第4章　自顶向下的分析

本章要点

- 使用递归下降分析算法进行自顶向下的分析
- LL(1)分析
- First 集合和Follow集合
- TINY 语言的递归下降分析程序
- 自顶向下分析程序中的错误校正

自顶向下（top-down）的分析算法通过在最左推导中描述出各个步骤来分析记号串输入。之所以称这样的算法为自顶向下是由于分析树隐含的编号是一个前序编号，而且其顺序是由根到叶（参见第3章的3.3节）。自顶向下的分析程序有两类：回溯分析程序（backtracking parser）和预测分析程序（predictive parser）。预测分析程序试图利用一个或多个先行记号来预测出输入串中的下一个构造，而回溯分析程序则试着分析其他可能的输入，当一种可能失败时就要求输入中备份任意数量的字符。虽然回溯分析程序比预测分析程序强大许多，但它们都非常慢，一般都在指数的数量级上，所以对于实际的编译器并不合适。本书将不研究回溯程序（但读者可查看“注意与参考”部分以及练习以得到一些关于这个主题的提示）。

本章要学习的两类自顶向下分析算法分别是递归下降分析（ recursive-descent parsing）和LL(1)分析（LL(1) parsing）。递归下降分析很常用，且它对于手写的分析程序最为适合，所以我们最先学习它。之后再来学习 LL(1)分析，由于在实际中并不常用到它，所以只是将其作为一个带有显式栈的简单实例来学习，它是下一章更强大（但也更复杂）的自底向上算法的前奏。它对于将出现在递归下降分析中的一些问题形式化也有帮助。 LL(1)分析方法是这样得名的：第1个“L”指的是由左向右地处理输入（一些旧式的分析程序惯于自右向左地处理输入，但现在已不常用了）。第2个“L”指的是它为输入串描绘出一个最左推导。括号中的数字1意味着它仅使用输入中的一个符号来预测分析的方向（“LL(k)分析”也是有可能的——它利用向前看的 k个符号，本章后面将简要地介绍到它，但是向前看的一个符号是最为常见的）。

递归下降程序分析和 LL(1)分析一般地都要求计算先行集合，它们分别称作 First集合和Follow集合[⊖]。由于无需显式地构造出这些集合就可以构造出简单的自顶向下的分析程序，所以在基本算法的介绍之后我们再讨论它们。之后我们还要谈到一个由递归下降分析构造的TINY分析程序，本章的最后是自顶向下的分析中的错误校正。

4.1　使用递归下降分析算法进行自顶向下的分析

4.1.1　递归下降分析的基本方法

递归下降分析的概念极为简单：将一个非终结符 *A*的文法规则看作将识别 *A*的一个过程的定义。*A*的文法规则的右边指出这个过程的代码结构：一个选择中的终结符与非终结符序列与

⊖　下一章将要研究的自底向上的分析算法有一些也需要这些集合。

相匹配的输入以及对其他过程的调用相对应，而选择与在代码中的替代情况（ case语句和if语句）相对应。

例如，考虑前一章的表达式文法：

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***
factor → **(** *exp* **)** | ***number***

及*factor* 的文法规则，识别*factor* 并用相同名称进行调用的递归下降程序过程可用伪代码编写如下：

```
procedure factor ;
begin
  case token of
  ( :   match( ( ) ;
        exp ;
        match( ) ) ;
  number  :
         match (number) ;
  else error ;
  end case ;
end factor ;
```

在这段伪代码中，假设有一个在输入中保存当前下一个记号的 *token*变量（以便这个例子使用先行的一个符号）。另外还假设有一个*match*过程，它用它的参数匹配当前的下一个记号。如果成功则前移，如果失败就表明错误：

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

现在脱离开在*match*和*factor*中被调用的未指定的*error*过程。可以假设它会打印出一个出错信息并退出。

请注意，在*match* (**(**)调用和*factor*中的*match* (***number***) 调用中，我们知道*expectedToken*和*token*是一样的。但是在*match* (**)**)调用中，不能将*token*假设为一个右括号，所以就需要有一个测试。*factor*的代码也假设已将过程 *exp*定义为可以调用。在表达式文法的递归下降分析中，*exp*过程将调用*term*，*term*过程将调用*factor*，而*factor*过程将调用*exp*，所以所有的这些过程都必须能够互相调用。不幸的是，为表达式文法中的其余规则编写递归下降程序过程并不像为*factor*编写一样简单，而且它需要使用EBNF，下面就介绍这一点。

4.1.2 重复和选择：使用EBNF

例如，一个if语句（简化了的）文法规则是：

if-stmt → **if** (*exp*) *statement*
| **if** (*exp*) *statement* **else** *statement*

可将它翻译成以下过程

```
procedure ifStmt ;
begin
  match (if) ;
  match (( ) ;
  exp ;
  match ( )) ;
  statement ;
  if token = else then
    match (else) ;
    statement ;
  end if ;
end ifStmt ;
```

在这个例子中，不能立即区分出文法规则右边的两个选择（它们都以记号 **if**开始）。相反地，我们必须直到看到输入中的记号**else**时，才能决定是否识别可选的else部分。因此，if语句的代码与EBNF

if-stmt → **if** (*exp*) *statement* [**else** *statement*]

匹配的程度比与BNF的匹配程序要高，上面的EBNF的方括号被翻译成ifStmt的代码中的一个测试。实际上，EBNF表示法是为更紧密地映射递归下降分析程序的真实代码而设计的，如果使用的是递归下降程序，就应总是将文法翻译成 EBNF。另外还需注意到即使这个文法有二义性（参见前一章），编写一个每当在输入中遇到**else**记号时就立即匹配它的分析程序也是很自然的。这与最近嵌套的消除二义性的规则精确对应。

现在考虑一下BNF中简单算术表达式文法中的*exp*情况：

exp → *exp addop term* | *term*

如要根据我们的计划试着将它变成一个递归的 *exp*过程，则首先应做的是调用 *exp*本身，而这将立即导致一个无限递归循环。由于 *exp*和*term*可以以相同的记号开头（一个数或左括号），所以要想测试使用哪个选择（*exp* → *exp addop term*或*exp* → *term*）就会出现问题了。

解决的办法是使用EBNF规则

exp → *term* { *addop term* }

花括号表示可将重复部分翻译到一个循环的代码中，如下所示：

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
```

```
        match (token) ;
        term ;
      end while ;
    end exp ;
```

相似地，*term*的EBNF规则：

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

就变成代码

```
    procedure term ;
    begin
      factor ;
      while token = * do
        match (token) ;
        factor ;
      end while ;
    end term ;
```

在这里当分隔过程时，删去了非终结符 *addop* 和*mulop*，这是因为它们仅有匹配算符功能：

$$addop \rightarrow \mathbf{+} \mid \mathbf{-}$$
$$mulop \rightarrow \mathbf{*}$$

这里所做的是*exp*和*term*中的匹配。

这个代码有一个问题：由花括号（和原始的 BNF中的显式）表示的左结合是否仍然保留。例如，假设要为本书中简单整型算术的文法编写一个递归下降程序计算器，就可通过在循环中轮转来完成运算，从而就保证了该运算是左结合（现在假设分析过程是返回一个整型结果的函数）：

```
    function exp : integer ;
    var temp : integer ;
    begin
      temp := term ;
      while token = + or token = - do
        case token of
          + : match (+) ;
              temp := temp + term ;
          - : match (-) ;
              temp := temp - term ;
        end case ;
      end while ;
      return temp ;
    end exp ;
```

*term*与之也类似。我们已利用这种方法创建了一个可运行的简单计算器，程序清单 4-1中有它

的C代码。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**的过程。

程序清单4-1 简单整型算术的递归下降程序计算器

```
/* Simple integer arithmetic calculator
   according to the EBNF:

   <exp> -> <term> { <addop> <term> }
   <addop> -> + | -
   <term> -> <factor> { <mulop> <factor> }
   <mulop> -> *
   <factor> -> ( <exp> ) | Number

   Inputs a line of text from stdin
   Outputs "Error" or the result.
*/

#include <stdio.h>
#include <stdlib.h>

char token; /* global token variable */

/* function prototypes for recursive calls */
int exp(void);
int term(void);
int factor(void);

void error(void)
{ fprintf(stderr,"Error\n");
  exit(1);
}

void match( char expectedToken)
{ if (token==expectedToken) token = getchar();
  else error();
}

main()
{ int result;
  token = getchar(); /* load token with first
                        character for lookahead */
  result = exp();
  if (token=='\n') /* check for end of line */
    printf("Result = %d\n",result);
  else error(); /* extraneous chars on line */
  return 0;
}
int exp(void)
{ int temp = term();
  while ((token=='+')||(token=='-'))
    switch (token) {
    case '+': match('+');
              temp+=term();
              break;
```

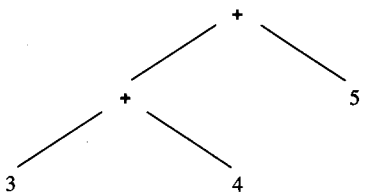
```
      case '-': match('-');
                temp-=term();
                break;
      }
    return temp;
}

int term(void)
{ int temp = factor();
  while (token=='*') {
     match('*');
     temp*=factor();
  }
  return temp;
}

int factor(void)
{ int temp;
  if (token=='(') {
    match('(');
    temp = exp();
    match(')');
  }
  else if (isdigit(token)) {
    ungetc(token,stdin);
    scanf("%d",&temp);
    token = getchar();
  }
  else error();
  return temp;
}
```

这个将EBNF中的文法规则转变成代码的办法十分有效， 4.4节还将利用它为TINY语言给出一个完整的分析程序。但是它仍有一些缺点，且须留意在代码中安排的动作。例如：在前面*exp*的伪代码中，运算的匹配必须发生在对*term*的重复调用之前（否则*term*会将一个运算看作是它的第1个记号，这就会生成一个错误了）。实际上现在必须严格遵循用于保持 *token*变量的协议：必须在分析开始之前先设置*token*，且对一个记号的测试（将它放在伪代码的*match*过程中）一旦成功，就立即调用*getToken*（或它的等价物）。

在构造语法树中也须注意动作的安排。我们已看到可通过在执行循环时完成计算来保持带有重复的EBNF的左结合。它再也不能与分析树或语法树中的自顶向下的构造相对应。但相反地，如考虑表达式**3+4+5**，其语法树



在根节点（表示其与5的和的节点）之前应先建立表示3与4之和的节点。将它翻译为真正的语法树结构就有了以下的*exp*过程的伪代码：

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         newtemp := makeOpNode(+) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
     - : match (-) ;
         newtemp := makeOpNode(-) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

或更简单的

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    newtemp := makeOpNode(token) ;
    match (token) ;
    leftChild(newtemp) := temp ;
    rightChild(newtemp) := term ;
    temp := newtemp ;
  end while ;
  return temp ;
end exp ;
```

这个代码使用了新函数*makeOpNode*，它接受作为参数的运算符记号并返回新建的语法树节点。我们还通过写出*leftChild* (*t*) := *p* 或*rightChild* (*t*) := *p* 指出将一个语法树*p* 的赋值作为语法树*t* 的一个左子树或右子树。有了这个伪代码之后， *exp*过程实际构造了语法树而不是分析树。这是因为一个对*exp*的调用并不总是构造一个新的树节点；如没有运算符， *exp*则仅仅是传送回从最初调用到*term*收到的树来作为它自己的值。也可以写出与 *term*和*factor*相对应的伪代码（参见练习）。

相反地，递归下降分析程序在严格的自顶向下的风格中可构造出if语句的语法树：

```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
   match (if) ;
   match (() ;
   temp := makeStmtNode(if) ;
   testChild(temp) := exp ;
   match ( ) ) ;
   thenChild(temp) := statement ;
   if token = else then
      match (else) ;
      elseChild(temp) := statement ;
   else
      elseChild(temp) := nil ;
   end if ;
end ifStatement ;
```

正因为递归下降分析允许程序设计人员可调整动作的安排，这样它就可以选择手工生成的分析程序了。

4.1.3 其他决定问题

前面描述的递归下降分析虽然非常强大，但它仍有特殊性。若使用的是一个设计精巧的小型语言（例如TINY，甚至是C），那么对于构造一个完整的分析程序，这些办法是适合的。读者应注意在复杂情况中还需要更形式的方法。此时还会出现一些问题。首先，将原先在 BNF中编写的文法转变成 EBNF格式可能会有些困难。下一节将学习不用 EBNF的另一种方法，其中构造了一个与EBNF基本相等的转变了的BNF。其次，在用公式表达一个用以区分两个或更多的文法规则选项的测试

$$A \rightarrow \alpha \mid \beta \mid \ldots$$

时，如果α和β均以非终结符开始，那么就很难决定何时使用$A \rightarrow \alpha$选项，何时又使用$A \rightarrow \beta$选项。这个问题就要求计算α和β的**First**集合：可以正规地开始每个串的记号集合。 4.3节将详细介绍这个计算。再次，在写ε 产生式的代码

$$A \rightarrow \varepsilon$$

时，需要了解什么记号可以正规地出现在非终结符 *A*之后，这是因为这样的记号指出 *A*可以恰当地在分析中的这个点处消失。这个集合被称作 *A*的**Follow**集合。4.3节中也有这个集合的准确计算。

