第四章 语法分析

1. 语法分析的功能、基本任务
2. 自顶向下分析法
3. 自底向上分析法

4.1 语法分析概述

功能：根据文法规则，从源程序单词符号串中识别出语法成分，并进行语法检查。

基本任务：识别符号串S是否为某语法成分。

两大类分析方法：

自顶向下分析

自底向上分析

自顶向下分析算法的基本思想为：

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

? 存在主要问题：
- 左递归问题
- 回溯问题

■ 主要方法：
- 递归子程序法
- LL分析法

自底向上分析算法的基本思想为：

若$Z \Leftarrow S$　则　$S \in L(G[Z])$　　否则 $S \notin L(G[Z])$

存在主要问题：

- 句柄的识别问题
- 若两个以上规则的右部都能够构成句柄时，选哪个？

主要方法：

- 算符优先分析法
- LR分析法

4.2 自顶向下分析

4.2.1 自顶向下分析的一般过程

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

若成功，则S最终被识别为某一语法成分。即

$S \in L(G[Z])$ 其中G[Z]为某语言成分的文法。

若不成功，则 $S \notin L(G[Z])$ 。

- 我们可以通过一例子来说明语法分析过程

例

2型文法！
上下文无关文法

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

分析过程是设法建立一棵语法树，使语法树的末端结点与给定符号串相匹配。

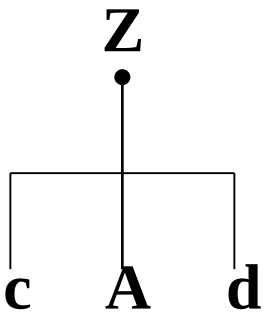
1.开始：令Z为根结点。

2.用Z的右部符号串去匹配输入串

完成一步推导 $Z \Rightarrow c\,A\,d$

检查 c - c匹配

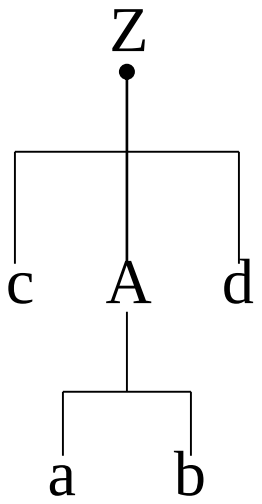
A是非终结符，将匹配任务交给A



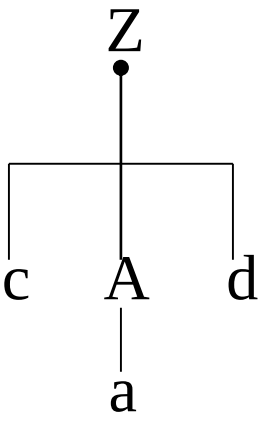
S = c a d　G[Z]: Z::= c A d
A::= a b | a

3.选用A的右部符号串匹配输入串
A有两个右部，选第一个。

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



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



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

自顶向下分析方法的特点

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

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

3. 最左推导可以编出程序来实现，但在实际上价值不大，效率低，代价高。

4.2.2 自顶向下分析存在的问题及解决方法

1、 左递归文法:

有如下文法:

令U是文法的任一非终结符，文法中有规则

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

这个规则文法是左递归的。

该方法的基本缺点是不能处理具有左递归性的文法。

自顶向下分析为什么不能处理左递归文法?

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

如果文法具有间接左递归，则也将发生上述问题，只不过兜的圈子更大。

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

消除直接左递归

方法一：使用扩充的BNF表示来改写文法

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

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

a. 改写以后的文法消除了左递归。

b. 可以证明，改写前后的文法是等价的，表现在

$L(G_{改前}) = L(G_{改后})$

如何改写文法能消除左递归，又前后等价？

现在我们可以给出两条规则。

规则一（提因子）

注意若是 y x | w x |....| z x
则x不是公因子！

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

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

总把ε安置成最后的选择！

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

规则二

若有文法规则：U::= x | y |...| z | U v

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

U ⇒ Uv ⇒ Uvv ⇒ Uvvv ⇒ ……

∴ 可以改写为U::= (x | y |...| z) { v }

通过以上两条规则，就能消除文法的直接左递归，并且保证文法的等价性。

方法二：将左递归规则改为右递归规则

规则三

若：$P ::= P\alpha \mid \beta$

则可改写为：

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

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

下面有两个例题：

若：P∷=Pα|β
则可改写为：P∷=βP’
P’∷=αP’|ε

例1　E∷=E+T|T

产生式右部无公因子，所以不能用规则一。

为了使用规则二，我们

令E∷=T|E+T

∴由规则二我们可以得到

E∷=T{+T}

右递归：

E ::= T E’

