第九章 语法制导翻译技术

- 翻译文法(TG)和语法制导翻译
- 属性翻译文法(ATG)
- 自顶向下语法制导翻译
  - 翻译文法的自顶向下语法制导翻译
  - 属性翻译文法的自顶向下语法制导翻译

9.0 本章导言

★ **词法分析，语法分析：解决单词和语言成分的识别及词法和语法结构的检查。语法结构可形式化地用一组产生式来描述。给定一组产生式，能够很容易地将其分析器构造出来。**

本章要介绍的是语义分析和代码生成技术。

★ **程序语言的语义形式化描述目前有三种基本描述方法，即：**

- **操作语义:**使用抽象机和抽象解释程序来定义语言的定义，着重描述语言的执行过程。
- **指称语义：**通过执行结果来定义语言的语义，着重于语言的执行结果而非过程。
- **公理语义：**通过使用数学中的公理化方法，用公理系统定义程序设计语言的语义，是程序正确性研究的理论基础。

9.1 翻译文法和语法制导翻译

有上下无关文法G[E]:

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

此文法是一个中缀算术表达式文法

翻译的任务是: 中缀表达式 ⟶ 波兰表示

a+b*c ⟶ abc*+

假如翻译任务是要将中缀表达式简单变换为波兰后缀表示，只需在上述文法中插入相应的动作符号。

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

1. E →E+T@+
2. E →T
3. T →T*F @*
4. T →F
5. F →(E)
6. F →i@i

其中：

@+,@*,@i 为动作符号。@为动作符号标记，后面为字符串。在本例中，其对应语义子程序的功能是要输出动作符号标记后面的字符串。

所以，产生式1：E→E+T@+ 的语义是分析E, + 和T，输出+

产生式6：F→i @ i的语义是分析i，输出i

下面给出输入文法和翻译文法的概念：

输入文法： 未插入动作符号时的文法。

由输入文法可以通过推导产生输入序列。

翻译文法： 插入动作符号的文法。

由翻译文法可以通过推导产生活动序列。

⇩

输入序列

动作序列

例：(i+i) * i

可以用输入文法推导：

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E+T)*F$

$\overset{*}{\Rightarrow}(i+i)*i$

ii+i*　　如何得到波兰表示?

用相应的翻译文法推导，可得：

1. E →E+T@+
2. E →T
3. T →T*F @*
4. T →F
5. F →(E)
6. F →i@i

$E \Rightarrow T$

$\Rightarrow T*F@*$

$\Rightarrow F*F@*$

$\Rightarrow (E)*F@*$

$\Rightarrow (E+T@+)*F@* \quad \overset{*}{\Rightarrow}(i@i+i@i\ @+)*i@i@*$

(i@i+i@i @+)*i@i@*

活动序列：由翻译文法推导出的符号串，由**终结符**和**动作符号**组成。

- 从**活动序列**中，抽去**动作符号**，则得**输入序列**(i+i)*i
- 从**活动序列**中，抽去**输入序列**，则得**动作序列**，执行动作序列，则完成翻译任务：

$$@i@i@+@i@* \Rightarrow ii+i*$$

定义9.1

翻译文法是上下文无关文法，其**终结符号集**由**输入符号**和**动作符号**组成。由翻译文法所产生的终结符号串称为**活动序列**。

上例题中的翻译文法为：

$G_T = (V_n, V_t, P, E)$

$V_n = \{E, T, F\}$

$V_t = \{i, +, *, (,), @+, @*, @i\}$

$P = \{E \rightarrow E+T@+,\ E \rightarrow T,\ T \rightarrow T*F@*,\ T \rightarrow F,\ F \rightarrow (E),$
$F \rightarrow i@i\}$

符号串翻译文法：若插入文法中的动作符号对应的语义子程序是输出动作符号标记@后的字符串的文法。

语法制导翻译：按翻译文法进行的翻译。
给定一输入符号串，根据翻译文法获得翻译该符号串的动作序列，并执行该序列所规定的动作的过程。

语法制导翻译的实现方法：

在文法的适当位置插入语义动作符号，当按文法分析到动作符号时就调用相应的语义子程序，完成翻译任务。

翻译文法所定义的翻译是由输入序列和动作序列组成的对偶集。

如：(i+i)*i,　@i@i@+@i@*　→ii + i*

i+i*i　　@i@i@i@*@+

因此，给定一个翻译文法，就给定了一个对偶集。

9.2 属性翻译文法

在翻译文法的基础上，可以进一步定义**属性文法**，翻译文法中的符号，包括终结符、非终结符和动作符号均可带有属性，这样能更好的描述和实现编译过程。

属性可以分为两种：

- 综合属性
- 继承属性

9.2.1 综合属性

基本操作数带有属性的表达式文法G[E]

1.E→E+T　　4.T→F

2.E→T　　5.F→(E)

