第4章　自顶向下语法分析方法

- 语法分析是编译程序的核心部分。
- 语法分析的作用是识别由词法分析给出的单词符号序列是否是给定文法的正确句子(程序)
- 常用的语法分析方法有自顶向下分析和自底向上分析

- 自顶向下分析法也称面向目标的分析方法，也就是从文法的开始符号出发企图推导出与输入的单词串完全相匹配的句子。
  - 确定的自顶向下分析方法：对文法有一定的限制，但实现方法简单、直观，便于手工构造或自动生成语法分析器，目前仍是常用的方法之一。
  - 不确定的自顶向下分析方法：带回溯的分析方法，是一种穷举的试探方法，效率低，代价高，极少使用

主要内容

4.1 确定的自顶向下分析思想

确定的自顶向下分析方法，是从文法的开始符号出发，考虑如何根据当前的输入符号(单词符号)唯一地确定选用哪个产生式替换相应非终结符以向下推导，或如何构造一课相应的语法树。

例4.1

文法G1[S]：

S → pA | qB

A → cAd | a

B → dB | b

口入串：

W=pccadd

存在推导过程：

S

⇒pA

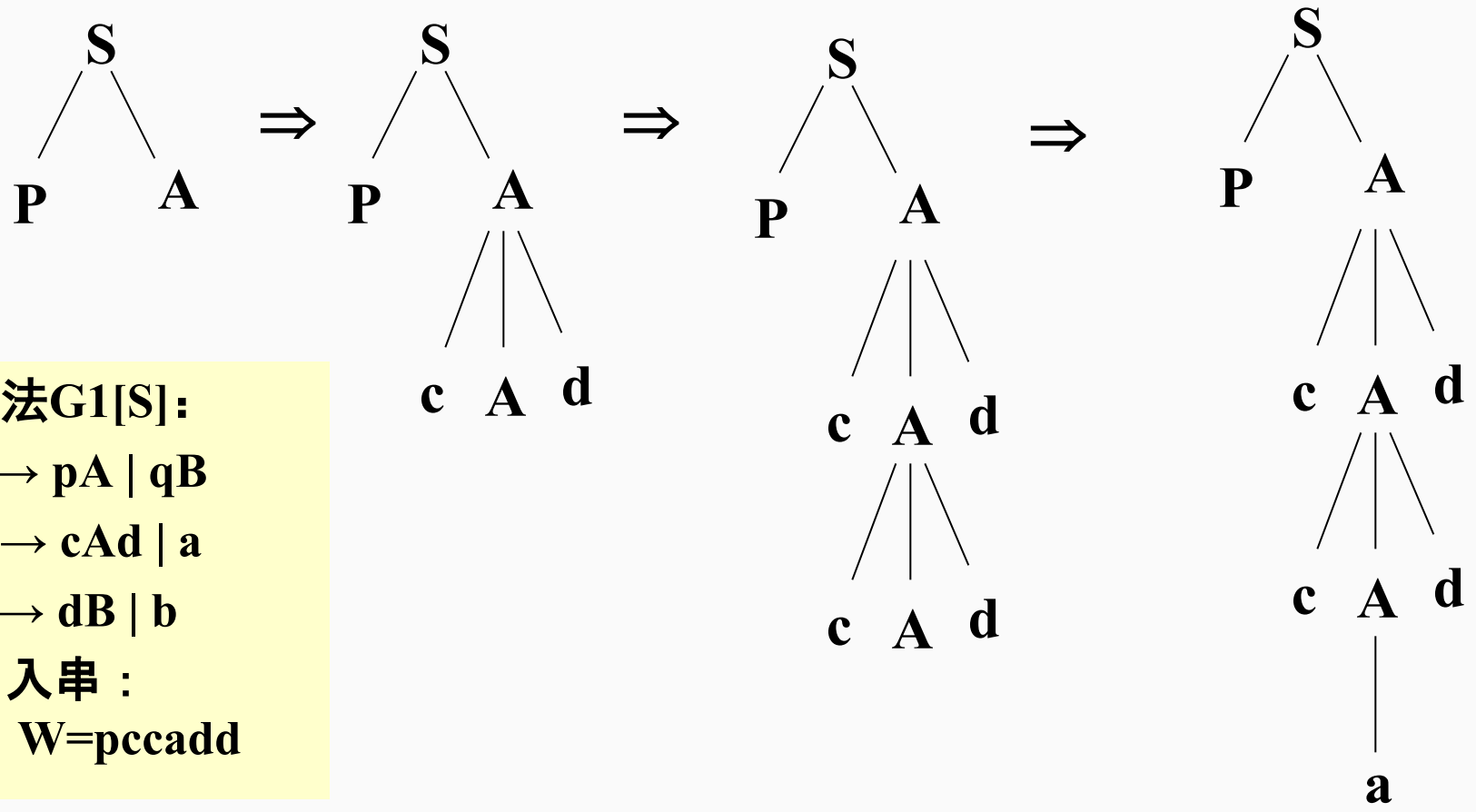
⇒pcAd

⇒pccAdd

⇒pccadd

W是文法合法的句子

相应的语法树：



文法G1[S]：

S → pA | qB

A → cAd | a

B → dB | b

□入串：

W=pccadd

对于这样的文法显然在推导过程中完全可以根据当前的输入符号决定选择哪个产生式往下推导，因此分析过程是唯一的。

这个文法有以下特点：

(1)每个产生式的右边都由终结符号开始；

(2)如果两个产生式有相同的左部，那么它们的右部由不同的终结符号开始。

文法G1[S]：

S → pA | qB

A → cAd | a

B → dB | b

□入串：

W=pccadd

例4.2

文法G2[S]：

S → Ap | Bq

A → cA | a

B → dB | b

对于输入串：

W = ccap

存在推导过程：

S

⇒Ap

⇒cAp

⇒ccAp

⇒ccap

W是文法合法的句子

语法树为：

文法G2[S]：

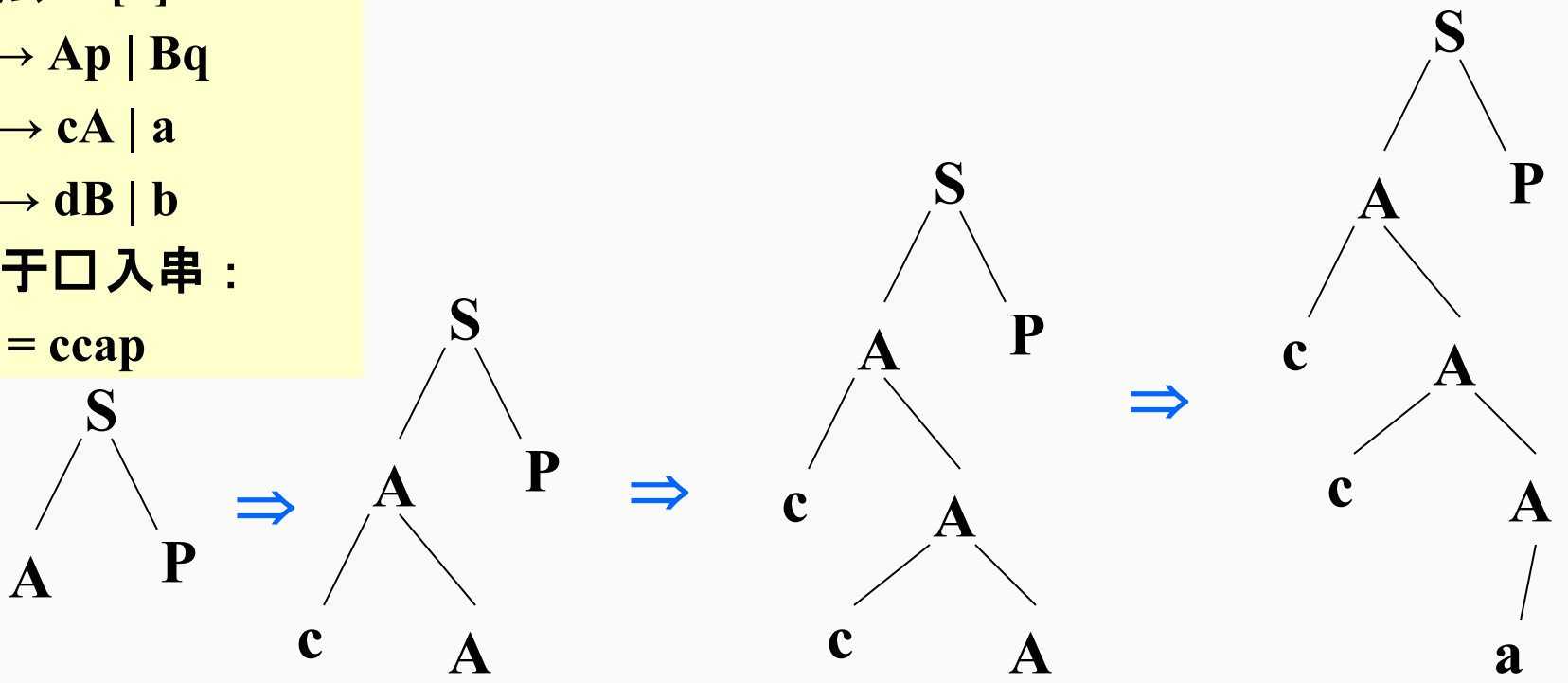
S → Ap | Bq

A → cA | a

B → dB | b

□于□入串：

W = ccap



这个文法的特点是：

(1)产生式的右部不全是由终结符开始

(2)如果两个产生式有相同的左部，它们的右部由不同的终结符或非终结符开始。

(3)文法中无空产生式

对于相同左部的产生式含有非终结符开始的产生式时，在推导过程中选用哪个产生式不太直观。

文法G2[S]：

S → Ap | Bq

A → cA | a

B → dB | b

□于□入串：

W = ccap

定义4.1

设G=(V_N,V_T,P，S)是上下文无关文法

FIRST（α）={a|$\alpha \overset{*}{\Rightarrow} a\beta$, a∈$V_T$, α, β∈V*}

若$\alpha \overset{*}{\Rightarrow} \varepsilon$则规定ε∈FIRST（α）

FIRST（α）为α的开始符号集或首符号集

例：文法G2中

FIRST（Ap）= {a, c}

FIRST（Bq）= {b, d}

文法G2[S]：

S → Ap | Bq

A → cA | a

B → dB | b

口于口入串：

W = ccap

考虑文法G2，关于S的两个产生式的右部虽然都以非终结符开始，但它们右部的符号串可以推导出的首符号集合不相交，因而可以根据当前的输入符号是属于哪个产生式的首符号集合而决定选择相应产生式进行推导。

