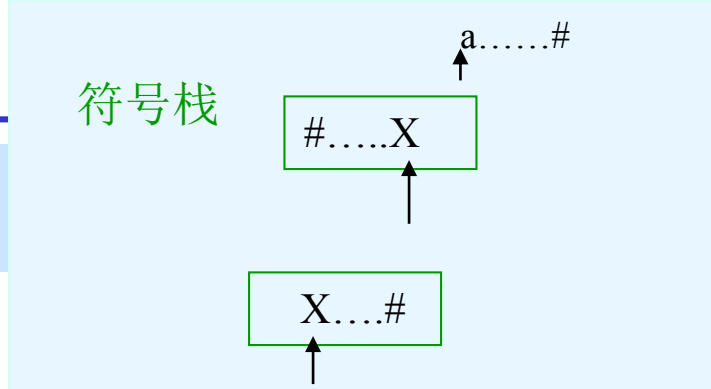
8.	#E’T	i	*i#	
9.	#E’T’F	i	*i#	T::=FT’
10.	#E’T’ i	i	*i#	F::= i
11.	#E’T’	*	i#	
12.	#E’T’F*	*	i#	T’::=*FT’
13.	#E’T’F	i	#	
14.	#E’T’ i	i	#	F::= i
15.	#E’T’	#		
16.	#E’	#		T’::= ε
17.	#	#		E’::= ε

推导过程：

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow iT'E' \Rightarrow iE'$$
$$\Rightarrow i+TE' \Rightarrow i+FT'E' \Rightarrow i+iT'E'$$
$$\Rightarrow i+i*FT'E' \Rightarrow i+i*iT'E'$$
$$\Rightarrow i+i*iE' \Rightarrow i+i*i$$

最左推导。

复习执行程序



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，读入下一个符号，重复下述过程直到正常结束或出错。

2. 测定栈顶符号X和当前输入符号a,执行如下操作：

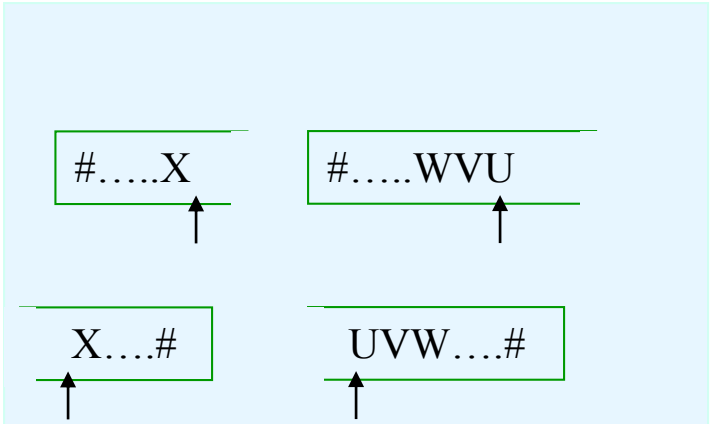
(1) 若X=a=#，分析成功，停止。E匹配输入串成功。
(2) 若X=a≠#，把X推出栈，再读入下一个符号。
(3) 若X∈V_n，查分析表M。

(3) 若X∈V_n，查分析表M。

a) M[X,a] = X::= UVW
则将X弹出栈，将UVW压入
注：U在栈顶 （最左推导）

b) M[X, a] = error 转出错处理

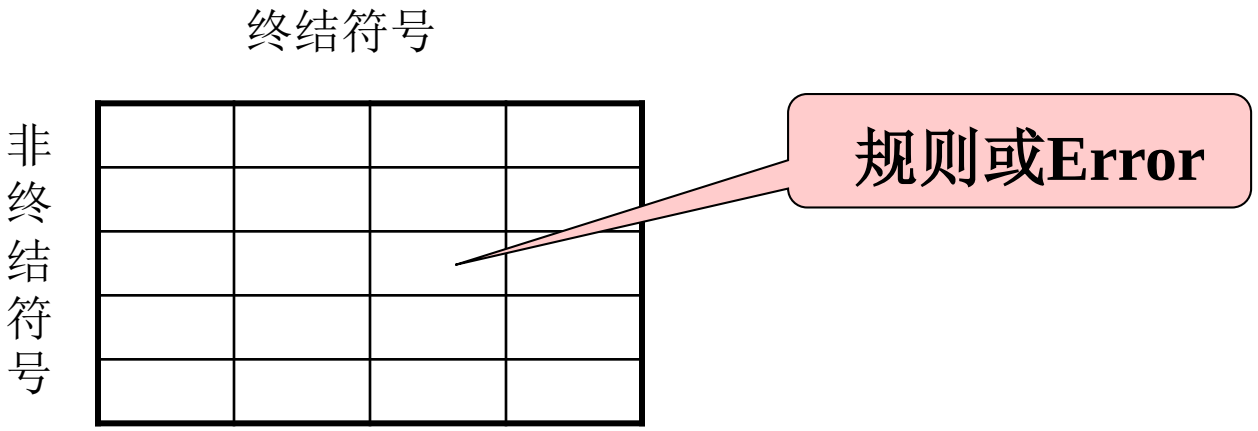
c) M[X, a] = X::=ε，
— a为X的后继符号
则将X弹出栈 (不读下一符号)
继续分析。



会编写算法。

2、分析表的构造

目标：



a) M[X, a] = error　转出错处理

b) M[X,a] = X∷= UVW ———需要求First集

c) M[X, a] = X::=ε，

— a为X的后继符号———如何知道X的后继符号，引入Follow（X）

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

$A\in V_n$ ， Z识别符号

该集合称为A的后继符号集合。

特殊地：若$Z \overset{*}{\Rightarrow} \ldots A$ 则 $\#\in FOLLOW(A)$

构造集合FIRST的算法

设$\alpha=X_1X_2...X_n$，　　$X_i \in V_n \cup V_t$　（即 $X_i \in V$ ）
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 $X_i ::= a.....|\varepsilon$，　$a \in V_t$
　　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($X_i$)

FIRST(X_i) $\Leftarrow$ FIRST(y_1) $-\{\varepsilon\}$;

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

若$\varepsilon \in$FIRST(y_1)

$\varepsilon \in$FIRST(y_2) 则将FIRST(y_3)-$\{\varepsilon\}$加入FIRST(X_i)

.....

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

若$\varepsilon \in$FIRST(y_1) ～ FIRST(y_k)

则将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

★ 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

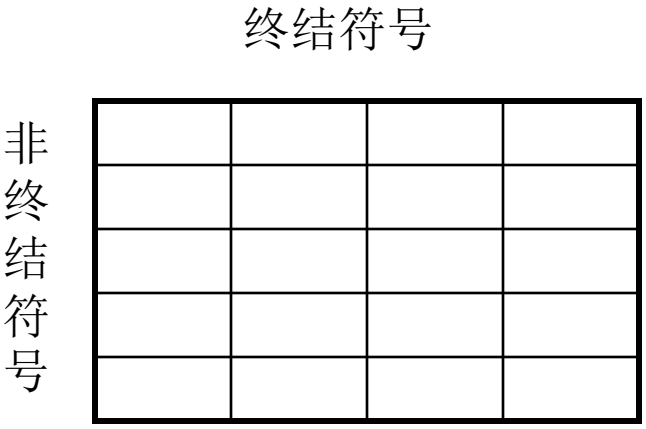
(2) 若A::=αBβ (β≠ε),则把FIRST(β)-{ε}加入FOLLOW(B)

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

注：FOLLOW集合中不能有ε

2、构造分析表

基本思想是:
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的哪一条规则去匹配输入串(即进行一步最左推导)



根据这个思想, 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A::= α_i ==> 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（A），

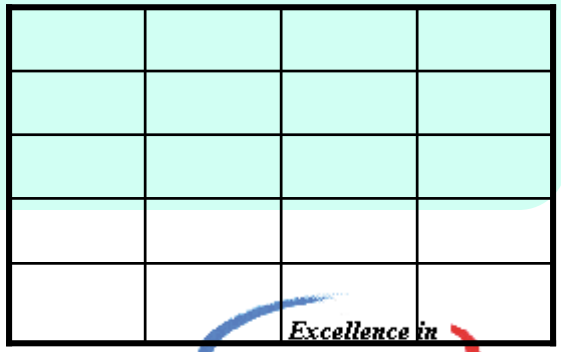
则A::= α_i ==>M[A,a]，表示A已经匹配输入串成功，

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

终结符号

非终结符号



例：G[E]分析表的构造

E::=TE’

E’::=+TE’|ε

T::=FT’

T’::=*FT’|ε

F::=(E)|i

求FIRST：

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(*FT’)={*}　　FIRST(ε)={ε}

FIRST(（E）)={(}　　FIRST(i)={i}

求FOLLOW

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

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

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

FOLLOW(T’)= FOLLOW(T)= {+,),#}
∵T::=FT’　∴ FOLLOW(T)加入FOLLOW(T’)

FOLLOW(F)={*, + ,), #} ∵T::=FT’ ∴ FOLLOW(F)=FIRST（T’）-{ε}
又T’$\overset{*}{\Rightarrow}$ε ∴ FOLLOW(T)加入FOLLOW(F)

构造分析表

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

FIRST(TE’)= {(,i}
FIRST(+TE’)={+}
FIRST(FT’)={(,i}
FIRST(*FT’)={*}
FIRST(（E）)={(}

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

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A::= α_i ==> M[A,a]

表示：A在栈顶，输入符号是a，应选择α_i 去匹配

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

FOLLOW(E)={#,)} FOLLOW(E’)={#,)}
FOLLOW(T)={+,), #} FOLLOW(T’)= {+,),#}
FOLLOW(F)={*, + ,), #}

	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)		

2、若αi =ε或αi ⇒ ε,而且a ∈FOLLOW（A），
则A::= αi ==>M[A,a]，表示A已经匹配输入串成功，
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

	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]=一条的规则或Error。**对于能用上述算法构造分析表的文法称为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(即分析表中无二条以上规则)，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的每一个非终结符A的任意两条规则A::=α|β,下列条件成立：

1、FIRST(α) ∩ FIRST(β) = Φ

2、若β=*=> ε, 则FIRST(α) ∩ FOLLOW(A) = Φ

用此构造分析表的算法,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是左递归的,或者是二义性的文法,则至少有一个多重入口。

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个最左推导,即在推导的某些步上存在多重定义,有两条规则可用，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

----- 自学 ---要求

作业：p264:1,2,6，补充题

补充题：有文法G[S]

S::= iCtSS’|a

S’::=eS|ε

C ::= b

(1) 证明该文法是二义性文法

(2) 求FIRST和FOLLOW

(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（不要求)

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，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的同步符号集合，跳过输入符号直到出现FOLLOW(A)的元素，便把A从栈中弹出，继续往下分析。

2) 为了避免仅按1)来确定同步符号集合会使跳读过多(如输入串中缺少语句结束符号“;”)，可将程序高层语法结构(成分)的开始符号(通常是关键词)加入到低层语法结构的同步集合中。

3) 把FOLLOW(A)的符号加入A的同步集合中。

4) 如果栈顶的非终结符号A可以产生空串,可以将A从栈中弹出。

5) 如果终结符在栈顶而不能匹配,则可弹出该终结符,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。

复习： 语法分析

语法分析方法：
- 自顶向下分析法 $Z \overset{+}{\Rightarrow} S$
- 自底向上分析法 $S \overset{+}{\Leftarrow} Z$

$S \in L[Z]$

(一) 自顶向下分析

①概述自顶向下分析的一般过程

存在问题
- 左递归问题 ——— 消除左递归的方法
- 回溯问题 ———
  - 无回溯的条件
  - 改写文法
  - 超前扫描

②两种常用方法:

(1)递归子程序法
- a)改写文法,消除作递归,回溯
- b)写递归子程序

手工编程

(2)LL(1)分析法
- LL(1)分析器的逻辑结构及工作过程
- LL(1)分析表的构造方法
  - 1.构造First集合的算法
  - 2.构造Follow集合的算法
  - 3.构造分析表的算法
- LL(1)文法的定义以及充分必要条件