E’ ::= + T E’ | ε　　规则三

例2　T∷=T*F|T/F|F

T∷=F|T*F|T/F

T∷=F|T(*F|/F)　规则一

T∷=F{(*F|/F)}　规则二

即T∷=F{*F|/F}

右递归：

T ::= F T’

T’ ::= * F T’ | / F T’ | ε　规则三

消除一般左递归

一般左递归也可以通过改写法予以消除。

消除所有左递归的算法:

1.把G的非终结符整理成某种顺序A_1，A_2，……A_n，使得

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

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

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

……

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

3. 化简由 2 得到的文法，即去掉那些多余的规则。

例：文法G[S]为

S ::= Q c | c

Q ::= R b | b

R ::= S a | a

该文法无直接左递归，但有间接左递归

$S \Rightarrow Q\,c \Rightarrow R\,b\,c \Rightarrow S\,a\,b\,c$　　$\therefore S \overset{+}{\Rightarrow} S\,a\,b\,c$

非终结符顺序重新排列

R ::= S a | a

Q ::= R b | b

S ::= Q c | c

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

1.检查规则R是否存在直接左递归　　R::= S a | a

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

3.检查Q是否直接左递归

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

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

最后得到文法为:

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

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

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

2、 回溯问题

什么是回溯?

分析工作要部分地或全部地退回去重做叫回溯。

造成回溯的条件:

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

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

回溯带来的问题:

效率严重低下，只有在理论上的意义而无实际意义。

效率低的原因

1) 语法分析要重做

2) 语法处理工作要推倒重来

怎么样才能避免回溯?

[定义] 设文法G（不具左递归性），$U \in V_n$

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

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

为避免回溯，对文法的要求是:

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

消除回溯的途径:

1.改写文法

对具有多个右部的规则反复提取左因子

例1　U::= x V | x W　　U, V, W∈V_n,　x∈V_t

改写为U::= x (V | W)

更清楚表示

U::= x Z

Z::= V | W

注意：问题到此并没有结束，还需要进一步检查V和W的首符号是否相交

若V::= a b | c d　　FIRST(V) = { a , c }

W::= d e | f g　　FIRST(W) = { d , f }

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

则：Z::= a (b | c)

例2: 文法G[<程序>]

<程序> ::= <分程序> | <复合语句>

<分程序>::= begin<说明串>; <语句串> end

<复合语句>::= begin<语句串> end

FIRST(<分程序>) = { begin }

FIRST(<复合语句>) = { begin }

改写文法:

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

引入 <程序*>

<程序> ::= begin <程序*>

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

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

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

2. 超前扫描

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

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

例

<程序> ::= <分程序> | <复合语句>

<分程序>::= begin <说明串>; <语句串> end

<复合语句>::= begin <语句串> end

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

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

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

文法的两个条件

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

1、文法是非左递归的；

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

在上述条件下，就可以根据文法构造有效的、不带回溯的自顶向下分析器。

4.2.3 递归子程序法（递归下降分析法）

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

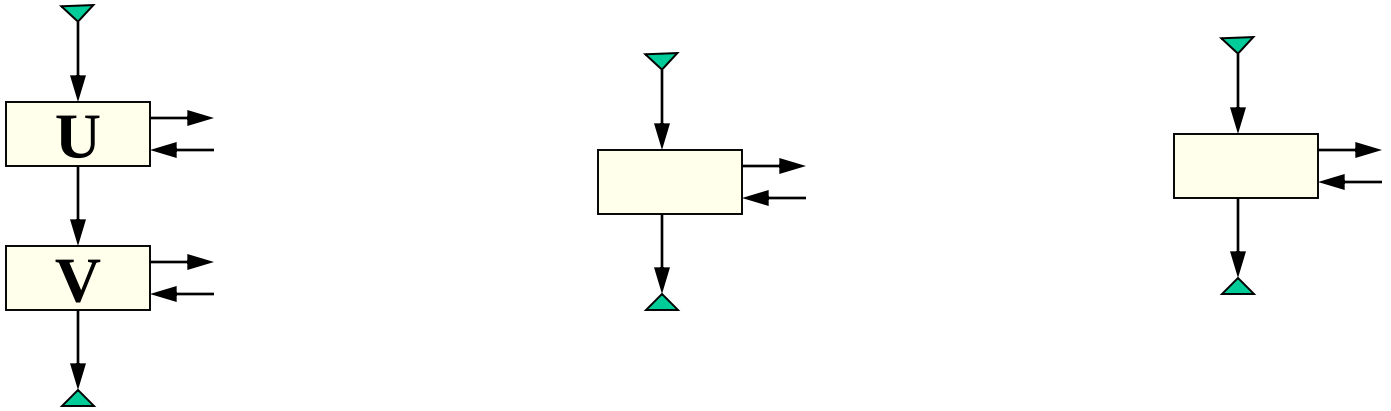
下面我们可以通过举例说明如何根据文法构造该文法的语法分析程序。

如文法G[E]: E ::= U V

U ::= ….