这样仍能构造确定的自顶向下分析。

例4.3

文法G3[S]：

S → aA | d

A → bAS | ε

口于口入串：

W = abd

存在推导过程：

S

⇒aA

⇒abAS

⇒abS

⇒abd

W是文法合法的句子

语法树为：

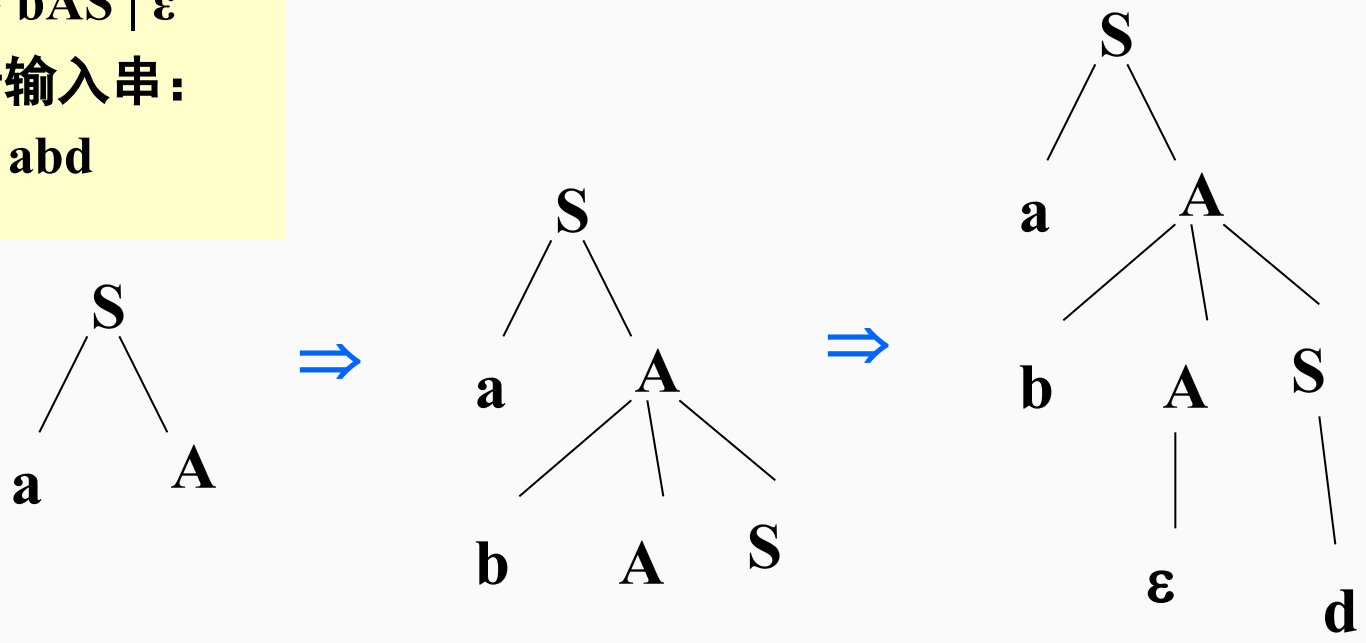
文法G3[S]：

S → aA | d

A → bAS | ε

对于输入串：

W = abd



可以看出，当某一非终结符的产生式中含有空产生式时，它的非空产生式中右部的首符号集两两不相交，并与在推导过程中紧跟该非终结符右边可能出现的终结符集也不相交，仍可构造确定的自顶向下分析。

定义4.2

设$G=(V_N,V_T,P，S)$是上下文无关文法，$A \in V_N$，S是开始符号

FOLLOW（A）$=\{a|S \overset{*}{\Rightarrow} \mu A \beta$且$a \in$ FIRST（β），

$\mu \in V^*$， $\beta \in V^+ \}$

若$S \overset{*}{\Rightarrow} \mu A \beta$,且$\beta \overset{*}{\Rightarrow} \varepsilon$，则$\# \in$FOLLOW(A)

也可以定义为：

FOLLOW（A）$=\{a|S \overset{*}{\Rightarrow} ..Aa...$且$a \in V_T \}$

若有$S \overset{*}{\Rightarrow} ..A \beta$,且$\beta \overset{*}{\Rightarrow} \varepsilon$，则$\# \in$FOLLOW(A)

‘#’为输入串的结束符，或称输入串括号

当文法中含有形如A→α， A→β的产生式时，其中A∈V_N, α, β∈V*。若α和β不能同时推导出空，假定α $\overset{*}{\Rightarrow}$ ε, β $\overset{*}{\Rightarrow}$ ε，则当FIRST(α)∩(FIRST(β)∪FOLLOW(A))=Φ□，□于非□□符A的替□仍可唯一地确定候□。

即：

□入符号∈ FIRST(α)：选用产生式A→α

□入符号∈ FIRST(β)∪FOLLOW(A))：选用产生式A→β

定义4.3

给定上下文无关文法的产生式$A\to\alpha$，$A\in V_N$, $\alpha\in V^*$。

若$\alpha\overset{*}{\not\Rightarrow}\varepsilon$，则SELECT(A→α)=FIRST(α)

若$\alpha\overset{*}{\Rightarrow}\varepsilon$，则

SELECT(A→α)=(FIRST(α)-{ε})∪FOLLOW(A)

定义4.4

一个上下文无关文法是LL(1)文法的充分必要条件是，对每个非终结符A的两个不同产生式$A\rightarrow\alpha$，$A\rightarrow\beta$，□足。

$$SELECT(A\rightarrow\alpha)\cap SELECT(A\rightarrow\beta)=\Phi$$

其中α，β不能同时$\overset{*}{\Rightarrow}\varepsilon$。

LL(1)文法能够采用确定的自顶向下分析技术

LL(1)文法的含义

第1个L表明自顶向下分析是从左向右扫描输入串

第2个L表明分析过程中将用最左推导

1表明只需要向右看一个符号便可决定如何推导，即选用哪个产生式

例4.3

文法G3[S]：S → aA | d　A → bAS | ε

SELECT(S → aA)={a}　SELECT(S → d)={d}

SELECT(A → bAS)={b}

SELECT(A → ε)={a, d, #}

而

SELECT(S → aA)∩SELECT(S → d)={a}∩{d}=Φ

SELECT(A → bAS)∩SELECT(A → ε)

={b}∩{a, d, #}=Φ

该文法是LL(1)文法，可用确定的自顶向下分析

例4.4

文法G4[S]：S → aAS | b　A → bA | ε

SELECT(S → aAS)={a}　SELECT(S → b)={b}

SELECT(A → bA)={b}

SELECT(A → ε)={a, b}

而

SELECT(S → aAS)∩SELECT(S→b)={a}∩{b}=Φ

SELECT(A → bA)∩SELECT(A →ε)

={b}∩{a, b}≠Φ

该文法不是LL(1)文法

4.2 LL(1)文法的判别

当我们选用自顶向下分析技术时，首先必须判别所给文法是否是LL(1)文法，因而对任给文法需计算FIRST，FOLLOW，SELECT集合，进而判别文法是否为LL(1)文法

例4.5

文法G[S]：

S→AB | bC

A →ε| b

B →ε| aD

C → AD | b

D → aS | c

1) 求出能推出ε的非终结符

- 如果所有以某个非终结符为左部的产生式的右部都含有终结符号，则该非终结符不能推出ε
- 如果以某一非终结符为左部的某一产生式的右部为ε，则该非终结符能推出ε
- 其他非终结符号是否能够推出ε可以推导得出
- 文法G的所有非终结符中：
  - 可以推出ε的非终结符：S, A, B
  - 不能推出ε的非终结符：C, D

