

# 语义规则

- 语义规则也叫语义子程序或语义动作
- 语义规则通常有两种表现形式：
  - 语法制导定义
  - 翻译模式

# 语法制导定义

语法制导定义是关于语言翻译高层次规格说明，它隐藏了具体实现细节，使用户不用显式地说明翻译发生的顺序

例：下面是将中缀表达式转化为后缀表达式的文法和相应的语法制导定义

产生式	语法制导定义
$L \rightarrow E$	<code>print(E.val)</code>
$E \rightarrow E1 + E2$	<code>E.val = E1.val    E2.val    '+'</code>
$E \rightarrow \text{digit}$	<code>E.val = Digit.lexval</code>

语法制导定义 只考虑“做什么”，用抽象的属性表示文法符号所代表的语义

# 翻译方案 *Syntax-Directed Translation Scheme*

- 第二种语义描述方法，也叫翻译模式，缩写为SDT
- 一个翻译方案是一个上下文无关文法，其中被称为语义动作的程序段用花括号 {} 括起来，被嵌入到产生式的右部
- 一个翻译方案类似于语法制导定义，只是语义规则的计算顺序是显式给出的，这样就可以把默写实现细节表示出来
- 画出一个翻译方案的语法分析树时，要同时描述语义信息：  
为每个语义动作分别构造一个额外的子结点，并用虚线将它和该产生式头部对应的结点相连。

# 例

$expr \rightarrow expr_1 + term \text{ \textcolor{red}{\{ print('+') \}}}$

$expr \rightarrow expr_1 - term \text{ \{ print('-') \}}$

$expr \rightarrow term$

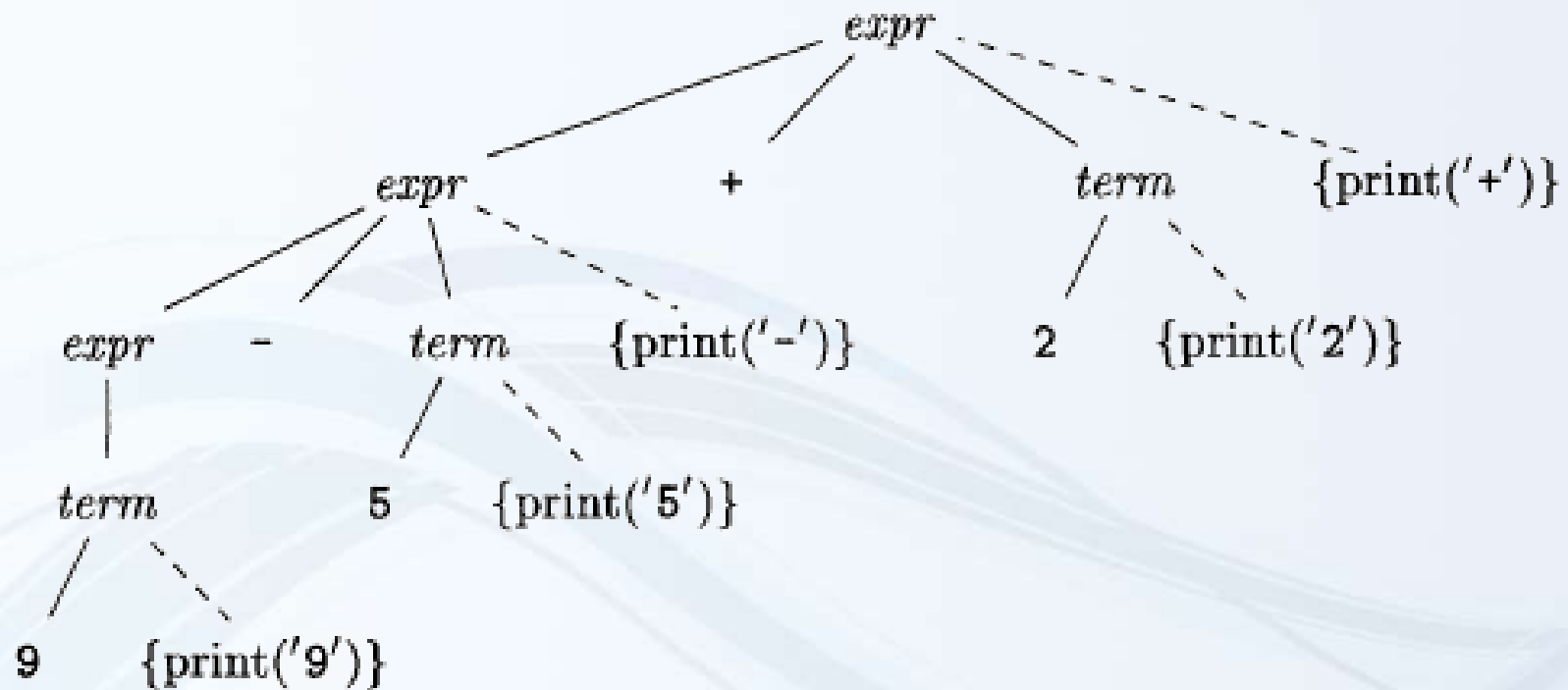
$term \rightarrow 0 \text{ \{ print ('0') \}}$

$term \rightarrow 1 \text{ \{ print('1') \}}$

...

$term \rightarrow 9 \text{ \{ print ('9') \}}$

例



后序遍历，执行语义结点动作

# 语法制导翻译法

- 所谓**语法制导翻译法**，直观地说就是为文法中的每个产生式配上一组语义规则，并且在语法分析的同时执行这些语义规则
- 语义规则被计算的时机
  - ✓ 在自上而下分析中，一个产生式匹配输入串成功时
  - ✓ 在自下而上分析中，当一个产生式被用于进行规约时

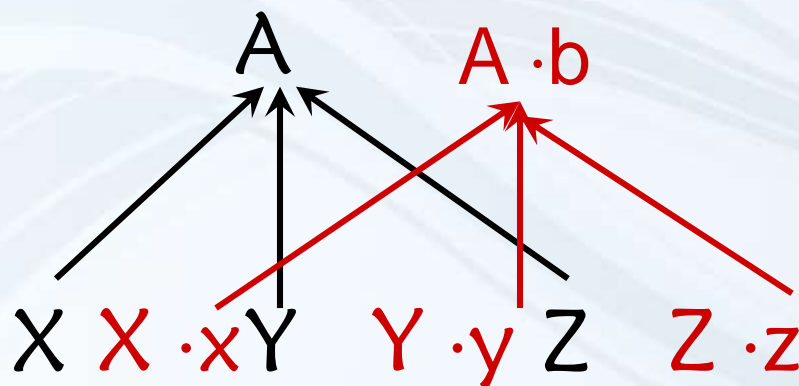
# 移入/归约时的S-属性定义计算

- ❖ **S-属性定义**：只包含综合属性
- ❖ **综合属性**可以在分析输入符号串的同时由自下而上的分析器来计算
- ❖ 分析器可以保存与栈中文法符号有关的**综合属性**值，每当进行归约时，新的属性值就由栈中正在归约的产生式右边符号的属性值来计算



## S-属性定义的计算

- 在分析栈中使用一个附加的域来存放综合属性值。
- 假设语义规则  $A.a = f(X.x, Y.y, Z.z)$  是对应于产生式  $A \rightarrow XYZ$  的



top →

$S_m$	Z	Z.val
$S_{m-1}$	Y	Y.val
$S_{m-2}$	X	X.val
...	...	...

state                  val

归约后，分析栈为：

top →

$S'_{m-2}$	A	A.a

state                  val

$A.a = f(X.x, Y.y, Z.z)$  (抽象)



$\text{val}[ntop] = f(\text{val}[top-2], \text{val}[top-1], \text{val}[top])$  (具体代码)

在执行代码段之前执行:

$ntop := top - r + 1$

其中:  $r$  是句柄的长度,  $ntop$  为归约后栈顶

执行代码段后执行:  $top := ntop;$

产生式

语义规则

例 用LR分析器实现台式计算器

$L \rightarrow E \mathbf{n}$

$print(E.val)$

$E \rightarrow E_1 + T$

$E.val = E_1.val + T.val$

$E \rightarrow T$

$E.val = T.val$

$T \rightarrow T_1 * F$

$T.val = T_1.val * F.val$

$T \rightarrow F$

$T.val = F.val$

$F \rightarrow (E)$

$F.val = E.val$

$F \rightarrow \mathbf{digit}$

$F.val = \mathbf{digit}.lexval$

讨论：为何E、T、F没有代码段？

产生式

代码段

$L \rightarrow E \mathbf{n}$

$print(stack[top-1].val); top=top-1;$

$E \rightarrow E_1 + T$

$stack[top-2].val = stack[top-2].val + stack[top].val; top=top-2;$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$stack[top-2].val = stack[top-2].val * stack[top].val; top=top-2;$

