北京大学计算机学院

2024 年春季学期　　《编译原理》

第四讲　语义分析

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
- 语义分析的自动生成

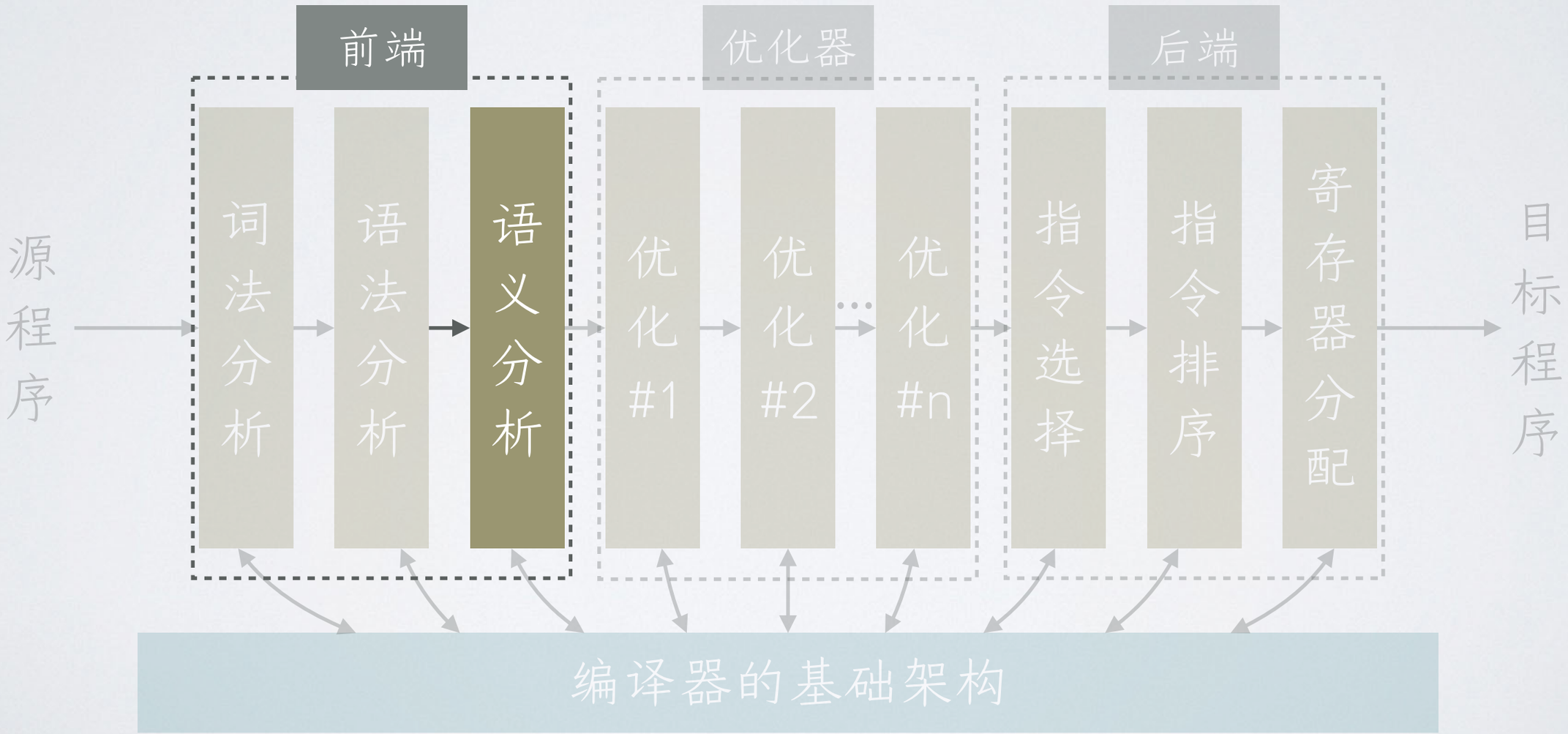
- 对应章节：第 5 章

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
- 语义分析的自动生成

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