文法G[S]：

S→AB | bC

A →ε| b

B →ε| aD

C → AD | b

D → aS | c

2)计算FIRST集

方法一：根据定义计算

对于每一个文法符号$x \in V$，计算first(x)的方法如下：

a) 若$x \in V_T$，则first(x)＝{x}

b) 若$x \in V_N$，且有$x \to a\ldots$，$a \in V_T$，则$a \in$first(x)

c) 若$x \in V_N$，$x \to \varepsilon$，□$\varepsilon \in$first(x)

d) 若$x \in V_N$，y_1，y_2，…，y_i都$\in V_N$，若有产生式
$x \to y_1 y_2 \ldots y_n$，当y_1，y_2，…y_{i-1}都$\overset{*}{\Rightarrow} \varepsilon$□$(1 \le i \le n)$，
则first(y_1)－{ε}，first(y_2)－{ε}，…，first(y_{i-1})－{ε}，
first(y_i)都包含在first(x)中。

e) 当上式中所有$y_i \overset{*}{\Rightarrow} \varepsilon$ $(1 \le i \le n)$，

则first(x)＝(first(y_1)- {ε}) ∪(first(y_2) - {ε}) ∪…∪ (
first(y_n) - {ε}) ∪ {ε}

按上面的规则可得上例文法中每个非终结符号的first集合如下：

$S \rightarrow AB \mid bC$

$A \rightarrow \varepsilon \mid b$

$B \rightarrow \varepsilon \mid aD$

$C \rightarrow AD \mid b$

$D \rightarrow aS \mid c$

$first(S)=(first(A)-\{\varepsilon\}) \cup (first(B)-\{\varepsilon\}) \cup \{\varepsilon\} \cup \{b\}=\{a,b,\varepsilon\}$

$first(A)=\{b\} \cup \{\varepsilon\}=\{b,\varepsilon\}$

$first(B)=\{\varepsilon\} \cup \{a\}=\{a,\varepsilon\}$

$first(C)=(first(A)-\{\varepsilon\}) \cup first(D) \cup first(b)=\{a,b,c\}$

$first(D)=\{a\} \cup \{c\}=\{a,c\}$

一个文法符号串的first集合计算方法：

如果文法符号串$\alpha \in V^*$, $\alpha=X_1X_2\ldots X_n$,

1、当$X_1 \not\Rightarrow \varepsilon$，则$first(\alpha)=first(X_1)$

2、当对任何j（$1\le j\le i-1$，$2 \le i \le n$），$\varepsilon \in first(X_j)$

$\varepsilon \in first(X_i)$，则$first(\alpha)=(first(X_1)-\{\varepsilon\}) \cup (first(X_2)-\{\varepsilon\}) \cup \ldots \cup (first(X_{i-1})-\{\varepsilon\}) \cup first(X_i)$

3、当$first(X_j)$都含有ε时$(1 \le j \le n)$，则
$first(\alpha)=first(X_1) \cup first(X_2) \cup \ldots \cup first(X_j) \cup \{\varepsilon\}$

根据上面规则，每个产生式的右部符号串开始符号集合为：

first(AB)=first(A) ∪first(B) ∪{ε}={a, b, ε}

first(bC)= {b}

first(ε)= { ε}

first(aD)= {a}

first(AD)=(first(A)-{ε}) ∪first(D)={a,b,c}

first(b)= {b}

first(aS)= {a}

first(c)= {c}

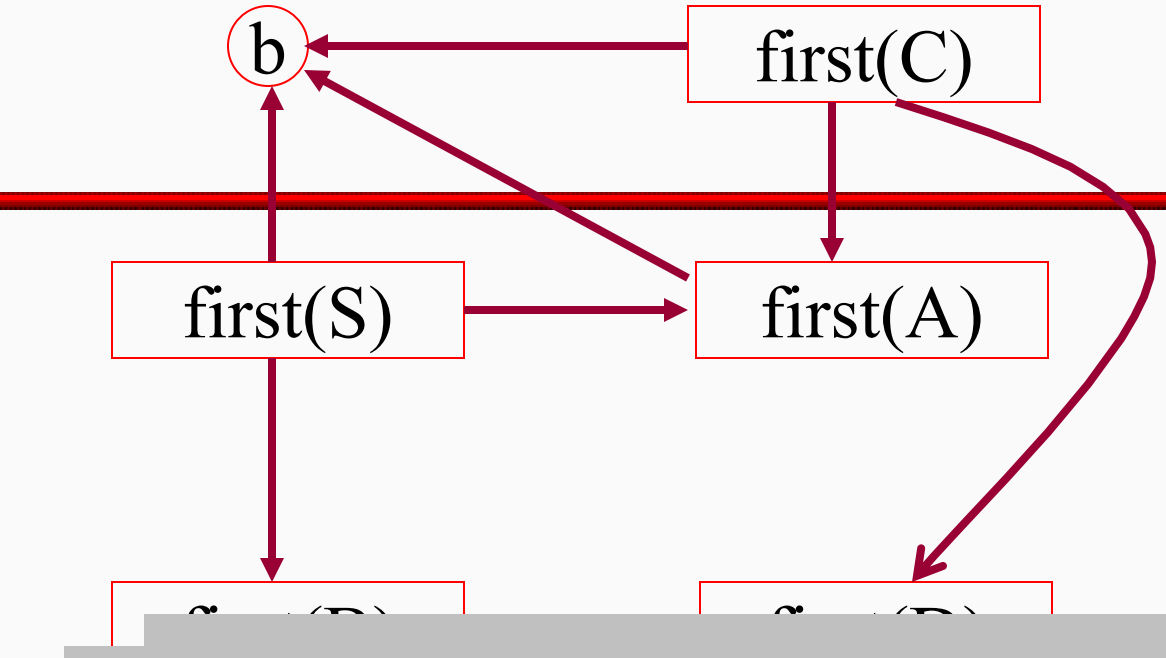
方法二：由关系图法求文法符号的first集合

1、每个文法符号对应图中的一个结点，终结符结点用符号本身标记，非终结符结点用first(A)标记。

2、若文法中有$A\rightarrow\alpha X\beta$，$\alpha\overset{*}{\Rightarrow}\varepsilon$，则从对应A的结点到对应X结点连一条箭弧。

3、凡是从first(A)结点有路径可到达的终结符结点所标记的终结符都为first(A)的成员。

4、根据判别步骤1)确定ε是否为某非终结符first的成员，若是则将ε加入该非终结符的first集中。



文法为：

S →AB | bC

A →ε | b

B →ε | aD

C →AD | b

D →aS | c

S 同理：从A到b，从B到a， =C
从C到A、D、b，
从D到a、c画一条弧。

(2) 规则1：非终 (3)规则2：A→αXβ， α⇒ε，
标记。本文法中 则A到X画一条箭弧。

(4)规则3： first(A)结点有路径可到达的终结符结点所标记的终结符都为first(A)的成员。

(5)规则4： 若ε是某非终结符first的成员，则将ε加入该非终结符的first集中。

所以，first(S)＝｛a,b｝
first(A)＝｛b｝
first(B)＝｛a｝
first(C) ＝｛a,b,c｝
first(D)＝｛a,c｝

所以，first(S)＝｛a,b, ε｝
first(A)＝｛b, ε ｝
first(B)＝｛a, ε ｝
first(C) ＝｛a,b,c｝
first(D)＝｛a,c｝

（3）计算Follow 集：

方法一：根据Follow定义计算：

follow集的定义：

FOLLOW（A）={a|S $\overset{*}{\Rightarrow}$…Aa…且a $\in V_T$ }
若有S$\overset{*}{\Rightarrow}$…A β ,且β $\overset{*}{\Rightarrow}$ε，则#∈FOLLOW(A)

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为开始符号,则把“#”加入FOLLOW(S)中

(2)若A $\rightarrow$ αBβ (β≠ε),则把FIRST(β)-{ε}加入FOLLOW(B)

(3)若A $\rightarrow$ αB 或A $\rightarrow$ αBβ， 且β$\stackrel{*}{\Rightarrow}$ε则把FOLLOW(A)加入FOLLOW(B)