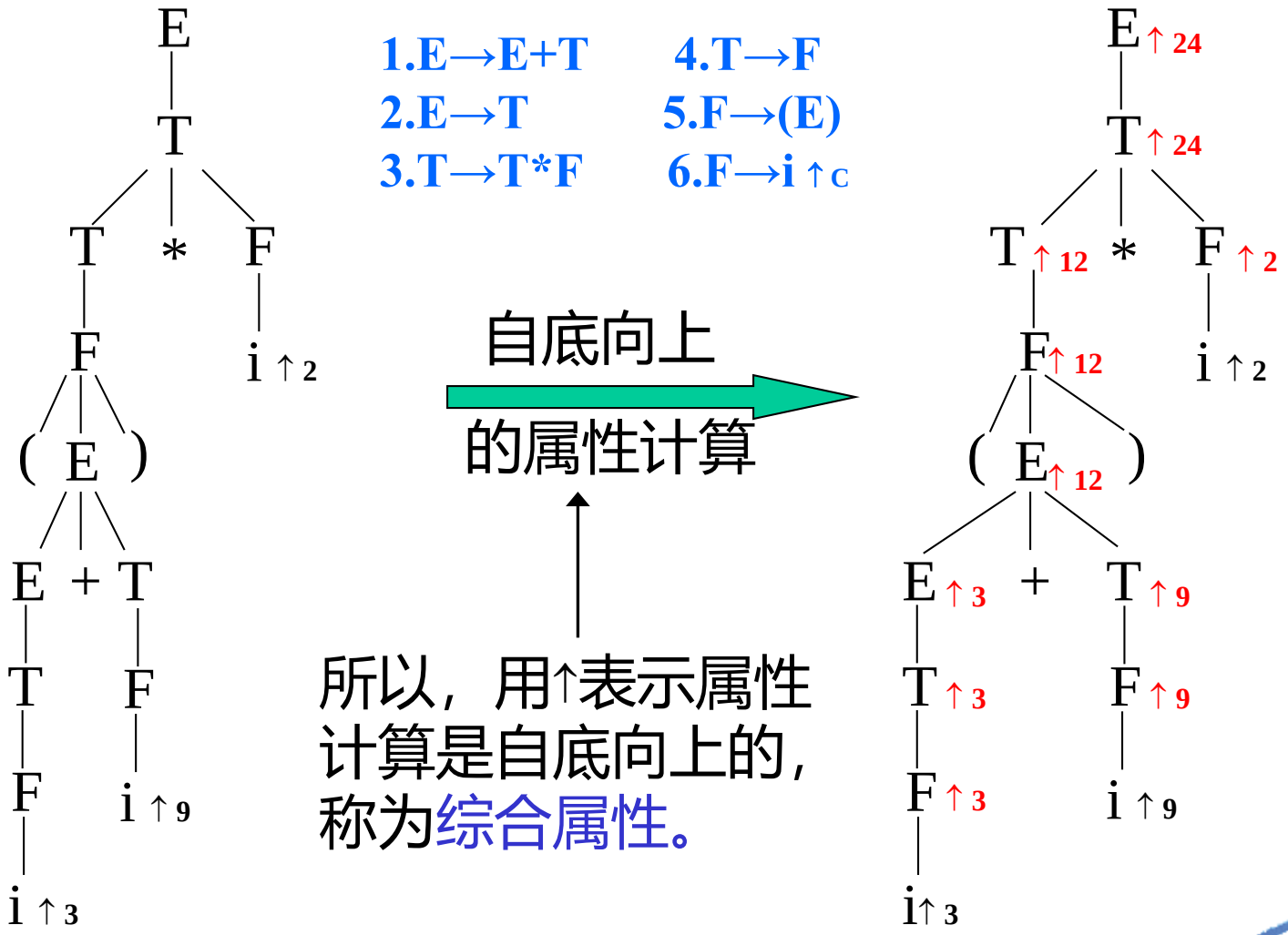
3.T→T*F　　6.F→i↑c

其中↑c是综合属性符号，↑为综合属性标记，C为属性变量或者属性值。

此文法能够产生如下的输入序列：

$$(i\uparrow_3 + i\uparrow_9) * i\uparrow_2$$

根据给定的文法，可画出该输入序列的语法树



为了形式地表示上述表达式的属性求值过程，可以改写上述文法：

产生式	求值规则
1.E ↑p4 ⟶ E ↑q5 + T ↑r2	p 4:=q 5+r 2 ;
2.E ↑p3 ⟶ T ↑q4	p 3:=q 4 ;
3.T ↑p2 ⟶ T ↑q3 * F ↑r1	p 2:=q 3*r 1 ;
4.T ↑p2 ⟶ F ↑q2	p 2:=q 2 ;
5.F ↑p1 ⟶ (E ↑q1)	p 1 :=q 1 ;
6.F ↑p1 ⟶ i ↑q1	p 1 :=q 1 ;

说明：

- p,q,r为属性变量名。
- 属性变量名局部于每个产生式，也可使用不同的名字。
- 求值规则：综合属性是自右向左，自底向上求值。

9.2.2 继承属性

考虑下列文法：G[<说明>]：

1. <说明> → Type id <变量表>
2. <变量表> → , id <变量表>
3. <变量表> → ε

其中

Type: 类型名（值：int, real, bool等）

id： 变量名（值：指向该变量符号表项的指针）

上述文法所产生的语句：int A,BC

该文法的翻译任务：将声明的变量填入符号表

完成该工作的动作符号：@set_table

翻译文法：

1. <说明> → Type id @set_table <变量表>
2. <变量表> → , id @set_table <变量表>
3. <变量表 > → ε

填表时需要的信息：类型，名字，以及填的位置（可以用全程变量或指针）

如何得到?

