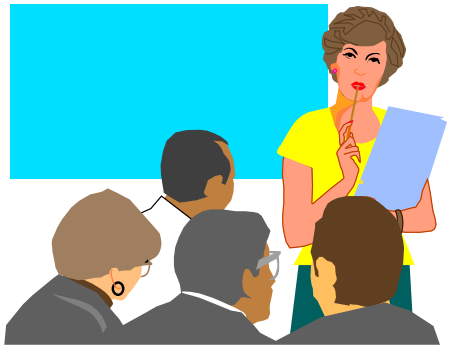
编译技术



胡春明
hucm@buaa.edu.cn

2019.9-2019.12

第四章 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

语法分析的任务

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

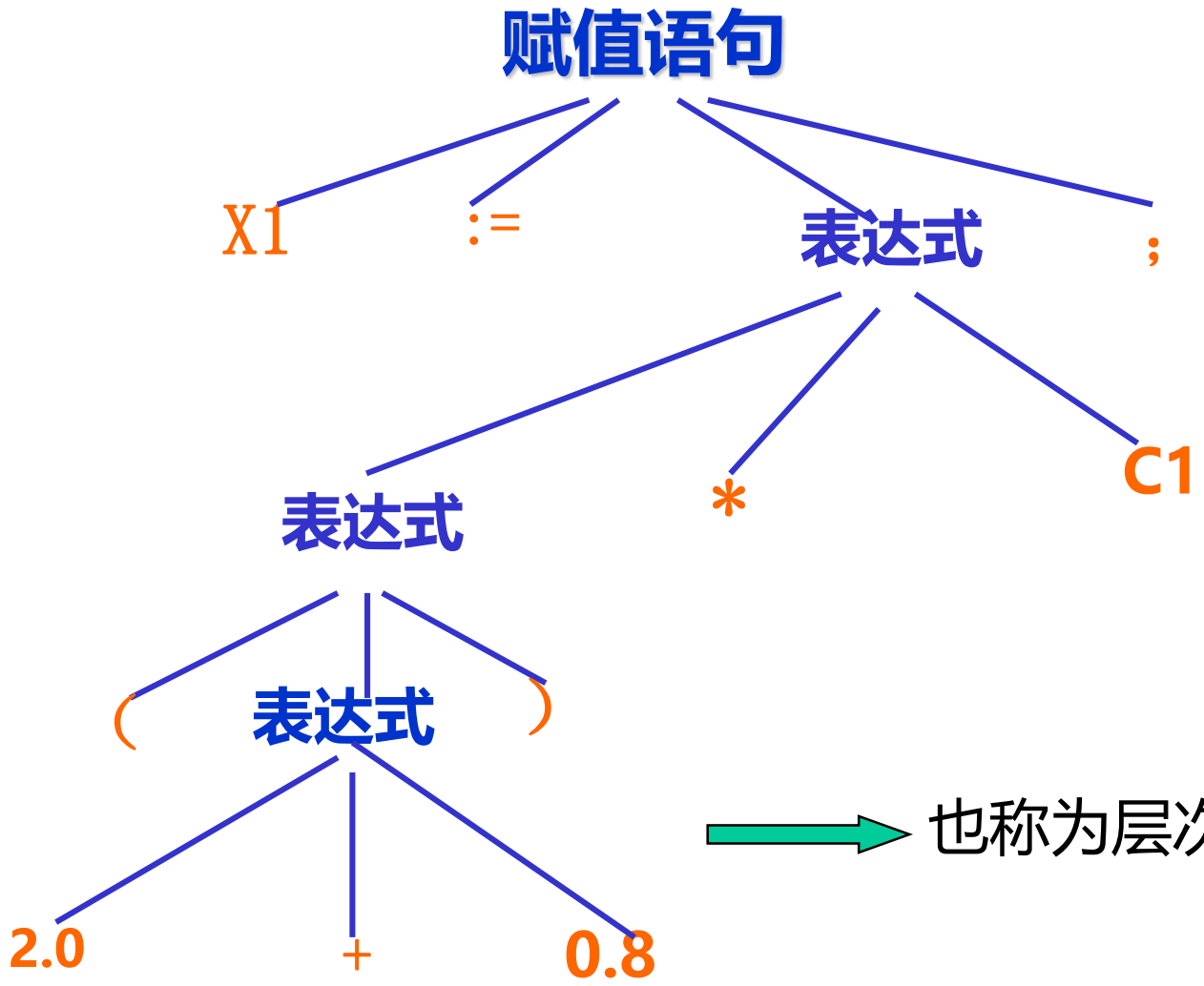
任务：根据语法规则（即语言的文法），分析并识别出各种语法成分，如表达式、各种说明、各种语句、过程、函数等，并进行语法正确性检查。

