



第四讲 语义分析（基础版）

Semantic Analysis

主要内容

- ◎ 语义分析的作用
- ◎ 语义分析的规约
- ◎ 语义分析的手动实现

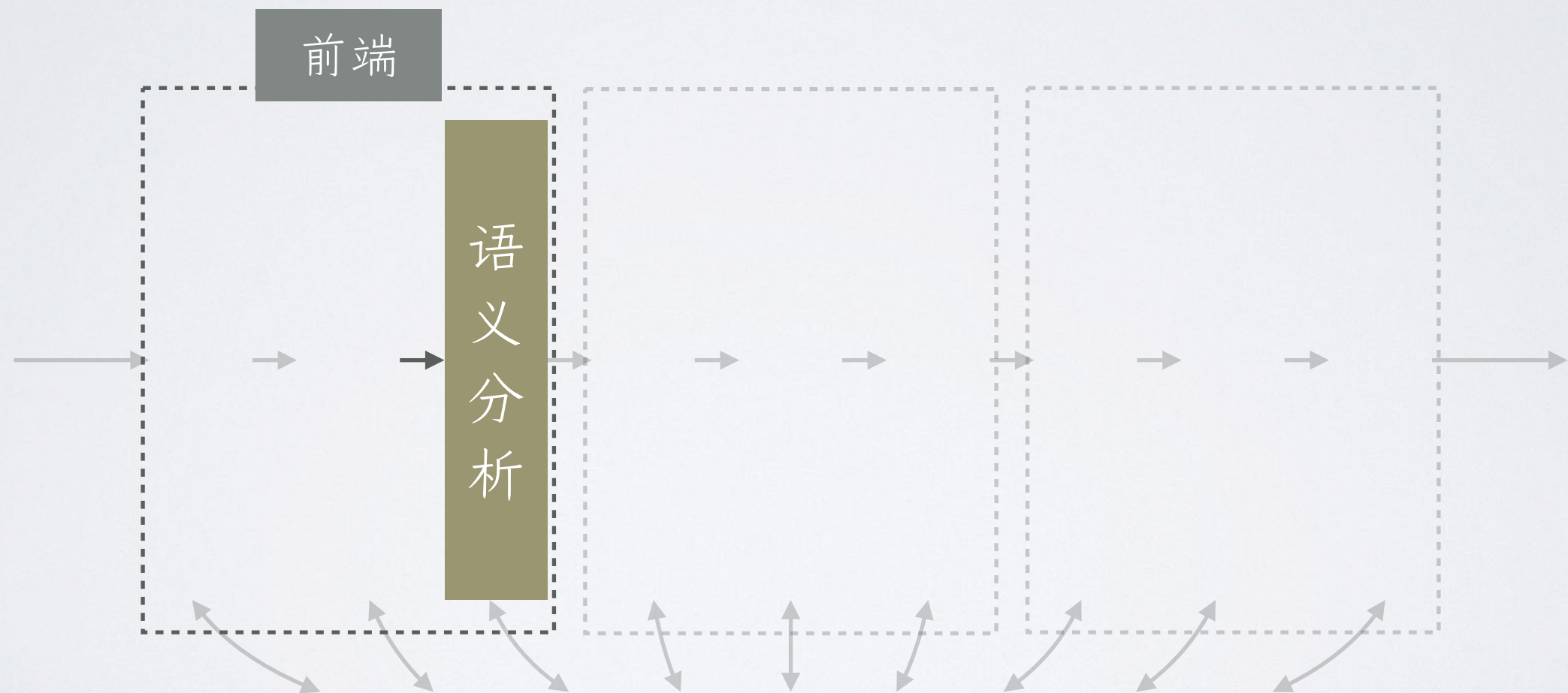
- ◎ 对应龙书章节：第 5 章

主要内容

- ◎ 语义分析的作用
- ◎ 语义分析的规约
- ◎ 语义分析的手动实现

语义分析的作用

- 基于语法分析树，提取程序的核心语义
- 语义分析在不同场景下可以指不同的工作，但通常包括**翻译到中间表示、类型检查、解释执行**等



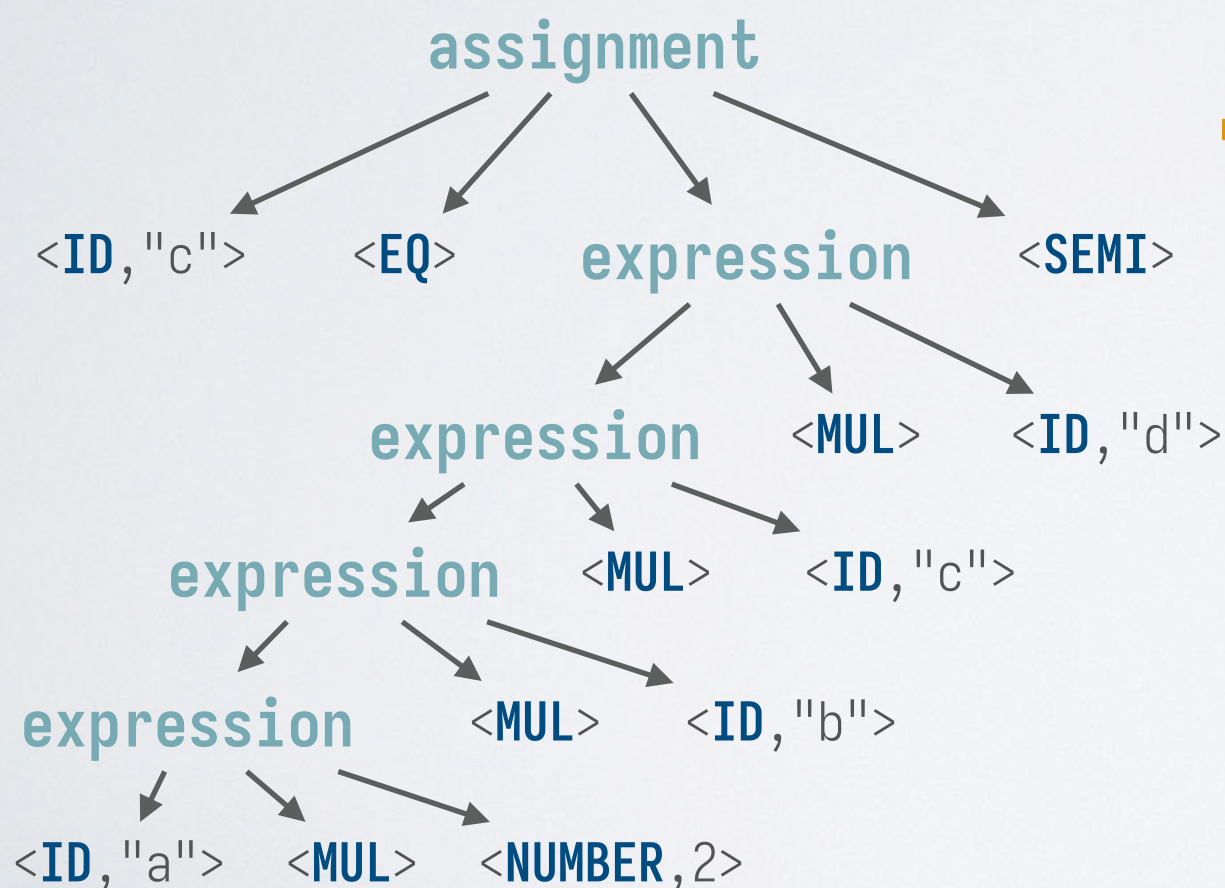
语义分析示例：翻译到中间表示

c = a * 2 * b * c * d;

词法分析

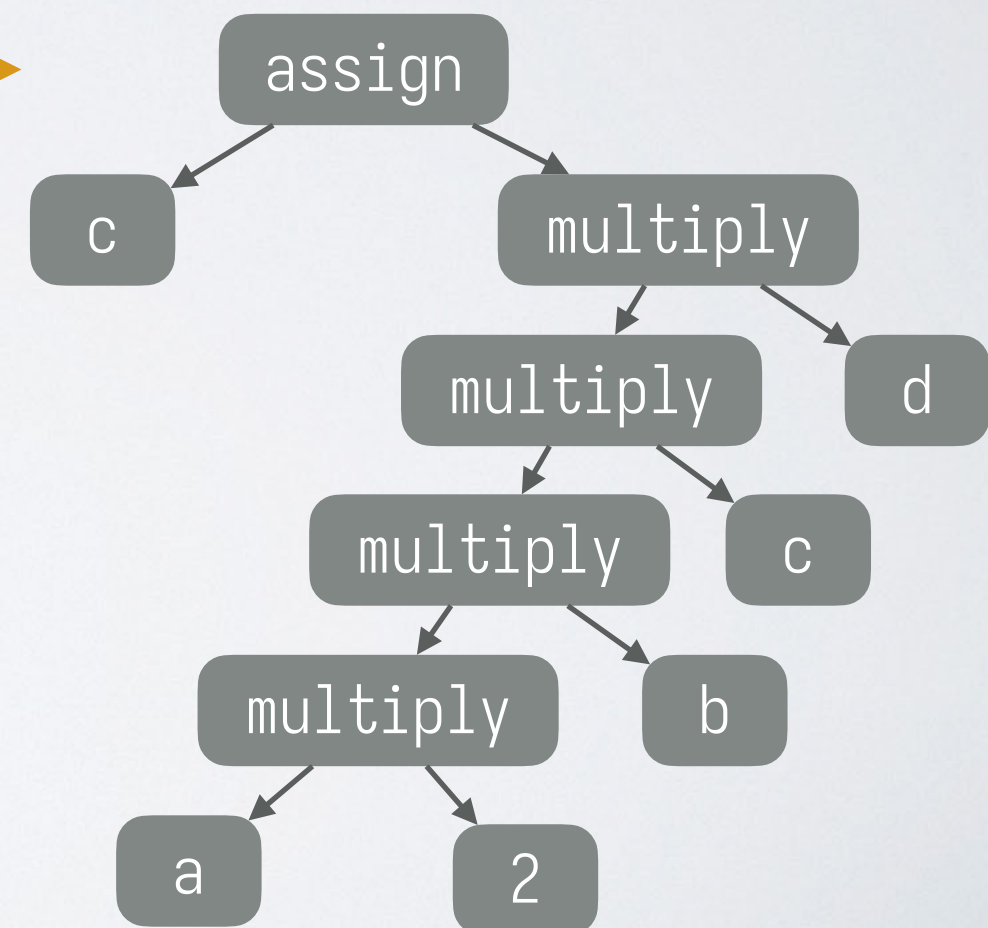
<ID, "c"> <EQ> <ID, "a"> <MUL> <NUMBER, 2> <MUL> <ID, "b"> <MUL> <ID, "c"> <MUL> <ID, "d"> <SEMI>

语法分析



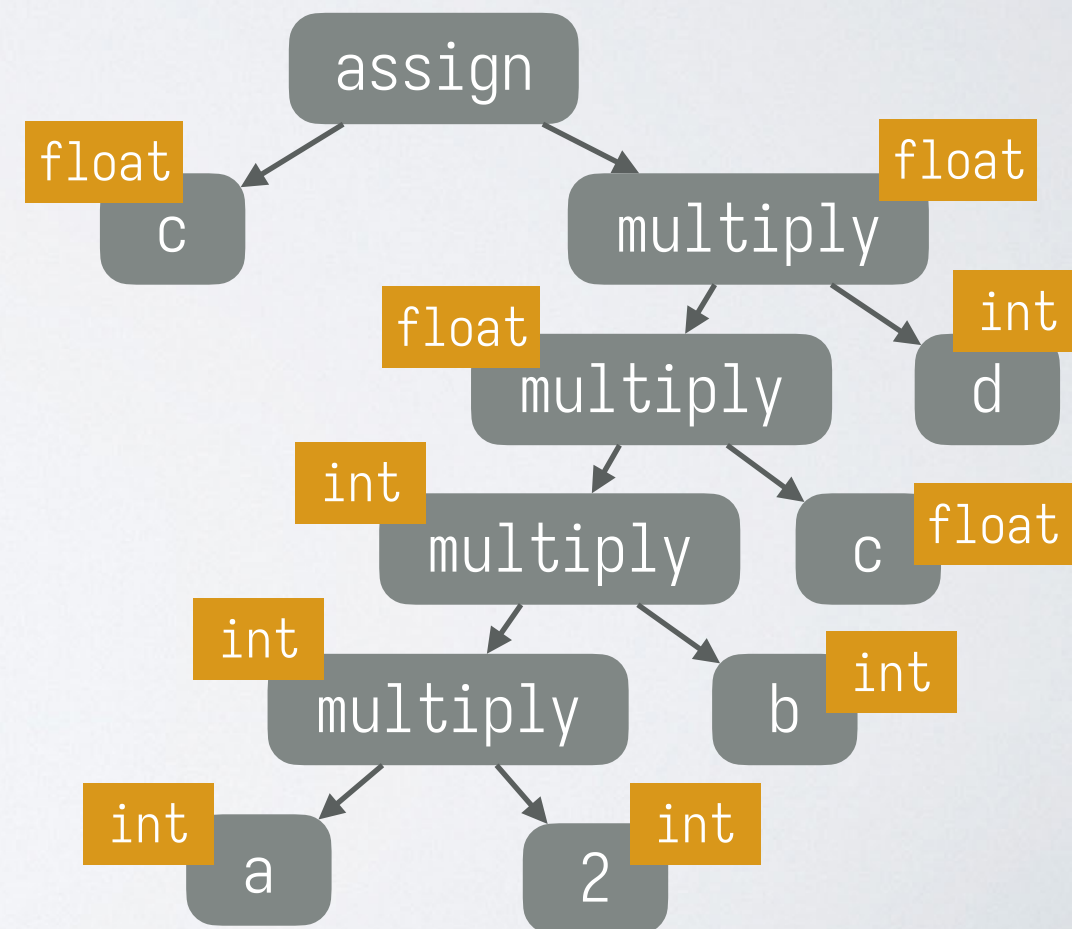
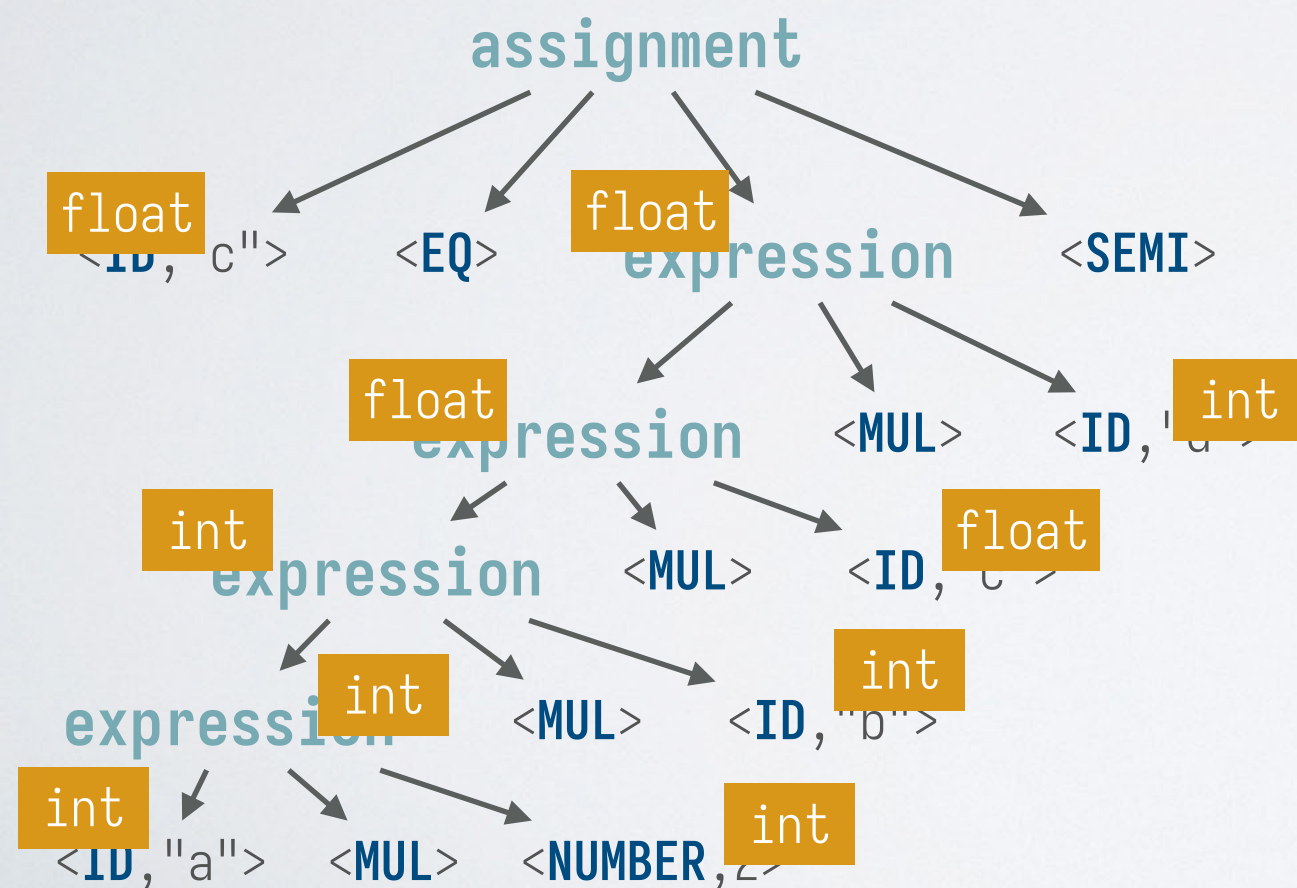
语义分析