类型和名字在词法分析时得到，可设两个综合属性。

Type $\uparrow_t$ t中放类型值

id $\uparrow_n$ n中放变量名

填表动作符号也可带有属性：

@set_table$\downarrow_{t_1, n_1}$ $\downarrow t_1, n_1$ 可从前面得到，所以称为继承属性，继承前面的值

<变量表> $\downarrow_{t_2}$ $\downarrow t_2$ 同上

属性翻译文法：

1.<说明> → Type$\uparrow_t$ id$\uparrow_n$ @set_table$_{\downarrow t1,n1}$ <变量表> $_{\downarrow t2}$ $t_2, t_1 := t;$ $n_1 := n;$

2.<变量表> $_{\downarrow t2}$ → ， id $\uparrow_n$ @set_table$_{\downarrow t1,n1}$ <变量表>$_{\downarrow t3}$ $t_3, t_1 := t_2;$ $n_1 := n;$

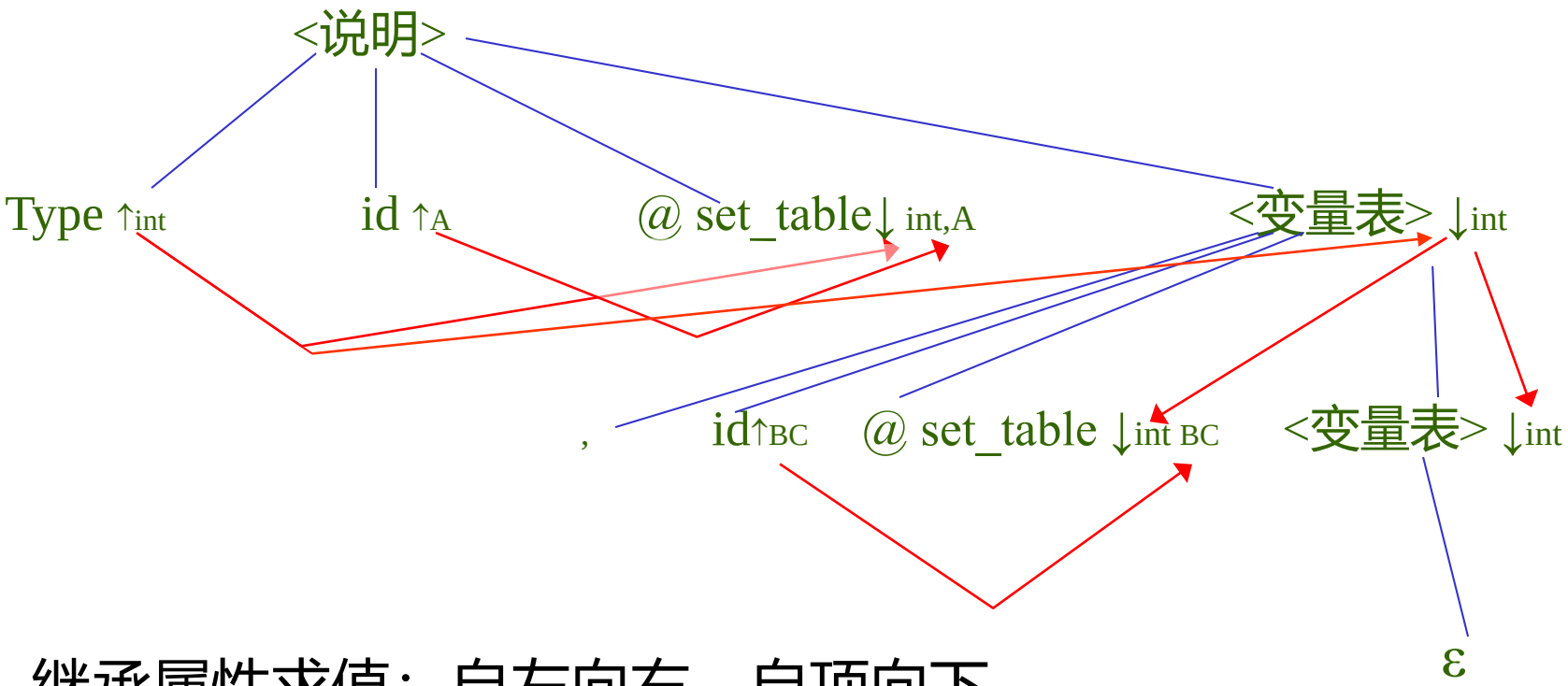
3.<变量表 > $_{\downarrow t2}$ → ε

1.<说明> → Type$_{\uparrow t}$ id$_{\uparrow n}$ @set_table$_{\downarrow t1,n1}$ <变量表> $_{\downarrow t2}$

2.<变量表> $_{\downarrow t2}$ → ， id $_{\uparrow n}$ @set_table$_{\downarrow t1,n1}$ <变量表>$_{\downarrow t3}$　　3.<变量表 > $_{\downarrow t2}$ → ε

例：int A,BC ⇨ Type $_{\uparrow int}$ id $_{\uparrow A}$， id$_{\uparrow BC}$

语法树：



继承属性求值：自左向右，自顶向下

综合属性求值：自右向左，自底向上

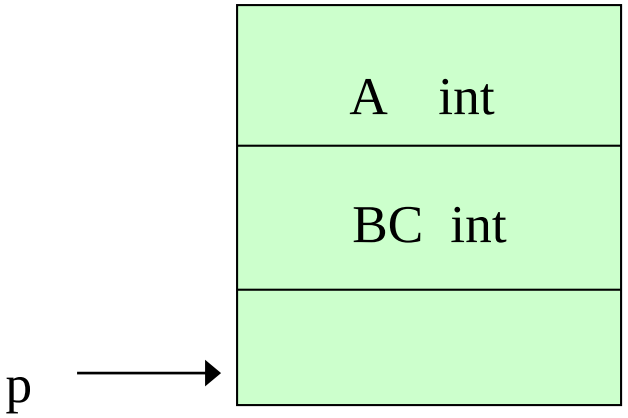
int A, BC 的分析翻译过程：

<说明> $\Rightarrow$ Type $\uparrow_t$ id $\uparrow_{n1}$@ set_table$_{\downarrow t,n1}$ <变量表>$_{\downarrow t}$

