第四讲　语法制导的语义计算基础

对于词法和语法正确的源程序，编译程序就可以对它进行语义分析。在语义分析阶段，首先是收集或计算源程序的上下文相关信息，并将这些信息分配到相应的程序单元记录下来。在这一过程中，若发现程序存在静态一致性或完整性方面的问题，则报告静态语义错误。若是不存在静态一致性或完整性方面的问题，我们称该程序通过了静态语义检查。语义分析过程中与静态语义检查相关的部分我们称之为静态语义分析。对于已通过静态语义检查的程序，编译程序可将其翻译到后续的中间表示形式，即中间代码生成。中间代码生成的过程体现了如何在更低的级别诠释程序的动态语义。关于静态语义分析和中间代码生成，可参阅本课程下一讲的内容。

在编译程序的实现中，一种经典的方法是由语法分析程序的分析过程来主导语义分析以及翻译的过程，本课程将其称为**语法制导的语义计算**。在编译方面的许多书籍中也称其为**语法制导的翻译**。对于特定的翻译遍，比如静态语义分析与中间代码生成，则可以相应地称其为语法制导的静态语义分析和语法制导的中间代码生成。

为描述完成什么样的语义计算，需要我们在语法定义的基础上建立适当的语义计算模型。如果语法定义采用上下文无关文法，则建立这种语义计算模型的基本途径是对上下文无关文法进行扩展，为文法符号附加语义信息，并针对产生式设计适当的语义动作，以便告诉分析引擎在语法分析过程中可以执行的语义动作。

这一讲中，我们将介绍两种重要的语义计算模型：属性文法和翻译模式。属性文法是一种基本的语义计算模型，宜于对一般原理的理解；而翻译模式是面向实现的语义计算模型，有助于理解语法制导的语义计算程序的自动构造方法（比如Yacc工具的工作原理）。

1 基于属性文法的语义计算

1.1 属性文法

1.1.1　引子

属性文法的理论体系较为复杂，由于我们仅将其作为描述语义计算的工具，所以本课程不从理论研究的视角出发，而是从实际应用的角度对其进行非形式的介绍。我们先通过通过例子来引入与属性文法相关的基本概念和术语。

考虑 $\{a, b, c\}$ 上的语言 $L = \{ a^n b^n c^n \mid n \geq 1\}$。可以证明（参考形式语言与自动机课程）：$L$ 不是任何上下文无关文法的语言。设有如下文法 $G[S]$：

$$
\begin{aligned}
&S \to ABC \\
&A \to Aa \mid a \\
&B \to Bb \mid b \\
&C \to Cc \mid c
\end{aligned}
$$

易知，这一文法的语言 $L(G) = L(a^+b^+c^+)$，这里 $a^+ = aa^*$，b^+ 和 c^+ 类似。显然有 $L \subseteq L(G)$。

如果对 $G[S]$ 附加某种限定条件，使其只产生满足这一限定条件的字符串，则可能接受语言 L。这一想法可以通过属性文法来实现。

我们在文法 $G[S]$ 基础上，为文法符号关联有特定意义的**属性**，并为产生式关联相应的**语义动作**或**条件谓词**，称之为**属性文法**，并称文法 $G[S]$ 是这一属性文法的**基础文法**。以下是基于 $G[S]$ 的一个属性文法：

$S \to ABC$	{ (*A.num*=*B.num*) *and* (*B.num*=*C.num*) }
$A \to A_1 a$	{ *A.num* := A_1 *.num* + 1 }
$A \to a$	{ *A.num* := 1 }
$B \to B_1 b$	{ *B.num* := B_1 *.num* + 1 }
$B \to b$	{ *B.num* := 1}
$C \to C_1 c$	{ *C.num* := C_1 *.num* + 1 }
$C \to c$	{ *C.num* := 1 }

文法符号的属性可用来刻画关联与该文法符号的任何特定意义的信息，如：值，名字串，类型，偏移地址，代码片断，等等。设文法符号 X 关联一个属性 a，我们用 $X.a$ 来表示对这个属性的访问。例如在上述属性文法中，*A.num* 表示对文法符号 A 所关联属性 *num* 的访问。

这里要注意的是，为了明确指出一个属性值对应于当前产生式中哪个位置的文法符号，在书写同一个文法符号时，我们会用到文法符号的下标形式。例如，在以上属性文法的第二条产生式中，A 和 A_1 分别表示出现于不同位置的文法符号 A。对于同一个属性 *num*，它们相应的属性值分别用 *A.num* 和 A_1*.num* 来访问。

在属性文法中，每个产生式 $A \to \alpha$ 都关联一个语义计算规则的集合，如上面例子中花括号内的部分。每个语义计算规则或者是一个语义动作，或者是一个条件谓词。本课程中，我们将语义动作的一般形式表示为：

$$b := f(c_1, c_2, \ldots, c_k)$$

其中，$b, c_1, c_2, \ldots, c_k$ 对应产生式中某些文法符号的属性。f 是用于描述如何计算属性值的函数，或称为**语义函数**。

语义动作也可以只包含一个语义函数，形如：

$$f(c_1, c_2, \ldots, c_k)$$

另外，形如 $X.a := Y.b$ 的语义动作称为**复写规则**。后面将会看到，复写规则对于语义计算程序的构造有独特的作用。

在具体应用中，语义函数可以通过实际的代码片断来实现，但一般不要有副作用，否则会使语义计算复杂化。

上面例子中的 (*A.num*=*B.num*) *and* (*B.num*=*C.num*) 是一个条件谓词，表示由当前属性的取值所决定的一个限定条件。

对于给定的属性文法，在基于语法分析过程进行语义计算时，使用某个产生式完成一步分析时将执行相应的语义动作，但其前提是必须满足相应的条件谓词。

通过以上解释或后续内容的学习，我们不难理解，上述属性文法可以接受的语言是 $L = \{ a^n b^n c^n \mid n \geq 1\}$。

由于在目前较实用的语法制导的语义计算程序构造工具中很少有相应的支持，所以本课

程不讨论包含条件谓词的属性文法。在实际应用中，我们可以采取像提示信息或其它方式达到同样的语义计算效果。

例 1 对于语言 $L = \{ a^n b^n c^n \mid n \geq 1\}$，我们可以设计如下属性文法作为语义计算模型：

$S \rightarrow ABC$	{ if (*A.num*=*B.num*) *and* (*B.num*=*C.num*) then *print* (“*Accepted*!”) else *print* (“*Refused*!”) }
$A \rightarrow A_1 a$	{ *A.num* := A_1 . *num* + 1 }
$A \rightarrow a$	{ *A.num* := 1 }
$B \rightarrow B_1 b$	{ *B.num* := B_1 . *num* + 1 }
$B \rightarrow b$	{ *B.num* := 1}
$C \rightarrow C_1 c$	{ *C.num* := C_1 . *num* + 1 }
$C \rightarrow c$	{ *C.num* := 1 }

对于该属性文法，若输入串属于 *L*，则语义计算结果会执行 *print* (“*Accepted*!”)，否则将会执行 *print* (“*Refused*!”)。当然，如果输入串不属于正规式 $aa^*bb^*cc^*$ 所表示的串，那么就会报告语法错误。

1.1.2 综合属性和继承属性

对关联于产生式 $A \rightarrow \alpha$ 的语义动作 $b := f(c_1, c_2, \ldots, c_k)$，如果 *b* 是 *A* 的某个属性, 则称 *b* 是 *A* 的一个**综合属性**。从分析树的角度来看，由于计算综合属性是对父结点的属性赋值，所以是“自底向上”传递信息。

对关联于产生式 $A \rightarrow \alpha$ 的语义动作 $b := f(c_1, c_2, \ldots, c_k)$，如果 *b* 是产生式右部某个文法符号 *X* 的某个属性，则称 *b* 是文法符号 *X* 的一个**继承属性**。从分析树的角度来看，由于计算继承属性是对子结点的属性赋值，所以是“自顶向下”传递信息。

在例 1 的属性文法例子中，文法符号 *A*，*B* 和 *C* 的属性 *num* 都是综合属性。图 1（*a*）是针对输入串 *aaabbbccc* 的一棵分析树。对此分析树进行自底向上（后序）遍历，并执行关联于相应产生式的语义动作，得到针对该输入串的一个语义计算过程，如图 1（*b*）所示。

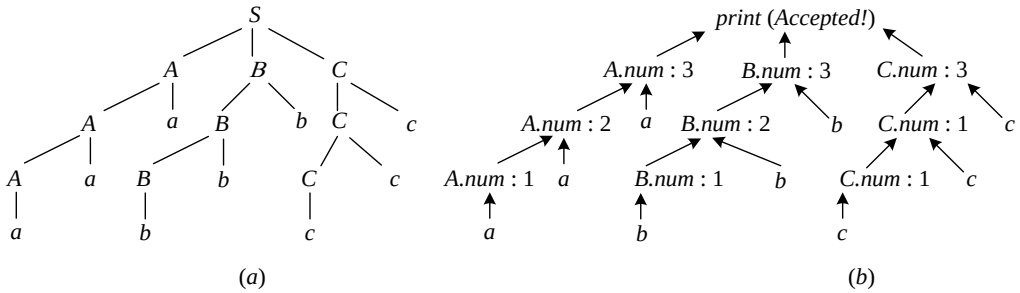

图 1 综合属性的计算：自底向上传递信息

再看一个含有继承属性的属性文法。

例 2 对于语言 $L = \{ a^n b^n c^n \mid n \geq 1\}$，我们还可以设计一个含有继承属性的属性文法作为语义计算模型（开始符号为 *S*）：

$S \to ABC$	{ $B.in\_num := A.num$; $C.in\_num := A.num$; if ($B.num$=0 *and* ($C.num$=0) then *print* (“*Accepted*!”) else *print* (“*Refused*!”) }
$A \to A_1\, a$	{ $A.num := A_1.num + 1$ }
$A \to a$	{ $A.num := 1$ }
$B \to B_1\, b$	{ $B_1.in\_num := B.in\_num$; $B.num := B_1.num$ -1 }
$B \to b$	{ $B.num := B.in\_num$ -1 }
$C \to C_1\, c$	{ $C_1.in\_num := C.in\_num$; $C.num := C_1.num$ -1 }
$C \to c$	{ $C.num := C.in\_num$ -1 }