最后读者需要注意：人们进行 First集合和Follow集合的计算是为了对早期错误进行探测。例如：在程序清单4-1的计算器程序中，假设有输入 `)3-2)`，分析程序在报告错误之前，将从`exp`到`term`再到`factor`下降；由于在表达式中，右括号作为第 1个字符并不合法，而在`exp`中可能早已声明了这个错误。 *exp*的*First*集合将会告诉我们这一点，从而可以进行较早的错误探测（本章最后将更详细地讨论错误探测和恢复）。

4.2 LL(1)分析

4.2.1 LL(1)分析的基本方法

LL(1)分析使用显式栈而不是递归调用来完成分析。以标准方式表示这个栈非常有用，这样LL(1)分析程序的动作就可以快捷地显现出来。在这个介绍性的讨论中，我们使用了生成成对括号的串的简单文法：

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

(参见第3章的例3.5)。

假设有这个文法和串( )，则表4-1给出了自顶向下的分析程序的动作。此表共有 4列。第1列为便于以后参考给每个步骤标上了号码。第 2列显示了分析栈的内容，栈的底部向左，栈的顶部向右。栈的底部标注了一个美元符号，这样一个在顶部包含了非终结符 S的栈就是：

$$\$\ S$$

且将额外的栈项推向右边。表的第 3列显示了输入。输入符号由左列向右。美元符号标出了输入的结束（它与由扫描程序生成的EOF记号相对应）。表的第4列给出了由分析程序执行的动作的简短描述，它将改变栈和（有可能）输入，如在表的下一行中所示一样。

表4-1 自顶向下的分析程序的分析动作

	分 析 栈	输 入	动 作
1	$ *S*	( ) $	*S* → (*S*) *S*
2	$ *S*) *S* (	( ) $	匹配
3	$ *S*) *S*	) $	*S* →ε
4	$ *S*)	) $	匹配
5	$ *S*	$	*S* →ε
6	$	$	接受

自顶向下的分析程序是从将开始符号放在栈中开始的。在一系列动作之后，它接受一个输入串，此时栈和输入都空了。因此，成功的自顶向下的分析的一般示意法应是：

$ *StartSymbol* *InputString* $

... ...

... ...

$ $ accept

在上面的例子中，开始符号是*S*，输入串是( )。

自顶向下的分析程序通过将栈顶部的非终结符替换成文法规则中（ BNF中）该非终结符的一个选择来作出分析。其方法是在分析栈的顶部生成当前输入记号，在顶部它已匹配了输入记号并将它从栈和输入中舍弃掉。这两个动作

1) 利用文法选择A→α将栈顶部的非终结符A替换成串α。

2) 将栈顶部的记号与下一个输入记号匹配。

是自顶向下的分析程序中的两个基本动作。第 1个动作称为生成（generate）：通过写出在替换中使用的BNF选择（它的左边在当前必须是栈顶部的非终结符）来指出这个动作。第 2个动作将栈顶部的一个记号与输入中的下一个记号匹配（并通过取出栈和将输入向前推进而将二者全

部舍弃掉）；这个动作是通过书写单词来指出的。另外还需注意在生成动作中，从 BNF中替换掉的串α必须颠倒地压在栈中（这是因为要保证串α按自左向右的顺序进到栈的顶部）。

例如，在表4-1的分析的第1步中，栈和输入分别是

$ *S* ( ) $

且用来替换栈顶部的*S*的规则是*S*→ (*S*) *S*，所以将串*S*) *S* (压入到栈中得到

$ *S*) *S* (( ) $

现在已生成了下一个输入终结符，即在栈的顶部的一个左括号，我们还完成了一个匹配以得到以下的情况：

$ *S*) *S*) $

表4-1中生成动作的列表与串( )最左推导的步骤完全对应：

S (*S*) *S* [*S* → (*S*) *S*]
( ) *S* [*S*→ε]
( ) [*S*→ε]

这是自顶向下分析的特征。如果要在分析进行时构造一个分析树，则可当将每个非终结符或终结符压入到栈中时添加节点来构造动作。因此分析树根节点的构造是在分析开始时进行的（与开始符号对应）。而在表4-1的第2步中，当(*S*) *S*替换*S*时，用于4个替换符号中的每个符号的节点将作为放到栈中的符号来构造，并且作为子节点与在栈中替换的 *S*节点连接。为了使其具有高效率，就必须修改栈以包括指向这些构造的节点的指针，而不是仅仅是指向非终结符或终结符本身的指针。另外，读者还将看到如何将这个处理进行修改以生成语法树的结构而非分析树的结构。

4.2.2 LL(1)分析与算法

当非终结符*A*位于分析栈的顶部时，根据当前的输入记号（先行），必须使用刚刚描述过的分析办法做出一个决定：当替换栈中的 *A*时应为*A*选择哪一个文法规则，相反地，当记号位于栈顶部时，就无需做出这样的决定，这是因为无论它是当前的输入记号（由此就发生一个匹配），还是不是输入记号（从而就发生一个错误），两者都是相同的。

通过构造一个**LL(1)**分析表(LL(1) parsing table)就可以表达出可能的选择。这样的表格基本上是一个由非终结符和终结符索引的二维数组，其中非终结符和终结符包括了要在恰当的分析步骤（包括代表输入结束的$）中使用的产生式选择。这个表被称为***M* [*N*, *T*]**，这里的*N*是文法的非终结符的集合，*T*是终结符或记号的集合（为了简便，禁止将 $加到*T*上），*M*可被认为是“运动的”表。我们假设表 *M* [*N*, *T*]在开始时，它的所有项目均为空。任何在构造之后继续为空的项目都代表了在分析中可能发生的潜在错误。

根据以下规则在这个表中添加产生式：

1) 如果*A*→α是一个产生式选择，且有推导α *a*β成立，其中*a* 是一个记号，则将*A*→α添加到表项目*M* [*A*, *a*] 中。
2) 如果*A*→α是一个产生式选择，且有推导α ε 和*S* $ βA*a*γ 成立，其中*S*是开始符号，*a*是一个记号（或$），则将*A*→α添加到表项目*M* [*A*, *a*]中。

这个规则的观点是：在规则1中，在输入中给出了记号*a*，若α可为匹配生成一个*a*，则希望挑选规则*A*→α。在规则2中，若*A*派生了空串（通过*A*→α），且如*a* 是一个在推导中可合法地出

现在*A*之后的记号，则要挑选*A*→α以使*A*消失。请注意规则2中当α=ε 时的特殊情况。

很难直接完成这些规则。在下一节中，我们将为此开发一个算法，这其中包括了早已提到过的First和Follow集合。但在极为简单的例子中，这些规则是可手工完成的。

读者可考虑在前一小节中使用成对括号的文法的第 1个示例，它有一个非终结符 (*S*)、3个记号（左括号、右括号和 $），以及两个选择。由于*S*只有一个非空产生式，即*S*→**(***S***)***S*，每一个可从*S*派生的串必须或为空或以左括号开始，并将这个产生式选择添加到项目*M*[*S*, **(**] 中（且仅能在这里）。这样就完成了规则 1之下的所有情况。因为有 *S* **(***S***)** *S*，规则2应用了α=ε，β= **(**, *A* = *S*, *a* = **)** 且γ = *S* $，所以*S*→ε 就被添加到*M*[*S*, **)**] 中了。由于*S* $ *S$（空推导），所以*S*→ε 也被添加到*M* [*S*, $] 中。这样就完成了这个文法的 LL(1)分析表的构造，我们可以将它写在下面的格式中：

M[*N*, *T*]	**(**	**)**	**$**
S	*S* → **(** *S* **)** *S*	*S* → ε	*S* → ε

为了完善LL(1)分析算法，该表必须为每个非终结符 -记号对给出唯一选择，可以从下面的定义开始。

定义：如果文法G相关的**LL(1)**分析表的每个项目中至多只有一个产生式，则该文法就是LL(1)文法(LL(1) grammar)。

由于上面的定义暗示着利用 LL(1)文法表就能构造出一个无二义性的分析，所以 LL(1)文法不能是二义性的。当给出 LL(1)文法时，程序清单 4-2中有使用 LL(1)分析表的一个分析算法。这个算法完全导致了在前一小节的示例中描述的那些动作。

虽然程序清单4-2中的算法要求分析表的每个项目最多只能有一个产生式，但仍有可能在表构造中建立消除二义性的规则，它可处理简单的二义性情况，如在与递归下降程序相似的方式中的悬挂else问题。

程序清单4-2 基于表的LL(1)分析算法

```
(* assumes $ marks the bottom of the stack and the end of the input *)
push the start symbol onto the top of the parsing stack ;
while the top of the parsing stack ≠ $ and the next input token ≠ $ do
  if the top of the parsing stack is terminal a
       and the next input token = a
  then (* match *)
    pop the parsing stack ;
    advance the input ;
  else if the top of the parsing is nonterminal A
       and the next input token is terminal a
       and parsing table entry M[A, a] contains
            production A → X1X2 . . . Xn
  then (* generate *)
    pop the parsing stack ;
    for i := n downto 1 do
      push Xi onto the parsing stack ;
  else error ;
if the top of the parsing stack = $
       and the next input token = $
then accept
else error ;
```

例如，if语句简化了的文法（参见第3章3.6节）：

statement → *if-stmt* | **other**
if-stmt → **if (*exp*)** *statement else-part*
else-part → **else** *statement* |ε
exp → **0** | **1**

构造LL(1)分析表就得出了表4-2中的结果，我们并未在表中列出括号终结符 **(**或**)**，这是因为它们并不引起动作（将在下一节中详细地解释这个表的构造）。

表4-2 （二义性的）if语句的LL(1)分析表

***M*[*N*, *T*]**	**if**	***Other***	**else**	**0**	**1**	**$**
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if (** exp **)** *statement else-part*					
else-part			*else-part* → **else** *statement* *else-part*→ε			*else-part* →ε
exp				*Exp* → **0**	*exp* → **1**	

在表4-2中，项目M[*else-part*, **else**]包括了两个项目，且它与悬挂else二义性对应。与在递归下降程序中一样，当构造这个表时，我们可以提供总是倾向于生成当前先行记号规则的消除二义性的规则，则倾向于产生式

else-part → **else** *statement*

而不是产生式*else-part*→ε。这实际上与最接近嵌套消除二义性规则对应。通过这个修改，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

（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*** ）。

表4-3 为if语句使用最接近嵌套消除二义性规则的 LL(1)分析

分 析 栈	输 入	动 作
$ *S*	**i (0) i (1) *o e o*** $	*S* → I
$ *I*	**i (0) i (1) *o e o*** $	*I* → **i (*E*)** *S* L
$ *LS* **)** *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* → **0**
$ *LS* **)** **o**	**0) i (1) *o e o*** $	匹配

（续）

分 析 栈	输 入	动 作
$ *L S*)	) i (1) o e o $	匹配
$ *L S*	i (1) o e o $	*S* → *I*
$ *L I*	i (1) o e o $	*I* → i (*E*) *S L*
$ *L L S*) *E* (i	i (1) o e o $	*I* → 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* → 1
$ *L L S*)	) o e o $	匹配
$ *L L S*	o e o $	*S* → o
$ *L L* o	o e o $	匹配
$ *L L*	e o $	*L* → e *S*
$ *L S* e	e o $	匹配
$ *L S*	o $	*S* → o
$ *L* o	o $	匹配
$ *L*	$	*L* →ε
$	$	接受

4.2.3 消除左递归和提取左因子

LL(1)分析中的重复和选择也存在着与在递归下降程序分析中遇到的类似问题，而且正是由于这个原因，还不能够为前一节的简单算法表达式文法给出一个 LL(1)分析表。我们利用EBNF表示法解决了递归下降程序中的这些问题，但却不能在LL(1)分析中使用相同的办法；而必须将BNF表示法中的文法重写到LL(1)分析算法所能接受的格式上。此时应用的两个标准技术是左递归消除（left recursion removal）和提取左因子（left factoring）。我们将逐个考虑它们。必须指出的是：这两个技术无法保证可将一个文法变成LL(1)文法，这就同EBNF一样无法保证在编写递归下降程序中可以解决所有的问题。然而，在绝大多数情况下，它们都十分有用，并且具有可自动操作其应用程序的优点，因此，假设有一个成功的结果，利用它们就可自动地生成LL(1)分析程序（参见“注意与参考”一节）。

1) 左递归消除 左递归被普遍地用来运算左结合，如在简单表达式文法中，

$$exp \rightarrow exp\ addop\ term \mid term$$

使得运算由*addop*左结合来代表。这是左递归的最简单情况，在其中只有一个左递归产生式选择。当有不止一个的选择是左递归时，就略微复杂一些了，如可将 *addop*写出来：

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

这两种情况都涉及到了直接左递归（immediate left recursion），而左递归仅发生在一单个非终结符（如exp）的产生式中。间接左递归是更复杂的情况，如在规则

$$A \rightarrow B\ b \mid \ldots$$
$$B \rightarrow A\ a \mid \ldots$$

中。这样的规则在真正的程序设计语言文法中几乎不可能发生，但本书为了完整仍给出它的解决方法。首先考虑直接左递归。

情况1：简单直接左递归

在这种情况下，左递归只出现在格式

$$A \rightarrow A\alpha \mid \beta$$

的文法规则中，其中α和β是终结符和非终结符的串，而且β不以A开头。在3.2.3节中，我们看到这个文法规则生成了格式$\beta\alpha^n$ ($n \geq 0$）的串。选择$A \rightarrow \beta$是基本情况，而$A \rightarrow A\alpha$是递归情况。