X1:= (2.0 + 0.8) * C1

赋值语句的文法：

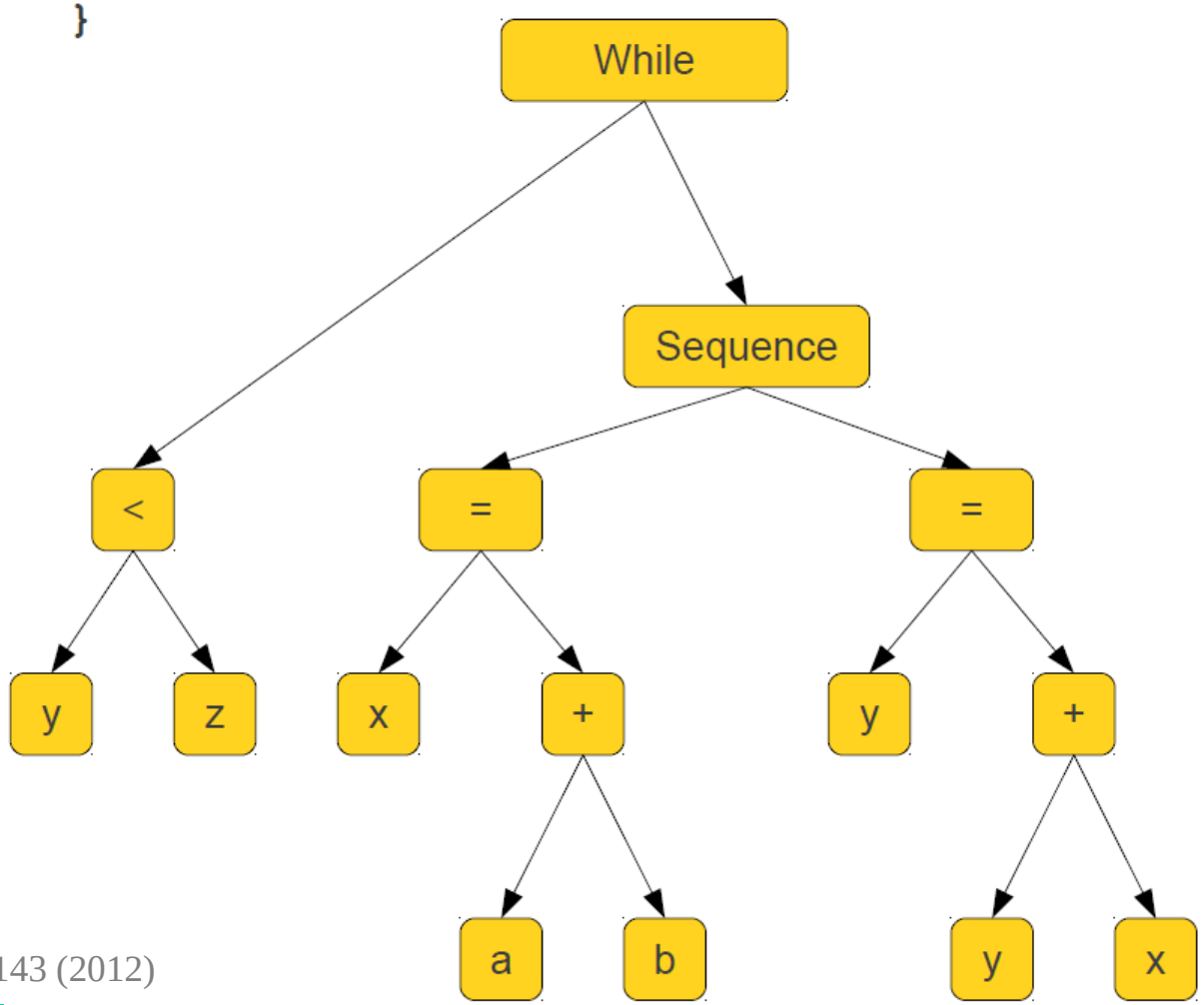
<赋值语句>→<变量><赋值操作符><表达式>
<变量>→<简单标识符>
<赋值操作符>→:=
<表达式>→ ……

X1:= (2.0 + 0.8) * C1;



→ 也称为层次分析。

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



Source: Stanford CS143 (2012)

4.1 语法分析（syntax analysis, parsing）

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，报告错误。

基本任务：识别符号串S是否为某语法成分。

依据：文法

？和词法分析有什么不同？

<条件语句>::=if <条件> then <语句>
<当循环语句>::=while <条件> do <语句>
<复合语句>::=begin <语句> {;<语句>} end
……

4.1 语法分析（syntax analysis, parsing）

依据：文法

？和词法分析有什么不同？

<条件语句>::=if <条件> then <语句>
<当循环语句>::=while <条件> do <语句>
<复合语句>::=begin <语句> {;<语句>} end
.....

差别1：文法的类型

词法分析：3型（正则文法）
语法分析：2型（上下文无关文法）

差别2：字母表的变化

词法分析：字符串
语法分析：符号串

4.1 语法分析（syntax analysis, parsing）

依据：文法

？和词法分析有什么不同?

<条件语句>::=if <条件> then <语句>
<当循环语句>::=while <条件> do <语句>
<复合语句>::=begin <语句> {;<语句>} end
……

差别1：文法的类型

词法分析：3型（正则文法）
语法分析：2型（上下文无关文法）

Context Free Grammar (CFG)

差别2：字母表的变化

词法分析：字符串
语法分析：符号串

自顶向下（Top-Down）分析：推导（Derivations）

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

自底向上（Bottom-Up）分析：规约（Reductions）

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

自顶向下（Top-Down）分析：推导（Derivations）

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

（1）推导顺序：有多个“非终结符”，优先用哪个?

（2）避免二义性：避免文法有多个可用规则

? 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（Bottom-Up）分析：

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

自顶向下分析

4.2 自顶向下分析

4.2.1 自顶向下分析的一般过程

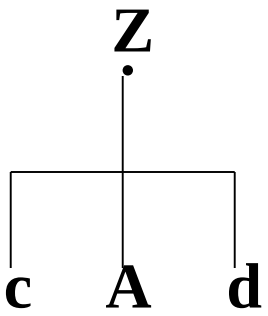
给定符号串S，若预测是某一语法成分，则可根据该语法成分的文法,设法为S构造一棵语法树，
若成功,则S最终被识别为某一语法成分,即
$S \in L(G[Z])$，其中G[Z]为某语法成分的文法
若不成功, 则 $S \notin L(G[Z])$

