

第4章 自顶向下的分析

定义:从文法开始符号开始,不断利用文法规则进行推导,直到推导出所要分析的符号串为止。

$$G[E]=\{$$
$$E \rightarrow E+T \mid T,$$
$$T \rightarrow T * F \mid F,$$
$$F \rightarrow (E) \mid i$$
$$\}$$

符号串 $i+i*i$ 自顶向下分析过程:

$$\begin{aligned} E &\Rightarrow E+T \\ &\Rightarrow T+T \\ &\Rightarrow F+T \\ &\Rightarrow i+T \\ &\Rightarrow i+T * F \\ &\Rightarrow i+F * F \\ &\Rightarrow i+i * F \\ &\Rightarrow i+i * i \end{aligned}$$

4.1 带回溯的自顶向下分析思想

带回溯的自顶向下分析也称不确定的自顶向下分析

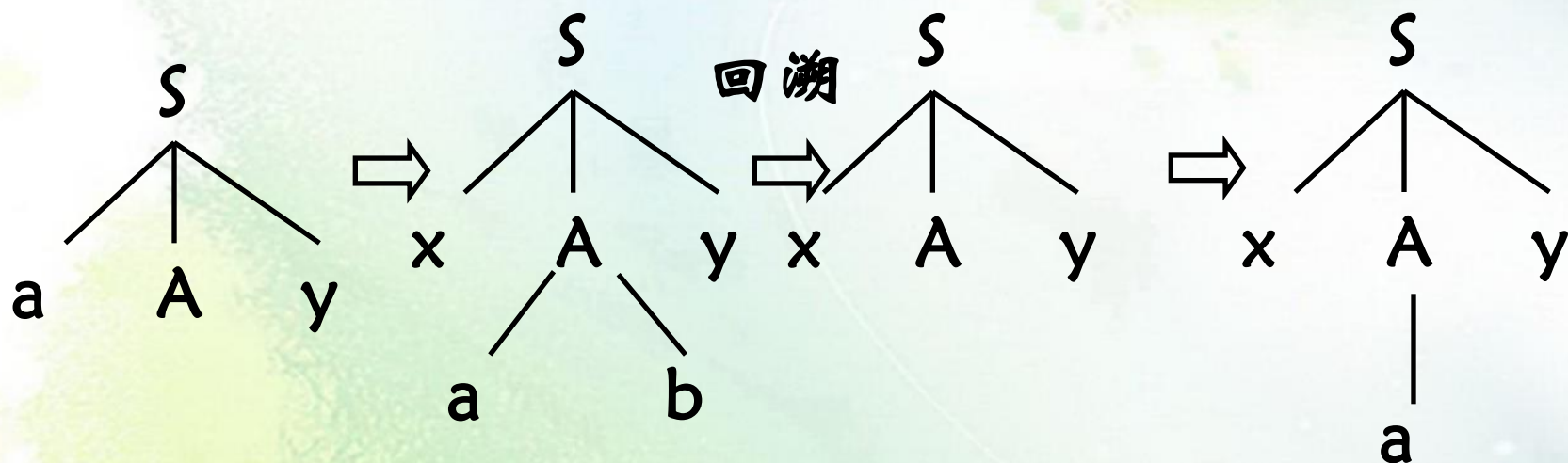
- 定义:

不确定是指某个非终结符有**多条规则**，而面临当前输入符无法唯一确定选用哪条规则进行推导，只好逐个试探。当分析不成功时，则推翻分析退回到适当位置重新试探其余候选可能的推导，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。

例1. 文法 $G[S]$:

$S \rightarrow xAy$ $A \rightarrow ab|a$,

对输入串 $w=xay$, 分析过程:



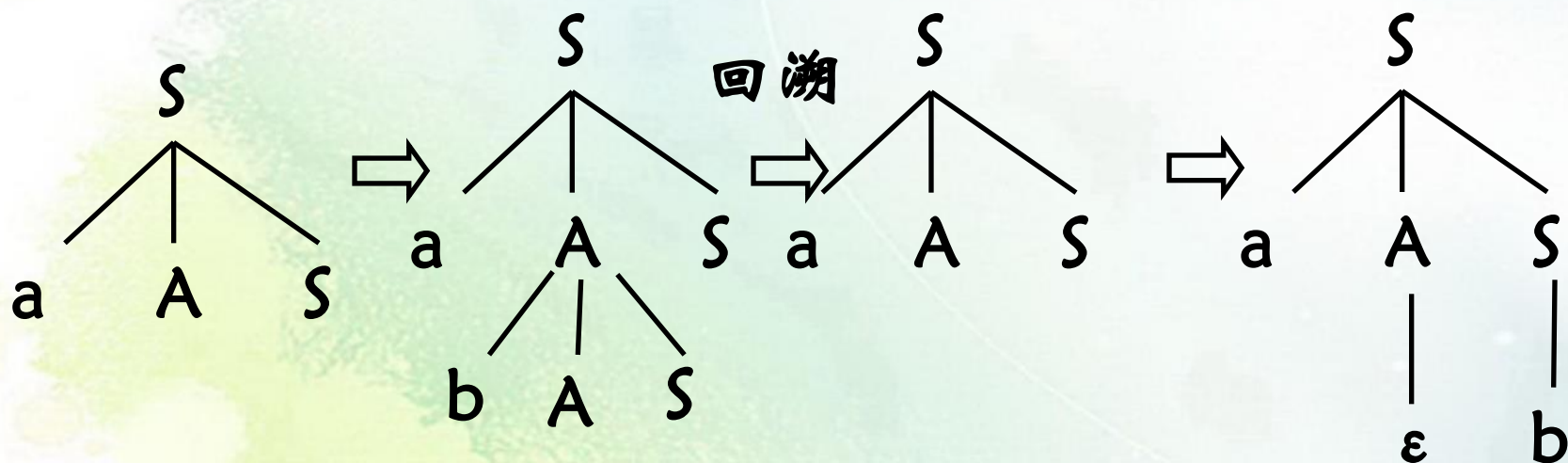
由于 A 的两条规则: $A \rightarrow ab$ 和 $A \rightarrow a$ 右部 **First 集交集不为空**, 从而引起回溯

例2. 文法 $G[S]$:

$$S \rightarrow aAS \mid b$$

$$A \rightarrow bAS \mid \varepsilon$$

输入串 $w=ab$, 分析过程:



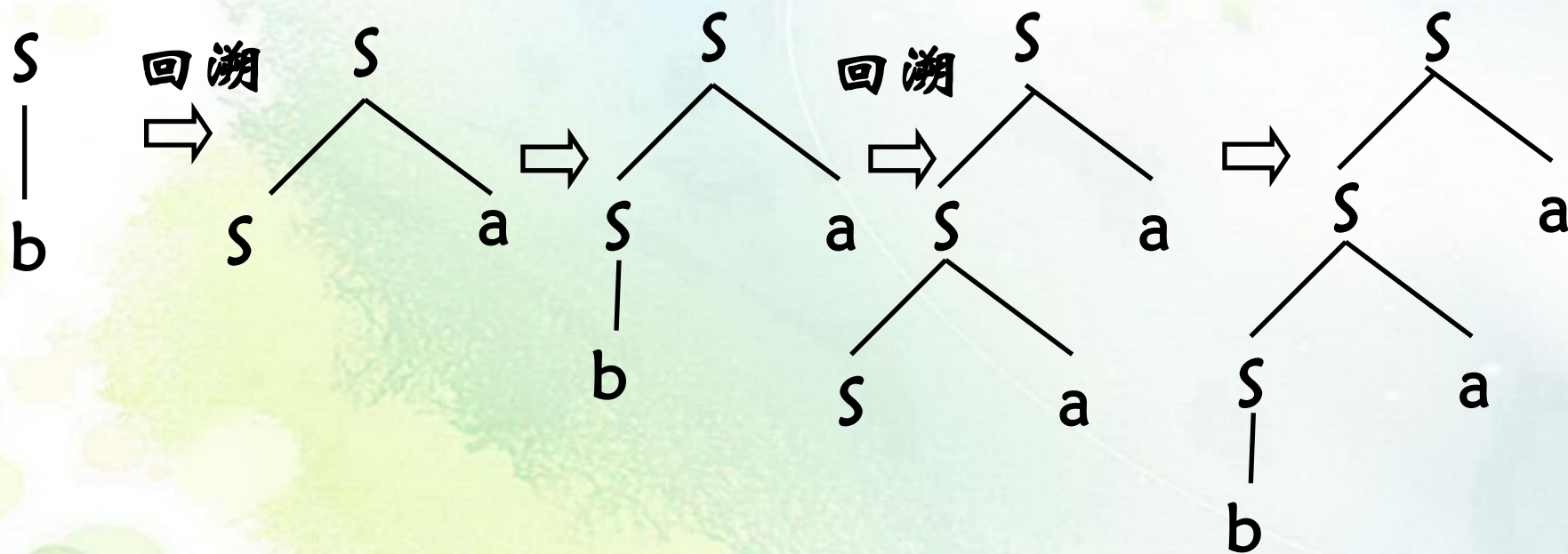
由于 A 的规则 $A \rightarrow \varepsilon$ 右部能 $\Rightarrow^* \varepsilon$, 且

$\text{Follow}(A) \cap \text{First}(bAS) = \{b\} \neq \phi$, 从而引起回溯

例3 文法G[S]:

$S \rightarrow Sa \mid b$

输入串 $w=baa$, 分析过程:



由于文法含有**左递归**而引起回溯

带回溯自顶向下分析技术主要存在下列两个问题

1) 效率问题

回溯、规则选择效率

2) 左递归问题

左递归的存在使自顶向下分析过程——死递归。

4.2 预测性的自顶向下分析方法

预测性也称确定的自顶向下分析方法

- **非终结符选择**和**规则选择**都是确定的在每一步推导中，总是对最左边的非终结符进行展开，且选择哪一个规则是确定的，因此是一种无回溯的方法。
- 具体实施策略有：
 - 递归下降分析法 (Recursive-Descent Parser)
 - LL(1) 分析法 (LL(1) Parser)

4.2 无回溯的自顶向下分析技术

应用条件

为应用无回溯的自顶向下分析技术，文法必须满足下列条件：

1) 无左递归性

文法中关于任何非终结符号 U ，都不具有规则左递归和文法左递归，即，不存在形如 $U ::= U \cdots$ 的规则，也不存在 $U \Rightarrow^+ U \cdots$ 。

2) 无回溯性

一个文法的任何 $U \in V_N$ ，存在 $U ::= u_1 | u_2 | \cdots | u_k$ ，若 $u_i \Rightarrow^* T_i \cdots$ 与 $u_j \Rightarrow^* T_j \cdots$ ， $T_i, T_j \in V_T, i \neq j$ ，就有 $T_i \neq T_j$ 。

递归下降分析法

递归下降分析法 (Recursive-Descent Parsing)

对每个非终结符按其规则结构产生相应语法分析子程序。

终结符产生匹配命令

非终结符则产生调用命令

由于文法递归相应函数也递归，所以称这种方法为**递归子程序方法**或**递归下降法**。

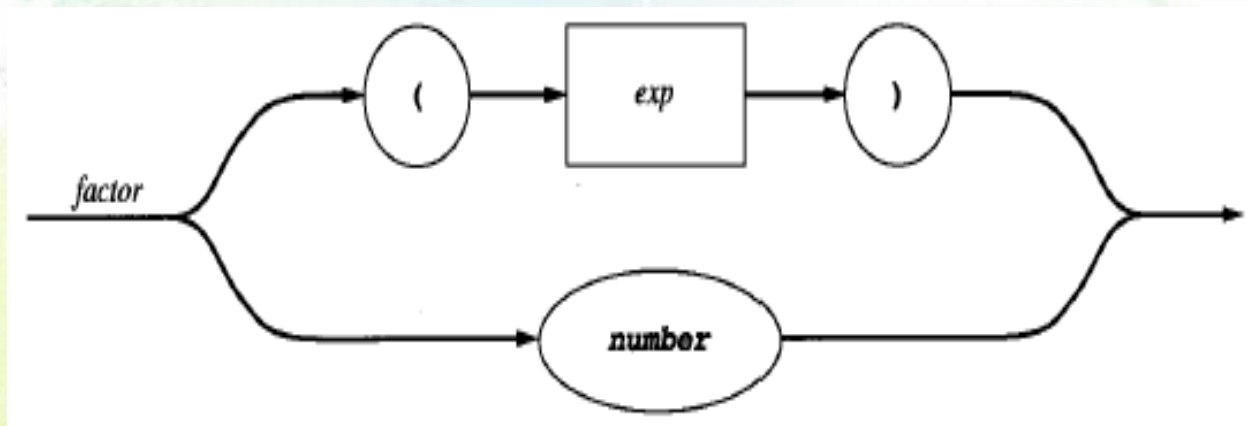
语法图

- 语法规则的直观表达
- 递归下降语法分析程序处理逻辑的直观表达
- 每一个非终结符的文法规则定义一个语法图
- **箭头**：表示序列和选择的
- **终结符**：圆形框和椭圆形框
- **非终结符**：方形框和矩形框

例如，文法规则

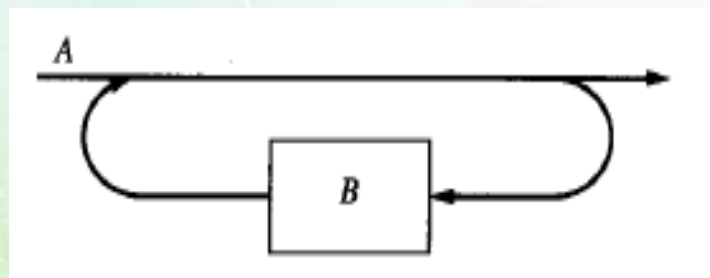
$factor \rightarrow (exp) \mid number$

用语法图表示则是：



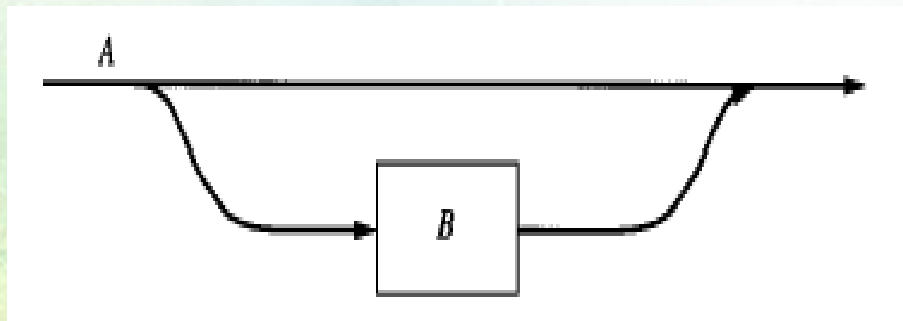
重复结构的画法

EBNF: $A \rightarrow \{ B \}$



- 可选结构的画法

EBNF: $A \rightarrow [B]$



例3.10 画出简单算术表达式的语法图

$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$

$\text{addop} \rightarrow + \mid -$

$\text{term} \rightarrow \text{term mulop factor} \mid \text{factor}$

$\text{mulop} \rightarrow *$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

解答 (1) 由于语法图是基于EBNF而画，因此先将文法规则转换为EBNF。

EBNF 表示:

$\text{exp} \rightarrow \text{term} \{ \text{addop term} \}$

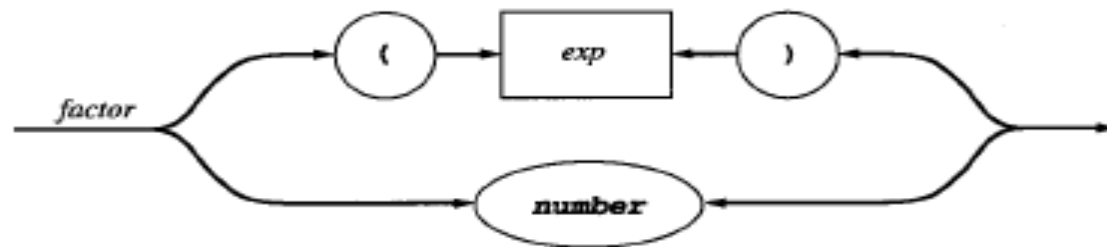
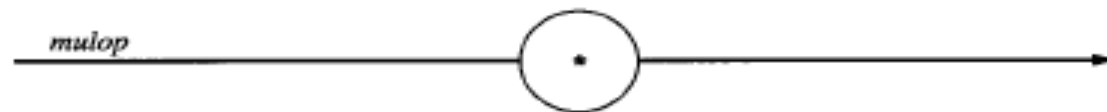
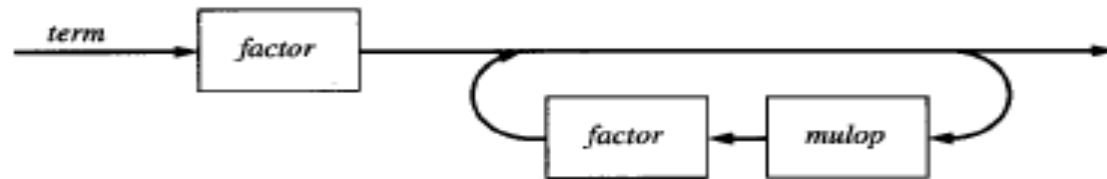
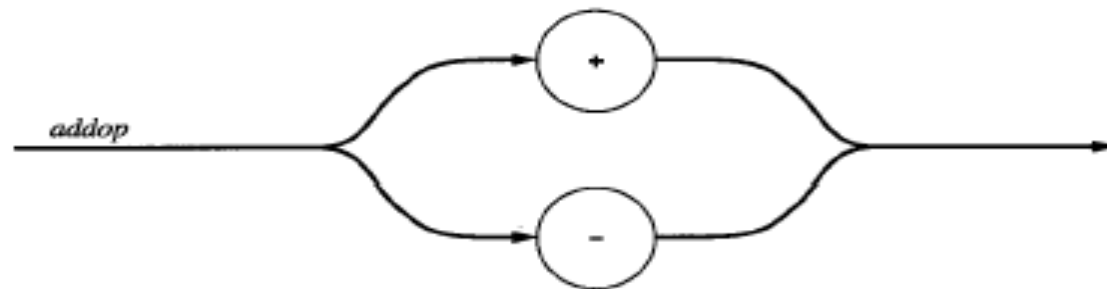
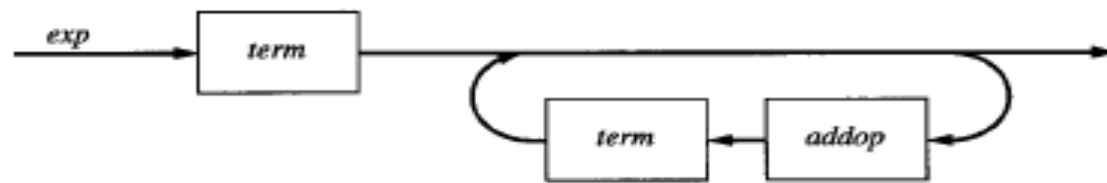
$\text{addop} \rightarrow + \mid -$

$\text{term} \rightarrow \text{factor} \{ \text{mulop factor} \}$

$\text{mulop} \rightarrow *$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