为了消除左递归，将这个文法规则重写为两个规则：一个是首先生成β，另一个是生成α的重复，它不用左递归却用右递归：

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

例4.1　再次考虑在简单表达式文法中的左递归规则：

$$exp \rightarrow exp\ addop\ term \mid term$$

它属于格式$A \rightarrow A\alpha \mid \beta$，且$A = exp$，$\alpha = addop\ term$，$\beta = term$。将这个规则重写以消除左递归，就可得到

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$$

情况2：普遍的直接左递归

这种情况发生在有如下格式的产生式

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

中。其中$\beta_1, \ldots, \beta_m$均不以A开头。在这种情况下，其解法与简单情况类似，只需将选择相应地扩展：

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

例4.2　考虑文法规则

$$exp \rightarrow exp + term \mid exp - term \mid term$$

如下消除左递归：

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow + term\ exp' \mid - term\ exp' \mid \varepsilon$$

情况3：一般的左递归

这里描述的算法仅是指不带有ε产生式且不带有循环的文法，其中循环（cycle）是至少有一步是以相同的非终结符：A　α　$*A$开始和结束的推导。循环几乎肯定能导致分析程序进入无穷循环，而且带有循环的文法从不作为程序设计语言文法出现。程序设计语言文法确实是有ε产生式，但这经常是在非常有限的格式中，所以这个算法对于这些文法也几乎总是有效的。

该算法的方法是：为语言的所有非终结符选择任意顺序，如$A_1, \ldots, A_m$，接着再消除不增加A_i'索引的所有左递归。它消除所有$A_i \rightarrow A_j\gamma$，其中$j \leq i$形式的规则。如果按这样操作从1到$m$的每一个$i$，则由于这样的循环的每个步骤只增加索引，且不能再次到达原始索引，因此就不会留下任何递归循环了。程序清单4-3是这个算法的详细步骤。

程序清单4-3 普遍左递归消除的算法

```
for i := 1 to n do
   for j := 1 to i−1 do
      replace each grammar rule choice of the form Ai → Aj β by the rule
         Ai → α1 β | α2 β | . . . | αk β, where Aj → α1 | α2 | . . . | αk is
         the current rule for Aj
   remove, if necessary, immediate left recursion involving Ai
```

例4.3 考虑下面的文法：

$$A \rightarrow B\ a \mid A\ a \mid c$$
$$B \rightarrow B\ b \mid A\ b \mid d$$

（由于该情况在任何标准程序设计语言中都不会发生，所以这个文法完全是自造的）。

为了使用算法，就须令B的数比A的数大（即$A_1 = A$，且$A_2 = B$）。因为$n = 2$，则图4-3中的算法的外循环执行两次，一次是当$i = 1$，另一次是当$i = 2$。当$i = 1$时，不执行内循环（带有索引j），这样唯一的动作就是消除A的直接左递归。最后得到的文法是

$$A \rightarrow B\ a\ A' \mid c\ A'$$
$$A' \rightarrow a\ A' \mid \varepsilon$$
$$B \rightarrow B\ b \mid A\ b \mid d$$

现在为$i = 2$执行外循环，且还执行一次内循环，此时$j = 1$。在这种情况下，我们通过用第1个规则中的选择替换A而省略了规则$B \rightarrow A\ b$。因此就得到了文法

$$A \rightarrow B\ a\ A' \mid c\ A'$$
$$A' \rightarrow a\ A' \mid \varepsilon$$
$$B \rightarrow B\ b \mid B\ a\ A'b \mid c\ A'b \mid d$$

最后消除B的直接左递归以得到

$$A \rightarrow B\ a\ A' \mid c\ A'$$
$$A' \rightarrow a\ A' \mid \varepsilon$$
$$B \rightarrow c\ A'\ b\ B' \mid d\ B'$$
$$B' \rightarrow b\ B' \mid a\ A'bB' \mid \varepsilon$$

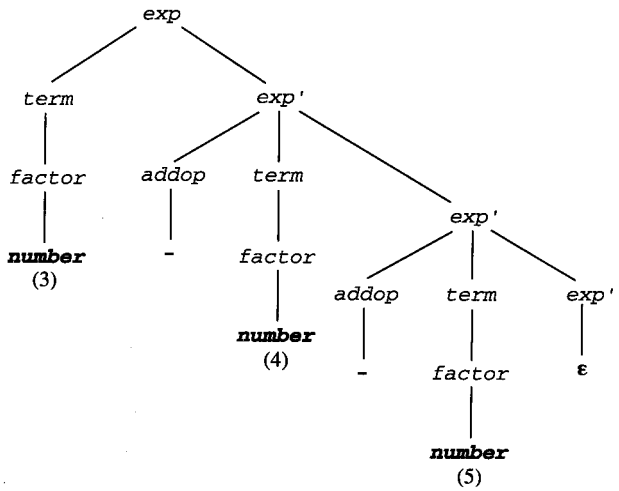
这个文法没有左递归。

左递归消除并不改变正被识别的语言，但它却改变了文法和分析树。这种改变确实导致了分析程序变得复杂起来（对于分析程序设计人员而言，也更困难了）。例如在前面作为标准示例的简单表达式文法中，该文法是左递归的，表达运算的结合性的表达式也是左递归。若要像在例4.1中一样消除直接左递归，就可得到图4-1中所给出的文法。

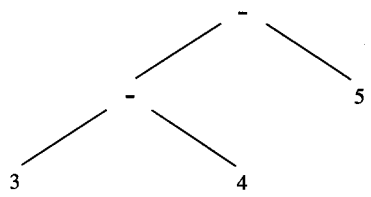
$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$$
$$addop \rightarrow \mathbf{+} \mid \mathbf{-}$$
$$term \rightarrow factor\ term'$$
$$term' \rightarrow mulop\ factor\ term' \mid \varepsilon$$
$$mulop \rightarrow *$$
$$factor \rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$

图4-1 消除了左递归的简单算术表达式文法

现在考虑表达式3-4-5的分析树：



这个树不再表达减法的左结合性了，然而，分析程序应仍构造出恰当的左结合语法树：



使用新文法来完成它不是完全微不足道的。为了看清原因，可先考虑利用给出的分析树来代替计算表达式的值这样略为简单的问题。为了做到这一点，必须将值 3从根*exp*节点传送到它的右子节点*exp′*。之后，这个*exp′* 节点必须减去4，并将新值-1向下传到它的最右边的子节点（另一个*exp′*）。这个*exp′* 节点按顺序也必须减去5并将值-6传送给最后的*exp′* 节点。这个节点仅有一个ε 子节点，且它仅仅是将值-6传回来。接着，该值被向上返回到树的根*exp*节点，它就是表达式的最终值。

考虑它在递归下降分析程序中是如何工作的。其左递归消除的文法将产生如下过程 *exp* 和*exp′*：

```
procedure exp ;
begin
   term ;
   exp′ ;
end exp ;
procedure exp′ ;
begin
   case token of
    + : match (+) ;
        term ;
        exp′ ;
    - : match (-) ;
```

```
            term ;
            exp' ;
        end case ;
    end exp' ;
```

为了使这些过程真正计算表达式的值，应将其如下所示重写：

```
    function exp : integer ;
    var temp : integer ;
    begin
        temp := term ;
        return exp'(temp) ;
    end exp ;

    function exp' ( valsofar : integer ) : integer ;
    begin
        if token = + or token = - then
            case token of
             + : match ( + ) ;
                 valsofar := valsofar + term ;
             - : match ( - ) ;
                 valsofar := valsofar - term ;
            end case ;
            return exp' (valsofar) ;
        else return valsofar ;
    end exp' ;
```

请注意*exp'* 过程现在是如何需要一个来自*exp*过程的参数。若这些过程将返回一个（左结合的）语法树，则会发生类似的情况。在4.1节里，给出的代码使用了基于EBNF的更为简单的解法中，它并不要求额外的参数。

最后，我们留意到程序清单 4-1中的新表达式文法实际上是一个 LL(1)文法。LL(1)分析表在表4-4中给出了。正如对前面表格的处理一样，我们将在下一节再谈到它的构造。

表4-4 程序清单4-1中文法的LL(1)分析表

M [*N*, *T*]	(	number	)	+	-	*	$
exp	*exp* → *term exp'*	*exp* → *term exp'*					
exp'			*exp'*→ε	*exp'*→ *addop term exp'*	*exp'*→ *addop term exp'*		*exp'*→ε
addop				*addop* → +	*addop* → -		

（续）

M [N, T]	(	number	)	+	-	*	$
term	term → factor term′	term → factor term′					
term′			term′ → ε	term′ → ε	term′ → ε	term′ → mulop factor term′	term′ → ε
mulop						mulop → *	
factor	factor → (exp)	factor → number					

2) 提取左因子 当两个或更多文法规则选择共享一个通用前缀串时，需要提取左因子。如

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

以下是语句序列的右递归示例（第3章的例3.7）：

stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*
stmt → **s**

以下是if语句的随后版本：

if-stmt → **if (** *exp* **)** *statement*
| **if (** *exp* **)** *statement* **else** *statement*

很明显，LL(1)分析程序不能区分这种情况中的产生式选择。这个简单情况的解法是将左边的 α 分解出来，并将该规则重写为两个规则

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

（如想用括号作为文法规则中的元符号，则也可写作 $A \rightarrow \alpha(\beta \mid \gamma)$，它看起来很像算术中的因子分解）。为了使提取左因子能够正常进行，就必须确保 α 实际上是与右边共享的最长串。这里还可能有享有相同前缀的超过两个的选择。程序清单 4-4中的是普通算法，之后还有一些示例。请注意，在处理算法时，每个共享相同前缀的非终结符的产生式的选择数目，每一步至少减少一个，这样算法才能够保证终止。

程序清单4-4 提取左因子文法的算法

while *there are changes to the language* **do**
  for *each nonterminal A* **do**
    let α *be a prefix of maximal length that is shared*
      by two or more production choices for A
  if $\alpha \neq \varepsilon$ **then**
    let $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *be all the production choices for A*
      and suppose that $\alpha_1, \ldots, \alpha_k$ *share* α, *so that*
      $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$, *the* β_j*'s share*

no common prefix, and the $\alpha_{k+1}, \ldots, \alpha_n$ *do not share* α
replace the rule $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *by the rules*
$A \rightarrow \alpha A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
$A' \rightarrow \beta_1 \mid \ldots \mid \beta_k$

例4.4 考虑语句序列的文法，它写在右递归格式中就是：

stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*
stmt → **s**

*stmt-sequence*的文法规则有一个共享的前缀，它可按如下提取左因子：

stmt-sequence → *stmt stmt-seq′*
stmt-seq′ → **;** *stmt-sequecne* | ε

请注意，如果已经是左递归，而不是右递归地写出了*stmt-sequence*规则

stmt-sequence → *stmt-sequence* **;** *stmt* | *stmt*

则消除直接左递归将会导致规则

stmt-sequence → *stmt stmt-seq′*
stmt-seq′ → **;** *stmt stmt-seq′* | ε

这与从提取左因子中得到的结果几乎一样，在最后的规则中将 *stmt stmt-seq′*替换成*stmt-sequence*就使得两个结果一样。

例4.5 考虑if语句的如下（部分）文法：

if-stmt → **if (** *exp* **)** *statement*
| **if (** *exp* **)** *statement* **else** *statement*

在这个文法中，提取了左因子的格式是

if-stmt → **if (** *exp* **)** *statement else-part*
else-part → **else** *statement* |ε

这正是在4.2.2节中所用到的格式（参见表4-2）。

例4.6 假设写出一个算术表达式文法，在其中给出了算术运算右结合性，而并非左结合性（这里用**+**仅是作为一个具体示例）：

exp → *term* **+** *exp* | *term*

这个文法需要提取左因子，则得到规则

exp → *term exp′*
exp′ → **+** *exp* |ε

现在继续做例4.4，假设在第2个规则中将*exp*替换为*term exp′*（由于这个扩展无论如何都会在推导中的下一步发生，所以这是正规的）。接着就得到

exp → *term exp′*
exp′ → **+** *term exp′* |ε

这与通过消除左递归而从左递归规则中得到的文法相同。因此，提取左因子和消除左递归都可导致语言结构的语义晦涩（在这种情况下，它们都会阻碍结合性）。

例如，若要保存来自上面文法规则的右结合性（在别的格式中），就必须将每个+运算安排在结尾而不是开头。请读者为它写出一个递归下降程序过程。

例4.7　因为过程调用和赋值均以标识符开头，所以这是程序设计语言文法不能成为LL(1)文法的典型情况。我们写出这个问题的以下表示：

statement → *assign-stmt* | *call-stmt* | **other**
assign-stmt → **identifier** := *exp*
call-stmt → **identifier** (*exp*-list)

因为**identifier**作为*assign-stmt* 和*call-stmt* 共享的第1个记号，所以这个文法不是LL(1)；而**identifier**也就是两个的先行记号。不幸的是，该文法不是位于一个可提取左因子的格式中。我们必须要做的是首先将*assign-stmt* 和*call-stmt* 用它们的定义产生式的右边代替，如下：

statement → **identifier** := *exp*
| **identifier** (*exp-list*)
| **other**

接着提取左因子以得到

statement → **identifier** *statement′*
| **other**
statement′ → := *exp* | (*exp-list*)