自动生成

- ***Morden Compiler Implementation in Java***

JAVACC

JavaCC 是一个LL(k)的语法分析生成器。

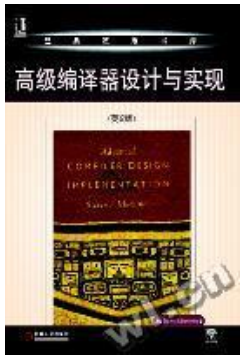
SABLECC

SableCC是一个LALR(1)的语法分析生成器

著名教材

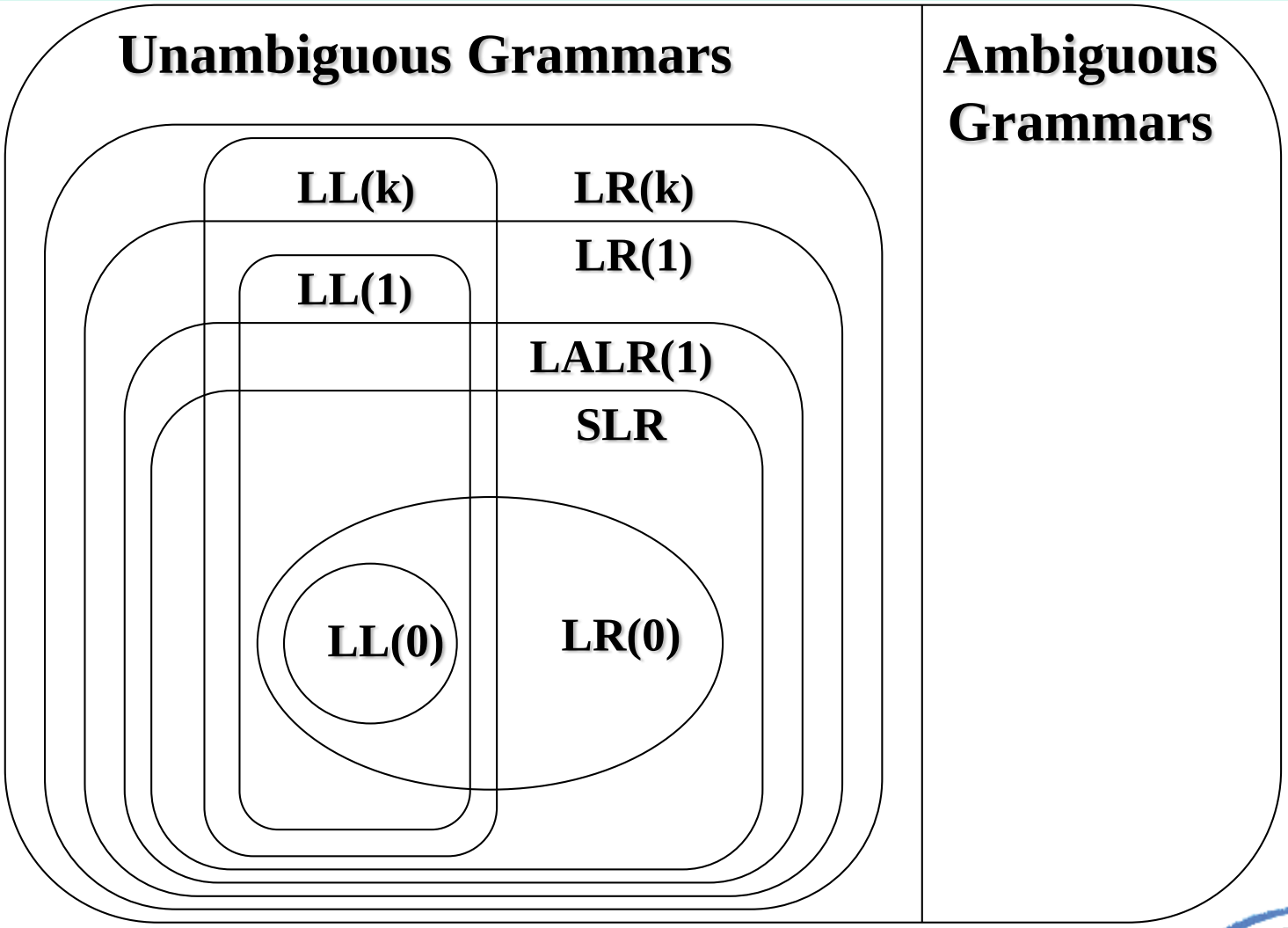
- 1.**龙书(Dragon book)**：书名是Compilers: Principles,Techniques,and Tools；作者是：Alfred V.Aho,Ravi Sethi,Jeffrey D.Ullman ；国内所有的编译原理教材基本都是参考了本书，重点是编译的前端技术。

- 2.**鲸书(Whale book)**：书名是：Advanced Compiler Design and Implementation；作者是：Steven S.Muchnick ；也就是高级编译原理。



- 3.**虎书(Tiger book)**：书名是：Modern Compiler Implementation in Java/C++/ML, Second Edition；作者是：Andrew W.Appel,with Jens Palsberg 。这本书是3本书中最薄的一本，也是最最牛的一本！

不同文法类的层次结构


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LR(1)
LL(1)
LALR(1)
SLR
LL(0)
LR(0)

第十二章　语法分析

- 语法分析的功能、基本任务
- 自顶向下分析法
- 自底向上分析法

12.2 自底向上分析

基本算法思想:

若采用自左向右的描述和分析输入串,那么自底向上的基本算法是:

从输入符号串开始，通过重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归约，若能归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 归约为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。

例：文法G[E]

(1) E ::= E+T

(2) E ::= T

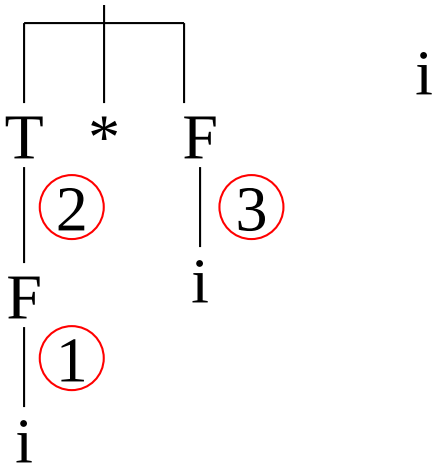
(3) T ::= T*F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

句子：i*i+i



例：文法G[E]

(1) E ::= E+T

(2) E ::= T

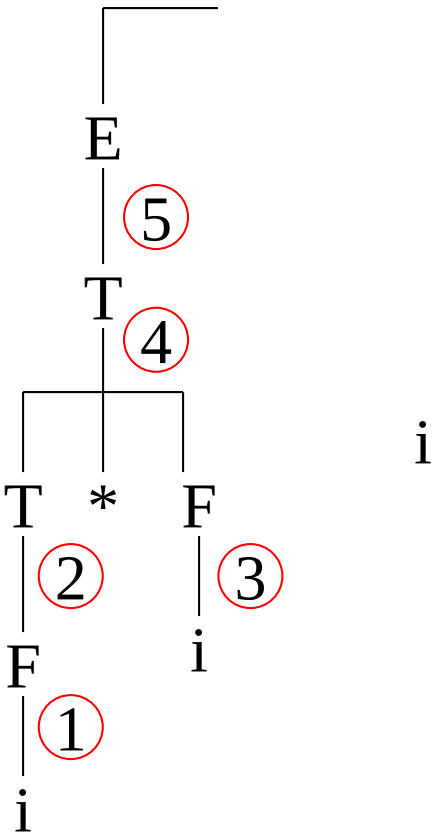
(3) T ::= T*F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

句子：i*i+i



例：文法G[E]

(1) E ::= E+T

(2) E ::= T

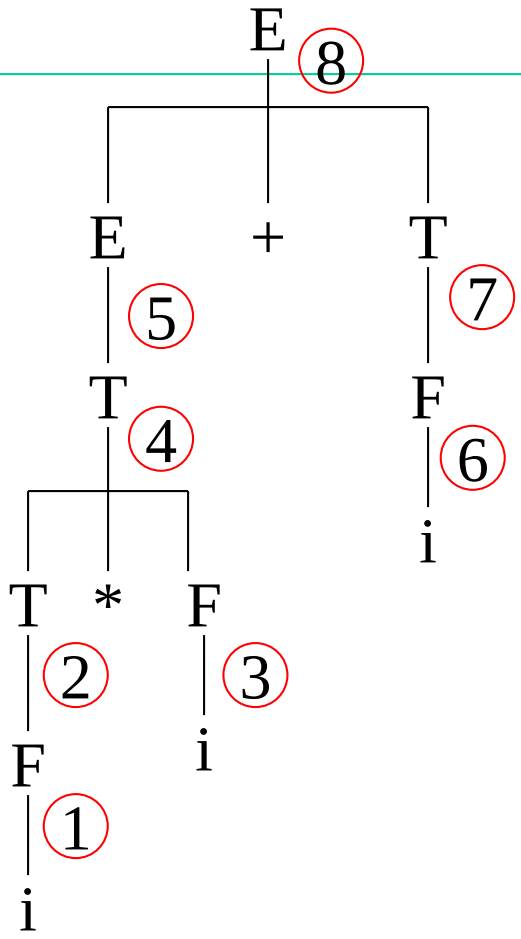
(3) T ::= T*F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

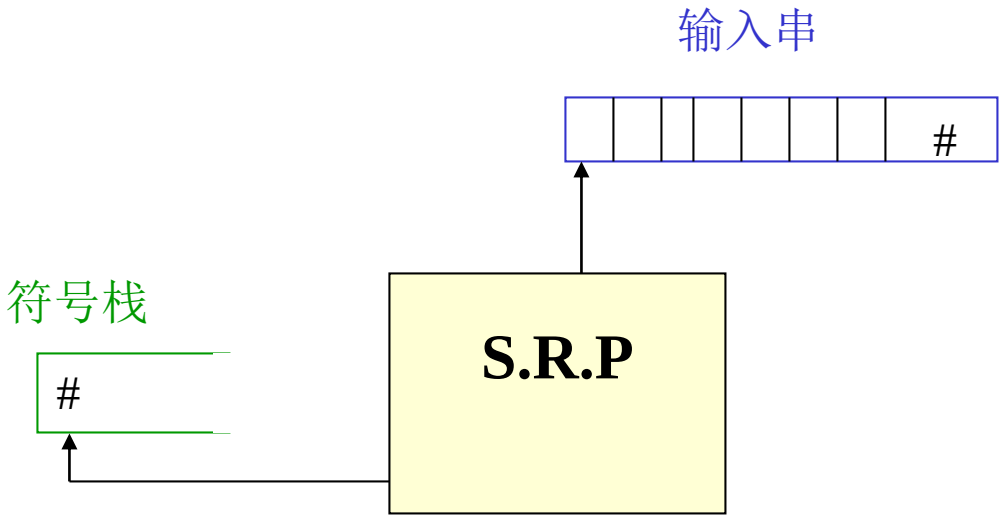
句子：i*i+i

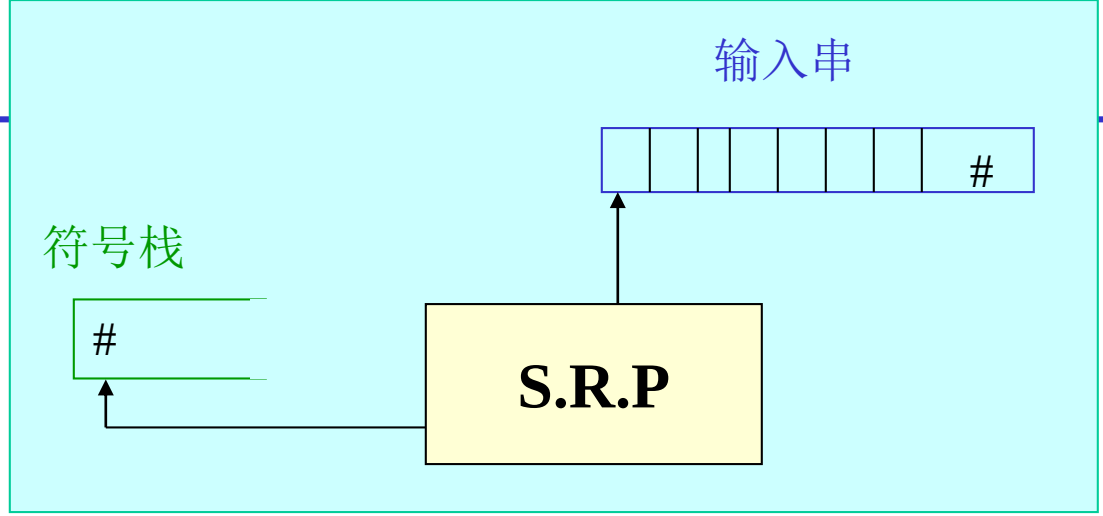


- 自底向上分析的一般过程（移进-归约分析）

移进—归约分析（Shift-reduce parsing)

要点： 建立符号栈，用来记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归约。





分析过程：

1）把输入符号串按扫描顺序一一地移进符号栈（一次移一个）

2）移进过程中，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，就根据规则进行归约：

- 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，若有就再进行归约，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：

S ::= aAcBe　　abbcde

A::= b

A::= Ab

B::= d

输入串为abbcde，检查是否是该文法的合法句子：

若采用自底向上分析，即能否一步步归约当前句型的句柄，最终归约到识别符号S。先设立一个符号栈，将符号“#”作为待分析的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:

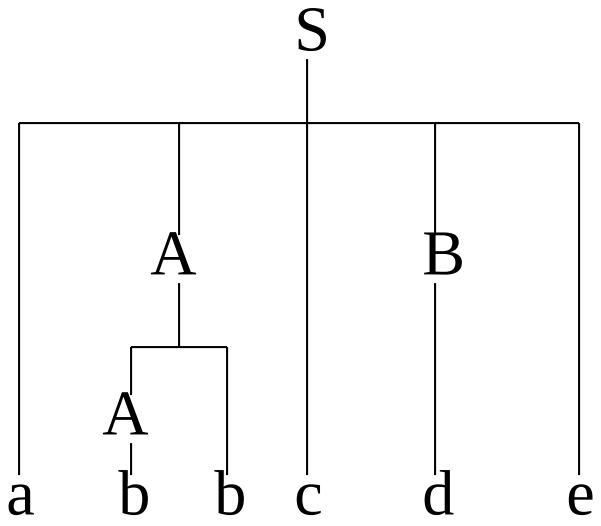
例：G[S]: S∷=aAcBe A∷=b
A∷=Ab B∷=d

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#	归约(A∷=b)
5	#aAb	cde#	移进
6	#aA	cde#	归约(A∷=Ab)
7	#aAc	de#	移进
8	#aAcd	e#	移进
9	#aAcB	e#	归约(B∷=d)
10	#aAcBe	#	移进
11	#S	#	归约(S∷=aAcBe)
12	#S	#	成功

这一方法简单明了,不断地进行移进归约,关键是确定当前句型的句柄。

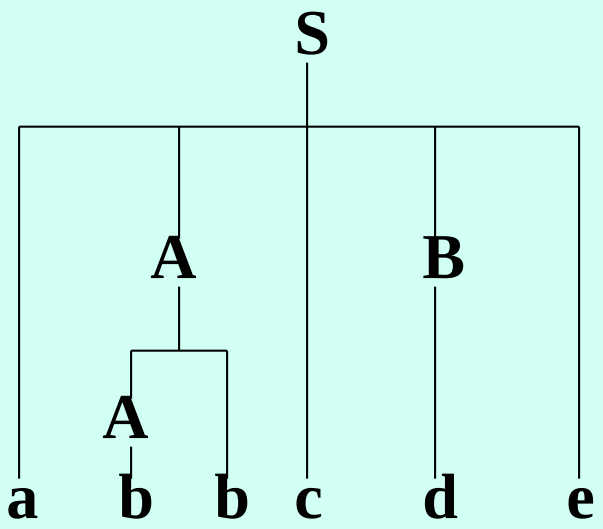
说明: 1) 例子的分析过程是一步步地归约当前句型的句柄

该句子的唯一语法树为:



例：G[S]:　S∷=aAcBe　A∷=b

A∷=Ab　B∷=d



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

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。

上述分析过程是怎样识别句柄的，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。

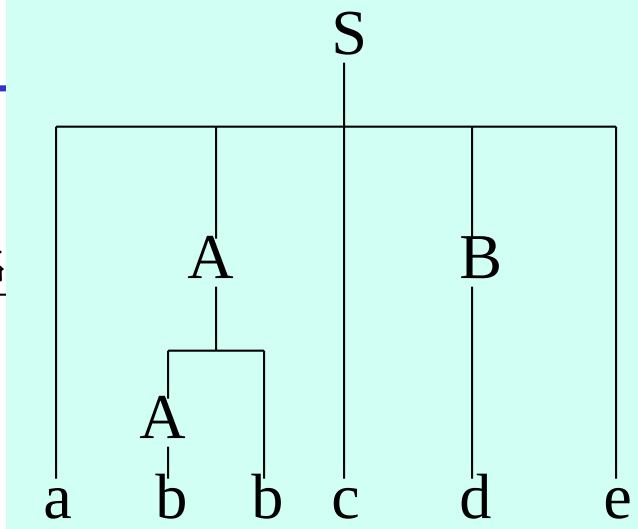
因为不能认为: **对句型 xuy 而言**

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 $Z \overset{*}{\Rightarrow} xUy$

例：G[S]:　S∷=aAcBe　A∷=b

A∷=Ab　B∷=d



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

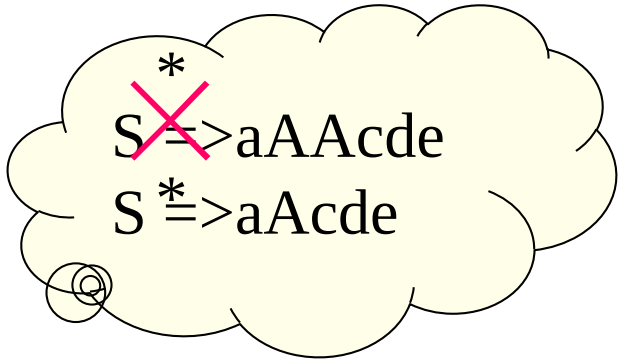
步骤	符号栈	输入符号串	当前句型
5	#aAb	cde#	aAbcde

A::= b ∴ A => b

A::= Ab ∴ A => Ab

若用A::= b 归约， 得 aAAcde

若用A::= Ab归约，得 aAcde



S::= abABC

A::= a | d

B::= b | e

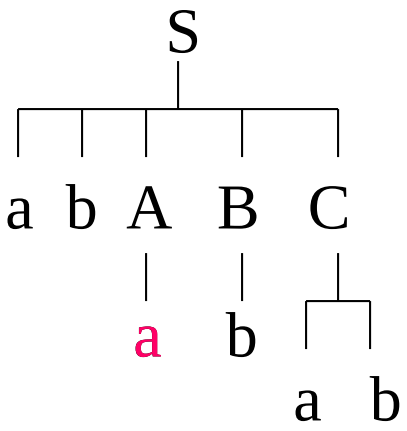
C::= ab | f

句子：ababab

先归约a ？

先归约b?

先归约ab?



12.3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无关的文法。

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，作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，规定运算符之间的优先关系。

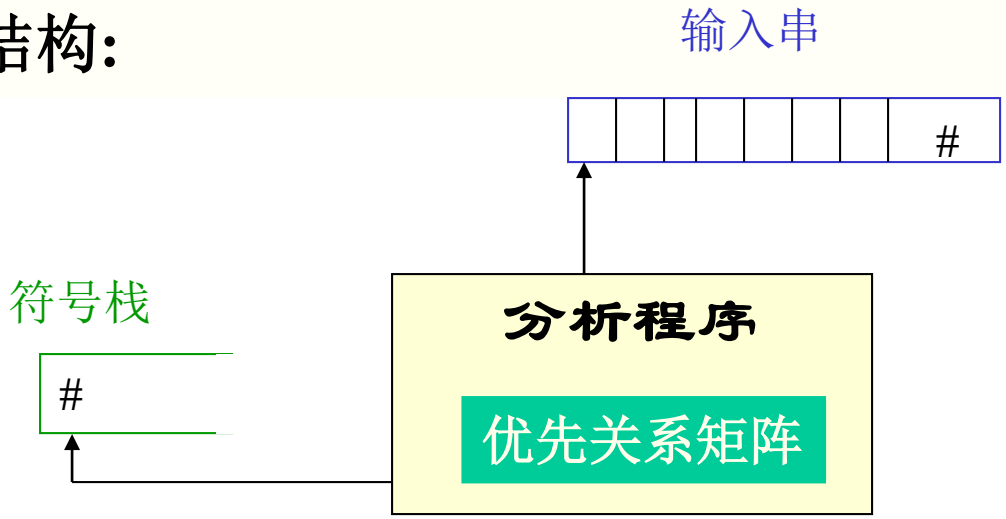
运算法则:

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”，并进行归约。

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

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

1) 例中文法终结符之间的优先关系可以用一个矩阵M来表示

b(右,栈外) / a(左,栈内)	+	*	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，则进行归约，否则移进。

E∷=E+E | E*E | (E) | 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>+	归约
3	#E	+i*i#	#<+	移进
4	#E+	i*i#	+<i	移进
5	#E+i	*i#	i>*	归约
6	#E+E	*i#	+<*	移进
7	#E+E*	i#	*<i	移进
8	#E+E*i	#	i>#	归约
9	#E+E*E	#	*>#	归约
10	#E+E	#	+>#	归约
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”,并进行归约,直到识别符号E,最后分析成功:　　$i+i*i \in L(G[E])$

出错情况:

1. 相邻终结符之间无优先关系
2. 对双目运行符进行归约时,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归约（即不一定是规范归约）也就是每次归约不一定是归约当前句型的句柄，而是句柄的变形，但也是短语。

(2) 文法的终结符优先关系可以用一个矩阵表示,也可以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

算符优先函数值的

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

1. 把各算符优先级由小到大定为j=1～n

(+ *) i

1 2

2. 对于各算符的优先顺序

若为左结合,则 f (op)=2j g (op)=2j-1

若为右结合,则 f (op)=2j g (op)=2j

设m>2n, 则 f ()) = f (i) = m+1

g (() = g (i) = m，其他为0

f(#) = f(() = g(#) = g()) = 0

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

f(+) > g(+)　　左结合
f(+) < g(*)　　先乘后加
f(+) < g(()　　先括号内后括号外
:
:

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

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)

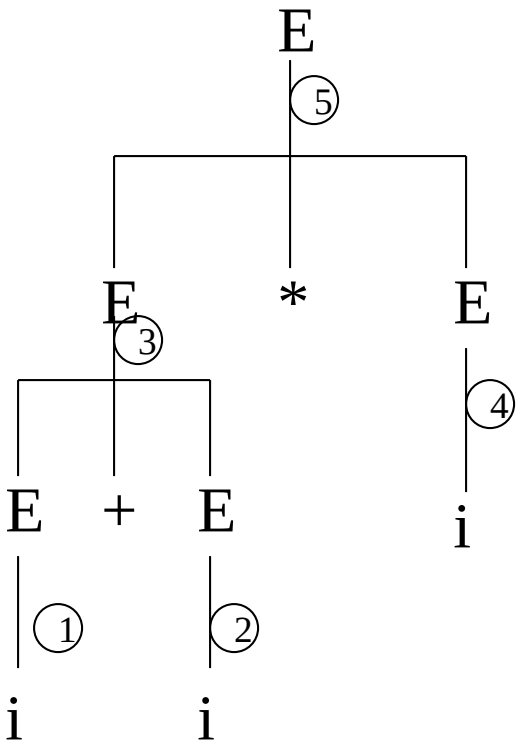
优点：便于比较,只需将输入符号与运算符栈的栈顶符号相比较

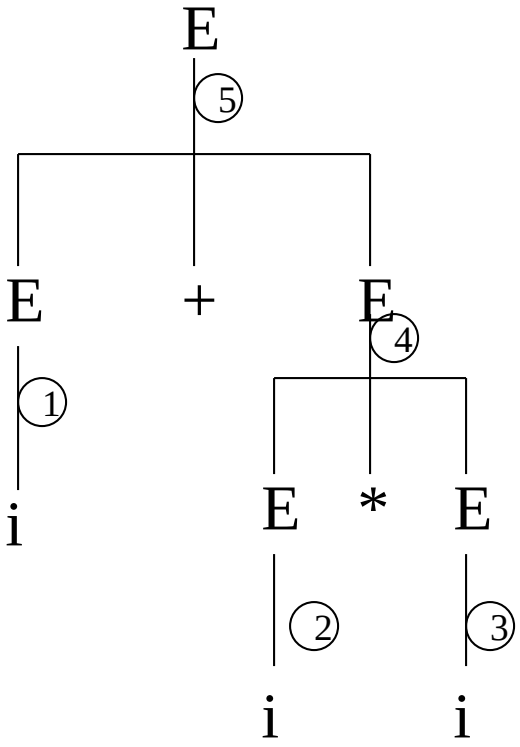
(4) 使用算符优先分析方法可以分析二义性文法所产生的语言

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





按规范归约,句柄不唯一, E+E*i　所以整个归约过程就不唯一,编译所得的结果也将不唯一。

问题

- 是否是所有文法都可以用算符优先分析法呢？
- 如何构造算符优先关系矩阵呢？

12.3.3 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。

优先关系的定义

若G是一OG文法，a,b∈V_t，　U,V,W∈V_n

分别有以下三种情况:

1) a=b iff 文法中有形如 U∷= …ab…或U∷= …aVb… 的规则。

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

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

例：

例： 文法G[E]