(2) 画出每一个非终结符号的语法图



例3.11 画出语法图

$\text{statement} \rightarrow \text{if-stmt} \mid \text{other}$

$\text{if-stmt} \rightarrow \text{if (exp) statement} \mid \text{if (exp)}$
 $\text{statement else statement}$

$\text{exp} \rightarrow 0 \mid 1$

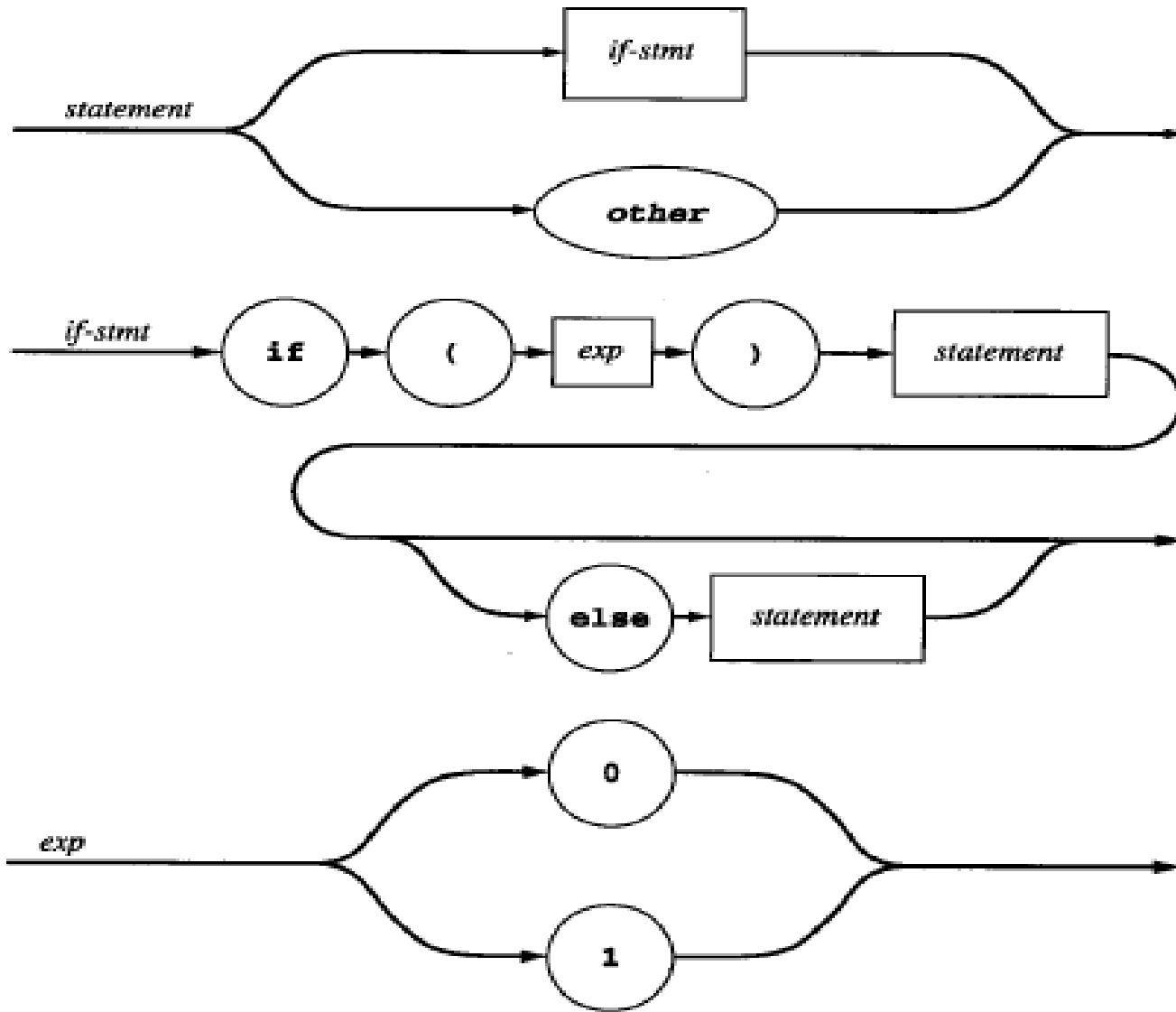
(1) 改写为EBNF:

$\text{statement} \rightarrow \text{if-stmt} \mid \text{other}$

$\text{if-stmt} \rightarrow \text{if}(\text{exp}) \text{ statement} [\text{else statement}]$

$\text{exp} \rightarrow 0 \mid 1$

(2) 画出每一个非终结符号的语法图



递归下降分析程序的设计

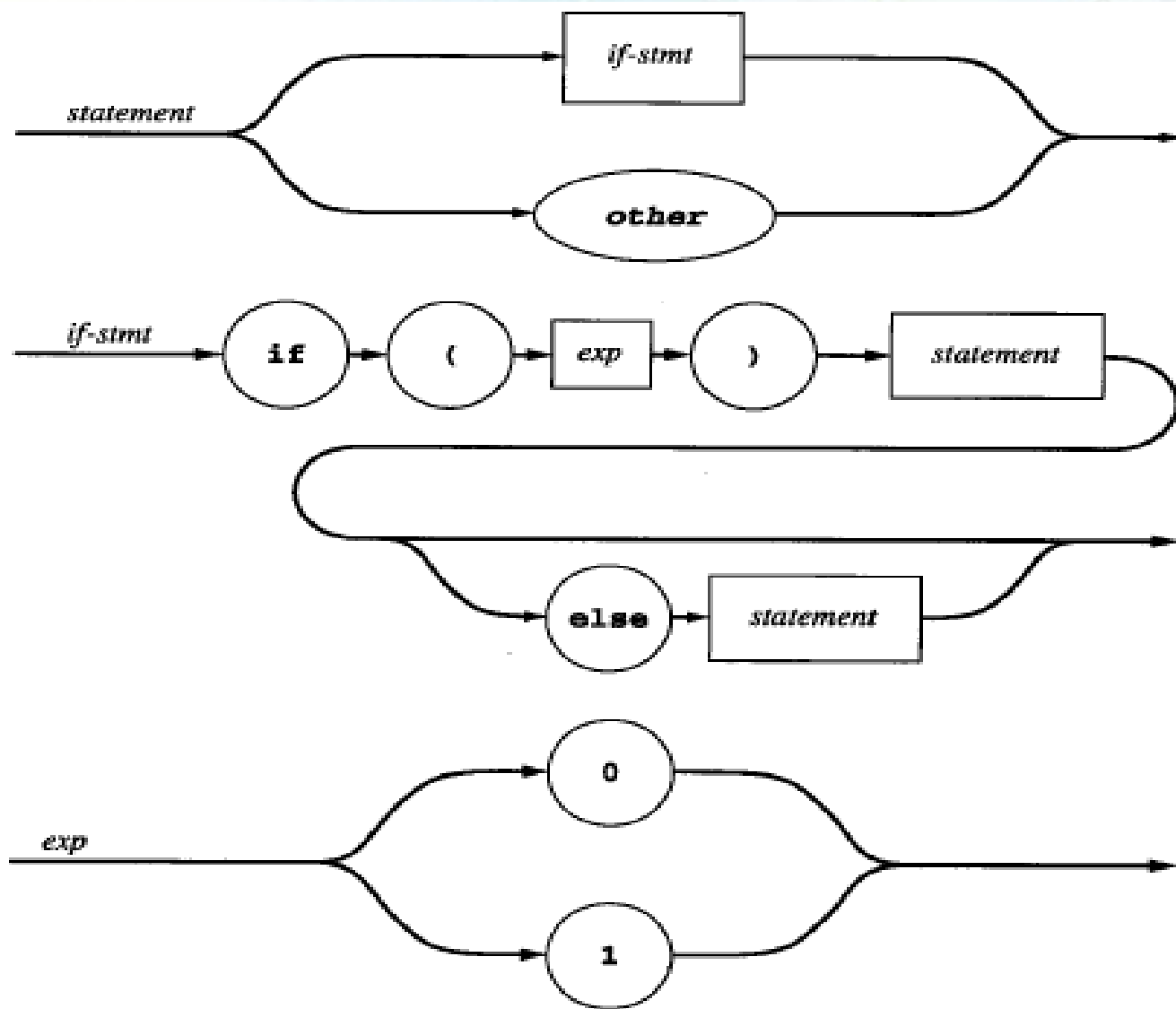
递归下降分析程序的设计方法：**根据语法图的处理逻辑进行描述。**

每个非终结符都对应一个函数。

该函数根据语法的描述来明确：根据下一个输入符号来确定按照哪一个规则进行处理，再根据该规则的右端，

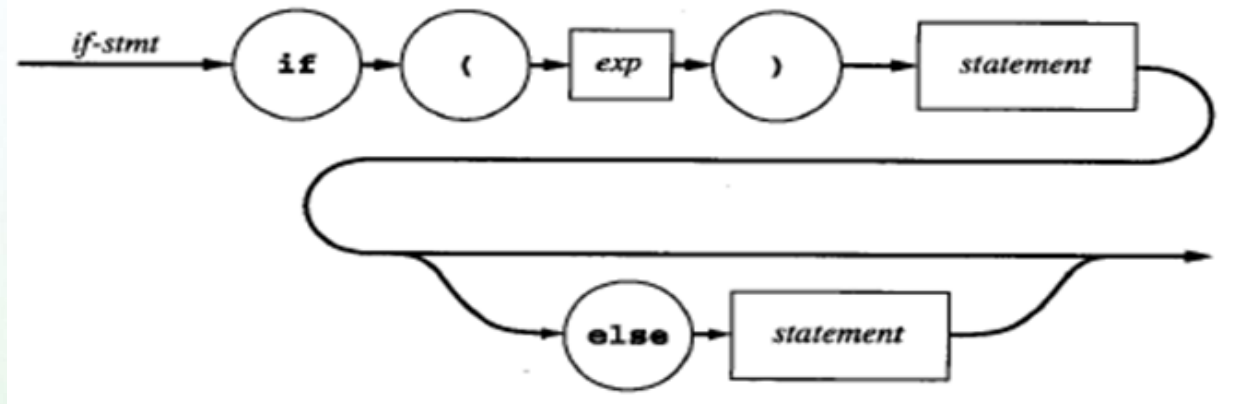
- 每遇到一个终结符，则判断当前读入的单词是否与该终结符相匹配，若匹配，再读取下一个单词继续分析；不匹配，则进行出错处理
- 每遇到一个非终结符，则调用相应的函数