$\overset{+}{\Rightarrow}$ Type $\uparrow_t$ id $\uparrow_{n1}$ @ set_table$_{\downarrow t,n1}$

, id $\uparrow_{n2}$ @ set_table $_{\downarrow t,n2}$

符号表



对于G的每一个非终结符A的任意两条规则A::=α|β:

1、FIRST(α) ∩ FIRST(β) = Φ

2、若β$\overset{*}{\Rightarrow}$ ε, 则FIRST(α) ∩ FOLLOW(A) = Φ

9.2.3 (1) L-属性翻译文法（L-ATG)

Attribute Translation Grammar

这是属性翻译文法中较简单的一种。其输入文法要求是LL(1)文法，可用自顶向下分析构造分析器。在分析过程中可进行属性求值。

定义9.2:

L-属性翻译文法是带有下列说明的翻译文法：

1. 文法中的终结符，非终结符及动作符号都带有属性，且每个属性都有一个值域。

2. 非终结符及动作符号的属性可分为继承属性和综合属性。

3. 开始符号的继承属性具有指定的初始值。

4. 输入符号（终结符号）的每个综合属性具有指定的初始值。

5. 属性的求值规则：

1.<说明> → Type$_{\uparrow t}$ id$_{\uparrow n}$ @set_table$_{\downarrow t1,n1}$ <变量表> $_{\downarrow t2}$

2.<变量表> $_{\downarrow t2}$ → ， id $_{\uparrow n}$ @set_table$_{\downarrow t1,n1}$ <变量表>$_{\downarrow t3}$　　3.<变量表 > $_{\downarrow t2}$ → ε

属性的求值规则：

体现自顶向下，自左向右的求值特性。

继承属性：

（1） 产生式左部非终结符号的继承属性值，取前面产生式右部该符号已有的继承属性值。比如t2

（2）产生式右部符号的继承属性值，用该产生式左部符号的继承属性或出现在该符号左部的符号的属性值进行计算。　比如t1、n1

综合属性：

(1) 产生式右部非终结符号的综合属性值，取其下部产生式左部同名非终结符号的综合属性值。

(2) 产生式左部非终结符号的综合属性值，用该产生式左部符号的继承属性或某个右部符号的属性进行计算。

(3) 动作符号的综合属性用该符号的继承属性或某个右部符号的属性进行计算。

体现自底向上，自右向左的求值特性

适合在自顶向下分析过程中求值

例：A→BC

（2）产生式左部非终结符号的综合属性值，用该产生式左部符号的继承属性或某个右部符号的属性进行计算。

求值顺序：

1) A的继承属性 （若A为开始符号，则有指定值,否则由上面产生式右部符号的继承属性求得）
2) B的继承属性 （由A的继承属性求得）
3) B的综合属性 （由下面产生式中左部符号为B的综合属性求得）
4) C的继承属性 （由A的继承属性和B的属性求得）
5) C的综合属性 （由下面产生式中左部符号为C的综合属性求得）
6) A的综合属性 （由前述(2)，即A的继承属性或产生式某右部符号属性计算）

(2) 简单赋值形式的L_属性翻译文法(SL-ATG)

- 一般属性值计算： x:=f (y , z)

SL-ATG属性值计算： x:=某符号的属性值或常量。

例 x:=y, x,y,z:=17 —— 称为复写规则

为了实现上的方便，常希望文法符号的属性求值规则为上述简单形式的。为此，对现有的L-ATG的定义做一点改变，从而形成一个称为简单赋值形式的L-ATG。

•定义9.4 一个L-ATG被定义为简单赋值形式的(SL-ATG)，当且仅当满足如下条件:

1. 产生式右部符号的继承属性是一个常量，它等于左部符号的继承属性值或等于出现在所给符号左边符号的一个综合属性值。

2.产生式左部非终结符号的综合属性是一个常量，它等于左部符号的继承属性值或等于右部符号的综合属性值。

因此，一个简单赋值形式的L-ATG除动作符号外，其余符号的属性求值规则其右部是属性或是常量。

- L-ATG ⇒ SL-ATG

给定一个L-ATG，如何找一个等价的赋值形式的L-ATG?

考虑产生式:

$$<A> \longrightarrow a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I}, \qquad I := f(R, S)$$

显然: 该属性求值规则不是简单赋值形式的，因为它需要对f求值。

$<A> \rightarrow a\ {}_{\uparrow R}<B>{}_{\uparrow S}<C>{}_{\downarrow I}$,　　$I := f(R, S)$

第一步：设动作符号“@ f ”表示函数f求值，该动作符号有两个继承属性和一个综合属性。

$@f\ {}_{\downarrow I_1, I_2 \uparrow S_1}$　　且 $S_1 := f(I_1, I_2)$

第二步：修改产生式

1. 插入“@ f ”（在适当位置）
2. 引进新的复写规则（将R , S 赋给I_1和I_2, f值赋给S_1 ）
3. 删去原有包含f 的规则

$<A>\rightarrow @f_{\downarrow I_1, I_2 \uparrow S_1}\ a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I}$,

$I_1 := R$, $I_2 := S$, $S_1 := f(I_1, I_2)$, $I := S_1$

$<A>\rightarrow a_{\uparrow R} @f_{\downarrow I_1, I_2 \uparrow S_1} <B>_{\uparrow S} <C>_{\downarrow I}$,