抽象语法树



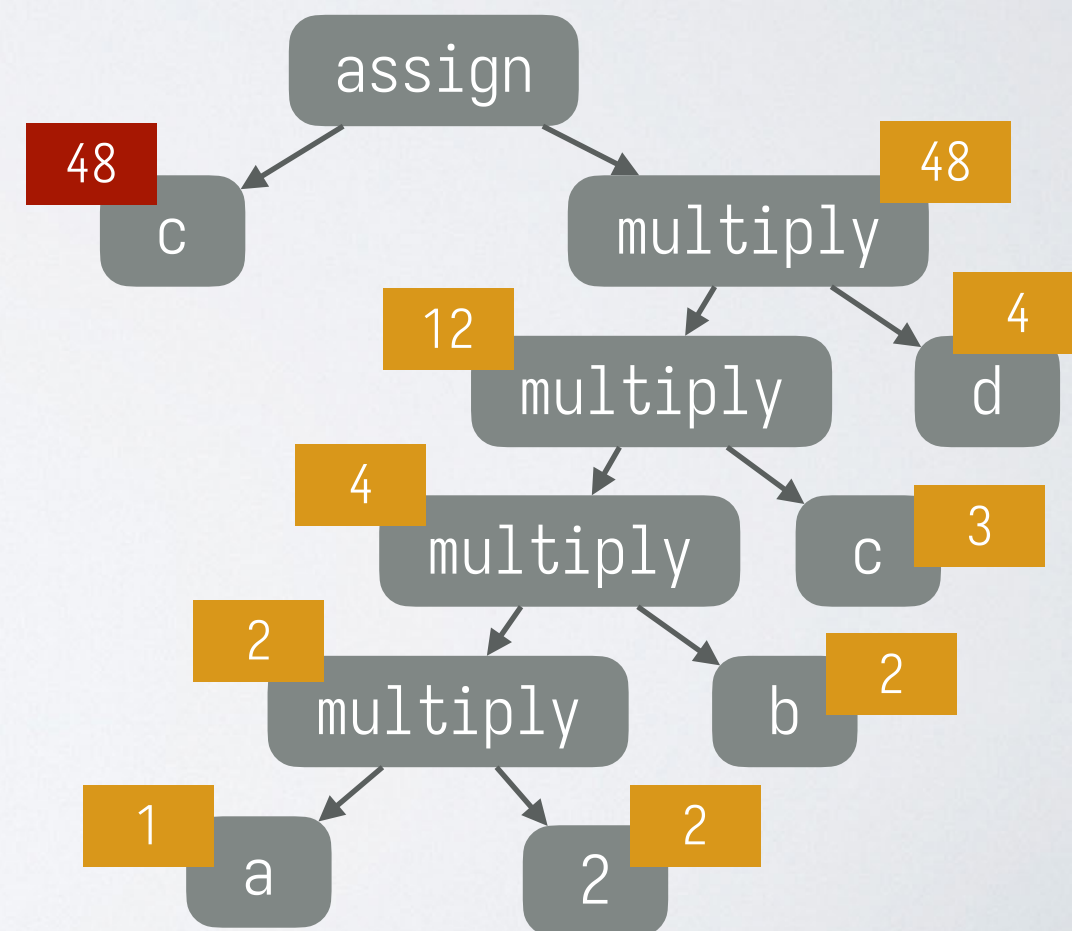
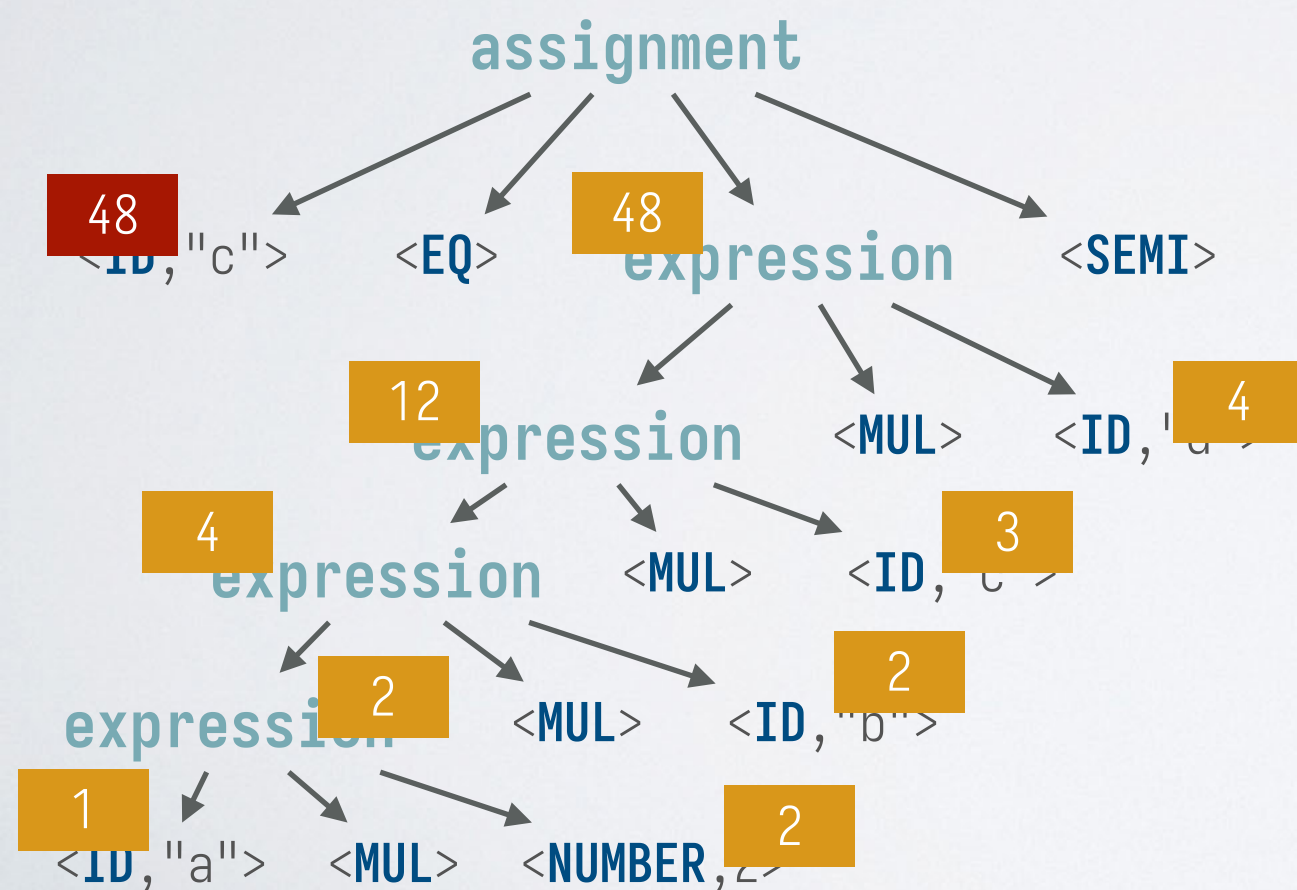
语义分析示例：类型检查

- 分析**名字**和**表达式**的类型，确保在上下文中的使用是合法的
- 避免运行时类型检查的开销
- 可以在语法分析树或抽象语法树上进行



语义分析示例：解释执行

- **解释器**：不生成目标程序，而是直接计算源程序的执行结果
- 比如：对表达式，需要计算其值；对赋值语句，需要完成赋值操作
- 与类型检查类似，可以在语法分析树或抽象语法树上进行

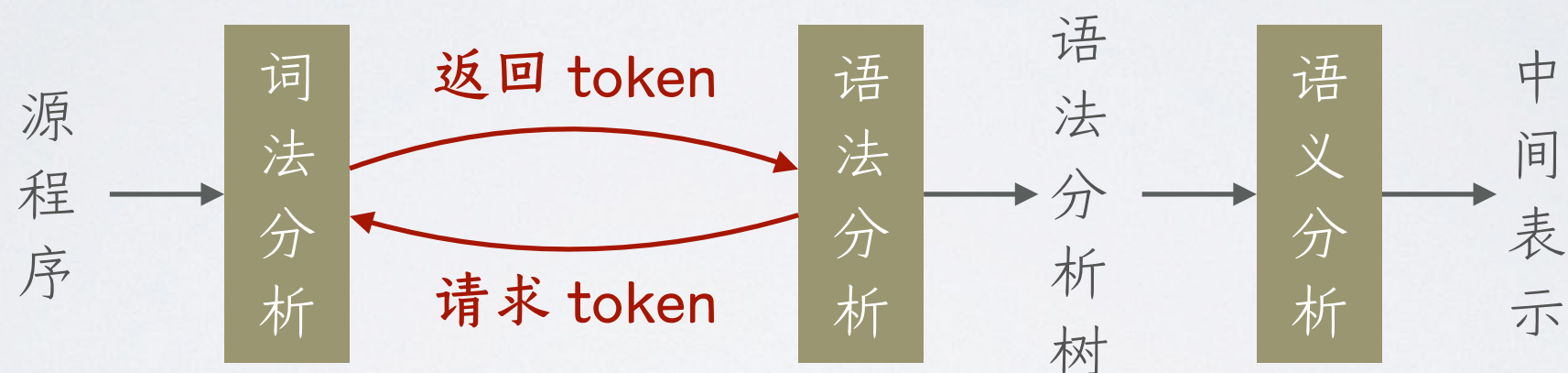


语义分析的工作

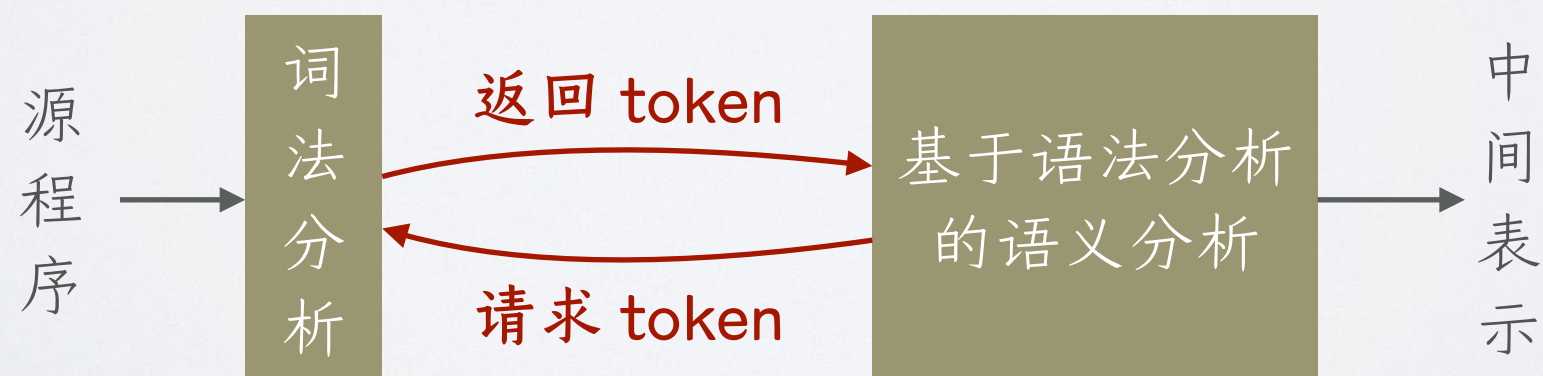
- ◎ 为翻译到目标程序或直接进行解释执行做准备
- ◎ 获取程序在语法之外的各种上下文有关信息
 - ❖ x 中存储了什么种类的值?
 - ❖ x 需要多大的存储空间?
 - ❖ 如果名字 x 是一个函数, 它的参数和返回值是什么?
 - ❖ x 中存储的值的生命周期有多久?
 - ❖ x 应该被谁、在何时进行空间分配和初始化?
 - ❖
- ◎ 上下文有关分析(context-sensitive analysis)
 - ❖ 回顾: 语法分析中使用上下文无关文法

语义分析的工作

- ◎ 与优化器中的分析不同，语义分析主要关注**翻译到中间表示**
 - ❖ 也被称为**语义推敲**(semantic elaboration)
- ◎ 可以独立于语法分析进行实现



- ◎ 也可以与语法分析同步进行



主要内容

- ◎ 语义分析的作用
- ◎ **语义分析的规约**
- ◎ 语义分析的手动实现

回顾：词法分析的规约

- 使用一系列的正则表达式, 标明它们每个对应的 token 类别

token 类别	正则表达式
IF	<code>i f</code>
WHILE	<code>w h i l e</code>
ID	<code>[A-Za-z_]([A-Za-z_0-9])*</code>
NUMBER	<code>[+-]?([0-9])⁺</code>
LPAREN	<code>(</code>
NEQ	<code>!=</code>

- 每个正则表达式 r 表示了符号表 Σ 上的一个语言 $L(r)$
- 语法分析的规约需要描述由 token 组成的语言

回顾：语法分析的规约

- 使用上下文无关文法，其**终结符号**为词法分析所得的 token，其**非终结符号**为语言的语法变量

$$\begin{aligned} \text{Expr} &::= \text{Expr} + \text{Term} \\ &\quad | \text{Expr} - \text{Term} \\ &\quad | \text{Term} \\ \text{Term} &::= \text{Term} * \text{Factor} \\ &\quad | \text{Term} / \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &::= (\text{Expr}) \\ &\quad | \text{ID} \mid \text{INT} \end{aligned}$$
$$\begin{aligned} \text{Stmt} &::= \{ \text{Block} \} \\ &\quad | \text{ID} = \text{Expr} ; \\ &\quad | \text{if} (\text{Expr}) \text{Stmt} \text{ else } \text{Stmt} \\ &\quad | \text{while} (\text{BDisj}) \text{Stmt} \\ &\quad | \text{return } \text{Expr} ; \\ \text{Block} &::= \epsilon \\ &\quad | \text{Stmt Block} \end{aligned}$$

- 每个文法 $G[S]$ 表示了一个终结符号串的语言 $L(G)$ ，其中的每个符号串 w 可以由**初始符号** S 通过**产生规则**推导而出 ($S \Rightarrow^* w$)
- 语义分析的规约需要**针对 $L(G)$ 来分析和提取语义信息**

使用上下文有关文法作为规约?

◎ 问题: 是否可以使用上下文有关文法描述语义分析?

❖ 允许产生规则 $\alpha \rightarrow \beta$ 的左侧长度大于 1, 但要求 $|\alpha| \leq |\beta|$

◎ 考虑语言 $\{a^n b^n c^n \mid n \geq 1\}$

❖ 可以把 a 想象成函数定义

❖ 把 b 和 c 想象成两处函数调用

❖ 右侧的上下文有关文法描述了该语言

❖ 但是挺麻烦的, 尤其考虑各种复杂语言特性

[1]	$S \rightarrow a B C$
[2]	$S \rightarrow a S B C$
[3]	$C B \rightarrow B C$
[4]	$a B \rightarrow a b$
[5]	$b B \rightarrow b b$
[6]	$b C \rightarrow b c$
[7]	$c C \rightarrow c c$

◎ 观察: 检查一个串是否形如 $a^n b^n c^n$ 是容易的

❖ 编译器只要在语法分析树上能进行「检查」即可

属性文法 (Attribute Grammar)

- ◎ 属性文法 = 上下文无关文法 + 属性计算规则
 - ❖ 也被称为**语法制导定义** (Syntax-Directed Definition, SDD)
- ◎ 属性与**非终结符号**相关联
 - ❖ 用 $X.a$ 表示语法分析树中某个 X 结点的 a 属性值
- ◎ 属性计算规则与**产生规则**相关联
 - ❖ 每条产生规则可以有多条计算规则, 用来处理不同的属性
 - ❖ 计算规则可以使用产生规则中涉及的非终结符号的属性
- ◎ 从**语法分析树**的视角来看, 属性文法规定了一个内部结点及其孩子结点的属性之间的关系

属性文法示例：四则运算表达式

- 文法 $G = (V_T, V_N, Expr, P)$ 表示四则运算表达式, 其中
 $V_T = \{+, -, *, /, (,), INT\}$, $V_N = \{Expr, Term, Factor\}$

用下标区分相同符号的不同结点

属性 val 记录表达式的求值结果

产生规则

属性计算规则

```
Expr ::= Expr + Term
      | Expr - Term
      | Term
Term  ::= Term * Factor
      | Term / Factor
      | Factor
Factor ::= ( Expr )
      | INT
```