- 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

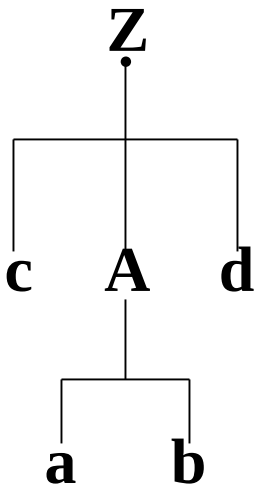
2. 用Z的右部符号串去匹配输入串

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

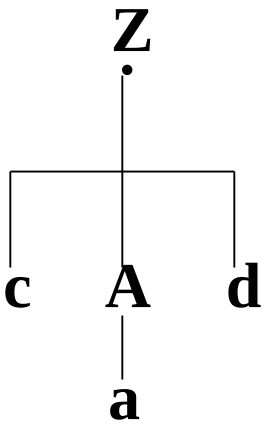
S=cad　　G[Z]:　Z∷=cAd
A∷=ab|a

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。

二义性问题

2.4.2 文法的二义性

定义14.1 若对于一个文法的某一句子（或句型）存在两棵不同的**语法树**，则该文法是**二义性文法**，否则是无二义性文法。

换而言之，无二义性文法的句子**只有一棵语法树**，尽管推导过程可以不同。

二义性文法举例：

G[E]: E ::= E+E | E*E | (E) | i

Vn={E}

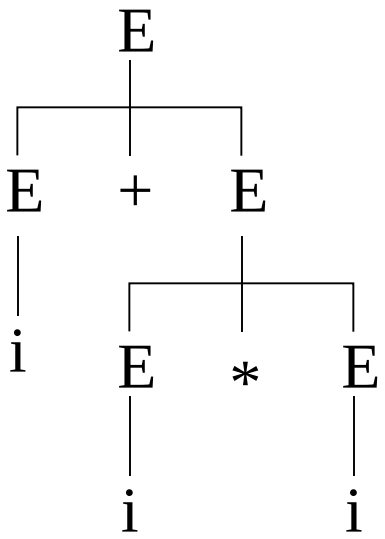
Vt={ +, * , (,) , i }

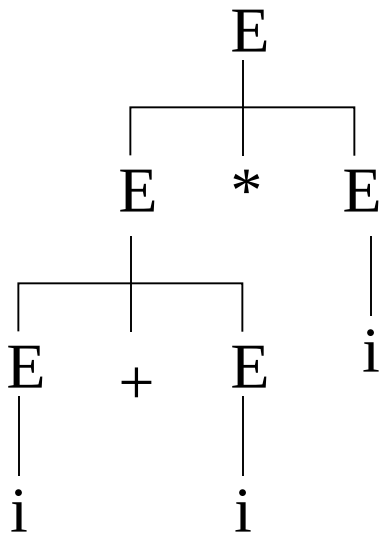
对于句子S = i + i * i ∈ L(G[E])，存在不同的规范推导:

(1) E=⊨>E+E=⊨>E+E*E =⊨>E+E*i =⊨>E+i*i =⊨> i + i * i

(2) E=⊨> E*E =⊨=> E*i =⊨> E+E*i =⊨> E+i*i =⊨=> i + i * i

这两种不同的推导对应了两棵不同的语法树:

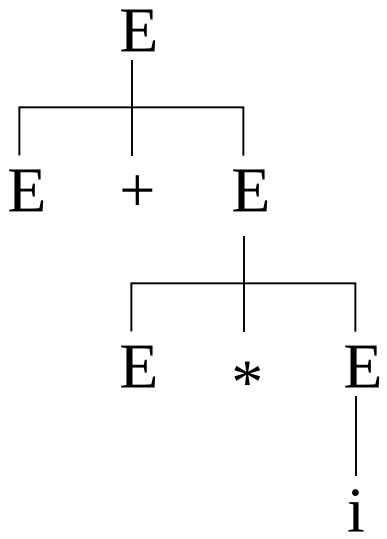




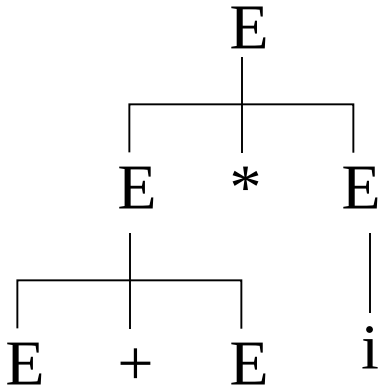
定义14.2 若一个文法的某句子存在两个不同的规范推导，则该文法是二义性的，否则是无二义性的。

(2) E=╞=> E*E ╡=> E*i ╡=> E+E*i ╡=> E+i*i ╡=> i + i * i

从自底向上的归约过程来看，上例中规范句型E+E*i 是由i + i * i通过两步规范归约得到的，但对于同一个句型E+E* i，它有两个不同的句柄（对应上述两棵不同的语法树）：i 和 E + E。因此，文法的二义性意味着句型的句柄不唯一。



句柄：i



句柄：E + E

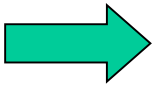
定义14.3 若一个文法的某规范句型的句柄不唯一，则该文法是二义性的，否则是无二义性的。c

若文法是二义性的，则在编译时就会产生不确定性，遗憾的是在理论上已经证明：**文法的二义性是不可判定的**，即不可能构造出一个算法，通过有限步骤来判定任一文法是否有二义性。