$I_1 := R$, $I_2 := S$, $S_1 := f(I_1, I_2)$, $I := S_1$

$<A>\rightarrow a_{\uparrow R} <B>_{\uparrow S} @f_{\downarrow I_1, I_2 \uparrow S_1} <C>_{\downarrow I}$,

$I_1 := R$, $I_2 := S$, $S_1 := f(I_1, I_2)$, $I := S_1$

$<A>\rightarrow a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I} @f_{\downarrow I_1, I_2 \uparrow S_1}$,

$I_1 := R$, $I_2 := S$, $S_1 := f(I_1, I_2)$, $I := S_1$

1. 产生式右部符号的继承属性是一个常量，它等于左部符号的继承属性值或等于出现在所给符号左边符号的一个综合属性值。

2.产生式左部非终结符号的综合属性是一个常量，它等于左部符号的继承属性值或等于右部符号的综合属性值。

（3）动作符号的综合属性用该符号的继承属性或某个右部符号的属性进行计算。

$<A> \rightarrow a\ {}_{\uparrow R} <B>\ {}_{\uparrow S} <C>\ {}_{\downarrow I}$,　　$I := f(R, S)$

$<A> \rightarrow a\ {}_{\uparrow R} <B>\ {}_{\uparrow S}\ @\ f\ {}_{\downarrow I_1, I_2 \uparrow S_1} <C>\ {}_{\downarrow I}$,

$I_1 := R$,　$I_2 := S$,　　$S_1 := f(I_1, I_2)$,　　$I := S_1$.

该文法是简单赋值形式的L-ATG.

注意：　无参函数过程作为常数处理，如

$<A> \rightarrow <B>\ {}_{\uparrow x} <C>\ {}_{\uparrow y}$　　x , y := NEWT

9.3 自顶向下语法制导翻译

9.3.1 翻译文法的自顶向下翻译

——递归下降翻译器

9.3.2 属性翻译文法的自顶向下翻译的实现

——递归下降属性翻译器

9.3.1 翻译文法的自顶向下翻译——递归下降翻译器

按翻译要求，在文法中插入语义动作符号，在分析过程中调用相应的语义处理程序，完成翻译任务。

例：输入文法

1. < S > ⟶ a< A >< S >
2. < S > ⟶ b
3. < A > ⟶ c< A >< S >b
4. < A > ⟶ ε

翻译文法（符号串翻译文法）

< S > ⟶ a < A > @ x < S >

< S > ⟶ b @ z

< A > ⟶ c @ y < A > < S > @ v b

< A > ⟶ @ w

1. < S > → a< A >< S >
2. < S > → b
3. < A > → c< A >< S >b
4. < A > → ε

< S > → a < A > @ x < S >
< S > → b @ z
< A > → c @ y < A > < S > @ v b
< A > → @ w

主程序

```
int main
{
    nextsym();    /* 预读一个输入符号 */
    if (class == 'a' || class == 'b')
    {
        proc_S();    /* 调用S的分析程序 */
    }
    if (class != '#')
    {
        error();
    }
    return 0;
}
```

主程序

```
int main
{
    nextsym();    /* 预读一个输入符号 */
    if (class == 'a' || class == 'b')
    {
        proc_S();    /* 调用S的分析程序 */
    }
    if (class != '#')
    {
        error();
    }
    return 0;
}
```

1. < S > → a< A >< S >
2. < S > → b
3. < A > → c< A >< S >b
4. < A > → ε

```
void proc_S()     /* S的分析程序  */
{
   if (class == 'a')
   {
      nextsym();   /* a被匹配，读下一个符号
*/
      proc_A();    /* 调用A的分析程序  */
      proc_S();     /* 调用S的分析程序  */
   }
   else if (class = 'b')
  {
      nextsym();    /* b被匹配，读下一个符号
*/
   }
   else
  {
      error ();
   }
}
```

< S > → a < A > @ x < S >
< S > → b @ z
< A > → c @ y < A > < S > @ v b
< A > → @ w

```
void proc_S()     /* S的分析程序  */
{
   if (class == 'a')
   {
      nextsym();   /* a被匹配，读下一个符号
*/
      proc_A();    /* 调用A的分析程序  */
      printf("x");      /*  输出x          */
      proc_S();     /* 调用S的分析程序  */
   }
   else if (class = 'b')
  {
      nextsym();    /* b被匹配，读下一个符号
*/
      printf("z");      /*  输出z          */
   }
   else
  {
      error ();
   }
}
```

1. < S > → a< A >< S >
2. < S > → b
3. < A > → c< A >< S >b
4. < A > → ε

```
void proc_A()      /* A的分析程序  */
{
   if (class == 'c')
   {
      nextsym();      /* 预读一个输入符号  */

      proc_A();       /* 调用A的分析程序  */
      proc_S();       /* 调用S的分析程序  */

     if (class != 'b')
     {
         error();
     }
     nextsym();       /* 预读一个输入符号  */
   }
   else if (class == 'a'|| class == 'b')
   {

      return;
   }
   else
   {
       error();
   }
}
```

< S > → a < A > @ x < S >
< S > → b @ z
< A > → c **@ y** < A > < S > **@ v** b
< A > → **@ w**

```
void proc_A()      /* A的分析程序  */
{
   if (class == 'c')
   {
      nextsym();      /* 预读一个输入符号  */
      printf("y");    /*  输出y          */
      proc_A();       /* 调用A的分析程序  */
      proc_S();       /* 调用S的分析程序  */
      printf("v");    /*  输出v          */
      if (class != 'b')
      {
          error();
      }
      nextsym();       /* 预读一个输入符号  */
   }
   else if (class == 'a'|| class == 'b')
   {
      printf("w");    /*  输出w          */
      return;
   }
   else
   {
       error();
   }
}
```

