第九章 语法制导翻译技术

1. 翻译文法(TG)
2. 语法制导翻译
3. 属性翻译文法(ATG)
4. 自顶向下的语法制导翻译
5. 自底向上的语法制导翻译

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，我们应该能够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
- 指称语义
- 公理语义

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 ⟶ 逆波兰表示

a + b * c ⟶ a b c * +

假如我们的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，其后为字符串。

在该具体例示中，其对应语义子程序的功能是要输出打印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　分析 i 输出 i

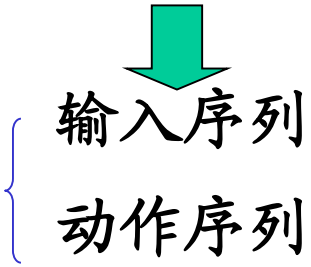
下面给出输入文法和翻译文法的概念:

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



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

例: (i + i) * i

可以用输入文法推出:

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

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

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

用相应的翻译文法推导，可得:

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

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

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

(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 @* ⇒ i i + i *

定义9.1

翻译文法是上下文无关文法，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。

以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：输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 @* → i i+i*

i+i*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，我们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，这样能更好地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]

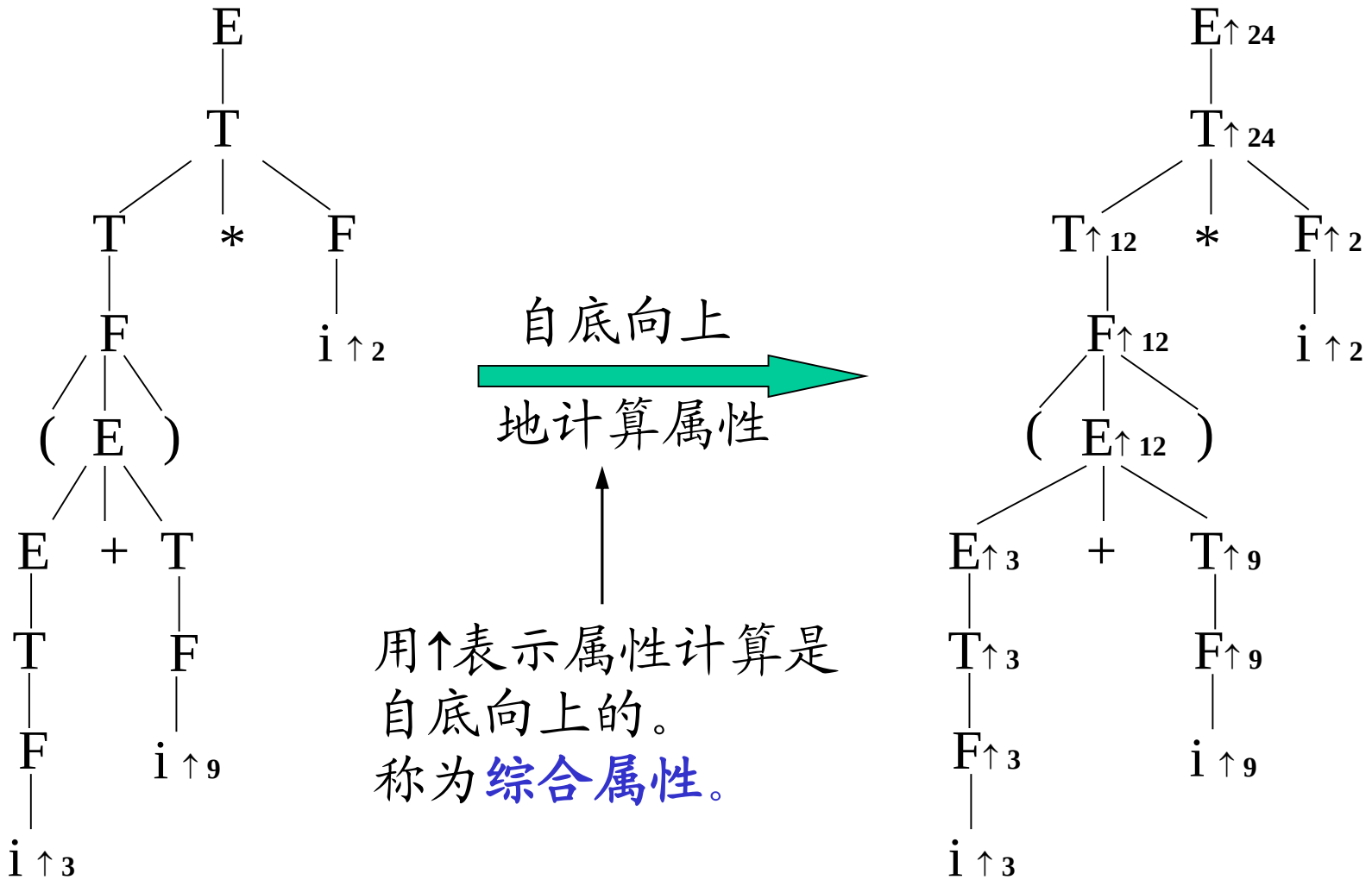
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow i \uparrow_c$