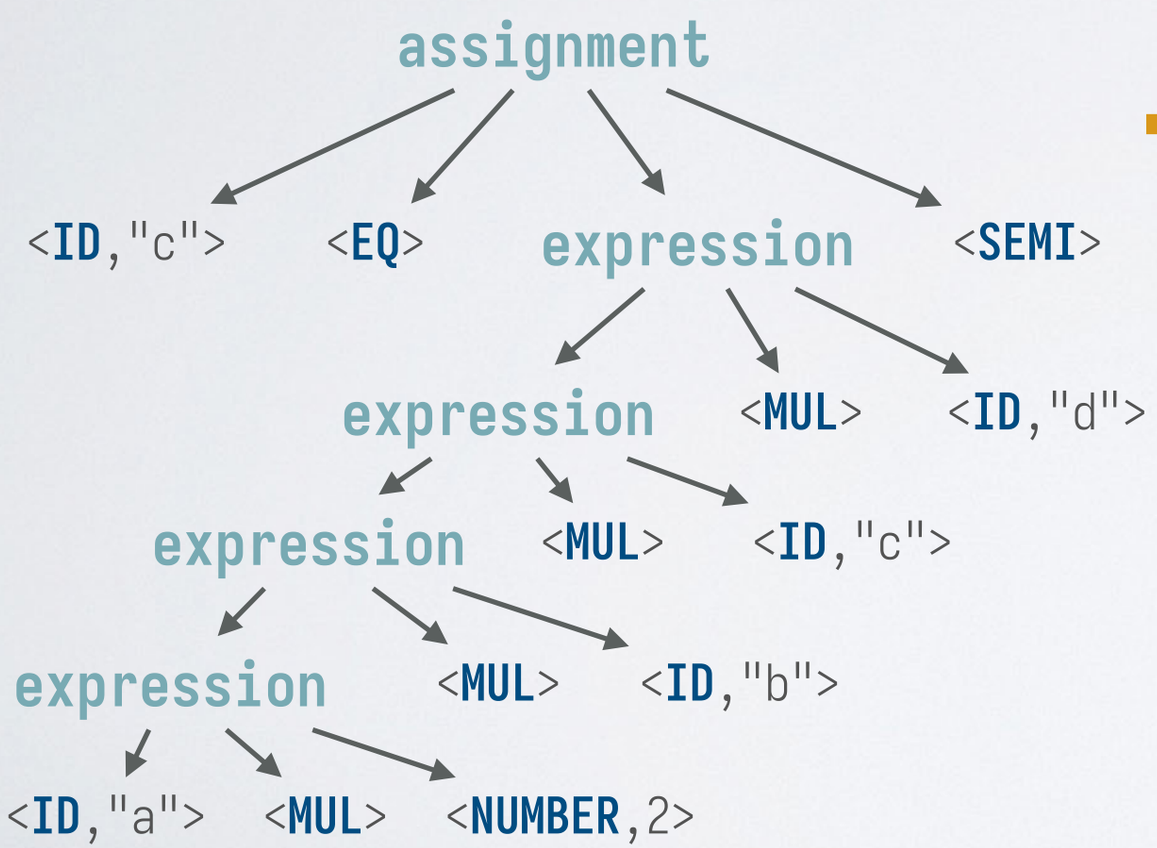
语义分析示例：翻译到中间表示

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

词法分析

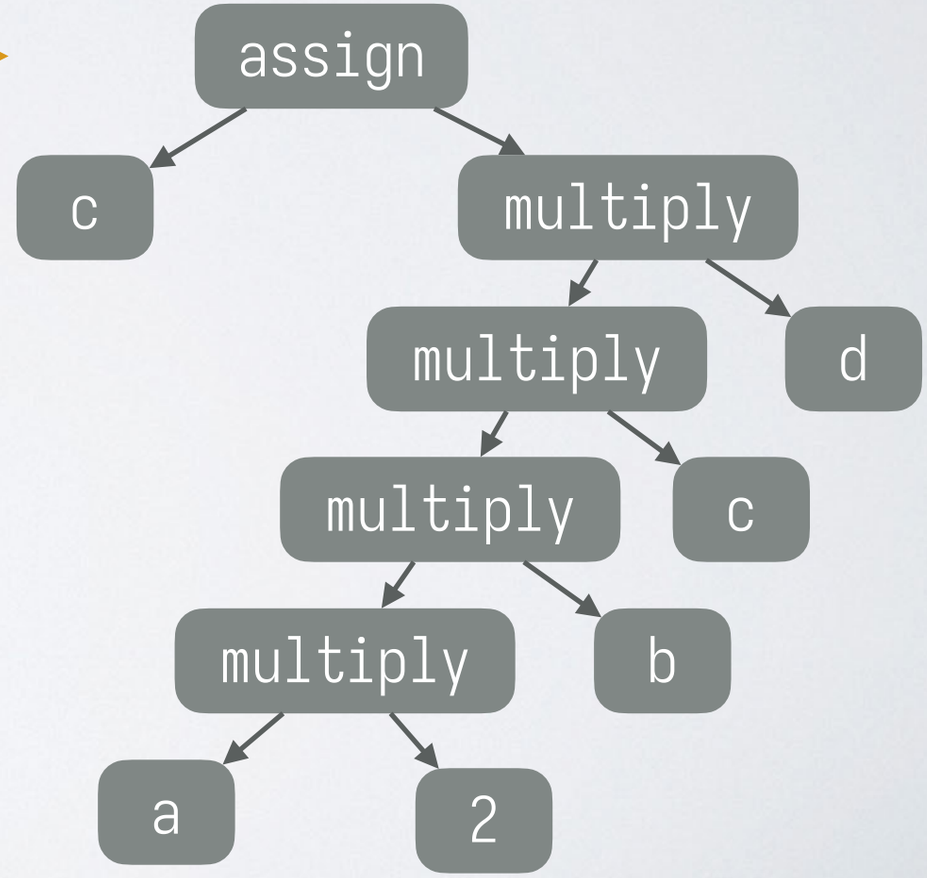
<ID,"c"> <EQ> <ID,"a"> <MUL> <NUMBER,2> <MUL> <ID,"b"> <MUL> <ID,"c"> <MUL> <ID,"d"> <SEMI>