现在的解决办法是：提出一些**限制条件**，称为无二义性的充分条件，当文法满足这些条件时，就可以判定文法是无二义性的。

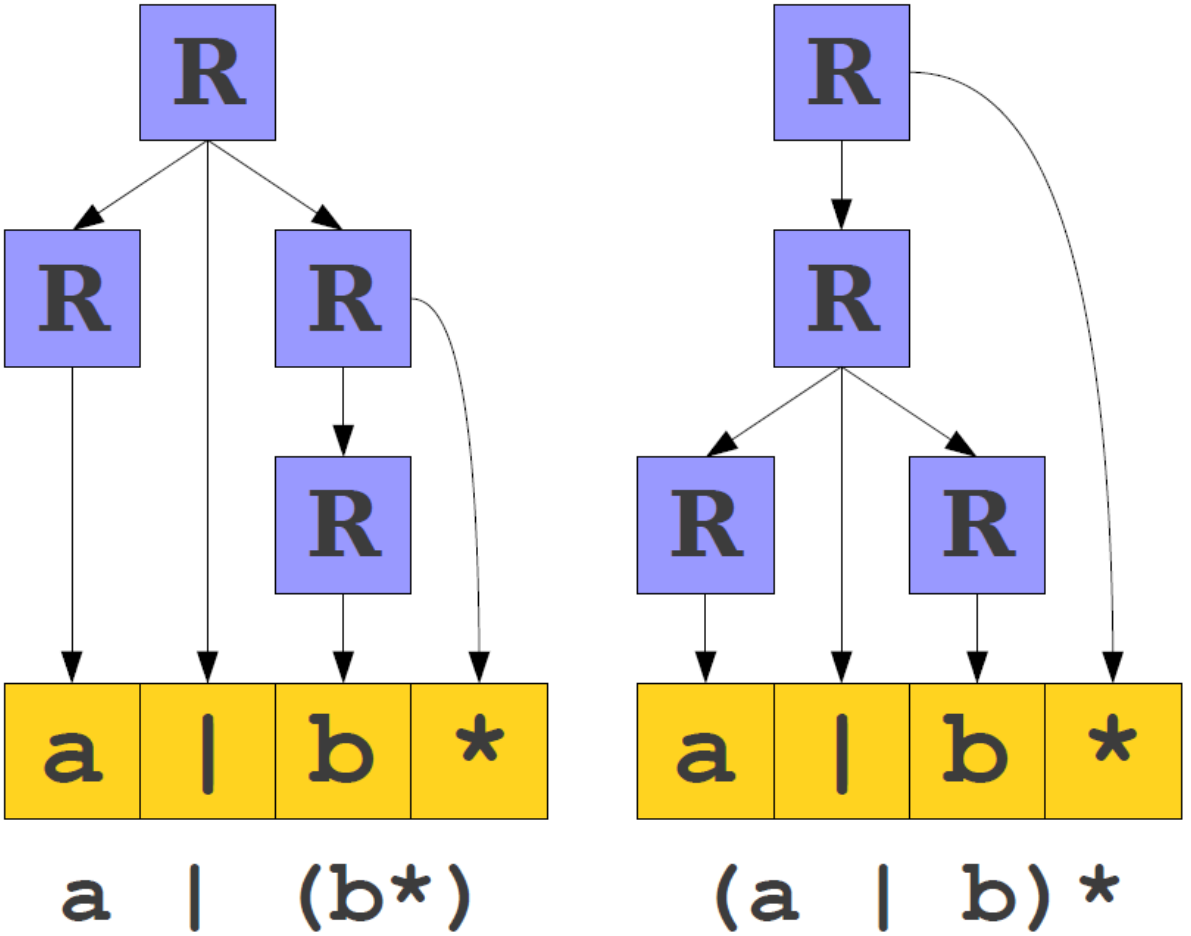
例:算术表达式的文法

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



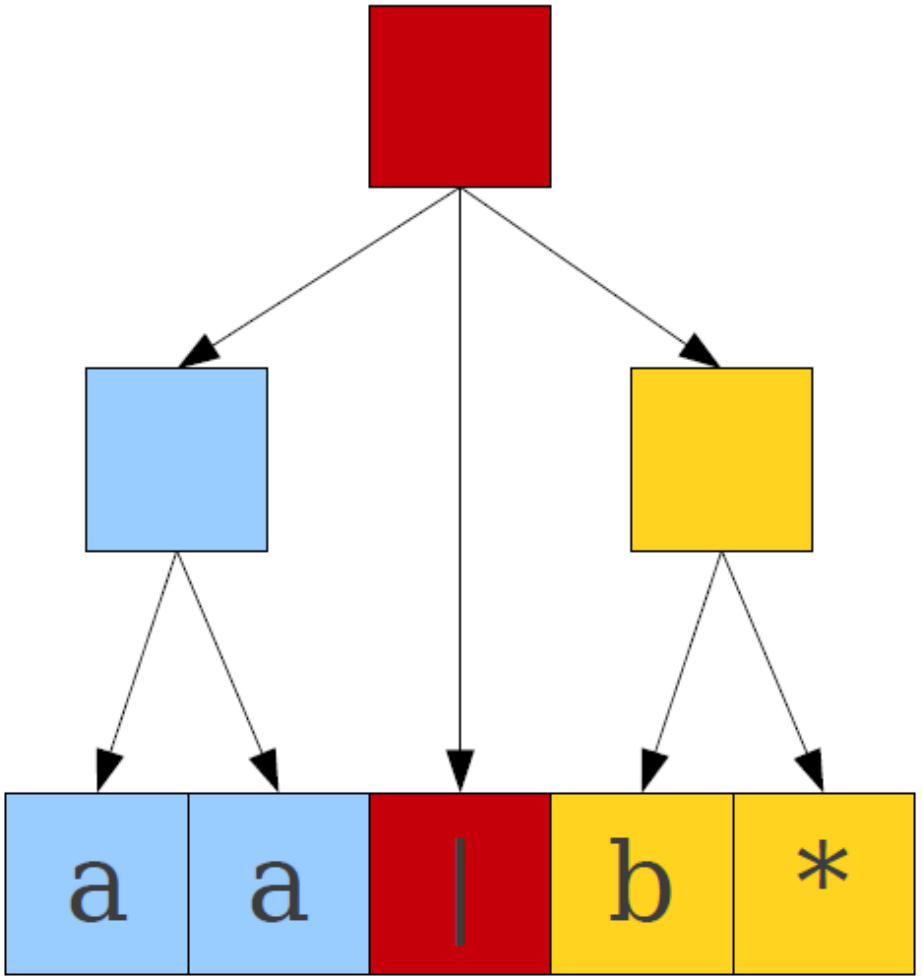
E::= E+T | T
T ::= T*F | F
F ::= (E) | i