$T \rightarrow F$

$F \rightarrow (E)$

$stack[top-2].val = stack[top-1].val; top=top-2;$

$F \rightarrow \mathbf{digit}$

## 例 翻译输入 $3*5+4n$

输入	state	val	使用的产生式
$3*5+4n$	-	-	
$*5+4n$	3	3	
$*5+4n$	F	3	$F \rightarrow \text{digit}$
$*5+4n$	T	3	$T \rightarrow F$
$5+4n$	$T^*$	3-	
$+4n$	$T^* 5$	3-5	
$+4n$	$T^* F$	3-5	$F \rightarrow \text{digit}$

输入	state	val	使用的产生式
+4n	T	15	$T \rightarrow T * F$
+4n	E	15	$E \rightarrow T$
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	$F \rightarrow \text{digit}$
n	E+T	15-4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	En	19 -	
	L	19	$L \rightarrow En$

## 小结:

采用自底向上分析, 例如LR分析, 首先给出S-属性定义, 然后, 把S-属性定义变成可执行的代码段, 这就构成了翻译程序。这样, 随着语法分析的进行, 归约前调用相应的语义子程序, 在这种分析模式中, 语法分析是主动的, 语义分析是从动的, 语法分析制导着语义分析

## ❖ 语义动作嵌入文法中（与语法制导定义不同）

例

$\text{expr} \rightarrow \text{expr} + \text{term} \{ \text{print}(\text{"+"}) ; \}$

$\text{expr} \rightarrow \text{expr} - \text{term} \{ \text{print}(\text{"-"}) ; \}$

$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow 0 \{ \text{print}(\text{"0"}) ; \}$

...

$\text{term} \rightarrow 9 \{ \text{print}(\text{"9"}) ; \}$

3+5时，输出什么？

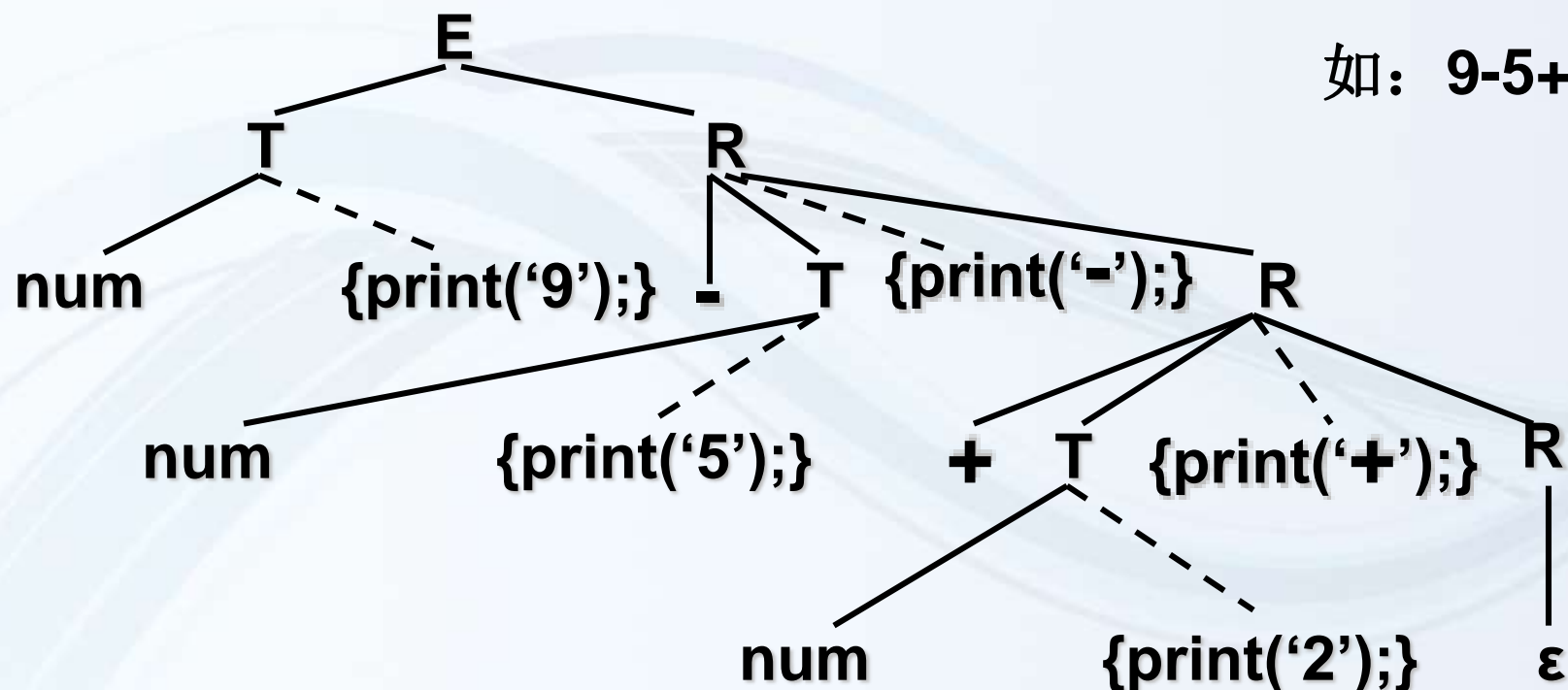


翻译模式示例：把带加号和减号的中缀表达式翻译成相应的后缀表达式

$E \rightarrow TR$

$R \rightarrow +T\{\text{print}(' + '); \}R \mid -T\{\text{print}(' - '); \}R \mid \varepsilon$

$T \rightarrow \text{num}\{\text{print}(\text{num.val}); \}$



# 设计翻译模式原则（根据语法制导定义）

条件：保证语义动作不会引用还没有计算的属性值。**L-属性定义**本身就能确保每个动作不会引用尚未计算出来的属性

■ 只需要**综合属性**时：

为每一个语义规则建立一个包含赋值的动作，并**把这个动作放在相应的产生式右边的末尾**。例如：

$$T \rightarrow T_1 * F \quad T.\text{val} := T_1.\text{val} * F.\text{val}$$
$$T \rightarrow T_1 * F \quad \{ T.\text{val} := T_1.\text{val} * F.\text{val} \}$$

# 例

错误:

$S \rightarrow A_1 \{S.s = A_1.s + A_2.s\} A_2$

$A \rightarrow \mathbf{a} \{A.s = 1\}$



综合属性应该在后

正确:

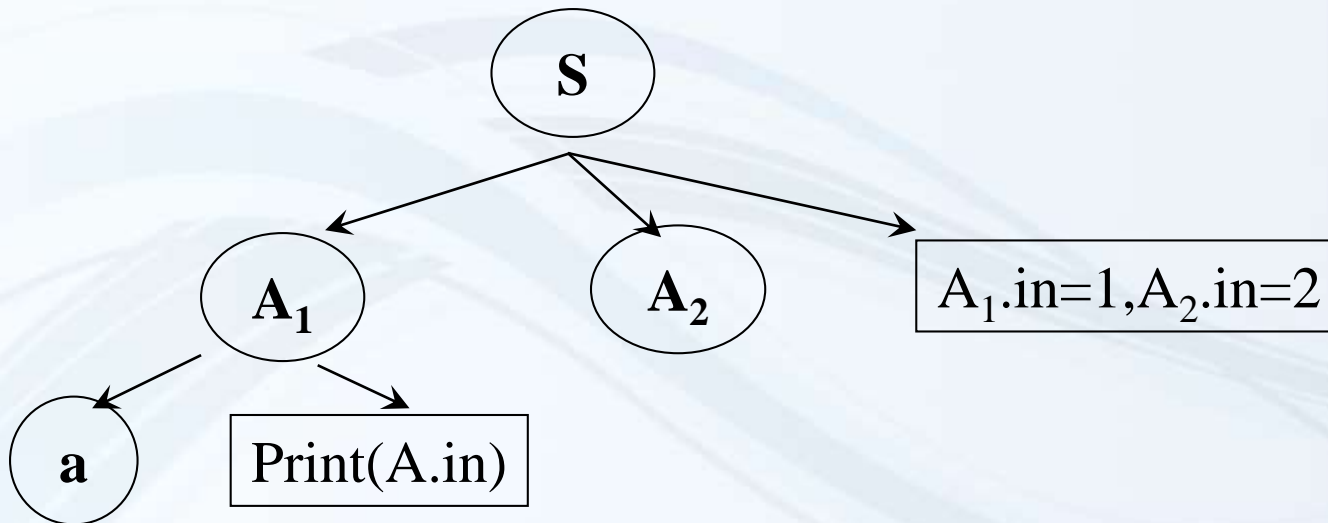
$S \rightarrow A_1 A_2 \{S.s = A_1.s + A_2.s\}$