语法分析



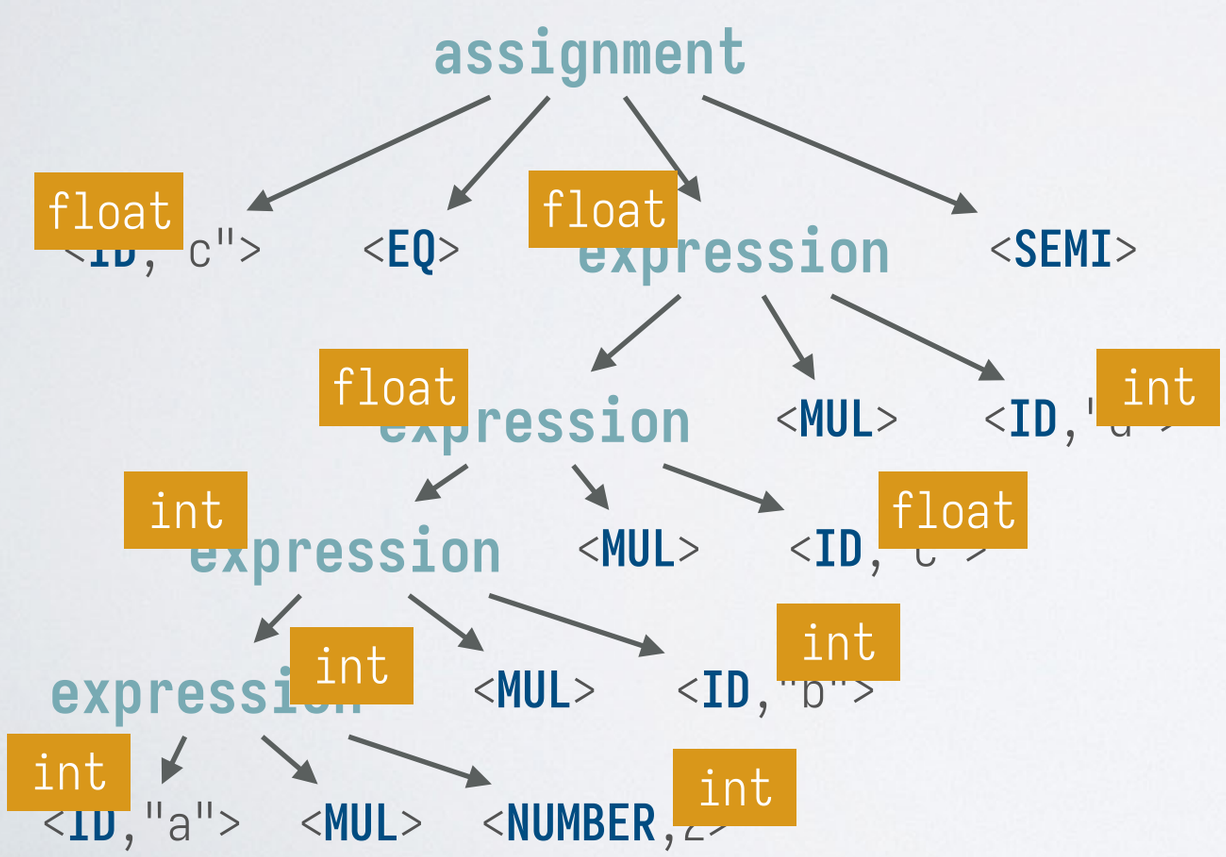
语义分析

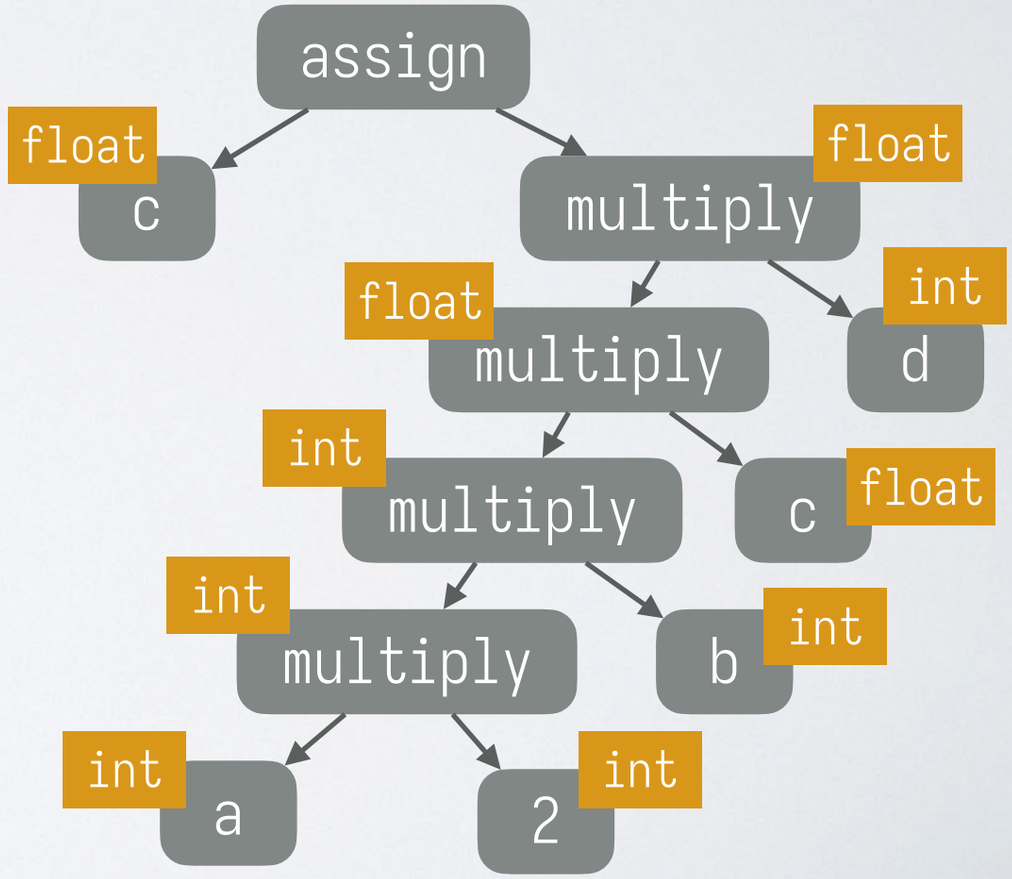
抽象语法树



语义分析示例：类型检查