1. 根据语法图来构造递归下降分析程序



非终结符号if_Stmt分析程序：

```
void ifStmt()  
{  
    match('if');  
    match('(');  
    exp();  
    match(')');  
    statement();  
    if (TOKEN == 'else') {  
        match('else');  
        statement();  
    }  
} // ifStmt
```



2. 根据文法直接构造递归下降分析程序

$U \rightarrow x_1 \mid x_2 \mid \cdots \mid x_n$, 其中 x_1, \dots, x_n 均 $\neq \varepsilon$

$U \rightarrow x_1 \mid x_2 \mid \cdots \mid x_n \mid \varepsilon$

规则选择问题：通过分别求出first集合和follow集合来解决。

非终结符相应的分析子程序的构造方法

(1) 对于每个非终结符 U ，编写一个相应的子程序 $P(U)$;

(2) 对于规则 $U \rightarrow x_1 \mid x_2 \mid \cdots \mid x_n$, x_1, \dots, x_n 都 $\neq \varepsilon$ 关于 U 的函数 $P(U)$ 按如下方法构造:

```
if (TOKEN in first( $x_1$ ) ) p( $x_1$ );
```

```
else if (TOKEN in first( $x_2$ ) ) p( $x_2$ );
```

```
else ...
```

```
...
```

```
if (TOKEN in first( $x_n$ ) ) p( $x_n$ );
```

```
else ERROR();
```

(3) 如果U还有空规则 $U \rightarrow \varepsilon$, 则算法中的语句:

if (TOKEN in first(x_n)) p(x_n) ; else ERROR();

改写为

if (TOKEN in first(x_n)) p(x_n) ;

else if (TOKEN not in follow(U)) ERROR();

(4) 对于符号串 $x = y_1 y_2 \cdots y_n$; p(x) 的含义为:

{ p(y_1); p(y_2); \cdots ; p(y_n); }

如果 $y_i \in V_N$, 则 P(y_i) 就代表调用 y_i 的函数; $y_i \in V_T$, 则为 P(y_i) 设计下述的一段代码或设计一个 match 函数。

if (TOKEN == y_i) getToken(TOKEN) else ERROR();

注意事项

递归下降分析器由一个主程序main和每个非终结符对应的递归函数组成。

用到的一些函数：

- 函数getToken()负责读入下一个TOKEN单词
- 函数ERROR()负责报告语法错误
- 函数match()终结符号的匹配处理

约定：

- 全局变量TOKEN存放已读入的TOKEN单词

注意： TOKEN也可以安排为函数引用参数变量，

函数进入时变量TOKEN存放了一个待匹配的TOKEN字

退出函数时，变量TOKEN中仍存放着一个待匹配的TOKEN字。

实例分析

问题0:

$G[S]=$

{

$S \rightarrow aAb$

$A \rightarrow cB$

$B \rightarrow e$

}

试编写一个能分析该文法所对应任何串(如串aceb)的程序。

算法构造过程：

(1) 每一个非终结符号就对应一个分析函数

(2) 对于非终结符号 S ，其对应的分析函数为：

```
void S() //规则  $S \rightarrow aAb$   
{  
    match('a'); //匹配函数  
    A();  
    match('b'); //匹配函数  
} //S
```


void A() //规则 $A \rightarrow cB$

{

match('c'); //匹配函数

B();

} //A

```
void B( ) //规则  $B \rightarrow e$ 
```

```
{
```

```
    match('e'); //匹配函数
```

```
} //B
```

match 函数

- 功能：用来匹配当前符号是否为预想的符号，如果是则前移，否则就产生出错

```
void match ( expectedToken )  
{ if (TOKEN==expectedToken)  
    getToken(); // 获取下一个符号  
  else  
    ERROR(); // 产生出错信息  
} // match
```

实例分析

问题1:

$G[S]=$

{

$S \rightarrow aA \mid bB$

$A \rightarrow cdA \mid d$

$B \rightarrow efB \mid f$

}

试编写一个能分析该文法所对应任何串(如串acdd)的程序。

fcdd、agdd、bc

算法构造过程：

(1) 每一个非终结符号就对应一个分析函数

(2) 对于非终结符号 S ，其对应的分析函数为：

```
void S() //规则  $S \rightarrow aA | bB$ 
{
    if (TOKEN == 'a') // First(aA)
    {
        match('a'); //匹配函数
        A();
    }
    else if (TOKEN == 'b') // First(bB)
    {
        match('b'); //匹配函数
        B();
    }
    else ERROR(); // 出错处理函数
} //S
```


(3) 对于 **非终结符号A**，其对应的分析函数为：

```
void A( ) //规则:  $A \rightarrow cdA \mid d$ 
{
    if (TOKEN=='c') // First(cdA)
    {
        match('c'); //匹配函数
        match('d'); //匹配函数
        A();
    }
    else if (TOKEN=='d') // First(d)
    {
        match('d'); //匹配函数
    }
    else ERROR(); // 出错处理函数
} //A
```

(4) 对于 **非终结符号B**，其对应的分析函数为：

```
void B( ) //规则:  $B \rightarrow efB \mid f$ 
{
    if (TOKEN == 'e') // First( $efB$ )
    {
        match('e'); // 匹配函数
        match('f'); // 匹配函数
        B();
    }
    else if (TOKEN == 'f') // First( $f$ )
    {
        match('f'); // 匹配函数
    }
    else ERROR(); // 出错处理函数
} //B
```

Main函数的安排

- (1) 安排TOKEN 的全局变量
- (2) 读入输入串的第一符号到TOKEN 中
- (3) 调用文法开始符号所对应函数。

```
void main()  
{  
    getToken();  
    s();  
} //main
```

match 函数

- 功能：用来匹配当前符号是否为预想的符号，如果是则前移，否则就产生出错

```
void match ( expectedToken )  
{ if (TOKEN==expectedToken)  
    getToken(); // 获取下一个符号  
  else  
    ERROR(); // 产生出错信息  
} // match
```


ERROR 函数

- 方法一：负责产生提示信息及跳到下一个符号进行分析

```
void ERROR ( )  
{  
    cout<<"ERROR Message!";  
    getToken();  
} //ERROR
```

缺陷：

- (1) 提示信息太抽象，不准确
- (2) 直接跳过当前符号，会导致更多的出错

ERROR 函数

- 缺陷一的解决方案：增加编号来引导准确的出信息

```
void ERROR ( int ErrorNO )
```

```
{
```

```
    switch(ErrorNO)
```

```
{
```

```
    case 1: cout<<“漏了一个a或b” ;break;
```

```
    case 2: cout<<“漏了一个c或d”;break;
```

```
    case 3: cout<<“漏了一个e或f”;break;
```

```
    ..... 
```

```
    }// switch
```

```
} //ERROR
```

实例分析

写出下面文法的递归子程序：

$\text{if-stmt} \rightarrow \text{if (exp) statement} \mid$
 $\text{if (exp) statement else statement}$

- if语句规则改写为:

$\text{if-stmt} \rightarrow \text{if}(\text{exp})\text{statement}(\epsilon \mid \text{else statement})$

分析程序:

```
void ifStmt()  
{  
    match ('if') ;  
    match ('(') ;  
    exp() ;  
    match (')') ;  
    statement() ;  
    if (TOKEN == 'else') {  
        match ('else') ;  
        statement() ;  
    }  
} // ifStmt ;
```

解决思路2：直接利用EBNF []

- if语句规则的EBNF为：

if-stmt \rightarrow if(exp)statement [else statement]

分析程序：

```
void ifStmt()  
{  
    match ('if') ;  
    match ('(') ;  
    exp();  
    match (')') ;  
    statement();  
    if (TOKEN == 'else') {  
        match ('else') ;  
        statement();  
    }  
} // ifStmt
```


实例分析

文法：

$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$

$\text{addop} \rightarrow + \mid -$

$\text{term} \rightarrow \text{term mulop factor} \mid \text{factor}$

$\text{mulop} \rightarrow * \mid /$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

请写出递归子程序分析算法，并分析表达式
 $3+4*5$ 。

规则：factor \rightarrow (exp) | number 的递归子程序为

```
void factor()  
{  
    switch( TOKEN )  
    {  
        case '(':  
            match('(') ;  
            exp() ;  
            match(')') ;  
            break;  
        case number :  
            match (number) ;  
            break;  
        default:  
            ERROR() ;  
    } // switch  
} // factor
```

- 考虑非终结符号exp
- $\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$
- 问题：出现了左递归
- 解决办法：使用EBNF规则,消除原规则中的左递归。

$\text{exp} \rightarrow \text{term} \{ \text{addop term} \}$

新问题：无法将 $\{ \}$ 实现循环

再进一步改写：

$\text{exp} \rightarrow \text{term} \{ (+ \mid -) \text{term} \}$

规则： $\text{exp} \rightarrow \text{term} \{ (+ | -) \text{term} \}$

```
void exp()  
{  
    term();  
    while ((TOKEN == '+' ) || (TOKEN == '-' ))  
    {  
        match (TOKEN) ;  
        term();  
    } // while  
} // exp
```