$A \rightarrow \mathbf{a} \{A.s = 1\}$

- 如果既有综合属性又有继承属性
  - 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。
  - 一个动作不能引用这个动作右边符号的综合属性。
  - 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾。

例

$S \rightarrow A_1 A_2$    $\{A_1.in:=1; A_2.in:=2\}$   
 $A \rightarrow a$    $\{\text{print}(A.in)\}$




错误:

$S \rightarrow A_1 A_2 \{A_1.in = 1; A_2.in = 2\}$

$A \rightarrow a \{print(A.in)\}$

继承属性应该在前



正确:

$S \rightarrow \{A_1.in = 1; A_2.in = 2\} A_1 A_2$

$A \rightarrow a \{print(A.in)\}$

也正确:

$S \rightarrow \{A_1.in = 1\} A_1 \{A_2.in = 2\} A_2$

$A \rightarrow a \{print(A.in)\}$


# 自顶向下翻译

- 动作是在处于相同位置的符号被展开（匹配成功）时执行的
- 为了构造不带回溯的自顶向下语法分析，必须消除文法中的左递归，没有左公共因子
- 当消除一个翻译模式的基本文法的左递归时，同时考虑**属性**——适合带**综合属性**的翻译模式

# ◆ 消除左递归

例：关于算术表达式的左递归文法相应的翻译模式：

$E \rightarrow E_1 + T$	$\{E.val = E_1.val + T.val\}$
$E \rightarrow E_1 - T$	$\{E.val = E_1.val - T.val\}$
$E \rightarrow T$	$\{E.val = T.val\}$
$T \rightarrow (E)$	$\{T.val = E.val\}$
$T \rightarrow \text{num}$	$\{T.val = \text{num}.val\}$



$E \rightarrow T R$
$R \rightarrow + T R_1$
$R \rightarrow - T R_1$
$R \rightarrow \varepsilon$
$T \rightarrow ( E )$
$T \rightarrow \text{num}$



## ■ 消除左递归，构造新的翻译模式：

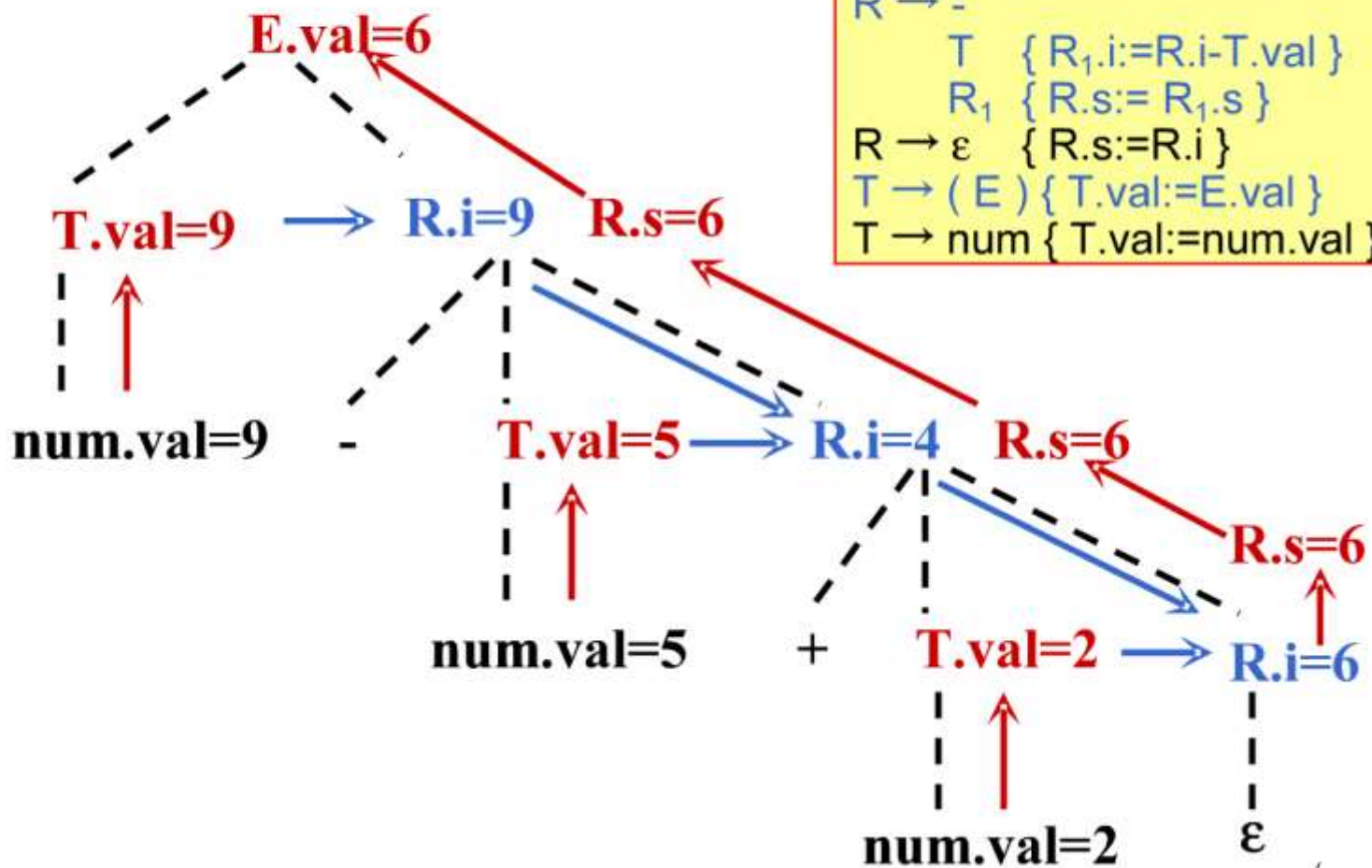
$$\begin{array}{ll}
 E \rightarrow T & \{R.i = T.val\} \\
 R & \{E.val = R.s\} \\
 R \rightarrow + & \\
 T & \{R_1.i = R.i + T.val\} \\
 R_1 & \{R.s = R_1.s\} \\
 R \rightarrow - & \\
 T & \{R_1.i = R.i - T.val\} \\
 R_1 & \{R.s = R_1.s\} \\
 R \rightarrow \varepsilon & \{R.s = R.i\} \\
 T \rightarrow ( E ) & \{T.val = E.val\} \\
 T \rightarrow num & \{T.val = num.val\}
 \end{array}$$

$$\begin{array}{l}
 E \rightarrow T R \\
 R \rightarrow +TR_1 \\
 R \rightarrow -TR_1 \\
 R \rightarrow \varepsilon \\
 T \rightarrow ( E ) \\
 T \rightarrow num
 \end{array}$$