9.3.2 属性翻译文法自顶向下翻译的实现
——递归下降属性翻译器

方法：

- 对于每个非终结符号都编写一个翻译子程序（过程）。根据该非终结符号具有的属性数目，设置相应的参数。

继承属性：声明为赋值形参

综合属性：声明为变量形参

$U_{\downarrow x,\ \uparrow y} \longrightarrow \ldots$

Procedure U(x,y);

x—赋值形参

y—变量形参

- 过程调用语句的实参：

  继承属性 ：继承属性值

  综合属性 ：属性变量名（传地址，返回时有值）

- 关于属性名的约定：

1）产生式左部的同名非终结符使用相同的属性名。

（递归下降分析法所必须）

$<L>_{\uparrow a\downarrow b} \rightarrow e_{\downarrow I} <R>_{\downarrow J}$

$<L>_{\uparrow x\downarrow y} \rightarrow <H>_{\downarrow z\uparrow w}$

$<L>_{\uparrow x\downarrow y} \rightarrow e_{\downarrow I} <R>_{\downarrow J}$

$<L>_{\uparrow x\downarrow y} \rightarrow <H>_{\downarrow z\uparrow w}$

2）具有相同值的属性取相同的属性名。

具有简单赋值形式的属性变量名取相同的属性名，可删去属性求值规则。

$<S> \rightarrow I_{\uparrow a} <B>_{\downarrow b} <C>_{\downarrow c}$ b,c:= a

$<S> \rightarrow I_{\uparrow x} <B>_{\downarrow x} <C>_{\downarrow x}$

下面通过一个例子，较详细地介绍如何构造属性文法 的递归下降翻译器。

例：有如下属性翻译文法G[< S>]

$R_2 := R_1$
$T_2 := T_1$
$Q_2 := Q_1$

1. $< S >_{\downarrow R_1} \longrightarrow a \uparrow_{T_1} < A > \uparrow_{Q_1} @ x \downarrow_{T_2, R_2} < S > \downarrow_{Q_2}$

2. $< S >_{\downarrow R_1} \longrightarrow b @ z_{\downarrow R_2}$,　　$R_2 := R_1$

3. $< A > \uparrow_P \longrightarrow C \uparrow_{U_1} @ y_{\downarrow U_2} < A > \uparrow_Q < S > \downarrow_z @ v \downarrow_P b$

$U_2 := U_1$, $P := Q + U_1$, $Z := U_1 - 3$

4. $< A > \uparrow_P \longrightarrow @ w$　　$P := 8$

对简单赋值形式的属性变量取相同的属性名，其求值规则可以删去。开始符号的继承属性 $R_1=7$。

1. $< S >_{\downarrow R} \longrightarrow a \uparrow_T < A > \uparrow_Q @ x \downarrow_T < S > \downarrow_Q$

2. $< S >_{\downarrow R} \longrightarrow b @ z_{\downarrow R}$,

3. $< A > \uparrow_P \longrightarrow C \uparrow_U @ \ y_{\downarrow U} < A > \uparrow_Q < S > \downarrow_Z @ v \downarrow_P b$

   $P := Q + U, Z := U-3$

4. $< A > \uparrow_P \longrightarrow @ w$ $P := 8$

1. $<S>_{\downarrow R} \rightarrow a \uparrow_T <A> \uparrow_Q @ x \downarrow_{T,R} <S> \downarrow_Q$
2. $<S>_{\downarrow R} \rightarrow b @ z_{\downarrow R}$,
3. $<A> \uparrow_P \rightarrow C \uparrow_U @ y_{\downarrow U} <A> \uparrow_Q <S> \downarrow_Z @ v \downarrow_P b$

   P := Q + U, Z := U-3
4. $<A> \uparrow_P \rightarrow @ w$ P := 8

全局变量和过程声明:

```
CLASS;          / * 存放单词类别码 * /
TOKEN;          / * 存放单词值 */
NEXTSYM;        / *词法分析程序，每调用一次单词类别码 ⇒ CLASS,
                  单词值 ⇒ TOKEN,读符号指针指向下一个单词 */
```

主程序:

```
NEXTSYM;
PROCS(7);
if  CLASS  ≠  右界符     then      ERROR;
ACCEPT
```

过程 PROCS(R)

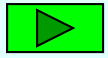
```
R;                          /* 值形参声明 */
```

1. $<S>_{\downarrow R}$ —> a $\uparrow_T$ $<A>_{\uparrow Q}$ @ X $\downarrow_{T,R}$ $<S>_{\downarrow Q}$

```
case     CLASS      of
a:       P1 ;
b:       P2 ;
其它:    ERROR;
end      of         case;
```

P1 :

```
T , Q ;                          / * 产生式1的代码 * /
                                 / * 局部变量声明 * /
T := TOKEN;                      / * 单词值赋给终结符的综合属性 * /
NEXTSYM;
PROCA(Q)
OUT(X ↓T,R) ;
PROCS(Q)
RETURN;
```



2. < S > ↓R —> b @ z ↓R

3. < A > ↑P —> C↑u @ y↓u < A > ↑Q < S >↓Z @ v↓P b

4. < A > ↑P —> @ w P:=8

P_2: / * 产生式2的代码 * /

NEXTSYM;

OUT(Z ↓R);

RETURN;

过程 PROCA(P)

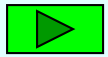
P; /*变量形参声明*/

case CLASS of

C : p3

其它: p4

end of case;