其中$\uparrow_c$是综合属性符号，$\uparrow$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，可写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，我们可以改写上述文法：

产生式	求值规则	
1. $E\uparrow_{p4} \rightarrow E\uparrow_{q5} + T\uparrow_{r2}$	$(q_5 := p_3;)$	$p_4 := q_5 + r_2$;
2. $E\uparrow_{p3} \rightarrow T\uparrow_{q4}$		$p_3 := q_4$;
3. $T\uparrow_{p2} \rightarrow T\uparrow_{q3} * F\uparrow_{r1}$		$p_2 := q_3 * r_1$;
4. $T\uparrow_{p2} \rightarrow F\uparrow_{q2}$		$p_2 := q_2$;
5. $F\uparrow_{p1} \rightarrow (E\uparrow_{q1})$		$p_1 := q_1$;
6. $F\uparrow_{p1} \rightarrow i\uparrow_{q1}$		$p_1 := q_1$;

说明：

- p, q, r 为属性变量名。
- 属性变量名局限于每个产生式，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。

9.2.2 继承属性

考虑到下列文法：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，值为integer, real, boolean等，词法分析程序返回的类型的类别码。

id: 变量（值：标识符本身）

对于上述文法的说明语句：integer A, B1

该文法的翻译任务：将声明的变量填入符号表（语义）

完成该工作的动作符号：@set_table

符号表

A 整型
B1 整型

@set_table的插入位置表示填表动作的时机

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、以及位置（可以用全程变量的指针）如何得到？

终结符（输入符号）的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 中是类型值

id $\uparrow_n$　　n 是变量名

填表动作符号也可带有属性:

@set_table$_{\downarrow t1\ ,\ n1}$　　$\downarrow t_1, n_1$可从其前面的符号得到，称为继承属性，继承前面符号的值。

<变量表>$_{\downarrow t2}$　　$\downarrow t_2$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>$_{\downarrow t1}$ → ε