V ::= ….

E的分析程序 U的分析程序 V的分析程序



注：消除左递归后，可有其它递归（如右递归或自嵌入递归）：

U ::= …U… U ::= …W…

W::= …U…

例：文法G[Z]

Z::= (U) | a U b

U::= d Z | U d | e

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

1.检查并改写文法

Z::= (U) | a U b

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

改写后无左递归且首符集不相交：

{ (}∩{ a } =φ

{ d }∩{ e } =φ

2.检查文法的递归性

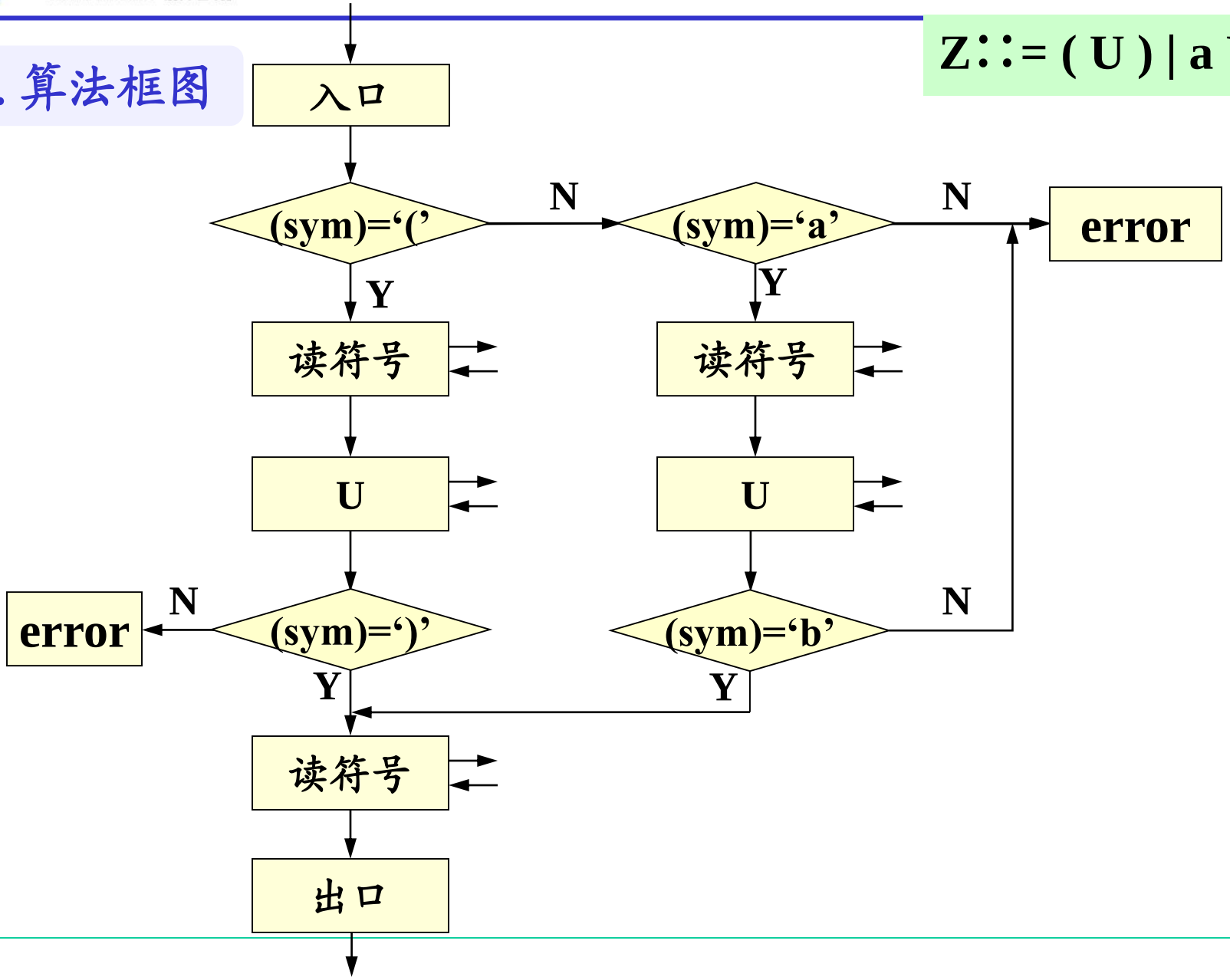
Z ⇒…U… ⇒ …Z…　　∴ Z $\overset{+}{\Rightarrow}$…Z…