R.i: R前面子表达式的值

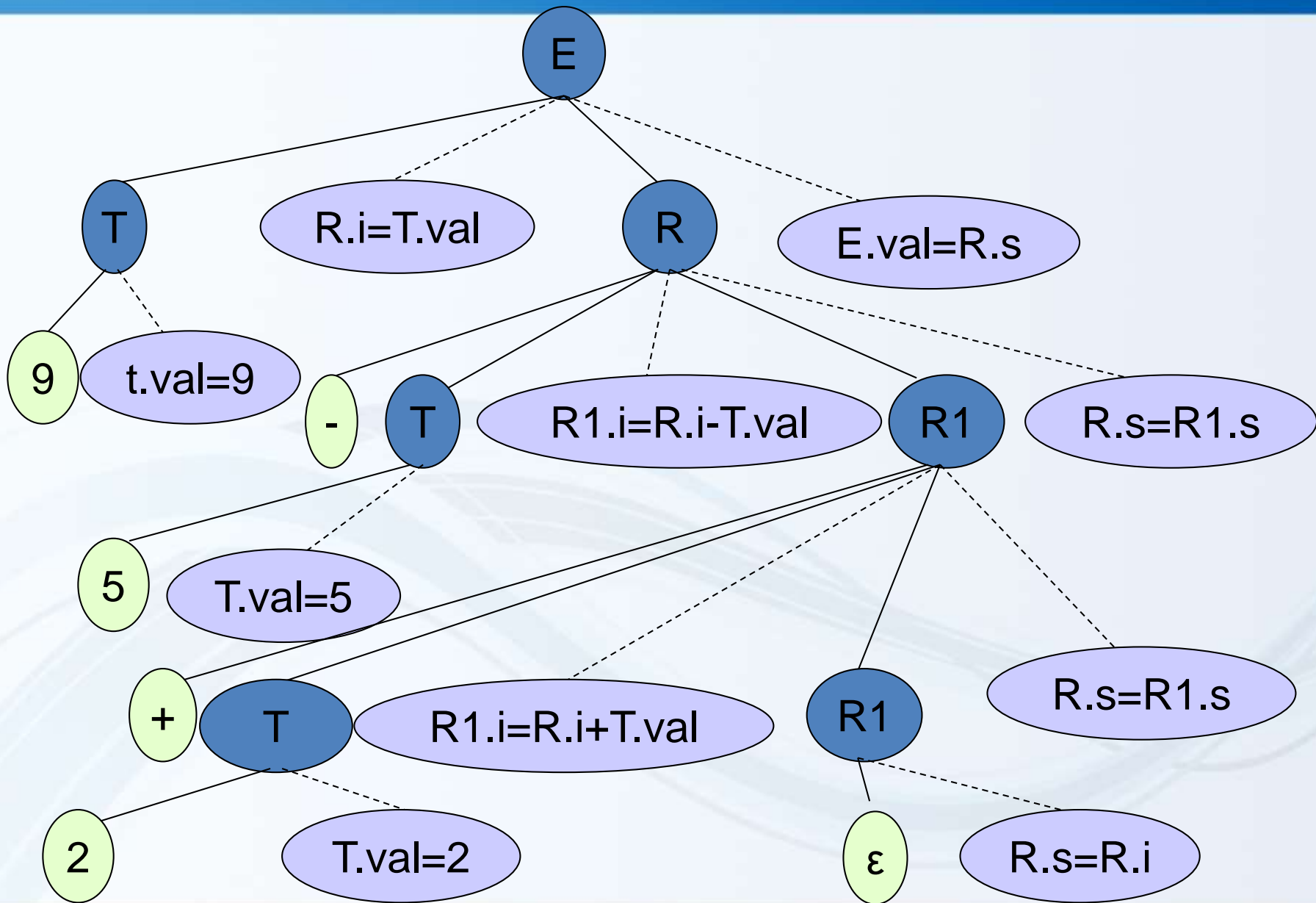
R.s: 分析完R时子表达式的值

# 计算表达式 9 - 5 + 2



```

E → T { R.i:=T.val }
      R { E.val:=R.s }
R → +
      T { R1.i:=R.i+T.val }
      R1 { R.s:= R1.s }
R → -
      T { R1.i:=R.i-T.val }
      R1 { R.s:= R1.s }
R → ε { R.s:=R.i }
T → ( E ) { T.val:=E.val }
T → num { T.val:=num.val }
    
```



# ◆ 消除左递归的一般方法

含有综合属性的翻译方案（语法有左递归）

$$A \rightarrow A_1 Y \{A.a = g(A_1.a, Y.y)\}$$

$$A \rightarrow X \{A.a = f(X.x)\}$$

它的每个文法符号都有一个综合属性，用小写字母表示， $g$ 和 $f$ 是任意函数

□ 消除左递归：

$$A \rightarrow XR$$

$$R \rightarrow YR \mid \varepsilon$$

$R.i$ :  $R$ 前面子表达式的值

$R.s$ : 分析完 $R$ 时子表达式的值

□ 翻译模式变为：

$$A \rightarrow X \{R.i = f(X.x)\}$$

$$R \{A.a = R.s\}$$

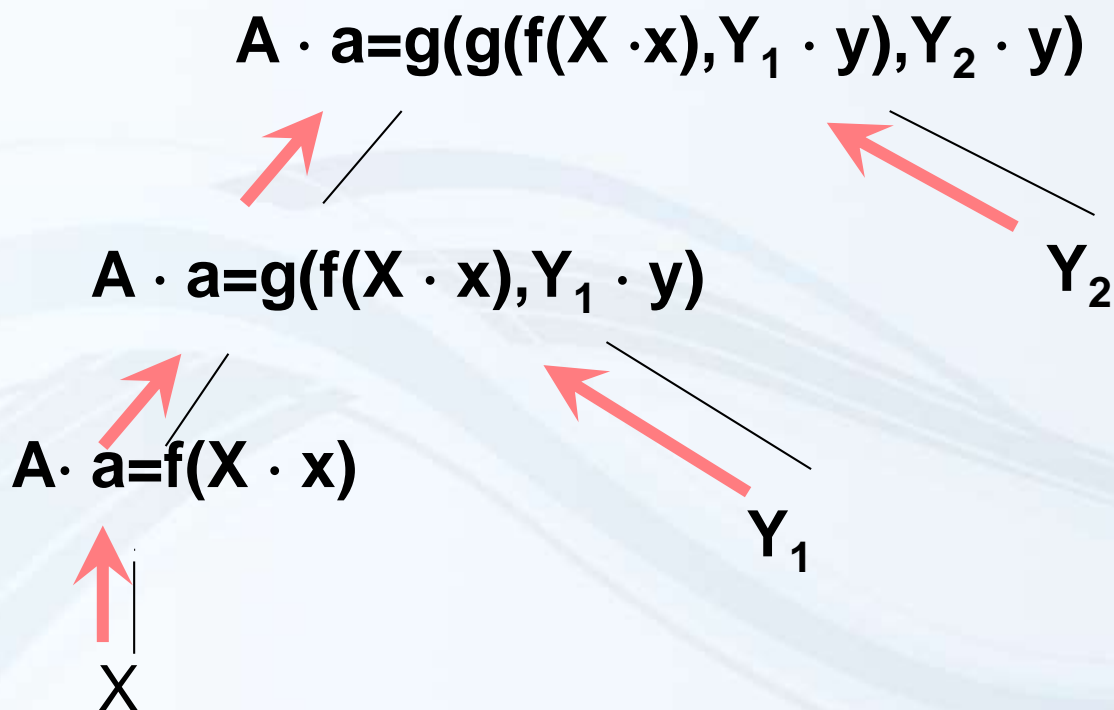
$$R \rightarrow Y \{R_1.i = g(R.i, Y.y)\}$$