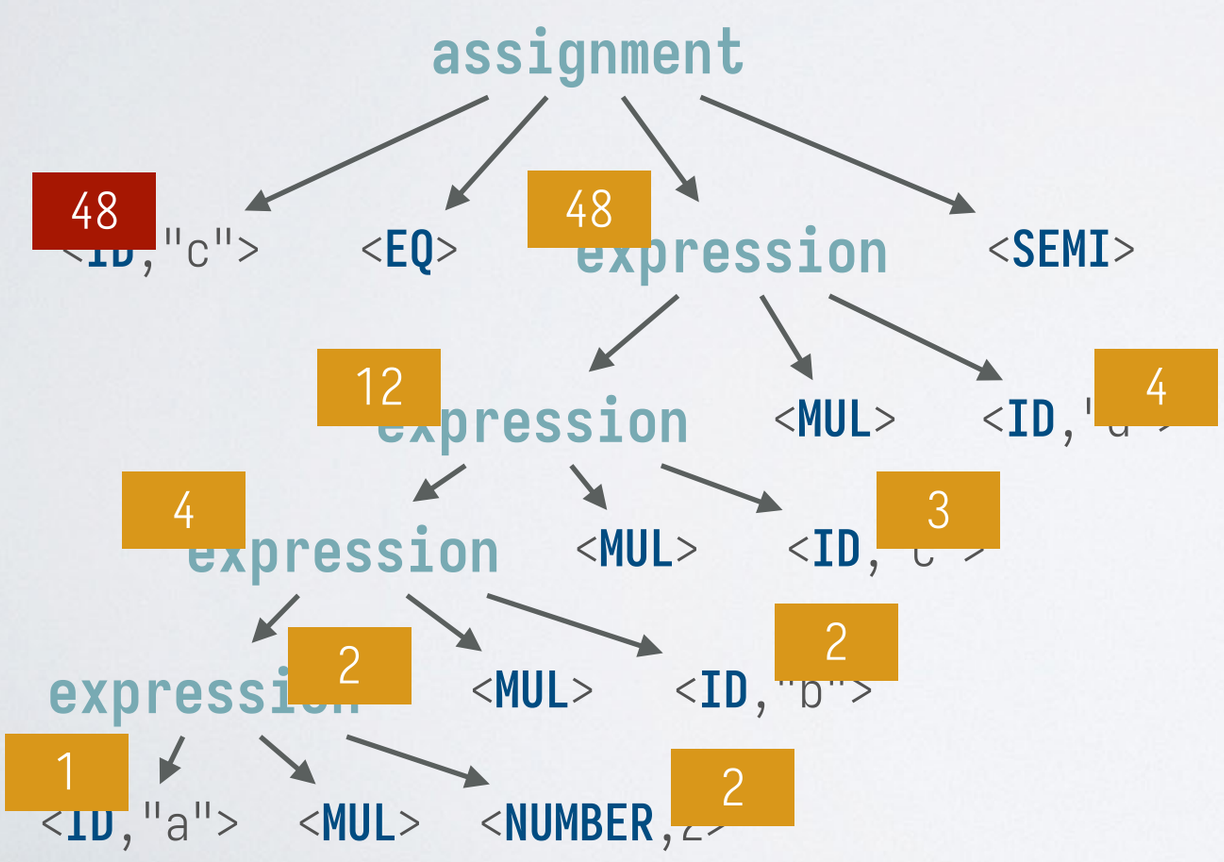
- 分析**名字**和**表达式**的类型，确保在上下文中的使用是合法的
- 避免运行时类型检查的开销
- 可以在语法分析树或抽象语法树上进行