相似地，term的EBNF规则：

$\text{term} \rightarrow \text{factor} \{ (* | /) \text{factor} \}$ 则其代码为：

```
void term()  
{  
    factor();  
    while ((TOKEN == '*' ) || (TOKEN == '/'))  
    {  
        match (TOKEN);  
        factor();  
    } //while  
} // term
```


main函数的安排

```
main()
{
    getToken();
    exp();
}
```

如： $3+4*5$

• **新问题**：如何实现算术表达式值的计算？

• **方法**：

- 1.使用栈的方法（数据结构课本中的算法）
- 2.使用文法规则加递归子程序分析算法的方法

• **新方法**：

- (1)根据算术表达式的组成特点写出文法规则
- (2)写出该文法规则的递归子程序**分析算法**
- (3)为了使其拥有**计算功能**，为每个递归函数加上返回当前计算结果

新方法

步骤1: 先设计文法:

$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

步骤2：文法问题的分析→文法的改造

$\text{exp} \rightarrow \text{term} \{ (+|-) \text{term} \}$

$\text{term} \rightarrow \text{factor} \{ (*|/) \text{factor} \}$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

步骤3：对每个非终结符写出相应的递归子程序

规则： $\text{exp} \rightarrow \text{term} \{ (+|-) \text{term} \}$

```
void exp()  
{  
    term();  
    while ((TOKEN == '+') || (TOKEN == '-'))  
    {  
        match (TOKEN);  
        term();  
    } // while  
} // exp
```

$\text{term} \rightarrow \text{factor} \{ (* | /) \text{factor} \}$

分析程序为：

```
void term()
```

```
{
```

```
    factor();
```

```
    while ((TOKEN == '*' ) || (TOKEN == '/' ))
```

```
    {
```

```
        match (TOKEN);
```

```
        factor();
```

```
    } //while
```

```
} // term
```

规则：factor \rightarrow (exp) | number 的递归子程序为

```
void factor()  
{  
    switch( TOKEN )  
    {  
        case '(':  
            match('(') ;  
            exp() ;  
            match(')') ;  
            break;  
        case number :  
            match (number) ;  
            break;  
        default:  
            ERROR() ;  
    } // switch  
} // factor
```

- 步骤4：如何在递归下降分析程序中增加计算功能

- 遇到运算对象则返回
- 遇到运算符号就将对应运算对象进行**计算**
- 遇到括号就返回括号中表达式的计算结果
- 因此
 - (1) 需要为每个递归函数加上返回当前计算结果
 - (2) 每次计算均要保存计算结果


```
int exp()  
{ int temp;  
  temp = term();  
  while ((TOKEN == '+' ) || (TOKEN == '-'))  
  { switch (TOKEN)  
    {  
      case '+' : match ('+') ;  
                  temp = temp + term();  
                  break;  
      case '-' : match ('-') ;  
                  temp = temp - term();  
                  break;  
    } //switch  
  } while  
  return temp ;  
} //exp
```

```
int term()  
{ int temp;  
  temp = factor();  
  while ((TOKEN == '*') || (TOKEN == '/'))  
  { switch (TOKEN)  
    {  
      case '*': match('*') ;  
                temp = temp * factor();  
                break;  
      case '/': match('/') ;  
                temp = temp / factor();  
                break;  
    } //switch  
  } while  
  return temp ;  
} //term
```

```
int factor()  
{  
    switch(TOKEN )  
    {  
        case '(':  
            match('(') ;  
            temp=exp();  
            match(')') ;  
            break;  
        case number :  
            match (number) ;  
            temp=number;  
            break;  
        default:  
            ERROR() ;  
    } // switch  
    return temp;  
} // factor
```

main函数的安排

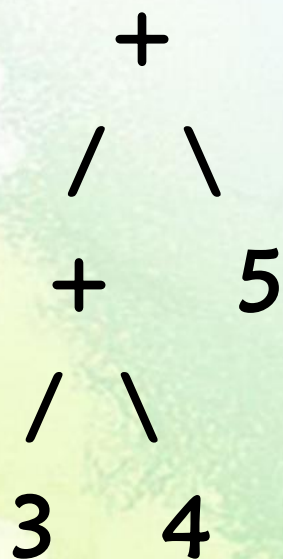
```
main()
{
    getToken();
    cout<<exp();
}
```

如： $3+4*5$

- 详细代码请看云盘电子参考书程序清单4-1 简单整型算术的递归下降程序计算器(请自行阅读)
- 其中并未写出一个完整的扫描程序，而是选择使用了对getchar和scanf的调用来代替getToken的函数。

- **问题：**算术表达式对应语法树的构造？

- **例：**表达式 $3 + 4 + 5$ ，其语法树为：



- **例：**表达式 $3 + 4 * 5$ ，其语法树则为

解决方法：

- (1) 写出文法规则
- (2) 写出递归下降分析程序
- (3) 在递归下降分析程序中增加语法树生成功能
 - 遇到运算对象生成叶子结点并返回
 - 遇到运算符就将对应运算对象进行新树根的构造
 - 遇到括号就返回括号中表达式对应语法树返回
 - 因此，需要为每个递归函数加上返回当前所生成的语法树树根指针

新方法

步骤1: 先设计文法:

$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

步骤2: 文法问题的分析 → 文法的改造

$\text{exp} \rightarrow \text{term} \{ (+|-) \text{term} \}$

$\text{term} \rightarrow \text{factor} \{ (*|/) \text{factor} \}$

$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$

步骤3：对每个非终结符写出相应的递归子程序

规则： $\text{exp} \rightarrow \text{term} \{ (+|-) \text{term} \}$

```
void exp()  
{  
    term();  
    while ((TOKEN == '+') || (TOKEN == '-'))  
    {  
        match (TOKEN);  
        term();  
    } // while  
} // exp
```

$\text{term} \rightarrow \text{factor} \{ (* | /) \text{factor} \}$

分析程序为：

```
void term()
```

```
{
```

```
    factor();
```

```
    while ((TOKEN == '*' ) || (TOKEN == '/' ))
```

```
    {
```

```
        match (TOKEN);
```

```
        factor();
```

```
    } //while
```

```
} // term
```

规则：factor \rightarrow (exp) | number 的递归子程序为

```
void factor()  
{  
    switch( TOKEN )  
    {  
        case '(':  
            match('(') ;  
            exp() ;  
            match(')') ;  
            break;  
        case number :  
            match (number) ;  
            break;  
        default:  
            ERROR() ;  
    } // switch  
} // factor
```


步骤4：修改分析程序，让其具有相应的新功能。

——语法树的生成

遇到+或-运算符时该如何生成语法树

```
void exp()  
{  
    term();  
    while ((TOKEN == '+' ) || (TOKEN == '-' ))  
    {  
        match (TOKEN) ;  
        term();  
    } // while  
} // exp
```

```

BTreeNode * exp()
{
    BTreeNode * temp, *newtemp;
    temp = term();
    while ( TOKEN == '+' || TOKEN == '-' )
    {
        switch ( TOKEN )
        {
            case '+': match ('+');
                    newtemp = new BTreeNode;
                    newtemp->data = '+';
                    newtemp->lchild = temp;
                    newtemp->rchild = term();
                    temp = newtemp;

            case '-': match ('-');
                    newtemp = new BTreeNode;
                    newtemp->data = '-';
                    newtemp->lchild = temp;
                    newtemp->rchild = term();
                    temp = newtemp;

        }
    }
    return temp;
} // exp

```

进一步合并代码...

```
BTreeNode * exp()  
{  
    BTreeNode * temp, *newtemp;  
    temp = term() ;  
    while ( TOKEN == '+' || TOKEN == '-' )  
    {  
        newtemp = new BTreeNode ;  
        newtemp->data= TOKEN ;  
        match (TOKEN) ;  
        newtemp->lchild = temp ;  
        newtemp->rchild= term() ;  
        temp = newtemp ;  
    }  
    return temp ;  
} // exp
```


遇到*或/运算符时该如何生成语法树

```
void term()  
{  
    factor();  
    while ((TOKEN == '*' ) || (TOKEN == '/'))  
    {  
        match (TOKEN);  
        factor();  
    } //while  
} // term
```

```
BTreeNode * term()  
{  
    BTreeNode * temp, *newtemp;  
    temp = factor();  
    while ( TOKEN == '*' || TOKEN == '/' )  
    {  
        newtemp = new BTreeNode ;  
        newtemp->data= TOKEN ;  
        match (TOKEN) ;  
        newtemp->lchild = temp ;  
        newtemp->rchild= factor() ;  
        temp = newtemp ;  
    }  
    return temp ;  
} // term
```

遇到()或number时该如何生成语法树

```
void factor()  
{  
    switch( TOKEN )  
    {  
        case '(':  
            match('(');  
            exp();  
            match(')');  
            break;  
        case number :  
            match (number);  
            break;  
        default:  
            ERROR();  
    } // switch  
} // factor
```

```
BTreeNode * factor()  
{  
    BTreeNode *temp;  
    switch( TOKEN )  
    {  
        case '^':  
            match('^') ;  
            temp= exp();  
            match(')') ;  
            break;  
        case number :  
            match (number) ;  
            temp = new BTreeNode ;  
            temp ->data= number ;  
            temp->lchild = NULL ;  
            temp->rchild=NULL;  
            break;  
        default:  
            ERROR() ;  
    } // switch  
    return temp;  
} // factor
```


main函数的安排

```
main()  
{  
    BTreeNode *root;  
  
    getToken();  
    root=exp();  
}
```

如： $3+4*5$

一个新问题...