这个属性文法既包含综合属性，又包含继承属性。其中，$A.num$，$B.num$ 和 $C.num$ 是综合属性，而 $B.in\_num$ 和 $C.in\_num$ 是继承属性。

同例 1，对于该属性文法，当输入串属于 L，则语义计算结果会执行 *print* (“*Accepted*!”)，否则将会执行 *print* (“*Refused*!”)。如果输入串不属于正规式 $aa^*bb^*cc^*$ 所表示的串，那么就会报告语法错误。

在图 1（a）的分析树基础上，对于综合属性值进行自底向上计算，而对于继承属性值进行自顶向下计算，得到输入串 *aaabbbccc* 的一个语义计算过程，如图 2 所示。

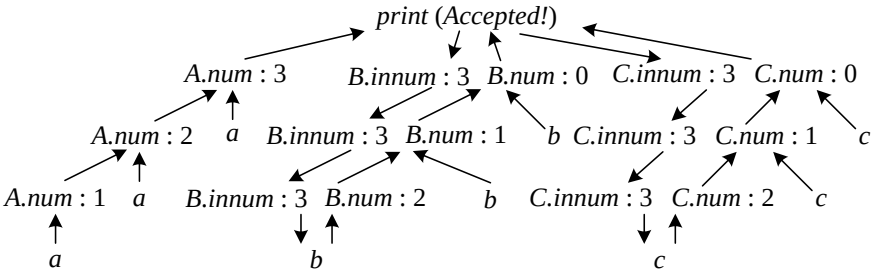

图 2 继承属性的计算：自顶向下传递信息

下面看一个更复杂的例子。

例 3 以下属性文法可用于将二进制无符号小数转化为十进制小数（开始符号为 N）：

$N \to S_1.S_2$	{ $N.v := S_1.v + S_2.v$; $S_1.f := 1$; $S_2.f := 2^{-S_2.l}$ }
$S \to S_1 B$	{ $S_1.f := 2S.f$; $B.f := S.f$; $S.v := S_1.v + B.v$; $S.l := S_1.l + 1$ }
$S \to B$	{ $S.l := 1$; $S.v := B.v$; $B.f := S.f$ }
$B \to 0$	{ $B.v := 0$ }
$B \to 1$	{ $B.v := B.f$ }

其中，各个属性具有如下含义：

- 符号 N 的综合属性 v 表示十进制小数形式的转化结果。
- 符号 S 的综合属性 v 表示二进制整数或定点小数（小数点之后的二进制数）对应的十进制数值。
- 符号 S 的综合属性 l 表示二进制整数或定点小数的 0、1 串长度。

- 符号 S 的继承属性 f 表示 S 推导的 0、1 串中最末一位为 1 时应该对应的十进制数值。从属性文法的第一行中的 $S_1.f := 1$ 和第二行中的 $S_1.f := 2S.f$ 可知：小数点前第一位数为 1 时对应的十进制数值为 $2^0 = 1$，小数点前第二位数为 1 时对应的十进制数值为 $2^1 = 2$，小数点前第三位数为 1 时对应的十进制数值为 $2^2 = 4$，等等。从属性文法的第一行中的 $S_2.f := 2^{-S_2.l}$ 和第二行中的 $S_1.f := 2S.f$ 可知：小数点后第一位数为 1 时对应的十进制数值为 $2^{-1} = 0.5$，小数点后第二位数为 1 时对应的十进制数值为 $2^{-2} = 0.25$，小数点后第三位数为 1 时对应的十进制数值为 $2^{-3} = 0.125$，等等。

- 符号 B 的综合属性 v 表示二进制数的当前这一位数字（0 或 1）对应的十进制数值。

- 符号 B 的继承属性 f 表示二进制数的当前这一位数字是 1 时应该对应的十进制数值。含义类似于符号 S 的继承属性 f。

基于该属性文法进行语义计算的过程比起前面两个例子要复杂一些，各个属性之间有比较复杂的依赖关系。在随后的小节里，我们将以这一属性文法为例来介绍一种通用的方法：遍历分析树进行语义计算。

1.2 遍历分析树进行语义计算

基于属性文法，通过遍历分析树进行语义计算可以采取下列步骤：

- 构造输入串的语法分析树。

- 构造依赖图。

- 若该依赖图是无圈的，则按造此无圈图的一种拓扑排序对分析树进行遍历，从而计算所有的属性值。若依赖图含有圈，则这一步骤失效。

这里，**依赖图**是一个有向图，用来描述分析树中的属性与属性之间的相互依赖关系。图 3 描述了构造依赖图的一般过程。

---

for 分析树中每一个结点 n **do**
  for 结点 n 所用产生式的语义动作中涉及的每一个属性 a **do**
    为 a 在依赖图中建立一个结点；
  for 结点 n 所用产生式中每个形如 $f(c_1, c_2, \cdots, c_k)$ 的语义动作 **do**
    为该规则在依赖图中也建立一个结点（称为虚结点）；
for 分析树中每一个结点 n **do**
  for 结点 n 所用产生式对应的每个语义动作 $b := f(c_1, c_2, \cdots, c_k)$ **do**
    （可以只是 $f(c_1, c_2, \cdots, c_k)$，此时结点 b 为一个虚结点）
    for $i := 1$ **to** k **do**
      从结点 c_i 到结点 b 构造一条有向边

图 3　构造依赖图的一般过程

例 4 对于例 3 的属性文法，考虑针对输入串 10.01 的语义计算过程。首先，基于输入串 10.01 的分析树，根据图 3 描述的方法构造依赖图。为分析树中所有结点的每个属性建立一个依赖图中的结点，并给定一个标记序号。结果，该依赖图共有 21 个结点，分别标记为 1~21，如

图 4 所示。依赖图中的有向边如图 5 所示。

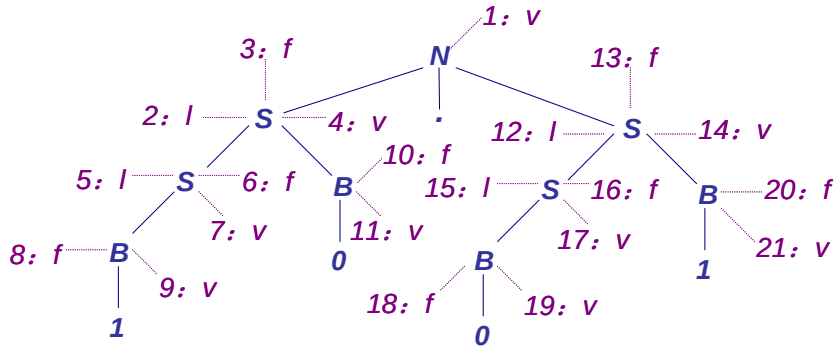

图 4　依赖图的结点

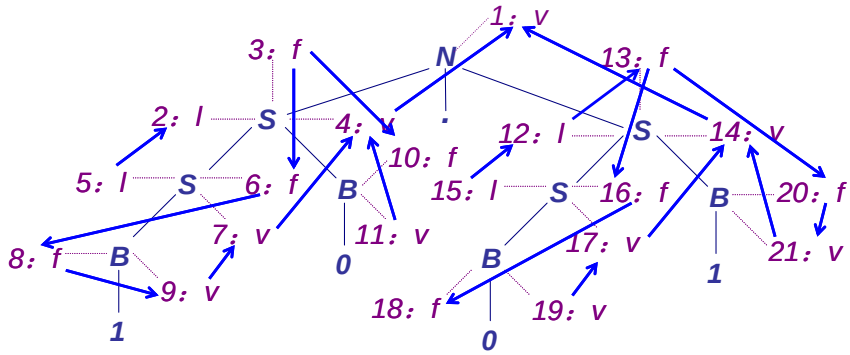

图 5　依赖图的有向边

然后，我们可以判定，图 5 中描述的依赖图是无圈的。接着，我们可以按造这个有向无圈图结点的任何一种拓扑排序来计算所有的属性值。比如，以下结点序列为一种拓扑排序：3，5，2，6，10，8，9，7，11，4，15，12，13，16，20，18，21，19，17，14，1。按照这一次序依次计算各结点对应的属性值，可以得到如图 6 所示的结果，其中每个结点对应的属性取值在离该结点名称最近的方框内给出。

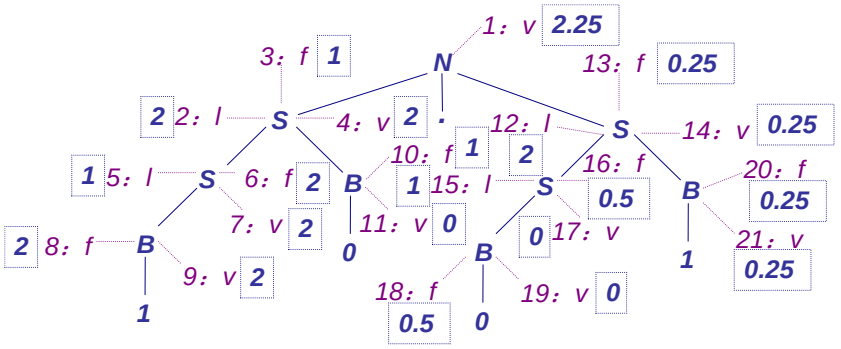

图 6　计算依赖图中各结点对应的属性取值

语法分析树中各结点属性取值的计算过程被称为对语法分析树进行**标注**。可以用**带标注语法分析树**表示属性的计算结果。如，图 6 中的计算结果可以表示为图 7 中的带标注语法分析树。

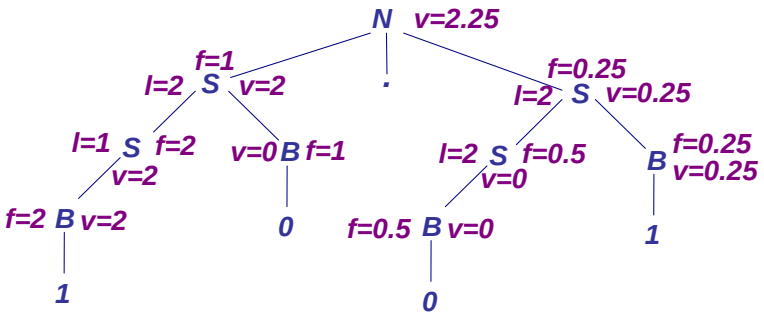

图 7　带标注语法分析树

虽然通过遍历分析树进行属性计算的方法有一定的通用性，但它是在语法分析遍之后进行的，不能体现语法制导方法的优势。实际的编译程序中，语法制导的语义计算大都采取单遍的过程，即语法分析过程的同时就完成相应的语义动作。这样，属性计算仅对应一个自顶向下或是自底向上的简单过程。然而，并非所有属性文法都适合单遍的处理过程，所以在实践中一般会要求对属性文法进行某种限制。随后，我们主要讨论两类受限的属性文法，即 *S*-属性文法和 *L*-属性文法。