$$R_1 \{R.s = R_1.s\}$$

$$R \rightarrow \varepsilon \{R.s = R.i\}$$

## 例：句子XYY带左递归的语法树

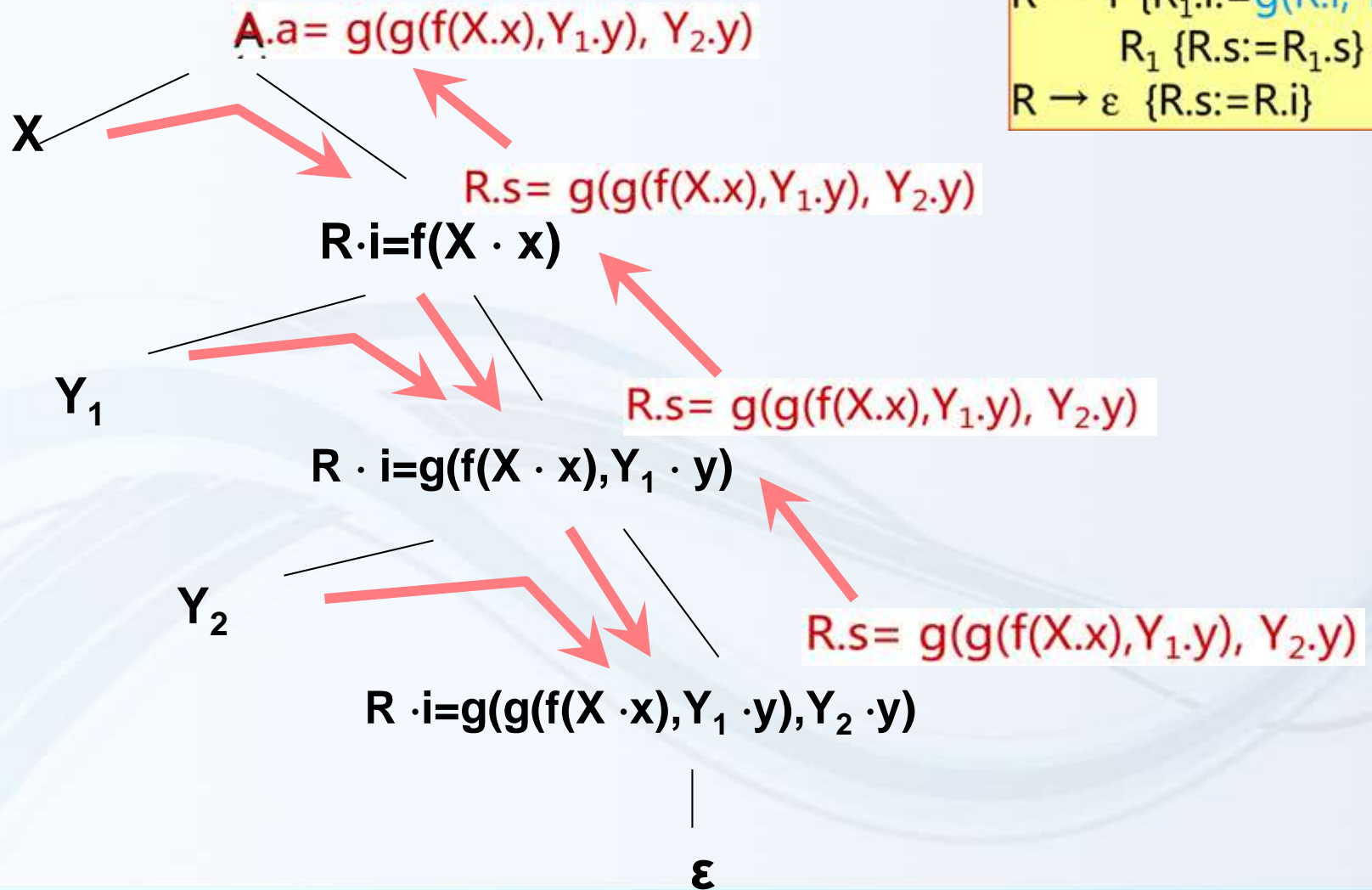
$A \rightarrow A_1 Y$	$\{A.a := g(A_1.a, Y.y)\}$
$A \rightarrow X$	$\{A.a := f(X.x)\}$



# 例：句子XYY消除左递归的语法树

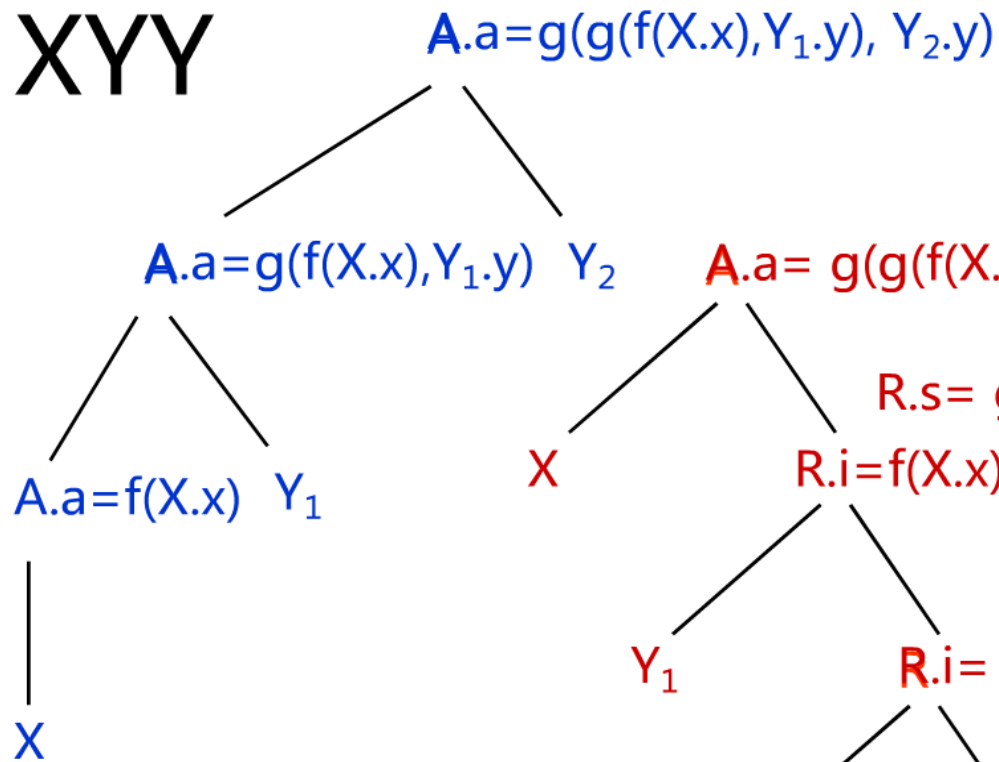
```

A → X {R.i:=f (X.x) }
      R {A.a:=R.s}
R → Y {R1.i:=g(R.i, Y.y)}
      R1 {R.s:=R1.s}
R → ε {R.s:=R.i}
    
```

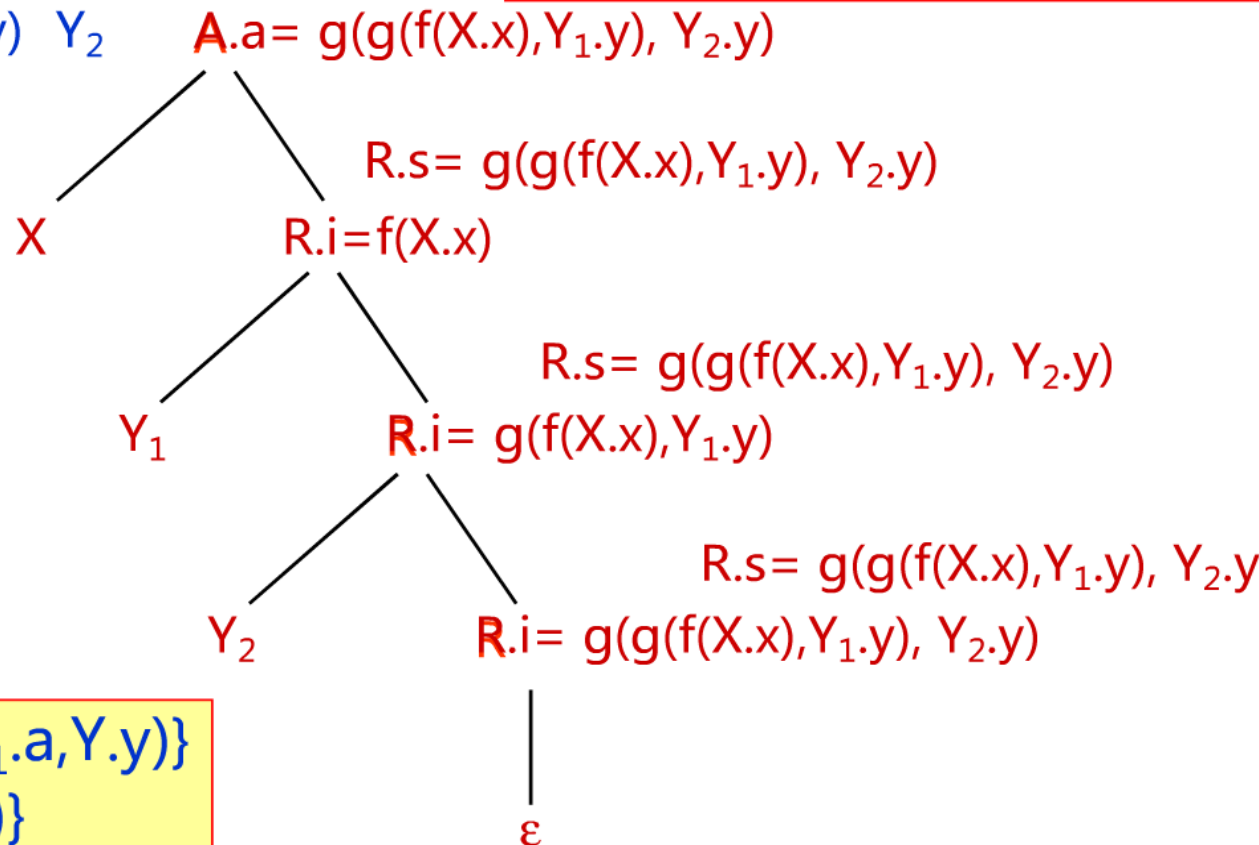




XY<sub>1</sub>Y<sub>2</sub>



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



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

# 例

## 构造抽象语法树的属性文法定义转化成翻译模式

$E \rightarrow E_1 + T \quad \{E.nptr := mknnode( \text{'+'}, E_1.nptr, T.nptr) \}$