E::= E + T | T

T::= T * F | F

F::= (E) | i

1) a≐b iff 文法中有形如 U∷=...ab...或U∷= ...aVb...的规则。

2) a⋖b iff 文法中有形如 U∷=...aW...的规则，其中 W⇒ b...或W⇒ Vb...。

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

E::=E + T　　E => E + T　　∴ + ⋗ +

T => T * F　　∴ + ⋖ *

T =>F =>(E)　　∴ + ⋖ (

T =>F =>i　　∴ + ⋖ i

F::= (E)　　E => E + T　　∴ + ⋗)

∴ (≐)

∴ (⋖ +

算符优先文法（OPG）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文法的几点说明:

(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 “ $<\cdot$ ”、 “ $\cdot>$ ”， 需定义两个集合

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

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

- 求“$\dot{<}$”、“$\dot{>}$”：

若文法有规则
W∷=...aU...，对任何b, b∈FIRSTVT(U)
则有：$a \dot{<} b$

若文法有规则
W∷=...Ub...，对任何a, a∈LASTVT(U)
则有：$a \dot{>} b$

构造FIRSTVT(U)的算法

1) 若有规则U∷= b…或U∷= Vb…(存在$U \overset{+}{\Rightarrow} b\ldots$或$U \overset{+}{\Rightarrow} Vb\ldots$)
则b∈FIRSTVT(U)

2) 若有规则U∷= V…且b∈FIRSTVT(V), 则b∈FIRSTVT(U)

说明:因为$V \overset{+}{\Rightarrow} b\ldots$或$V \overset{+}{\Rightarrow} Wb\ldots$, 所以有$U \Rightarrow V\ldots \overset{+}{\Rightarrow} b\ldots$或
$U \Rightarrow V\ldots \overset{+}{\Rightarrow} Wb\ldots$

具体方法如下：

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
BEGIN  {main}
    FOR  每个非终结符号U和终结符b   DO
            F[U,b]:=FALSE;
    FOR 每个形如U::=b…或U::=Vb… 的规则  DO
            INSERT(U,b);
```

```
WHILE   S栈非空      DO
    BEGIN
        把S栈的栈顶项弹出,记为 (V,b)  /* b∈FIRSTVT(V)*/
        FOR    每条形如U::=V…的规则   DO
                    INSERT (U,b);   /* b∈FIRSTVT(U)*/
    END OF WHILE
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

$$FIRSTVT(U) = \{ b \mid F[U,b] = TRUE \}$$

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;
```

```
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$U::=X_1X_2\ldots X_n$ DO

FOR i:=1 TO n-1 DO

BEGIN

IF X_i和X_{i+1}均为终结符,THEN 置 $X_i=X_{i+1}$

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 $X_i=X_{i+2}$

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$X_i<b$

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$a>X_{i+1}$

END

（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他素短语。

例：文法G[E]

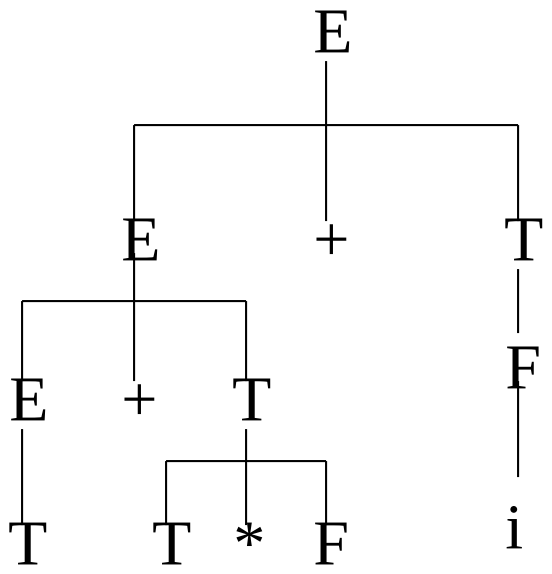
E::=E+T|T

T::=T*F|F

F::=(E)|i

求句型T+T*F+i 的素短语

文法的语法树：



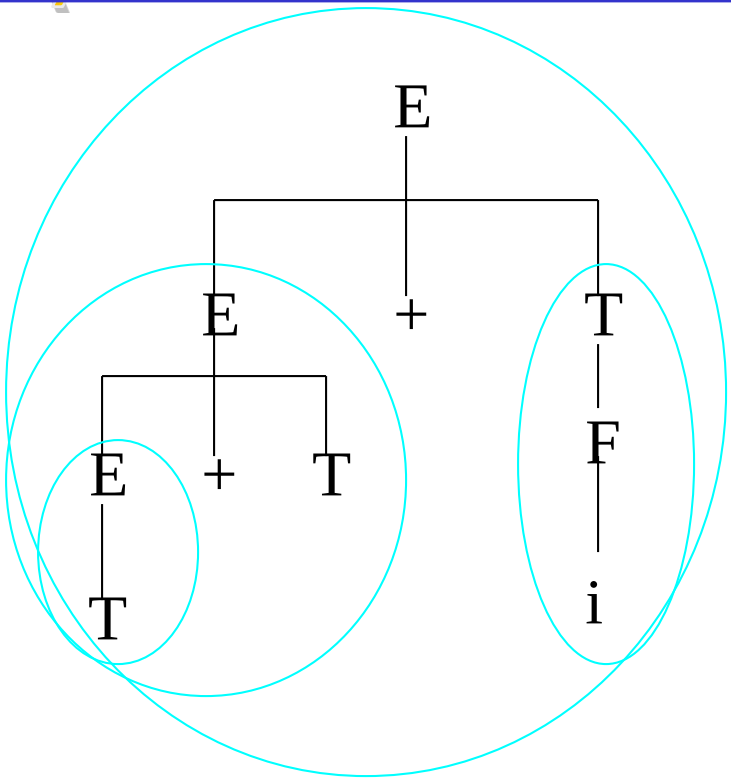
短语:T+T*F+i, T+T*F

T(最左), T*F, i

其中T不包含终结符,T是句柄

而T+T*F+i和T+T*F包含其他素短语。

只有T*F和i为素短语,其中T*F为最左素短语,而该句型句柄为T。



句型：T + T + i

短语：T+T+i

T +T

T

i

句柄：T

素短语：T+T, i

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:

$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$

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：$a_{j-1}N_ja_j\ldots N_ia_iN_{i+1}a_{i+1}$

其中 $a_{j-1}<a_j$

$a_j=a_{j+1}$, $a_{j+1}= a_{j+2}$,…, $a_{i-2}= a_{i-1}$, $a_{i-1}= a_i$

$a_i> a_{i+1}$

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。

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

★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,因为运算是中缀形式给出的(OPG文法的特点)NaNaNaN ⇒NaWaN

例: 文法G[E]

E::=E+T|T

T::=T*F|F

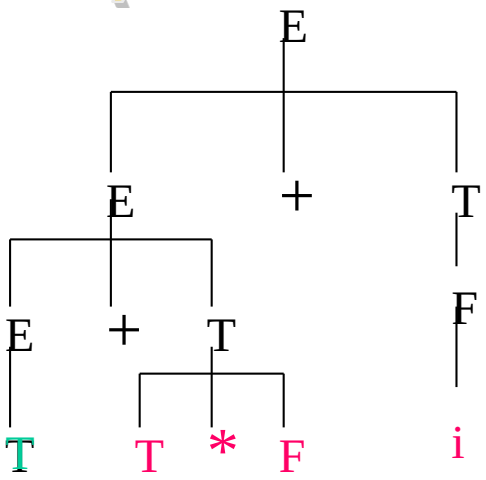
F::=(E)|i

分析文法的句型T+T*F+i

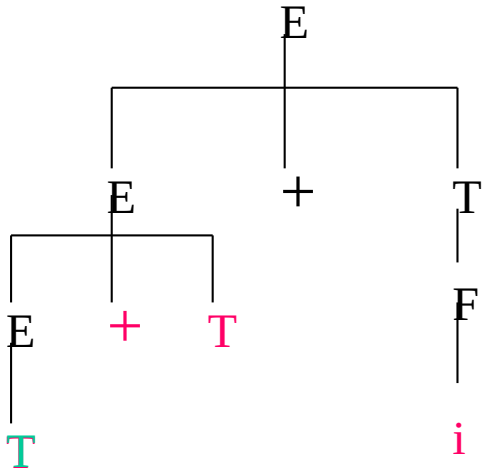
步骤	句型	关系	最左子串	归约符号
1	#T+T*F+i#	#<+<*>+<i>>#	T*F	T
2	#T+T+i#	#<+>+<i>>#	T+T	E
3	#E+i#	#<+<i>>#	i	F
4	#E+F#	#<+>#	E+F	E

可以看出:

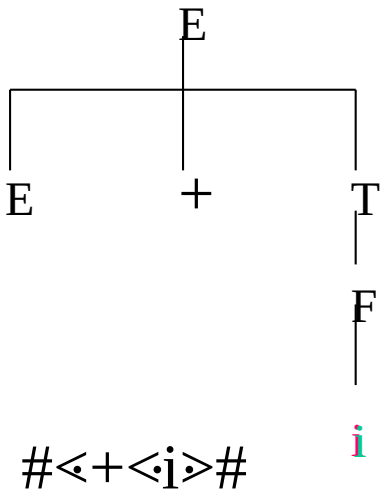
1. 每次归约最左子串,确实是当前句型的最左素短语(语法树）
2. 归约的不都是真句柄（仅i归约为F是句柄，但它是最左素短语)
3. 没有完全按规则进行归约,因为素短语不一定是简单短语



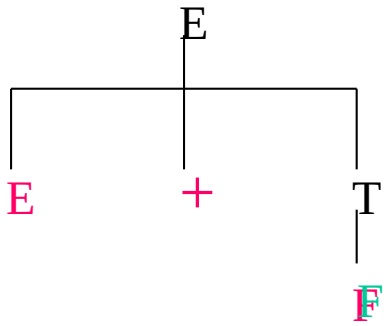
#<+<*>+<i>#



#<+>+<i>#



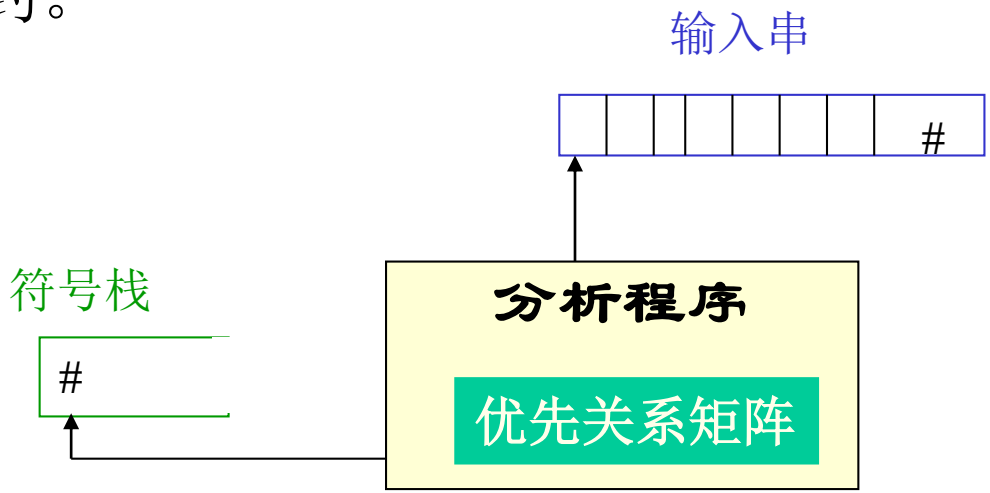
#<+<i>#



#<+>#

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）并进行归约。



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

自学书上例子。
说明：为什么每次归约的非终结符都用N？

1　分析过程中找素短语时，与非终结符的名字无关
2　实现时：

习题：P279 2(2), 4, 5

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. 构造算法优先关系矩阵
3. 判断该文法是否为算符优先文法。