$Expr \rightarrow Expr_1 + Term$

$Expr.val = Expr_1.val + Term.val$

$Expr \rightarrow Expr_1 - Term$

$Expr.val = Expr_1.val - Term.val$

$Expr \rightarrow Term$

$Expr.val = Term.val$

$Term \rightarrow Term_1 * Factor$

$Term.val = Term_1.val \times Factor.val$

$Term \rightarrow Term_1 / Factor$

$Term.val = Term_1.val \div Factor.val$

$Term \rightarrow Factor$

$Term.val = Factor.val$

$Factor \rightarrow (Expr)$

$Factor.val = Expr.val$

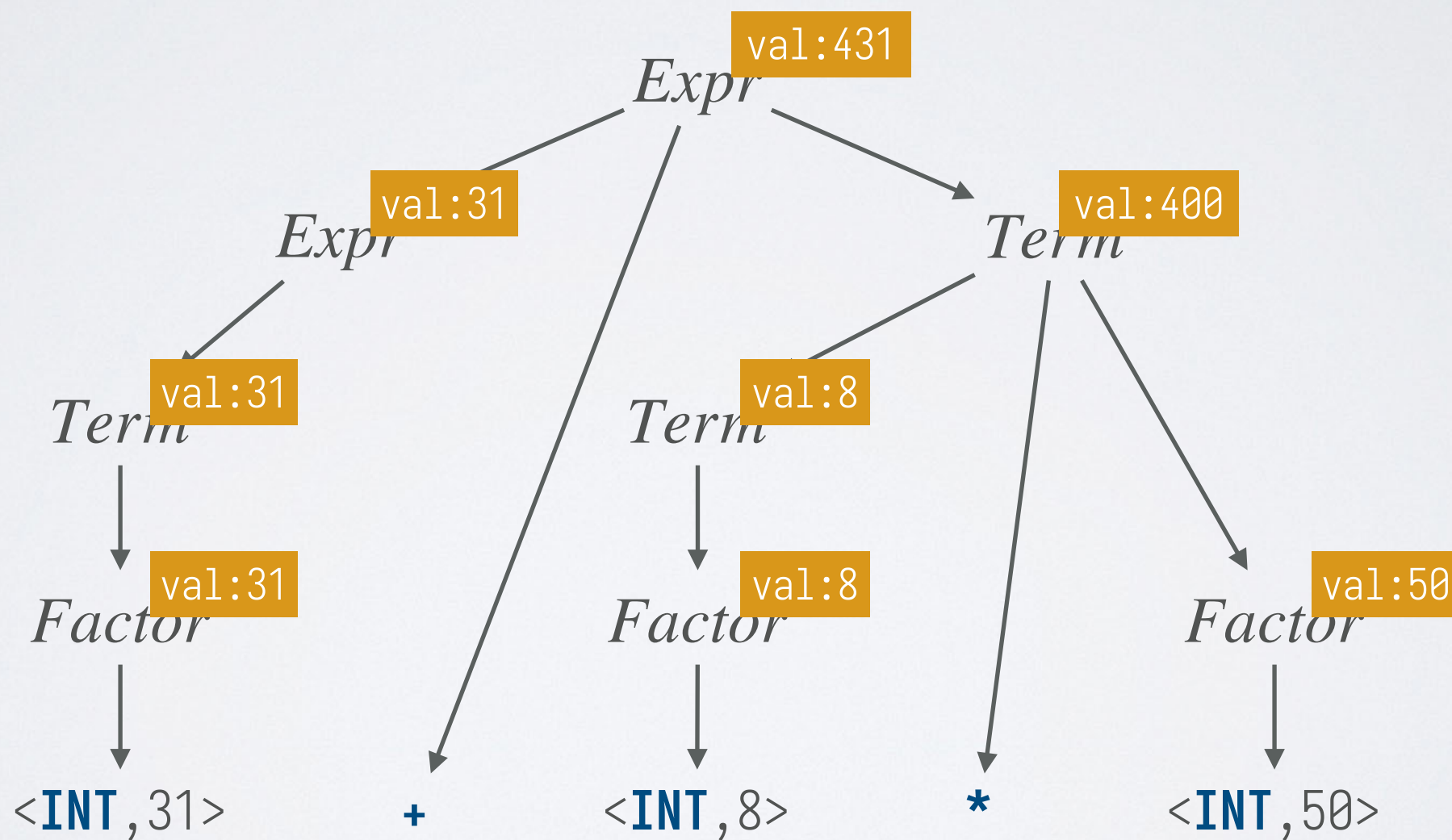
$Factor \rightarrow INT$

$Factor.val = INT.intval$

INT 词法单元附带属性
intval 表示其值

属性文法示例：四则运算表达式

31 + 8 * 50



属性文法示例：有符号二进制数

- 文法 $G = (V_T, V_N, Number, P)$ 表示有符号二进制数，其中
 $V_T = \{+, -, 0, 1\}$, $V_N = \{Number, Sign, List, Bit\}$

$Number ::= Sign List$

$Sign ::= +$

$| -$

$List ::= Bit$

$| List Bit$

$Bit ::= 0$

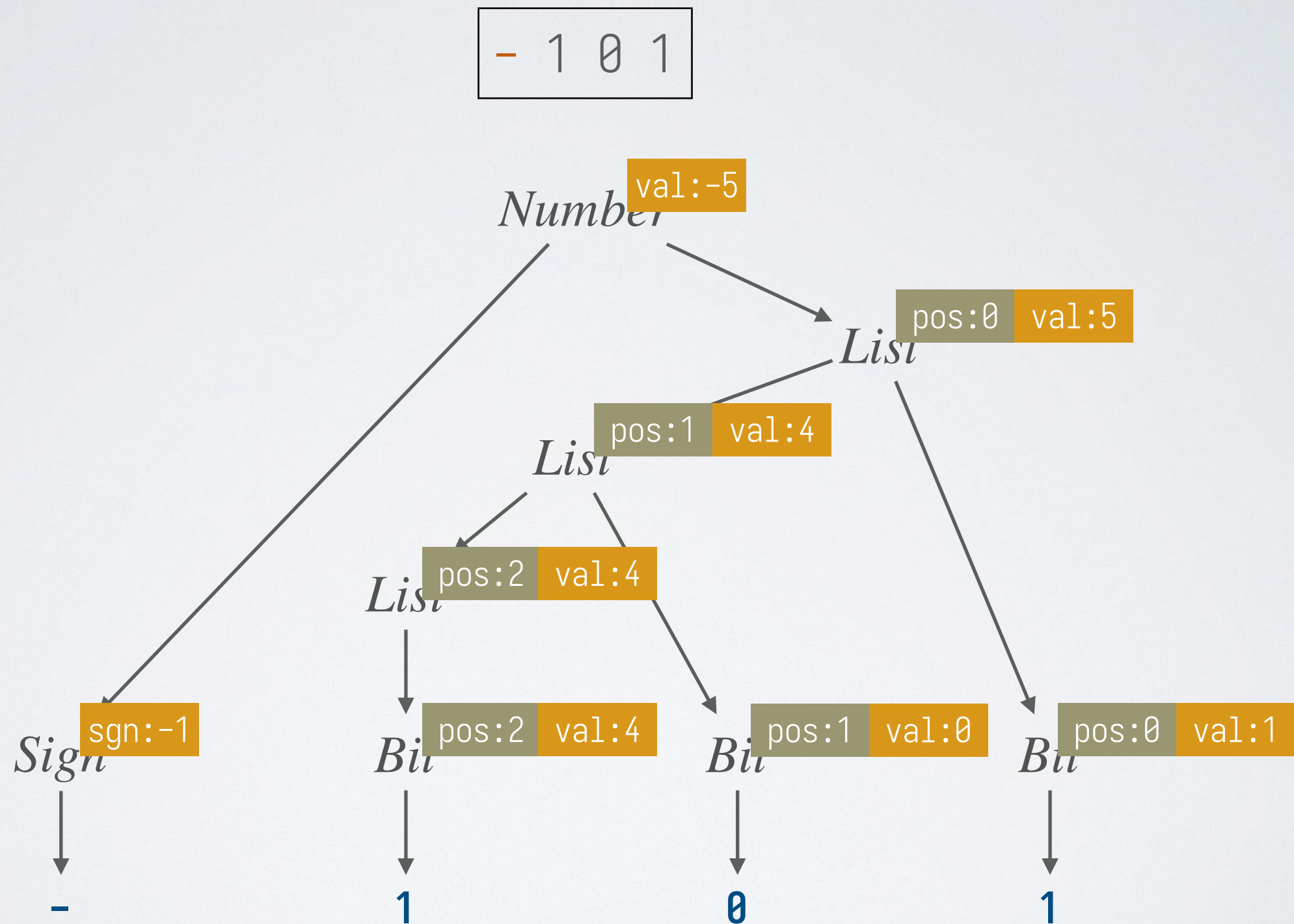
$| 1$

属性文法示例：有符号二进制数

- 文法 $G = (V_T, V_N, Number, P)$ 表示有符号二进制数，其中 $V_T = \{+, -, 0, 1\}$, $V_N = \{Number, Sign, List, Bit\}$

产生规则	属性 val 记录 对应的整数值	属性计算规则	属性 sgn 记录 正负符号值
$Number \rightarrow Sign List$		$Number.val = Sign.sgn \times List.val$	
	$List.pos = 0$		属性 pos 记录二进制位所 在位置(最低位的位置为 0)
$Sign \rightarrow +$	$Sign.sgn = 1$		
$Sign \rightarrow -$	$Sign.sgn = -1$		
$List \rightarrow Bit$	$List.val = Bit.val$ $Bit.pos = List.pos$		
$List \rightarrow List_1 Bit$	$List.val = List_1.val + Bit.val$ $List_1.pos = List.pos + 1$ $Bit.pos = List.pos$		
$Bit \rightarrow 0$	$Bit.val = 0$		
$Bit \rightarrow 1$	$Bit.val = 2^{Bit.pos}$		

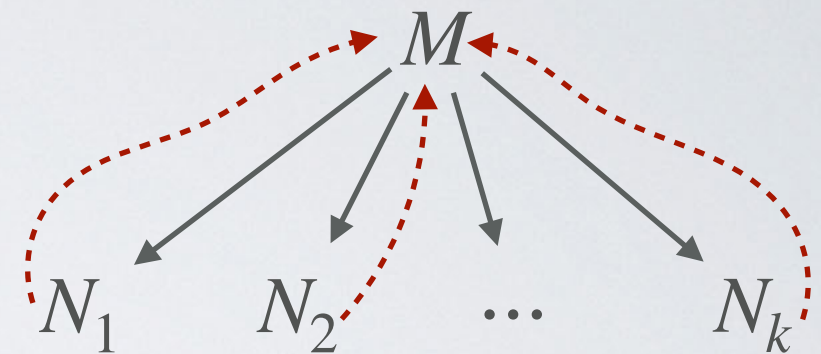
属性文法示例：有符号二进制数



综合属性和继承属性

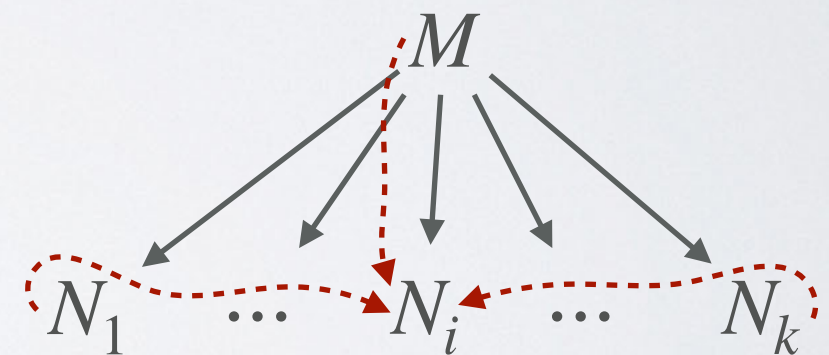
● 综合属性 (synthesized attribute)

- ❖ 信息流向自底向上
- ❖ 结点 M 的一个综合属性可以通过 M 自身和它的孩子结点 $N_1, N_2, \dots, N_k$ 的属性来定义



● 继承属性 (inherited attribute)

- ❖ 信息流向自顶向下
- ❖ 结点 N_i 的一个继承属性可以通过 N_i 自身、它的 parent 结点 M 以及它的 sibling 结点 $N_1, \dots, N_{i-1}, N_{i+1}, \dots, N_k$ 的属性来定义



综合属性示例

Expr、*Term*、*Factor* 的属性
val 都是综合属性

产生规则	属性计算规则
$Expr \rightarrow Expr_1 + Term$	$Expr.val = Expr_1.val + Term.val$
$Expr \rightarrow Expr_1 - Term$	$Expr.val = Expr_1.val - Term.val$
$Expr \rightarrow Term$	$Expr.val = Term.val$
$Term \rightarrow Term_1 * Factor$	$Term.val = Term_1.val \times Factor.val$
$Term \rightarrow Term_1 / Factor$	$Term.val = Term_1.val \div Factor.val$
$Term \rightarrow Factor$	$Term.val = Factor.val$
$Factor \rightarrow (Expr)$	$Factor.val = Expr.val$
$Factor \rightarrow INT$	$Factor.val = INT.intval$

- 通常表现为产生规则**左侧**的属性由**右侧**符号的属性计算得出

继承属性示例

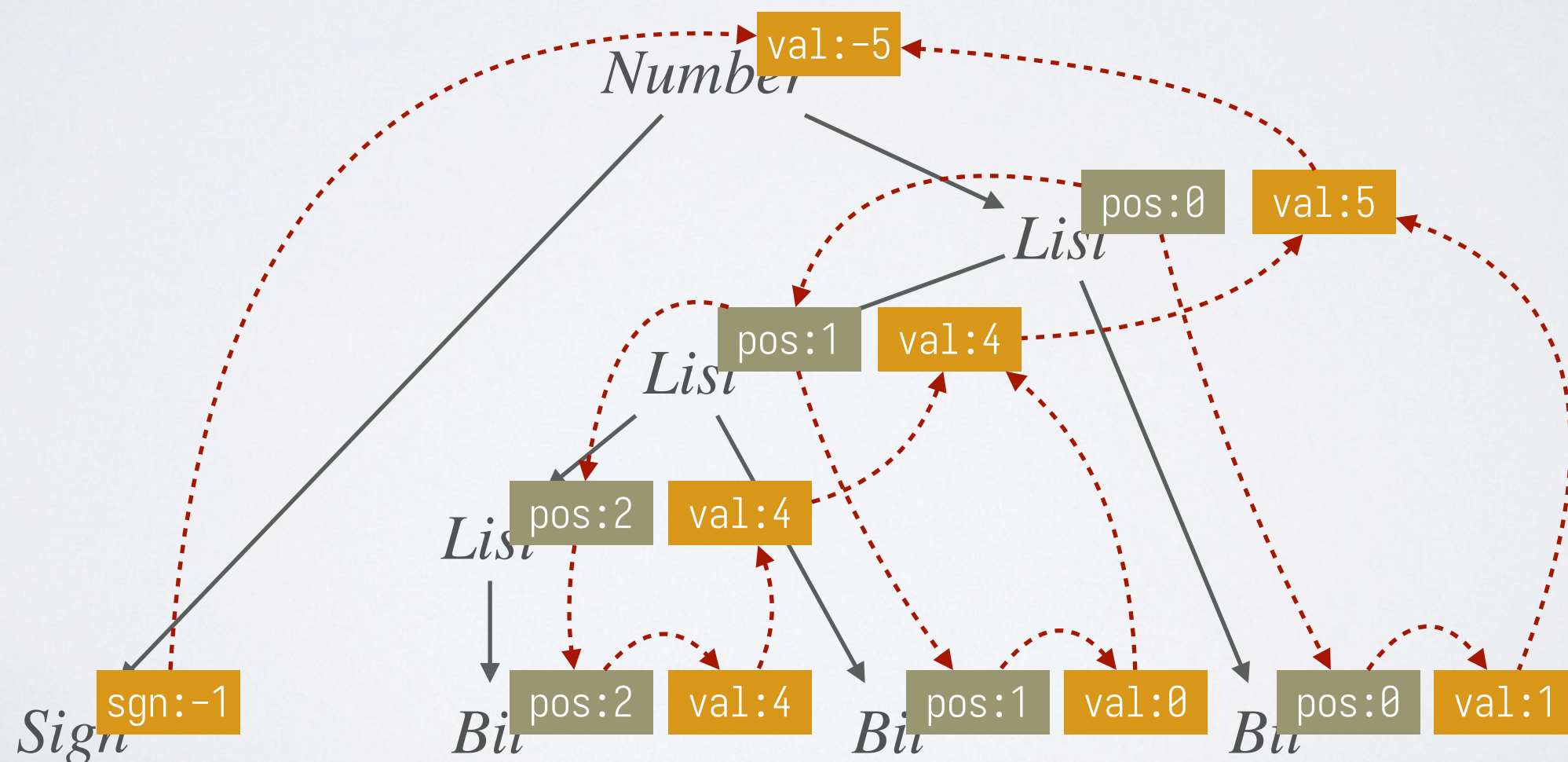
产生规则	属性计算规则
$Number \rightarrow Sign\ List$	$Number.val = Sign.sgn \times List.val$ $List.pos = 0$
$Sign \rightarrow +$	$Sign.sgn = 1$
$Sign \rightarrow -$	$Sign.sgn = -1$
$List \rightarrow Bit$	$List.val = Bit.val$ $Bit.pos = List.pos$
$List \rightarrow List_1\ Bit$	$List.val = List_1.val + Bit.val$ $List_1.pos = List.pos + 1$ $Bit.pos = List.pos$
$Bit \rightarrow 0$	$Bit.val = 0$
$Bit \rightarrow 1$	$Bit.val = 2^{Bit.pos}$