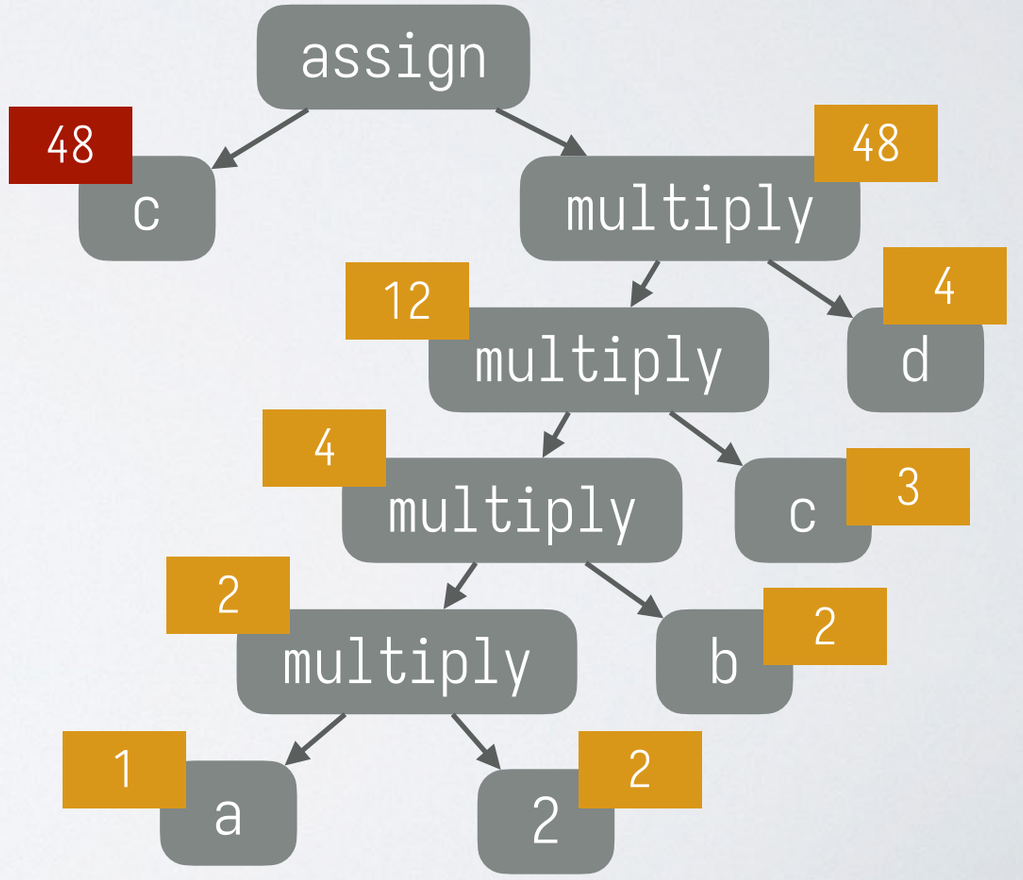




语义分析示例：解释执行

- **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



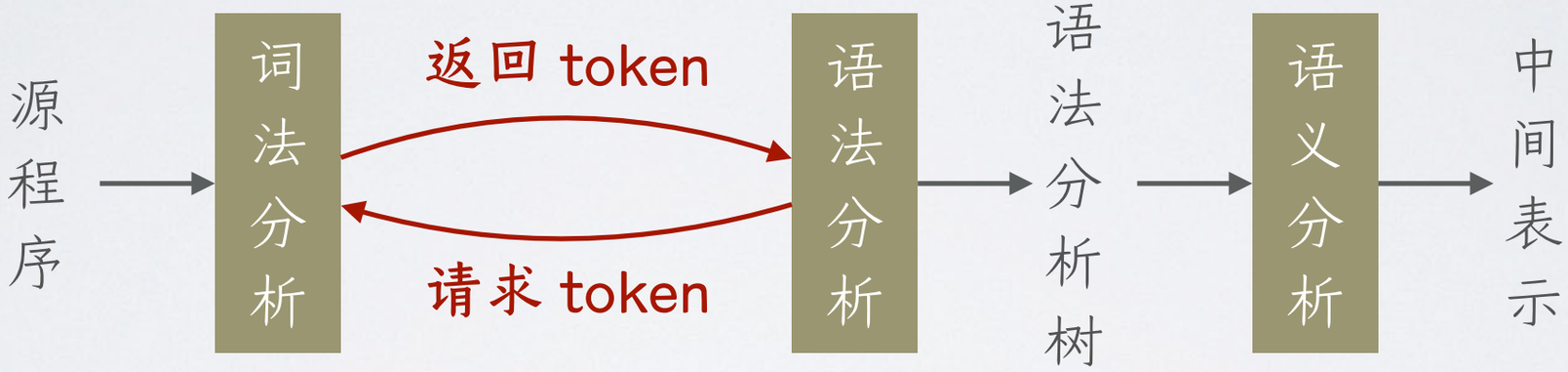


语义分析的工作

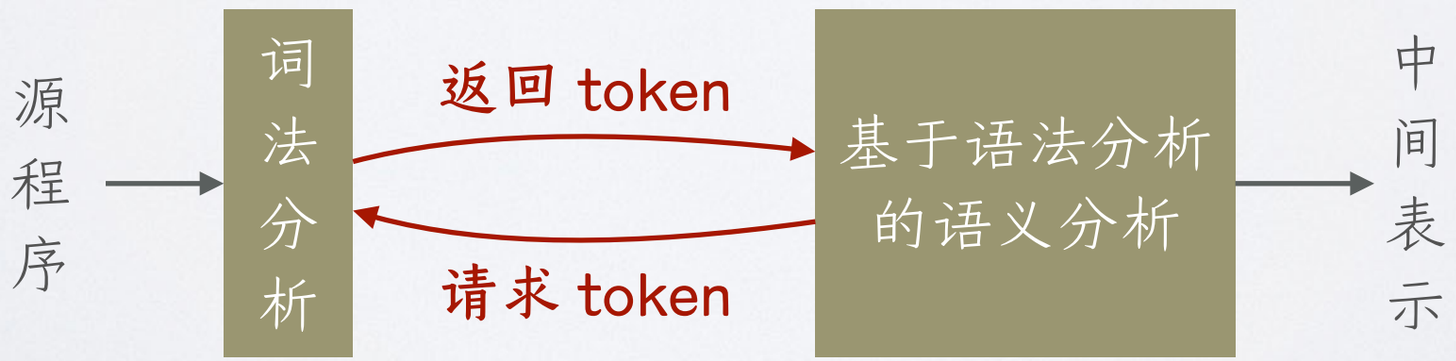
- 为**翻译到目标程序**或**直接进行解释执行**做准备
- 获取程序在语法之外的各种**上下文有关信息**
  - x 中存储了什么种类的值?
  - x 需要多大的存储空间?
  - 如果名字 x 是一个函数，它的参数和返回值是什么?
  - x 中存储的值的生命周期有多久?
  - x 应该被谁、在何时进行空间分配和初始化?
  - ……
- **上下文有关分析(context-sensitive analysis)**
  - 回顾: 语法分析中使用上下文无关文法

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
- 语义分析的自动生成

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	`if`
WHILE	`while`
ID	`[A-Za-z_]([A-Za-z_0-9])*`
NUMBER	`[+-]?([0-9])+`
LPAREN	`(`
NEQ	`!=`

- 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

$$
\begin{aligned}
Expr &::= Expr \texttt{ + } Term \\
&\mid Expr \texttt{ - } Term \\
&\mid Term \\
Term &::= Term \texttt{ * } Factor \\
&\mid Term \texttt{ / } Factor \\
&\mid Factor \\
Factor &::= \texttt{(} Expr \texttt{)} \\
&\mid \texttt{ID} \mid \texttt{INT}
\end{aligned}
$$

$$
\begin{aligned}
Stmt &::= \texttt{\{ } Block \texttt{ \}} \\
&\mid \texttt{ID = } Expr \texttt{ ;} \\
&\mid \texttt{if (} Expr \texttt{)} Stmt \texttt{ else } Stmt \\
&\mid \texttt{while (} BDisj \texttt{)} Stmt \\
&\mid \texttt{return } Expr \texttt{ ;} \\
Block &::= \epsilon \\
&\mid Stmt\ Block
\end{aligned}
$$