12.4 LR分析法

1、概述

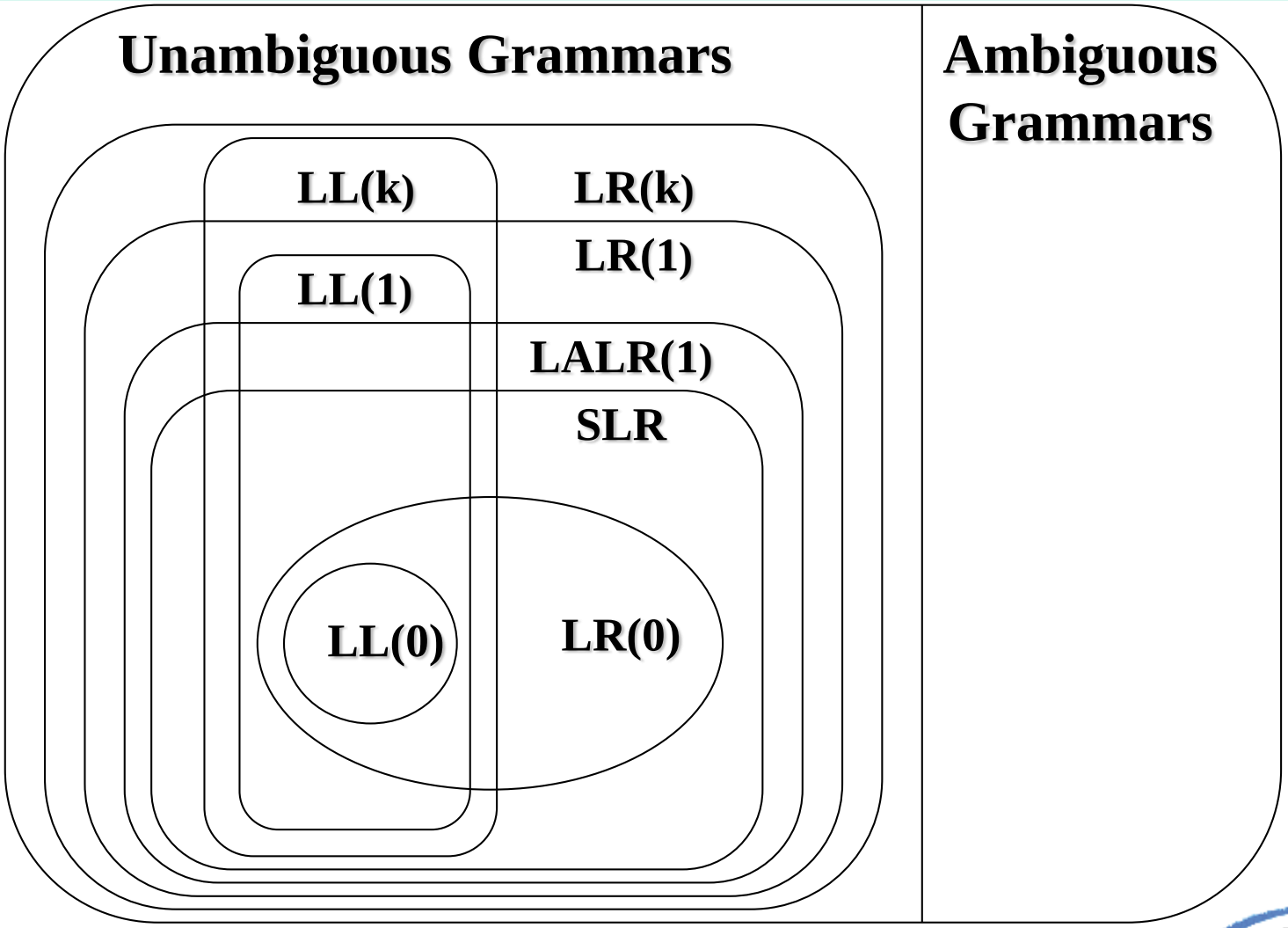
什么是LR分析：从左到右扫描(L)自底向上进行归约(R)(是规范归约），是自底向上分析方法的高度概括和集中

历史 + 展望 + 现状 => 句柄

(1) LR分析法的优缺点:

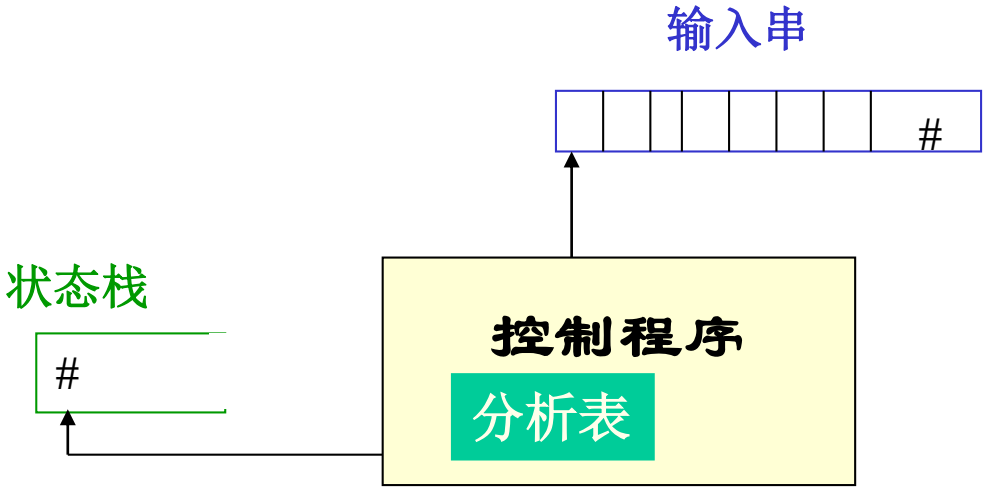
1) 适合文法类足够大
2) 分析效率高
3) 报错及时
4) 可以自动生成
5) 手工实现工作量大

不同文法类的层次结构


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LR(1)
LL(1)
LALR(1)
SLR
LL(0)
LR(0)

(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，是移进还是归约，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

a) 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。

b) LR分析表(规范LR分析表)

适用文法类最大,几乎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。

c) 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

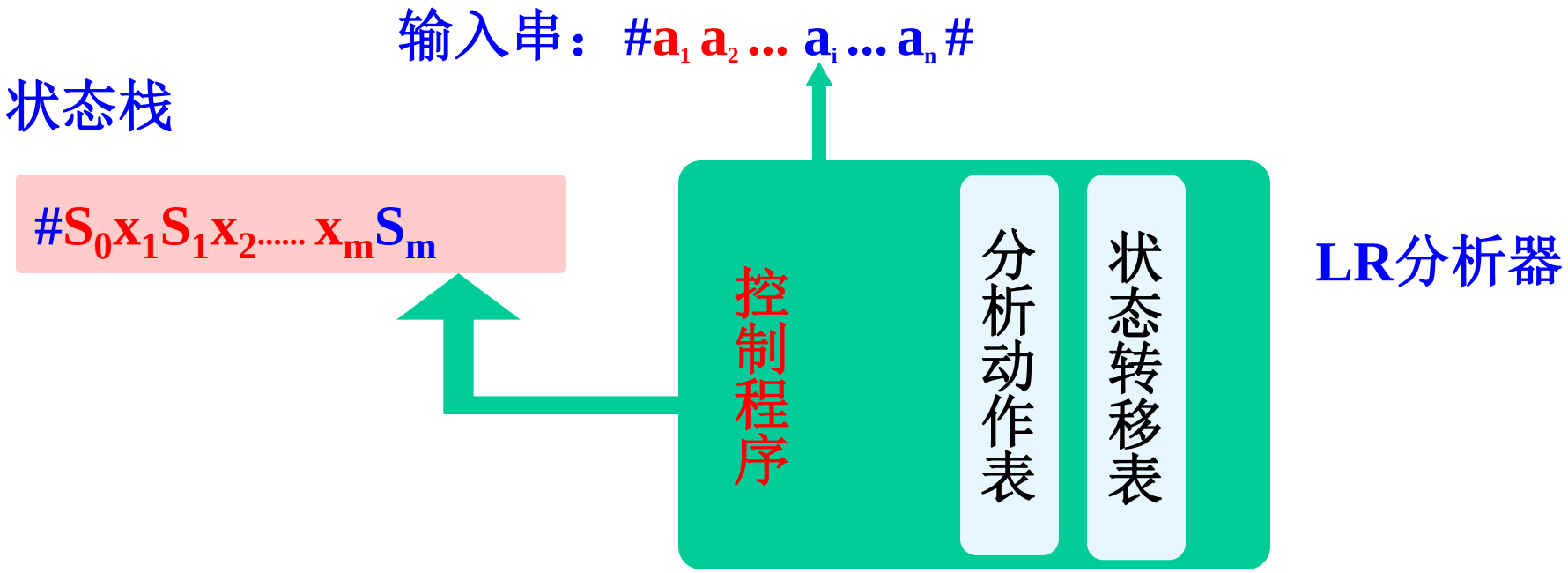
1. 三种分析表对应三类文法

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

（1）逻辑结构



$\#S_0x_1S_1x_2\cdots x_mS_m$ → $S_0S_1\cdots S_m$ / $\# x_1x_2\cdots x_m$

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: $x_1x_2\cdots x_m$

为从开始状态(S_0)到当前状态(S_m)所识别的规范句型的活前缀。

规范句型: 通过规范归约得到的句型。

规范句型前缀: 将输入串的剩余部分与其连接起来就构成了规范句型。

如： $x_1x_2\ldots x_m a_i \ldots a_n$为规范句型

活前缀: 若分析过程能够保证栈中符号串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= $u_1u_2\ldots u_r$
那么符号串$u_1u_2\ldots u_i(1\le i\le r)$即是句型αβt的活前缀

例: 有文法G[E]: E→T|E+T|E-T
T→i|(E)
拓广文法G'[S]: S→E#
E→T|E+T|E-T
T→i|(E)
句型E-(i+i) #
活前缀： E, E-, E-(, E-(i 是句型E-(i+i)# 的活前缀。

- 分析表

a. 状态转移表 (GOTO表)

是一个矩阵：
行---分析器的状态
列---文法符号

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(状态转移)

$\#S_0x_1S_1x_2\ldots\ldots x_{i-1}S_{i-1}\,x_iS_i$

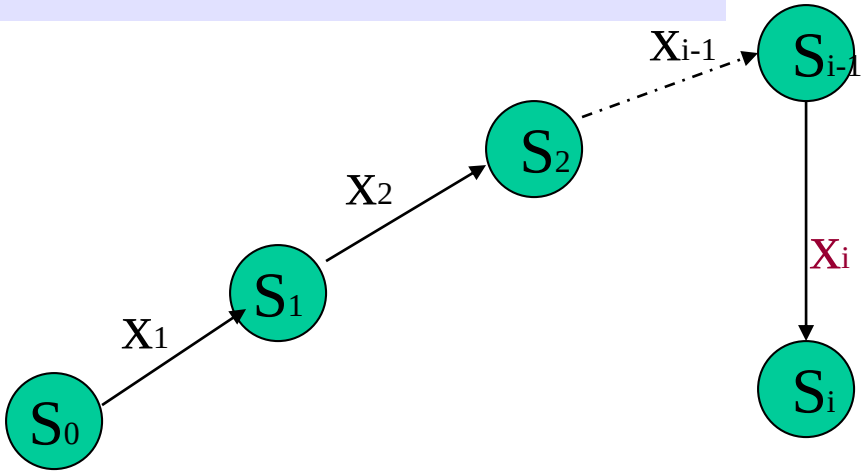
S_i需要满足条件是:

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。

通过对有穷自动机的了解, 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，并设置新的栈顶状态S_j

$S_j = GOTO[S_i,a]$，将指针指向下一个输入符号

(2) 归约(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，把A推进栈，并设置新的栈顶状态S_j

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

(3) 接受(accept)

ACTION[S_i,#]=accept

(4) 出错(error)

ACTION[S_i,a]=error

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

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　　GOTO 表

状态 \ 输入符号	i	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(1)E::=E+T

(2)E::=T

(3)T ::=T*F

(4)T::=F

(5)F::=(E)

(6)F::=i

ACTION 表 GOTO 表

状态 \ 输入符号	i	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			

(1)E::=E+T (2)E::=T

(3)T ::=T*F (4)T::=F

(5)F::=(E) (6)F::=i

分析过程 i*i+i

步骤	状态栈	符号	输入串	动作
1	#0	#	i*i+i#	初始化
2	#0i5	#i	*i+i#	S
3	#0F3	#F	*i+i#	r6
4	#0T2	#T	*i+i#	r4

ACTION 表 GOTO 表

状态 \ 输入符号	i	+	*	(	)	#	E	T	F
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(1)E::=E+T (2)E::=T

(3)T ::=T*F (4)T::=F

(5)F::=(E) (6)F::=i

分析过程 i*i+i

步骤	状态栈	符号	输入串	动作
1	#0	#	i*i+i#	初始化
2	#0i5	#i	*i+i#	S
3	#0F3	#F	*i+i#	r6
4	#0T2	#T	*i+i#	r4
5	#0T2*7	#T*	i+i#	S
6	#0T2*7i5	#T*i	+i#	S
7	#0T2*7F10	#T*F	+i#	r6