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> $_{\downarrow t1}$ →ε

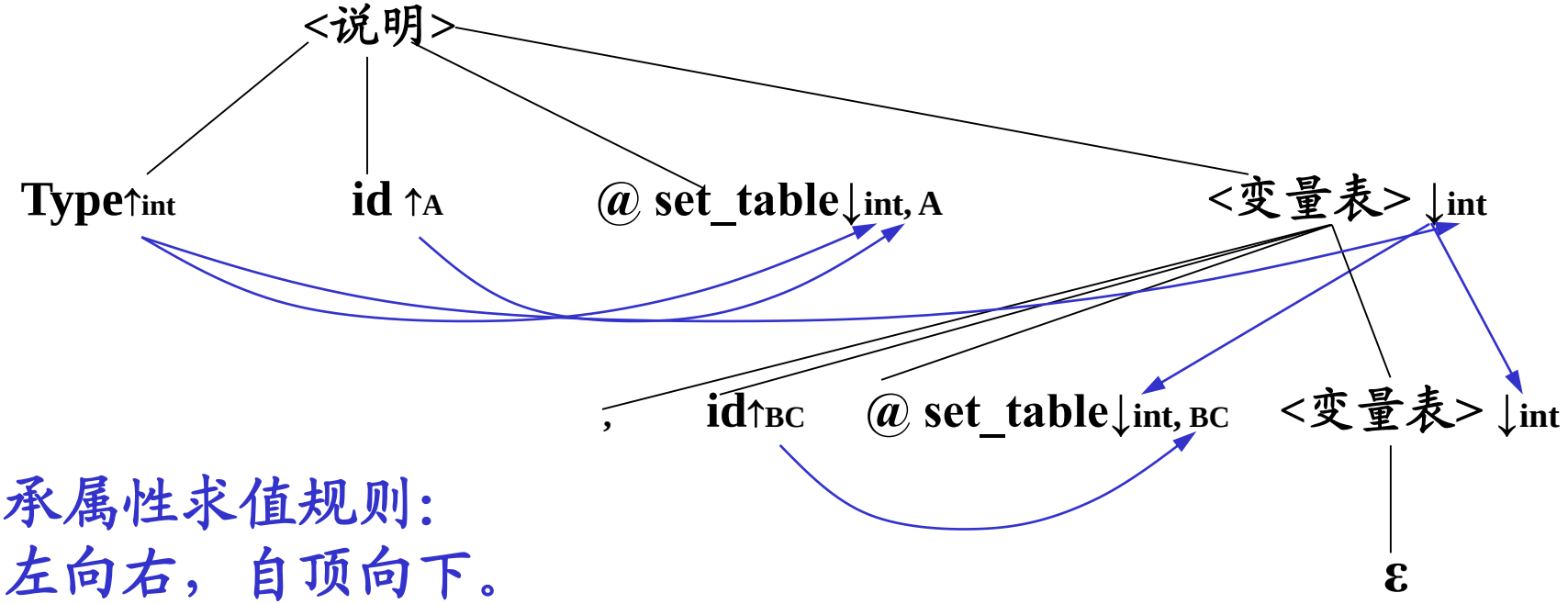
例：int A, BC ⇨ Type $_{\uparrow int}$ id $_{\uparrow A}$， id$_{\uparrow BC}$

<说明> ⇒ Type$_{\uparrow int}$ id$_{\uparrow A}$ @set_table$_{\downarrow int, A}$ <变量表>$_{\downarrow int}$

⇒ Type$_{\uparrow int}$ id$_{\uparrow A}$ @set_table$_{\downarrow int, A}$, id $_{\uparrow BC}$ @set_table$_{\downarrow int, BC}$ <变量表>$_{\downarrow int}$

⇒ Type$_{\uparrow int}$ id$_{\uparrow A}$ @set_table$_{\downarrow int, A}$, id $_{\uparrow BC}$ @set_table$_{\downarrow int, BC}$

语法树:



继承属性求值规则:

自左向右，自顶向下。

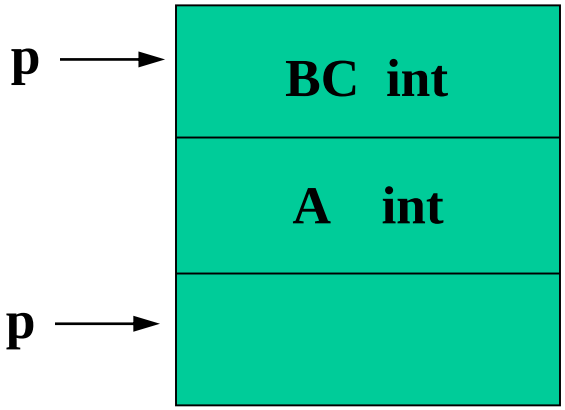
int A, BC 的分析翻译过程:

<说明>

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

⇒ Type $\uparrow_t$ id $\uparrow_{n1}$ @set_table$_{\downarrow t, n1}$, id $\uparrow_{n2}$ @ set_table$_{\downarrow t, n2}$

符号表



9.2.3属性文法的自顶向下翻译

(一) L-属性翻译文法（L-ATG）

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，可用自顶向下分析方法构造分析器。在分析过程中可进行属性求值。

特点：

某个符号的继承属性只依赖于该符号左边的信息!