- 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 $\alpha \to \beta$ 的左侧长度大于 1，但要求 $|\alpha| \le |\beta|$

- 考虑语言 $\{\mathrm{a}^n \mathrm{b}^n \mathrm{c}^n \mid n \ge 1\}$
  - 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]	$S \to \mathrm{a}BC$
[2]	$S \to \mathrm{a}SBC$
[3]	$CB \to BC$
[4]	$\mathrm{a}B \to \mathrm{ab}$
[5]	$\mathrm{b}B \to \mathrm{bb}$
[6]	$\mathrm{b}C \to \mathrm{bc}$
[7]	$\mathrm{c}C \to \mathrm{cc}$

- **观察：**检查一个串是否形如 $\mathrm{a}^n \mathrm{b}^n \mathrm{c}^n$ 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

$$V_T = \{+, -, *, /, (,), \texttt{INT}\},\ V_N = \{Expr, Term, Factor\}$$

$$
\begin{aligned}
Expr &::= Expr + Term \\
&\mid\ Expr - Term \\
&\mid\ Term \\
Term &::= Term * Factor \\
&\mid\ Term / Factor \\
&\mid\ Factor \\
Factor &::= (Expr) \\
&\mid\ \texttt{INT}
\end{aligned}
$$

用下标区分相同符号的不同结点

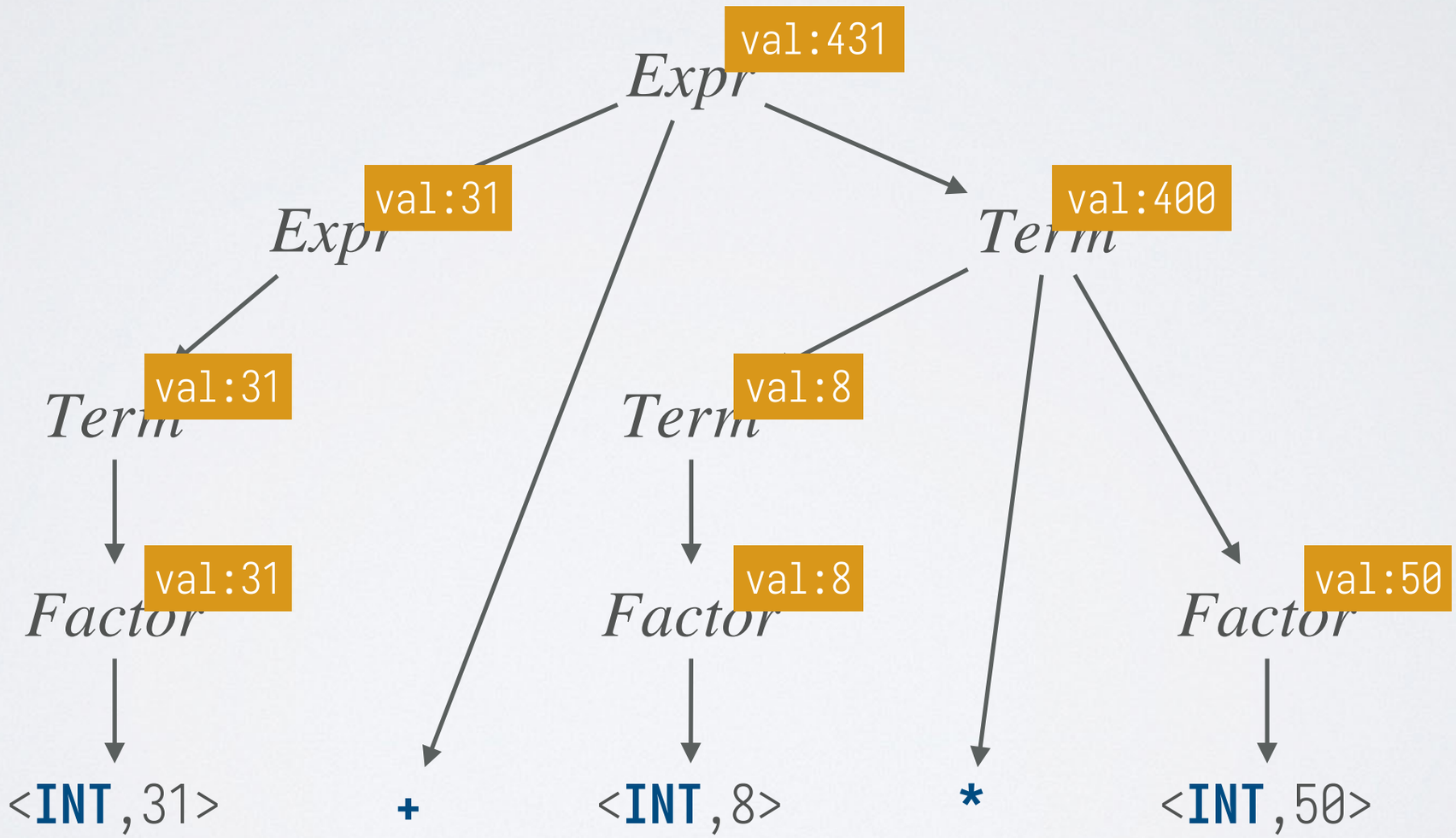
属性 val 记录表达式的求值结果

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

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 $V_T = \{+, -, 0, 1\}$, $V_N = \{Number, Sign, List, Bit\}$

$$
\begin{aligned}
Number &::= Sign\ List \\
Sign &::= + \\
&\mid\ - \\
List &::= Bit \\
&\mid\ List\ Bit \\
Bit &::= 0 \\
&\mid\ 1
\end{aligned}
$$