R → a | b | c | …
R → “ε”
R → RR
R → R “|” R
R → R*
R → (R)



Source: Stanford CS143 (2012)

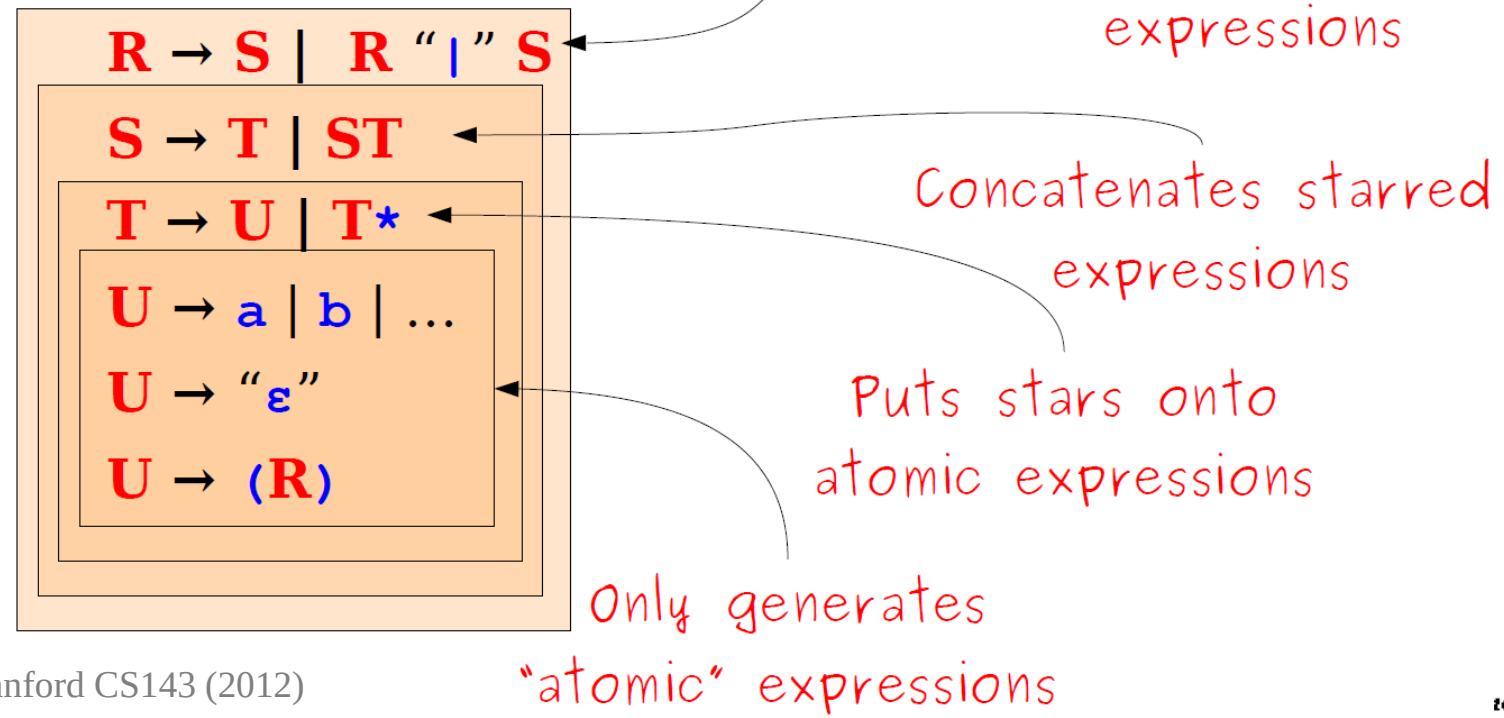
R → a | b | c | …
R → “ε”
R → RR
R → R “|” R
R → R*
R → (R)



Source: Stanford CS143 (2012)

优先级的实现：层次定义

越靠上层，优先级越低；越靠下层，优先级越高



Source: Stanford CS143 (2012)

左递归问题

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 $U \overset{+}{\Rightarrow} U\cdots$

这个文法是左递归的。

自顶向下分析的基本缺点是：

不能处理具有左递归性的文法

为什么？

如果在匹配输入串的过程中，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，这样无限的循环下去将无法终止。

如果文法具有间接左递归，则也将发生上述问题，只不过环的圈子兜得更大。

要实行自顶向下分析，必须要消除文法的左递归，下面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|T ⇒ E::=T{+T}

(2) T::=T*F|T/F|F ⇒ T ::=F{*F|/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，
可以给出两条规则:

规则一（提因子）

若：U::=xy|xw|….|xz

则可改写为：U::=x(y|w|….|z)

若：$y=y_1y_2$, $w=y_1w_2$

则　U::=x(y_1(y_2|w_2)|….|z)

若有规则：U::=x|xy

则可以改写为：U::=x(y|ε)

注意：不应写成U::=x(ε|y)

使用提因子法，不仅有助于消除直接左递归，而且有助于压缩文件的长度，使我们能更有效地分析句子。

规则二

若有文法规则：U::=x|y|……|z|Uv

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其后随有零个或多个v组成。

$U \Rightarrow Uv \Rightarrow Uvv \Rightarrow Uvvv \Rightarrow \cdots\cdots$

∴ 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，并保持文法的等价性。

方法二，将左递归规则改为右递归规则

规则三

若：$P ::= P\alpha | \beta$

则可改写为：$P ::= \beta P'$

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

规则一：（提因子）

规则二：U∷=x|y|……|z|Uv, 则U∷=(x|y|……|z){v}

规则三：右递归P∷=Pα| β，则P ∷= β P’，P’ ∷= αP’| ε

例1　E∷=E+T|T

右部无公因子，所以不能用规则一。

为了使用规则二，

令E∷=T|E+T

∴由规则二可以得到

E∷=T{+T}

例2　T∷=T*F|T/F|F

T∷=T(*F|/F) | F　规则一

T∷=F|T(*F|/F)