$E \rightarrow E_1 - T \quad \{E.nptr := mknnode( \text{'-'}, E_1.nptr, T.nptr) \}$

$E \rightarrow T \quad \{E.nptr := T.nptr\}$

$A \rightarrow A_1 Y \quad \{A.a := g(A_1.a, Y.y) \}$

$A \rightarrow X \quad \{A.a := f(X.x) \}$

$A \rightarrow X \quad \{R.i := f(X.x) \}$

$R \quad \{A.a := R.s\}$

$R \rightarrow Y \quad \{R_1.i := g(R.i, Y.y) \}$

$R_1 \quad \{R.s := R_1.s\}$

$R \rightarrow \varepsilon \quad \{R.s := R.i\}$

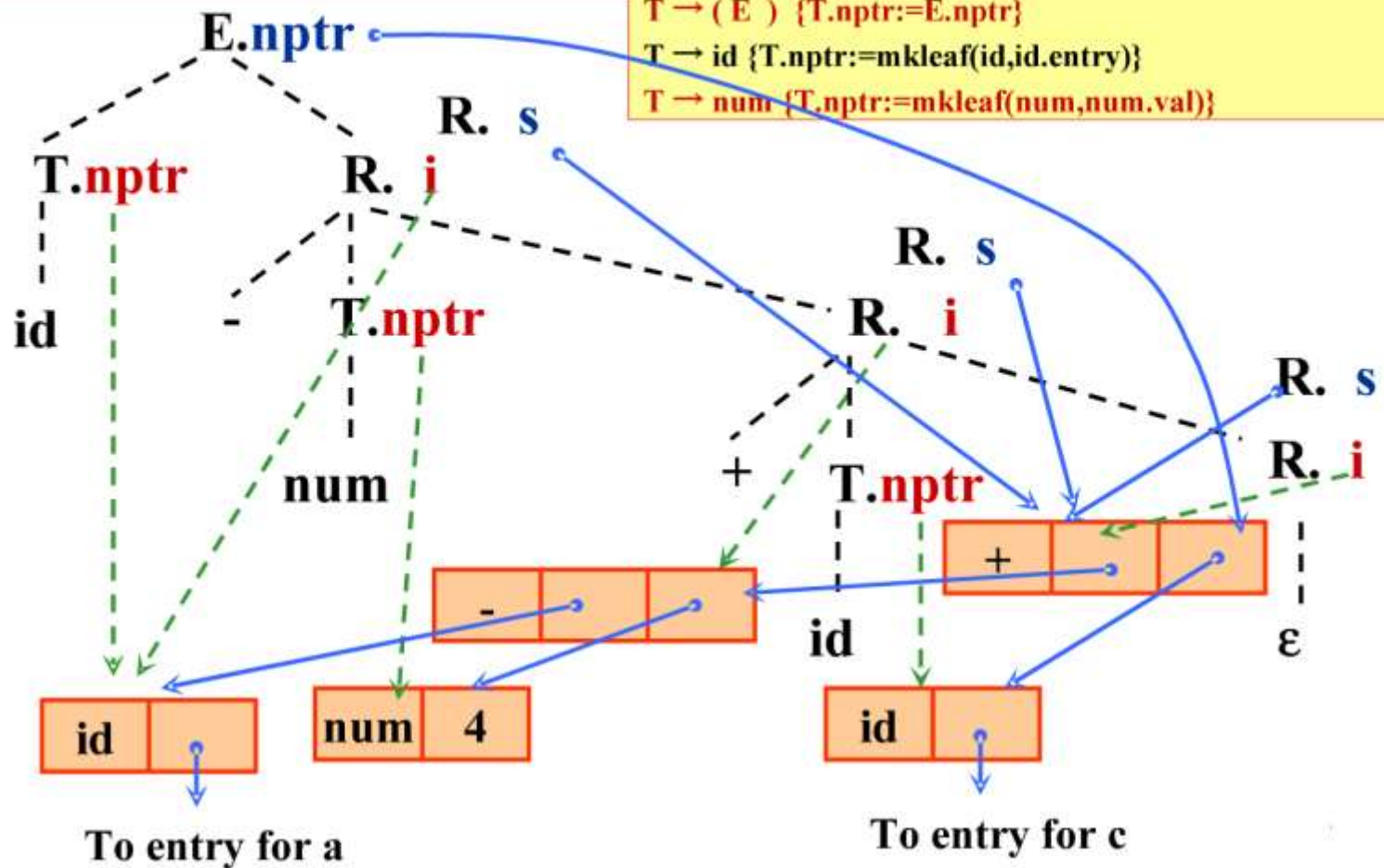


## 构造抽象语法树的翻译模式

$E \rightarrow$	$T$	$\{R.i := T.nptr\}$
	$R$	$\{E.nptr := R.s\}$
$R \rightarrow$	$+$	
	$T$	$\{R_1.i := \text{mknode}( ' + ' , R.i, T.nptr )\}$
	$R_1$	$\{R.s := R_1.s\}$
$R \rightarrow$	$-$	
	$T$	$\{R_1.i := \text{mknode}( ' - ' , R.i, T.nptr )\}$
	$R_1$	$\{R.s := R_1.s\}$
$R \rightarrow$	$\epsilon$	$\{R.s := R.i\}$
$T \rightarrow$	$( E )$	$\{T.nptr := E.nptr\}$
$T \rightarrow$	$\text{id}$	$\{T.nptr := \text{mkleaf}(\text{id}, \text{id.entry})\}$
$T \rightarrow$	$\text{num}$	$\{T.nptr := \text{mkleaf}(\text{num}, \text{num.val})\}$

# 使用继承属性构造 a - 4 + c 的抽象语法树

$E \rightarrow T \{R.i := T.nptr\} \quad R \{E.nptr := R.s\}$   
 $R \rightarrow + T \{R_1.i := \text{mknode}('+', R.i, T.nptr)\} \quad R_1 \{R.s := R_1.s\}$   
 $R \rightarrow - T \{R_1.i := \text{mknode}('-', R.i, T.nptr)\} \quad R_1 \{R.s := R_1.s\}$   
 $R \rightarrow \epsilon \{R.s := R.i\}$   
 $T \rightarrow ( E ) \{T.nptr := E.nptr\}$   
 $T \rightarrow \text{id} \{T.nptr := \text{mkleaf}(\text{id}, \text{id.entry})\}$   
 $T \rightarrow \text{num} \{T.nptr := \text{mkleaf}(\text{num}, \text{num.val})\}$



# ■ 递归下降分析器的设计

## ❖ 扩展分析器（设 $P$ 为分析非终结符 $A$ 的过程）

- $P$ 的参数中，加上 $A$ 的继承属性
- $P$ 的返回值，为 $A$ 的综合属性
- 过程体：以局部变量形式保存
  - ✓ 过程体涉及到的非终结符的属性（继承属性和综合属性）
  - ✓ 为了计算这些属性值所需的其它变量

方法：在预测分析器中加入语义动作代码。

1. 对每个非终结符A建立一个可递归调用的函数过程。

- 为A的每一个继承属性都设置一个形式参数
- 函数的返回值是A的综合属性。
  - 作为记录，或指向记录的一个指针，记录汇总有若干域，每个属性对应一个域

2. A对应的函数过程中为出现在A的产生式中的每一个文法符号的每一个属性都设置一个局部变量

3. 每一个产生式对应的代码，按照从左到右的顺序，根据产生式右部是**单词符号（终结符）**、**非终结符**还是**语义动作**，分别做以下工作：

（a）对于带有综合属性 $x$ 的**单词符号 $X$** ，把 $x$ 的值存入为 $X.x$ 设置的变量中，再 $\text{Match}(X)$ 。并继续读入一个输入符号。

（b）对于每个非终结符 $B$ ，编写一个赋值语句

$c := B(b_1, b_2, \dots, b_k)$ ，其中， $b_1, b_2, \dots, b_k$ 是为 $B$ 的**继承属性**设置的变量， $c$ 是为 $B$ 的**综合属性**设置的变量。