1.3　*S*-属性文法和*L*-属性文法

只包含综合属性的属性文法称为 ***S*-属性文法**。

一个属性文法称为 ***L*-属性文法**，如果对其中每一个产生式 $A \to X_1 X_2 ... X_n$，其每个语义动作所涉及的属性或者是综合属性，或者是某个 X_i（$1 \le i \le n$）的继承属性，而这个继承属性只能依赖于：

（1） X_i 左边的符号 $X_1 X_2 ... X_{i-1}$ 的属性；
（2） A 的继承属性。

通俗地说，*L*-属性文法既可以包含综合属性，也可以包含继承属性，但要求产生式右端某文法符号的继承属性的计算只取决于该符号左边符号的属性 （对于产生式左部的符号，只能是继承属性）。

容易看出，*S*-属性文法是 *L*-属性文法的一个特例。

1.4 基于*S*-属性文法的语义计算

由于综合属性是自底向上传递信息，因而基于 *S*-属性文法的语义计算通常采用自底向上的方式进行。

若 *S*-属性文法的基础文法可以采用 LR 分析技术进行语法分析，那么我们可以通过扩充分析栈中的域，形成语义栈来存放综合属性的当前取值，使得分析引擎在每一步归约发生之前的时刻启动并完成产生式左部文法符号综合属性值的计算。附加了语义栈的 LR 分析模型如图 8 所示，其中我们假设初始状态下状态栈、符号栈和语义栈中的内容为“S_0”，“#”和“-”。语义栈中存放的是符号栈中同一位置符号的综合属性值。若该符号有多个属性，可以对应多元组的形式来描述。分析引擎在访问产生式的同时需要执行相应的语义动作

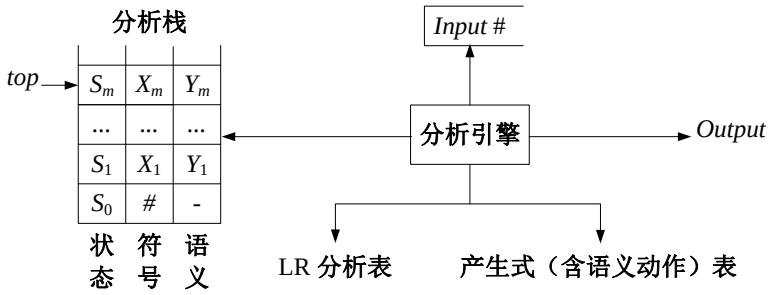

图 8　附加语义栈的 LR 分析模型

在采用 LR 分析技术进行基于 S-属性文法的语义计算时，语义动作中的综合属性总是可以通过存在于当前语义栈顶部的属性值进行计算。例如，假设有相应于产生式 $A \to XYZ$ 的语义动作

$$A.a := f（X.x, Y.y, Z.z）$$

在 XYZ 归约为 A 之前，$Z.z$，$Y.y$，和 $X.x$ 分别存放于语义栈的 top，top-1 和 top-2 的相应域中（top 指向栈顶位置），因此 $A.a$ 可以顺利求出。归约之后，$Z.z$，$Y.y$，和 $X.x$ 被弹出，而在新的栈顶位置（原 top-2 的位置）上存放 $A.a$。

例 5 给定一个简单表达式求值属性文法（开始符号为 S）：

$S \to E$	{ $print(E.val)$ }
$E \to E_1 + T$	{ $E.val := E_1.val + T.val$ }
$E \to T$	{ $E.val := T.val$ }
$T \to T_1 * F$	{ $T.val := T_1.val \times F.val$ }
$T \to F$	{ $T.val := F.val$ }
$F \to (E)$	{ $F.val := E.val$ }
$F \to d$	{ $F.val := d.lexval$ }

其中，$d.lexval$ 是由词法分析程序所确定的属性；$F.val$，$T.val$ 和 $E.val$ 都是综合属性；语义函数 $print(E.val)$ 用于显示 $E.val$ 的结果值。

对于该属性文法的基础文法，我们可构造一个如图 9 所示的 LR 分析表。基于这一 LR 分析表进行自底向上分析，每一步归约的同时执行相应的语义动作。试给出以常量表达式 2+3*5 为输入串的分析过程中，并通过语义栈体现 2+3*5 的求值过程。

状态	ACTION						GOTO		
	d	*	+	(	)	#	*E*	*T*	*F*
0	s5			s4			1	2	3
1			s6			acc			
2		s7	r2		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8			s6		s11				
9		s7	r1		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

图 9　一个 LR 分析表

解　我们把分析栈的元素用三元组形式表示，分别记录状态栈、符号栈和语义栈的内容。图 10 描述了基于图 9 中 LR 分析表的自底向上分析步骤，可以同时体现 2+3∗5 作为输入串的分析过程和语义计算过程（即简单表达式的求值过程）。符号“-”表示未定义的语义值。

步骤	分析栈（状态，符号，语义值）	余留符号串	分析动作	语义动作
0	0 # -	2 + 3 ∗ 5 #	s5	
1	0 # - 5 2 2	+ 3 ∗ 5 #	r6	*F.val* := *d.lexval*
2	0 # - 3 *F* 2	+ 3 ∗ 5 #	r4	*T.val* := *F.val*
3	0 # - 2 *T* 2	+ 3 ∗ 5 #	r2	*E.val* := *T.val*
4	0 # - 1 *E* 2	+ 3 ∗ 5 #	s6	
5	0 # - 1 *E* 2 6 + -	3 ∗ 5 #	s5	
6	0 # - 1 *E* 2 6 + - 5 3 3	∗ 5 #	r6	*F.val* := *d.lexval*
7	0 # - 1 *E* 2 6 + - 3 *F* 3	∗ 5 #	r4	*T.val* := *F.val*
8	0 # - 1 *E* 2 6 + - 9 *T* 3	∗ 5 #	s7	
9	0 # - 1 *E* 2 6 + - 9 *T* 3 7 ∗ -	5 #	s5	
10	0 # - 1 *E* 2 6 + - 9 *T* 3 7 ∗ - 5 5 5	#	r6	*F.val* := *d.lexval*
11	0 # - 1 *E* 2 6 + - 9 *T* 3 7 ∗ - 10 *F* 5	#	r3	$T.val := T_1.val \times F.val$
12	0 # - 1 *E* 2 6 + - 9 *T* 15	#	r1	$E.val := E_1.val + T.val$
13	0 # - 1 *E* 17	#	acc	*print*(*E.val*)

图 10　LR 分析过程中同时进行语义计算

1.5 基于*L*-属性文法的语义计算

对于 *L*-属性文法，我们先来说明：可以采用自顶向下深度优先从左至右遍历分析树的方法计算所有属性值。如图 11 描述了这样一种计算过程。

```
function visit(n: node);
  begin
    for n 的每一孩子m, 从左到右 do
        begin
          计算 m 的继承属性值;      /*只依赖于 m 左边兄弟的属性或 n 的继承属性*/
          visit(m)
        end;
        计算 n 的综合属性值
  end
```

图 11　深度优先从左至右遍历计算属性值（适用于 *L*-属性文法）

这一计算过程的核心是：某一节点的继承属性只依赖于该节点左边兄弟的属性（综合属性或继承属性），或者其父亲节点的继承属性。*L*-属性文法的特性能够保证这一点。

下面，我们通过具体例子来理解这一过程。

例 6 下面的属性文法可用于将二进制无符号定点小数转化为十进制小数（开始符号为 N）：

(1)	$N \to .S$	{ $S.f := 1$； $print(S.v)$ }
(2)	$S \to BS_1$	{ $S_1.f := S.f + 1$; $B.f := S.f$; $S.v := B.v + S_1.v$ }
(3)	$S \to \varepsilon$	{ $S.v := 0$ }
(4)	$B \to 0$	{ $B.v := 0$ }
(5)	$B \to 1$	{ $B.v := 2^{-B.f}$ }

其中，各个属性的含义为：

- 符号 S 的继承属性 f 表示 S 推导的 0、1 串中第一位为 1 时应该对应的十进制数值为 2^{-f}。从属性文法的第一行中的 $S_1.f := 1$ 和第二行中的 $S_1.f := S.f + 1$ 可知：小数点后第一位数为 1 时对应的十进制数值为 $2^{-1} = 0.5$，小数点后第二位数为 1 时对应的十进制数值为 $2^{-2} = 0.25$，小数点后第三位数为 1 时对应的十进制数值为 $2^{-3} = 0.125$，等等。
- 符号 S 的综合属性 v 表示 S 推导的 0、1 串对应的十进制数值。
- 符号 B 的继承属性 f 表示二进制数的当前这一位数字是 1 时应该对应的十进制数值为 2^{-f}，其含义类似于符号 S 的继承属性 f。
- 符号 B 的综合属性 v 表示二进制数的当前这一位数字（0 或 1）对应的十进制数值。