3. < A > ↑P —> C↑U @ $y_{\downarrow U}$ < A > ↑Q < S > $_{\downarrow Z}$ @ $v_{\downarrow P}$ b

P := Q + U, Z := U - 3

P3:

U , Q , Z ; /*局部变量声明*/

U := TOKEN;

NEXTSYM;

Z := U – 3 ; 插在U已知，使用Z之前

OUT($y_{\downarrow U}$);

PROCA(Q); 返回时有值

P := Q+U; 插在Q,U已知，使用P之前。

PROCS(Z);

OUT($V_{\downarrow P}$);

if CLASS ≠ b then ERROR;

NEXTSYM;

RETURN;

4. < A > ↑P —> @ w P := 8

```
P4:
    P :=  8;
    OUT(w);
    RETURN;
```

一个例子

例：构造将算术表达式翻译成四元式的属性翻译文法，并写出递归下降分析程序。由该属性翻译文法来描述翻译过程。

翻译的输入：算术表达式　　a + b

翻译的输出：四元式　　ADD , P_a , P_b , P_r

其中P_a , P_b , P_r 为变量a , b和结果单元的地址。

表达式：(a + b) * c

输入：(Id $\uparrow_1$ + Id $\uparrow_2$) * Id $\uparrow_4$

Id由词法分析程序返回，

$\uparrow_1$.....综合属性，变量在数据区地址。

输出：ADD, 1, 2, 3

MULT, 3, 4, 5

数据区：

(1) 翻译文法设计：

E →E+T @ADD　　T →F

E →T　　F →(E)

T →T*F @MULT　　F →Id

@ADD为输出ADD四元式的动作符号

@MULT为输出MULT四元式的动作符号

} 对应于完成翻译的语义动作程序

在文法中的插入位置：在分别处理完成两个操作数之后

输入序列：(a+b)*c

翻译文法产生的活动序列: (a+b@ADD)*c@MULT

动作符号序列：@ADD　@MULT　　反映生成四元式的顺序，语法分析过程中语义程序的调用顺序。

(2) 属性翻译文法的设计

- 输入符号（操作数）有一个综合属性，它是该符号在数据区的地址。
- 每个非终结符有一个综合属性，该属性是由它产生的代表该子表达式在数据区的地址。
- 动作符号有三个继承属性，它们分别是左右操作数和运算结果在数据区地址。