编写一个能生成简单算术表示式对应汇编代码。

例如: $3+4*5$

Ldc 3

Ldc 4

Ldc 5

Mpi

Adi

与栈打交道的汇编语言

所有操作都依赖与栈来完成

汇编指令的介绍

Ldc n 把常数n压入栈

Mpi 取出栈顶与次栈顶元素做乘法运算，结果入栈

Adi 取出栈顶与次栈顶元素做加法运算，结果入栈

Sbi 取出栈顶与次栈顶元素做减法运算，结果入栈

Dvi 取出栈顶与次栈顶元素做除法运算，结果入栈

解决方法：

- (1) 写出文法规则
- (2) 写出递归下降分析程序
- (3) 在递归下降分析程序中增加汇编代码生成功能
 - 遇到运算对象则生成Ldc指令
 - 遇到运算符就生成相应运算的指令
 - 遇到括号则不做指令生成处理


```
void exp()  
{  
    term();  
    while ((token == '+' ) || (token == '-'))  
    {  
        switch (token )  
        {  
            case '+' : match (token) ;  
                        term();  
                        Gen(Adi)  
                        break;  
            case '-': match (token) ;  
                        term();  
                        Gen(Sbi)  
                        break;  
        } //switch  
    } while  
} //exp
```

```
void term()  
{  
    factor();  
    while ((token == '*') || (token == '/'))  
    {  
        switch (token )  
        {  
            case '* ': match (token) ;  
                        factor();  
                        Gen(Mpi);  
                        break;  
            case '/': match (token) ;  
                      factor();  
                      Gen(Dvi);  
                      break;  
        } //switch  
    } while  
} //term
```

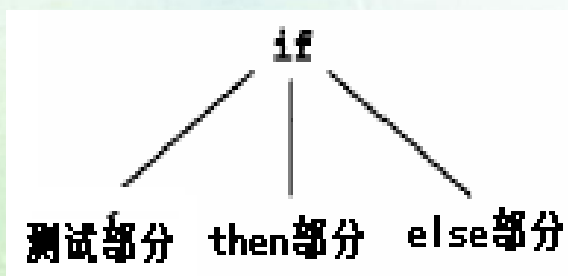
```
void factor()  
{  
    switch( token )  
    {  
        case '(':  
            match('(');  
            exp();  
            match(')');  
            break;  
        case n :  
            match (n);  
            Gen( Ldc, n )  
            break;  
        default:  
            error();  
    } // switch  
} // factor
```

main函数的安排

```
main()
{
    getToken();
    Exp();
}
```

如： $3+4*5$

问题：如何为if语句构造相应的语法树？



(1)设计文法:

$\text{if-stmt} \rightarrow \text{if (exp) statement} \mid$
 $\text{if (exp) statement else statement}$

(2) 文法改造

$\text{if-stmt} \rightarrow \text{if(exp)statement[else statement]}$

(3) 写出递归分析程序

- if语句的EBNF规则:

$\text{if-stmt} \rightarrow \text{if}(\text{exp})\text{statement}[\text{else statement}]$

分析程序:

```
void ifStmt()  
{  
    match ('if') ;  
    match ('(') ;  
    exp();  
    match (')') ;  
    statement();  
    if (TOKEN == 'else') {  
        match ('else') ;  
        statement();  
    }  
} // ifStmt ;
```

(4) 语法树生成

按照递归子程序在严格的自顶向下风格构造出if语句的语法树：

```
syntaxTree ifStatement()  
{ syntaxTree temp;  
  match ("if");  
  match '(';  
  temp = makeStmtNode(if); //生成新结点  
  temp->testChild = exp();  
  match (')');  
  temp->thenChild = statement();  
  if (TOKEN == "else") {  
    match ("else");  
    temp->elseChild = statement();  
  } else  
    temp->elseChild = NULL;  
} // ifStatement;
```


递归下降法的总结

- (1) 递归下降分析功能非常强大，比较适合于手工方法；
- (2) 分析效率低，仅适合于小型语言语法分析程序的构造。
- (3) 为了解决一些问题，通常先将BNF文法规则改写为EBNF规则。
- (4) 在生成语法树时，要注意二义性问题的分析。

问题求解之新方法

问题:

$G[S]=\{$

$S \rightarrow AB \mid CD$

$A \rightarrow aB \mid dD$

$B \rightarrow cC \mid bD$

$C \rightarrow ef \mid gh$

$D \rightarrow i \mid j$

$\}$

试编写出能分析该文法所对应任何串（如串acefbgh）
的程序。

```
void S( ) //规则:  $S \rightarrow AB|CD$ ,  
{  
    if (TOKEN in {'a','d'}) // First(AB)  
    {  
        A();  
        B();  
    }  
    else if (TOKEN in {'e','g'}) // First(CD)  
    {  
        C();  
        D();  
    }  
    else ERROR(); // 出错处理函数  
} //S
```

```
void A( ) //规则:  $A \rightarrow aB \mid dD$ ,  
{  
    if (TOKEN == 'a') // First(aB)  
    {  
        match('a');  
        B();  
    }  
    else if (TOKEN == 'd') // First(dD)  
    {  
        match('d');  
        D();  
    }  
    else ERROR(); // 出错处理函数  
} //A
```



```
void B( ) //规则：  $B \rightarrow cC | bD$ ,  
{  
    if (TOKEN == 'c') //First(cC)  
    {  
        match('c');  
        C();  
    }  
    else if (TOKEN == 'b') // First(bD)  
    {  
        match('b');  
        D();  
    }  
    else ERROR(); // 出错处理函数  
} //B
```

LL(1) 分析法

问题1:

$G[S]=\{$

$S \rightarrow Ab \mid Bc$

$A \rightarrow aA \mid dB$

$B \rightarrow c \mid e$

$\}$

试编写出能分析该文法所对应任何串（如串adcb）
的程序。

- (1) 应该有一个存储结构存储**何时选择哪条规则**？
- 根据递归分析法，是查看当前符号来决定选择哪条规则的，因此，应该存储某个非终结符号遇到某个终结符号时该选择哪条规则。

	a	b	c	d	e		
S	$S \rightarrow Ab$		$S \rightarrow Bc$	$S \rightarrow Ab$	$S \rightarrow Bc$		
A	$A \rightarrow aA$			$A \rightarrow dB$			
B			$B \rightarrow c$		$B \rightarrow e$		

LL(1) 分析表

- (2) **分析过程**的存储结构——分析栈

步骤	符号栈	输入串	动作
1	S	adcb	$S \rightarrow Ab$
2	bA	adcb	$A \rightarrow aA$
3	bAa	adcb	匹配
4	bA	dc b	$A \rightarrow dB$
5	bBd	dc b	匹配
6	bB	cb	$B \rightarrow c$
7	bc	cb	匹配
8	b	b	匹配
9			成功

LL(1)分析法

LL(1) 分析法

问题2:

$G[S]=\{$

$S \rightarrow AbB \mid Bc$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow d \mid e$

$\}$

试编写能分析该文法所对应任何串（如串abd）的程序。

```
void S( ) //规则:  $S \rightarrow AbB \mid Bc$ ,  
{  
    if (TOKEN in { 'a' , 'b' }) // First(AbB)  
    {  
        A();  
        match('b');  
        B();  
    }  
    else if (TOKEN in { 'd' , 'e' }) //First(Bc)  
    {  
        B();  
        match('c');  
    }  
    else ERROR(); // 出错处理函数  
} //S
```

```
void A( ) //规则:  $A \rightarrow aA | \epsilon$ ,  
{ if (TOKEN == 'a') // First(aA)  
  {  
    match('a');  
    A();  
  }  
else if (TOKEN == 'b') // Follow(A)  
  { }  
else ERROR();  
} //A
```

• (1)构造LL(1)分析表

	a	b	c	d	e	\$	
S	$S \rightarrow AbB$	$S \rightarrow AbB$		$S \rightarrow Bc$	$S \rightarrow Bc$		
A	$A \rightarrow aA$	$A \rightarrow \epsilon$					
B				$B \rightarrow d$	$B \rightarrow e$		

- (2) 分析过程

步骤	符号栈	输入串	动作
1	S	abd	$S \rightarrow AbB$
2	BbA	abd	$A \rightarrow aA$
3	BbAa	abd	匹配
4	BbA	bd	$A \rightarrow \epsilon$
5	Bb	bd	匹配
6	B	d	$B \rightarrow d$
7	d	d	匹配
8			成功

问题3:

$G[S]=\{$

$S \rightarrow bB \mid ACc$

$A \rightarrow aA \mid bB \mid \epsilon$

$B \rightarrow e \mid d$

$C \rightarrow f \mid \epsilon$

$\}$

试编写能分析该文法所对应任何串的程序。

问题4:

$G[S]=\{$

$S \rightarrow bB \mid Cc$

$A \rightarrow aAB \mid \epsilon$

$B \rightarrow a \mid d$

$C \rightarrow e \mid \epsilon$

$\}$

试编写能分析该文法所对应任何串的程序。

- **定理**：若满足以下条件，则BNF中的文法就是LL(1)文法(LL(1) grammar)。
 - 1. 在每个规则 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ 中,对于所有的 i 和 $j: 1 \leq i, j \leq n, i \neq j, \text{First}(\alpha_i) \cap \text{First}(\alpha_j)$ 为空。
 - 2. 若对于每个非终结符 A 都有 $\text{First}(A)$ 包含了 ϵ , 那么 $\text{First}(A) \cap \text{Follow}(A)$ 为空。

总结

LL(1)方法的步骤

- **步骤 (1)** : 构造LL(1)分析表(LL(1) parsing table)
- **步骤 (2)** : 分析过程

LL(1) 分析表

这个表通常被称为 $M[N, T]$

- N (即行)是文法的非终结符的集合;
- T (即列)是终结符或记号的集合
- $M[N, T]$ 即表示非终结符 N 面临输入符号 T 该选择的规则。
- $M[N, T]$ 缺省时(即为空), 则表示在分析中可能发生的潜在错误。