Follow(S)={#}

Follow(A)={a，#，c}

Follow(B)={#}

Follow(C)={#}

Follow(D)={#}

文法为：

S → AB | bC

A →ε | b

B →ε | aD

C →AD | b

D →aS | c

方法二：由关系图法求非终结符号的follow集

1、每个文法符号和“#”对应图中的一个结点，终结符结点和“#”用符号本身标记，非终结符结点($A \in V_N$)用Follow(A)或First(A)标记。

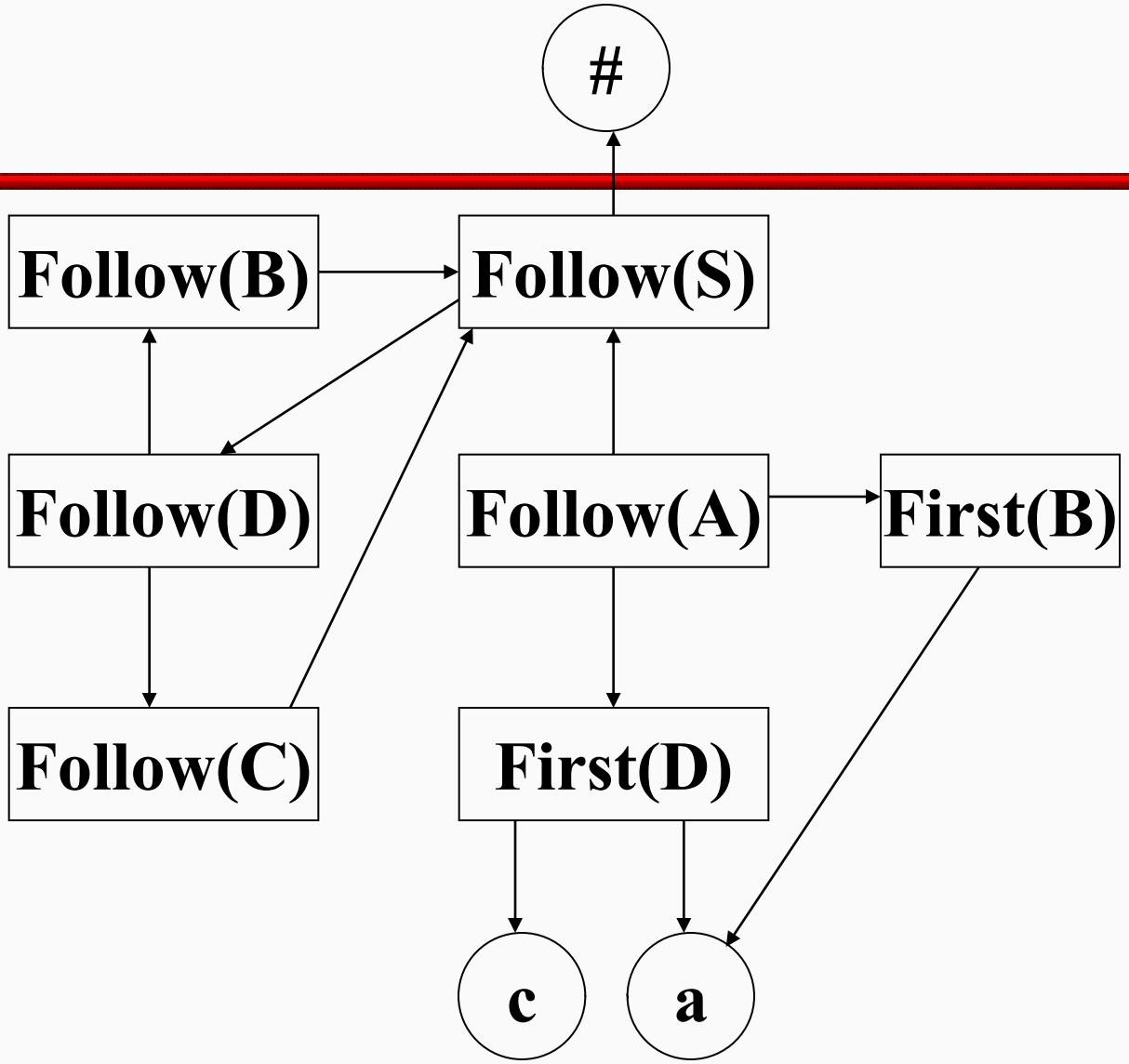
2、从开始符号S的Follow(S)结点到“#”号的结点连一条箭弧。

3、若文法中有$A \rightarrow \alpha B \beta X$，且$\beta \overset{*}{\Rightarrow} \varepsilon$，则从Follw(B)结点到FIRST(X)结点连一条弧，当$X \in V_T$时，则与X相连。

4、若文法中有A→αBβ，且β $\overset{*}{\Rightarrow}$ε，则从Follw(B)结点到 Follow(A)结点连一条箭弧。

5、对每一first(A)结点，如果有产生式A→αXβ，且α $\overset{*}{\Rightarrow}$ε，则从First(A)结点到 First(X)结点连一条箭弧。

6、凡是从Follow(A)结点有路径可以到达的终结符或“#”号的结点，其所标记的终结符或“#”即为Follow(A)的成员。



文法为：

S → AB | bC

A →ε | b

B →ε | aD

C →AD | b

D →aS | c

(4) 计算Select集：

选择集合的定义

给定上下文无关文法的产生式$A\to\alpha$，$A\in V_N$，

$\alpha\in V^*$，若$\alpha\overset{*}{\Rightarrow}\varepsilon$，则Select（$A\to\alpha$）=First($\alpha$)，

若$\alpha\overset{*}{\Rightarrow}\varepsilon$，则Select($A\to\alpha$)=（First($\alpha$)-{$\varepsilon$}）∪Follow($A$)

(4) 计算Select集：

每个产生式的Select集合计算为：

因为A⇒ε
B ⇒ε

Select(S→AB)= (first (AB)-{ε})∪Follow(S)={b,a,#}

Select(S→ bC)= first (bC)={b}

Select(A→ε)= (first (ε)-{ε})∪Follow (A)={c,a,#}

Select(A→b)= first (b) ={b }

Select(B→ε)= (first (ε) -{ε}) ∪Follow (B)={ #}

Select(B→aD)= first (aD) ={a}

Select(C→AD)= first (AD) ={b,a,c}

Select(C→b)= first (b) ={b }

Select(D→aS)= first (aS) ={ a }

Select(D→c)= first (c) ={c}

所以select的交集为：

Select(S→AB) ∩Select (S→bC)= {b} ≠ ϕ

Select(A→ε) ∩ Select (A→b)= ϕ

Select(B→ε) ∩ Select (B→aD)=ϕ

Select(C→AD) ∩ Select (C→b)= {b} ≠ϕ

Select(D→aS) ∩ Select (D→c)= ϕ

因此该文法不是LL(1)文法。

4.3 某些非LL(1)文法到LL(1)文法的等价变换

- 判定任何一个给定文法是不是LL(1)文法，一般需要通过判定方法进行判别。
- 若文法中含有直接或间接左递归，含有左公共因子，该文法肯定不是LL(1)文法，要进行变换。

1、提取左公共因子

若文法中含有形如$A\to\alpha\beta|\alpha\gamma$的□生式，□□致了对相同左部的产生式其右部的First集相交,也就是$Select(A\to\alpha\beta)\cap Select(A\to\alpha\gamma)\neq\phi$，不□足LL(1)文法的充分必要条件。

则须进行提取左公共因子的等价变换：$A\to\alpha(\beta|\gamma)$

写成一般形式：$A\to\alpha A'$

$A'\to\beta|\gamma$

例1：若文法G1的产生式为：

S→aSb

S→aS

S→ε

提取左公因子后得：S→aS(b |ε)

S→ε

进一步变换：S→aSA

A→b

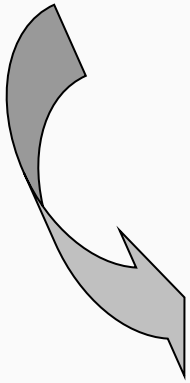
A→ε

S→ε

例2：若文法G2的产生式：　A→ad
A→Bc
B→aA
B→bB

解：　A → ad
A → aAc
A → bBc
B → aA
B → bB

A→a(d|Ac)



A→aA’
A’ →d
A’ →Ac

最后变换为：

$A \rightarrow aA'$

$A \rightarrow bBc$

$A' \rightarrow d$

$A' \rightarrow Ac$

$B \rightarrow aA$

$B \rightarrow bB$

例1经提取左公共因子后文法仍不是LL(1)文法，例2提取左公共因子后文法是LL(1)文法 。

经过文法提取左公共因子后的文法不一定是LL(1)文法。

经过文法提取左公共因子后的文法，若有多余的产生式，则必须进行化简 。

例3：若有文法G3的产生式：$S \rightarrow aSd$

$S \rightarrow Ac$

$A \rightarrow aS$

$A \rightarrow b$

解：文法替换：S→aSd
S → aSc
} S→aS(d|c)
S→bc
A → aS
A → b

S→aSA′

A′ →d|c

S→bc

A→aS

A→b

文法中非终结符A变成不可到达的符号,产生式也就变为多余的，所以应删除.

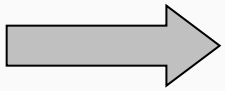
此外也存在某些文法不能在有限步骤内提取左公共因子。

例4: 若有文法**G4**的产生式:

S →Ap|Bq

A →aAp|d

B →aBq|e

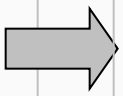


S→aApp|aBqq

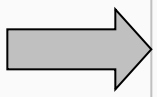
S→dp|eq

A→aAp|d

B →aBq|e



S→a(App|Bqq)



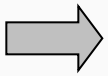
S→aS′

S′→App|Bqq

S→dp|eq

A→aAp|d

B →aBq|e



S→aS′

S→dp|eq

S′→aAppp|aBqqq

S'→dpp|eqq

A→aAp|d

B →aBq|e

S→aS′

S→dp|eq

S′→aS″

S′→dpp|eqq

S″→Appp|Bqqq

A→aAp|d

B →aBq|e

可以看出产生式A，B继续替换，只能使文法的产生式愈来愈多无限增加下去，变成循环递归，不能得到提取左公共因子的预期结果。

上面例子说明：

不一定每个文法的左公共因子都能在有限的步骤内替换成无左公共因子的文法。

一个文法提取了左公共因子后，只解决了相同左部产生式右部的FIRST集不相交问题。当改写后的文法不含空产生式，且无左递归时，则改写后的文法是LL(1)文法。否则还需用LL(1)文法的判别方式进行判断才能确定是否为LL(1)文法。