T∷=F{(*F|/F)}　规则二

即T∷=F{*F|/F}

右递归：

T ::= FT’

T’ ::= *FT’ | /FT’ | ε

消除一般左递归

一般左递归也可以通过改写文法予以消除。

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：

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

$A_2 ::= A_1 r\cdots\cdots$

$A_3 ::= A_2u \ | A_1v\cdots..$

……。

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$A_j ::= \delta_1|\delta_2|\ldots\ldots|\delta_k$是当前全部$A_j$ 的规则;
     消除A_i规则中的直接左递归
   end

3. 化简由2得到的文法即可。

例：文法G[s]为

S ::= Qc|c

Q ::= Rb|b

R ::= Sa|a

该文法无直接左递归，但有间接左递归

$S \Rightarrow Qc \Rightarrow Rbc \Rightarrow Sabc$　　$\therefore S \overset{+}{\Rightarrow} Sabc$

非终结符顺序重新排列

R::=Sa|a

Q::=Rb|b

S::=Qc|c

R∷=Sa|a
Q∷=Rb|b
S∷=Qc|c

1. 检查规则R是否存在直接左递归　　R∷=Sa|a

2. 把R代入Q的有关选择，改写规则Q　　Q∷=Sab|ab|b

3. 检查Q是否存在直接左递归

4. 把Q代入S的右部选择　　S∷=Sabc|abc|bc|c

5. 消除S的直接左递归　　S∷=(abc|bc|c){abc}

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

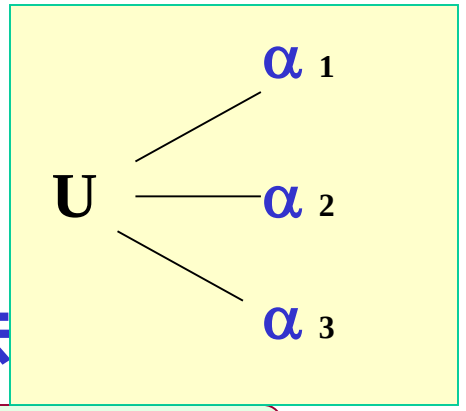
∴经过压缩后 S∷=(abc|bc|c){abc}

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。

回溯问题

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

造成回溯的条件:

U::= α 1 | α 2 | α 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选择时，就可能出现回溯。

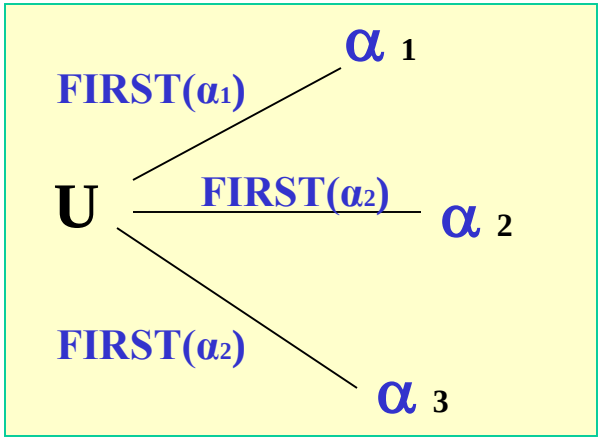
回溯带来的问题:

严重的低效率，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) 语义处理工作要推倒重来



设文法G（不具左递归性），$U \in V_n$

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

[定义] $FIRST(\alpha_i) = \{a \mid \alpha_i \overset{*}{\Rightarrow} a\ldots, a \in V_t\}$

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::=xV|xW　　$U, V, W \in V_n$，　$x \in V_t^+$

改写为U::=x(V|W)

更清楚地表示为：

U::=xZ

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::=ab|cd　　FIRST(V) = {a,c}

W::=de|fg　　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::= ab　w::= ac

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（偷看）

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，但是假读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的，故读到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”的首符，便是第一个选择；若是“语句”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，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。

[定义] 设文法G（不具有左递归性），$U \in V_n$

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

$FIRST(\alpha_i) = \{a | \alpha_i \stackrel{*}{\Rightarrow} a\ldots, \ a \in V_t\}$

为避免回溯，对文法的要求是:

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

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。

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 $\# \in FOLLOW(A)$

不带回溯的充分必要条件是：对于G的每一个非终结符A的任意两条规则$A::=\alpha|\beta$,下列条件成立：

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

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

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。

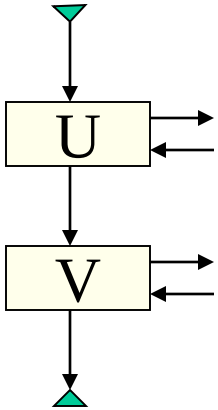
下面通过举例说明如何根据文法构造该文法的语法分析程序

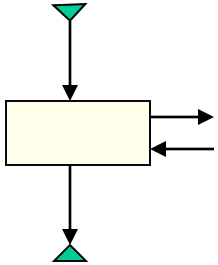
如文法G[Z]:　Z ::= UV

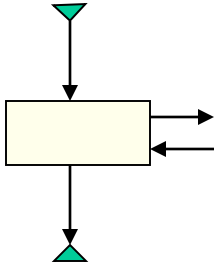
U ::= ….

V ::= ….