LL(1)分析表的构造步骤

为每个非终结符 A 和规则 $A \rightarrow \alpha$ 重复以下两个步骤：

- 1) 对于 $\text{First}(\alpha)$ 中的每个记号 a ，都将 $A \rightarrow \alpha$ 添加到项目 $M[A, a]$ 中。
- 2) 若 ϵ 在 $\text{First}(\alpha)$ 中，则对于 $\text{Follow}(A)$ 的每个元素 a (记号或是 $\$$)，都将 $A \rightarrow \alpha$ 添加到 $M[A, a]$ 中。
- 3) 把分析表 A 中每个未定义元素置为ERROR。

注意：通常用空白表示即可

例4.16 考虑if语句的简化了的文法：

$statement \rightarrow if-stmt \mid other$

$if-stmt \rightarrow if (exp) statement else-part$

$else-part \rightarrow else statement \mid \varepsilon$

$exp \rightarrow 0 \mid 1$

该文法各非终结符的First集合和Follow集合分别为：

$First(statement) = \{if, other\}$

$First(if-stmt) = \{if\}$

$First(else-part) = \{else, \varepsilon\}$

$First(exp) = \{0, 1\}$

$Follow(statement) = \{\$, else\}$

$Follow(if-stmt) = \{\$, else\}$

$Follow(else-part) = \{\$, else\}$

$Follow(exp) = \{\}$

- 构造出的LL(1)分析表如下：

M [N, T]	if	other	else	0	1	\$
statement	$statement \rightarrow$ $if-stmt$	$statement \rightarrow$ $other$				
if-stmt	$if-stmt \rightarrow$ $if(exp) statement$ $else-part$					
else-part			$else-part \rightarrow$ $else$ $statement$ $else-part \rightarrow \epsilon$			$else-part \rightarrow \epsilon$
exp				$exp \rightarrow$ 0	$exp \rightarrow$ 1	

二义性问题？

- **定理**：若满足以下条件，则BNF中的文法就是LL(1)文法(LL(1) grammar)。
 1. 在每个规则 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ 中,对于所有的 i 和 $j: 1 \leq i, j \leq n, i \neq j, \text{First}(\alpha_i) \cap \text{First}(\alpha_j)$ 为空。
 2. 若对于每个非终结符 A 都有 $\text{First}(A)$ 包含了 ϵ , 那么 $\text{First}(A) \cap \text{Follow}(A)$ 为空。

二义性消除的新方法

- 利用LL(1)分析表中每个项目最多只能有一个规则来消除文法的二义性。
- 我们知道程序设计语言中IF语句的else应该遵循最接近匹配原则。
- 所以，我们可对这个LL(1)分析表进行修改：
$$M[N,T] = \textit{else-part} \rightarrow \textit{else statement}$$

这样，表4-2就变成无二义性的了，而且可以对文法进行分析，这就好像它是一个LL(1)文法一样。

- 例如，表4-3显示了LL(1)分析算法的分析动作，它给出了串
if(0) if(1) other else other

\$ S	i (0) i (1) o e o \$	$S \rightarrow I$
\$ I	i (0) i (1) o e o \$	$I \rightarrow i (E) S L$
\$ L S) E (i	i (0) i (1) o e o \$	匹配
\$ L S) E ((0) i (1) o e o \$	匹配
\$ L S) E	0) i (1) o e o \$	$E \rightarrow 0$
\$ L S) o	0) i (1) o e o \$	匹配
\$ L S)) i (1) o e o \$	匹配
\$ L S	i (1) o e o \$	$S \rightarrow I$
\$ L I	i (1) o e o \$	$I \rightarrow i (E) S L$
\$ L L S) E (i	i (1) o e o \$	$I \rightarrow i (E) S L$
\$ L L S) E (i	i (1) o e o \$	匹配
\$ L L S) E ((1) o e o \$	匹配
\$ L L S) E	1) o e o \$	$E \rightarrow 1$
\$ L L S)) o e o \$	匹配
\$ L L S	o e o \$	$S \rightarrow o$
\$ L L o	o e o \$	匹配
\$ L L	e o \$	$L \rightarrow e S$
\$ L S e	e o \$	匹配
\$ L S	o \$	$S \rightarrow o$
\$ L o	o \$	匹配
\$ L	\$	$L \rightarrow e$
\$	\$	接受

- 为了简便，我们将图中的词进行缩写： $statement = S$ 、 $if-stmt = I$ 、 $else-part = L$ 、 $exp = E$ 、 $if = i$ 、 $else = e$ 、 $other = o$

• 分析过程

(1) 初始化：文法开始符号入栈

(2) 查表

(3) 替换

(4) 反复 (2) (3) 步骤，直到分析成功或失败

void LL_1() // LL (1) 分析算法

{ st.push(\$); st.push(S); /* 把\$和开始符号S压入分析栈 st中*/

T=下一输入符号; Flag=true;

while (Flag)

{ X=st.top();

if ($X \in V_T \cup \{\$ \}$)

if ($X == T$)

if ($T == \$$) Flag=false;

else { st.pop (); T=下一输入符号; }

else ERROR();

else /* $X \in V_N$ */

if ($A[X][T] == "X ::= x_1 x_2 \cdots x_k"$)

{ st.pop(); st.push($x_k x_{k-1} \cdots x_1$);

打印输出规则 $X ::= x_1 x_2 \cdots x_k$;

} else ERROR();

} /* while */

Accept(); /* 分析成功而结束 */

}

小结

- 递归子程序或递归下降分析法
- LL(1)分析方法
- 左递归
- 左公共因子

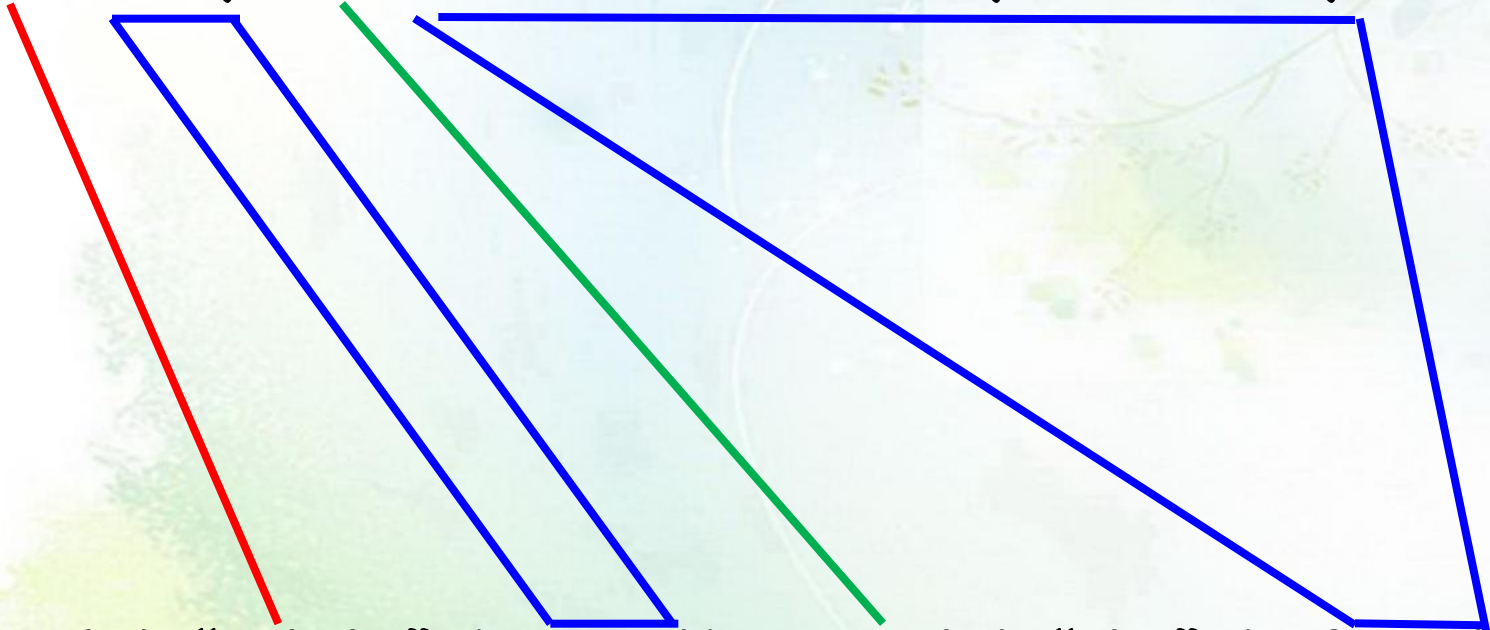
例：stmt \rightarrow while Exp do stmt

递归子程序如下：

```
void Stmt()  
{  
    Match( "while" );  
    Exp();  
    Match( "do" );  
    Stmt();  
}
```


while $x > y$ do if $x > z$ then $x := x + y$ else $x := y$

{ Match("while"); Exp(); Match("do"); Stmt(); }



- TINY语言的递归下降分析程序(自行阅读并运行相关程序(云盘))
- TINY语言的文法 P97 及 P136
- 实验四

程序清单 3-1 BNF 中的 TINY 的文法

```
program → stmt-sequence
stmt-sequence → stmt-sequence ; statement | statement
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence end
        | if exp then stmt-sequence else stmt-sequence end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp comparison-op simple-exp | simple-exp
comparison-op → < | =
simple-exp → simple-exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number | identifier
```