请注意它如何通过从真正的调用或赋值动作（由 *statement′* 代表）中分隔标识符（被赋予的值或被调用的过程）而使调用的语义和赋值变得晦涩。 LL(1)分析程序必须通过使标识符以某种方式（如一个参数）对调用或赋值步骤有用，或调整语法树来弥补它。

最后，应指出在所有的例子中，我们已使提取左因子确实在转换之后变成了 LL(1)文法。下一节将为其中的一些构造LL(1)分析表，另一些则留在练习中。

4.2.4　在LL(1)分析中构造语法树

我们还需探讨LL(1)分析如何适应于构造语法树而不是分析树（在 4.2.1节中已描述了如何利用分析栈构造分析树）。我们看到有关适用于语法树构造方法的递归下降程序分析的章节相对简单，而LL(1)分析程序却难以适用。其部分原因在于，正如我们所看到的，语法树的结构（如左结合性）会因提取左因子和消除左递归而变得晦涩。但是它的主要原因却是分析栈所代表的仅是预测的结构，而不是已经看到的结构。因此，语法树节点的构造必须推迟到将结构从分析栈中移走时，而不是当它们首次被压入时。一般而言，这就要求使用一个额外的栈来记录语法树节点，并在分析栈中放入“动作”标记来指出什么动作何时将在树栈中发生。自底向上分析程序（下一章）则易于适应使用了分析栈的语法树的构造，而且因此也就倾向于基于栈的表驱动分析方法。所以我们只给出一个简要的示例来解释它是如何用于 LL(1)分析的。

例4.8　我们使用一个仅带有一个加法运算地表达式文法。其BNF是

$E \to E$ **+ n | n**

这样就使得加法为左结合了。相应地带有消除左递归的LL(1)文法是

$E \to$ **n** E'
$E' \to$ **+ n** $E' \mid \varepsilon$

现在说明如何使用这个文法来计算表达式的算术值（语法树的构造是类似的）。

为了计算一个表达式的值，就要用一个单独的栈来存储计算的中间值，这个栈称作值栈（value stack）。在这个栈上必须安排两个运算。第1个运算是在输入中匹配数时将它压入，第2个是在栈中两数相加的运算。第1个运算由match过程完成（基于它匹配的记号），第2个需要被安排在分析栈上。它是通过将一个特殊符号压入分析栈中，而当其弹出时，则表示完成了一个加法运算。此处所用的符号是 (#)。这个符号现在成为一个新的栈符号，而且必须也被添加到匹配一个“ + ”的文法规则中，即E' 规则：

$$E' \rightarrow + \ \mathbf{n} \ \# \ E' \mid \varepsilon$$

请注意，将加法放在紧随下一个数之后的位置上，但是却在处理任何其他的 E′非终结符之前。这就保证了是左结合。现在来看看如何计算表达式 **3+4+5**的值。如前所述指出分析栈、输入和动作，但却将值栈放在右边（它向左增长）。表4-5是分析程序的动作。

表4-5 例4.8的带有值栈动作的分析栈

分 析 栈	输 入	动 作	值 栈
$ E	**3 + 4 + 5**$	$E \rightarrow$ ***n*** E'	$
$ E' ***n***	**3 + 4 + 5**$	匹配/压入	$
$ E'	**+ 4 + 5**$	$E' \rightarrow$ **+** ***n*** **#** E'	3 $
$ E' **#** ***n*** **+**	**+ 4 + 5**$	匹配	3 $
$ E' **#** ***n***	**4 + 5** $	匹配/压入	3 $
$ E' **#**	**+ 5** $	加法栈	4 3 $
$ E'	**+ 5** $	$E' \rightarrow$ **+** ***n*** **#** E'	7 $
$ E' **#** ***n*** **+**	**+ 5** $	匹配	7 $
$ E' **#** ***n***	**5** $	匹配/压入	7 $
$ E' **#**	$	加法栈	5 7 $
$ E'	$	$E' \rightarrow$e	12 $
$	$	接受	12 $

请注意，当发生一个加法时，操作数按相反顺序位于栈中。这是这种基于栈的赋值的典型安排。

4.3 First集合和Follow集合

为了完成LL(1)分析算法，我们开发了一个构造 LL(1)分析表的算法。正如早先已指出的，它涉及到计算 First集合和 Follow集合，本节将给出这些集合的定义和构造，之后再准确描述LL(1)分析表的构造。本节最后简要地介绍如何将该构造扩展为多于一个向前看符号。

4.3.1 First集合

定义：令X为一个文法符号（一个终结符或非终结符）或 ε，则集合First (X) 由终结符组成，此外可能还有ε，它的定义如下：

1. 若X是终结符或ε，则First (X) = {X}。
2. 若X是非终结符，则对于每个产生式 $X \rightarrow X_1 \ X_2 \ \ldots \ X_n$，First ($X$)都包含了First ($X_1$) - {ε}。若对于某个$i < n$，所有的集合First ($X_1$), . . . , First ($X_i$) 都包括了

ε，则First (*X*) 也包括了First (X_{i+1}) - {ε}。若所有集合First (X_1), . . . , First (X_n)包括了ε，则First (*X*)也包括ε。

现在为任意串α = $X_1 X_2 \ldots X_n$（终结符和非终结符的串）定义**First (α)**，如下所示：First (α)包括First (X_1) - {ε}。对于每个*i* = 2, . . . , *n*，如果对于所有的*k* = 1, . . . , *i* - 1，First (X_k) 包括了ε，则First (α)就包括了First (X_i) - {ε}。最后，如果对于所有的*i* =1, . . . , *n*，First (X_i) 包括了ε，则First (α)也包括了ε。

这个定义可很容易地转化为一个算法。实际上，唯一困难的是为每个非终结符 *A*计算First (*A*)，这是因为终结符的First 集合是很简单的，并且串α的First 集合从几乎*n* 个单个符号的First集合建立，其中*n* 是在α中的符号数。因此只有在非终结符时才为算法写成伪代码，如程序清单4-5所示。

程序清单4-5　为所有的非终结符A计算First (A)的算法

```
for all nonterminals A do First(A) := {};
while there are changes to any First(A) do
  for each production choice A → X1X2 . . . Xn do
    k := 1 ; Continue := true ;
    while Continue = true and k <= n do
      add First(Xk)−{ε} to First(A);
      if ε is not in First(Xk) then Continue := false ;
      k := k + 1 ;
    if Continue = true then add ε to First(A) ;
```

也可以很容易看出如何在没有ε 产生式的情况下解释这个定义：只需不断为每个非终结符*A*和产生式选择*A*→X_1 . . . ，向First (*A*)增加First (X_1)一直到再没有增加什么。换而言之，只需考虑程序清单4-5中*k* = 1的情况，而不需要while 内循环。我们将这个算法单独写在程序清单4-6中。当存在ε 产生式时，情况就复杂一些了，这是因为必须查清ε 是否在First (X_1)中，若是则为X_2 继续相同的处理，等等。该处理将一直延续到有穷步骤之后才结束；但是实际上，这个处理不但计算可作为从非终结符派生的串的第 1个符号而出现的终结符，而且它还决定非终结符是否可派生空串（即：消失）。这样的非终结符称为可空的：

程序清单4-6　当没有ε 产生式时，程序清单4-5的简化了的算法

```
for all nonterminals A do First(A) := {};
while there are changes to any First(A) do
  for each production choice A → X1X2 . . . Xn do
    add First(X1) to First(A);
```

定义：当存在一个推导*A*　*ε 时，非终结符*A*称作可空的（nullable）。

现在给出以下的法则。

定理：当且仅当First (*A*)包含ε 时，非终结符*A*为可空的。

证明：下面证明若*A*是可空的，则First (*A*)包含了ε。其逆命题也可用相似的方法得到证明。我们利用对产生式长度的归纳来推导。若 *A*　ε，则必有一个产生式*A*→ε，且由定义可得First (*A*)包含了First (ε) ={ε}。现在假设对于长度< *n* 的推导的语句为真，并令 *A*　$X_1 \ldots X_k$　*ε 是长度为*n* 的一个推导（利用产生式选择*A*→$X_1 \ldots X_k$)。如果任何

一个X_i都是终结符，则它们都不能派生ε，所以所有的X_i都必须是非终结符。实际上，上面的推导并不完整，它意味着每个X_i *ε，且推导少于n个步骤。因此，通过归纳假设，对于每个i，First (X_i) 都包含了ε。最后，由定义可得First (A) 必须包含ε。

下面给出一些非终结符的First集合的计算示例。

例4.9 考虑简单整型表达式文法[⊖]：

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***
factor → **(** *exp* **)** | ***number***

我们分别写出每个选择，这样就可按顺序思考它们了（还可为了引用作上编号）：

(1) *exp* → *exp addop term*
(2) *exp* → *term*
(3) *addop* → **+**
(4) *addop* → **-**
(5) *term* → *term mulop factor*
(6) *term* → *factor*
(7) *mulop* → ***
(8) *factor* → **(** *exp* **)**
(9) *factor* → ***number***

这个文法并未包括ε产生式，所以可使用程序清单4-6中简化了的算法。我们还注意到左递归规则1和5都未向First集合的计算增加任何东西[⊜]。例如，文法规则1规定只有那个First (*exp*)应被添加到First (*exp*)中。因此，可从计算中删除这些产生式。但在这个例子中，为了表示清楚仍把它们保留在列表中。

现在应用程序清单4-6的算法，并同时按照刚才给出的顺序考虑产生式。产生式1未做任何变化。产生式2将First (*term*)的内容增加到First (*exp*)，但是First (*term*)当前为空，所以它也没有任何变化。规则3和4分别给First (*addop*)增添“+”和“-”，所以First (*addop*) = {**+**, **-**}。规则5并未增加任何东西。规则6将First (*factor*)增添到First (*term*)中，但是First (*factor*)当前仍为空，所以也未发生任何改变。规则7向First (*mulop*)增添*，所以First (*mulop*) = {*}。规则8将(增加到First (*factor*)中，而规则9将***number***增加到First (*factor*)中，所以First (*factor*) = { **(**, ***number*** }。现在再从规则1开始，这是由于它并未有任何变化。现在规则1到5都未有任何变化(First(*term*)仍为空)。规则6将First (*factor*)增加到First (*term*)中，且First (*factor*) = {**(**, ***number*** }，所以现在也有First (*term*) = {**(**, ***number*** }。规则8和9未带来任何变化。我们必须再一次从规则1开始，这是因为有一个集合改变了，规则2最后将First (*term*)的内容增加到First (*exp*)中，且First (*exp*) = {**(**, ***number*** }。需要使用多遍文法规则，直到再没有改变发生，所以在4遍之后，

⊖ 这个文法具有左递归且不是LL(1)，所以我们不能为它建立一个LL(1)分析表。但是它仍是一个对于解释如何计算First集合的有用示例。

⊜ 当存在ε产生式时，左递归规则可用于First集合。

就已计算出以下的First集合：

First (*exp*) = { **(**, ***number*** }
First (*term*) = { **(**, ***number*** }
First (*factor*) = { **(**, ***number*** }
First (*addop*) = { **+**, **-** }
First (*mulop*) = { ***** }

(请注意，如果是一开始而不是最后为 *factor*列出文法规则，则可将文法规则使用遍数由 4减少为2)。表4-6列出了这个计算。在这个表中，只是在它们发生的恰当位置中将记录改变。空的项目表示在该步骤中串没有发生变化。由于没有改变发生，所以最后一遍不再进行。

表4-6 为例4.9中的文法计算First集合

文法规则	第 1 遍	第 2 遍	第 3 遍
exp → *exp* *addopterm*			
exp → *term*			First (*exp*) = {**(**, ***number***}
addop → **+**	First (*addop*) = {**+**}		
addop → **-**	First (*addop*) = {**+**, **-**}		
term → *term* *mulop factor*			
term → *factor*		First(*term*) = {**(**, ***number***}	
mulop → *****	First (*mulop*) = {*****}		
factor → **(** *exp* **)**	First (*factor*) = {**(**}		
factor → ***number***	First (*factor*) = {**(**, ***number***}		

例4.10 考虑if语句（例4.5）的（提取左因子）文法：

statement → *if-stmt* | **other**
if-stmt → **if** (*exp*) *statement else-part*
else-part → **else** *statement* |ε
exp → **0** | **1**

这个文法确实有ε 产生式，但只有*else-part*非终结符是可空的，所以计算的难度最小。由于没有一个是以其First 集合包括了ε 的非终结符开头，所以其实只需要在某一步中增加 ε，并保持其他步骤不受影响。这对于真正的程序设计语言文法是非常典型的，其中的 ε 产生式几乎总是非常有限的，且极少显示在普通情况中的复杂性。

同前面的一样，单独写出各个文法规则选择，并给其编号：

(1) *statement* → *if-stmt*
(2) *statement* → ***other***
(3) *if-stmt* → **if (** *exp* **)** *statement else-part*
(4) *else-part* → **else** *statement*
(5) *else-part* →ε
(6) *exp* → **0**
(7) *exp* → **1**