ACTION 表　　GOTO 表

输入符号 / 状态	i	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3

(1)E::=E+T (2)E::=T

(3)T ::=T*F (4)T::=F

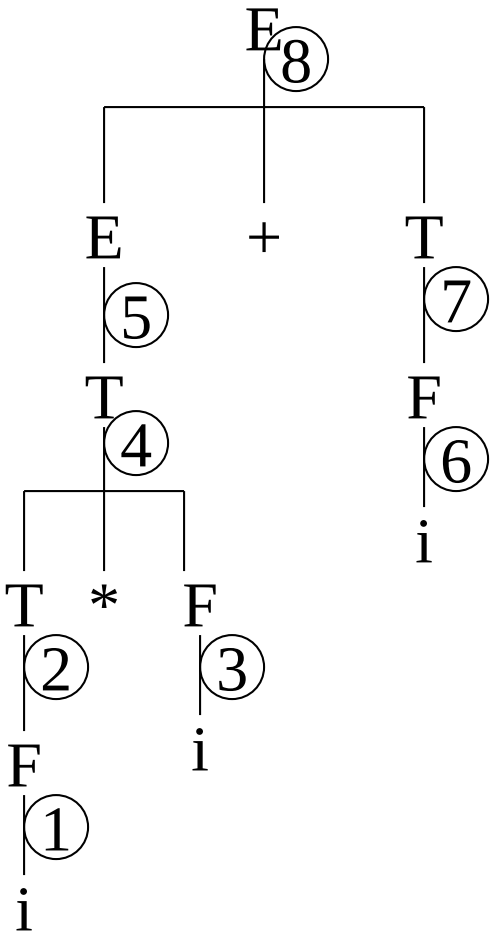
(5)F::=(E) (6)F::=i

8	#0T2	#T	+i#	r3
9	#0E1	#E	+i#	r2
10	#0E1+6	#E+	i#	S
11	#0E1+6i5	#E+i	#	S
12	#0E1+6F3	#E+F	#	r6
13	#0E1+6T9	#E+T	#	r4
14	#0E1	#E	#	r1
15		#E		accept

由分析过程可以看到:

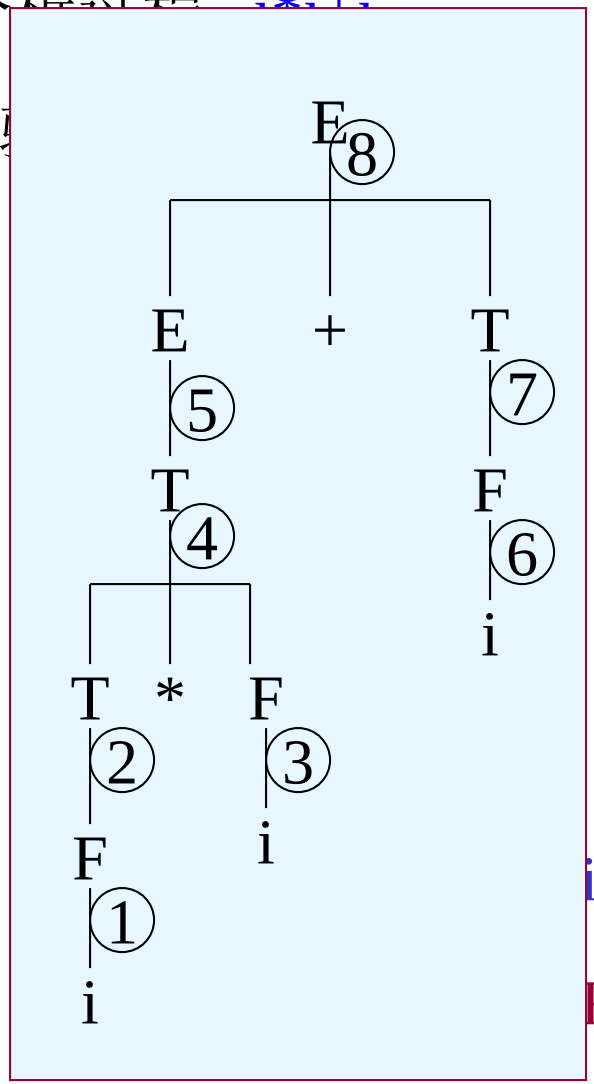
(1) 每次归约总是归约当前句型的句柄,是规范归约。
(算符优先分析归约最左素短语)

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

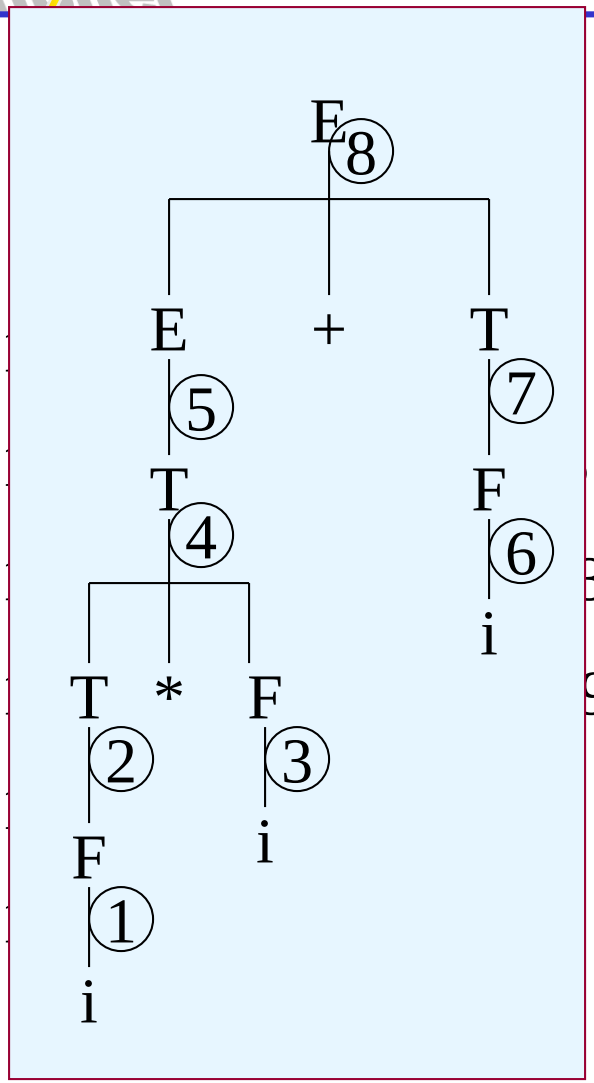


(1)E::=E+T (2)E::=T
(3)T ::=T*F (4)T::=F
(5)F::=(E) (6)F::=i

分析过程 i*i+i



步骤	符号	输入串	动作
	#	i*i+i#	初始化
	#i	*i+i#	S
	#F	*i+i#	r6
	#T	*i+i#	r4
	#T*	i+i#	S
i5	#T*i	+i#	S
F10	#T*F	+i#	r6



#T	+i#	r3
#E	+i#	r2
#E+	i#	S
#E+i	#	S
#E+F	#	r6
#E+T	#	r4
#E	#	r1
#E		accept

(1)E::=E+T (2)E::=T

(3)T ::=T*F (4)T::=F

(5)F::=(E) (6)F::=i

3、构造SLR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 $S_0 \in S$

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面临文法符号为X时，应将状态转移到S_j

Z：终态集合　$Z = S-\{S_0\}$

即除S_0以外，其余全部是终态

构造DFA：

一、确定**S**集合，即**LR（0）项目集规范族**，同时确定$\mathbf{S_0}$

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已经归约的部分和等待归约部分。

构造LR(0)的方法(三步)

1) 将文法拓广

目的：使构造出来的分析表只有一个接受状态,这是为了实现的方便。

方法：修改文法，使识别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；
所有状态再组合成一个集合，即LR（0）项目集规范族

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.

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，先定义：

- 项目集闭包closure
- 状态转移函数GOTO

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(项目集合)