这样可得表达式的属性翻译文法

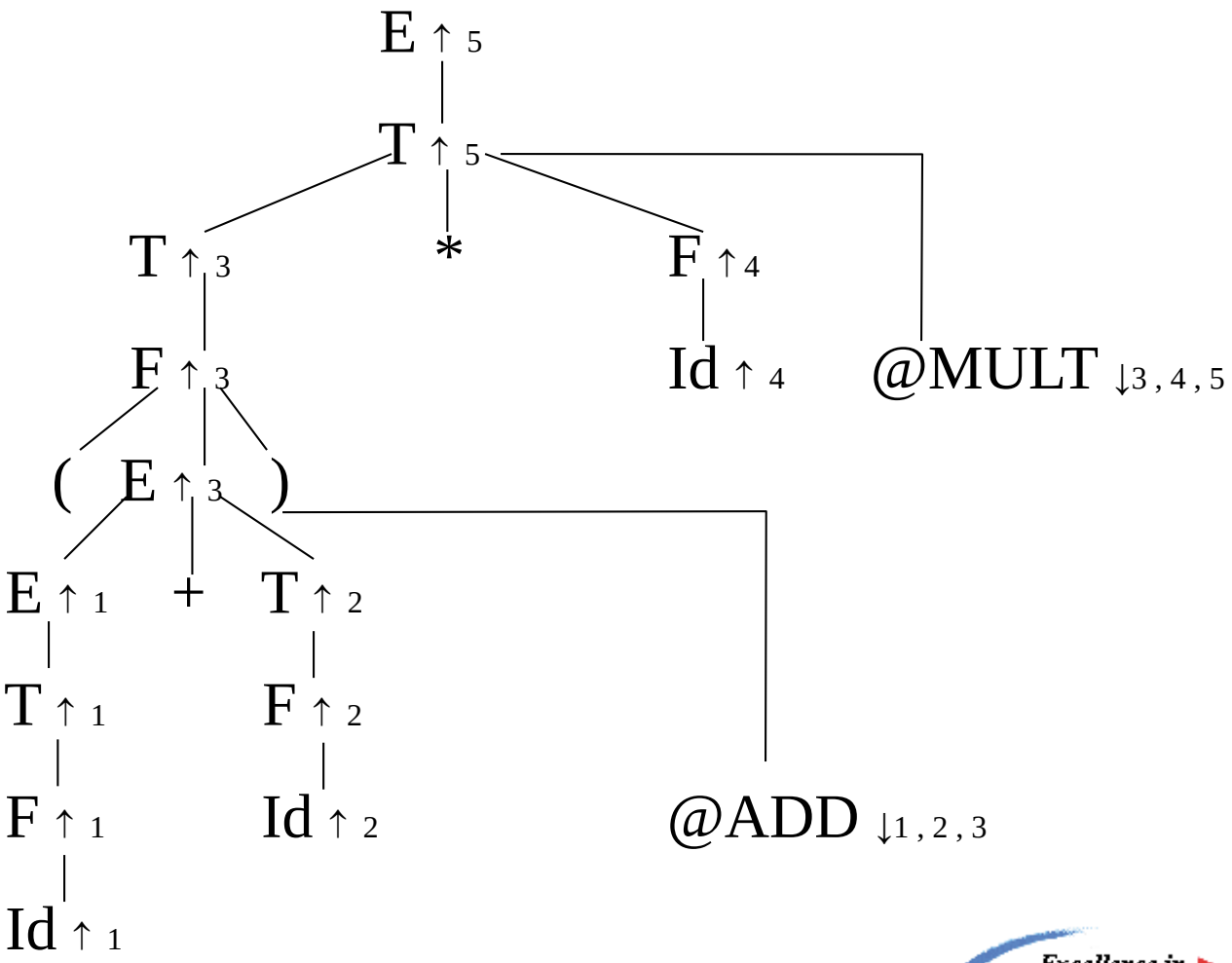
——可将中缀表达式翻译成四元式

1.	$E\uparrow_x \rightarrow E\uparrow_q + T\uparrow_r @ADD_{\downarrow y,z,p}$	x := NEW p := x y := q z := r
2.	$E\uparrow_x \rightarrow T\uparrow_p$	x := p
3.	$T\uparrow_x \rightarrow T\uparrow_q * F\uparrow_r @MULT_{\downarrow y,z,p}$	x := NEW p := x y := q z := r
4.	$T\uparrow_x \rightarrow F\uparrow_p$	x := p
5.	$F\uparrow_x \rightarrow (E\uparrow_p)$	x := p
6.	$F\uparrow_x \rightarrow Id\uparrow_p$	x := p

说明:

Id的综合属性p是数据区地址，NEWT为系统过程，返回数据区地址。

反映属性求值的语法树：



语义动作程序如何设计在后面介绍

@ADD $\downarrow y, z, p \Rightarrow$ fprintf(objfile, “ADD %d %d %d \n”, y, z, p)

(3)写递归下降翻译程序

(留作作业)

作业：P166 1.(前缀式), 2, 3, 4, 5

小结：

本章介绍了语法制导翻译的概念和技术，是在抽象层次上讲的。

第十章对过程语言的编译就是采用该法。

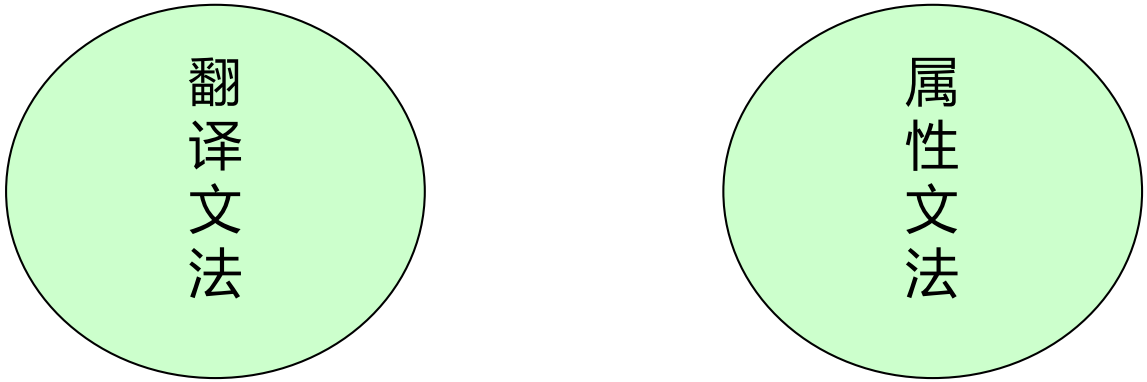
翻译文法：在输入文法中插入语义动作符号（完成翻译任务的语义子程序）

属性翻译文法：文法符号（包括动作符号）可带有属性，并定义相应的属性求值规则，就成为属性翻译文法。比翻译文法能更细地描述翻译过程。（属性有综合属性和继承属性之分）

程序语言的属性翻译文法都是L-属性的，一般无求值规则。

输入文法	翻译文法	符号串翻译文法	属性翻译文法	L-属性翻译文法	SL-属性翻译文法
Vn,Vt P, Z 输入序列	增加动作符号:@ 活动序列 输入序列 动作序列 语法制导翻译	语义子程序是输出动作符号@后的符号串	综合属性 继承属性	LL(1)文法满足属性求值顺序	除动作符号外，其余符号的属性求值规则是复写规则，是属性值或常量

自顶向下的语法制导翻译（递归下降翻译）



在本章的基础上，第十章将介绍典型的过程语言的语法制导翻译。

本章未讲的部分不要求

例子--消除左递归

1. $E \uparrow_x \rightarrow E \uparrow_q + T \uparrow_r @ADD_{\downarrow y,z,p}$
2. $E \uparrow_x \rightarrow T \uparrow_p$
3. $T \uparrow_x \rightarrow T \uparrow_q * F \uparrow_r @MULT_{\downarrow y,z,p}$
4. $T \uparrow_x \rightarrow F \uparrow_p$
5. $F \uparrow_x \rightarrow (E \uparrow_p)$
6. $F \uparrow_x \rightarrow Id \uparrow_p$

右递归:

1. $E\uparrow_x \longrightarrow E\uparrow_q A\downarrow_q$
2. $A\downarrow_q \longrightarrow +T\uparrow_r @ADD_{\downarrow q,r,\uparrow p} A\downarrow_q$
2. $E\uparrow_x \longrightarrow T\uparrow_p$
3. $T\uparrow_x \longrightarrow T\uparrow_q *F\uparrow_r @MULT_{\downarrow y,z,p}$
4. $T\uparrow_x \longrightarrow F\uparrow_p$
5. $F\uparrow_x \longrightarrow (E\uparrow_p)$
6. $F\uparrow_x \longrightarrow Id\uparrow_p$