我们再次一个一个地运行产生式选择的步骤，一旦一个First集合在前一遍中有了改变，就制造出新的一遍来。因为First (*if-stmt*)为空，所以文法规则1开始时没有变化。规则2将终结符***other***增加到First (*if-stmt*)中，所以First (*statement*) = {***other***}。规则3将**if**增加到First (*if-stmt*)中，所以First (*if-stmt*) = {**if**}。规则4在First (*else-part*)中增加**else**，所以First (*else-part*) = {**else**}。规则5在First (*else-part*)中增加ε，所以First (*else-part*) = {**else**, ε}。规则6和规则7分别将**0**和**1**增加到First (*exp*)中，所以First (*exp*) = {**0**, **1**}。现在从规则1开始做另一遍。由于First (*if-stmt*)包括了**if**终结符，所以现在该规则又将**if**增加到First (*statement*)中。由此，First (*statement*) = {**if**,***other***}。第2遍中再没有其他改变了，且第3遍也无任何改变。因此，我们计算出以下First集合：

First (*statement*) = {**if**, ***other***}
First (*if-stmt*) = {**if**}
First (*else-part*) = {**else**,ε }
First (*exp*) = {**0**, **1**}

表4-7以与表4-6类似的方式展示了这个计算。同前面一样，该表只显示改变，且不显示最后一遍（因为在其中没有改变）。

表4-7 为例4.10中的文法计算First集合

文法规则	第 1 遍	第 2 遍
statement → *if-stmt*		First (*statement*) = {**if**, **other**}
statement → ***other***	First (*statement*) = {***other***}	
if-stmt → **if (** *exp* **)** *statement else-part*	First (*if-stmt*) = {**if**}	
else-part → **else** *statement*	First (*else-part*) = {**else**}	
else-part→ε	First (*else-part*) = {**else**,ε}	
exp → **0**	First (*exp*) = { **0**}	
exp → **1**	First (*exp*) = { **0**, **1**}	

例4.11 考虑下面的语句序列文法（参见例4.4）：

stmt-sequence → *stmt stmt-seq′*
stmt-seq′ → **;** *stmt-sequence* |ε

stmt → **s**

我们再一次单独列出每个产生式选择：

(1) *stmt-sequence* → *stmt stmt-seq′*
(2) *stmt-seq′* → **;** *stmt-sequence*
(3) *stmt-seq′* →ε
(4) *stmt* → **s**

在第1遍中，规则1并未增加任何东西。规则2和规则3导致First (*stmt-seq′*) = {**;**,ε}。规则4导致First (*stmt*) = {**s**}。在第2遍中，规则1这次使First (*stmt-sequence*) = First (*stmt*) = {**s**}。除此之外就再也没有其他改变了。第3遍也没有任何改变。我们计算出以下的First集合：

First (*stmt-ewquence*) = {**s**}
First (*stmt*) = {**s**}
First (*stmt-seq′*) = {**;**,ε }

请读者自己构造一个与表4-6和表4-7类似的表格。

4.3.2 Follow 集合

定义：给出一个非终结符*A*，那么集合**Follow(*A*)**则是由终结符组成，此外可能还有$。集合Follow(*A*)的定义如下：

1. 若*A*是开始符号，则$就在Follow(*A*)中。
2. 若存在产生式*B*→α*A*γ，则First (γ) - {ε}在Follow(*A*)中。
3. 若存在产生式*B*→α*A*γ，且ε 在First(γ)中，则Follow(*A*)包括Follow(*B*)。

首先检查这个定义的内容，之后为由此引出的Follow集合的计算写出算法。读者首先应注意到用作标记输入结束的“ $ ”，它就像是Follow集合计算中的一个记号。若没有它，那么在整个被匹配的串之后就没有符号了。由于这样的串是由文法的开始符号生成的，所以 $必须总是要增加到开始符号的Follow集合中（若开始符号从不出现在产生式的右边，那么它就是开始符号的Follow集合的唯一成分）。

读者其次需要注意的是：空的“ 伪记号 ” ε 永远也不是Follow集合的元素，它之所以有意义是因为ε 在First集合中是被仅仅用来标记那些可消失的串。在输入中它不能真正地被识别出来。而另一方面， Follow符号则总是相对于现存的输入（包括 $符号，它将匹配由扫描程序生成的EOF）来匹配的。

大家还应注意到Follow集合仅是针对非终结符定义的，然而First 集合却还可为终结符以及终结符串和非终结符串定义。我们可将 Follow集合的定义扩展到符号串，但由于在构造 LL(1)分析表时，仅需要非终结符的Follow集合，所以这是不必要的。

最后，我们还要留意Follow集合的定义在产生式的“ 右边 ”起作用，而 First集合的定义却是在“ 左边 ”起作用。由此就可说若 α不包括*A*，则产生式*A*→α 就没有任何有关*A*的Follow集合的信息。只有当*A*出现在产生式的右边时，才可得到Follow (*A*)。所以一般地，每个文法规则选择都可得到右边的每个非终结符的 Follow 集合。与在First 集合中的情况不同，每个文法规则选择添加唯一的非终结符（在左边的那个）的 First 集合。

此外，假设有文法规则 *A*→α*B*，那么Follow (*B*) 将通过定义中的情况 (3)包含Follow (*A*)。这是因为在任何包括了*A*的串中，*A*可被α*B*代替（这是动作中的“ 上下文无关 ”）。这个特性在

某种意义上与First 集合的情况相反：若有$A \rightarrow B\alpha$，那么First (*A*) 就包括了First (*B*)（除了ε 的可能）。

程序清单4-7给出了计算由Follow集合的定义得出的Follow集合的算法。利用这个算法为相同的3个文法计算Follow集合，我们曾在前面为这3个文法计算了First 集合（同在First集合中一样，当没有ε 产生式时，算法就简化了，我们把它留给读者）。

程序清单4-7 计算Follow集合的算法

```
Follow(start-symbol) := {$} ;
for all nonterminals A ≠ start-symbol do Follow(A) := {} ;
while there are changes to any Follow sets do
  for each production A → X1X2 . . . Xn do
    for each Xi that is a nonterminal do
      add First(Xi+1Xi+2 . . . Xn) − {ε} to Follow(Xi)
      (* Note: if i=n, then Xi+1Xi+2 . . . Xn = ε *)
      if ε is in First(Xi+1Xi+2 . . . Xn) then
        add Follow(A) to Follow(Xi)
```

例4.12 我们再考虑一下在例4.9中曾计算过的First集合的简单表达式文法，如下所示：

First (*exp*) = { **(**, ***number*** }
First (*term*) = { **(**, ***number*** }
First (*factor*) = { **(**, ***number*** }
First (*addop*) = { **+**, **-** }
First (*mulop*) = { ***** }

再次写出带有编号的产生式：

(1) *exp* → *exp addop term*
(2) *exp* → *term*
(3) *addop* → **+**
(4) *addop* → **-**
(5) *term* → *term mulop factor*
(6) *term* → *factor*
(7) *mulop* → *****
(8) *factor* → **(** *exp*)
(9) *factor* → ***number***

规则3、规则4、规则7和规则9的右边均无非终结符，所以它们未向Follow集合的计算增加任何东西。我们再按顺序考虑其他规则。在开始之前，先设Follow (*exp*) = {$}；其他的Follow集合都先设置为空。

规则1影响了3个非终结符的Follow集合：*exp*、*addop*和*term*。将First (*addop*)增加到Follow (*exp*)，所以现在Follow (*exp*) = {$, **+**, **-**}。接着，将First (*term*)增加到Follow (*addop*)中，所以现在Follow (*addop*) = { **(**, ***number***}。最后，将Follow (*exp*)增加到Follow (*term*)中，所以Follow (*term*) = {$, **+**, **-**}。

规则2再次导致Follow (*exp*)被添加到Follow (*term*)中，但是这刚才已由规则1完成，所以Follow集合就不再发生什么改变了。

规则5有3个影响：将First (*mulop*)添加到Follow (*term*)中，所以Follow (*term*) = {$, **+**, **-**, ***}。接着，将First (*factor*)被添加到Follow (*mulop*)中，所以Follow (*mulop*) = {**(**, ***number***}。最后将Follow (*term*)增加到Follow (*factor*)中，所以Follow (*factor*) = {$, **+**, **-**, ***}。

规则6导致的结果与规则5的最后一步相同，所以也就没有任何改变了。

最后，规则8将First (**)**) = { **)** }增加到Follow (*exp*)中，所以Follow (*exp*) = {$, **+**, **-**, **)**}。

在第2遍中，规则1将**)**增加到Follow (*term*)中(所以Follow (*term*) = {$, **+**, **-**, ***, **)**})；规则5将）增加到Follow (*factor*)中(所以Follow (*factor*) = {$, **+**, **-**, ***, **)**})。第3遍未引起任何改变，所以算法结束。这样我们就计算了以下的Follow集合：

Follow (*exp*) = {$, **+**, **-**, **)**}
Follow (*addop*) = {**(**, ***number***}
Follow (*term*) = {$, **+**, **-**, ***, **)**}
Follow (*mulop*) = {**(**, ***number***}
Follow (*factor*) = {$, **+**, **-**, ***, **)**}

同在计算First集合中的一样，我们将计算过程放在表 4-8中。同前面一样，在该表中省略了结束的遍而只指出当改变发生时 Follow集合的变化。我们还省略了对计算不可能有影响的 4个文法规则选择（之所以包括了两个规则*exp* → *term*和*term* → *factor*，是因为它们虽然没有真正的影响，但仍存在可能的影响）。

表4-8 为例4.12中的文法计算Follow集合

文法规则	第 1 遍	第 2 遍
exp → *exp addop* *term*	Follow (*exp*) = {$, **+**, **-**} Follow (*addop*) = {**(**, ***number***} Follow (*term*) = {$, **+**, **-**}	Follow (*term*) = {$, **+**, **-**, ***, **)**}
exp → *term*		
term → *term mulop* *factor*	Follow (*term*) = {$, **+**, **-**, ***} Follow (*mulop*) = {**(**, ***number***} Follow (*factor*) = {$, **+**, **-**, ***}	Follow (*factor*) = {$, **+**, **-**, ***, **)**}
term → *factor*		
factor → **(** *exp* **)**	Follow (*exp*) = {$, **+**, **-**, **)**}	

例4.13 再次考虑if语句简化了的文法，在例4.10中已计算了它的First集合，如下：

First (*statement*) = {**if**, ***other***}
First (*if-stmt*) = {**if**}
First (*else-part*) = {**else**,ε }

First (*exp*) = {**0**, **1**}

这里再重复一下带有编号的产生式：

(1) *statement* → *if-stmt*
(2) *statement* → ***other***
(3) *if-stmt* → **if (** *exp* **)** *statement else-part*
(4) *else-part* → **else** *statement*
(5) *else-part* →ε
(6) *exp* → **0**
(7) *exp* → **1**

规则2、规则5、规则6和规则7对Follow集合的计算没有影响，所以忽略它们。

首先设Follow (*statement*) = {$}，并将其他非终结符的Follow集合初始化为空。规则1现在将Follow (*statement*) 添加到Follow (*if-stmt*)中，所以Follow (*if-stmt*) = {$}。规则3影响到了*exp*、*statement*和*else-part*的Follow集合。首先，Follow (*exp*)得到First (**)**) = {**)**}，所以Follow (*exp*) = {**)**}。接着，Follow (*statement*) 得到First (*else-part*) - {ε}，所以Follow (*statement*) = {$, **else**}。最后，将Follow (*if-stmt*) 添加到Follow (*else-part*) 和Follow (*statement*) 中（这是因为*if-stmt*会消失）。第1个加法得到了Follow (*else-part*) = {$}，但第2个却无任何变化。最后，规则4将Follow (*else-part*)添加到Follow (*statement*)中，同样也没有任何改变。

在第2遍中，规则1再次将Follow (*statement*)添加到Follow (*if-stmt*)中，导致了Follow (*if-stmt*) = {$, **else**}。规则3现在将终结符**else**增加到Follow (*else-part*)中，所以Follow (*else-part*) = {$, **else**}。最后，规则4并未带来任何改变。第3遍同样也无任何变化，所以计算出以下的Follow集合：

Follow (*statement*) = {$, **else**}
Follow (*if-stmt*) = {$, **else**}
Follow (*else-part*) = {$, **else**}
Follow (*exp*) = {**)**}

请读者像上例一样为这个计算构造一个表。

例4.14 为例4.11中简化了的语句序列文法（并且带有文法规则选择）计算 Follow集合：

(1) *stmt-sequence* → *stmt stmt-seq′*
(2) *stmt-seq′* → **;** *stmt-sequence*
(3) *stmt-seq′* →ε
(4) *stmt* → **s**

在例4.11中，计算了以下的First集合：

First (*stmt-ewquence*) = {**s**}
First (*stmt*) = {**s**}
First (*stmt-seq′*) = {**;**,ε}

文法规则3和规则4并未对Follow集合的计算有任何影响。我们从Follow (*stmt-sequence*) = {$}开始，并使其他Follow集合为空。规则1导致了Follow (*stmt*) = {**;**}和Follow (*stmt-seq′*) = {$}。规则2未带来任何影响。另一遍也未产生更多的改变。所以计算出 Follow集合

Follow (*stmt-eequence*) = {$}

Follow (*stmt*) = { **;** }
Follow (*stmt-seq*′) = {$}

4.3.3 构造LL(1)分析表

现在考虑LL(1)分析表中各项的初始结构，如在4.2.2节中给出的：

1) 如果$A \to \alpha$ 是一个产生式选择，且有推导$\alpha \quad \alpha\beta$ 成立，其中α是一个记号，就将$A \to \alpha$ 添加到表项$M[A, a]$中。