Z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：

U ::= …U…　　U ::= …W…

W::= …U…

例：文法G[Z]

Z::='('U')'|aUb

U::=dZ|Ud|e

1.检查并改写文法

Z::=' ('U') '|aUb

U::=(dZ|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⇒¨·$\dot{Z}$¨·

U⇒¨·Z¨·⇒¨·U¨·　　∴U⇒¨·$\dot{U}$¨·

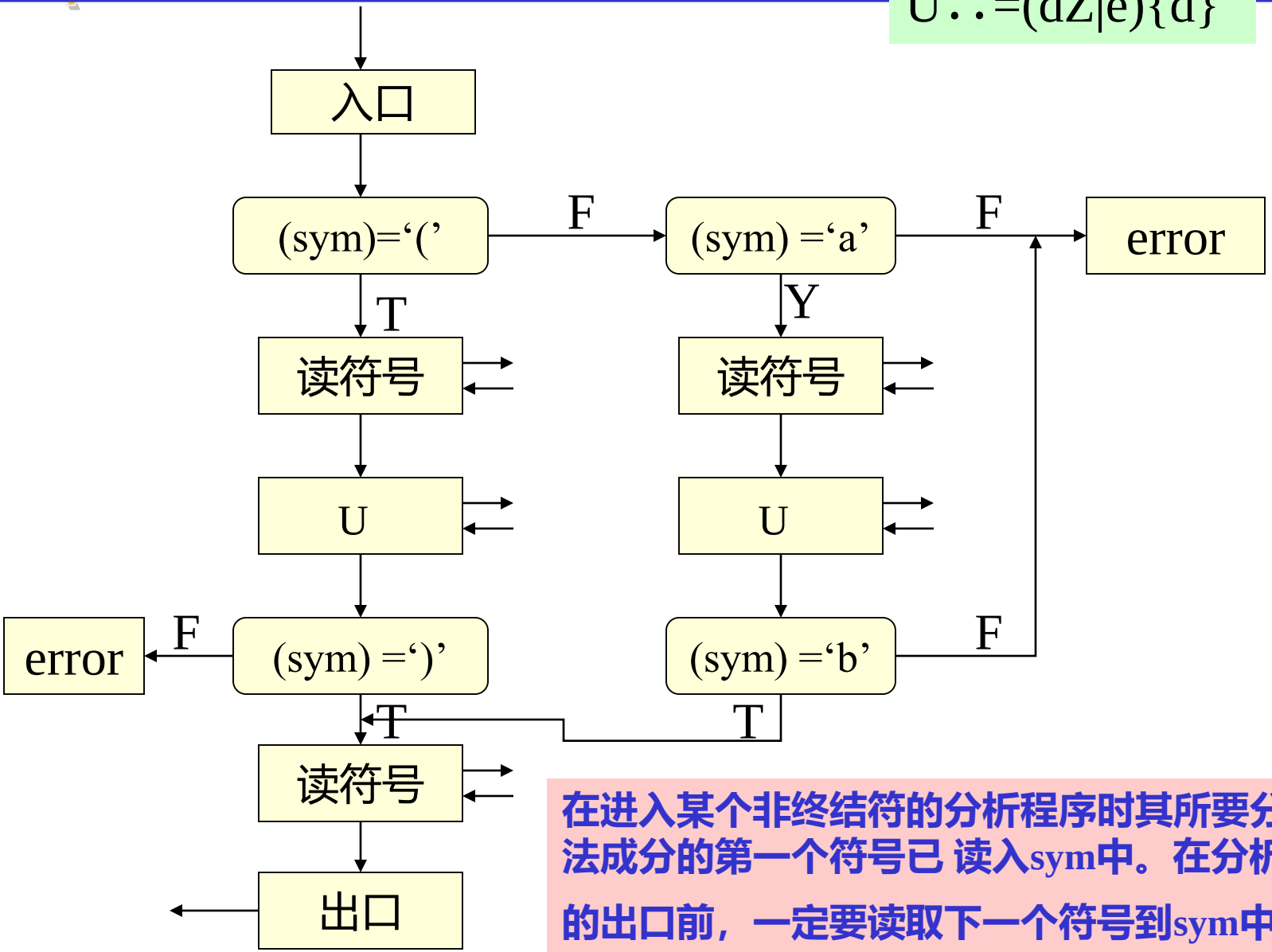
因此，Z和U的分析程序要编成递归子程序

3.算法框图

非终结符号的分析子程序的功能是：
用规则右部符号串去匹配输入串。

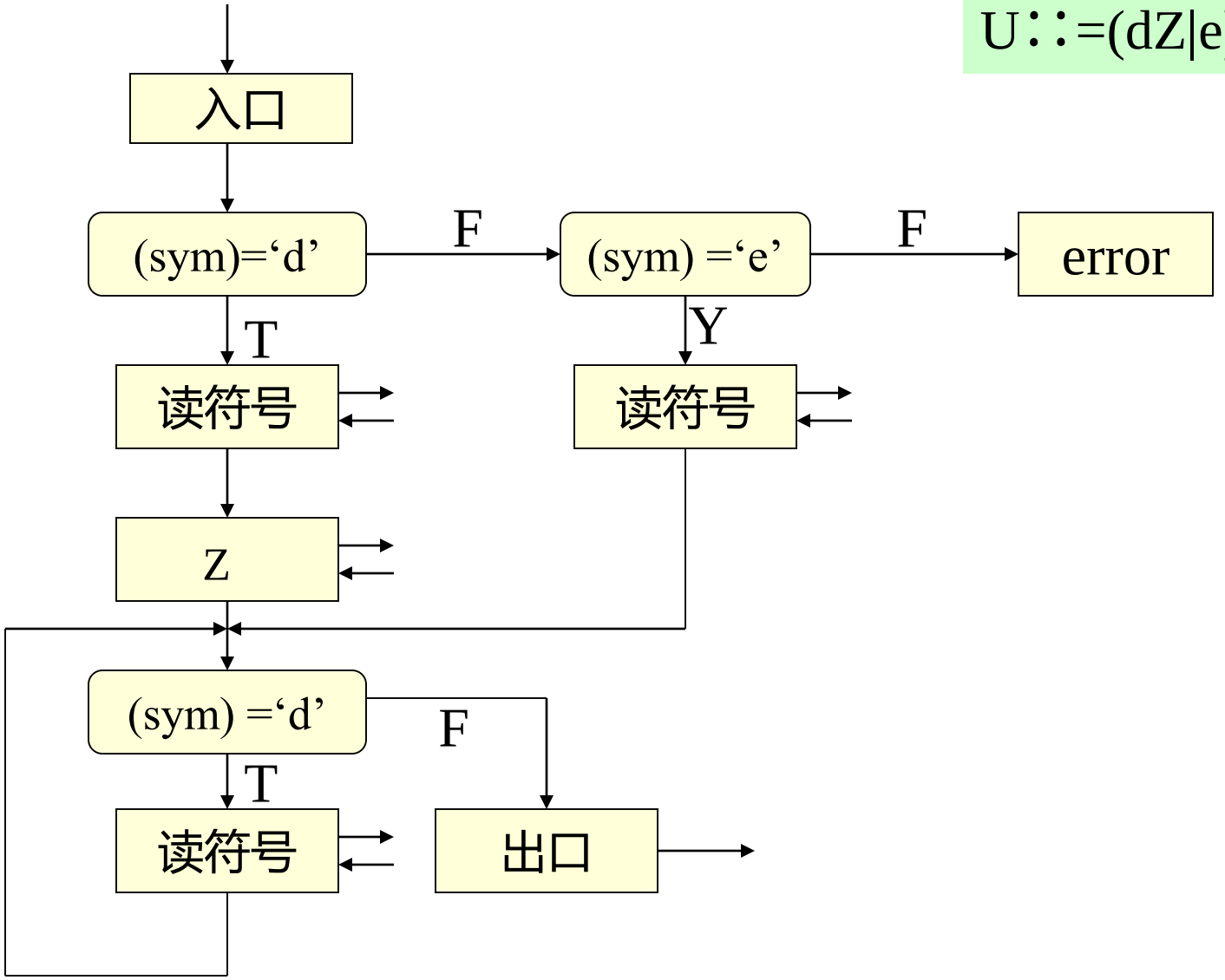
以下是以框图形式给出的两个子程序：

Z::='('U')'|aUb
U::=(dZ|e){d}



在进入某个非终结符的分析程序时其所要分析的语法成分的第一个符号已 读入sym中。在分析子程序的出口前，一定要读取下一个符号到sym中。

Z::='('U')'|aUb
U::=(dZ|e){d}



说明

- 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时其所要分析的语法成分的第一个符号已读入sym中。

递归子程序法对应的是最左推导过程

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>')'

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;                          /*语句分析子程序*/
   IF  sym =‘IF’ THEN
      BEGIN    nextsym;  expr;
          IF  sym≠‘THEN’  THEN   error
              ELSE  BEGIN  nextsym; state;
                      IF  sym= ‘ELSE’
                        THEN   BEGIN
                                    nextsym;
                                    state;
                                END
                    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
```

<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>‘)’

```
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
```

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

```
void  statement( )
{
    if (sym == " IF")  {
            getsym ( );
            expr ( );
            if (sym != " THEN")
                    error( ) ;
            else {  getsym ( );
                  statement ( );
             if (sym == "ELSE") {
                    getsym ( );
                    statment ()  ;
                    }
                }
     }
    else {  var ( );
            if (sym ! = ":=")      error
( );
            else {  getsym ( );
                  expr ( );
                }
        }
}
```

<变量>::= i['['<表达式>']']

```
void  var ( )
{    if (sym != "i")  error ( ) ;
     else {  getsym ( );
             if (sym == "[")  {
                    getsym ( );
                    expr ( );
                    if (sym !="]")
                     error ( );
                    else  getsym ( );
        }
     }
}
```

<表达式>::=<项>{+<项>}

```
void  expr ( )