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 $V_T = \{+, -, 0, 1\}$, $V_N = \{Number, Sign, List, Bit\}$

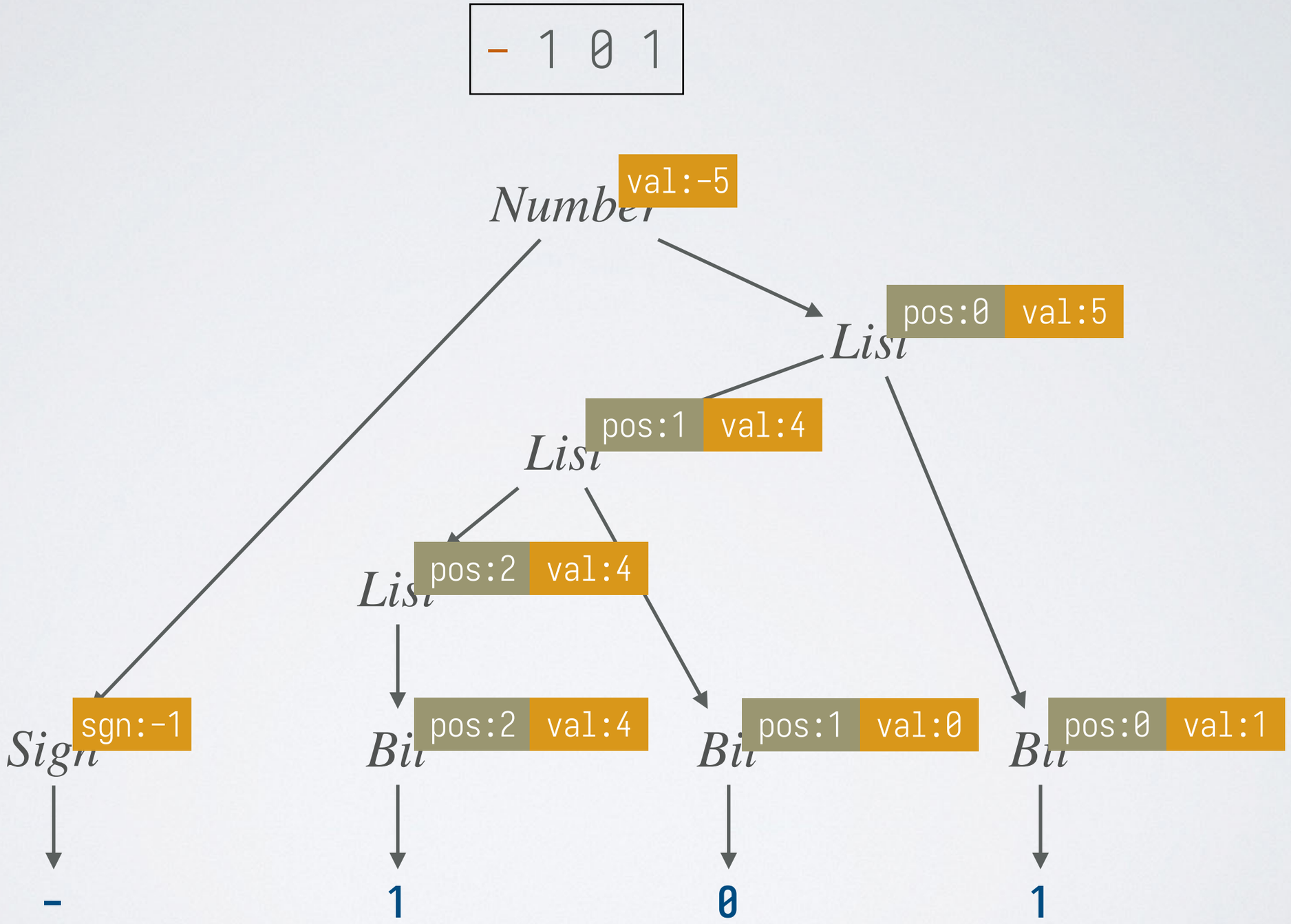
属性 val 记录对应的整数值

属性 sgn 记录正负符号值

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

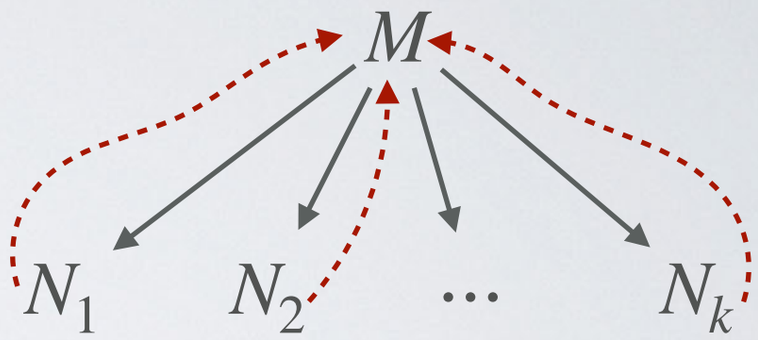
属性 pos 记录二进制位所在位置（最低位的位置为 0）

属性文法示例：有符号二进制数

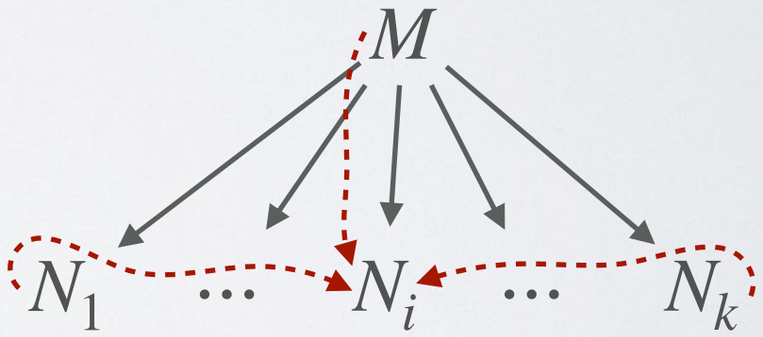


综合属性和继承属性

- **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 $N_1, N_2, \ldots, N_k$ 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 $N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_k$ 的属性来定义