2) 如果$A \to \varepsilon$ 是ε 产生式，且有推导$S\$ \quad \alpha Aa\ \beta$ 成立，其中S是开始符号，a 是一个记号（或$），就将$A \to \varepsilon$ 添加到表项$M[A, a]$ 中。

规则1中的记号a 很明显是在First (α) 中，且规则2的记号a 是在Follow (A)中，因此，就可得到LL(1) 分析表的以下算法构造：

LL(1) 分析表M[N, T] 的构造：为每个非终结符A和产生式$A \to \alpha$ 重复以下两个步骤：

1) 对于First (α)中的每个记号a，都将$A \to \alpha$添加到项目$M[A, a]$中。

2) 若ε 在First (α)中，则对于Follow (A) 的每个元素a（记号或是$），都将$A \to \alpha$ 添加到$M[A, a]$中。

下面的定理基本上是LL(1)文法的定义和刚才给出的分析表构造的直接结果，它的证明留在了练习中：

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. 在每个产生式$A \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$ 中，对于所有的i 和j：$1 \le i,\ j \le n,\ i \ne j$，First ($\alpha_i$) ∩First (α_j)为空。
2. 若对于每个非终结符A都有First (A) 包含了ε，那么First (A)∩Follow (A)为空。

现在针对前面的文法来看一些分析表的例子。

例4.15　考虑在本章中一直作为标准示例的简单表达式文法。该文法如最开始给出的（参见例4.9）一样为左递归。在前一节用消除左递归写出了一个相等的文法，如下：

exp → *term exp*′
exp′ → *addop term exp*′ |ε
addop → **+** | **-**
term → *factor term*′
term′ → *mulop factor term*′ |ε
mulop → ***
factor → **(** *exp* **)** | ***number***

必须为这个文法的非终结符计算First集合和Follow集合。我们把这个计算留到了练习中，这里只是给出结果：

First (*exp*) = { **(**, ***number*** }　　Follow (*exp*) = {$, **)** }
First (*exp*′) = {**+**, **-**,ε}　　Follow (*exp*′) = {$, **)** }
First (*addop*) = {**+**, **-** }　　Follow (*addop*) = { **(**, ***number*** }
First (*term*) = { **(**, ***number*** }　　Follow (*term*) = {$, **)**, **+**, **-** }
First (*term*′) = { ***** ,ε}　　Follow (*term*′) = {$, **)**, **+**, **-** }
First (*mulop*) = { ***** }　　Follow (*mulop*) = { **(**, ***number*** }

First (*factor*) = {**(**, ***number***} Follow (*factor*) = {$, **)**, **+**, **-**, *****}

应用刚才描述的LL(1)分析表就可得到如表4-4的表格了。

例4.16 考虑if 语句的简化了的文法：

statement → *if-stmt* | ***other***
if-stmt → **if (** *exp* **)** *statement else-part*
else-part → **else** *statement* |ε
exp → **0** | **1**

该文法的First集合和Follow集合分别在例4.10和例4.13中已做过了计算。这里再把它们列出来：

First (*statement*) = {**if**, ***other***} Follow (*statement*) = {$, **else**}
First (*if-stmt*) = {**if**} Follow (*if-stmt*) = {$, **else**}
First (*else-part*) = {**else**,ε} Follow (*else-part*) = {$, **else**}
First (*exp*) = {**0**, **1**} Follow (*exp*) = {**)**}

构造出的LL(1)分析表在表4-3中。

例4.17 考虑例4.4中的文法（带有了提取左因子）：

stmt-sequence → *stmt stmt-seq′*
stmt-seq′ → **;** *stmt-sequence* |ε
stmt → **s**

该文法有以下的First集合和Follow集合：

First (*stmt-ewquence*) = {**s**} Follow (*stmt-eequence*) = {$}
First (*stmt*) = {**s**} Follow (*stmt*) = {**;**, $}
First (*stmt-seq′*) = {**;**,ε} Follow (*stmt-seq′*) = {$}

下面是LL(1)分析表。

***M* [*N*, *T*]**	**s**	**;**	**$**
stmt-sequence	*stmt-sequence* → *stmt stmt-seq′*		
stmt	*stmt* → **s**		
stmt-seq′		*stmt-seq′* → **;** *stmt-sequence*	*stmt-seq′* →ε

4.3.4 再向前：LL(k)分析程序

前面的工作可推扩到先行k个符号。例如，可定义$\text{First}_k(\alpha) = \{ w_k \mid \alpha \quad * w \}$，其中$w$是一个记号串，且$w_k = w$的前$k$个记号（或若$w$中的记号少于$k$个，则为$w$）。类似地，还可定义$\text{Follow}_k(A) = \{ w_k \mid S\$ \quad * \alpha A w\}$。虽然这些定义比起当$k = 1$时的定义缺少一些“算法性”，但是仍可开发出计算这些集合的算法来，且LL(k) 分析表的构造也可像前面一样完成。

在LL(k) 分析中仍出现了一些复杂之处。首先，分析表变得大了许多，这是由于列数按 k 的指数次增加（对于一定的推扩，可利用表压缩方法算出来）。其次，分析表本身并不表达LL(k) 分析的全部能力，这主要是由于所有的 Follow串并不在所有的上下文中发生。因此，使

用如前所构造的表的分析与LL(*k*) 分析就不同了，LL(*k*) 分析被称作强LL(*k*) 分析(Strong LL(*k*) parsing)或SLL(*k*) 分析（SLL(*k*) parsing)。读者可查阅“注意与参考”一节以得到更多的信息。

LL(*k*) 分析程序和SLL(*k*) 分析程序在使用中都不普遍，其部分原因在于它们增加了难度，但主要原因还是在于对于任何的 k而言，在LL(1)中失败的文法在实际应用中也不可能会有LL(*k*) 。例如，无论k为多大，带有左递归的文法永远不会是LL(*k*) 。但是，递归下降分析程序却能在需要的时候选择使用更大的先行，甚至于如前所看到的对于任何 *k*，可使用特别的方法来分析不是LL(*k*) 的文法。

4.4 TINY语言的递归下降分析程序

本节将讨论附录B中列出的TINY语言的完整的递归下降程序。分析程序构造了如上一章的3.7节所描述的语法树，除此之外，它还将语法树的表示打印到列表文件中。分析程序使用如程序清单4-8中所给出的EBNF，它与第3章的程序清单3-1相对应。

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分析程序完全按照4.1节给出的递归下降程序的要点。这个分析程序包括两个代码文件：**parse.h**和**parse.c**。**parse.h**文件（附录B的第850行到第865行）极为简单：它由一个声明

```
TreeNode * parse (void);
```

组成，它定义了主分析例程**parse**，**parse**又返回一个指向由分析程序构造的语法树的指针。**parse.c**文件在附录B的第900行到第1114行中，它由11个相互递归的过程组成，这些过程与程序清单4-8中的EBNF文法直接对应：一个对应于*stmt-sequence*，一个对应于*statement*，5个分别对应于5种不同的语句，另有4个对应于表达式的不同优先层次。操作符非终结符并未包括到过程之中，但却作为与它们相关的表达式被识别。这里也没有过程与 *program*相对应，这是因为一个程序就是一个语句序列，所以**parse**例程仅调用**stmt_sequence**。

分析程序代码还包括了一个保存向前看记号的静态变量 **token**以及一个查找特殊记号的**match**过程，当它找到匹配时就调用**getToken**，否则就声明出错；此外代码还包括将出错信息打印到列表文件中的**syntaxError**过程。主要的**parse**过程将**token**初始化到输入的第1个记号中，同时调用**stmt_sequence**，接着再在返回由**stmt_sequence**构造的树之前检查源文件的末尾（如果在**stmt_sequence**返回之后还有更多的记号，那么这就是一个错误）。

每个递归过程的内容应有相对的自身解释性，**stmt_sequence**却有可能是一个例外，它可写在一个更为复杂的格式中以提高对出错的处理能力；本书将在出错处理的讨论中简要地解释这一点。递归分析过程使用 3种实用程序过程，为了方便已将它们都放在了文件 **util.c**（附录B的第350行到第526行）中，此外它还带有接口**util.h**（附录B的第300行到第335行）。这些过程是：

1) **newStmtNode**（第405行到第421行），它取用一个指示语句种类的参数，它还在分配该类的新语句节点的同时返回一个指向新分配的节点的指针。

2) **newExpNode**（第423行到第440行），它取用一个指示表达式种类的参数，它还在分配该类新表达式节点的同时返回一个指向新分配的节点的指针。

3) **copyString**（第442行到第455行），它取用一个串参数，为拷贝分配足够的空间，并拷贝串，同时返回一个指向新分配的拷贝的指针。

由于C语言不为串自动分配空间，且扫描程序会为它所识别的记号的串值（或词法）重复使用相同的空间，所以**copyString**过程是必要的。

util.c中也包括了过程**printTree**（第473行到第506行），**util.c**将语法树的线性版本写在了列表中，这样就能看到分析的结果了。这个过程在全程变量**traceParse**控制之下从主程序中调用。

通过打印节点信息以及在此之后缩排到孩子信息中来操作 **printTree**过程；从这个缩排中可构造真正的树。样本 TINY程序的语法树（参见第 3章的程序清单 3-3和图3-6）由**traceParse**打印到列表文件中，如程序清单4-9所示。

程序清单4-9 由printTree过程显示一个TINY语法树

```
Read: x
If
  Op: <
    const: 0
    Id: x
  Assign to: fact
    const: 1
  Repeat
    Assign to: fact
      Op: *
        Id: fact
        Id: x
    Assign to: x
      Op: -
        Id: x
        const: 1
    Op: =
      Id: x
      const: 0
  Write
    Id: fact
```

4.5 自顶向下分析程序中的错误校正

分析程序对于语法错误的反应通常是编译器使用中的主要问题。在最低限度之下，分析程序应能判断出一个程序在语句构成上是否正确。完成这项任务的分析程序被称作识别程序

(recognizer)，这是因为它的工作是在由正讨论的程序设计语言生成的上下文无关语言中识别串。值得一提的是：任何分析至少必须工作得像一个识别程序一样，即：若程序包括了一个语法错误，则分析必须指出某个错误的存在；反之若程序中没有语法错误，则分析程序不应声称有错误存在。

除了这个最低要求之外，分析程序可以显示出对于不同层次的错误的反应。通常地，分析程序会试图给出一个有意义的错误信息，这至少是针对于遇到的第 1个错误，此外它还试图尽可能地判断出错误发生的位置。一些分析程序甚至还可尝试着进行错误校正（ error correction）(或可能更恰当一些，应说是错误修复（ error repair))，在这里分析程序试图从给出的不正确的程序中推断出正确的程序。若它能这样做，那么这种情况绝大多数遇到的都是简单问题，如缺少了标点符号。在这里存在着一个算法体，它可被用来寻找在某种意义上与已给出的程序(通常是指必须被插入的、被删除的或被改变的记号）最近的正确程序。这样的最小距离错误校正（minimal distance error correction）当用于每个错误时通常效率就太低了，而且不管怎样它也是与程序员真正希望的相去甚远。因此，在真正的分析程序中极少能看到它。编译器编写者发现若不做错误校正则很难生成有意义的错误信息。

用于错误校正的大多数技术都很特别，在某种意义上说，它们需要特殊的语言和特殊的分析算法，在许多特殊情况下还要求有单独的解法，对此很难得到一般原理。以下是一些重要的事项。

1) 分析程序应试着尽可能快地判断出错误的发生。在声明错误之前等待太久就意味着真正的错误的位置可能已丢失了。

2) 在错误发生之后，分析程序必须挑选出一个可能的位置来恢复分析。分析程序应总是尝试尽可能多地分析代码，这是为了在翻译中尽可能多地找到真实的错误。

3) 分析程序应避免出现错误级联问题（error cascade problem），在这里错误会产生一个冗长的虚假的出错信息。

4) 分析程序必须避免错误的无限循环，此时不用任何输入都会产生一个出错信息的无限级联。

其中的一些目标自相矛盾，所以编译器的编写者必须在构造错误处理器时应作一些折衷。例如，避免错误级联和无穷循环问题会引起分析程序跳过一些输入，它与处理尽可能多的输入的目标相折衷。

4.5.1 在递归下降分析程序中的错误校正

递归下降分析程序中的错误校正的一个标准形式叫做应急方式（ panic mode)。这个名称是这样得来的：在复杂情况下，错误处理器将可能要试图在大量的记号中找到一个恢复分析的位置（在最糟的情况中，它可能甚至会用完所有剩余的程序，此时只有在错误出现后退出了)。但是，若在操作时小心一些，那么在进行错误校正时要比其名称的意味要好一些 ⊖。应急方式还具有一个优点：它能真正保证在错误校正时，分析程序不会掉到无穷循环之中。

应急方式的基本机制是为每个递归过程提供一个额外的由同步记号（ synchronizing token）组成的参数。在分析处理时，作为同步记号的记号在每个调用发生时被添加到这个集合之中。如若遇到错误，分析程序就向前扫描（scan ahead)，并一直丢弃记号直到看到输入中记号的一个同步集合为止，并从这里恢复分析。在做这种快速扫描时，通过不生成新的出错信息来（在某种程度上）避免错误级联。