定义9.2:

L-属性翻译文法是带有下列说明的翻译文法:

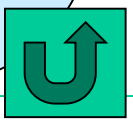
1. 文法中的终结符，非终结符及动作符号都带有属性，且每个属性都有一个值域。
2. 非终结符及动作符号的属性可分为继承属性和综合属性。
3. 开始符号的继承属性具有指定的初始值。
4. 输入符号（终结符号）的每个综合属性具有指定的初始值。
5. 属性值的求值规则如下:

属性求值规则：

继承属性——体现自顶向下，自左向右的求值特性。

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。

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。



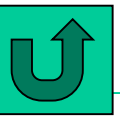
综合属性——体现自底向上，自右向左的求值特性。

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些右部符号的（任意）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些右部符号的（任意）属性进行计算。

适合在自顶向下分析过程中求值。



例：A→BC　　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A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A的继承属性或产生式某些右部符号属性求得）

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些右部符号的（任意）属性进行计算。

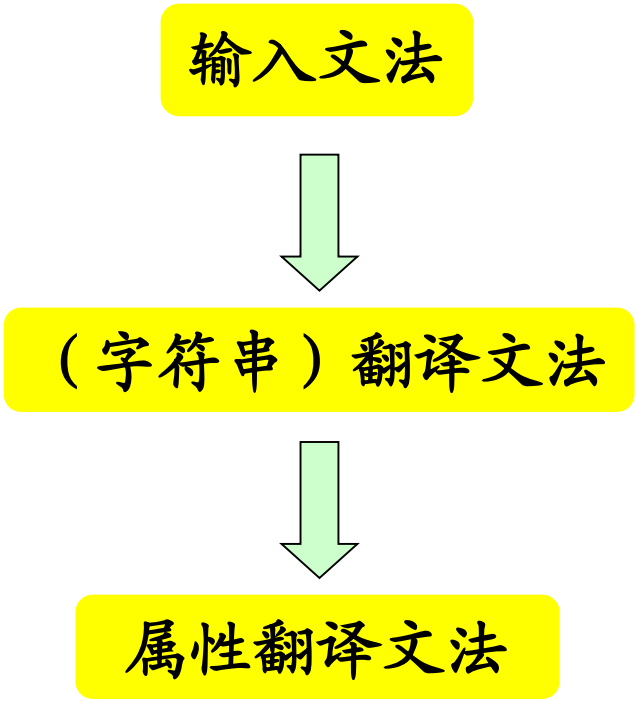
注意：

1) 终结符只有综合属性（它们由词法分析器提供）。
2) 非终结符和动作符号既可以有综合属性也可有继承属性。
3) 文法开始符号的所有继承属性作为属性计算前的初始值。

一般来说，对出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个求值规则。

属性求值规则中只能使用相应产生式中的文法符号的属性（这有助于在产生式范围内“封装”属性的依赖性）。

但出现在产生式左边的继承属性和出现在产生式右边的综合属性不由所给的产生式的属性求值规则进行计算。它们由其它产生式的属性规则计算。


输入文法
（字符串）翻译文法
属性翻译文法

(二) 简单赋值形式的L-属性翻译文法(SL-ATG)

- 一般情况 x := f (y , z) x的属性值是y和z的属性值的函数
- SL-ATG: 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，我们对现有的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部某个符号的综合属性值。

2. 产生式左部非终结符号的综合属性是一个常量，它等于其自身的继承属性值或等于右部某个符号的综合属性值。

目的： 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右部是属性或是常量（——简单化）。

- 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:

$$<A> \rightarrow a\ \uparrow_R <B>\ \uparrow_S <C>\ \downarrow_I \qquad I := f(R, S)$$