容易看出，这是一个 *L*-属性文法。针对输入串 .101 的分析树，给出采用图 11 所描述的方法计算所有属性值的过程。

解　如图 12 所示，采用图 11 所述方法计算属性值的过程是在深度优先从左至右遍历分析树

的同时进行属性计算。直观上可以将其分解为自顶向下计算继承属性值的过程，以及自底向上计算综合属性值的过程。

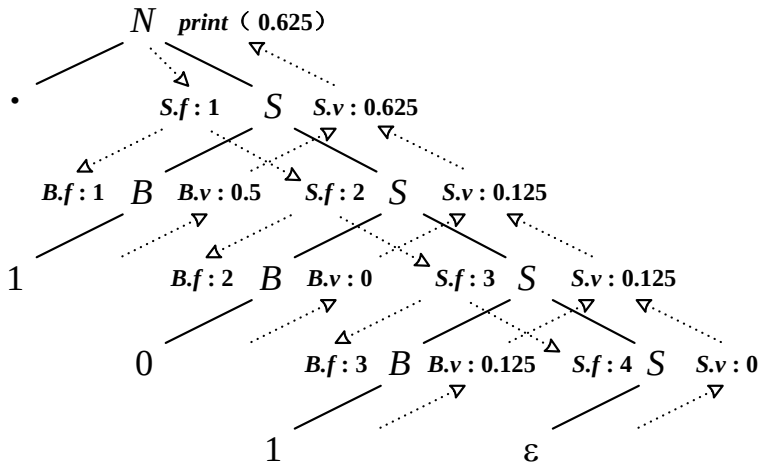

图 12　通过深度优先后序遍历分析树计算属性值的例子

容易看出，可以把图 11 所描述的计算过程与自顶向下预测分析过程完全对应起来。对于例 6，我们可以把图 12 所描述的语义计算过程穿插到到使用下推栈的表驱动预测分析过程中[1]，如图 13 所示（稍后解释）。

如果将图 13 中所穿插的语义计算步骤以及语义信息全部去掉，则可以得到图 14（为方便对照，沿用了图 13 的步骤编号）。对于例 6 的属性文法，我们容易验证其基础文法是 LL(1) 文法。事实上，图 14 描述了根据该 LL(1) 文法针对输入串 .101 的预测分析过程。

换句话说，图 13 是在图 14 的预测分析过程中附加了语义计算信息：

- 我们将 5 个产生式对应的自底向上计算（或综合属性求值）的语义动作分别表示为①~⑤（推广到一般情形，每个产生式可对应一个自底向上计算的语义动作集合）。同时，我们将该属性文法中用于继承属性求值的语义动作分别表示为 ⑴~⑶（推广到一般情形，可以是语义动作集合，因为每个产生式中同一个位置的文法符号可以有多个继承属性的计算）。

- 每当开始使用一个产生式的推导步骤时，先是将该产生式左部的非终结符（即距离栈顶最近的文法符号）从下推栈弹出；紧接着，先将自底向上计算的语义动作入栈，以备当前产生式对应的分析子过程结束后执行；之后，再将产生式右部的符号从右到左依次入栈，在每个符号入栈后紧接着也将其对应的用于继承属性求值的语义动作入栈。比如，图 13 中的步骤 4、10 和 16 使用了例 6 中属性文法的产生式（2），在各自的下一步中：首先，从下推栈弹出 S；紧接着，将语义动作 ②（$S.v := B.v + S_1.v$）入栈；之后，再将 S，语义动作 ⑵（$S_1.f := S.f + 1$），B，以及语义动作 ⑶（$B.f := S.f$）依次入栈。

- 此外，图 13 还维护一个属性栈。每当开始某个产生式的推导步，就将该产生式涉及到的所有属性列表以占位符形式作为初始取值入栈。比如，对于例 6 中属性文法的产生式（2），我们设属性列表为（$S.f, S.v, B.f, B.v$）；相应的初始取值用（-, -, -, -）

[1] 本课程目前暂不要求掌握这部分内容，可作为课后阅读内容

表示。又如，对于产生式（5），我们设属性列表为（*B.f, B.v*）；相应的初始取值用（-, -）表示。对于其他产生式的情形，读者可从图 13 推断出来。

步骤	下推栈	属性栈	余留串	下一推导步/语义动作
1	# *N*	( ) #	. 1 0 1 #	$N \to .S$ {□ ; □ }
2	# □ *S* □ .	(-,-) ( ) #	. 1 0 1 #	匹配栈顶和当前输入符号
3	# □ *S* □	(-,-) ( ) #	1 0 1 #	执行 □
4	# □ *S*	(1,-) ( ) #	1 0 1 #	$S \to BS_1$ {□ ; □ ; □ }
5	# □ □ *S* □ *B* □	(-,-,-,-) (1,-) ( ) #	1 0 1 #	执行 □
6	# □ □ *S* □ *B*	(-,-,1,-) (1,-) ( ) #	1 0 1 #	$B \to 1$ {□ }
7	# □ □ *S* □ □ 1	( ) (-,-,1,-) (1,-) ( ) #	1 0 1 #	匹配栈顶和当前输入符号
8	# □ □ *S* □ □	(-,-,1,-) (1,-) ( ) #	0 1 #	执行□
9	# □ □ *S* □	(-,-,1,0.5) (1,-) ( ) #	0 1 #	执行□
10	# □ □ *S*	(2,-,1,0.5) (1,-) ( ) #	0 1 #	$S \to BS_1$ {□ ; □ ; □ }
11	# □ □ □ *S* □ *B* □	(-,-,-,-) (2,-,1,0.5) (1,-) ( ) #	0 1 #	执行 □
12	# □ □ □ *S* □ *B*	(-,-,2,-) (2,-,1,0.5) (1,-) ( ) #	0 1 #	$B \to 0$ {□ }
13	# □ □ □ *S* □ □ 0	( ) (-,-,2,-) (2,-,1,0.5) (1,-) ( ) #	0 1 #	匹配栈顶和当前输入符号
14	# □ □ □ *S* □ □	(-,-,2,-) (2,-,1,0.5) (1,-) ( ) #	1 #	执行□
15	# □ □ □ *S* □	(-,-,2,0) (2,-,1,0.5) (1,-) ( ) #	1 #	执行□
16	# □ □ □ *S*	(3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	1 #	$S \to BS_1$ {□ ; □ ; □ }
17	# □ □ □ □ *S* □ *B* □	(-,-,-,-) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	1 #	执行 □
18	# □ □ □ □ *S* □ *B*	(-,-,3,-) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	1 #	$B \to 1$ {□ }
19	# □ □ □ □ *S* □ □ 1	( ) (-,-,3,-) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	1 #	匹配栈顶和当前输入符号
20	# □ □ □ □ *S* □ □	(-,-,3,-) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	#	执行□
21	# □ □ □ □ *S* □	(-,-,3,0.125) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	#	执行□
22	# □ □ □ □ *S*	(4,-,3,0.125) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	#	$S \to \varepsilon$ {□ }
23	# □ □ □ □ □	( ) (4,-,3,0.125) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	#	执行□
24	# □ □ □ □	(4,0,3,0.125) (3,-,2,0) (2,-,1,0.5) (1,-) ( ) #	#	执行□
25	# □ □ □	(3,0.125,2,0) (2,-,1,0.5) (1,-) ( ) #	#	执行□
26	# □ □	(2,0.125,1,0.5) (1,-) ( ) #	#	执行□
27	# □	(1,0.625) ( ) #	#	执行□
28	#	( ) #	#	成功返回

注：自下而上执行的语义动作 ① *print*(*S.v*) ； ② $S.v := B.v + S_1.v$ ； ③ $B.v := 2^{-B.f}$； ④ $B.v := 0$ ； ⑤ $S.v := 0$
自上而下执行的语义动作 □ $S.f := 1$； □ $S_1.f := S.f + 1$； □ $B.f := S.f$

图 13　L 属性文法的预测分析过程中穿插语义计算

- 当下推栈栈顶遇某个语义动作 ⑴~⑶，则在当前属性栈栈顶所表示的环境下执行这个语义动作，执行结果修改这个环境下对应的属性值。如在步骤 3，执行语义动作 ⑴ ($S.f := 1$)，执行结果属性栈栈顶从 (-,-) 变为 (1,-)，同时使⑴出栈。

- 当下推栈栈顶遇某个语义动作 ①~⑤，则在当前属性栈栈顶所表示的环境下执行这个语义动作，弹出这个栈顶，并根据执行结果修改新栈顶所表示的环境下对应的属

性值。如在步骤 26，执行语义动作 ②（$S.v := B.v + S_1.v$），执行结果为 $S.v := 0.125 + 0.5 = 6.125$，属性栈中将(2,0.125,1,0.5)弹出，新栈顶从 (1,-) 变为 (1,0.625)，同时使 ② 出栈。

步骤	下推栈	余留串	下一推导步
1	# N	.101#	$N \to .S$
2	#S.	.101#	匹配栈顶和当前输入符号
4	#S	101#	$S \to BS$
6	#SB	101#	$B \to 1$
7	#S1	101#	匹配栈顶和当前输入符号
10	#S	01#	$S \to BS$
12	#SB	01#	$B \to 0$
13	#S0	01#	匹配栈顶和当前输入符号
16	#S	1#	$S \to BS$
18	#SB	1#	$B \to 1$
19	#S1	1#	匹配栈顶和当前输入符号
22	#S	#	$S \to \varepsilon$
28	#	#	成功返回

图 14　与 L 属性文法的语义计算过程对应的预测分析过程

图 13 所描述的过程是将语义计算穿插到到表驱动预测分析过程中。实现 LL(1)预测分析的另一种方法是采用递归下降分析程序。我们可以很方便地对递归下降分析程序进行改造，使其同时具有语义计算的能力。由于随后讨论的翻译模式更方便实现，我们届时再考虑递归下降分析程序的改造，设计思想是一致的。

另外，对于某些 *L*-属性文法也可以设计基于 LR 分析的自底向上处理过程。同样，我们将在随后介绍的翻译模式基础上对此进行讨论。

2 基于翻译模式的语义计算

2.1 翻译模式

翻译模式是适合语法制导语义计算的另一种描述形式，它可以体现一种合理调用语义动作的算法。**翻译模式**在形式上类似于属性文法，但允许由{}括起来的语义动作出现在产生式右端的任何位置，以此显式地表达属性计算的次序。

类似于属性文法的情形，在设计翻译模式时也需要进行某些限定，以确保每个属性值在一个单遍的语义计算过程中被访问到的时候已经存在。与 *S*-属性文法和 *L*-属性文法相对应，我们仅讨论两类受限的翻译模式：

- ***S*-翻译模式**：是一种仅涉及综合属性的情形，通常将语义动作集合置于相应产生式右端的末尾。

- ***L*-翻译模式**：既可以包含综合属性，也可以包含继承属性，但需要满足：（1）产生式右端某个符号继承属性的计算必须位于该符号之前，其语义动作不访问位于它右边符号的属性，只依赖于该语义动作左边符号的属性 （对于产生式左部的符号，只能是继承属性）；（2）产生式左部非终结符的综合属性的计算只能在所用到的属性都已计算出来之后进行，通常将相应的语义动作置于产生式的尾部。