在进行该错误校正时需要作出一个重要的判断：在分析的每步中需添加哪些记号。一般地，

⊖ 实际上，Wirth [1976] 将应急方式称作“ 不应急 ”的规则，这大概是试图要改善它的形象吧。

Follow集合是这样的同步记号中的重要一员。Follow集合也可用来使错误处理器避免跳过开始新的主要结构的重要记号（如语句或表达式）。First集合也很重要，这是因为它们允许递归下降分析程序能在分析的早些时候检测出错误，它对于任何错误校正总是有用的。当编译器“知道”何时不应着急时，应急方式工作得最好，掌握这一点很重要。例如，丢失了诸如分号或逗号，甚至是右括号的标点符号不应总是致使错误处理器耗费记号。当然，必须留意要确保不能发生无穷循环。

我们通过用伪代码在4.1.2节中的递归下降计算器中勾画它的执行过程来讲解应急方式的错误校正（参见程序清单 4-1）。除了保持基本相同（但错误再也不能立即退出了）的 *match*和*error*这两个过程之外，还有两个过程，完成早期的先行检查的 *checkinput*，以及应急方式记号耗费者拥有的*scanto*：

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset    { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset     followset ) ;
  end if ;
end;
```

这里的$指的是输入的结尾（EOF）。

这些过程如下所示地用于*exp*和*factor*过程中（它们现在得到了一个*synchset*参数）：

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
  end if;
end exp ;

procedure factor ( synchset ) ;
begin
  checkinput ({(, number}, synchset ) ;
  if not ( token in synchset ) then
```

```
    case token of
    ( : match (( ) ;
        exp ( { ) } ) ;
        match( ) ) ;
    number :
        match(number) ;
    else error ;
    end case ;
    checkinput ( synchset, { ( , number }) ;
  end if ;
end factor ;
```

请读者注意，*checkinput*在每个过程中都被调用了两次：一次是核实 First 集合的记号是输入中的下一个记号，另一次是核实Follow集合（或*synchset*）的记号是退出的下一个记号。

应急方式的这种格式将产生合理的错误（有用的出错信息也可作为参数被添加到 *checkinput* 和*error* 中）。例如，输入串`(2+-3)*4-+5`将产生两个出错信息（一个在第1个减号上，另一个在第2个加号上）。

一般地，我们注意到在递归调用中向下传送 *synchset*，同时也添加相应的新同步记号。在*factor*的情况中，当看到一个左括号之后会出现了一个例外：只有在作为它的 Follow集合时，*exp*才与右括号一起被调用（丢弃 *synchset*）。这是一种伴随应急方式错误校正的典型特殊分析（这样做了以后，例如表达式`(2+*)`将不在右括号上生成一个虚假的错误信息。）我们将该代码行为的分析以及其在C中的实现留在练习中。不幸的是，为了得到最佳的出错信息和错误校正，在实际中必须检查每个记号测试以看看是否需要做更一般的或更早的测试，而这样可能会增加错误行为。

4.5.2 在LL(1)分析程序中的错误校正

应急方式错误校正也可在 LL(1)分析程序中以与在递归下降分析中相似的方式实现。由于该算法是非递归的，所以就要求用一个新栈来保存 synchset参数，而且在算法生成每个动作之前（当一个非终结符位于栈顶之时），算法必须安排一个对 checkinput的调用㊀。请读者注意，在出现最初错误时，在栈顶部有一个非终结符 *A*，且当前的输入记号不在 First(*A*)中（或若ε 在 First(*A*)时，则为First (*A*))。记号位于栈的顶部且与当前输入记号不同的情况是不常见的，这是因为就一般而言，当在输入中真正地看到记号时，它们只会被压入到栈中（表压缩方法可能会和它折衷一点）。我们将把程序清单4-2中的分析算法的修改留在练习里。

若不用一个额外的栈，也可静态地将同步记号的集合与 checkinput所采取的相应动作一起建立到LL(1)分析表中。假设有一个位于栈顶部的非终结符 *A*和一个不在 First(*A*)（或若ε 在 First(*A*)时，则为First (*A*))中的输入记号，那么就有3种其他方法：

1) 由栈弹出*A*。
2) 看到一个为了它可重新开始分析的记号之后，成功地从输入中弹出记号。
3) 在栈中压入一个新的非终结符。

㊀ 同在递归下降代码中一样，位于匹配末尾的向 *checkinput*的调用也能由一个特殊的栈符号以类似于第 4.2.4节中值计算安排的风格来安排。

若当前输入记号是$或是在Follow(*A*)中时，就选择方法1。若当前输入记号不是$或不在First(*A*) Follow(*A*)中，就选择方法2。在特殊情况中方法3有时会有用，但却很少是恰当的（后面将简要地讨论另一种方法）。第1个动作是通过记号*pop*在分析表中指出，第2个动作是通过记号scan指出（请注意，pop动作与ε 产生式的归约相等价）。

有了这些惯例之后，LL(1)分析表（表4-4）看起来应同表4-9一样。例如：串**(2+*)**，使用该表的LL(1)分析程序的行为显示在表 4-10中。在这个表中，分析只显示为由第 1个错误开始（所以前缀**(2+**早已被成功地匹配了）。这里还是使用缩写词E代表*exp*，用E′代表*exp′*等等。请注意，在分析成功地再继续之前两个相邻的错误发生了移动。我们可以通过适当的安排来抑制第2个错误的出错信息在第1个错误之后移动，这样分析程序就可在产生任何新的错误信息之前成功地移动一次或多次了。因此，就可避免出错信息级联了。

表4-9 带有错误校正项的LL(1)分析表（表4-4）

M [*N*, *T*]	**(**	***number***	**)**	**+**	**-**	*****	**$**
exp	*exp*→ *term exp′*	*exp*→ *term exp′*	pop	scan	scan	scan	pop
exp′	scan	scan	*exp′*→ε	*exp′*→ *addop term exp′*	*exp′*→ *addop term exp′*	scan	*exp′*→ε
addop	pop	pop	scan	*addop*→ **+**	*addop*→ **-**	scan	pop
term	*term*→ *factor term′*	*term*→ *factor term′*	pop	pop	pop	scan	pop
term′	scan	scan	*term′*→ε	*term′*→ε	*term′*→ε	*term′* → *mulop factor term′*	*term′*→ε
mulop	pop	pop	scan	scan	scan	*mulop*→*****	pop
factor	*factor*→ **(** *exp* **)**	*factor*→ ***number***	pop	pop	pop	pop	pop

表4-10 LL(1)分析程序使用表4-9的移动

分 析 栈	输 入	动 作
$ *E′ T′* **)** *E′ T*	***)$	扫描（错误）
$ *E′ T′* **)** *E′ T*	**)**$	弹出（错误）
$ *E′ T′* **)** *E′*	**)**$	*E′* →ε
$ *E′ T′* **)**	**)**$	匹配
$ *E′ T′*	$	*T′* →ε
$ *E′*	$	*E′* →ε
$	$	接受

4.5.3 在TINY分析程序中的错误校正

正如在附录B中给出的一样，TINY分析程序的错误处理是极为初步的：它只需实现一个格

式非常原始的应急方式恢复，而无需同步集合。**match**过程仅声明错误，说出它发现的是哪个不希望的记号。除此之外，过程**statement**和**factor**在没发现正确选择时就声明错误。若在分析结束时，发现的是一个记号而不是文件的结束，则**parse**过程就声明错误。产生的主要出错信息是“非期望的记号”，它对于用户没有用处。此外，分析程序也不试着避免错误级联。例如，在write语句之后添加了一个分号的样本程序

```
...
 5:  read  x  ;
 6:  if  0  <  x  then
 7:    fact  := 1;
 8:    repeat
 9:      fact  := fact  *  x;
10:      x  := x  -  1
11:    until  x  =  0 ;
12:    write fact ;  {<- -  BAD  SEMICOLON !  }
13:  end
14:
```

产生了以下的两个出错信息（当只有一个错误已发生时）：

```
Syntax error at line 13: unexpected token -> reserved word: end
Syntax error at line 14: unexpected token -> EOF
```

当在代码的第2行中删去小于号“<”时，上面的程序就是：

```
...
 5:  read  x  ;
 6:  if  0  x  then  {  <- -  COMPARISON MISSING HERE!  }
 7:    fact  := 1;
 8:    repeat
 9:      fact  := fact  *  x;
10:      x  := x  -  1
11:    until  x  =  0 ;
12:    write fact ;
13:  end
14:
```

这样就打印出了4个出错信息：

```
Syntax error at line 6 : unexpected token -> ID, name = x
Syntax error at line 6 : unexpected token -> reserved word: then
Syntax error at line 6 : unexpected token -> reserved word: then
Syntax error at line 7 : unexpected token -> ID, name = fact
```

另一方面，TINY的某些行为是恰当的。例如，一个丢失掉的（不是额外的）分号将只生成一个出错信息，而分析程序则就如同分号一直在那儿一样继续建立正确的语法树，由此就完成了错误校正的一个基本形式。这个行为是由两种代码引起的。第1个是**match**过程并不耗费记号，它引起的行为与插入丢失的记号一样。第2个是已将**stmt_sequence**过程写出来，这样就在一个错误的情况下联结了尽可能多的语法树的匹配。特别地，必须留意在每当找到一个非零指针时都应与属指针相联结（将分析程序过程设计为若发现错误则返回一个零语法树指针）。另外，基于EBNF

```
statement();
while (token==SEMI)
{ match(SEMI);
```

```
    statement();
  }
```

的**stmt_sequence**体的明显书写方式是用带有一个更为复杂的循环测试代替：

```
  statement() ;
  while ((token!= ENDFILE) && (token!= END) &&
         (token!= ELSE) && (token!= UNTIL))
  { match(SEMI);
    statement();
  }
```

读者会注意到在这个反面的测试中的4个记号包含了*stmt_sequence*的Follow集合。这并不是出了什么错，而是因为测试将或是寻找First集合中的一个记号（同*statement*和*factor*的过程相同）或是寻找不在Follow集合中的一个记号。因为若丢失了一个First符号，分析就会停止，所以后者在错误校正中尤为有效。我们在练习中描述了一个程序的行为，读者可看到若在第1个格式中给出了**stmt_sequence**，则丢失的分号确实将致使跳过程序的剩余部分。

最后，我们还应注意到分析程序的编写方式：当它遇到错误时，它不能进入一个无穷循环（当**match**不损失一个不期望的记号时，读者会担心到这一个问题）。这是因为，在分析过程的任意路径中，最终都会遇到**statement**或**factor**的缺省情况，而两者都会在产生错误信息时消耗掉一个记号。

练习

4.1 编写与4.1.2节中*exp*的伪代码相对应的*term*和*factor*的伪代码，以使其可构造出简单算术表达式的语法树。

4.2 若有文法$A \rightarrow (A)\ A \mid \varepsilon$，请写出由递归下降分析该文法的伪代码。

4.3 若有文法

statement → *assign-stmt* | *call-stmt* | **other**
assign-stmt → ***identifier*** **:=** *exp*
call-stmt → ***identifier*** **(** *exp-list* **)**

请写出由递归下降分析该文法的伪代码。

4.4 若有文法

lexp → ***number*** | **(** *op lexp -seq* **)**
op → **+** | **-** | *****
lexp -seq → *lexp -seq lexp* | *lexp*

请写出伪代码以通过递归下降来计算一个*exp*的数值（参见第3章的练习**3.13**）。

4.5 利用表4-4识别以下算术表达式的LL(1)分析程序，请写出它们的动作：

a. `3+4*5-6` **b.** `3*(4-5+6)` **c.** `3-(4+5*6)`

4.6 写出利用4.2.2节中第1个表的LL(1)分析程序来识别以下成对括号的串：

a. `(())()` **b.** `(()())` **c.** `()(())`

4.7 若有文法$A \rightarrow (A)A \mid \varepsilon$，

a. 为非终结符*A*构造First集合和Follow集合。

b. 说明该文法是LL(1)的文法。

4.8 考虑文法

lexp → *atom* | *list*
atom → ***number*** | **identifier**
list → **(** *lexp -seq* **)**
lexp -seq → *lexp -seq lexp* | *lexp*

a. 消除左递归。
b. 为得出的文法的非终结符构造First集合和Follow集合。
c. 说明所得的文法是LL(1)文法。
d. 为所得的文法构造LL(1)分析表。
e. 假设有输入串

```
(a (b (2)) (c))
```

请写出相对应的LL(1)分析程序的动作。

4.9 考虑以下的文法（与练习**4.8**类似但不同）：

lexp → *atom* | *list*
atom → ***number*** | ***identifier***
list → **(** *lexp - seq* **)**
lexp -seq → *lexp* **,** *lexp -seq* | *lexp*

a. 在该文法中提取左因子。
b. 为所得的文法的非终结符构造First集合和Follow集合。
c. 说明所得的文法是LL(1)文法。
d. 为所得的文法构造LL(1)分析表。
e. 假设有输入串

```
(a, (b, (2)), (c))
```

写出相对应的LL(1)分析程序的动作。

4.10 考虑简化了的C声明的以下文法：

declaration → *type var-list*
type → **int** | **float**
var-list → ***identifier***, *var-list* | ***identifier***

a. 在该文法中提取左因子。
b. 为所得的文法的非终结符构造First集合和Follow集合。
c. 说明所得的文法是LL(1)文法。
d. 为所得的文法构造LL(1)分析表。
e. 假设有输入串

```
int x,y,z
```