（c）对于每个**语义动作**，把动作的代码抄进语法分析器中，用代表属性的变量来代替对属性的每一次引用。

# 递归下降分析（例）

$S \rightarrow \text{while}(C)S_1$

$\{L_1 = \text{new}(\ );$

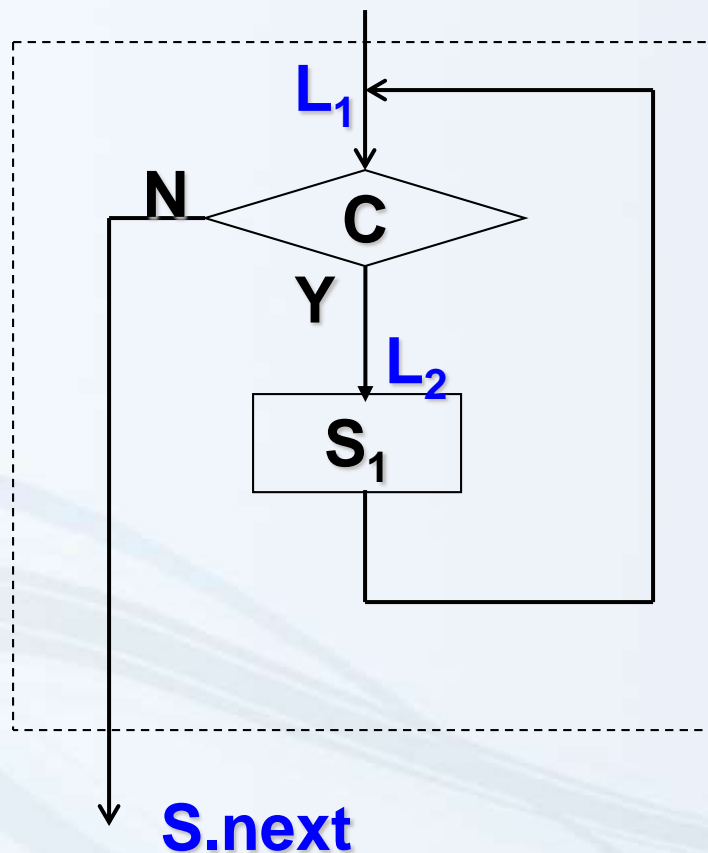
$L_2 = \text{new}(\ );$

$S_1.\text{next} = L_1;$

$C.\text{false} = S.\text{next}$

$C.\text{true} = L_2;$

$S.\text{code} = \text{label} \parallel L_1 \parallel C.\text{code} \parallel \text{label} \parallel L_2 \parallel S_1.\text{code}\}$



虚线框  
内有几个  
出口，  
需要引  
入标记  
吗？

# 递归下降分析（while语句语法处理的代码）

❖  $S \rightarrow \text{while}(C)S_1$

```
void S( ){  
    if(current input == token while){  
        advance input;  
        check '(' is next on the input , and advance;  
        C();  
        check ')' is next on the input , and advance;  
        S();  
    }  
    else /* other statement types*/  
}
```



❖  $S \rightarrow \text{while}(C)S_1$

```
string S(label next){  
    string S1code, Ccode;  
    label L1, L2;
```

```
    if(current input == token while){  
        advance input;  
        check '(' is next on the input, and advance;  
        L1 = new( );  
        L2 = new( );  
        Ccode = C(next, L2)  
        check ')' is next on the input , and advance;  
        S1code = S(L1);  
        return ("label"||L1||Ccode || "label" || L2 ||S1code);
```

```
    }
```

```
    else /* other statement types*/
```

```
}
```

```
L1 = new();          L2 = new()  
S1.next = L1;  
C.false = S.next  
C.true = L2;  
S.code = label||L1||C.code||label ||L2||S1.code
```

一次性输出可能需要  
非常大的存储空间



❖  $S \rightarrow \text{while}(C)S_1$

❖ **void** S(label **next**) {

label L1, L2;

if(current input == token **while**) {

advance input;

check '(' is next on the input , and advance;

L1 = new();

L2 = new();

**print**("label", L1);

**C**(next, L2);

check ')' is next on the input , and advance;

**print**("label" , L2);

**S**(L1);

}

else /\* other statement types\*/

}

$S \rightarrow \text{while} ( \{L1=\text{new}(); L2=\text{new}();$   
 $C.\text{false}=S.\text{next}; C.\text{true}=L2; \text{print}(\text{"label"}, L1);\}$   
 $C ) \{ \text{print} (\text{"label"}, L2); S1.\text{next}=L1; \} S1$

立即输出不需要太大  
存储空间

# 自底向上分析

❖ 自底向上分析显然可以处理S属性定义

❖ L属性定义要解决的问题:

➤ 嵌入语义动作的处理

✓ 引入标记非终结符号M（也称为占位符）和规则 $M \rightarrow \epsilon$ ，将原语义动作与规则 $M \rightarrow \epsilon$ 结合

✓ 去掉原来的嵌入语义动作

➤ 继承属性的处理

✓ 用综合属性代替继承属性

## ■ 自底向上分析（嵌入语义动作的处理）

$E \rightarrow T R$

$R \rightarrow + T \{\text{print}('+');\} R_1 \mid - T \{\text{print}('-');\} R_1 \mid \varepsilon$