{    term ( );
      while (sym == "+") {
            getsym ( );
            term ( );
       }
}
```

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

```
void  term ( )
{  factor( );
     while (sym == “*”) {
              getsym ( );
              factor ( );
      }
}
```

```
void main ( )
{
     getsym ( );
      statement ( );
}
```

```
void  factor ( )
{    if (sym == “(“ )  {
              getsym ( );
              expr ( );
              if (sym != “)”)  error ( );
              else  getsym ( );
     }
     else  var ( );
}
```

```
error ( )
{
     printf(“ syntex rror !\n”)
}
```

举例分析

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

作业：

p91: 1-3

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

```
void  statement( )
{
    if (sym == “if“)  {
            getsym ( );
            expr ( );
            if (sym != “then“)     error( ) ;
            else {  getsym ( );
                  statement ( );
                if (sty == “else”)   {
                        getsym ( );
                        statment ()   ;
                    }
                }
    }
    else { var ( );
                ( );
            }
        }
}
```

printf (“ it is a statement \n”);

<变量>::= i[‘[‘<表达式>‘]’]

```
void  var ( )
{    if (sym != “i”)  error ( ) ;
     else {  getsym ( );
             if (sym == “[”)  {
                     getsym ( );
                     expr ( );
                     if (sym != “]”)
                      error ( );
                     else  getsym ( );
             }
     }
}
```

printf (“ it is a variable \n”);

<表达式>::=<项>{+<项>}

```
void e
{    term ( );
     while (sym == “+”) {
             getsym ( );
             term ( );
     }
}
```

printf (“ it is a expresson \n”);

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

```
void  term ( )
{  factor( );
     while (sym == “*”) {
            getsym ( );
            factor ( );
      }
}
```

```
void main ( )
{
     getsym ( );
      state ( );
}
```

printf (“ it is a term \n”);

```
void  factor ( )
{    if (sym == “(“ )  {
            getsym ( );
            expr ( );
            if (sym != “)”)  error ( );
            else  getsym ( );
     }
     else  var ( );
}
```

```
error ( )
{
    printf(“ syntex error !\n”)
}
```

printf (“ it is a factor \n”);

谢谢！