写出相对应的LL(1)分析程序的动作。

4.11 诸如表4-4中的LL(1)分析表一般总有许多表示错误的空项。在许多情况下，一行中的所有空项可由一单个的缺省项（default entry）代替，由此就可将这个表大大地缩小。当非终结符有一个单独的产生式选择或当它有一个ε 产生式时，在非终结符行就有可能会出现缺省项。将该想法应用于表4-4，如果可以应用，有可能会出现怎样的问题？

4.12 **a.** LL(1)文法会有二义性吗？为什么？
b. 二义性文法会是LL(1)文法吗？为什么？

c. 非二义性的文法一定是LL(1)文法吗？为什么？

4.13 说明左递归文法不会是LL(1)文法。

4.14 证明4.3.3节中由其First集合和Follow集合的两个条件所得文法的定理。

4.15 将在记号串的两个集合 S_1 和 S_2 上的算符 定义为：S_1 S_2 = { First (*xy*) | $x \in S_1$, *y* S_2 }

a. 说明对于任何两个非终结符*A*和*B*，都有First(*AB*) = First (*A*) First(*B*)。

b. 说明4.3.3节中定理的两个条件可由某单一条件：若 $A \rightarrow \alpha$ 且$A \rightarrow \beta$，则(First (α) Follow(*A*))∩(First(β) Follow(*A*)) 为空来代替。

4.16 若由开始符号到*A*出现的记号串没有推导，则非终结符*A*是无用的。

a. 为该特性给出一个数学形式。

b. 程序设计语言有无可能具有一个无用的符号？请解释原因。

c. 若文法具有一个无用的符号，说明计算同在本章中给出的一样的 First集合和Follow集合会生成比用于构造真实的LL(1)分析表长得多的产生集合。

4.17 请给出无需改变被识别的语言就可从一个文法中删除无用非终结符（及相结合的产生式）的一个算法（参见前一个练习）。

4.18 若文法没有无用的非终结符，说明 4.3.3节中的定理的反命题也是正确的（参见练习**4.16**）。

4.19 给出例**4.15**中First集合和Follow集合的详细计算。

4.20 **a.** 为例**4.7**提取了左因子的文法中的非终结符构造First集合和Follow集合。

b. 利用(a)部分的答案构造LL(1)分析表。

4.21 若有文法$A \rightarrow a\ A\ a \mid \varepsilon$,

a. 说明该文法不是LL(1)文法。

b. 以下伪代码试图写出该文法的递归下降分析程序

```
procedure A ;
begin
  if token = a then
    getToken ;
    A ;
    if token = a then getToken ;
    else error ;
  else if token < > $ then error ;
end A ;
```

说明这个过程不能正确地运行。

c. 可以写出该语言的带回溯（backtracking）递归下降分析程序，但这样却要求使用将一个记号看作参数并将该记号返回到输入记号流前面的 unGettoken过程。它还要求将过程*A*写作返回成功或失败的布尔函数，这样当 *A*调用其自身时，它可在耗费另一个记号前先测试一下是否能成功；因此当 $A \rightarrow a\ A\ a$ 选择失败，代码还可继续尝试$A \rightarrow \varepsilon$。根据以上所述请重新写出 **(**b**)**部分的伪代码，并描绘其在串 *aaaa*$上的运算。

4.22 在程序清单4-8的TINY文法中，并没有将布尔表达式和算术表达式清楚地区分开。例如，以下是一个合乎语法的TINY程序：

```
if 0 then write 1>0 else x := (x<1)+1 end
```

请重新写出TINY文法以便只允许布尔表达式作为if语句或repeat语句的测试，且只允许算术表达式出现在write语句或assign语句中或作为任何算符的操作数。

4.23 将布尔算符**and**、**or**和**not**添加到程序清单4-8的TINY文法中。并给其赋予练习**3.5**中所描述的特性以及比所有算术算符都低的优先权。确保任何表达式都可以是布尔表达式或整型表达式。

4.24 对练习**4.23**中TINY语法的改变使得练习**4.22**所描述的问题变得更糟了。将练习**4.23**的答案重新改写以使布尔表达式和算术表达式能够严格地区分开来，并与练习**4.22**的解法相合并。

4.25 如4.5.1节所述：用作简单算术表达式文法的应急方式错误校正仍有一些缺点。其中之一是为算符测试的while循环应在特定的条件下继续运行。例如，表达式**(2)(3)**就在因子之间丢掉了算符，但是错误处理器在未重新开始分析之前就耗费了第2个因子。请读者重写伪代码以改进这样的行为。

4.26 利用表4-9中给出的错误恢复描述在输入**(2+-3)*4-+5**上的LL(1)分析程序的动作。

4.27 重写程序清单4-2中的LL(1)分析算法以使应急方式错误校正保持完整，并描绘其在输入**(2+-3)*4-+5**中的行为。

4.28 **a.** 描绘在TINY分析程序中的**stmt_sequence**过程的一个运算，它用于核实构造以下TINY程序的语法树（除了丢失的分号之外)的正确性：

```
x := 2
y := x + 2
```

b. 可为以下（不正确的）程序构造怎样的语法树：

```
x 2
y := x + 2
```

c. 假设用编写**stmt_sequence**过程来代替本章最后一节中的更为简单的代码：

```
statement();
while(token = SEMI)
{ match(SEMI);
  statement();
}
```

利用这种版本的**stmt_sequence**过程可为**a**部分和**b**部分中的程序构造怎样的语法树呢？

编程练习

4.29 将以下所述添加到程序清单4-1中的简单整型算术递归下降计算器内（确保它们具有正确的结合性和优先权）：

a. 带有符号**/**的整型除法。

b. 带有符号**%**的整型模。

c. 带有符号**^**的整型求幂（警告：该算符比乘法优先，且是右结合)。

d. 带有符号**-**的一目减（参见练习**3.12**)。

4.30 重写程序清单4-1的递归下降计算器，使其可与浮点数而不单是整型一起计算。

4.31 重写程序清单4-1的递归下降计算器，使其可区分浮点数和整型值，而不仅仅是将所有项都作为整型或浮点数来计算（提示：现在的“值”是指带有指示它是个整型还

是浮点的标志的记录)。

4.32 **a.** 重写程序清单4-1的递归下降计算器，使其根据3.3.2节的声明返回一个语法树。

b. 写出一个作为参数的函数，它得到由 **a**部分的代码生成的语法树，并由移动树来返回计算的值。

4.33 为与程序清单 4-1相似的整型算法写出一个递归下降计算器，但要使用图 4-1中的文法。

4.34 考虑以下的文法：

lexp → ***number*** | **(** *op lexp -seq* **)**
op → **+** | **-** | *****
lexp -seq → *lexp -seq lexp* | *lexp*

该文法可被看成是表示在类似 LISP前缀格式中的简单整型算术表达式。例如，表达式**34-3*42**在该文法中写作**(- 34 (* 3 42))**。
为该文法给出的表达式写出一个递归下降计算器。

4.35 **a.** 为前一个练习的文法设计一个语法树结构，并为返回语法树的它写出一个递归下降分析程序。

b. 写出一个作为参数的函数，它得到由 **a**部分的代码生成的语法树，并由编号树来返回计算的值。

4.36 **a.** 为练习**3.4**的正则表达式使用（被恰当地消除了二义性）的文法以构造一个读取正则表达式并执行NFA的Thompson结构（参见第2章）的递归下降分析程序。

b. 编写一个过程，它将NFA数据结构看作是由**a**部分的分析程序生成的，且根据子集结构构造等价的DFA。

c. 编写一个过程，它将数据结构看作是由 (b)部分的过程生成的，并根据它所代表的DFA在文本文件中寻找最长子串来进行匹配。（你的程序现在变成了 grep的一个“编译过的”版本了！）

4.37 将比较算符**<=**（小于或等于）、**>**（大于）、**>=**（大于或等于）以及**< >**（不等于）添加到TINY分析程序中（还将要求添加这些记号以及改变扫描程序，但是不应要求语法树有所改变）。

4.38 将在练习**4.22**中的文法变化合并到TINY分析程序中。

4.39 将在练习**4.23**中的文法变化合并到TINY分析程序中。

4.40 **a.** 将程序清单4-1中的递归下降计算器程序重新改写以完成如在第 4.5.1中描述的那样的应急方式错误校正。

b. 给**a**部分中你的错误处理添加有用的出错信息。

4.41 TINY分析程序只能产生极少的出错信息，其中的一个原因是 **match**过程被限制于在错误出现时只打印当前记号，而不是打印当前记号与所希望的记号，而且从其调用点没有特殊的出错信息被传送到**match**过程。将**match**过程重新改写以使它在打印当前记号的同时也打印出所希望的记号，并可将出错信息传给 **syntaxError**过程。这将要求重写**syntaxError**过程，以及改变**match**调用使其包括恰当的出错信息。

4.42 按4.5.1节中所述方法，给TINY分析程序添加记号的同步集合和应急方式错误校正。

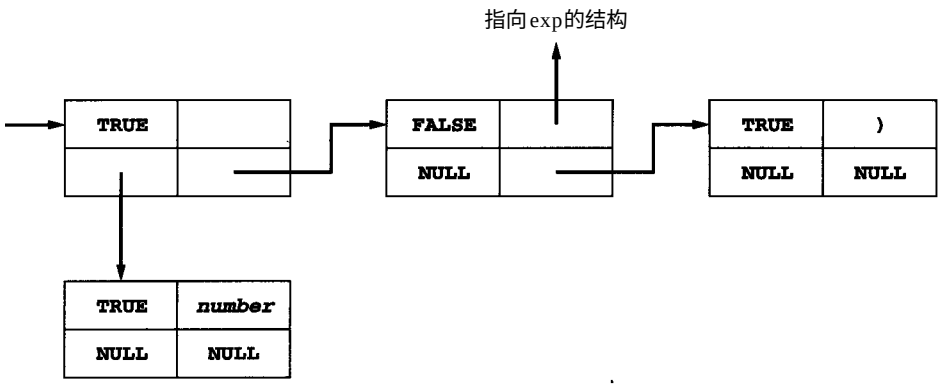
4.43 一个可分析LL(1)文法规则的任何集合的“一般的”递归下降分析程序可使用基于语法图的（来自Wirth [1976]的）数据结构。下面的C声明给出了一个适当的数据结构：

```
typedef struct rulerec
    { struct rulerec *next, *other;
      int isToken;
      union
      { Token name ;
        struct rulerec *rule ;
        } attr ;
    } Rulerec ;
```

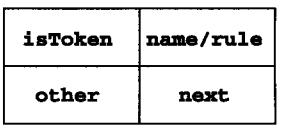
next域被用来指出文法规则中的下一个项目，而 **other**域则被用来指出由|元符号给出的替换项。因此，用于文法规则

factor → **(** *exp* **)** | ***number***

的数据结构看起来如下所示：



其中记录结构的域如下：



a. 为图4-1中的其余文法规则画出数据结构（提示：这将需要在数据结构内部代表 ε 的一个特殊记号）。

b. 写出使用这些数据结构识别输入串的一个普通分析过程。

c. 写出（从一个文件中或输入中）读取 BNF规则并生成前述的数据结构的分析程序生成器。

注意与参考

用于分析程序构造的递归下降分析自从20世纪60年代以及Algo160报告[Naur, 1963]的BNF规则的介绍中就已有了一个标准方法。若读者希望看到这一方法的较早描述，可参看 Hoare [1962]。回溯递归下降分析程序最近已在诸如 Haskell和Miranda的极为典型的懒惰函数语言中变得流行起来，其中这种递归下降分析的形式被称作组合器分析。读者可在 Peyton Jones 和 Lester [1992]或Hutton [1992]中找到这种方法的描述。在Wirth [1976]中，将EBNF连同递归下降分析一起使用已十分普通了。

LL(1)分析在20世纪的60年代和70年代早期已得到了广泛的研究。在 Lewis 和 Stearns

[1968]中可看到它的早期描述。在Fischer 和 LeBlanc [1991]中可找到对LL(k)分析的调查，其中还有一个不是SLL(2)的LL(2)文法的示例。Parr、Dietz和Cohen [1992]中有关于LL(k)分析的特殊应用。

当然，除了在本章所学到的两个之外，还有很多自顶向下的分析方法。 Graham、Harrison 和 Ruzzo [1980]中有其他的更为普通的方法。

Wirth [1976]和Stirling [1985]中有关于应急方式错误校正的研究。LL(k)分析程序中的错误校正可在Burke 和 Fisher [1987]中找到。读者还可在Fischer and LeBlanc [1991]和Lyon [1974]中学到更复杂的错误修改方法。

本章并未讨论到用于产生自顶向下的分析程序的自动工具，这主要是由于第 5章将讨论最常用的工具 —— Yacc。但是，好的自顶向下的分析程序生成器还是存在的。Antlr就是其中之一，它是Purdue Compiler Construction Tool Set(PCCTS)的一部分。读者可参看 Parr、Dietz和and Cohen [1992]中的相关内容。Antlr从EBNF中生成了一个递归下降分析程序。它具有大量有用的特征，这其中就包括了用于构造语法树的内部机制。读者可参看 Fischer 和 LeBlanc [1991]中有关称作LLGen的LL(1)分析程序生成器的大致内容。