显然，*S*-翻译模式是 *L*-翻译模式的特例。

例 7 对于语言 $L = \{ a^n b^n c^n \mid n \geq 1\}$，我们可以设计如下翻译模式：

$S \rightarrow A$
{ *B.in_num* := *A* .*num* } *B*
{ *C.in_num* := *A* .*num* } *C*
{ if (*B.num=0 and C.num=0*)
then *print*(*"Accepted!"*)
else *print*(*"Refused!"*) }

$A \rightarrow A_1 a$ { *A.num* := A_1 . *num* + 1 }

$A \rightarrow a$ { *A.num* := 1 }

$B \rightarrow$ { B_1.*in_num* := *B* . *in_num* } $B_1\, b$ { *B.num* := B_1 . *num* - 1 }

$B \rightarrow b$ { *B.num* := B_1 . *in_num* - 1}

$C \rightarrow$ { C_1.*in_num* := *C* . *in_num* } $C_1\, c$ { *C.num* := C_1 . *num* - 1 }

$C \rightarrow c$ { *C.num* := C_1 . *in_num* - 1}

容易看出，这是一个 *L*-翻译模式。其中，*num* 为综合属性；*in_num* 为继承属性。

对照例2的属性文法，例7的翻译模式描述了相同的语义计算效果：若输入串属于 *L*，则语义计算结果会执行 *print* (“*Accepted*!”)，否则，将会执行 *print* (“*Refused*!”) 或者报告语法错误。不同的是，后者显式描述了属性计算的次序（即确定了语义动作执行的位置）。

2.2 基于*S*-翻译模式的语义计算

S-翻译模式在形式上与 *S*-属性文法是一致的，可以采取同样的语义计算方法。类似 1.4 节的讨论，基于 *S*-翻译模式的语义计算一般基于自底向上分析过程，通过增加存放属性值域的语义栈来实现。这里，我们不再赘述这一过程。

在这一小节里，为了进一步解释这种基于自底向上分析（如 LR 分析）的语义计算过程，将要讨论每个产生式归约时需要执行的语义计算代码片断，特别是对语义栈上操作的描述。为此，我们假设语义栈由向量 *v* 表示，归约前栈顶位置为 *top*，栈上第 *i* 个位置所对应符号的综合属性值 *x* 用 *v*[*i*]. *x* 表示。

例如，假设一个 *S*-翻译模式中有如下产生式

$A \rightarrow XYZ$ { *A.a* := *f*（*X.x, Y.y, Z.z*）; … }

为了明确表达语义栈上的操作，我们将这个产生式变换为

$A \rightarrow XYZ$ { *v*[*top*-2].*a* := *f*（*v*[*top*-2].*x*, *v*[*top*-1].*y*, *v*[*top*].*z*）; … }

这里，*top* 为归约前栈顶位置，归约后 *top* 的取值将由分析引擎自动维护。

例 8 给定下列 *S*-翻译模式（同例 5 的 *S*-属性文法，为方便对照，这里我们重复给出）：

$S \to E$ { *print*(*E.val*) }
$E \to E_1 + T$ { *E.val* := E_1.*val* + *T.val* }
$E \to T$ { *E.val* := *T.val* }
$T \to T_1 * F$ { *T.val* := T_1.*val* × *F.val* }
$T \to F$ { *T.val* := *F.val* }
$F \to (E)$ { *F.val* := *E.val* }
$F \to d$ { *F.val* := *d.lexval* }

如果在 LR 分析过程中根据该翻译模式进行自底向上语义计算，试写出在按每个产生式归约时实现语义计算的一个代码片断，可以体现语义栈上的操作（设语义栈由向量 *v* 表示，归约前栈顶位置为 *top*，不用考虑对 *top* 的维护）。

解 我们可以将这个翻译模式变换为

$S \to E$ { *print*(*v*[*top*].*val*) }
$E \to E_1 + T$ { *v*[*top*-2].*val* := *v*[*top*-2].*val* + *v*[*top*].*val* }
$E \to T$ { *v*[*top*].*val* := *v*[*top*].*val* } // 相当于 { }
$T \to T_1 * F$ { *v*[*top*-2].*val* := *v*[*top*-2].*val* × *v*[*top*].*val* }
$T \to F$ { *v*[*top*].*val* := *v*[*top*].*val* } // 相当于 { }
$F \to (E)$ { *v*[*top*-2].*val* := *v*[*top*-1].*val* }
$F \to d$ { *v*[*top*].*val* := *d.lexval* }

2.3 基于*L*-翻译模式的自顶向下语义计算

与 *L*-属性文法相比，*L*-翻译模式已经规定好了产生式右端文法符号和语义动作（即属性计算）的处理次序，这可以在很大程度上简化语义计算程序的设计。

根据 1.5 节的讨论，图 11 描述的过程可用于基于 *L*-属性文法的语义计算，同样也适用于基于 *L*-翻译模式的语义计算。这一计算过程可与自顶向下预测分析过程完全对应起来。1.5 节中，我们通过例子（例 6）介绍了将 *L*-属性文法描述的语义计算融入到 LL(1)预测分析过程之中。这一小节里，我们将以递归下降分析程序的改造为例，介绍将 *L*-翻译模式描述的语义计算过程融入其中的方法。同样，我们假定所讨论的 *L*-翻译模式的基础文法是 LL(1)文法。

在递归下降 LL(1)分析程序的设计中，每个非终结符都对应一个分析子函数（过程），分析程序从文法开始符号所对应的分析子函数开始执行。分析子函数可以根据下一个输入符号来确定自顶向下分析过程中应该使用的产生式，并根据所选定的产生式右端依次出现的符号来设计其行为：

- 每遇到一个终结符，则判断当前读入的单词符号是否与该终结符相匹配，若匹配，则继续读取下一个下一个输入符号；若不匹配，则报告和处理语法错误。
- 每遇到一个非终结符，则调用相应的分析子程序。

对递归下降 LL(1)分析程序进行改造的核心思想是扩展各个分析子函数的定义。假设已为非终结符 *A* 构造了一个分析子函数。现在，只需对这个分析子函数的定义作如下约定：

以 A 的每个继承属性为形参，以 A 的综合属性为返回值（若有多个综合属性，可返回记录类型的值）。相应于分析子函数的设计，改造后子函数代码的流程也是根据当前的输入符号来决定调用哪个产生式，与每个产生式对应的代码同样也是根据该产生式右端的结构来构造（不同之处是要将语义动作嵌入其中），具体可描述为：

- 若遇到一个终结符 X，首先将其综合属性 x 的值保存至专为 $X.x$ 而声明的变量；然后，判断当前读入的输入符号是否与该终结符相匹配，若匹配，则继续读取下一个输入符号；若不匹配，则报告和处理语法错误。
- 若遇到一个非终结符 B，利用相应于 B 的子函数 $ParseB$ 产生赋值语句 $c := ParseB(b_1, b_2, \ldots, b_k)$，其中参量 $b_1, b_2, \ldots, b_k$ 对应于 B 的各继承属性，变量 c 对应 B 的综合属性（若有多个综合属性，则可使用记录类型的变量）。
- 若遇到一个语义动作集合，则直接复制其中每一语义动作所对应的代码，只是需要注意将属性的访问替换为相应变量的访问。

我们称改造后的分析子函数（过程）为语义计算子函数（过程），并称改造后的递归下降分析程序称为**递归下降（预测）语义计算程序**，或**递归下降（预测）翻译程序**。

例 9 下面的翻译模式可用于将二进制无符号定点小数转化为十进制小数（开始符号为 N）：

$N \to .\quad \{ S.f := 1 \}\quad S\quad \{ print(S.val) \}$
$S \to \{ B.f := S.f \}\quad B\quad \{ S_1.f := S.f + 1 \}\quad S_1\quad \{ S.val := B.val + S_1.val \}$
$S \to \varepsilon\quad \{ S.val := 0 \}$
$B \to 0\quad \{ B.val := 0 \}$
$B \to 1\quad \{ B.val := 2\text{^}(-B.f) \}$

其中，各个属性的含义同例 6。对于该 L-翻译模式，试构造相应的递归下降翻译程序。

解 该 L-翻译模式的基础文法为：

$N \to .\ S$
$S \to B\ S_1$
$S \to \varepsilon$
$B \to 0$
$B \to 1$

可以验证该文法为 LL(1) 文法。

针对该文法，我们可构造一个递归下降 LL(1) 分析程序，其中：

开始符号 N 对应的分析子函数为：

```
void ParseN( )
{
    MatchToken('.');          // 匹配 '.'
    ParseS( );
}
```

非终结符 S 对应的分析子函数为：

```
void ParseS( )
```

```
{
    if (lookahead == ’0’ or lookahead == ’1’ ) {    // lookahead: 当前扫描的输入符号
        ParseB( );
        ParseS( );
    }
    else if (lookahead == ‘#’ ) { }        // ‘#’为输入结束符
    else { printf("syntax error \n"); exit(0); }
}
```

非终结符 B 对应的分析子函数为：

```
void ParseB( )
{
    if (lookahead == ’0’) {
        MatchToken(’0’);            // 匹配 ’0’
    }
    else if    (lookahead == ’1’)    {
        MatchToken(’1’);
    }
    else { printf("syntax error \n"); exit(0); }
}
```

上面的函数 MatchToken 用于判别正在处理的终结符与当前输入符号是否匹配。若匹配，则读取狭义输入符号，继续分析过程；若不匹配，则报告语法错误，并退出。以下是 MatchToken 函数的一种简单的设计：

```
void MatchToken(int expected)
{
    if (lookahead != expected)     //判别当前输入符号是否与期望的终结符匹配
    {
        printf("syntax error \n");   //若不匹配，则报告出错信息，跳出
        exit(0);
    }
   else                                  //若匹配，消费掉当前输入符号
        lookahead = getToken();     //并向词法分析程序申请并读入下一个输入符号
}
```