List、*Bit* 的属性 pos 记录二进制位所在位置，它们都是继承属性

◎ 通常表现为产生规则**右侧**符号的属性由**左侧**的属性计算得出

属性计算顺序

- 问题：属性之间可能依赖，如何确定一个计算顺序？
- 以语法分析树中的所有属性为结点，用有向边 $c \rightarrow d$ 表示计算属性 d 的值需要用到属性 c 的值，建立属性依赖图
- 若依赖图中无环，则可以按照**拓扑序**进行计算



属性计算顺序：有环的情况

- 存在环的属性依赖关系在一些场景中是有意义的

$$\begin{array}{l} Loop ::= \text{repeat INT do } Stmt \\ Stmt ::= \epsilon \\ \quad \quad | \text{ cnt} += \text{INT} ; Stmt \end{array}$$

- 考虑上面的文法，它表示了一个操作变量 `cnt` 的简单循环
 - ❖ 例如：程序 `repeat 10 do cnt += 50`；执行结束后 `cnt` 的值为 500
- 问题：如何设计属性文法对这个循环语言进行解释求值？
 - ❖ 用属性 `cnt_init` 记录语句执行前 `cnt` 的值
 - ❖ 用属性 `cnt_final` 记录语句执行后 `cnt` 的值
 - ❖ 循环需要反复进行计算，必然导致属性间的依赖

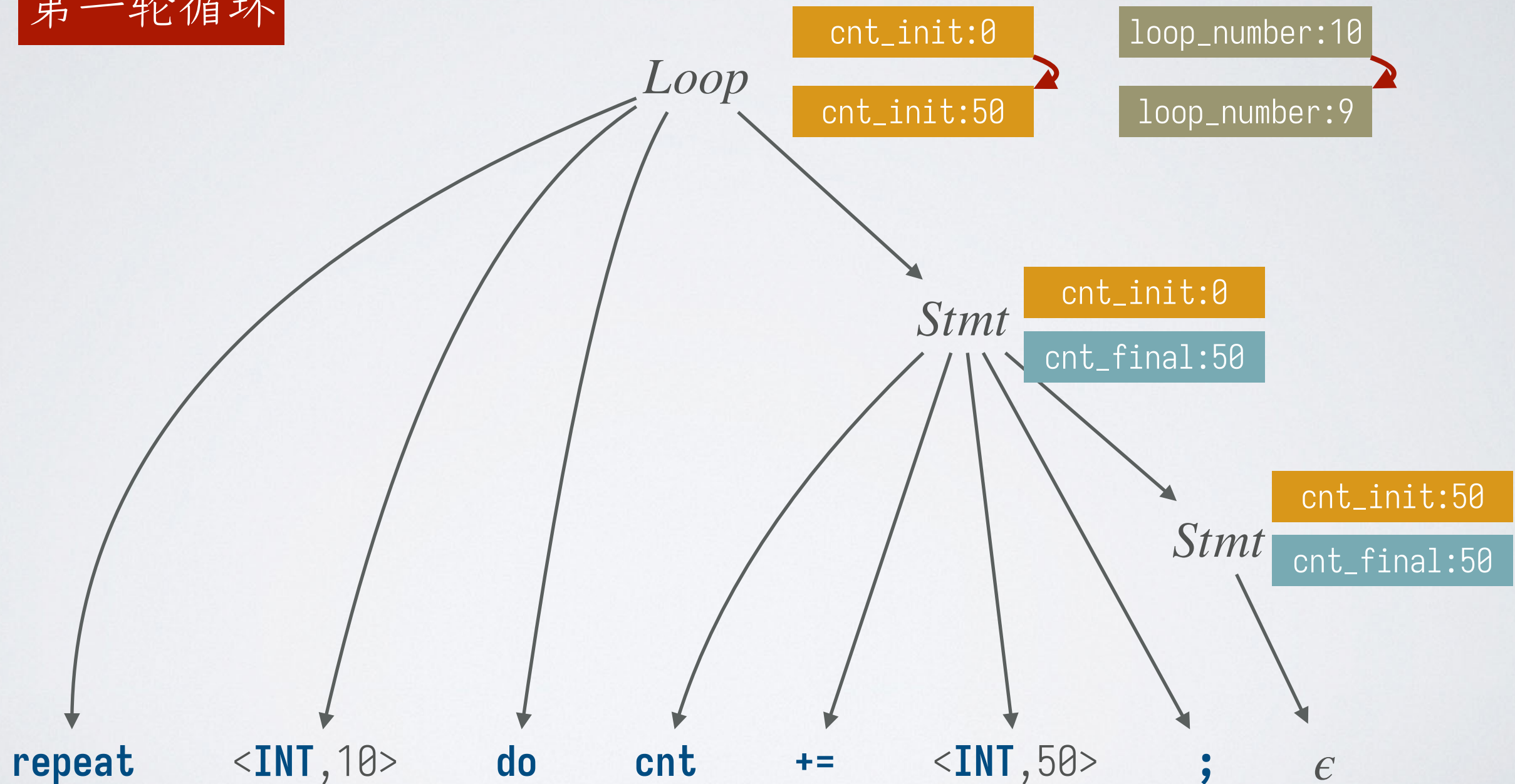
属性计算顺序：有环的情况

- 如果一个属性依赖的某个属性被更新了,那么它也需要更新

产生规则	属性计算规则
$Loop \rightarrow \text{repeat INT do } Stmt$	$Loop.cnt_init = 0$ $Loop.loop_number = INT.intval$ $\text{if } Loop.loop_number > 0 \text{ then}$ 在一轮循环中,需要先执行一遍循环体($Stmt$),通过继承属性传递 cnt 的值 $Stmt.cnt_init = Loop.cnt_init$ $Loop.cnt_init = Stmt.cnt_final$ $Loop.loop_number = Loop.loop_number - 1$ 循环体执行结束后,更新 cnt 的值并准备进入下一轮循环 else $Loop.cnt_final = Loop.cnt_init$ 更新循环计数器,完成一轮循环
$Stmt \rightarrow \epsilon$	$Stmt.cnt_final = Stmt.cnt_init$
$Stmt \rightarrow cnt += INT ; Stmt_1$	$Stmt_1.cnt_init = Stmt.cnt_init + INT.intval$ $Stmt.cnt_final = Stmt_1.cnt_final$ 循环体中的语句依次对 cnt 的值进行更新

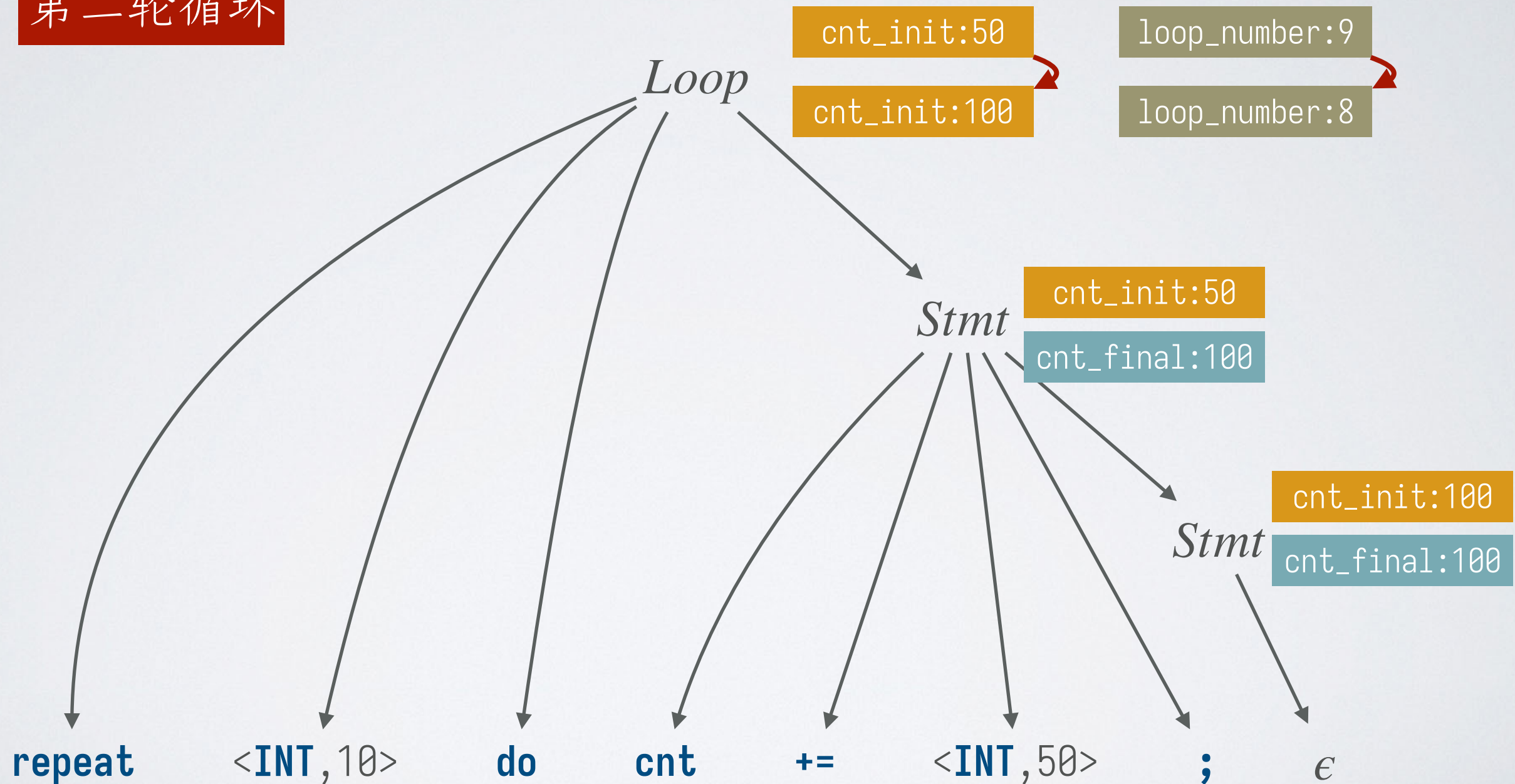
属性计算顺序：有环的情况

第一轮循环



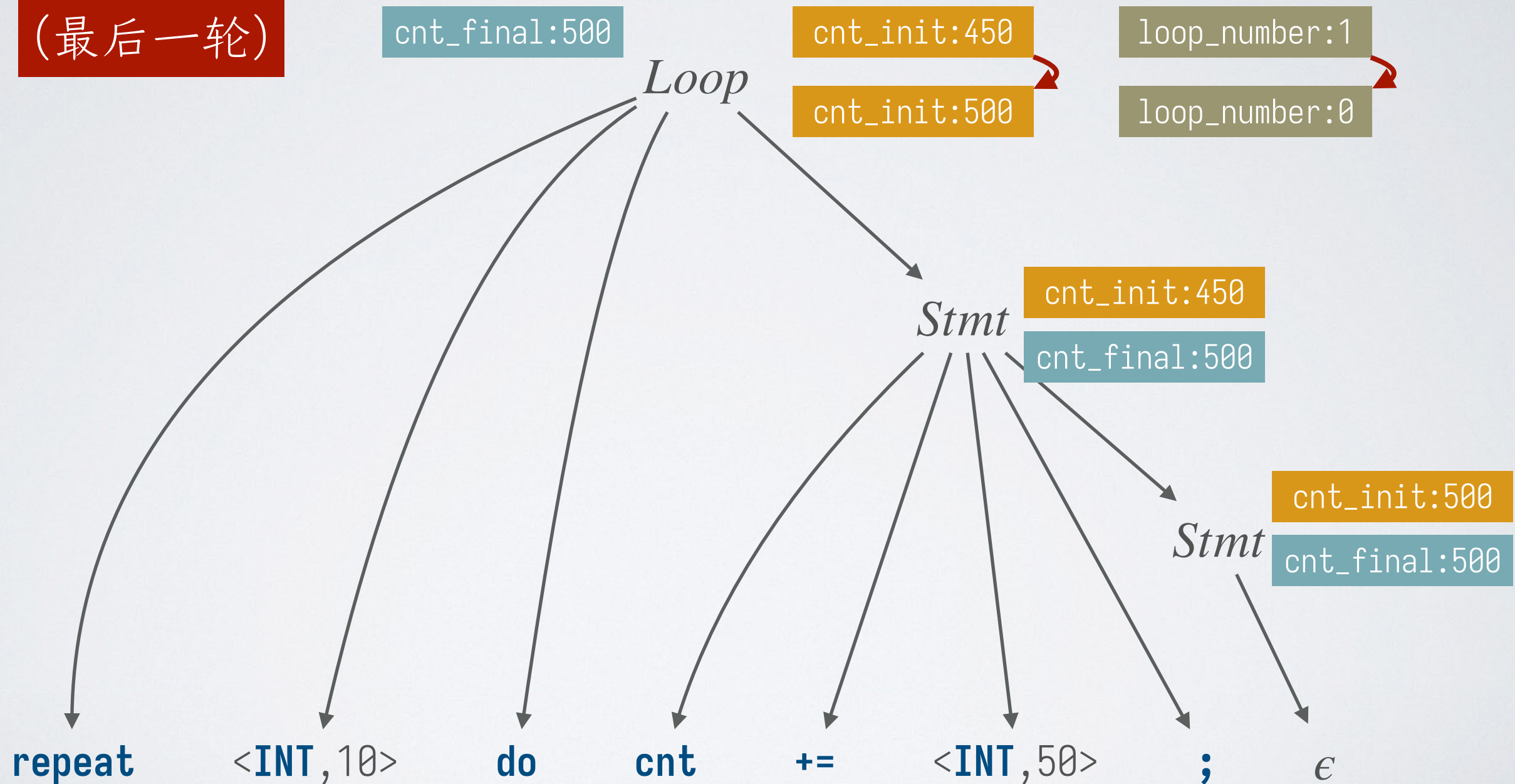
属性计算顺序：有环的情况

第二轮循环



属性计算顺序：有环的情况

第十轮循环
(最后一轮)



属性计算顺序：无环的情况

- ◎ 给定一个属性文法, 很难判定是否存在一棵语法分析树对应的属性依赖图包含环
- ◎ 特定类型的属性文法一定不包含环, 且有固定的计算顺序
 - ❖ S 属性的文法(或 S 属性的 SDD)
 - ❖ L 属性的文法(或 L 属性的 SDD)
- ◎ 这些类型的属性文法具备的优势:
 - ❖ 可以不用构造完整的语法分析树
 - ❖ 进而可以和语法分析同步进行

S 属性的文法

- ◎ 每个属性都是综合属性

- ❖ 在语法分析树上, 属性的信息流向是自底向上
- ❖ 在属性依赖图上, 通过 M 的孩子结点的属性值来计算 M 的属性值

- ◎ 例如: 前面的四则运算表达式文法

- ◎ 容易结合自顶向下的语法分析进行实现

L 属性的文法

- ◎ 每个属性可以是
 - ❖ 综合属性
 - ❖ 继承属性, 但是产生规则 $A \rightarrow X_1 X_2 \dots X_k$, 某个 X_i 的继承属性只依赖于 X_i 的左边 $(X_1, X_2, \dots, X_{i-1})$ 的属性, 以及 A 的继承属性
- ◎ 在语法分析树上:
 - ❖ 综合属性的信息流向自底向上
 - ❖ 继承属性的信息流向自顶向下、自左向右
- ◎ 例如: 前面的有符号二进制数文法
- ◎ 容易结合自顶向下的语法分析进行实现