显然: 继承属性求值规则不是简单赋值形式的，因为它需要对 f 求值。

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。

$@f\downarrow_{I_1, I_2}\uparrow_{S_1}$　　$S_1 := f(I_1, I_2)$

第二步：修改产生式

1) 插入“@ f”到右部的适当位置
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

$<A> \rightarrow a \uparrow_R <B> \uparrow_S @f \downarrow_{I_1, I_2} \uparrow_{S_1} <C> \downarrow_I$,

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

该文法是简单赋值形式的L-ATG。

9.3 自顶向下语法制导翻译

先介绍翻译文法的自顶向下翻译，然后介绍属性翻译文法的自顶向下翻译。

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，在文法中插入语法动作符号，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. $<S> \longrightarrow a<A><S>$
2. $<S> \longrightarrow b$
3. $<A> \longrightarrow c<A><S>b$
4. $<A> \longrightarrow \varepsilon$

翻译文法（符号串翻译文法）

$<S> \longrightarrow a<A>@x<S>$

$<S> \longrightarrow b@z$

$<A> \longrightarrow c@y<A><S>@ub$

$<A> \longrightarrow @w$

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>@ub
<A> ⟶ @w

主程序

```
NEXTSYM;
PROCS;
if CLASS≠ 右界符 then
   ERROR;
ACCEPT;
过程 PROCS
case CLASS of
a : P1;
b : P2;
其它: ERROR;
end of case;
```

主程序

```
NEXTSYM;
PORCS;
if CLASS≠ # then
   ERROR;
ACCEPT;
过程 PROCS
case CLASS of
a : P1 ;
b : P2 ;
其它: ERROR;
end of case;
```

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>@ub
<A> ⟶ @w

```
P1:          /*产生式的代码*/
     NEXTSYM;
     PROCA;
     PROCS;
     RETURN;
P2:
     NEXTSYM;
     RETURN;

过程 PROCA
        …
```

```
P1:
     NEXTSYM;
     PROCA;
     OUT(x);
     PROCS;
     RETURN;
P2:
     NEXTSYM;
     OUT(z);
     RETURN;
过程 PROCA
        …
```

9.3.2 属性文法自顶向下翻译的实现——递归下降翻译

我们把处理翻译文法的递归下降翻译器进行适当扩展，便可得到处理属性（翻译）文法的递归下降翻译器。

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。

U↓x, ↑y ⟶..　　继承属性：声明为赋值形参

Procedure U (x, y);　　综合属性：声明为变量形参

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（传实参值）

综合属性 ：属性变量名（传地址，返回时有值）

<S> →i ↑x <B>↓y <C>↓z　　y, z := x

<S> →i ↑x <B>↓x <C>↓x

关于属性名的约定:

1) 具有相同值的属性取相同的属性名。

（这样可省去不少属性求值规则）

2) 产生式左部的同名非终结符使用相同的属性名。

（递归下降分析法规定每个非终结符只编写一个子程序!）

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。

<L> ↑a↓b →e ↓I <R>↓J　　<L> ↑x ↓y →e ↓I <R>↓J

<L> ↑x↓y →<H> ↓z ↑w　　<L>↑x ↓y → <H> ↓z↑w

例:

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

$t_2, t_1 := t;$ $n_1 := n;$

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

下面通过一个例子，详细介绍如何构造属性文法的递归下降翻译器。（该文法应是具有L-属性的符号串翻译文法）。

例：有如下属性翻译文法G[<S>]

1. <S> $\downarrow R_1$ ⟶ a$\uparrow T_1$ <A>$\uparrow Q_1$ @x$\downarrow T_2, R_2$ <S>$\downarrow Q_2$

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

2. <S> $\downarrow R_1$ ⟶ b @z$\downarrow R_2$ $R_2 := R_1$

3. <A> $\uparrow P$ ⟶ 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> $\uparrow P$ ⟶ @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 = 7**。

简化后的属性翻译文法G[<S>]

1. $\langle S\rangle\downarrow_R \longrightarrow a\uparrow_T \ \langle A\rangle\uparrow_Q \ @X\downarrow_{T,R} \ \langle S\rangle\downarrow_Q$