其中，lookahead 为全局量，存放下一个输入符号。

下面，我们对这一递归下降分析程序根据翻译模式进行改造，得到递归下降（预测）翻译程序。为此，我们将每个非终结符对应的分析子函数改造为语义计算子函数。

根据以下产生式

$$N \to .\quad \{ S.f := 1 \}\quad S\quad \{ print(S.val) \}$$

开始符号 N 对应的语义计算子函数可以设计为：

```
void ParseN( )
```

```
{
    MatchToken('.');          // 匹配 '.'
    Sf : =1;                  // 变量 Sf 对应属性 S.f
    Sv : = ParseS(Sf);        // 变量 Sv 对应属性 S.val
    print(Sv);
}
```

根据以下产生式

$S \rightarrow \{\ B.f := S.f\ \}\quad B\quad \{\ S_1.f := S.f + 1\ \}\quad S_1\quad \{\ S.val := B.val + S_1.val\ \}$

$S \rightarrow \varepsilon\quad \{\ S.val := 0\ \}$

非终结符 S 对应的语义计算子函数可以设计为：

```
float ParseS( int f )
{
    if (lookahead == '0' or lookahead == '1' )   { // lookahead: 当前扫描的单词符号
        Bf : = f;                                // 变量 Bf 对应属性 B.f
        Bv : = ParseB(Bf);                       // 变量 Bv 对应属性 B.val
        S1f := f+1 ;                             // 变量 S1f 对应属性 S1.f
        S1v := ParseS(S1f);                      // 变量 S1v 对应属性 S1.val
        Sv := S1v + Bv;
    }
    else if (lookahead == '#' )
        Sv := 0;
    else { printf("syntax error \n"); exit(0); }
    return Sv;
}
```

根据以下产生式

$B \rightarrow 0\quad \{\ B.val := 0\ \}$

$B \rightarrow 1\quad \{\ B.val := 2\text{^}(\text{-}B.f)\ \}$

非终结符 B 对应的语义计算子函数可以设计为：

```
float ParseB( int f )
{
    if (lookahead == '0')   {
        MatchToken('0');
        Bv : = 0
    }
    else if   (lookahead == '1')   {
        MatchToken('1');
        Bv := 2^(-f)
    }
    else { printf("syntax error \n"); exit(0); }
    return Bv;
```

}

如果基础文法不是 LL(1) 文法，则不能套用这种模式。比如，例 8 中的 *S*-翻译模式（当然也是 *L*-翻译模式）是常用于定义常量表达式求值的翻译模式，但其基础文法含有左递归，因而不能用 LL（1）方法。有时，若消除某个文法的左递归后，则有可能使得该文法成为 LL(1) 文法。若需要消除翻译模式之基础文法中的左递归，那么翻译模式应该如何变化呢？下面介绍一种较简单但常用的一种情形。

假设有如下翻译模式：

$A \to A_1 Y$ { $A.a := g(A_1.a, Y.y)$ }
$A \to X$ { $A.a := f(X.x)$ }

消去关于 A 的直接左递归，基础文法变换为

$A \to X\ R$
$R \to Y\ R \mid \varepsilon$

再考虑语义动作，翻译模式可变换为

$A \to X$ { $R.i := f(X.x)$ } R { $A.a := R.s$ }
$R \to Y$ { $R_1.i := g(R.i, Y.y)$ } R_1 {$R.s := R_1.s$}
$R \to \varepsilon$ {$R.s := R.i$}

变换前后代表两种不同的计算方式，如图 15 所示。

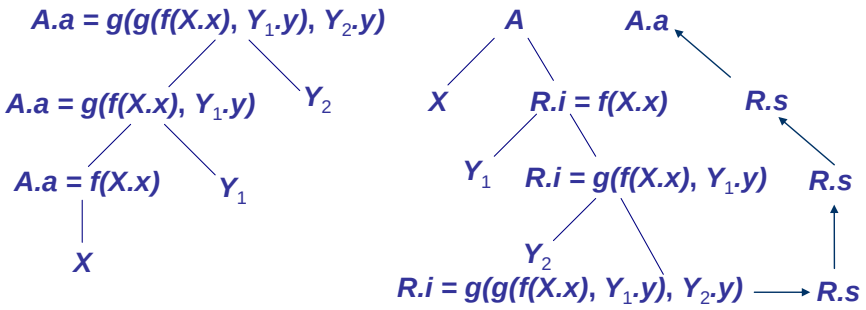

图 15　消除基础文法左递归前后代表两种不同的计算方式

例如，例 8 中的 *S*-翻译模式经消除基础文法左递归后，可变化为如下 *L*-翻译模式：

$S \to E$ { $print(E.val)$ }
$E \to T$ { $R.i := T.val$ } R { $E.val := R.s$ }
$R \to + T$ { $R_1.i := R.i + T.val$ } R_1 { $R.s := R1.s$ }
$R \to \varepsilon$ { $R.s := R.i$ }
$T \to F$ { $P.i := F.val$ } P { $T.val := P.s$ }
$P \to * F$ { $P_1.i := P.i \times F.val$ } P_1 { $P.s := P1.s$ }
$P \to \varepsilon$ { $P.s := P.i$ }
$F \to (E)$ { $F.val := E.val$ }
$F \to d$ { $F.val := d.lexval$ }

此时，便可以套用以上递归下降翻译程序的模式了。

2.4 基于*L*-翻译模式的自底向上语义计算

L-翻译模式中既有继承属性也有综合属性。综合属性是自底向上传递信息，因而在自底向上的语义计算中，可以将文法符号的综合属性值存放于语义栈中。因此，若 *L*-翻译模式中不包含继承属性，我们就可以采用 1.4 或 2.2 小节所述的方法实现自底向上的语义计算。此时，我们只需处理好嵌入在产生式中间的语义动作。对此，一种处理方法是：引入新的非终结符 N 和产生式 $N\to\varepsilon$；把嵌入在产生式中间的语义动作集用非终结符 N 代替，并把该语义动作集放在产生式 $N\to\varepsilon$ 后面。由于语义动作集中未关联任何继承属性，所以翻译模式经过这样的变换后实际上就可以看作 *S*-属性文法（翻译模式）来处理了。

例如，对于下列翻译模式：

$E \to T\ R$
$R \to +\ T \quad \{\ print(\text{'+'})\ \}\ R_1$
$R \to -\ T \quad \{\ print(\text{'-'})\ \}\ R_1$
$R \to \varepsilon$
$T \to \underline{num} \quad \{\ print(\underline{num}.val)\ \}$

可以变换为

$E \to T\ R$
$R \to +\ T\ M\ R_1$
$R \to -\ T\ N\ R_1$
$R \to \varepsilon$
$T \to \underline{num} \quad \{\ print(\underline{num}.val)\ \}$
$M \to \varepsilon \quad \{\ print(\text{'+'})\ \}$
$N \to \varepsilon \quad \{\ print(\text{'-'})\ \}$

然而，若语义动作集中关联有继承属性，情况会复杂一些。为此，我们需要考虑如何处理针对继承属性的求值和访问。由于在自底向上的语义计算中，语义栈中只可能存放综合属性值，所以在设计语义计算程序时应注意的一个原则是：继承属性的求值结果必须要以某个综合属性值存放于语义栈中，而继承属性的访问也要最终落实到对某个综合属性值的访问。

首先，我们来讨论一下继承属性的求值。一种简单情况是，继承属性是通过复写规则以某个综合属性直接定义的。例如，在自底向上语义计算程序根据产生式 $A\to XYZ$ 的归约过程中，假设 X 的综合属性值 $X.s$ 已经出现在语义栈上。因为在根据句柄 XYZ 进行归约之前，$X.s$ 的值一直存在，因此它可以被 Y 以及 Z 继承。如果用复写规则 $Y.i := X.s$ 来定义 Y 的继承属性 $Y.i$，则在需要 $Y.i$ 时，可以通过访问 $X.s$ 来实现。较之复杂一点的情况是，继承属性是间接地通过复写规则用某个综合属性来定义的。比如，用复写规则 $Z.i := Y.i$ 来定义 Z 的继承属性 $Z.i$，则在需要 $Z.i$ 时，也可以通过访问 $X.s$ 来实现。

若一个继承属性是通过普通函数而不是通过复写规则定义的，那么应该如何处理呢？考虑某个翻译模式的如下产生式规则：

$S \to a\ A \quad \{\ C.i := f(A.s)\ \} \quad \text{C}$

这里，继承属性 $C.i$ 不是通过复写规则，而是通过普通函数 $f(A.s)$ 来求值的。在计算 $C.i$ 时，$A.s$ 在语义栈上，但 $f(A.s)$ 并未存在于语义栈。一种处理方法是引入新的非终结符

号，比如 M，将以上产生式规则改造为：

$$S \to a\ A\ \{\ M.i := A.s\ \}\ M\ \{\ C.i := \mathrm{M.s}\ \}\ C$$
$$M \to \varepsilon\ \{\ M.s := f(M.i)\ \}$$

这样，就解决了上述问题。想一想，为什么？

其次，我们再进一步讨论一下继承属性的访问。根据刚才对复写规则的讨论，对继承属性的访问终究需要归结到访问某个综合属性。此时我们需要解决好的一个设计问题就是要避免不一致访问。考虑如下翻译模式：

$$S \to a\ A\ \{\ C.i := A.s\ \}\ C \mid b\ A\ B\ \{\ C.i := A.s\ \}\ C$$
$$C \to c\ \{C.s := g(C.i)\}$$

这里出现的问题是：在使用 $C \to c$ 进行归约时，$C.i$ 的值或存在于次栈顶（top-1），或存在于次次栈顶（top-2），不能确定用哪一个。一种可行的做法是引入新的非终结符 M，将以上翻译模式改造为：

$$S \to a\ A\ \{\ C.i := A.s\ \}\ C \mid b\ A\ B\quad \{\ M.i := A.s\ \}\ M\ \{\ C.i := M.s\ \}\ C$$
$$C \to c\ \{C.s := g(C.i)\}$$
$$M \to \varepsilon\ \{\ M.s := M.i\ \}$$

这样，在使用 $C \to c$ 进行归约时，$C.i$ 的值就确定地可以通过访问次栈顶（top-1）得到。

通常情况下，我们可以先考虑解决继承属性的普通函数求值问题，再解决其访问一致性。

实际中所遇到的情况可能会更加复杂。然而，无论采取何种方法来解决继承属性的访问和求值问题，我们的目标是：通过变换翻译模式（如增加新的文法符号，增加相应的复写规则和产生式），使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语法分析过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的综合属性唯一确定地进行访问。