L 属性的文法示例

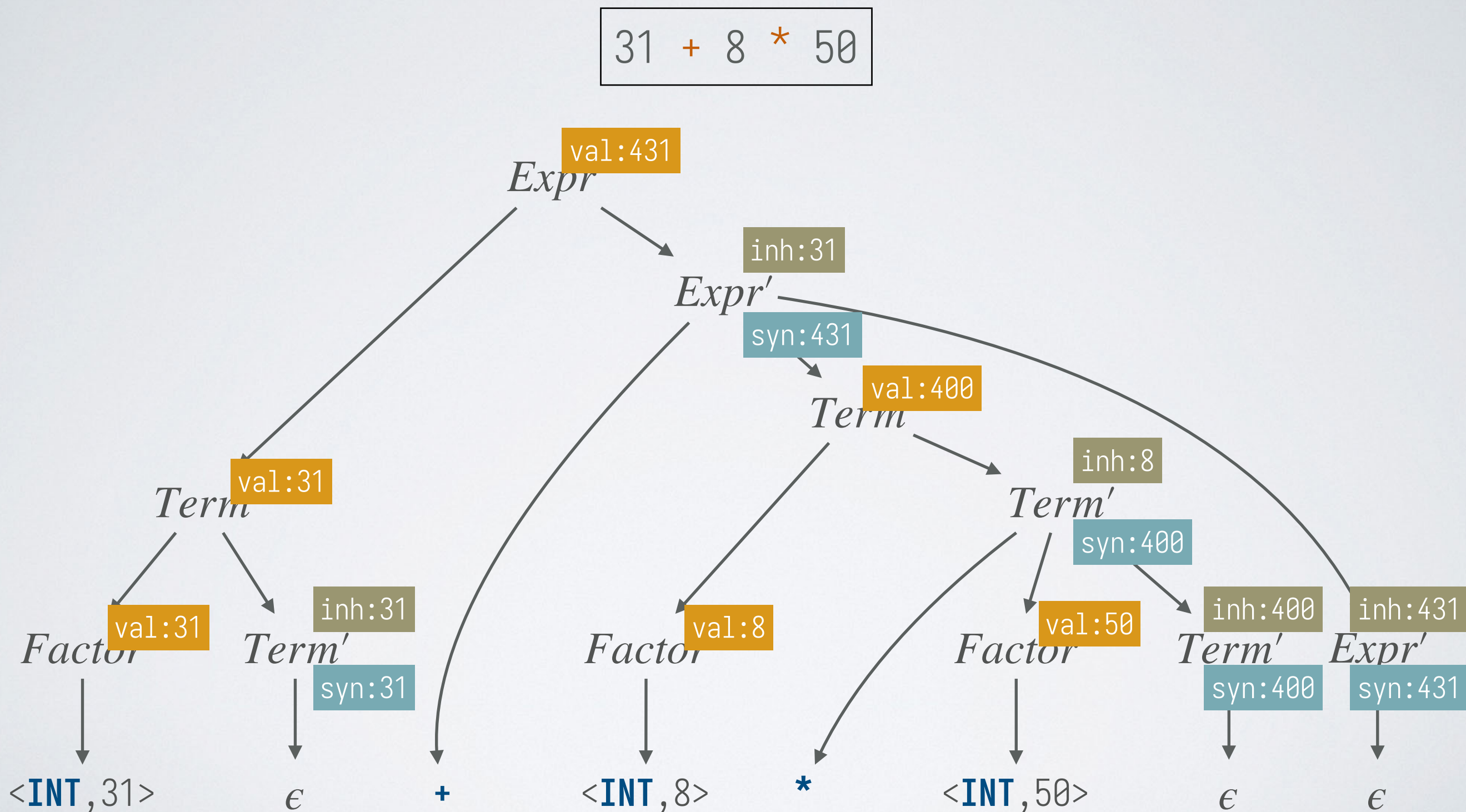
```

Expr ::= Term Expr'
Expr' ::= + Term Expr'
        | - Term Expr'
        | ε
Term  ::= Factor Term'
Term' ::= * Factor Term'
        | / Factor Term'
        | ε
Factor ::= ( Expr )
        | INT
    
```

产生规则	继承属性 inh 记录 左侧已经计算的值	属性计算规则
$Expr \rightarrow Term\ Expr'$		$Expr'.inh = Term.val$ $Expr.val = Expr'.syn$
$Expr' \rightarrow + Term\ Expr'_1$		$Expr'_1.inh = Expr'.inh + Term.val$ $Expr'.syn = Expr'_1.syn$
$Expr' \rightarrow \epsilon$		$Expr'.syn = Expr'.inh$
$Term \rightarrow Factor\ Term'$		$Term'.inh = Factor.val$ $Term.val = Term'.syn$
$Term' \rightarrow * Factor\ Term'_1$		$Term'_1.inh = Term'.inh \times Factor.val$ $Term'.syn = Term'_1.syn$
$Term' \rightarrow \epsilon$		$Term'.syn = Term'.inh$
$Factor \rightarrow (Expr)$		$Factor.val = Expr.val$
$Factor \rightarrow INT$		$Factor.val = INT.intval$

综合属性 syn 记录在 inh 基础上计算的值

L 属性的文法示例



属性文法应用：翻译到中间表示

◎ 抽象语法树 (Abstract Syntax Tree, AST)

◎ 一种四则运算表达式 AST 的定义：

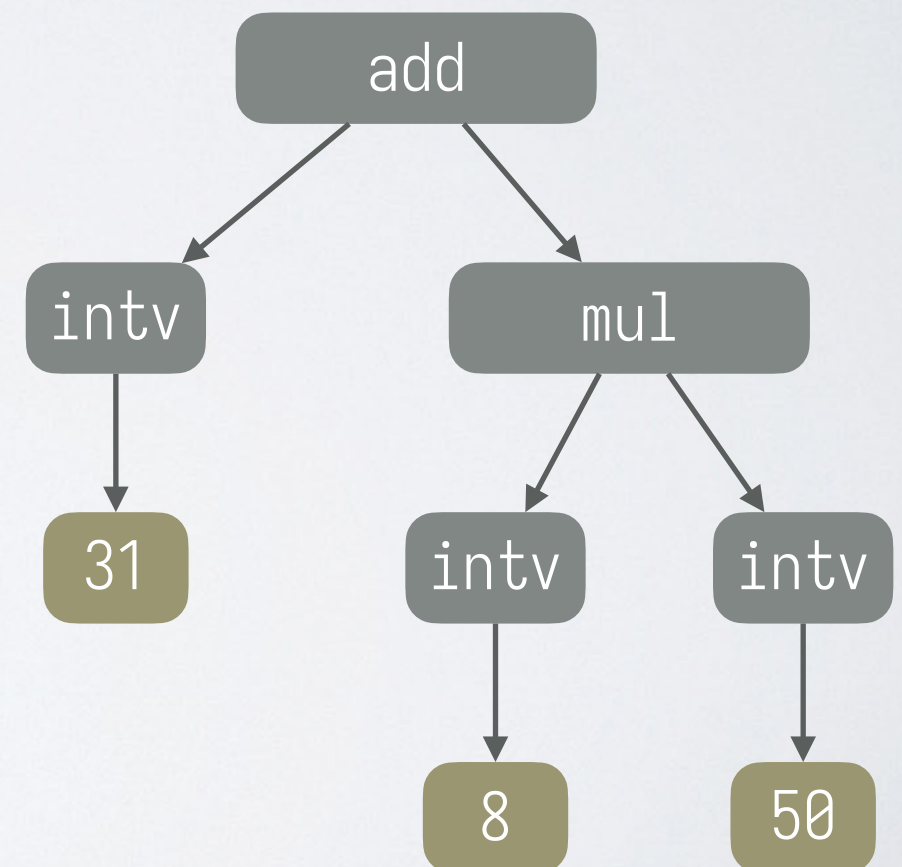
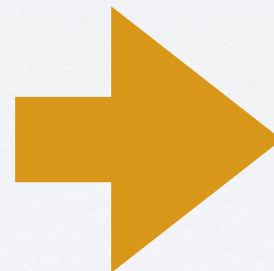
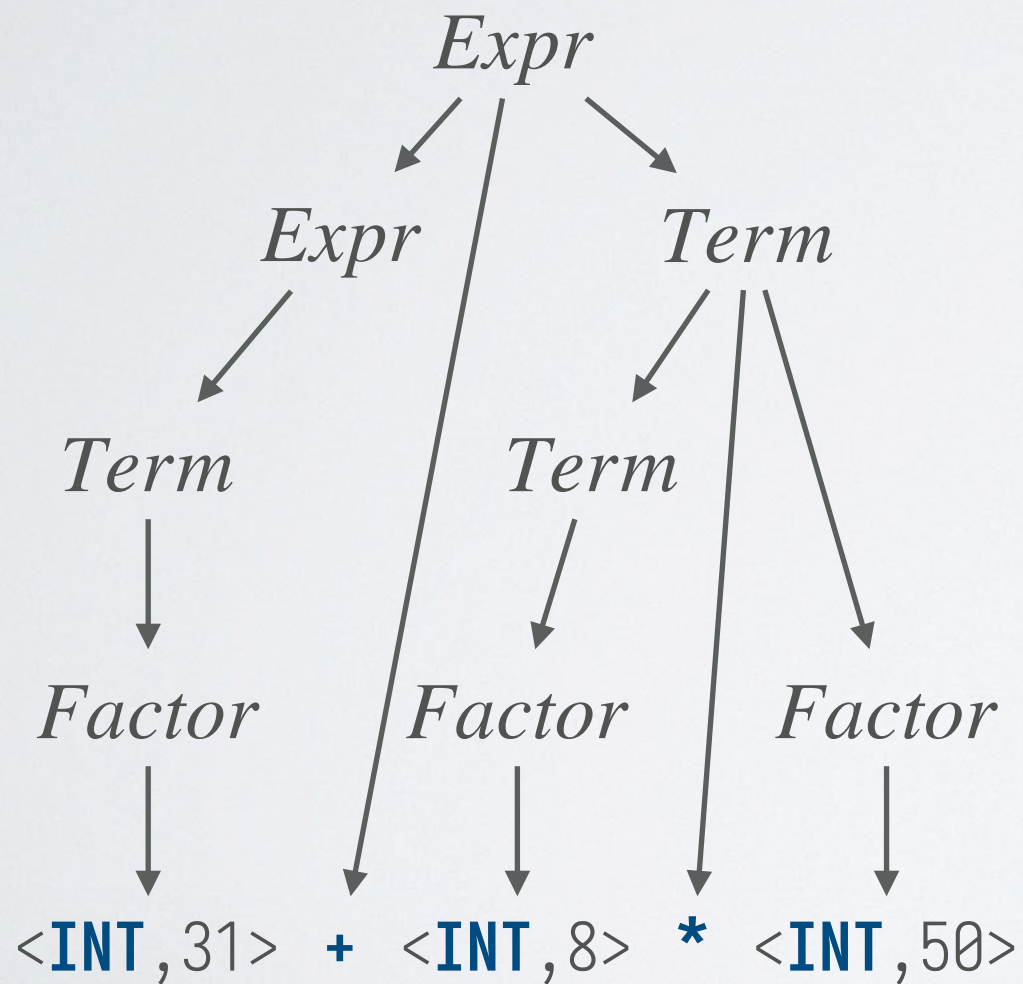
$$E ::= \text{add}(E, E) \mid \text{sub}(E, E) \mid \text{mul}(E, E) \mid \text{div}(E, E) \mid \text{intv}(i)$$

产生规则	属性计算规则
$Expr \rightarrow Expr_1 + Term$	$Expr.ast = \text{add}(Expr_1.ast, Term.ast)$
$Expr \rightarrow Expr_1 - Term$	$Expr.ast = \text{sub}(Expr_1.ast, Term.ast)$
$Expr \rightarrow Term$	$Expr.ast = Term.ast$
$Term \rightarrow Term_1 * Factor$	$Term.ast = \text{mul}(Term_1.ast, Factor.ast)$
$Term \rightarrow Term_1 / Factor$	$Term.ast = \text{div}(Term_1.ast, Factor.ast)$
$Term \rightarrow Factor$	$Term.ast = Factor.ast$
$Factor \rightarrow (Expr)$	$Factor.ast = Expr.ast$
$Factor \rightarrow INT$	$Factor.ast = \text{intv}(INT.intval)$

属性文法应用：翻译到中间表示

31 + 8 * 50

add(intv(31), mul(intv(8), intv(50)))



属性文法应用：类型检查

- 通过文法属性记录程序结构的类型信息
- 检查四则运算表达式计算结果的类型, 其中考虑整数和浮点数两种基本类型: $Factor ::= (Expr) \mid INT \mid FLOAT$

产生规则	属性计算规则
$Expr \rightarrow Expr_1 + Term$	$Expr.type = \mathcal{F}_+(Expr_1.type, Term.type)$
$Expr \rightarrow Expr_1 - Term$	$Expr.type = \mathcal{F}_-(Expr_1.type, Term.type)$
$Expr \rightarrow Term$	$Expr.type = Term.type$
$Term \rightarrow Term_1 * Factor$	$Term.type = \mathcal{F}_\times(Term_1.type, Factor.type)$
$Term \rightarrow Term_1 / Factor$	$Term.type = \mathcal{F}_\div(Term_1.type, Factor.type)$
$Term \rightarrow Factor$	$Term.type = Factor.type$
$Factor \rightarrow (Expr)$	$Factor.type = Expr.type$
$Factor \rightarrow INT$	$Factor.type = int$
$Factor \rightarrow FLOAT$	$Factor.type = float$

由编程语言规定的
类型计算规则

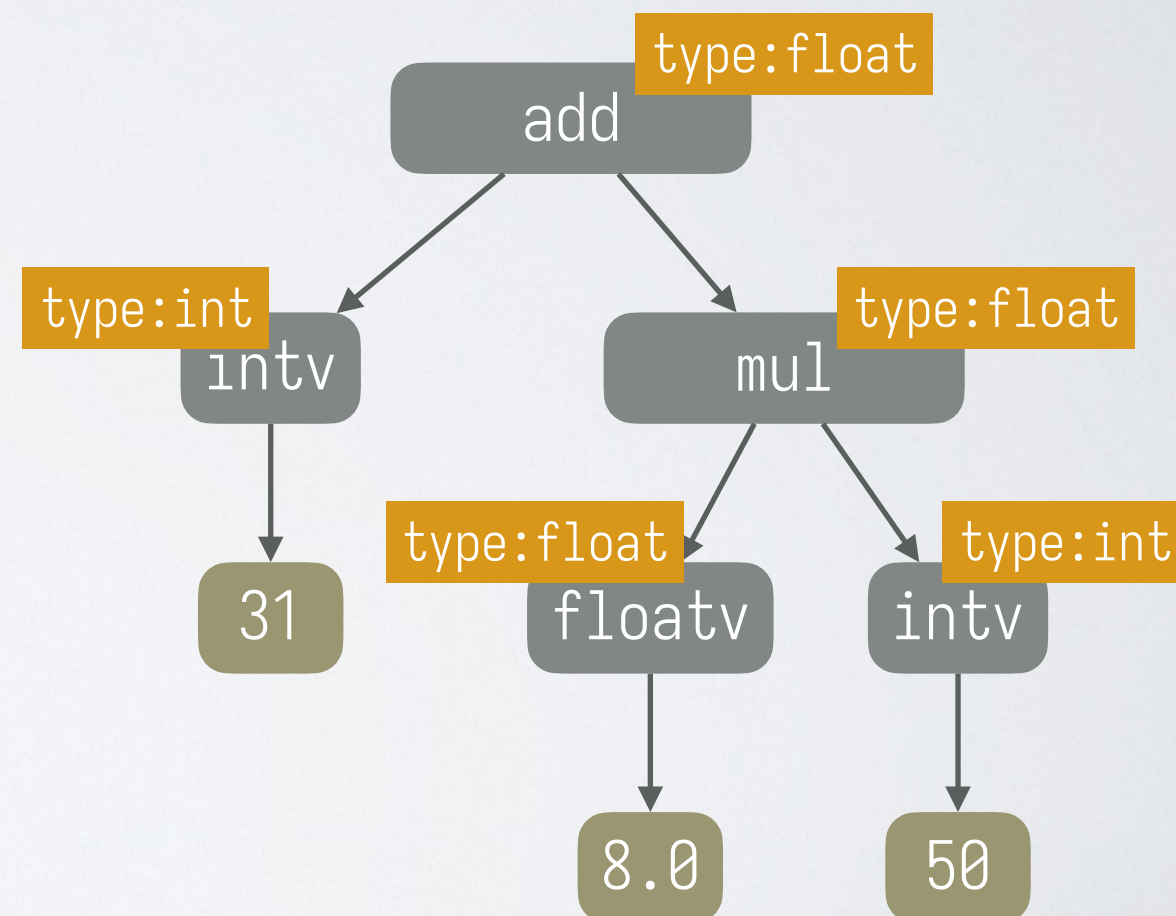
$\mathcal{F}_+$	int	float
int	int	float
float	float	float

属性文法应用：类型检查

- 也可以定义为**抽象语法树**(AST)上的属性文法

$$E ::= \text{add}(E, E) \mid \text{sub}(E, E) \mid \text{mul}(E, E) \mid \text{div}(E, E) \mid \text{intv}(i) \mid \text{floatv}(f)$$

产生规则	属性计算规则
$E \rightarrow \text{add}(E_1, E_2)$	$E.type = \mathcal{F}_+(E_1.type, E_2.type)$
$E \rightarrow \text{sub}(E_1, E_2)$	$E.type = \mathcal{F}_-(E_1.type, E_2.type)$
$E \rightarrow \text{mul}(E_1, E_2)$	$E.type = \mathcal{F}_\times(E_1.type, E_2.type)$
$E \rightarrow \text{div}(E_1, E_2)$	$E.type = \mathcal{F}_\div(E_1.type, E_2.type)$
$E \rightarrow \text{intv}(i)$	$E.type = \text{int}$
$E \rightarrow \text{floatv}(f)$	$E.type = \text{float}$



属性文法应用：解释执行

- 前面的简单循环的解释执行是通过存在依赖环的文法实现的
- 也可以通过属性文法定义一个小步 (small-step) 运算
 - 比如计算四则运算表达式 (AST) 一步运算后的结果表达式 (AST)

$$E ::= \text{add}(E, E) \mid \text{sub}(E, E) \mid \text{mul}(E, E) \mid \text{div}(E, E) \mid \text{intv}(i) \mid \text{floatv}(f)$$