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

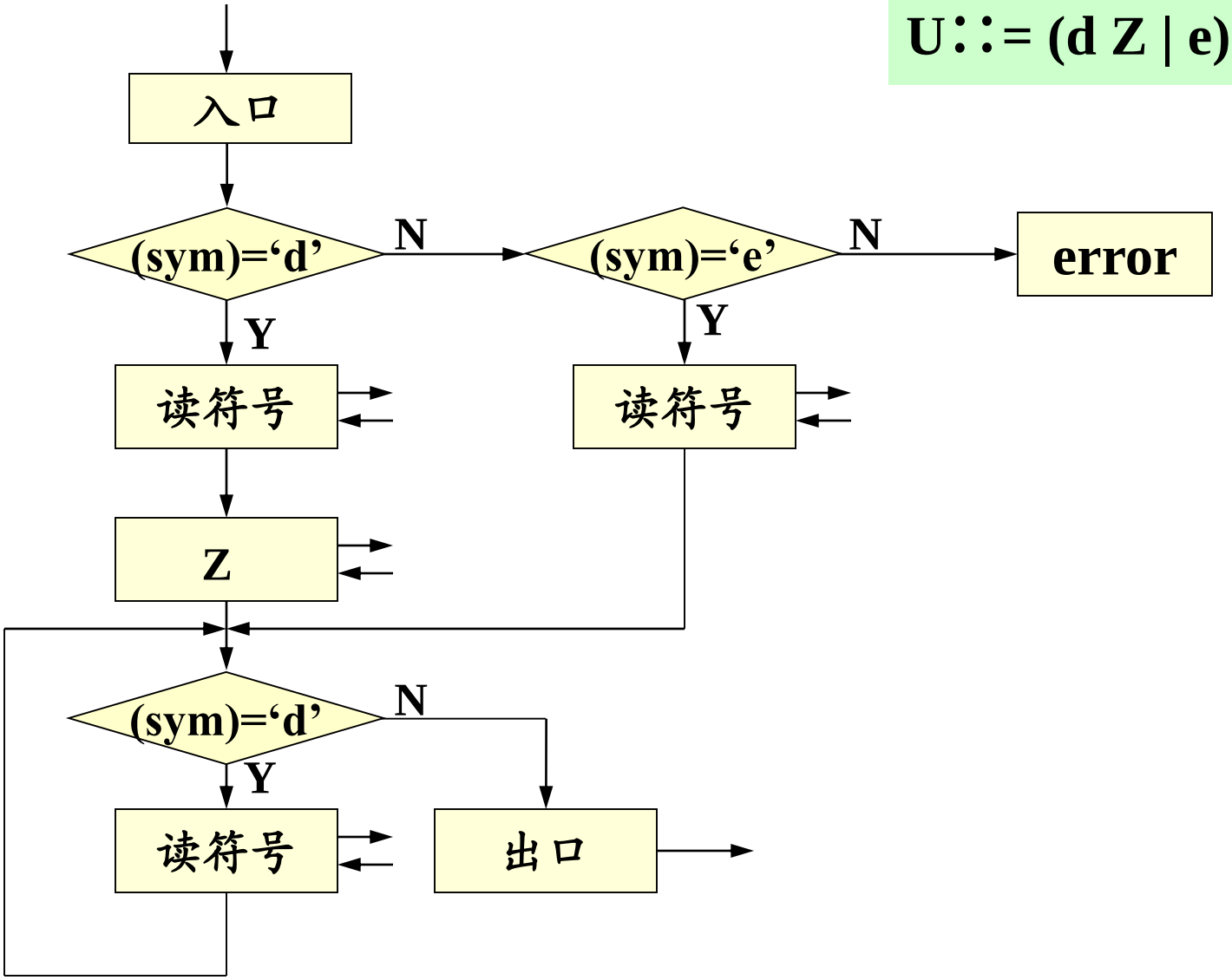
因此，Z和U的分析程序可以编成递归子程序

3. 算法框图

Z::= (U) | a U b



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



说明：

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

4.2.4 用递归子程序法构造语法分析程序的例子

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

用单引号括起来的符号表示终结符号

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

语法分析程序所要调用的子程序:

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

error: 出错处理程序。

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

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

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

<变量>::= i ['['< 表达式 >']']

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

<表达式>::= <项>{ + <项>}

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

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

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

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

举例分析

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

作业:

p85: 1

p90: 1-3

【注意：第3题第3个产生式应为B::=aB|a而不是B::=aA|a】