2、消除左递归

一个文法含有下列形式的产生式时，

a) $A \rightarrow A\beta$　　$A \in V_N, \beta \in V^*$

b) $A \rightarrow B\beta$

$B \rightarrow A\alpha$　　$A,B \in V_N, \alpha, \beta \in V^*$

称为左递归文法。其中a)是直接左递归，b)是间接左递归

如果一个 文法是左递归时，则不能采用自顶向下分析法。

左递归举例

例1：文法S →Sa

S →b

输入串：baaa#

是直接左递归

所产生的语言是：$L=\{ ba^n \mid n\geq 0\}$

例2： 文法为：A→aB

A→Bb

B→Ac

B→d

输入串：adbcbcbc#

是间接左递归

※ 消除左递归的变换方法：

(1) 消除直接左递归：

方法：改为右递归。如

$$\begin{cases} S \to Sa \\ S \to b \end{cases}$$ 改写为：$$\begin{cases} S \to b\ S' \\ S' \to a\ S' \mid \varepsilon \end{cases}$$

消除左递归的一般情况

一般情况下，假定关于A的全部产生式是：

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$$

其中$\alpha_i(1 \le i \le m)$不等于ε，$\beta_j(1 \le j \le n)$不以A开口，
消除直接左口口之后改写口：

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$$

(2) 消除间接左递归：

方法：间接左递归⇒直接左□□⇒右□□

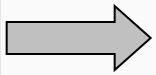
例3：文法G6为：

A→aB

A→Bb

B→Ac

B→d



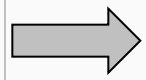
A→aB

A→Bb

B→aBc

B→Bbc

B→d

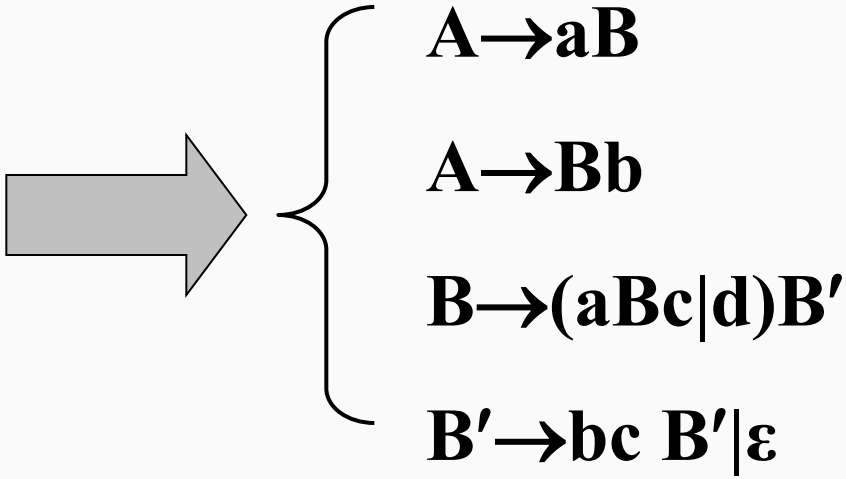


B→aBc|d

B→Bbc

B→(aBc|d)B'

B'→bcB'|ε


A→aB
A→Bb
B→(aBc|d)B′
B′→bc B′|ε

（3）消除文法中一切左递归的算法。

对文法中一切左递归的消除要求文法中不含回路，即无$A \overset{+}{\Rightarrow} A$的推导。

满足该文法的充分条件是：文法中不包含形如$A \rightarrow A$的有害规则和$A \rightarrow \varepsilon$的产生式。

算法步骤

（1）以某种顺序将文法非终结符排列$A_1, A_2 \ldots A_n$

（2）从A_1开始消除左部为A_1的产生式的直接左递归，然后把左部为A_1的所有规则的右部逐个替换左部为A_2右部以A_1开始的产生式中的A_1，并消除左部为A_2的产生式中的直接左递归。

（3）继而以同样方式把A_1, A_2的右部代入左部为A_3右部以A_1或A_2开始的产生式中，消除左部为A_3的产生式之直接左递归，直到把左部为A_1, A_2, …A_{n-1}的右部替换左部为A_n的产生式中，从A_n中消除直接左递归。

算法归结

（1） 以某种顺序将文法非终结符排列$A_1, A_2 \ldots A_n$

（2） for i:=1 to n do

begin

for j:=1 to i-1 do

若A_j的全部产生式为：

$A_j \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$

替代形如$A_i \rightarrow A_j r$的产生式变为：

$A_i \rightarrow \alpha_1 r | \alpha_2 r | \ldots | \alpha_k r$

end

消除A_i规则的一切直接左递归;

end;

（3）去掉无用产生式

例如：有文法的产生式为：

(1) S →Qc|c

(2) Q →Rb|b

(3) R →Sa|a

该文法的每个非终结符为间接左递归。

若非终结符排序为S、Q、R。

左部为S的产生式(1)无直接左递归，(2)中右部不含S，所以把(1)右部代入(3)得：

(4) R →Qca|ca|a 再将(2)的右部代入(4)得：

(5) R →Rbca|bca|bca|a对(5)消除直接左递归得：

R →(bca|ca|a)R’

R’ →bcaR’| ε

最终文法变为：

S →Qc|c

Q →Rb|b

R →bcaR’|caR’|aR’

R’ →bcaR’| ε

4.4 不确定的自顶向下分析思想

假定要被代换的最左非终结符号是B，且有n条规则：$B \rightarrow A_1|A_2|\ldots|A_n$，那么如何确定用哪个右部去替代B？

- **LL(1)文法：确定的自顶向下分析法：根据规则的选择集合来确定。**
- **非LL(1)文法：不确定的自顶向下分析法－带回溯的自顶向下分析法**
- **“不确定”的意思：当某个非终结符的产生式有多个候选，而面临当前的输入符号无法确定选用哪个产生式，从而引起回溯。**

下面讲三个例子说明回溯：

1、由于相同左部的产生式的右部First集交集不为空而引起回溯。

例：文法S→xAy

A→ab|a

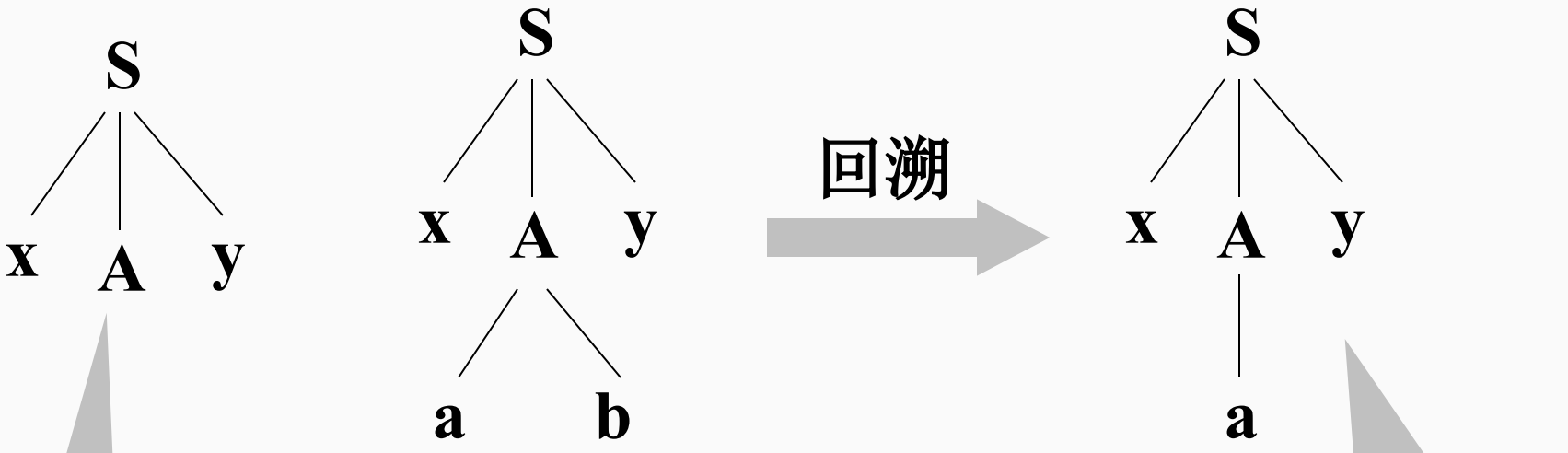
求输入串为xay语法树。

例：文法S→xAy

A→ab|a

求输入串为xay语法树。

first(ab)与first(a)的交集不为空



S→xAy

若选择A→ab，就于xay不匹配

回溯，用A→a就匹配

2、由于相同左部非终结符的右部能$\overset{*}{\Rightarrow}\varepsilon$，且该非终结符Follow集中含有其右部First集的元素而引起回溯。