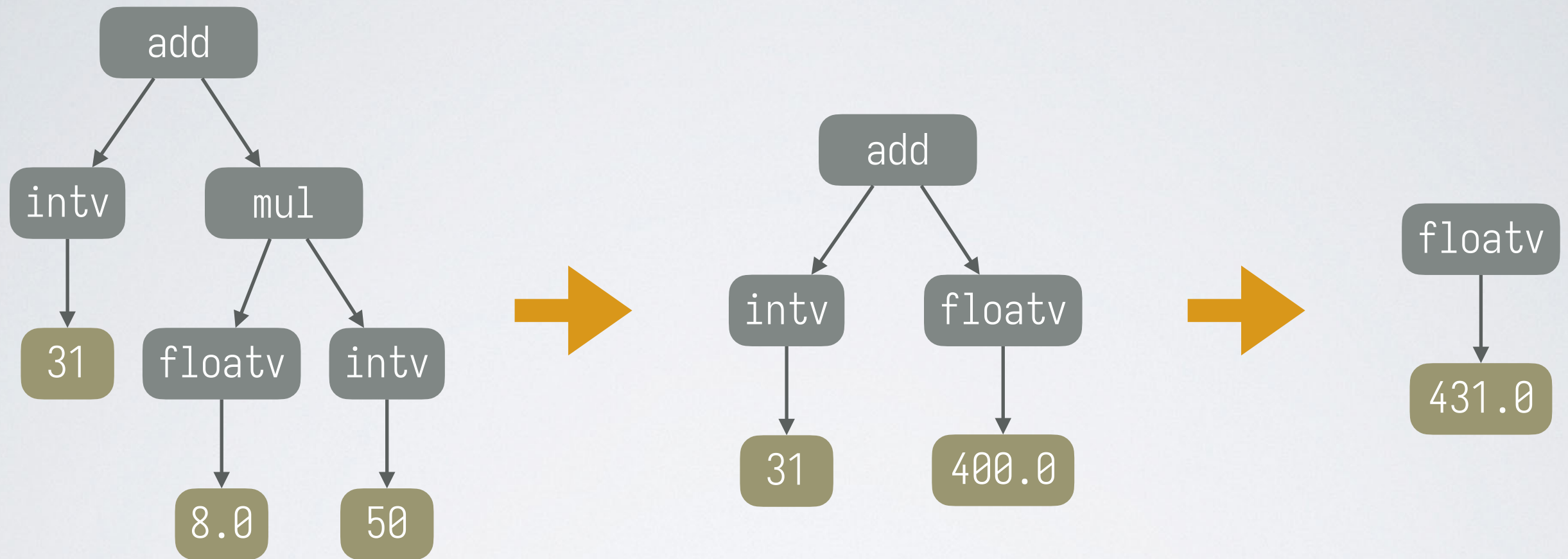
产生规则	属性计算规则
$E \rightarrow \text{add}(E_1, E_2)$	$E.\text{next} = \mathcal{C}_+(E_1, E_2)$
$E \rightarrow \text{sub}(E_1, E_2)$	$E.\text{next} = \mathcal{C}_-(E_1, E_2)$
$E \rightarrow \text{mul}(E_1, E_2)$	$E.\text{next} = \mathcal{C}_\times(E_1, E_2)$
$E \rightarrow \text{div}(E_1, E_2)$	$E.\text{next} = \mathcal{C}_\div(E_1, E_2)$
$E \rightarrow \text{intv}(i)$	$E.\text{next} = i$
$E \rightarrow \text{floatv}(f)$	$E.\text{next} = f$

$$\mathcal{C}_+(E_1, E_2) =$$

```

if  $E_1$  is not a value yet then
    add( $E_1.\text{next}$ ,  $E_2$ )
else if  $E_2$  is not a value yet then
    add( $E_1$ ,  $E_2.\text{next}$ )
else
    if both  $E_1$  and  $E_2$  are intv then
        intv( $E_1.\text{next} + E_2.\text{next}$ )
    else
        floatv( $E_1.\text{next} + E_2.\text{next}$ )
    
```

属性文法应用：解释执行



```
add(
  intv(31),
  mul(floatv(8.0), intv(50))
)
```

```
add(
  intv(31),
  floatv(400.0)
)
```

```
floatv(431.0)
```

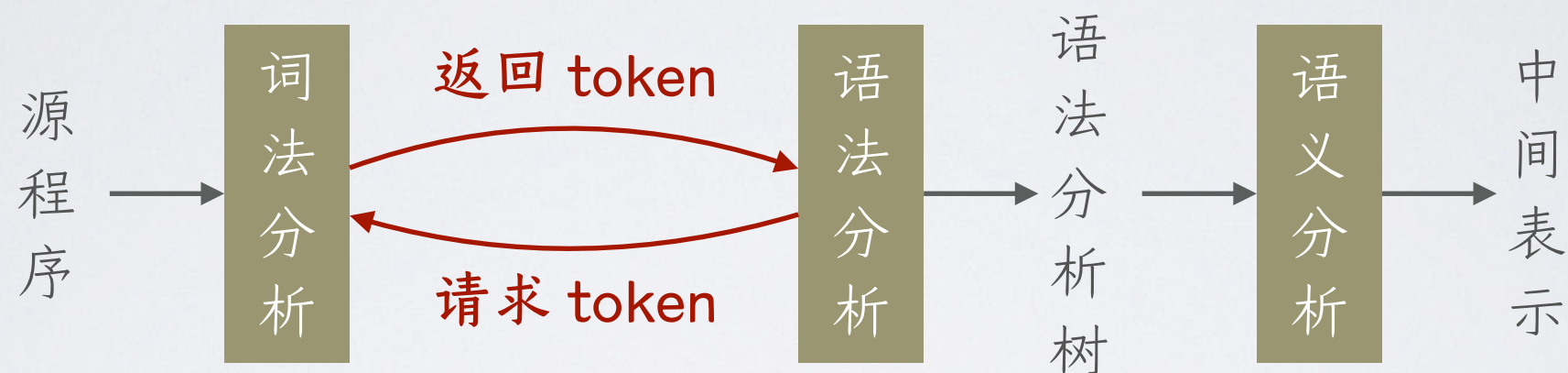


主要内容

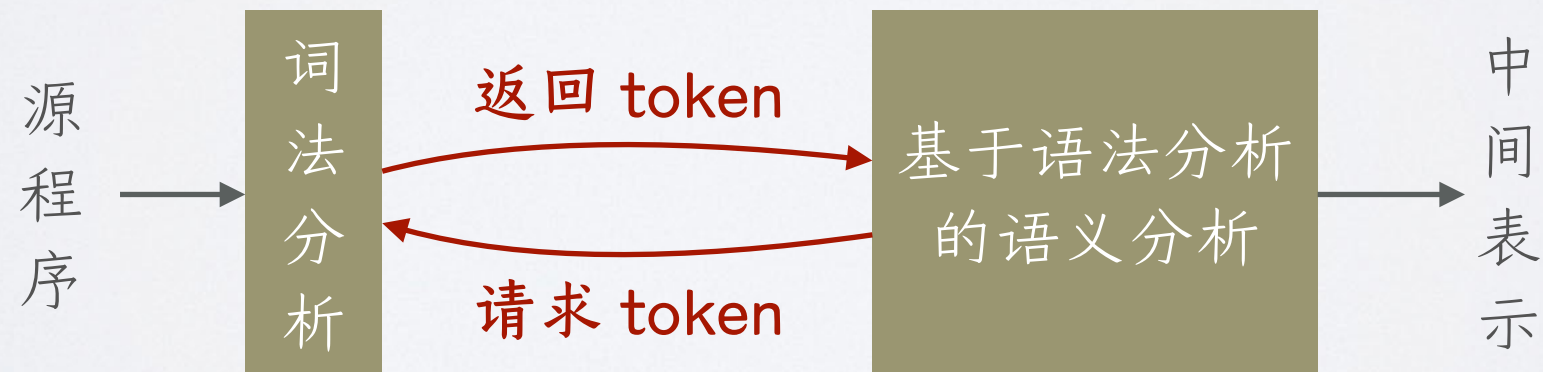
- ◎ 语义分析的作用
- ◎ 语义分析的规约
- ◎ 语义分析的手动实现

基于语法分析的实现

- 如果分开实现语法和语义分析, 后者可以通过遍历语法分析树来实现(通常可以使用深度优先遍历)



- 本部分侧重如何实现语法、语义分析的同步进行



回顾：自顶向下的语法分析

◎ 递归下降分析

- ❖ 为每个非终结符号实现一个递归过程
- ❖ 该过程通过「向前看」符号确定产生规则

	$S' \rightarrow S \text{ EOF}$
[1]	$S \rightarrow \epsilon$
[2]	$S \rightarrow [S]S$

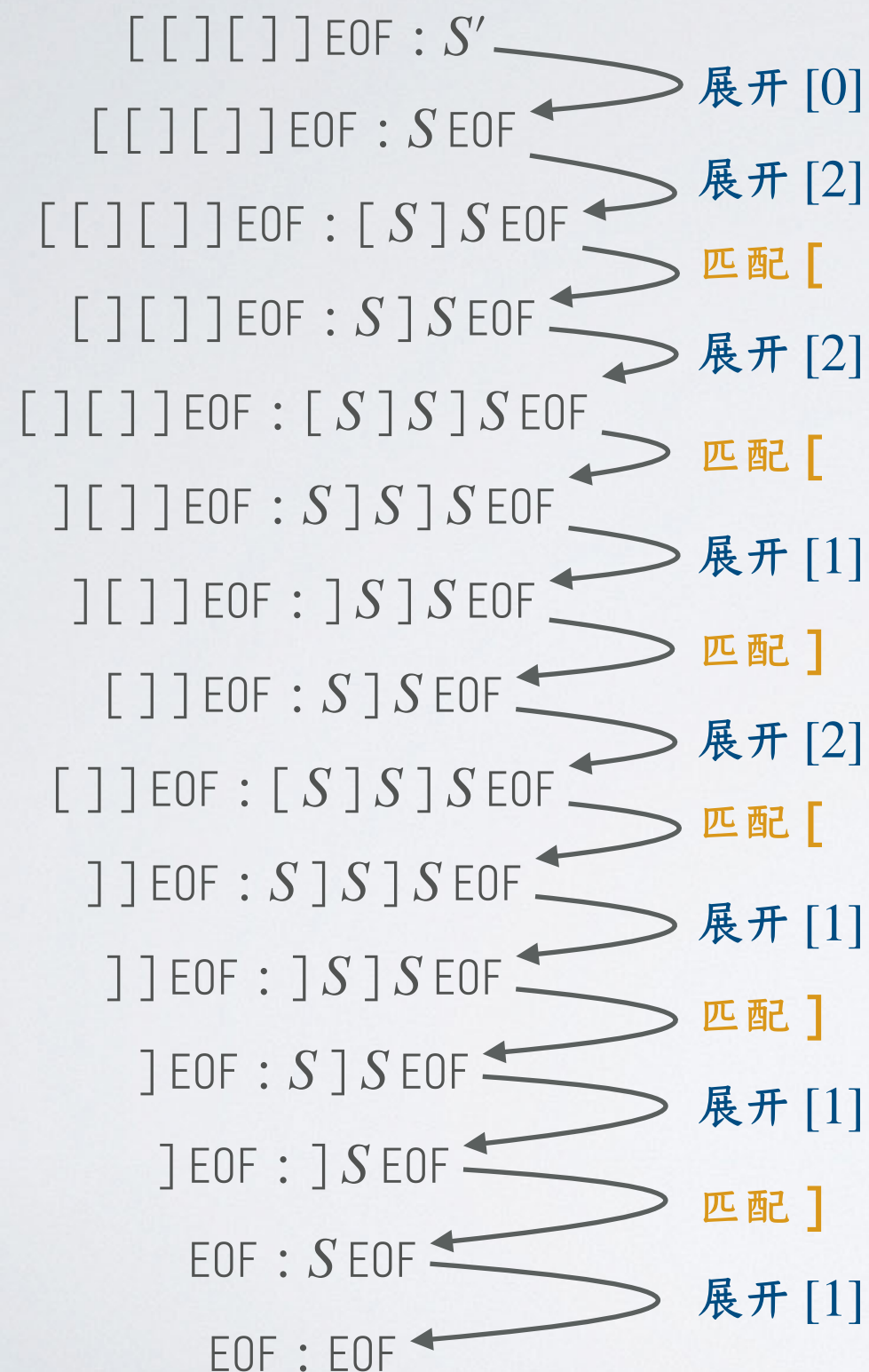
$\text{FIRST}(S) = \{[, \epsilon\}$

$\text{FOLLOW}(S) = \{\text{EOF},]\}$

```
S'() {  
    if (token == LSQUARE ||  
        token == EOF) {  
        if (S()) {  
            if (token == EOF) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

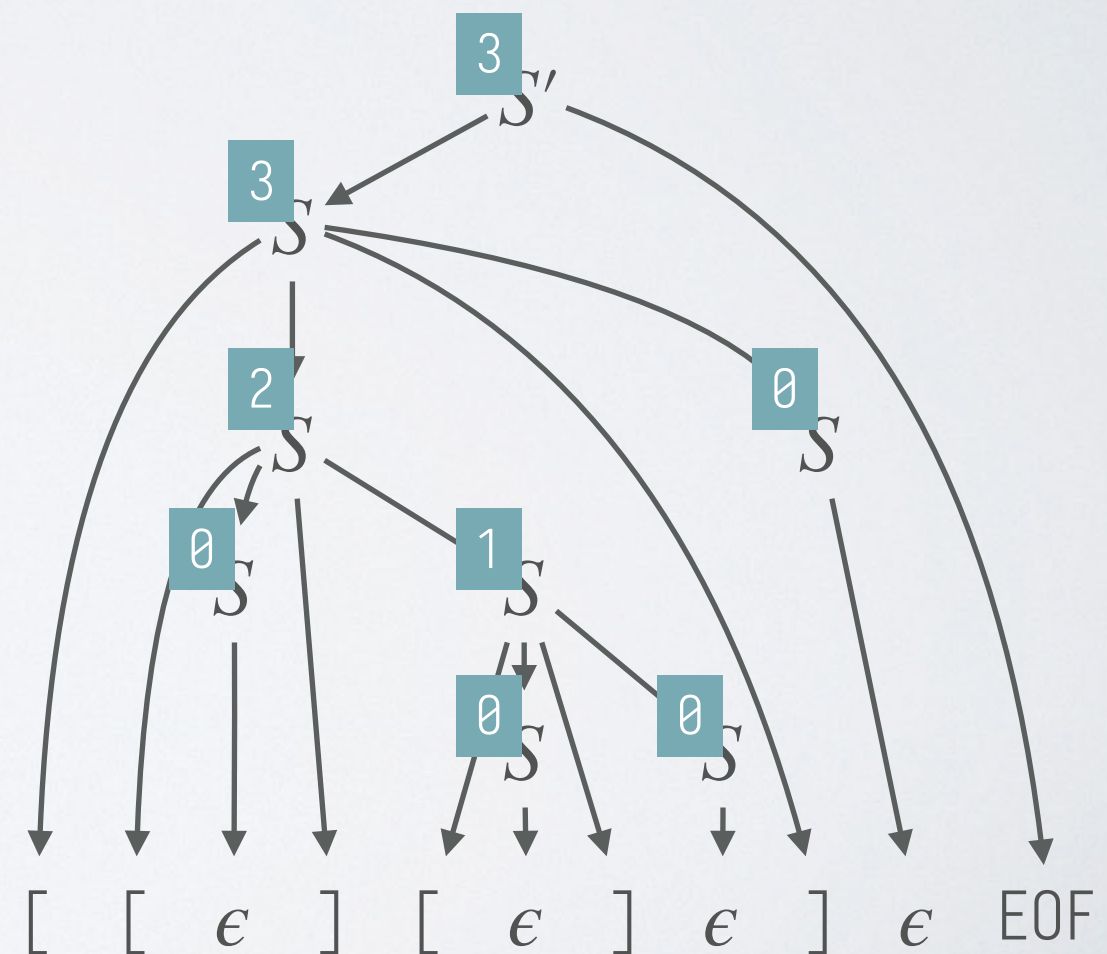
```
S() {  
    if (token == LSQUARE) {  
        token = next_token();  
        if (S()) {  
            if (token == RSQUARE) {  
                token = next_token();  
                if (S()) return true;  
            }  
        }  
    } else if (token == EOF ||  
                token == RSQUARE) {  
        return true;  
    }  
    return false;  
}
```

自顶向下的语义分析



综合属性 $S'.cnt$ 和 $S.cnt$ 记录匹配的括号对数

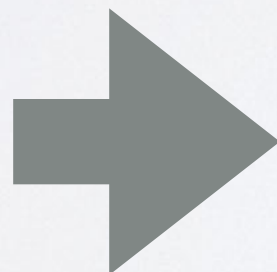
产生规则	属性计算规则
[0] $S' \rightarrow S EOF$	$S'.cnt = S.cnt$
[1] $S \rightarrow \epsilon$	$S.cnt = 0$
[2] $S \rightarrow [S_1] S_2$	$S.cnt = 1 + S_1.cnt + S_2.cnt$



S 属性的文法的递归下降实现

- 回顾: S 属性的文法中只有综合属性
- 回顾: 递归下降语法分析对每个非终结符号有一个递归过程
 - ❖ 过程没有参数, 返回一个布尔值表示是否分析成功
- S 属性的文法的递归下降分析
 - ❖ 每个非终结符号的过程返回该符号所有的综合属性

$A ::= \alpha_1 \mid \dots$