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)和GOTO的构造算法:

```
G'→LR(0), GOTO
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

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

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

$T \rightarrow T.*F$

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

I_4: $GOTO(I_0,() = closure(\{F \rightarrow (.E)\}) = I_4$

$F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .i$

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

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

$GOTO(I_0,*) = \varphi$

$GOTO(I_0,+) = \varphi$

$GOTO(I_0,)) = \varphi$

I_1: E'→E.
E→E.+T

I_2: E→T.
T→T.*F

I_3:T→F.

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_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,所有符号)为空

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

I_5: F→i.

I_8: F→(E.)
E→E.+T

GOTO(I_4,E) = closure({F→(E.),E→E.+T}) = I_8
GOTO(I_4,T) = I_2 ∈LR(0)
GOTO(I_4,F) = I_3 ∈LR(0)
GOTO(I_4,() = I_4 ∈LR(0)
GOTO(I_4,i) = I_5 ∈LR(0)
GOTO(I_4,+) =φ
GOTO(I_4,*) =φ
GOTO(I_4,)) =φ
GOTO(I_5,所有符号) =φ

I_6: E→E+.T T→.T*F
T→.F F→.(E)
F→.i

I_9: E→E+T.
T→T.*F

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

I_7: T→T*.F
F→.(E)
F→.i

I_{10}:T→T*F. GOTO(I_7,F) = closure({T→T*F .}) = I_{10}
GOTO(I_7,() = I_4
GOTO(I_7,i) = I_5

I_8: F→(E.)
E→E.+T

I_{11}:F→(E). GOTO(I_8,)) =closure({F→(E) .}) = I_{11}
GOTO(I_8,+) = I_6

I_9: E→E+T. T→T.*F

GOTO(I_9,*)= I_7

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

③ 构造DFA

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

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

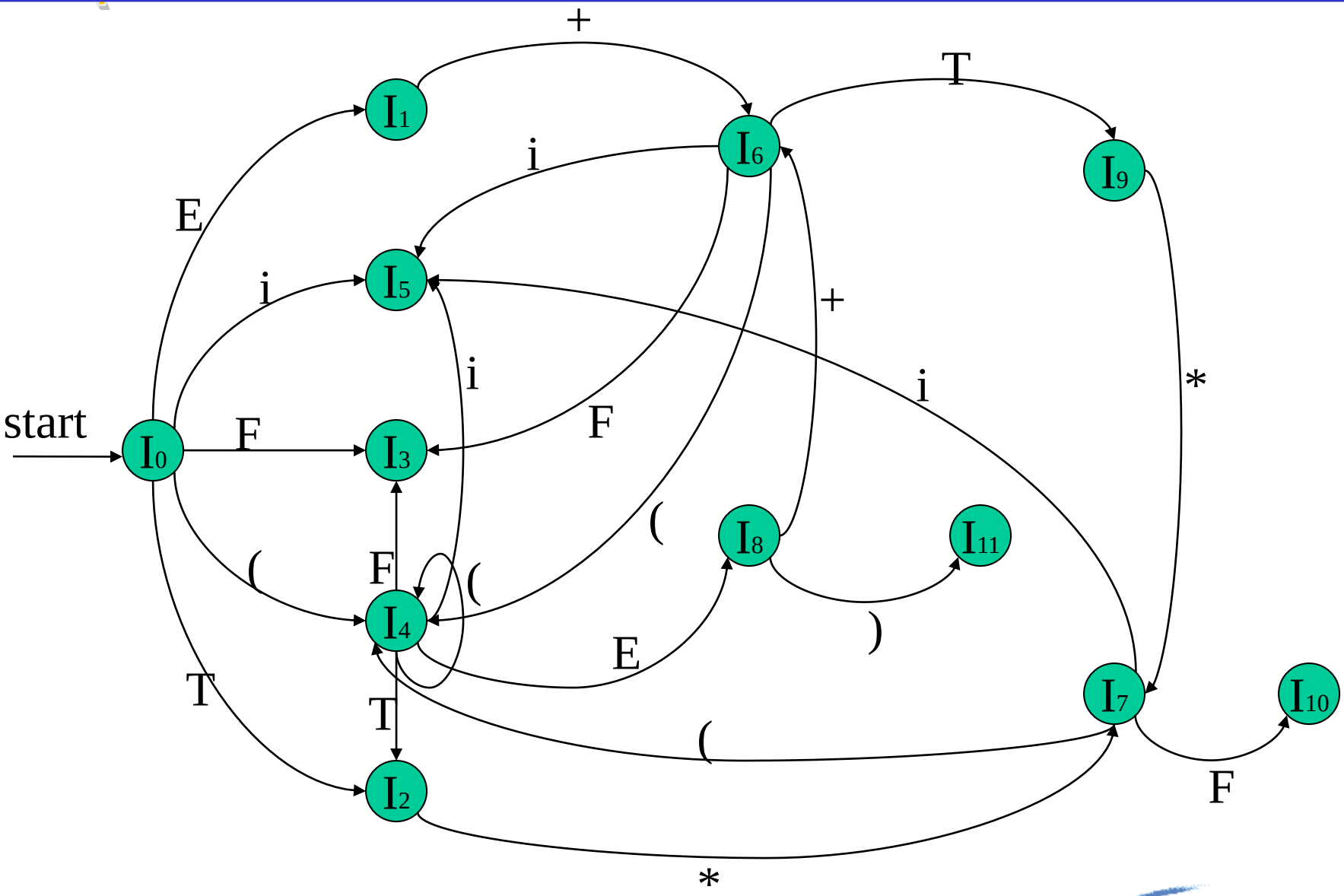
$V = \{+, *, i, (,), E, T, F\}$

$GOTO(I_m, X) = I_n$

$S_0 = I_0$

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

M的图解表示如下:


+
T
I1
I6
I9
i
E
i
I5
+
i
i
*
start
I0
F
I3
F
(
I8
I11
F
(
I4
(
E
)
T
T
I7
I10
(
I2
F
*

关于自动机的说明:

①除I_0以外,其余状态都是终态,从I_0到每一状态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 进行规范归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~I_1 识别规范句型的活前缀E(+i*i)

I_0~I_6 识别规范句型的活前缀E+(i*i)

I_0~I_7 识别规范句型的活前缀E+T*(i)

I_0~I_9 识别规范句型的活前缀E+T(*i)

I_0~I_{10} 识别规范句型的活前缀E+T*F

I0~I1	识别规范句型的活前缀E(+i*i)
I0~I6	识别规范句型的活前缀E+(i*i)
I0~I7	识别规范句型的活前缀E+T*(i)
I0~I9	识别规范句型的活前缀E+T(*i)
I0~I10	识别规范句型的活前缀E+T*F

步骤	状态栈	符号	输入串	动作
1	＃0	＃	i+i*i#	初始化，将栈顶置0状态
2	＃0i5	＃i	+i*i#	S
3	＃0F3	＃F	+i*i#	r6 F－> i
4	＃0T2	＃T	+i*i#	r4 T－>F
5	＃0E1	＃E	+i*i#	r2 E－>T
6	＃0E1+6	＃E+	i*i#	S
7	＃0E1+6i5	＃E+i	*i#	S
8	＃0E1+6F3	#E+F	*i#	r6 F－>i
9	＃0E1+6T9	#E+T	*i#	r4 T－>F
10	＃0E1+6T9*7	#E+T*	i#	S
11	＃0E1+6T9*7i5	#E+T*i	#	S
12	＃0E1+6T9*7F10	#E+T*F	#	r6 F－>i
13	＃0E1+6T9	#E+T	#	r3 T－>T*F
14	＃0E1	#E	#	r1 E－>E+T
15	＃0E1	#E		Accept

② 状态中每个项目对该状态能识别的活前缀都是有效的。

有效项目定义:若项目$A\rightarrow\beta_1.\beta_2$ 对活前缀 $\alpha\ \beta_1$ 有效，其条件是存在规范推导

$$E' \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$$

其中$\alpha,\beta1,\beta2 \in V^*,\ w \in Vt^*$

注意：项目中圆点前的符号串成为活前缀的后缀

③ 有效项目能预测分析的下一步动作：

$E\rightarrow E+T.$ 表示已将输入串归约为E+T,下一步应该将E+T归约为E

$$E' \overset{*}{\Rightarrow} (E) \Rightarrow (E+T)$$

$T\rightarrow T.*F$ 表示已将输入串归约为T,下一步动作是移进输入符号*

注意：经移进或归约后,在栈内仍是规范句型的活前缀

有效项目定义:若项目A→β1 . β2 对活前缀 α β 1有效，其条件是存在规范推导
$E' \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha \beta 1 \beta 2\, w$

例：

I_9 = { E—> E + T.　T—> T.*F} 能识别活前缀（E＋T

项目E—> E + T.　T—> T.*F对活前缀（E＋T有效

∵存在如下规范推导：

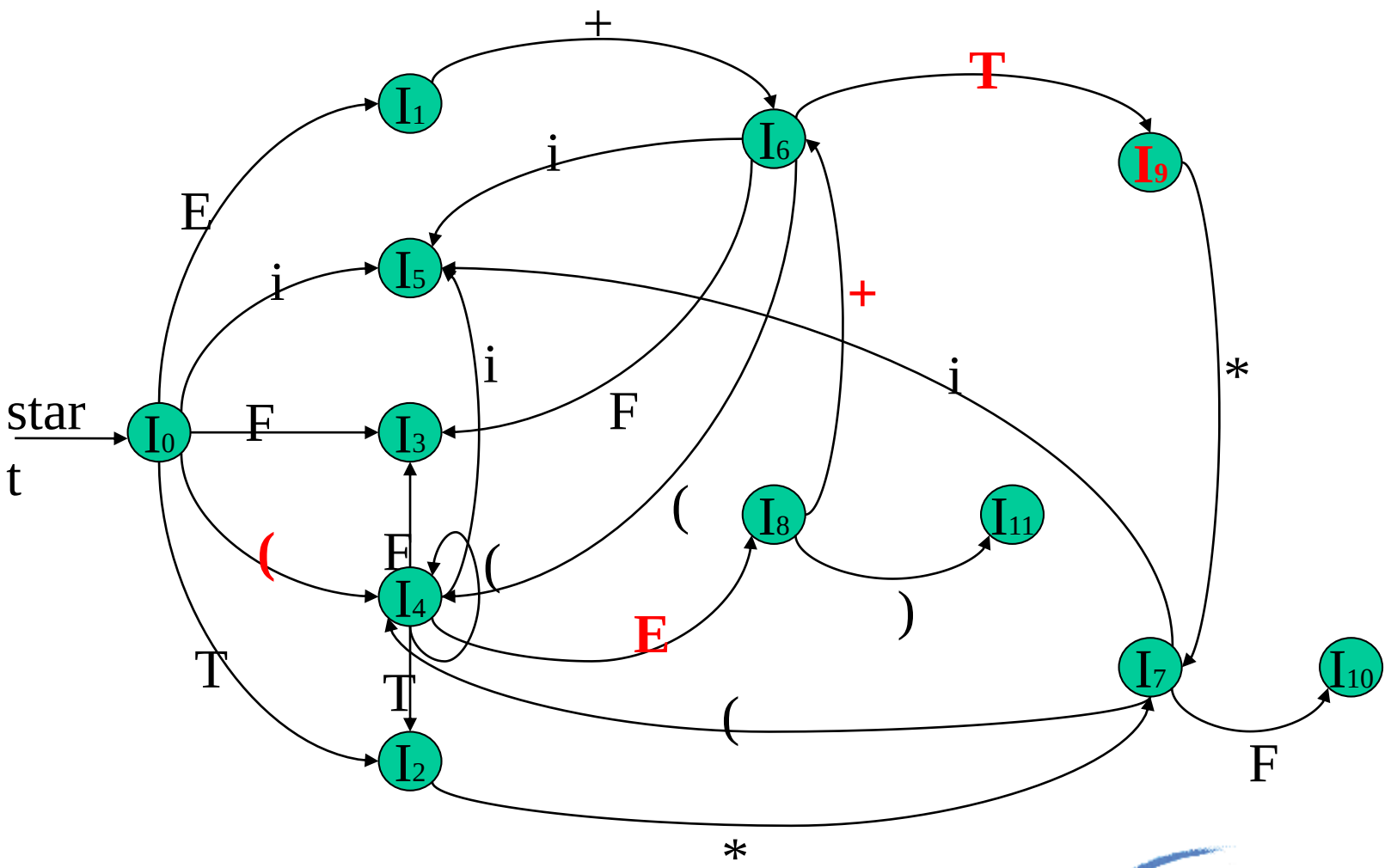
$E' \overset{*}{\Rightarrow} E \overset{*}{\Rightarrow} T \overset{*}{\Rightarrow} F \overset{*}{\Rightarrow} (\ E\ ) \overset{*}{\Rightarrow} (\ E+T\ )$