例：$S \rightarrow aAS \mid b$

$A \rightarrow bAS \mid \varepsilon$

求对输入串ab#的推导树。

例：S → aAS |b

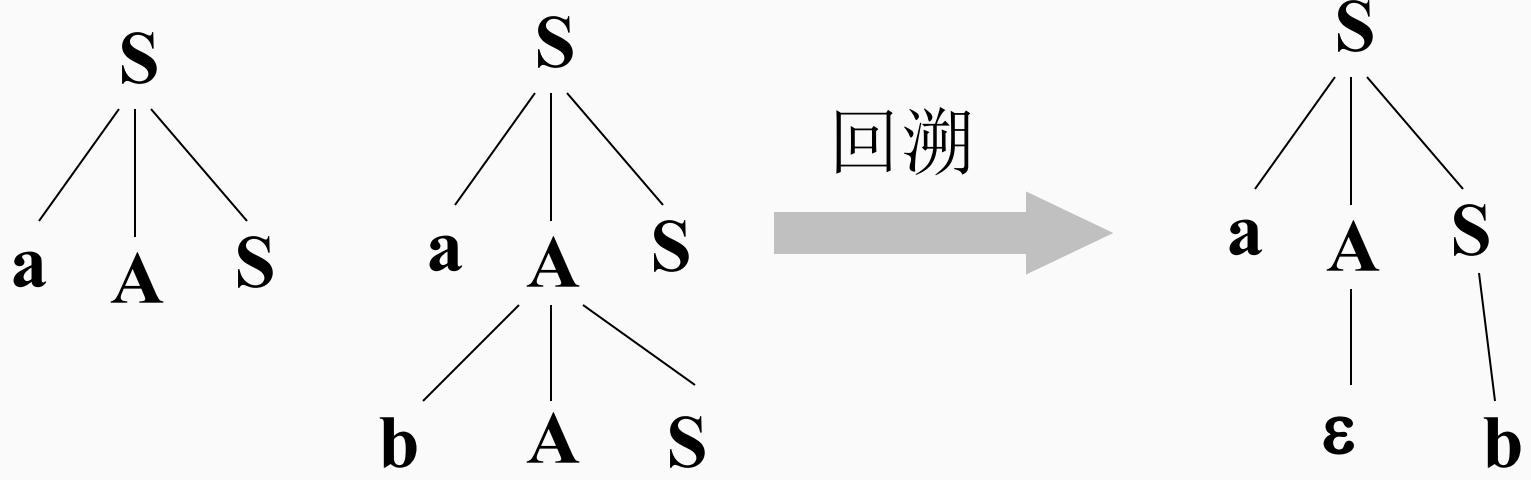
A →bAS |ε

求对输入串ab#的推导树。

A⇒ε

Follow(A)={a,b}

first(bAs)={b}



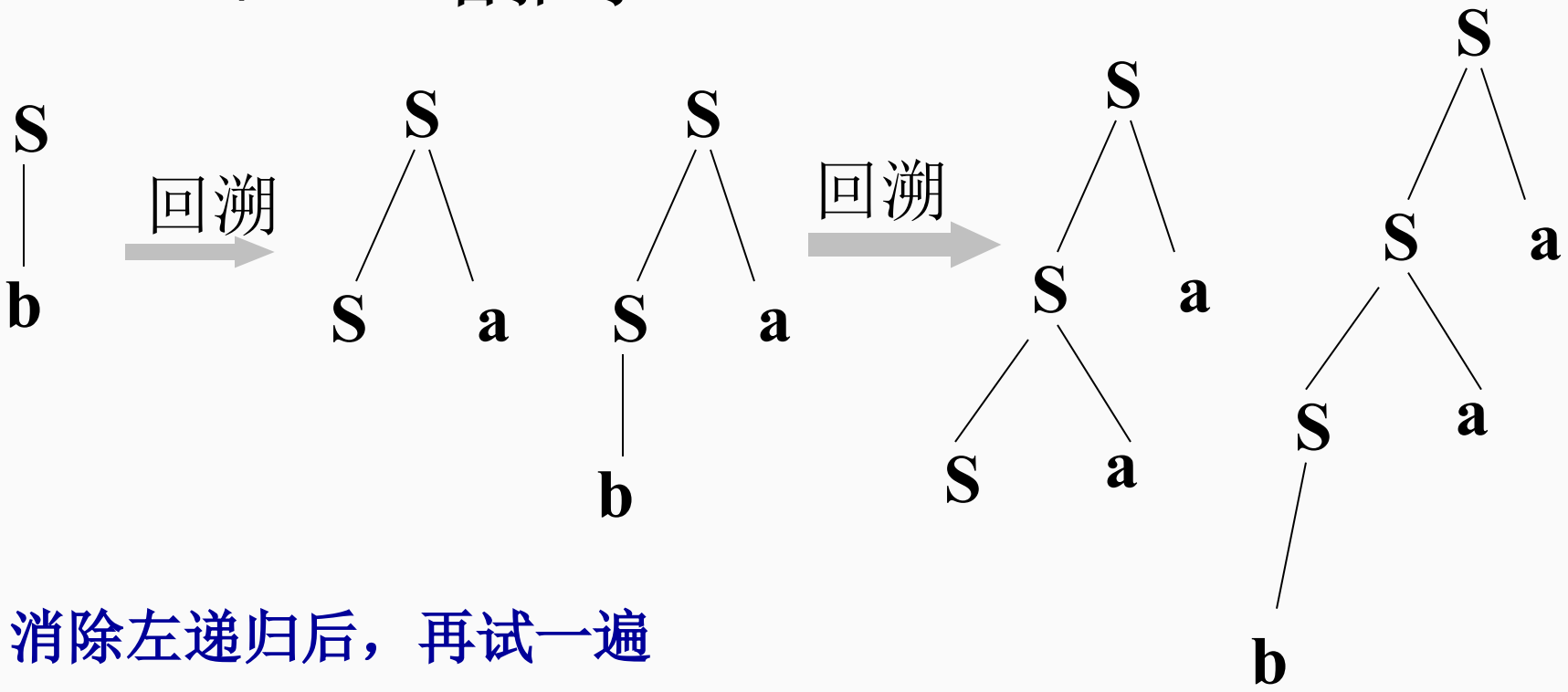
3、由于文法含有左递归而引起回溯

例：S→Sa

S→b　　若推导baa#

例：S→Sa

S→b　　若推导baa#



消除左递归后，再试一遍

由以上例子可以看出：带回溯的自顶向下分析是一个试探过程，当分析不成功时则推翻分析，退回到适当位置再重新试探其余可能的推导。

因此，需要记录所选过的产生式，直到把所有可能的推导序列都试完仍不成功，才能确认输入串不是该文法的句子。（为证明输入串不合法，需要穷举或遍历所有的推导）

编译程序真正实现时往往边分析边插入语义动作，因而带回溯分析代价很高，效率很低，在实用编译程序中几乎不用。

4.5 LL(1)分析的实现

1、递归下降LL(1)分析程序

要求文法满足LL(1)文法。是比较简单直观易于构造的一种语法分析方法。

基本思想是：对应文法中每个非终结符编写一个递归过程，每个过程的功能是识别由该非终结符推出的串。

1. 递归下降 LL（1）分析程序

–工作原理

每个非终结符都对应一个子程序。该子程序的行为根据语法描述来明确：根据下一个输入符号来确定按照哪一个产生式进行处理，再根据该产生式的右端，

- 每遇到一个终结符，则判断当前读入的单词是否与该终结符相匹配，若匹配，再读取下一个单词继续分析；不匹配，则进行出错处理
- 每遇到一个非终结符，则调用相应的子程序

✧非终结符对应的递归下降子程序

– 例 对于下列关于 function 的唯一产生式

<function> → FUNC ID (<parameter_list>) <statement>

（<function>，<parameter_list>，和 <statement> 是非终结符）

```
void ParseFunction()
{
   MatchToken(T_FUNC);      //匹配FUNC
   MatchToken(T_ID);        //匹配ID
   MatchToken(T_LPAREN); // 匹配 （
   ParseParameterList();
   MatchToken(T_RPAREN); // 匹配 ）
   ParseStatement();
}
```

✧非终结符对应的递归下降子程序

```
void MatchToken(int expected)
{
   if (lookahead != expected)  //判别当前单词是否与
   {                                    //期望的终结符匹配
      printf("syntax error \n");
      exit(0);
   }
  else                  // 若匹配,消掉当前单词并读入下一个
     lookahead = getToken();   //调用词法分析程序
}
```

注：Lookahead为全局量，存放当前所扫描单词符号的单词种别

递归下降 LL（1）分析程序

✧非终结符对应的递归下降子程序

– 一般结构

设 A 的产生式:

$A \to u_1 | u_2 | ... | u_n$,

相对于非终结符 A 的递归下降子程序 ParseA 的一般结构如右图所示

```
void ParseA()
{
    switch (lookahead) {
        case SELECT(A→u1):
            /* 根据u1设计的分析过程*/
            break;
        case SELECT(A→u2):
            /* 根据u2设计的分析过程 */
            break;
        ...
        case SELECT(A→un):
            /*根据un设计的分析过程 */
            break;
        default:
            printf("syntax error \n");
            exit(0);
    }
}
```

✧递归下降LL(1)分析程序举例

– 例 对于下列文法 G(S):

$S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

First（*AaS*）= {*a*}

First（*BbS*）= {*c*,*b*}

First（*d*）= {*d*}

First（*a*）= {*a*}

First（ε）= {ε}

First（*c*）= {*c*}

Follow（*S*）= {#}

Follow（*A*）= {*a*}

Follow（*B*）= {*b*}

产生式	select集合
(*S*→*AaS*)	{*a*}
(*S*→*BbS*)	{*c*,*b*}
(*S*→*d*)	{*d*}
(*A*→*a*)	{*a*}
(*B*→ε)	{*b*}
(*B*→*c*)	{*c*}

Select(*S*→*AaS*)，*Select*(*S*→*BbS*) 以及 *Select*(*S*→*d*) 互不相交，*Select*(*B*→ε) 和 *Select*(*S*→*d*)互不相交，所以，G(S) 是 LL(1) 文法