$T \rightarrow \text{num} \{\text{print}(\text{num.val});\}$



$E \rightarrow T R$

$R \rightarrow + T M R_1 \mid - T N R_1 \mid \varepsilon$

$T \rightarrow \text{num} \{\text{print}(\text{num.val});\}$

$M \rightarrow \varepsilon \{\text{print}('+');\}$

$N \rightarrow \varepsilon \{\text{print}('-');\}$

# ■ 自底向上分析（继承属性处理）

## ❖ 规则右部文法符号继承属性的处理

- 参照嵌入语义动作处理

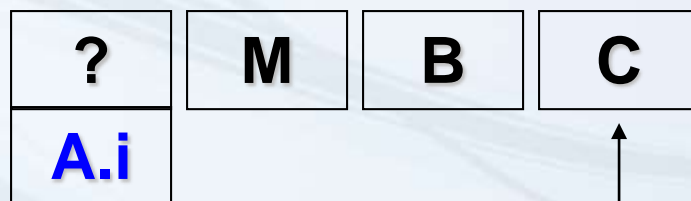
## ❖ 规则左部文法符号继承属性的处理

- 约定：信息直接埋入栈内该文法符号下面一个符号的属性值内

$A \rightarrow \{B.i=f(A.i);\} B C$

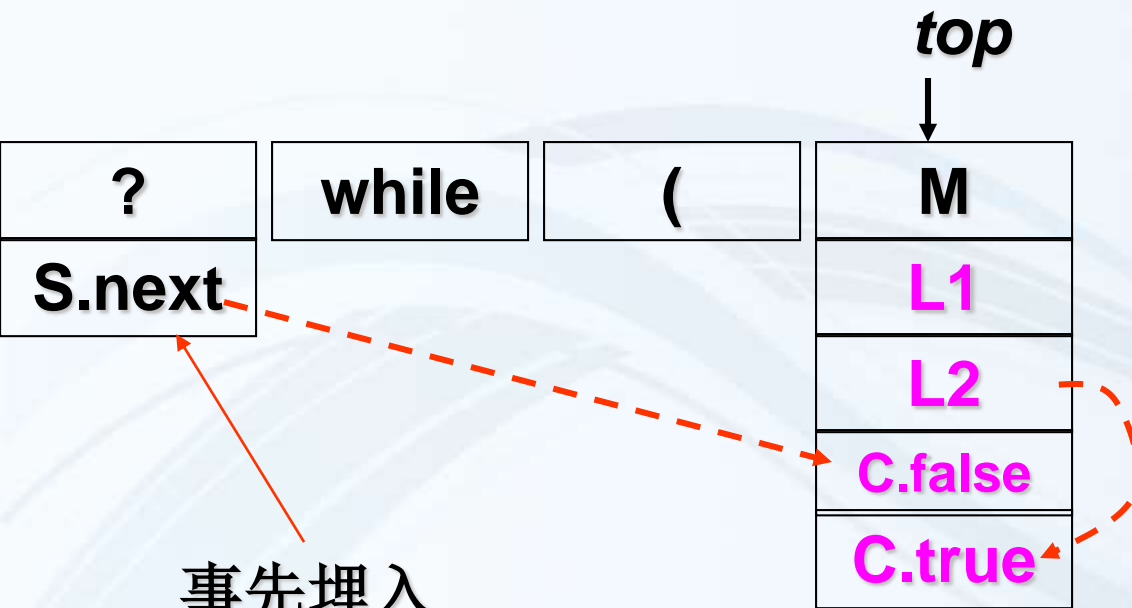


$A \rightarrow M B C$   
 $M \rightarrow \epsilon \{M.s=f(A.i);\}$



↑  
栈顶

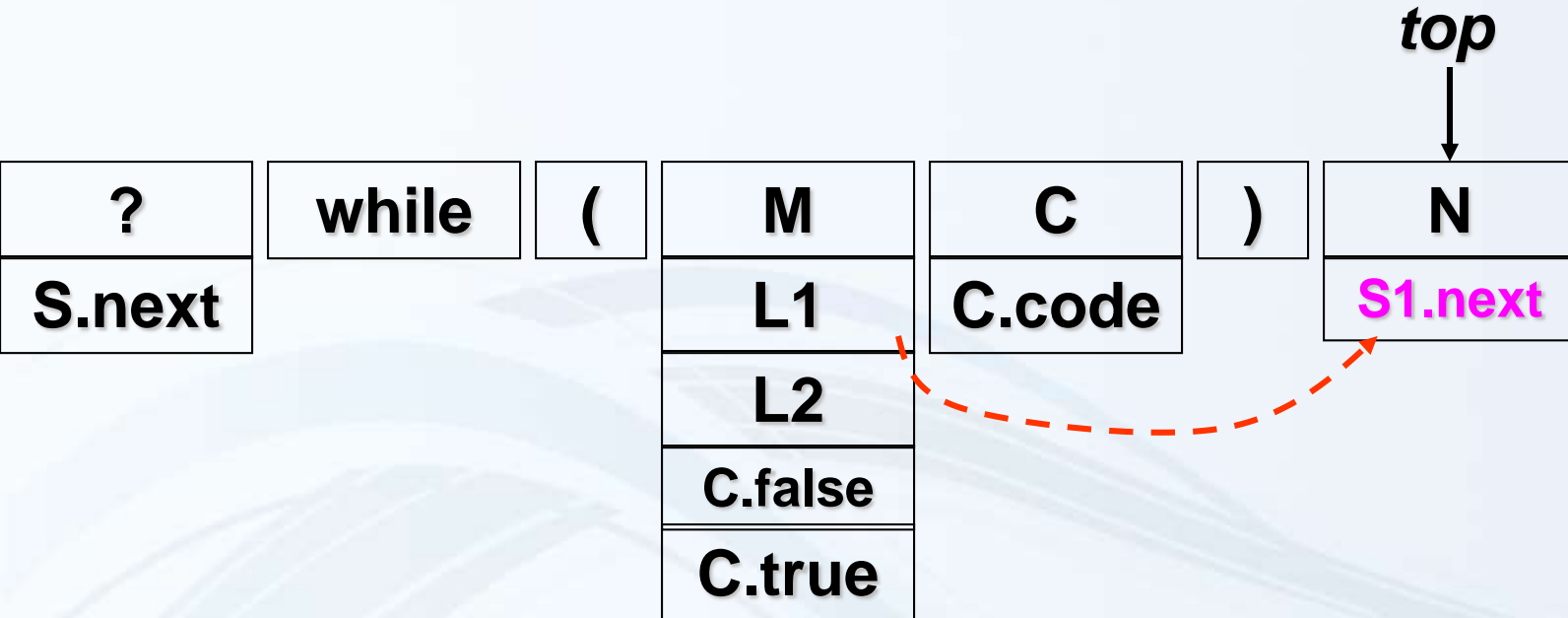
$S \rightarrow \text{while} ( M \quad C ) \quad N \quad S_1$   
 $M \rightarrow \varepsilon \quad \{L_1 = \text{new}(); L_2 = \text{new}(); \text{C.false} = \text{stack}[\text{top}-3].\text{next}; \text{C.true} = L_2\}$   
 $N \rightarrow \varepsilon \quad \{S_1.\text{next} = \text{stack}[\text{top}-3].L_1\}$



$S \rightarrow \text{while} ( M \quad C ) \quad N \quad S1$

$M \rightarrow \epsilon \quad \{L1=new(); L2=new(); C.false=stack[top-3].next; C.true=L2\}$

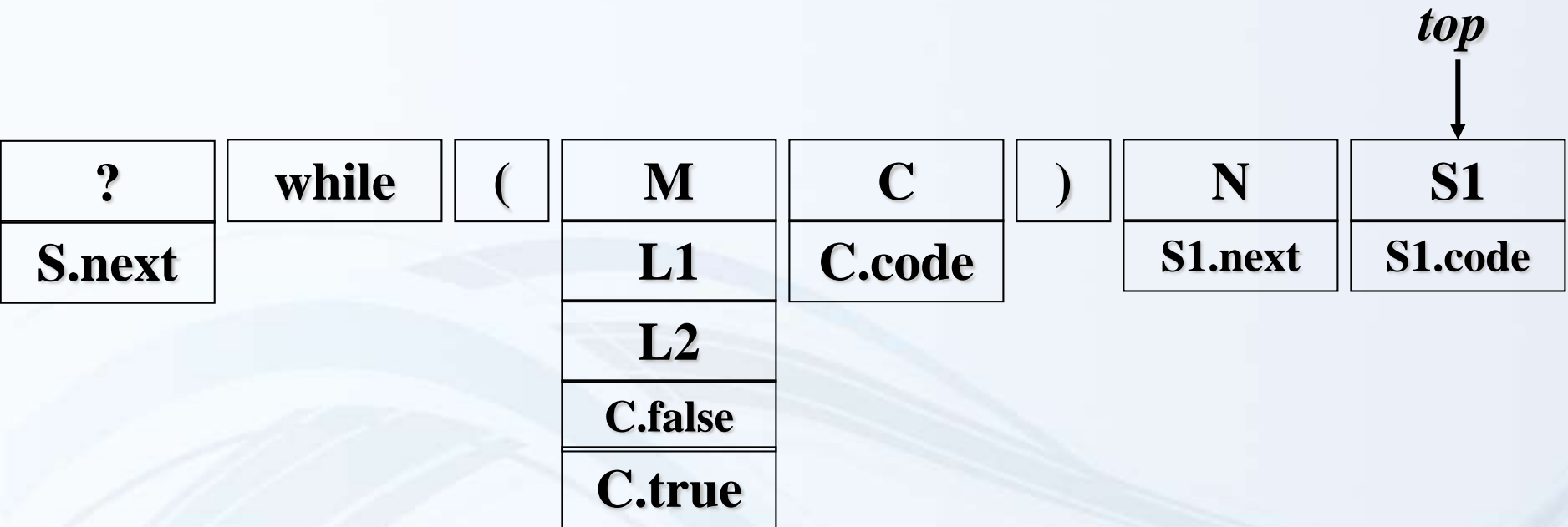
$N \rightarrow \epsilon \quad \{S1.next=stack[top-3].L1 \}$



**$S \rightarrow \text{while} ( M \quad C ) \quad N \quad S1$**

**$M \rightarrow \epsilon \quad \{L1=new(); L2=new(); C.false=stack[top-3].next; C.true=L2\}$**

**$N \rightarrow \epsilon \quad \{S1.next=stack[top-3].L1 \}$**



**$\text{tempCode} = \text{label} \parallel \text{stack}[\text{top}-4].L1 \parallel \text{stack}[\text{top}-3].\text{code} \parallel$**

**$\text{label} \parallel \text{stack}[\text{top}-4].L2 \parallel \text{stack}[\text{top}].\text{code};$**

**$\text{top} = \text{top} - 6;$**

**$\text{stack}[\text{top}].\text{code} = \text{tempCode};$**

# 作业7

1、构建一个语法制导定义，把算术表达式从中缀表示方式翻译成运算符在运算分量之前的前缀表示方式。给出输入 $9-5+2$ 和 $9-5*2$ 时的注释分析树。

构建一个翻译方案，把算术表达式从中缀表示翻译成前缀表示。给出输入 $9-5+2$ 和 $9-5*2$ 时的注释分析树。



# 作业7

2、练习5.1.2，并且对于输入串 $(3+4)*(5+6)n$ ，画出分析树，依赖图，注释分析树。

3、练习5.3.1(1)

4、练习5.4.3

5、练习5.4.4(1), 练习5.4.5(1)

6、练习5.5.2(1), 练习5.5.5(1)