```
A() {  
  if (token  $\in$  FIRST( $\alpha_1$ )) {  
    按照  $\alpha_1$  中符号依次进行 token 匹配或递归调用;  
    按照  $A \rightarrow \alpha_1$  的属性计算规则进行计算;  
    return A 的综合属性;  
  } else {  
    .....  
  }  
  error();  
}
```

S 属性的文法的递归下降实现

产生规则	属性计算规则
[0] $S' \rightarrow S \text{ EOF}$	$S'.\text{cnt} = S.\text{cnt}$
[1] $S \rightarrow \epsilon$	$S.\text{cnt} = 0$
[2] $S \rightarrow [S_1] S_2$	$S.\text{cnt} = 1 + S_1.\text{cnt} + S_2.\text{cnt}$

```

S'() {
    if (token == LSQUARE || token == EOF) {
        cnt = S();
        if (token == EOF) {
            return cnt;
        }
    }
    error();
}

```

```

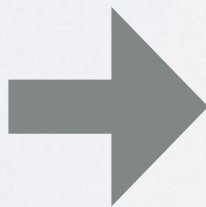
S() {
    if (token == LSQUARE) {
        token = next_token();
        cnt1 = S();
        if (token == RSQUARE) {
            token = next_token();
            cnt2 = S();
            return 1 + cnt1 + cnt2;
        }
    }
    else if (token == EOF || token == RSQUARE) {
        return 0;
    }
    error();
}

```

L 属性的文法的递归下降实现

- 回顾: L 属性的文法中既有综合属性也有继承属性
 - ❖ 每个继承属性可以依赖左侧结点的属性和 parent 结点的继承属性
- L 属性的文法的递归下降分析
 - ❖ 每个非终结符号的过程以其所有的继承属性为参数
 - ❖ 每个非终结符号的过程返回该符号所有的综合属性

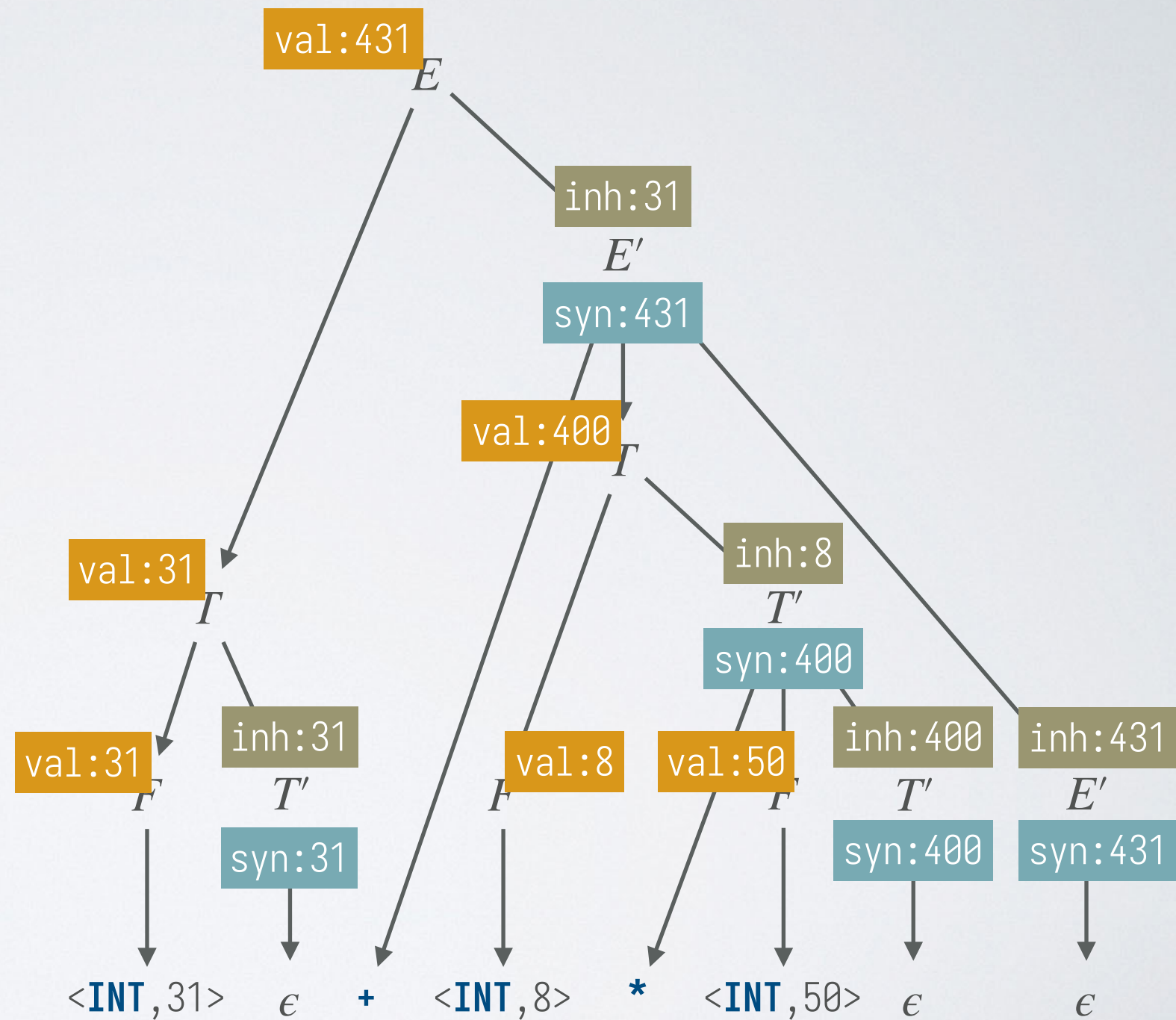
$A ::= \alpha_1 \mid \dots$



```
A(A 的继承属性) {  
  if (token  $\in$  FIRST( $\alpha_1$ )) {  
    按照  $\alpha_1$  中符号依次进行 token 匹配或递归调用;  
    因为是 L 属性的文法, 有足够的信息计算中途的递归调用所需的继承属性;  
    return A 的综合属性;  
  } else {  
    .....  
  }  
  error();  
}
```

L 属性的文法的递归下降实现

产生规则	属性计算规则
$E \rightarrow TE'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \rightarrow +TE'_1$	$E'_1.inh = E'.inh + T.val$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow INT$	$F.val = INT.intval$



L 属性的文法的递归下降实现

产生规则	属性计算规则
$E \rightarrow TE'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \rightarrow +TE'_1$	$E'_1.inh = E'.inh + T.val$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow INT$	$F.val = INT.intval$

```

E() {
    if (token == LPAREN || token == INT) {
        T_val = T();
        E'_syn = E'(T_val);
        return E'_syn;
    }
    error();
}

```

```

E'(E'_inh) {
    if (token == PLUS) {
        token = next_token();
        T_val = T();
        E'1_syn = E'(E'_inh + T_val);
        return E'1_syn;
    } else if (token == EOF || token == RPAREN) {
        return E'_inh;
    }
    error();
}

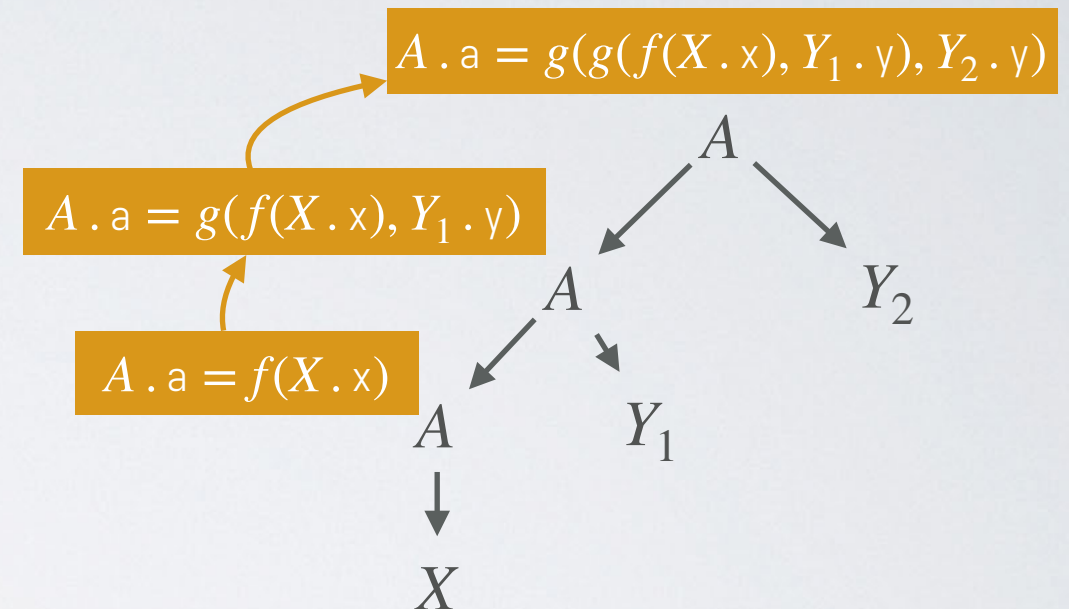
```

带左递归的 S 属性的文法

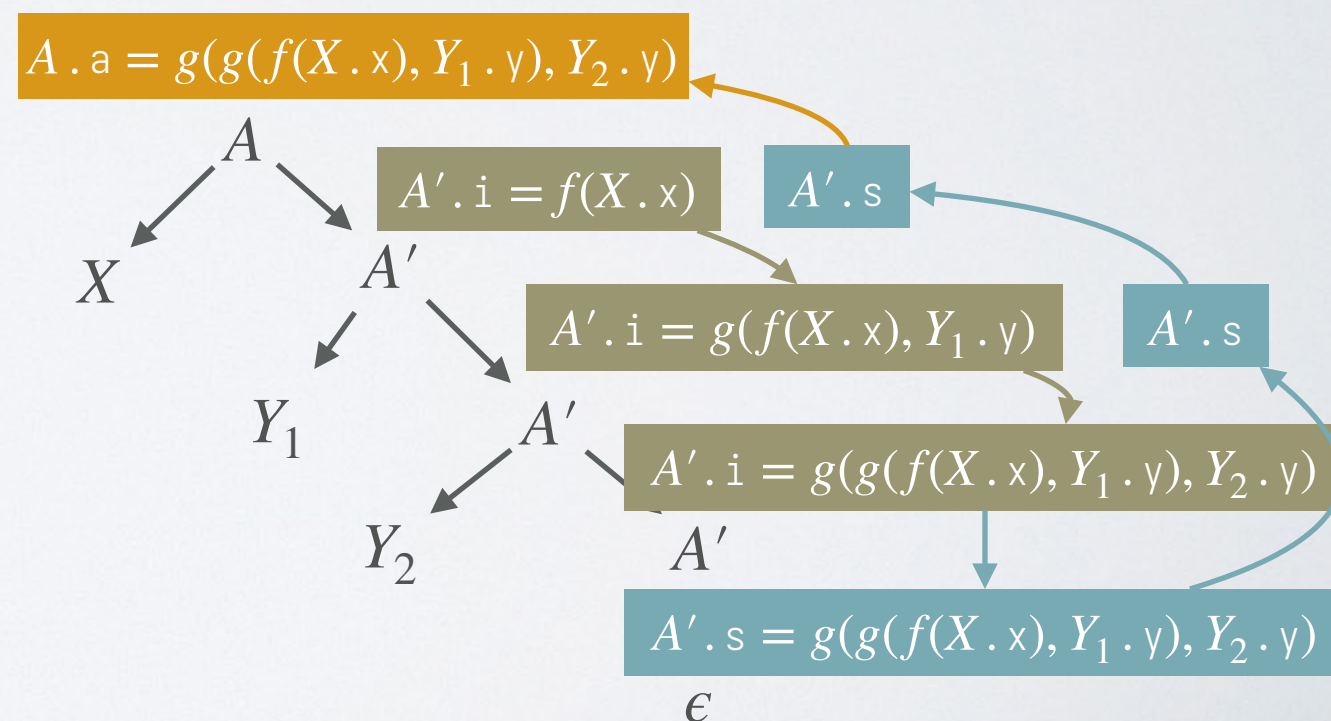
消除左递归时, 可能需要引入继承属性

例:

产生规则	属性计算规则
$A \rightarrow A_1 Y$	$A.a = g(A_1.a, Y.y)$
$A \rightarrow X$	$A.a = f(X.x)$



产生规则	属性计算规则
$A \rightarrow X A'$	$A'.i = f(X.x)$ $A.a = A'.s$
$A' \rightarrow Y A'_1$	$A'_1.i = g(A'.i, Y.y)$ $A'.s = A'_1.s$
$A' \rightarrow \epsilon$	$A'.s = A'.i$



主要内容

- ◎ 语义分析的作用
- ◎ 语义分析的规约
- ◎ 语义分析的手动实现

- ◎ 实践演示

玩具语言

- ◎ C 语言, 但只有主函数, 只有 int 类型, 只有直线代码

```
int main() {  
    int x = 10;  
    int y = x + 1 * x;  
    int z = x * (y + 1);  
    return z % 66;  
}
```

```
int main() {  
    int y;  
    y = 1;  
    int main;  
    main = y + 1;  
    return main;  
}
```

```
int main() {  
    int a = 1;  
    {  
        a = a + 2;  
        int a = 3;  
        a = a + 4;  
    }  
    a = a + 5;  
    return a;  
}
```

```
int main() {  
    int a = 1;  
    int b = a + 2;  
    {  
        b = a + b;  
        {  
            int a = 3;  
            b = b + a;  
            { int b = 1; }  
            { return b; }  
        }  
        int b = 1;  
    }  
}
```

玩具语言的抽象语法树表示

具体语法

$$\begin{aligned} Expr &\leftarrow Term ((+ \mid -) Term)^* \\ Term &\leftarrow Factor ((* \mid / \mid \%) Factor)^* \\ Factor &\leftarrow (Expr) \mid ID \mid NUMBER \\ Stmt &\leftarrow Block \mid \text{int } ID (= Expr)? ; \mid ID = Expr ; \mid \text{return } Expr ; \\ Block &\leftarrow \{ Stmt^* \} \\ Func &\leftarrow \text{int } ID () Block \\ Prog &\leftarrow Func EOF \end{aligned}$$

抽象语法

$$\begin{aligned} E &::= \text{binop}(op, E, E) \mid \text{id}(x) \mid \text{number}(i) \\ op &::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div} \mid \text{mod} \\ S &::= \text{block}(\vec{S}) \mid \text{decl}(x) \mid \text{decl}(x, E) \mid \text{assign}(x, E) \mid \text{ret}(E) \\ F &::= \text{func}(x, S) \\ P &::= \text{prog}(F) \end{aligned}$$

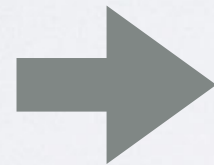
对抽象语法树进一步语义推敲

$$S ::= \text{block}(\vec{S}) \mid \text{decl}(x) \mid \text{decl}(x, E) \mid \text{assign}(x, E) \mid \text{ret}(E)$$

简化结构, 使得
语义更显然

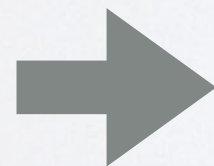
$$\hat{S} ::= \text{scope}(\hat{S}) \mid \text{decl}(x, \hat{S}) \mid \text{assign}(x, E) \mid \text{ret}(E) \mid \text{seq}(\hat{S}, \hat{S}) \mid \text{nop}$$

```
int x = 10;
int y = x + 1 * x;
int z = x * (y + 1);
return z % 66;
```



```
decl(x, seq(assign(x, 10),
            decl(y, seq(assign(y, x + (1 * x)),
                        decl(z, seq(assign(z, x * (y + 1)),
                                    ret(z % 66))))))))
```

```
int a = 1;
{
  a = a + 2;
  int a = 3;
  a = a + 4;
}
a = a + 5;
return a;
```