递归下降 LL（1）分析程序

– 接上例 针对文法G(S)构造的递归下降分析程序

G(S): $S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

$PS(S \rightarrow AaS) = \{a\}$

$PS(S \rightarrow BbS) = \{c,b\}$

$PS(S \rightarrow d) = \{d\}$

```
void ParseS( )
{
    switch (lookahead)  {
        case a:
            ParseA( );
            MatchToken(a);
            ParseS( );
            break;
        case b,c:
            ParseB( );
            MatchToken(b);
            ParseS( );
            break;
        case d:
            MatchToken(d);
            break;
        default:
            printf("syntax error
\n")
            exit(0);
    }
}
```

递归下降 LL（1）分析程序

– 接上例 针对文法G(S) 构造的递归下降分析程序

G(S): $S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

$PS(A \rightarrow a) = \{a\}$

$PS(B \rightarrow \varepsilon) = \{b\}$

$PS(B \rightarrow c) = \{c\}$

```
void ParseB( )
{
    if (lookahead==c)  {
        MatchToken(c);
    }
   else if (lookahead==b) {
   }
   else {
       printf("syntax error \n”);
       exit(0);
   }
}
```

```
void ParseA( )
{
    if (lookahead==a)  {
        MatchToken(a);
    }
    else {
       printf("syntax error \n”);
       exit(0);
    }
}
```

2、表驱动LL(1)分析程序

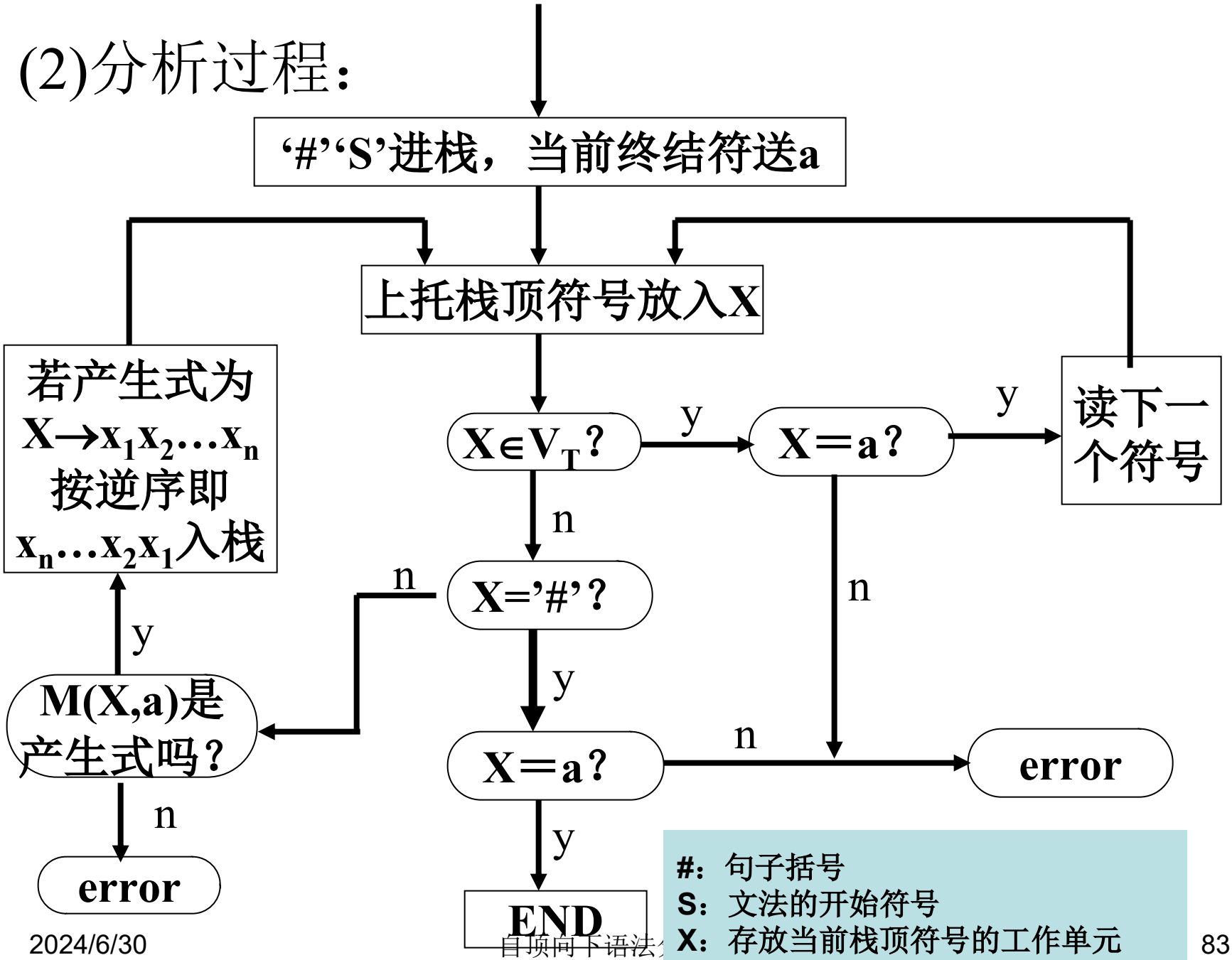
自顶向下分析的另一种方法

(1)表驱动LL(1)分析器的组成

- 预测分析程序
- 先进后出栈
- 预测分析表－与文法有关

- 预测分析表可用一个矩阵表示。
- 矩阵元素用M[A,a]表示，A表示非终结符， a表示终结符或句子结束符#
- 矩阵元素M[A,a]中的内容是一条关于A的产生式，表明当用非终结符A向下推导时，面临输入符a时，所采用的候选产生式。
- 当元素内容无产生式时，表明用A为左部向下推导时遇到了不该出现的符号，因此元素内容为转向出错处理。

(2)分析过程：


‘#’‘S’进栈，当前终结符送a
上托栈顶符号放入X
若产生式为
X→x1x2…xn
按逆序即
xn…x2x1入栈
X∈VT?
y
X=a?
y
读下一
个符号
n
n
X=’#’?
y
n
M(X,a)是
产生式吗？
y
X=a?
n
error
n
y
error
END
#：句子括号
S：文法的开始符号
X：存放当前栈顶符号的工作单元
a：存放当前输入符号a的工作单元

(3)实例分析：给定文法，构造预测分析表，并针对输入串i+i*i#构造预测分析过程。

例：文法为：E →E+T | T

T → T*F | F

F → i | (E)

步骤：

（1）判断文法是否为LL(1)文法。

✓如果文法中含有左递归，必须先消除左递归：

（2）构造预测分析表： Select(A → α)

（3）列出预测分析过程

例：文法为：$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow i \mid (E)$

$S \rightarrow Sa$ $\quad S \rightarrow bS'$

$S \rightarrow b$ $\quad S' \rightarrow aS' \mid \varepsilon$

$E \rightarrow E+T$

$E \rightarrow T$

构造步骤有：

❶ 判断文法是否为LL(1)文法。

✓ 由于文法中含有左递归，所以必须先消除左递归：

$E \rightarrow E+T \mid T$ ⇒

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow T*F \mid F$ ⇒ $T \rightarrow FT'$, $T' \rightarrow *FT' \mid \varepsilon$

文法变为：⇒

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow i \mid (E)$

✓ 求First集合：

First(E)={ (， i }

First(E ′)={ + ， ε}

First(T)={ (， i }

First (T ′)={ * ， ε}

First (F)={ (， i }

✓求Follow集：

Follow (E)={) ， #}

Follow (E ′)={) ， #}

Follow (T)={ + ，) ， #}

Follow (T ′)={ + ，) ， #}

Follow (F)={* ， + ，) ， #}

求Select集：

Select (E→TE′)={ (， i }

Select (E′ → +TE′)={+}

Select (E′ → ε)={) ， #}

Select (T → FT ′)={(， i}

Select (T′ → *FT′)={*}

Select (T′ → ε)={ +，)， #}

Select (F → i)={i}

Select (F → (E))={(}

由上可知有相同左部产生式的Select集合的交集为空，所以文法是LL (1)

❷构造预测分析表：

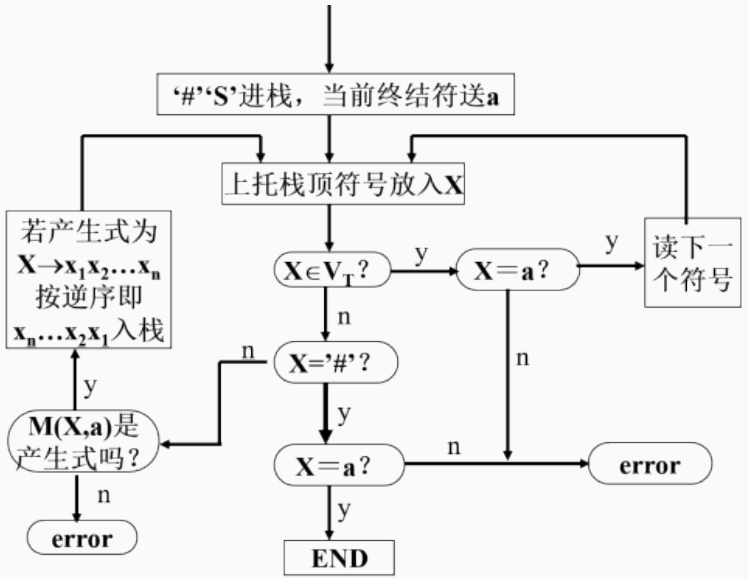
方法：对每个终结符或#用a表示。

若a∈Select(A→a)，则A→a放入M[A,a]中。

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

分析输入串#i+i#的步骤

	栈内容	栈顶符号	当前输入	余留串	M[X,b]
1	#E	E	i	+i#	E –> TE’
2	#E’T	T	i	+i#	T –> FT’
3	#E’T’F	F	i	+i#	F –> i
4	#E’T’i	i	i	+i#	
5	# E’T’	T’	+	i#	T’ –> ε
6	#E’	E’	+	i#	E’ –> +TE’
7	#E’T+	+	+	i#	
8	# E’T	T	i	#	T –> FT’
9	#E’T’F	F	i	#	F –> i
10	#E’T’i	i	i	#	
11	#E’T’	T’	#		T’ –> ε
12	#E’	E’	#		E ’ –> ε
13	#	#	#		