2. $\langle S\rangle\downarrow_R \longrightarrow b \ @Z\downarrow_R$

3. $\langle A\rangle\uparrow_P \longrightarrow c\uparrow_U \ @y\downarrow_U \ \langle A\rangle\uparrow_Q \ \langle S\rangle\downarrow_Z \ @v\downarrow_P \ b$

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

4. $\langle A\rangle\uparrow_P \longrightarrow @w$ **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

全局变量的过程声明:

```
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，该符号指针指向下一个单词 */
```

主程序:

```
NEXTSYM;   /* CLASS = 第一个输入符号的类型;
              TOKEN = 第一个输入符号的值; */
PROCS(7);
if CLASS ≠ 右界符 then ERROR;
ACCEPT;
```

过程 PROCS(R)

```
 R;                          /* 值形参声明 */
 case  CLASS  of
   a:     P1 ;
   b:     P2 ;
   其它:  ERROR;
 end  of  case;
P1 :                         /* 产生式 1 的代码 */
 T , Q ;                     /* 局部变量声明 */
 T := TOKEN;                 /* 词法分析得到的单词值，赋给终结符的综合属性 */
 NEXTSYM;
 PROCA(Q)                    /* 这里的Q为变量地址，从A返回时应当有值 */
 OUT( X↓T, R );
 PROCS(Q)                    /* 这里的Q已有具体值 */
 RETURN;
```

1. $\langle S\rangle_{\downarrow R} \rightarrow a_{\uparrow T}\ \langle A\rangle_{\uparrow Q}\ @x_{\downarrow T,R}\ \langle S\rangle_{\downarrow Q}$
2. $\langle S\rangle_{\downarrow R} \rightarrow b\ @z_{\downarrow R}$

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

P2:

```
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. $<A>_{\uparrow P} \rightarrow c_{\uparrow U}\ @y_{\downarrow U}\ <A>_{\uparrow Q}\ <S>_{\downarrow Z}\ @v_{\downarrow P}\ b$
P := Q + U, Z := U - 3

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↓U );
PROCA(Q);                 /* 返回后有值*/
P := Q + U;               /* 插在Q, U已知，使用P之前。 */
PROCS(Z);
OUT( v↓P );
if  CLASS ≠ b   then   ERROR;
NEXTSYM;
RETURN;
```

4. <A>$_{\uparrow P}$ → @w　　P := 8

P4:

P:=8;

OUT(w);

RETURN;

➢ 另一个例子

例：构造将算术表达式翻译成四元组的属性翻译文法，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：四元组　　ADD , P_a , P_b , P_r

P_a , P_b , P_r 为变量a , b和结果单元的地址(由编译分配)。

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，$\uparrow_1$词法综合属性，为变量数据区地址（由编译分配）。

输出：ADD, 1, 2, 3

MULT, 3, 4, 5

数据区：

1	a
2	b
3	部分结果
4	c
5	部分结果

(1) 翻译文法设计:

E →E + T @ADD　　T →F

E →T　　F →(E)

T →T * F @MULT　　F →id

其中: @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
} 对应于完成翻译中语义动作程序

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，也即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

在翻译文法的基础上可设计其属性翻译文法，以便语义分析过程中生成完整的四元式，完成翻译。

- 输入符号（操作数）有一综合属性，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中的地址（中间结果）。
- 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的地址。