```
decl(a,
    seq(assign(a, 1),
        seq(scope(seq(assign(a, a + 2),
                        decl(a,
                            seq(assign(a, 3),
                                assign(a, a + 4))))),
            seq(assign(a, a + 5),
                ret(a))))))
```

进行语义推敲的 R 属性的文法

$$\hat{S} ::= \text{scope}(\hat{S}) \mid \text{decl}(x, \hat{S}) \mid \text{assign}(x, E) \mid \text{ret}(E) \mid \text{seq}(\hat{S}, \hat{S}) \mid \text{nop}$$

$$\begin{aligned} S &::= \text{block}(SS) \\ &\mid \text{decl}(x) \\ &\mid \text{decl}(x, E) \\ &\mid \text{assign}(x, E) \\ &\mid \text{ret}(E) \\ SS &::= \epsilon \\ &\mid S, SS \end{aligned}$$

产生式	属性计算
$S \rightarrow \text{block}(SS)$	$SS.\text{cont} = \text{nop}$ $S.\text{elab} = \text{seq}(\text{scope}(SS.\text{elab}), S.\text{cont})$
$SS \rightarrow \epsilon$	$SS.\text{elab} = SS.\text{cont}$
$SS \rightarrow S, SS_1$	$SS_1.\text{cont} = SS.\text{cont}$ $S.\text{cont} = SS_1.\text{elab}$ $SS.\text{elab} = S.\text{elab}$
$S \rightarrow \text{decl}(x)$	$S.\text{elab} = \text{decl}(x, S.\text{cont})$
$S \rightarrow \text{decl}(x, E)$	$S.\text{elab} = \text{decl}(x, \text{seq}(\text{assign}(x, E), S.\text{cont}))$
$S \rightarrow \text{assign}(x, E)$	$S.\text{elab} = \text{seq}(\text{assign}(x, E), S.\text{cont})$
$S \rightarrow \text{ret}(E)$	$S.\text{elab} = \text{seq}(\text{ret}(E), S.\text{cont})$

继承属性 cont 记录后面已经推敲完的语句

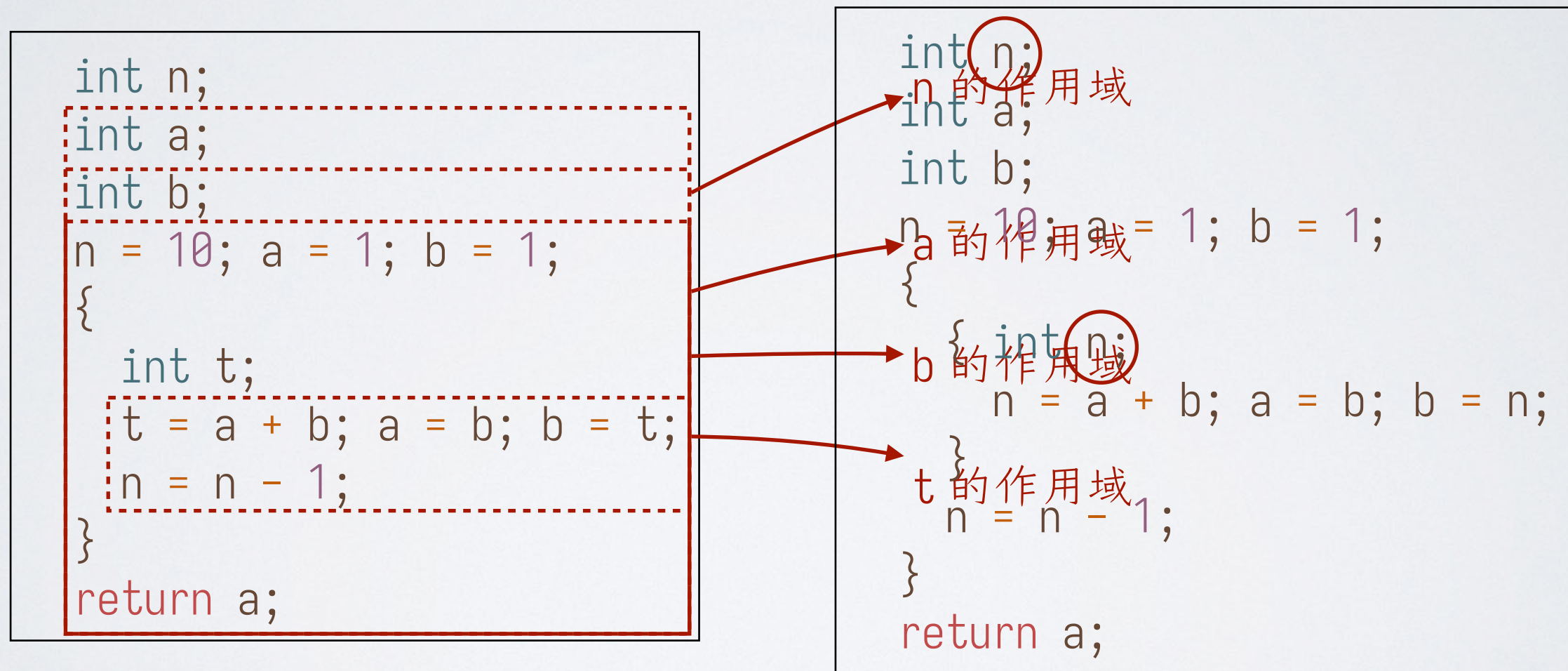
属性计算

综合属性 elab 记录在 cont 基础上完成推敲的语句

在抽象语法树上进行类型检查

● 需要检查的性质:

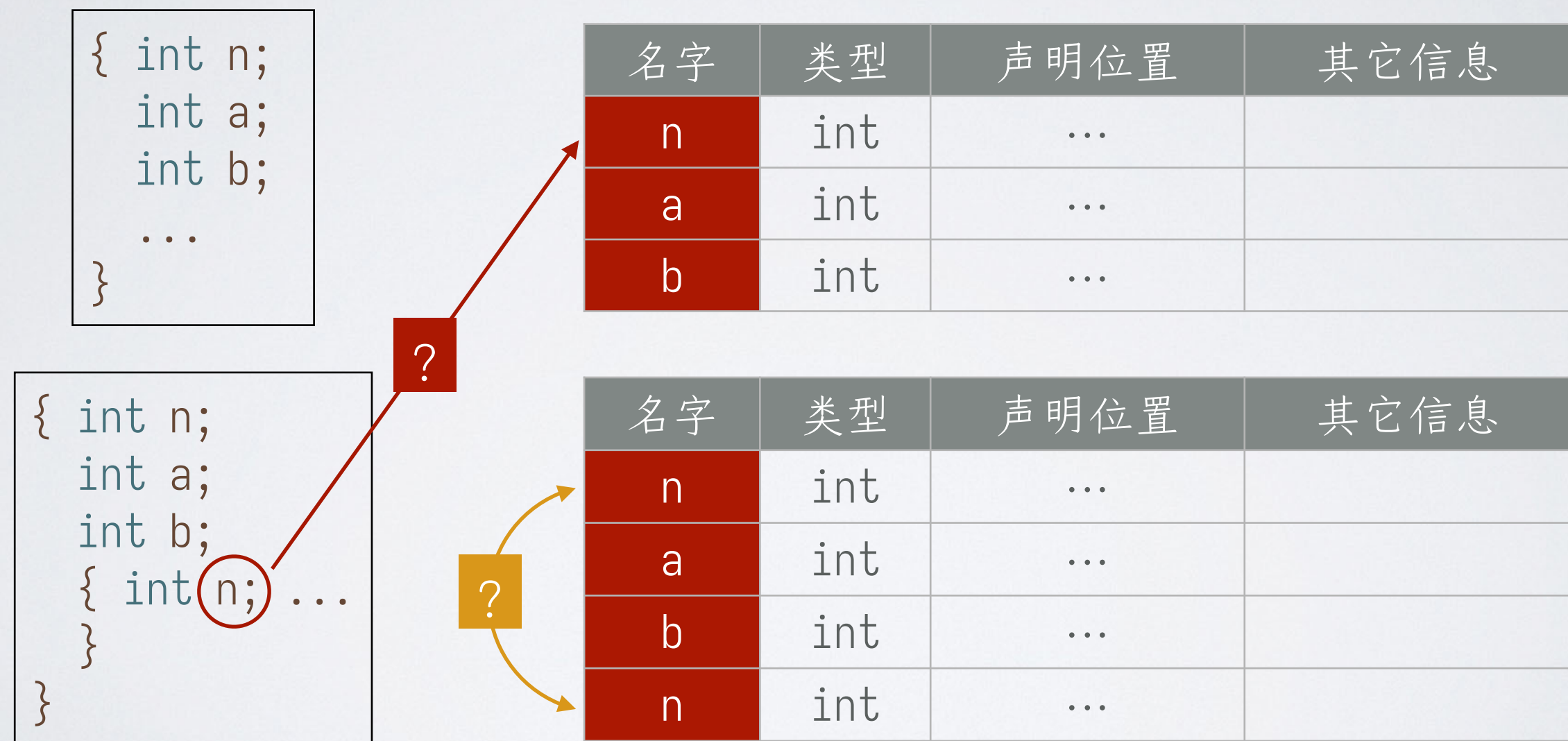
- ❖ 没有未经声明或赋值就使用的变量
- ❖ 同一作用域内没有重复声明的变量



● 问题: 如何正确处理作用域? 比如重复的变量名?

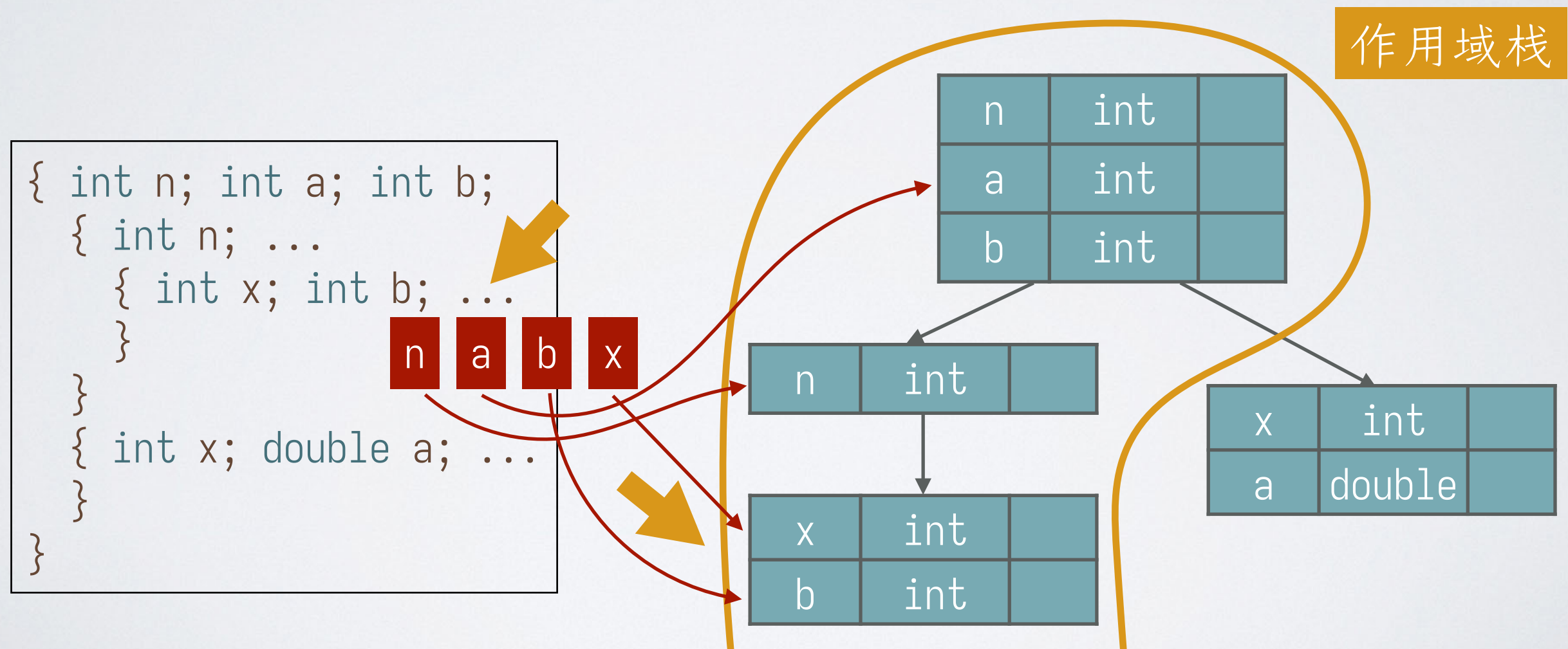
编译中使用的符号表

- 编译器通过**符号表(symbol table)**来汇总不同名字的信息
 - 在本部分我们重点考虑程序变量



通过符号表管理作用域

- 每个作用域对应一个符号表：多个符号表形成树状结构
- 在实现时，可以通过栈来存放当前符号表及其祖先
 - 对程序变量 x 的使用需查看**最靠近栈顶**的声明了 x 的作用域



符号表提供的接口

- `push_scope()`: 在栈顶压入一个新的作用域
- `pop_scope()`: 弹出栈顶的作用域
- `insert(name, info)`: 在栈顶作用域中记录名字和它的信息
- `lookup(name)`: 查找最靠近栈顶的名字对应的信息

```

{ int n; int a; int b;
  { int n; ...
    { int x; int b;
      x = n + a + b;
      ...
    }
  }
  { int x; double a; ...
  }
}
  
```

```

1. push_scope()
2. insert(n)
3. insert(a)
4. insert(b)
5. push_scope()
6. insert(n)
7. push_scope()
8. insert(x)
9. insert(b)
10. lookup(n)
  
```

```

11. lookup(a)
12. lookup(b)
13. lookup(x)
14. pop_scope()
15. pop_scope()
16. push_scope()
17. insert(x)
18. insert(a)
19. pop_scope()
20. pop_scope()
  
```

进行类型检查的 L 属性的文法

$$\hat{S} ::= \text{scope}(\hat{S}) \mid \text{decl}(x, \hat{S}) \mid \text{assign}(x, E) \mid \text{ret}(E) \mid \text{seq}(\hat{S}, \hat{S}) \mid \text{nop}$$

$$E ::= \text{binop}(op, E, E) \mid \text{id}(x) \mid \text{number}(i)$$

继承属性 table_in 记录处理

完前面语句后的符号表

属性计算规则

$\hat{S} \rightarrow \text{scope}(\hat{S}_1)$	$\hat{S}_1.\text{table_in} = \hat{S}.\text{table_in}.\text{push_scope}()$ $\hat{S}.\text{table_out} = \hat{S}_1.\text{table_out}.\text{pop_scope}()$	综合属性 table_out 表示 处理完该语句后的符号表
$\hat{S} \rightarrow \text{decl}(x, \hat{S}_1)$	$\hat{S}_1.\text{table_in} = \hat{S}.\text{table_in}.\text{insert}(x, \text{UNASSIGNED})$ $\hat{S}.\text{table_out} = \hat{S}_1.\text{table_out}.\text{remove}(x)$	
$\hat{S} \rightarrow \text{assign}(x, E)$	$E.\text{table} = \hat{S}.\text{table_in}$ $\hat{S}.\text{table_out} = \hat{S}.\text{table_in}.\text{update}(x, \text{ASSIGNED})$	
$\hat{S} \rightarrow \text{ret}(E)$	$E.\text{table} = \hat{S}.\text{table_in}$	$\hat{S}.\text{table_out} = \hat{S}.\text{table_in}$
$\hat{S} \rightarrow \text{seq}(\hat{S}_1, \hat{S}_2)$	$\hat{S}_1.\text{table_in} = \hat{S}.\text{table_in}$ $\hat{S}.\text{table_out} = \hat{S}_2.\text{table_out}$	$\hat{S}_2.\text{table_in} = \hat{S}_1.\text{table_out}$
$E \rightarrow \text{binop}(op, E_1, E_2)$	$E_1.\text{table} = E.\text{table}$	$E_2.\text{table} = E.\text{table}$
$E \rightarrow \text{id}(x)$	error if $E.\text{table}.\text{lookup}(x) == \text{UNASSIGNED}$	

Visitor Pattern

```
struct Stmt {  
    virtual void accept(StmtVisitor &visitor) = 0;  
};
```

```
struct StmtVisitor {  
    virtual void visit_block(BlockStmt &stmt) = 0;  
    virtual void visit_decl(DeclStmt &stmt) = 0;  
    virtual void visit_assign(AssignStmt &stmt) = 0;  
    virtual void visit_ret(RetStmt &stmt) = 0;  
};
```

```
void BlockStmt::accept(StmtVisitor &visitor) override { visitor.visit_block(*this); }  
void DeclStmt::accept(StmtVisitor &visitor) override { visitor.visit_decl(*this); }  
void AssignStmt::accept(StmtVisitor &visitor) override { visitor.visit_assign(*this); }  
void RetStmt::accept(StmtVisitor &visitor) override { visitor.visit_ret(*this); }
```

通过从 StmtVisitor
派生子类给 Stmt 添加
功能, 而不用改动
Stmt 本身的实现

```
struct StmtPrinter : public StmtVisitor {  
    ostream &os;  
    string prefix;  
    StmtPrinter(ostream &os, const string &prefix = "")  
        : os(os), prefix(prefix) {}  
  
    void visit_block(BlockStmt &stmt) override;  
    void visit_decl(DeclStmt &stmt) override;  
    void visit_assign(AssignStmt &stmt) override;  
    void visit_ret(RetStmt &stmt) override;  
};
```

本讲小结

- ◎ 语义分析基于语法分析树, 提取程序的核心语义
 - ❖ 通常体现为翻译到中间表示、类型检查、解释执行等任务
- ◎ 语义分析的规约可以使用属性文法(也称为语法制导定义)
 - ❖ 综合属性、继承属性、属性依赖图、S 属性和 L 属性的文法
- ◎ 语义分析可以独立于语法分析, 通过遍历语法分析树实现
- ◎ 语义分析也可以结合自顶向下语法分析进行同步实现
- ◎ 后续可能讲的专题:
 - ❖ 自底向上语义分析
 - ❖ 类型系统的设计和实现

思考问题

- ◎ 为什么编译过程需要语义分析？
- ◎ 语义分析需要知道编程语言的哪些信息？
- ◎ 属性文法作为语义分析的规约，有哪些优势和缺陷？
- ◎ 我们尤其关注 L 属性的文法，为什么？根据对称性，类似可以定义 R 属性的文法，其是否有意义？
- ◎ 属性依赖有环的文法是否有实际意义？是否可以对属性值域和计算加一些限制，使得有环文法的属性值仍是良定义的？
- ◎ 语义分析和语法分析结合，有什么优点？有什么缺点？