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

4.6 LL(1)分析中的出错处理

- 错误处理的任务
  - 报错，尽可能准确指出错误位置和错误属性
  - 错误恢复，尽可能进行校正

4.6.1 应急恢复

✧ 表驱动LL(1)分析中的错误处理

- 出错报告（error reporting）
  - 栈顶的终结符与当前输入符不匹配
  - 非终结符*A*于栈顶，面临的输入符为a，但分析表*M*的*M*[*A*,*a*]为空

✧表驱动LL(1)分析中的错误处理

– 简单的应急恢复（*panic-mode* error recovery）跳过输入串中的一些符号直至遇到同步符号（*synchronizing* token）为止

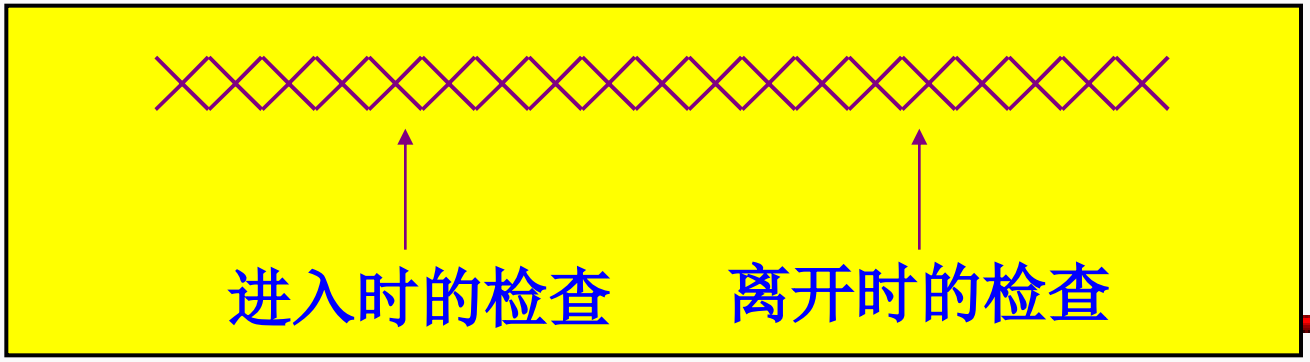
同步符号的选择：

- 把 Follow(A) 中的所有符号作为A的同步符号, 跳过输入串中的一些符号直至遇到这些“同步符号”，把A从栈中弹出，可使分析继续
- 把First(A)中的符号加到A的同步符号集，当First(A)中的符号在输入中出现时，可根据A恢复分析

4.6.2 短语层恢复

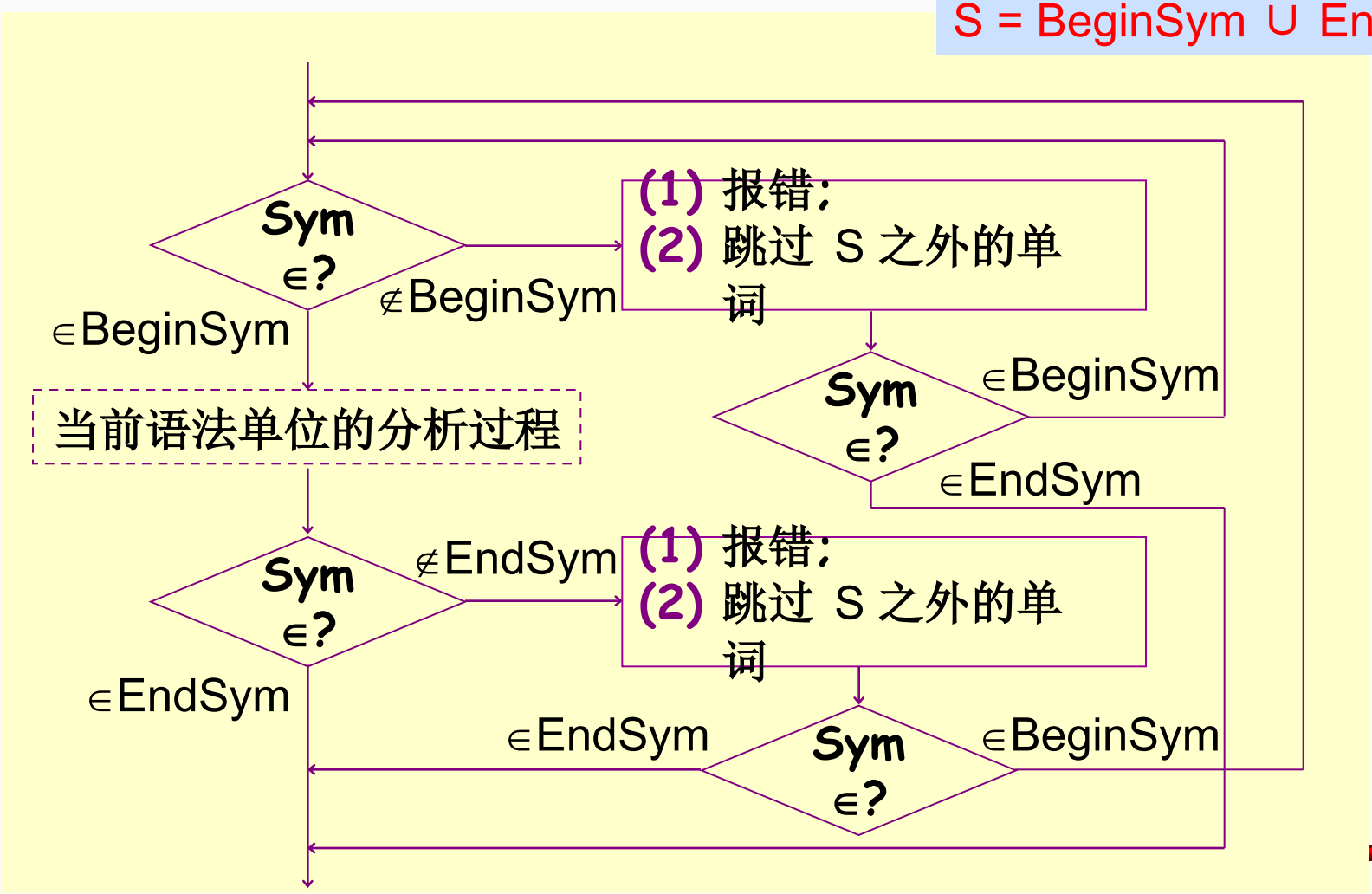
短语层恢复可采取的流程

- 在进入某个语法单位时，检查当前符号是否属于进入该语法单位需要的符号集合 BeginSym. 若不属于，则报错，并滤去补救的符号集合 S=BeginSym∪EndSym外的所有符号
- 在语法单位分析结束时，检查当前符号是否属于离开该语法单位时需要的符号集合EndSym. 若不属于，则报错，并滤去补救的符号集合 S=BeginSym∪EndSym 外的所有符号



短语层恢复可采用的流程

Sym为正扫描的符号
S为补救的符号集合
S = BeginSym ∪ EndSym



短语层恢复举例

$B \rightarrow [\ A\ ] \mid (\ A\ )$
$A \rightarrow a$

```
procedure ParseB ( EndSym )
{
   if ( sym ∉ { '[', '(' } ) {
          报错; 跳过 S 之外的单词;     /* S = { '[', '(' } ∪ EndSym */
   }
  while ( sym ∈ { '[', '(' } ) {
     if (sym == '[')
        { MatchToken('['); ParseA ( EndSym∪{']'} ); }
     else { MatchToken('('); ParseA ( EndSym∪{')'} ); }
     if ( sym ∉ EndSym ) {
           报错; 跳过 S 之外的单词;     /* S = { '[', '(' } ∪ EndSym */
        }
  }
}
```

短语层恢复举例

$B \rightarrow [\ A\ ] \mid (\ A\ )$
$A \rightarrow a$

```
procedure ParseA ( EndSym )
{
    if ( lookahead ∉ { 'a' } ) {
            报错; 跳过 S 之外的单词;     /* S = { 'a' } ∪ EndSym */
    }
    while ( lookahead ∈ { 'a' } ) {
        MatchToken ( 'a' );
        if ( lookahead ∉ EndSym ) {
                报错; 跳过 S 之外的单词;     /* S = { 'a' } ∪ EndSym */
            }
    }
}
```

课后作业

- **1: (3), (4)**
- **2: (1), (2), (3)**