$\alpha Aw \quad \alpha \quad \beta_1 \beta_2 w$

$E' \overset{*}{\Rightarrow} E \overset{*}{\Rightarrow} T \overset{*}{\Rightarrow} F \overset{*}{\Rightarrow} (\ E\ ) \overset{*}{\Rightarrow} (\ E+T\ ) \overset{*}{\Rightarrow} (\ E + T\ * F\ )$

$\alpha \quad Aw \quad \alpha \quad \beta_1 \beta_2 w$

计算有效项目的方法：查看状态转移图



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的规范归约过程,故由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+现实⇒ 句柄

（2）由DFA构造SLR分析表

* GOTO表在求LR（0）时已求出

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})								

* 求ACTION表

设k为状态编号,E为原文法识别符号，
E’为扩充文法识别符号

1、求出文法每个非终结符的FOLLOW集合

2、若项目$A\rightarrow\alpha.a\beta \in k$,且$a \in V_t$,则置
ACTION[k,a] = s (移进）

3、若项目A→ α.∈k, 那么对输入符号a,若a∈FOLLOW(A),则置ACTION[k,a]=r_j
其中A→ α为文法G'的第j个产生式。

4、若项目E'→E.∈k, 则置
ACTION[k , #] =accept

5、ACTION表中不能用步骤2~4填入信息的空白格，均置error

在状态中可有三种类型的项目,其中只有两种有移进或归约动作:

$A\rightarrow \alpha.a\beta$	$a\in V_t$	移进项目	分析动作:移进
$A\rightarrow \alpha.$		归约项目	分析动作:归约
$A\rightarrow \alpha.B\beta$	$B\in V_n$	待约项目	无动作

根据上述算法,可以构造出文法G'[E']的ACTION

对文法G’[E’]

k=2 (I_2)
有效项目 E→T.
T→T.*F
FOLLOW(E)={#, +,)}

k=1 (I_1)
E’→E.
E→E.+T
FOLLOW(E’)={#}

k=0 (I_0)
E’→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.i

根据算法造出的ACTION表为:

I_1: E'→E.
E→E.+T

I_2: E→T.
T→T.*F

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

ACTION表

状态s \ 输入符号a	i	+	*	(	)	#
0	S_5			S_4		
1		S_6				accpet
2		r_2	S_7		r_2	r_2
3						
4						
5						
6						
7						
8						
9						
10						
11						

两点说明:

1. 由DFA构造出的SLR分析表,在造表时, 只需向前看一个符号就能确定分析的动作是移进还是归约，所以称为SLR(1)分析表，简称SLR分析表，使用SLR分析表的分析器叫SLR分析器。
2. 对文法G,若应用上述算法所造出的分析表具有多重定义入口,分析动作不唯一, 则文法G就不是SLR的，需要用别的方法来构造分析表。
   - 归约-归约冲突
   - 移进-归约冲突

SLR(1）判断方法

- 分析表无多重入口，就是SLR（1）文法

- 分析同一个项目集的产生式

- 归约-归约冲突，同时满足以下条件
  - 若两个不同的产生式有相同的右部或者某一产生式右部是另一产生式右部的后缀
    - U->X. 和 V-> X.
    - U->XY.和V->Y.
  - Follow(U) ∩ Follow(V) ≠ Φ

SLR（1）判断方法

- 分析同一个项目集的产生式

- 移进-归约冲突，同时满足以下条件
  - 某一产生式的右部是另一产生式的前缀
    - U->X.aY 和 V->X.
  - Follow(V) ∩ {a} ≠ Φ

- 如果不存在归约-归约冲突、移进-归约冲突，就符合SLR（1）文法

作业

- P282
- P288 题目修改（1）E↑ 修改为（1）F↑
- P297　1，2，6，9

复习第四章　语法分析

语法分析方法：
- 自顶向下分析法　$Z \overset{+}{\Rightarrow} S$
- 自底向上分析法　$S \overset{+}{\Leftarrow} Z$

$S \in L[Z]$

(一) 自顶向下分析

① 概述自顶向下分析的一般过程

存在问题：
- 左递归问题 ——— 消除左递归的方法
- 回溯问题 ———
  - 无回溯的条件
  - 改写文法
  - 超前扫描

② 两种常用方法:

(1) 递归子程序法
- a) 改写文法,消除左递归,回溯
- b) 写递归子程序

(2) LL(1)分析法
- LL(1)分析器的逻辑结构及工作过程
- LL(1)分析表的构造方法
  - 1.构造FIRST集合的算法
  - 2.构造FOLLOW集合的算法
  - 3.构造分析表的算法
- LL(1)文法的定义以及充分必要条件

(二) 自底向上分析

归约过程:

(1)一般过程:　　移进-归约过程

问题:如何寻找句柄

(2)算法:

i)算符优先分析法:

1.分析器的构造,分析过程

根据算符优先关系矩阵来决定是移进还是归约。

2.算符优先法的进一步讨论

1) 适用的文法类----------引出的算符优先法的定义
2) 优先关系矩阵的构造
3) 什么是“句柄”,如何找
   由句柄引出的最左素短语的概念。
   最左素短语的定理,如何找。

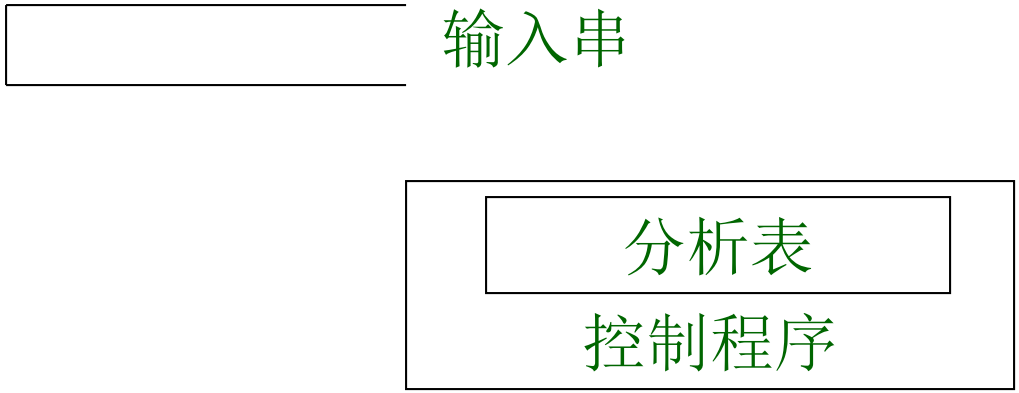
ii)LR分析法

1.概述----概念、术语　(活前缀、项目)

- 2.LR分析
  - 1)逻辑结构
    - 状态栈
    - 分析表
      - GOTO表
      - 分析动作表
    - 控制程序
  - 2)分析过程

- 3.如何构造SLR分析表
  - 1)构造DFA
  - 2)由DFA构造分析表

除了递归子程序法，其他几种方法逻辑结构很象：

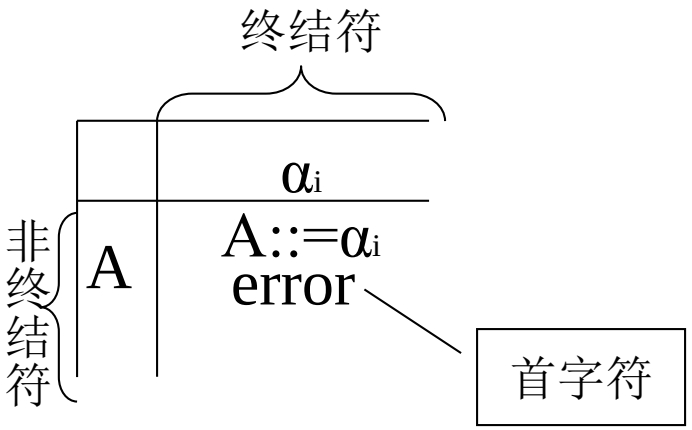


(1) 对于LL(1)分析法

符号栈

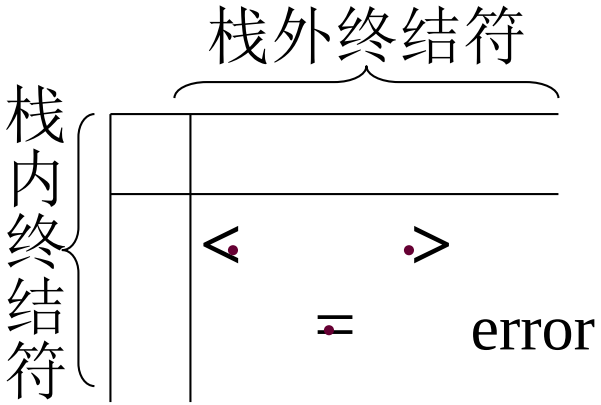
α_i

LL(1)分析表



(自顶向下，保证最左推导)

(2)对于算法优先分析:　符号栈



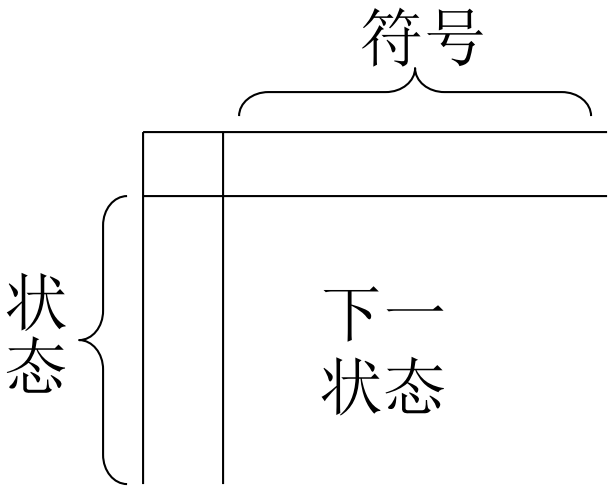
(3)LR分析:

符号栈

$S_0,S_1 \ldots S_m$
$X_1 \ldots X_m$

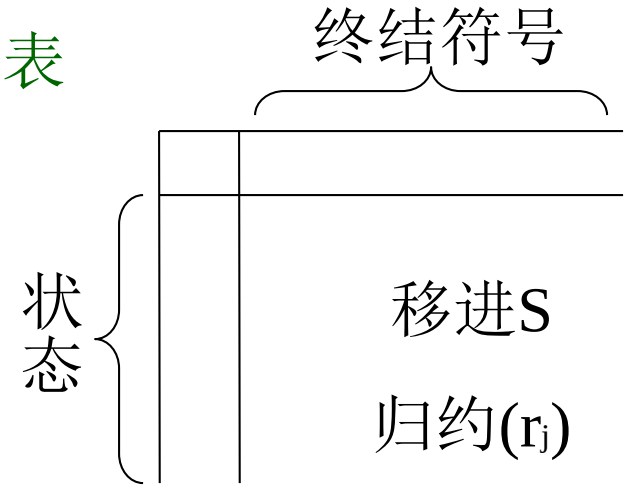
分析表 { 状态转移GOTO表；分析动作表

GOTO表



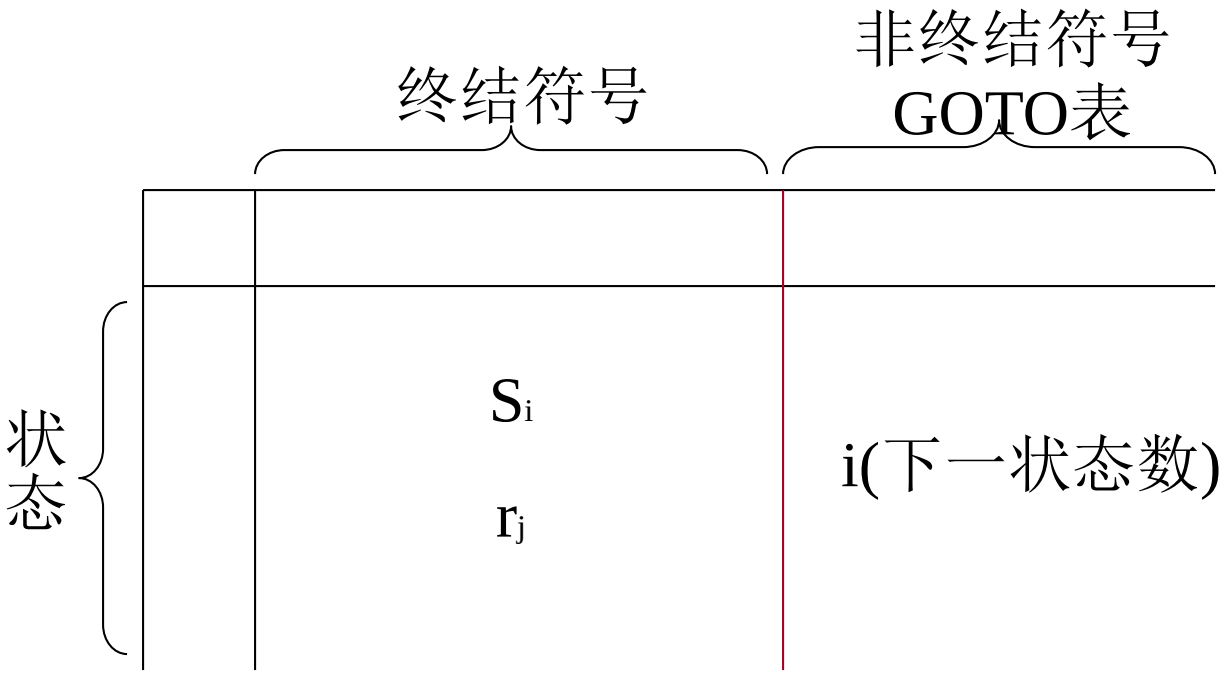
根据栈顶状态和栈顶符号推导出下一状态

分析动作表

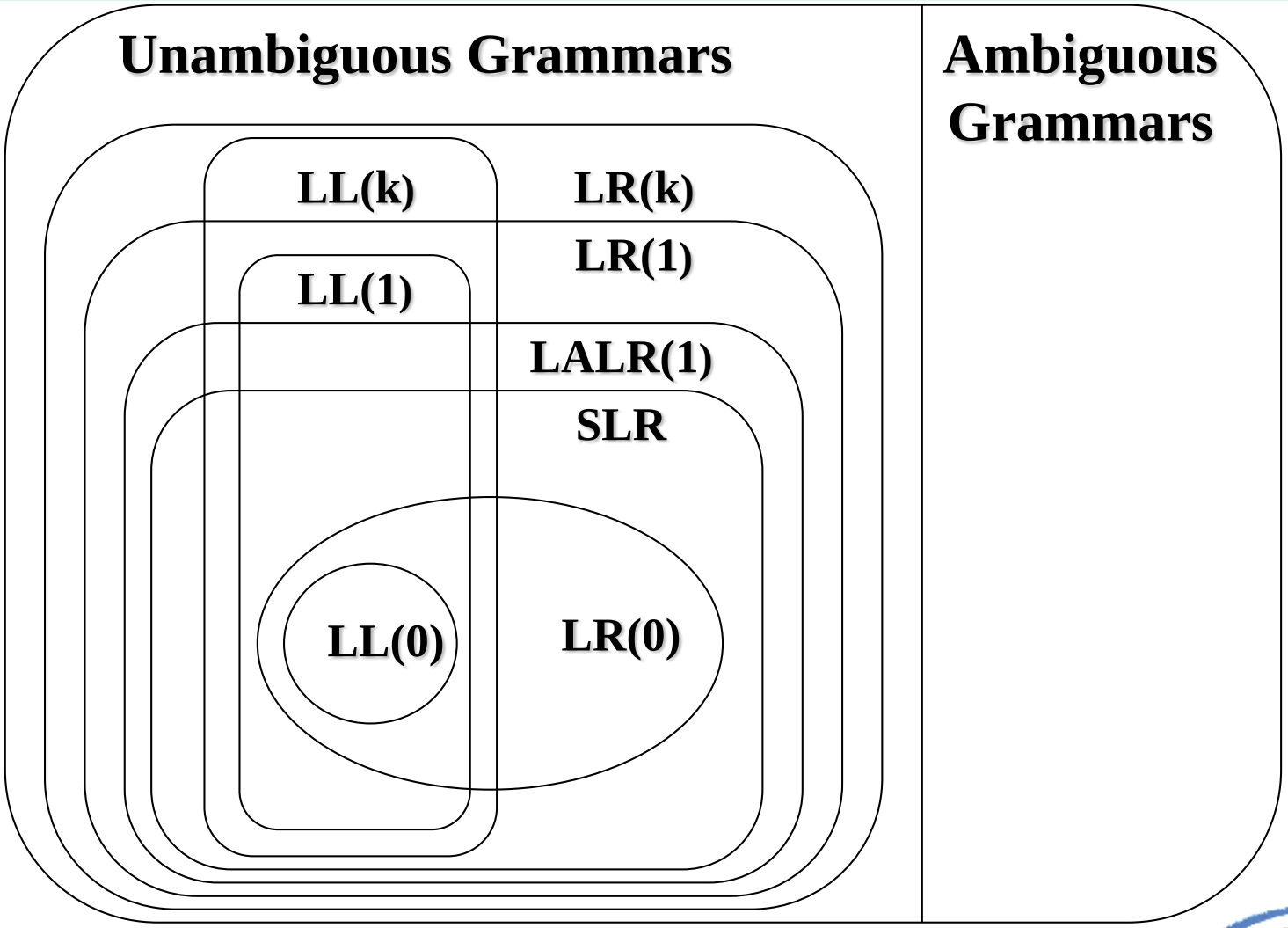


根据栈顶状态和输入符号推导出下一动作

将GOTO表和分析动作表压缩后得:



不同文法类的层次结构


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LR(1)
LL(1)
LALR(1)
SLR
LL(0)
LR(0)