在变换翻译模式时，还需要注意的是：不可以改变 L-翻译模式的特性。若不是 L-翻译模式，则不能保证归约前需要访问的综合属性已出现在分析栈中。

若变换 L-翻译模式后可以达到上述目标，那么就可以基于这个 L-翻译模式进行自底向上的语义计算了。此时，我们可以如 2.2 小节那样，给出每个产生式归约时需要执行的语义计算代码片断。下面我们通过一个例子来结束这一小节的讨论。

例 10 例 9 给出的 L-翻译模式可用于将二进制无符号定点小数转化为十进制小数。

（1） 变换该翻译模式，使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语义计算过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问。

（2） 如果在 LR 分析过程中根据该翻译模式进行自底向上的语义计算，试写出在按每个产生式归约时实现语义计算的一个代码片断，可以体现语义栈上的操作。设语义栈由向量 v 表示，归约前栈顶位置为 top，终结符没有语义值；而每个非终结符的综合属性都只对应一个语义值，用 $v[i].val$ 表示；不用考虑对 top 的维护。

（3） 根据（2）所得到的 L-翻译模式，试给出以 .101 为输入串的 LR 分析过程和语

义计算过程，即给出状态栈、符号栈以及语义栈的变化情况。

解 （1） 对于该翻译模式，只需引入新的非终结符，将第一行的 {*S.f* :=1} 和第二行的 {S_1.*f* := *S.f* + 1} 进行变换，使翻译模式只含复写规则。以下是一个变换结果：

（1）$N \to$. *M* { *S.f* : = *M.s* } *S* { *print*(*S.val*) }
（2）$S \to$ { *B.f* : =*S.f* } *B* { *P.i* :=*S.f* } *P* { S_1.*f* := *P.s* } S_1 { *S.val* := *B.val* + S_1.*val* }
（3）$S \to \varepsilon$ { *S.val* := 0 }
（4）$B \to 0$ { *B.val* := 0 }
（5）$B \to 1$ { *B.val* := 2^(-*B. f*)}
（6）$M \to \varepsilon$ { *M.s* :=1}
（7）$P \to \varepsilon$ { *P.s* := *P.i* +1 }

（2） 只需为每个产生式末尾的语义动作给出语义计算的代码片断，而产生式中间的复写规则只是在确定继承属性对应的综合属性在栈中的位置时会用到。以下是一个变换结果：

（1）$N \to$. *M S* { *print*(*v*[*top*].*val*) }
（2）$S \to B\ P\ S_1$ { *v*[*top*-2].*val* := *v*[*top*-2].*val* + *v*[*top*].*val* }
（3）$S \to \varepsilon$ { *v*[*top*+1].*val* := 0 }
（4）$B \to 0$ { *v*[*top*].*val* := 0 }
（5）$B \to 1$ { *v*[*top*].*val* := 2^(-*v*[*top*-1].*s*) }
（6）$M \to \varepsilon$ { *v*[*top*+1].*s* :=1}
（7）$P \to \varepsilon$ { *v*[*top*+1].*s* := *v*[*top*-1].*s* +1 }

对于这个结果，我们有选择地进行一些解释：按产生式（2）归约前 *top*、*top*-1 和 *top*-2 的位置分别对应符号 *S*、*P* 和 *B*，归约后新栈顶的位置是原来 *top*-2 的位置，栈顶符号变为 *S*；按产生式（3）归约后，在原栈顶 *top* 的上一个位置 *top*+1 压入符号 *S*，其属性值 *val* 被置为 0；按产生式（5）归约前，位于原栈顶 *top* 的符号是 1，继承属性 *B.f* 对应的综合属性位置可跟踪产生式（2）和（1）得到，是位于 *top*-1 位置上的 *M.s* 或 *P.s*（都可以用 *v*[*top*-1].*s* 访问）；产生式（7）中的继承属性 *P.i* 的综合属性位置同样也可跟踪产生式（2）和（1）得到，是位于 *top*-1 位置上的 *M.s* 或 *P.s*，可以用 *v*[*top*-1].*s* 访问。

（3） 可以验证，对于由（2）所得到的 *L*-翻译模式，其基础文法是 LR 文法，图 16 是基于该文法的一个 LR 分析表。若以 .101 作为输入串，则 LR 分析过程和语义计算过程可以描述为图 17。

状态	ACTION				GOTO				
	.	0	1	#	*N*	*S*	*B*	*M*	*P*
0	s2				1				
1				acc					
2		r6	r6	r6				3	
3		s5	s6	r3		4	7		
4				r1					
5		r4	r4	r4					
6		r5	r5	r5					
7		r7	r7	r7					8
8		s5	s6	r3		9	7		
9				r2					

图 16　基于二进制无符号定点小数文法的一个 LR 分析表

步骤	分析栈（状态，符号，语义值）	余留符号串	分析动作	语义动作
(0)	0 # -	. 1 0 1 #	s2	
(1)	0 # - 2 . -	1 0 1 #	r6	*v*[*top*+1].*s* := 1
(2)	0 # - 2 . - 3 *M* 1	1 0 1 #	s6	
(3)	0 # - 2 . - 3 *M* 1 6 1 -	0 1 #	r5	*v*[*top*].*val* := 2^(-*v*[*top*-1].*s*)
(4)	0 # - 2 . - 3 *M* 1 7 *B* 0.5	0 1 #	r7	*v*[*top*+1].*s* := *v*[*top*-1].*s*+1
(5)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2	0 1 #	s5	
(6)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 5 0 -	1 #	r4	*v*[*top*].*val* := 0
(7)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0	1 #	r7	*v*[*top*+1].*s* := *v*[*top*-1].*s*+1
(8)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3	1 #	s6	
(9)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3 6 1 -	#	r5	*v*[*top*].*val* := 2^(-*v*[*top*-1].*s*)
(10)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3 7 *B* 0.125	#	r7	*v*[*top*+1].*s* := *v*[*top*-1].*s*+1
(11)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3 7 *B* 0.125 8 *P* 4	#	r3	*v*[*top*+1].*val* := 0
(12)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3 7 *B* 0.125 8 *P* 4 9 *S* 0	#	r2	*v*[*top*-2].*val* := *v*[*top*].*val* + *v*[*top*-2].*s*
(13)	0 # - 2 . - 3 *M* 1 7 *B* 0.5 8 *P* 2 7 *B* 0 8 *P* 3 9 *S* 0.125	#	r2	*v*[*top*-2].*val* := *v*[*top*].*val* + *v*[*top*-2].*val*
(14)	0 # - 2 . - 3 *M* 1 4 *S* 0.625	#	r1	*print*(*v*[*top*] .*val*)
(15)	0 # - 1 *N* -	#	acc	

图 17　基于某个 *L*-翻译模式的 LR 分析过程和语义计算过程

课后作业

1　下面的文法 *G*[*S'*] 描述由布尔常量 *false*、*true* ，联结词 ∧（合取）、∨（析取）、¬（否定）构成的不含括号的二值布尔表达式的集合：

$$S' \to S$$
$$S \to S \vee T \mid T$$

$T \to T \wedge F \mid F$
$F \to \neg F \mid \underline{false} \mid \underline{true}$

试设计一个基于 $G[S']$ 的属性文法，它可以计算出每个二值布尔表达式的取值。如对于句子 $\neg true \vee \neg false \wedge true$，输出是 $true$。

2 给定文法 $G[S]$：

$S \to （L） \mid a$
$L \to L\ ,\ S \mid S$

如下是相应于 $G[S]$ 的一个属性文法（或翻译模式）：

$S \to （L）$ { $S.num := L.num + 1$; }
$S \to a$ { $S.num := 0$; }
$L \to L_1\ ,\ S$ { L.num := L_1.num + S.num; }
$L \to S$ { L.num := S.num; }

图 18 分别是输入串 (a，(a)) 的语法分析树和对应的带标注语法树，但后者的属性值没有标出，试将其标出（即填写右下图中符号 “=” 右边的值）。

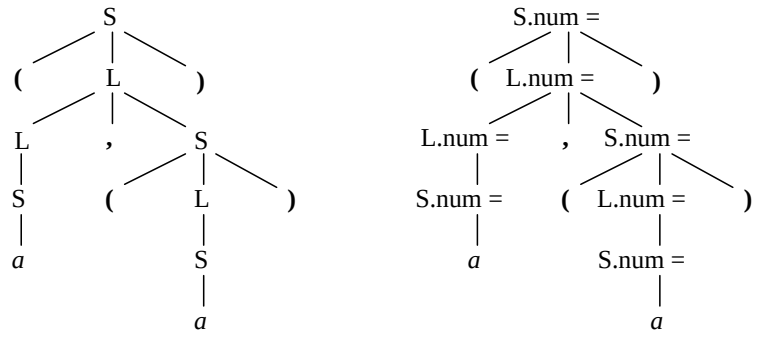

图 18 题 2 的语法分析树和带标注语法树

3 以下是简单表达式（只含加、减运算）计算的一个属性文法 $G(E)$：

$E \to T\ R$ { $R.in := T.val$; $E.val := R.val$ }
$R \to +TR_1$ { $R_1.in := R.in + T.val$; $R.val := R_1.val$ }
$R \to -TR_1$ { $R_1.in := R.in - T.val$; $R.val := R_1.val$ }
$R \to \varepsilon$ { $R.val := R.in$ }
$T \to \underline{num}$ { $T.val := lexval(\underline{num})$ }

其中，$lexval(\underline{num})$ 表示从词法分析程序得到的常数值。

试给出表达式 3+4–5 的语法分析树和相应的带标注语法分析树。

4 题 2 中所给的 $G[S]$ 的属性文法是一个 S-属性文法，故可以在自底向上分析过程中，增加语义栈来计算属性值。图 19 是 $G[S]$ 的一个 LR 分析表，图 20 描述了输入串 (a, (a)) 的分析和求值过程（语义栈中的值对应 $S.num$ 或 $L.num$），其中，第 14），15）行没有给出，试补齐之。

状态	ACTION					GOTO	
	a	,	(	)	#	S	L
0	s_3		s_2			1	
1					acc		
2	s_3		s_2			5	4
3		r_2		r_2	r_2		
4		s_7		s_6			
5		r_4		r_4			
6		r_1		r_1	r_1		
7	s_3		s_2			8	
8		r_3		r_3			