这样可得表达式的属性翻译文法

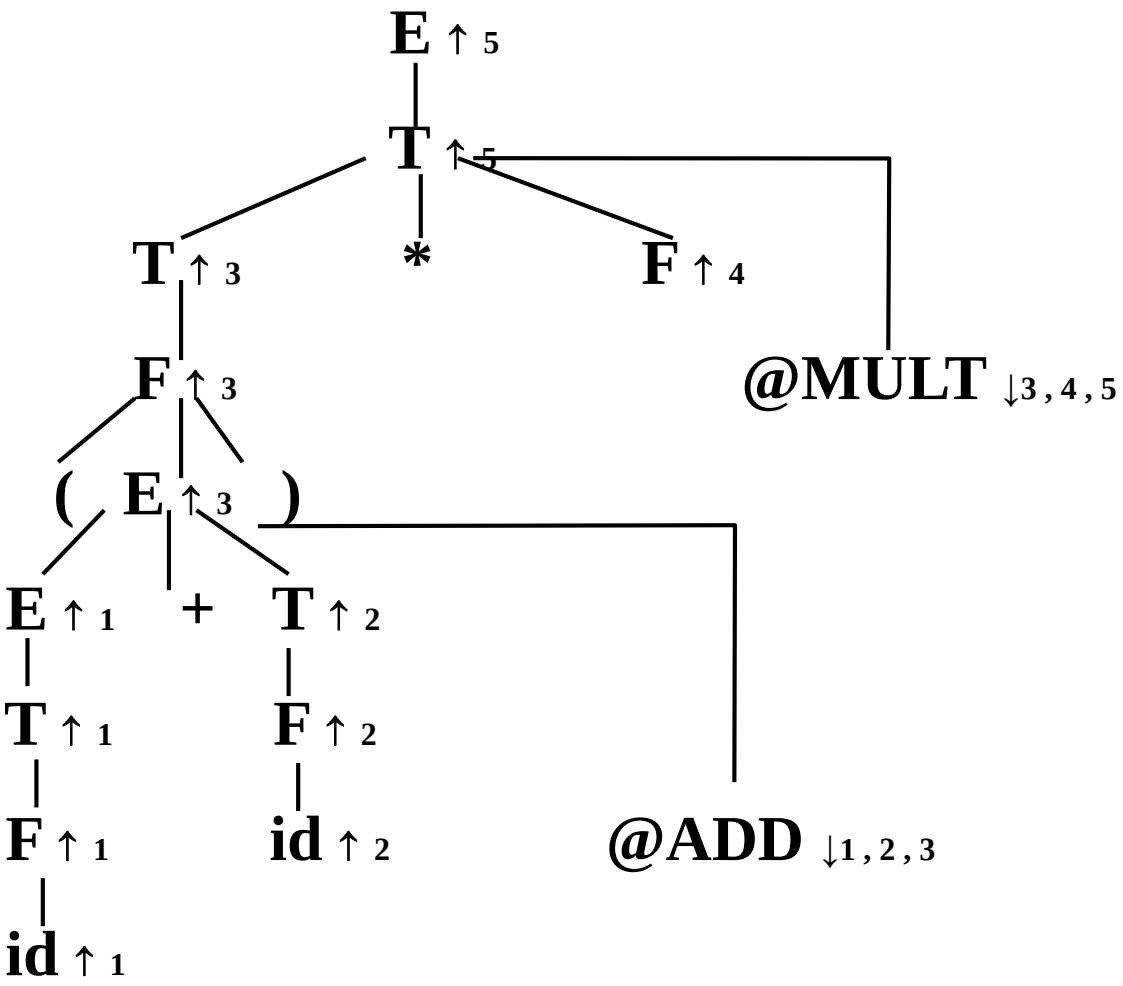
——可将中缀表达式翻译成四元式

1. $E\uparrow_x \longrightarrow E\uparrow_q + T\uparrow_r @ADD\downarrow_{y,z,p}$ x, p := NEW y := q z := r
2. $E\uparrow_x \longrightarrow T\uparrow_P$ x := p
3. $T\uparrow_x \longrightarrow T\uparrow_q *F\uparrow_r @MULT\downarrow_{y,z,p}$ x, p := NEW y := q z := r
4. $T\uparrow_x \longrightarrow F\uparrow_P$ x := p
5. $F\uparrow_x \longrightarrow (E\uparrow_P)$ x := p
6. $F\uparrow_x \longrightarrow id\uparrow_P$ x := p

说明:

id 的综合属性 P 是数据区地址。NEW为系统过程，返回的数据区地址。

反映属性求值的语法树:



语义动作程序

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

(3) 编写递归下降翻译程序

(自己编写！)

本章作业:

P166 第1（只做前缀式），2，3，4，5题

P188 第2、3题