程序清单4-8 EBNF中TINY语言的文法

$program \rightarrow stmt\text{-}sequence$
 $stmt\text{-}sequence \rightarrow statement \{ ; statement \}$
 $statement \rightarrow if\text{-}stmt \mid repeat\text{-}stmt \mid assign\text{-}stmt \mid read\text{-}stmt \mid write\text{-}stmt$
 $if\text{-}stmt \rightarrow \mathbf{if} \ exp \ \mathbf{then} \ stmt\text{-}sequence \ [\ \mathbf{else} \ stmt\text{-}sequence \] \ \mathbf{end}$
 $repeat\text{-}stmt \rightarrow \mathbf{repeat} \ stmt\text{-}sequence \ \mathbf{until} \ exp$
 $assign\text{-}stmt \rightarrow \mathbf{identifier} \ := \ exp$
 $read\text{-}stmt \rightarrow \mathbf{read} \ \mathbf{identifier}$
 $write\text{-}stmt \rightarrow \mathbf{write} \ exp$
 $exp \rightarrow simple\text{-}exp \ [\ comparison\text{-}op \ simple\text{-}exp \]$
 $comparison\text{-}op \rightarrow < \mid =$
 $simple\text{-}exp \rightarrow term \{ addop \ term \}$
 $addop \rightarrow + \mid -$
 $term \rightarrow factor \{ mulop \ factor \}$
 $mulop \rightarrow * \mid /$
 $factor \rightarrow (\ exp \) \mid \mathbf{number} \mid \mathbf{identifier}$

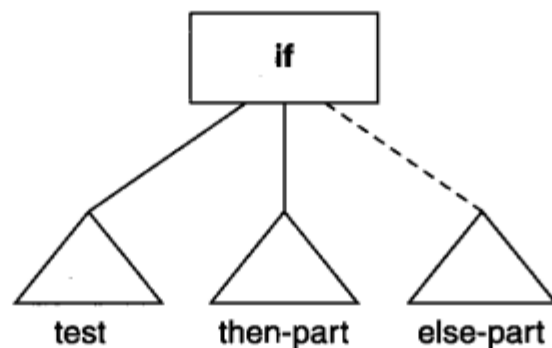
云盘参考书 P100

```

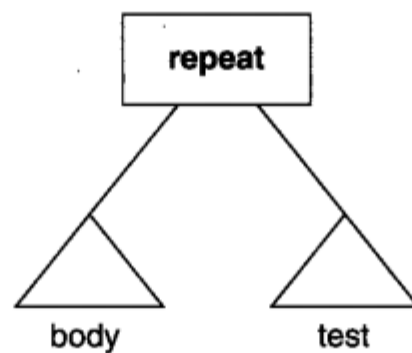
graph TD
    op("op (<opkind>") --- left_operand_triangle[ ]
    op --- right_operand_triangle[ ]
    left_operand_triangle --- left_operand_text["left operand"]
    right_operand_triangle --- right_operand_text["right operand"]

```

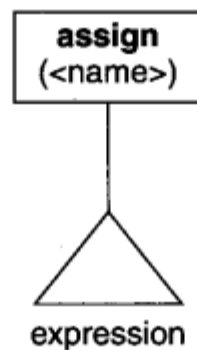
if 语句（带有3个可能的孩子）如下所示：



repeat 语句有两个孩子。第1个是表示循环体的语句序列，第2个是一个测试表达式：



assign 语句有一个表示其值是被赋予的表达式的孩子（被赋予的变量名保存在语句节点中）：



其他的语法树结构 云盘参考书 P100

语法树的存储结构（云盘参考书 P98）

程序清单 3-2 一个TINY语法树节点的C声明

```
typedef enum {StmtK, ExpK} NodeKind;
typedef enum {IfK, RepeatK, AssignK, ReadK, WriteK}
            StmtKind;
typedef enum {OpK, ConstK, IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void, Integer, Boolean} ExpType;

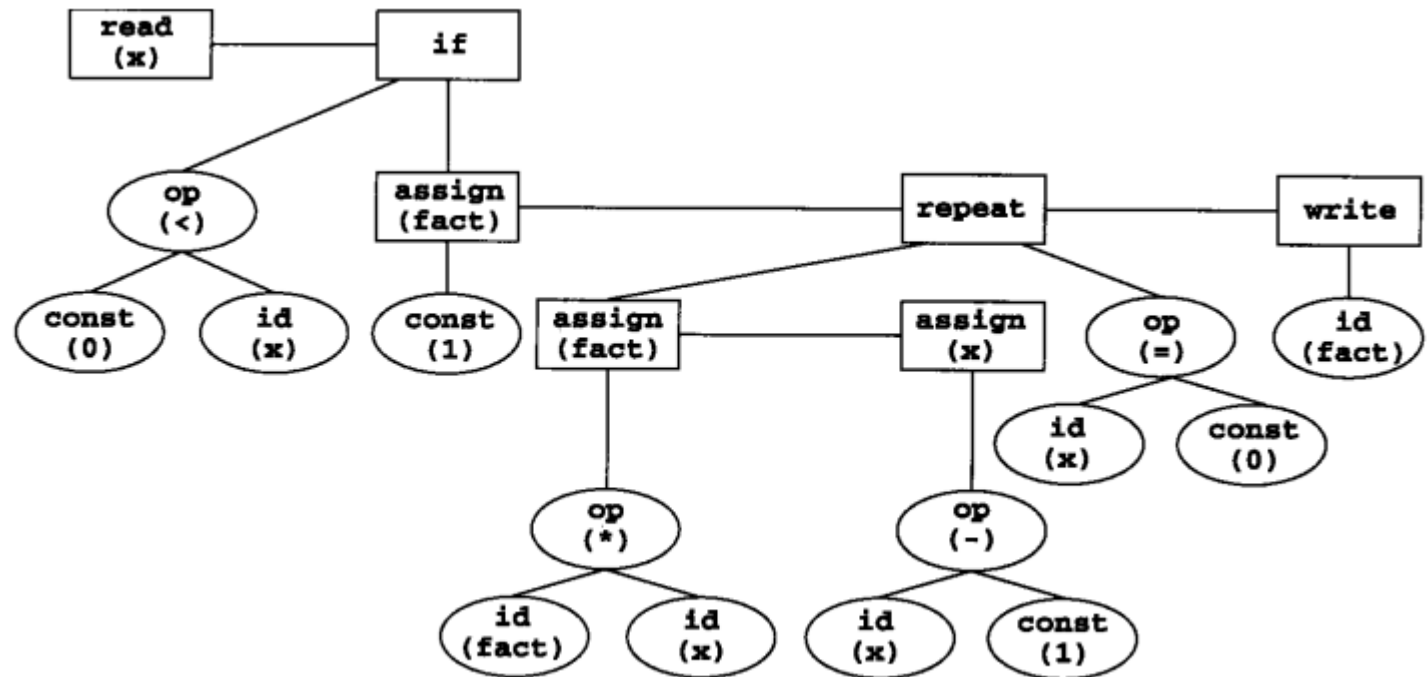
#define MAXCHILDREN 3

typedef struct treeNode
{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int lineno;
    NodeKind nodekind;
    union { StmtKind stmt; ExpKind exp; } kind;

    union { TokenType op;
            int val;
            char * name; } attr;
    ExpType type; /* for type checking of exps */
} TreeNode;
```


- Tiny样本语言：求输入值阶乘TINY语言程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```



TINY语言的递归子程序语法分析

程序清单4-8 EBNF中TINY语言的文法

```
program → stmt-sequence
stmt-sequence → statement { ; statement }
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence [ else stmt-sequence ] end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp [ comparison-op simple-exp ]
comparison-op → < | =
simple-exp → term { addop term }
addop → + | -
term → factor { mulop factor }
mulop → * | /
factor → ( exp ) | number | identifier
```

TINY语言语法分析源程序(云盘)

PARSE.C

PARSE.H

GLOBALS.H

结论——再分析

- 语言是一切句子的集合;

程序设计语言是一切程序的集合;

- 程序是程序设计语言的句子

程序清单4-8 EBNF中TINY语言的文法

program \rightarrow *stmt-sequence*
stmt-sequence \rightarrow *statement* { ; *statement* }
statement \rightarrow *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt \rightarrow **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt \rightarrow **repeat** *stmt-sequence* **until** *exp*
assign-stmt \rightarrow **identifier** := *exp*
read-stmt \rightarrow **read identifier**
write-stmt \rightarrow **write** *exp*
exp \rightarrow *simple-exp* [*comparison-op* *simple-exp*]
comparison-op \rightarrow < | =
simple-exp \rightarrow *term* { *addop* *term* }
addop \rightarrow + | -
term \rightarrow *factor* { *mulop* *factor* }
mulop \rightarrow * | /
factor \rightarrow (*exp*) | **number** | **identifier**

- Tiny样本语言：求输入值阶乘TINY语言程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```



结 论

- 自顶向下的分析程序有两类：

(1) 回溯分析程序 (backtracking parser)

(2) 预测分析程序 (predictive parser)。

- 预测分析程序尝试利用超前查看输入串的一个或多个符号来预测出该选择哪一条规则；
- 回溯分析程序则尝试分析所有的可能输入，当一种可能失败时就要求输入中备份任意数量的字符。

- 回溯分析程序的**特点**:

- (1) 功能强大,但非常慢,一般都在指数的数量级

- (2) 对于实际的编译器并不合适。

- 预测分析方法:

- (1) 递归下降分析 (recursive-descent parsing)

- (2) LL(1) 分析 (LL(1) parsing)。

- (1)递归下降分析法

常用，且比较适合于手写分析程序。

- (2)LL(1)分析方法

第1个“L”指由左向右地进行读入

第2个“L”指利用最左推导进行分析

括号中数字1——仅向前查看一个符号就能预测分析。

- (3)LL(k)分析——利用向前查看k个符号来进行预测分析。