图 19　题 4 的 LR 分析表

步骤	状态栈	语义栈	符号栈	余留符号串
1）	0	–	#	(*a* , (*a*)) #
2）	02	– –	# (	*a* , (*a*)) #
3）	023	– – –	# (*a*	, (*a*)) #
4）	025	– – 0	# (S	, (*a*)) #
5）	024	– – 0	# (L	, (*a*)) #
6）	0247	– – 0 –	# (L ,	(*a*)) #
7）	02472	– – 0 – –	# (L , (	*a*)) #
8）	024723	– – 0 – – –	# (L , (*a*	)) #
9）	024725	– – 0 – – 0	# (L , (S	)) #
10）	024724	– – 0 – – 0	# (L , (L	)) #
11）	0247246	– – 0 – – 0 –	# (L , (L)	) #
12）	02478	– – 0 – 1	# (L , S	) #
13）	024	– – 1	# (L	) #
14）				
15）				
16）	接受			

图 20　题 4 的分析和求值过程

5　给定 LL(1)文法 *G*[*S*]：

$$S \to A\ b\ B$$
$$A \to a\ A \mid \varepsilon$$
$$B \to a\ B \mid b\ B \mid \varepsilon$$

如下是以 *G*[*S*] 作为基础文法设计的翻译模式：

$S \to A\ b$ {*B.in_num := A .num*}　B　{ *if B.num=0 then*　*print(“Accepted!”)*
else　*print(“Refused!”)*　}

$A \to aA_1$　{*A. num* := A_1*.num* + 1}
$A \to \varepsilon$　{*A. num* := 0}
$B \to a$　{B_1*.in_num* := *B.in_num*}　B_1　{*B. num* := B_1*.num* - 1}
$B \to b$　{B_1*.in_num* := *B.in_num*}　B_1　{*B. num* := B_1*.num* }
$B \to \varepsilon$　{*B. num* := *B.in_num* }

试针对该翻译模式构造相应的递归下降（预测）翻译程序（参考例 9）。（可直接使用 2.3 中的 MatchToken 函数）

6 设题 2 中属性文法的基础文法为 $G(E)$。

（*a*）说明 $G(E)$ 是 LL(1)文法。

（*b*）如下是以 $G(E)$ 作为基础文法设计的翻译模式：

$E \to T$ { $R.in := T.val$ } R { $E.val := R.val$ }
$R \to +T$ { $R_1.in := R.in + T.val$ } R_1 { $R.val := R_1.val$ }
$R \to -T$ { $R_1.in := R.in - T.val$ } R_1 { $R.val := R_1.val$ }
$R \to \varepsilon$ { $R.val := R.in$ }
$T \to \underline{num}$ { $T.val := lexval(\underline{num})$ }

试针对该翻译模式构造相应的递归下降（预测）翻译程序。（如上题，可直接使用2.3节中的 MatchToken 函数）

7 以下是一个 S-翻译模式，其基础文法为 SLR(1)文法（开始符号为 S）：

$S \to E$	{ $print(E.val)$ }
$E \to E_1 + T$	{ $E.val := E_1.val + T.val$ }
$E \to T$	{ $E.val := T.val$ }
$T \to T_1 * F$	{ $T.val := T_1.val \times F.val$ }
$T \to F$	{ $T.val := F.val$ }
$F \to (E)$	{ $F.val := E.val$ }
$F \to d$	{ $F.val := d.lexval$ }

其中，$d.lexval$ 是由词法分析程序所确定的属性；$F.val$，$T.val$ 和 $E.val$ 都是综合属性；语义函数 $print(E.val)$ 用于显示 $E.val$ 的结果值。不难理解，这个属性文法描述了一个基于简单表达式文法进行算术表达式求值的语义计算模型。

由于是 S-翻译模式，所以在 LR 分析过程中根据该翻译模式进行自底向上语义计算时，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的综合属性进行访问。试写出在按每个产生式归约时语义计算的一个代码片断（设语义栈由向量 v 表示，归约前栈顶位置为 top，终结符不对应语义值，而每个非终结符的综合属性都只对应一个语义值，可用 v[i].val 访问，不用考虑对 top 的维护）。

8 给定文法 $G[S]$：

$S \to M\ A\ b\ B$
$A \to A\ a \mid \varepsilon$
$B \to B\ a \mid B\ b \mid \varepsilon$
$M \to \varepsilon$

在文法 $G[S]$ 基础上设计如下翻译模式：

$S \to M$ { $A.in\_num := M.num$ }
$A\ b$ { $B.in\_num := A.num$ }
B { *if* $B.num=0$ *then* $S.accepted := true$ else $S.accepted := false$ }
$A \to$ { $A_1.in\_num := A.in\_num$ } $A_1\ a$ { $A.num := A_1.num - 1$ }

$A \to \varepsilon$ {A.*num* := A.*in_num* }

$B \to$ {B_1.*in_num* := B.*in_num*} $B_1\ a$ {B. *num* := B_1.*num* - 1}

$B \to$ {B_1.*in_num* := B.*in_num*} $B_1\ b$ {B. *num* := B_1.*num* }

$B \to \varepsilon$ {B. *num* := B.*in_num* }

$M \to \varepsilon$ { M.*num* :=100 }

不难看出，嵌在产生式中间的语义动作集中仅含复写规则，并且在自底向上的语法分析过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的综合属性唯一确定地进行访问。试写出在按每个产生式归约时语义计算的一个代码片断（设语义栈由向量 *v* 表示，归约前栈顶位置为 *top*，终结符不对应语义值，而每个非终结符的综合属性都只对应一个语义值，可用 *v*[i].*num* 或 *v*[i]. *accepted* 访问，不用考虑对 *top* 的维护）。

9 变换如下翻译模式，使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语法分析过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问：

$D \to D_1$; T { L.*type* := T.*type*; L.*offset* := D_1.*width* ; L.*width* := T.*width* } L
{D.*width* := D_1.*width* + L.*num* × T.*width* }

$D \to T$ { L.*type* := T.*type*; L.*offset* := 0 ; L.*width* := T.*width* } L
{D.*width* := L.*num* × T.*width* }

$T \to$ integer { T.*type* := *int* ; T.*width* := 4 }

$T \to$ real { T.*type* := *real* ; T.*width* := 8 }

$L \to$ { L_1. *type* := L. *type* ; L_1. *offset* := L. *offset* ; L_1. *width* := L. *width* ; } L_1 , id
{ *enter* (id.*name*, L. *type*, L. *offset* + L_1.*num* × L. *width*) ; L.*num* := L_1.*num* +1 }

$L \to$ id { *enter* (id.*name*, L. *type*, L. *offset*) ; L.*num* := 1}

10 设有如下翻译模式，其基础文法是 *G*[*N*]：

$N \to$ { S.*f* : =1 } S { *print*(S.*v*) }

$S \to$ { S_1.*f* := 2 S.*f* } S_1 { B.*f* : =S.*f* } B { S.*val* := S_1.*v* + B.*v* }

$S \to$ ε { S.*v* := 0 }

$B \to$ 0 { B.*v* := 0 }

$B \to$ 1 { B.*v* := B. *f* }

（*a*）变换该翻译模式，使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语义计算过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问。

（*b*）如果在 LR 分析过程中根据（*a*）所得到的新翻译模式进行自底向上的语义计算，试写出在按每个产生式归约时语义计算的一个代码片断（设语义栈由向量 *v* 表示，归约前栈顶位置为 *top*，终结符不对应语义值；而每个非终结符的综合属性都只对应一个语义值，用 *v*[*i*].*val* 表示；不用考虑对 *top* 的维护）。

11 设有如下翻译模式，其基础文法是 *G*[*S*]：

$S \to A\ b$ {*B.in_num := A .num* + 100}
B { *if B.num=0 then S.accepted := true*
else S.accepted := false }

$S \to A\ b\ b$ {*B.in_num := A .num* + 50}

B { *if* $B.num=0$ *then* $S.accepted := true$
else $S.accepted := false$ }

$A \to A_1\, a$ {$A.\ num := A_1.num + 1$}

$A \to \varepsilon$ {$A.\ num := 0$}

$B \to$ {$B_1.in\_num := B.in\_num$} $B_1\, a$ {$B.\ num := B_1.num - 1$}

$B \to \varepsilon$ {$B.\ num := B.in\_num$ }

（*a*）变换该翻译模式，使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语义计算过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问。

（*b*）如果在 LR 分析过程中根据（*a*）所得到的新翻译模式进行自底向上的语义计算，试写出在按每个产生式归约时语义计算的一个代码片断（假设语义栈由向量 *v* 表示，归约前栈顶位置为 *top*，终结符不对应语义值，而每个非终结符的综合属性都只对应一个语义值，可用 *v*[i].*num* 或 *v*[i]. *accepted* 访问，不用考虑对 *top* 的维护）。

12 下面的属性文法（翻译模式）*G*[*N*] 可以将一个二进制小数转换为十进制小数，令 *N.val* 为 *G*[*N*] 生成的二进制数的值。例如，对输入串 101.101，*N.val*=5.625。

$N \to S_1 \cdot S_2$ { $N.val := S_1.val + 2^{-S_2.len} \times S_2.val$ }

$S \to S_1\, B$ { $S.val := 2 \times S_1.val + B.val$; $S.len := S_1.len + 1$ }

$S \to B$ { $S.val := B.val$; $S.len := 1$ }

$B \to 0$ { $B.val := 0$ }

$B \to 1$ { $B.val := 1$ }

（*a*）试用本讲中介绍的方法消除该属性文法（翻译模式）中的左递归，以便可以得到一个可以进行自上而下进行语义计算的翻译模式。

（*b*）对变换后的翻译模式，构造一个递归下降（预测）翻译程序。

13 对于题 12（*a*）所得到的翻译模式（结果应满足 L-属性的条件），在进行自下而上的语义计算时，语义栈中的值有两个分量，分别对应文法符号的综合属性 val 和 len。

（*a*）变换该翻译模式，使嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语义计算过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问。

（*b*）如果在 LR 分析过程中根据（*a*）所得到的新翻译模式，试写出在按每个产生式归约时语义计算的一个代码片断（设语义栈由向量 v 表示，归约前栈顶位置为 top，语义值 v[i]的两个分量分别用 v[i].val 和 v[i].len 表示）。