综合属性示例

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

Expr、*Term*、*Factor* 的属性 val 都是综合属性

- 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

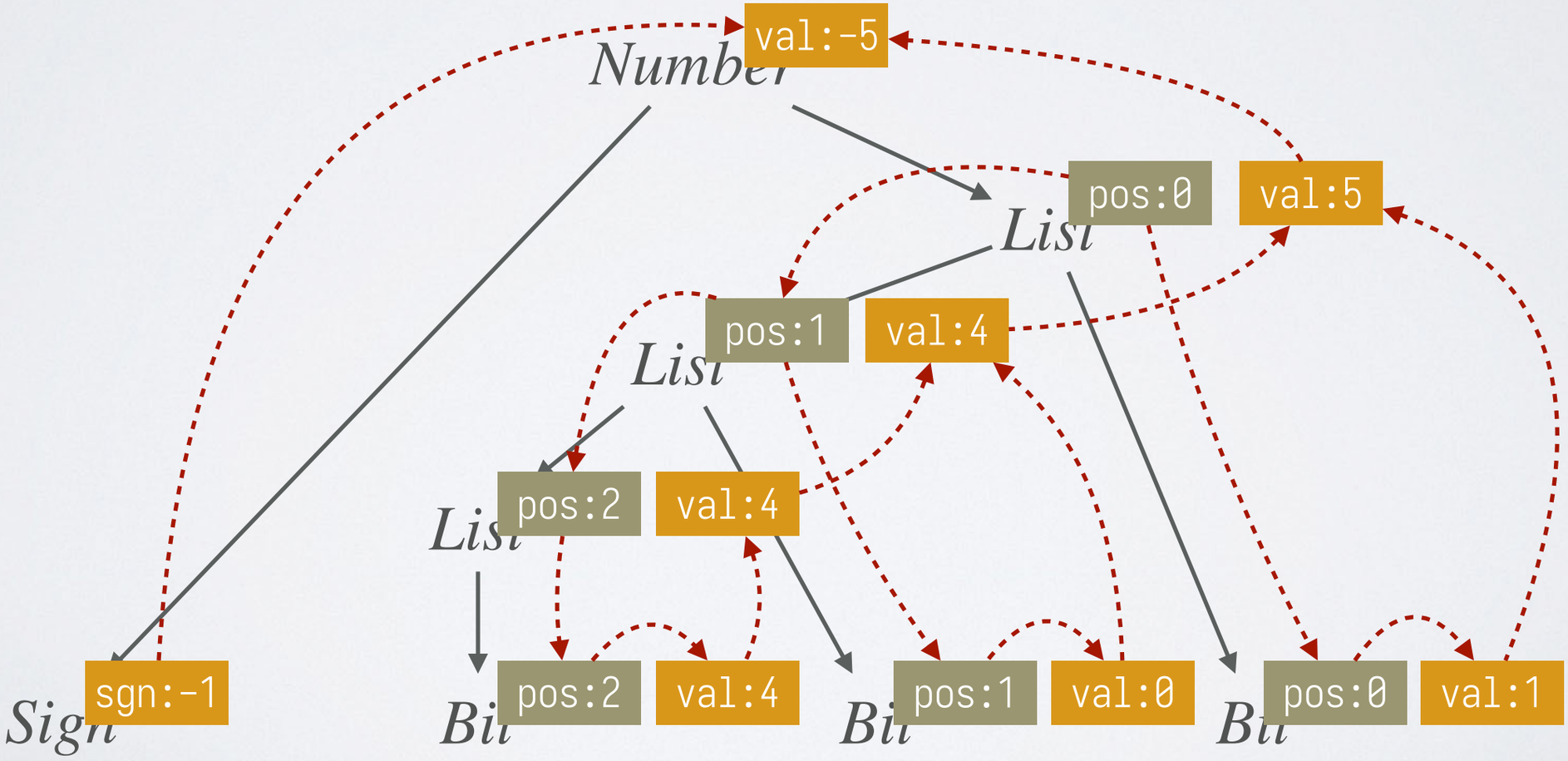
产生规则	属性计算规则
$Number \rightarrow Sign\ List$	$Number.\mathtt{val} = Sign.\mathtt{sgn} \times List.\mathtt{val}$ $List.\mathtt{pos} = 0$
$Sign \rightarrow +$	$Sign.\mathtt{sgn} = 1$
$Sign \rightarrow -$	$Sign.\mathtt{sgn} = -1$
$List \rightarrow Bit$	$List.\mathtt{val} = Bit.\mathtt{val}$ $Bit.\mathtt{pos}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

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 $c \to d$ 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

$$
\begin{aligned}
Loop &::= \texttt{repeat INT do } Stmt \\
Stmt &::= \epsilon \\
&\quad | \quad \texttt{cnt += INT ; } Stmt
\end{aligned}
$$

- 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

$Loop \rightarrow$ repeat INT do $Stmt$	$Loop$.cnt_init $= 0$
	$Loop$.loop_number $=$ INT.intval
	if $Loop$.loop_number > 0 then
	$Stmt$.cnt_init $= Loop$.cnt_init
	$Loop$.cnt_init $= Stmt$.cnt_final
	$Loop$.loop_number $= Loop$.loop_number $- 1$
	else
	$Loop$.cnt_final $= Loop$.cnt_init

$Stmt \rightarrow \epsilon$	$Stmt$.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

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

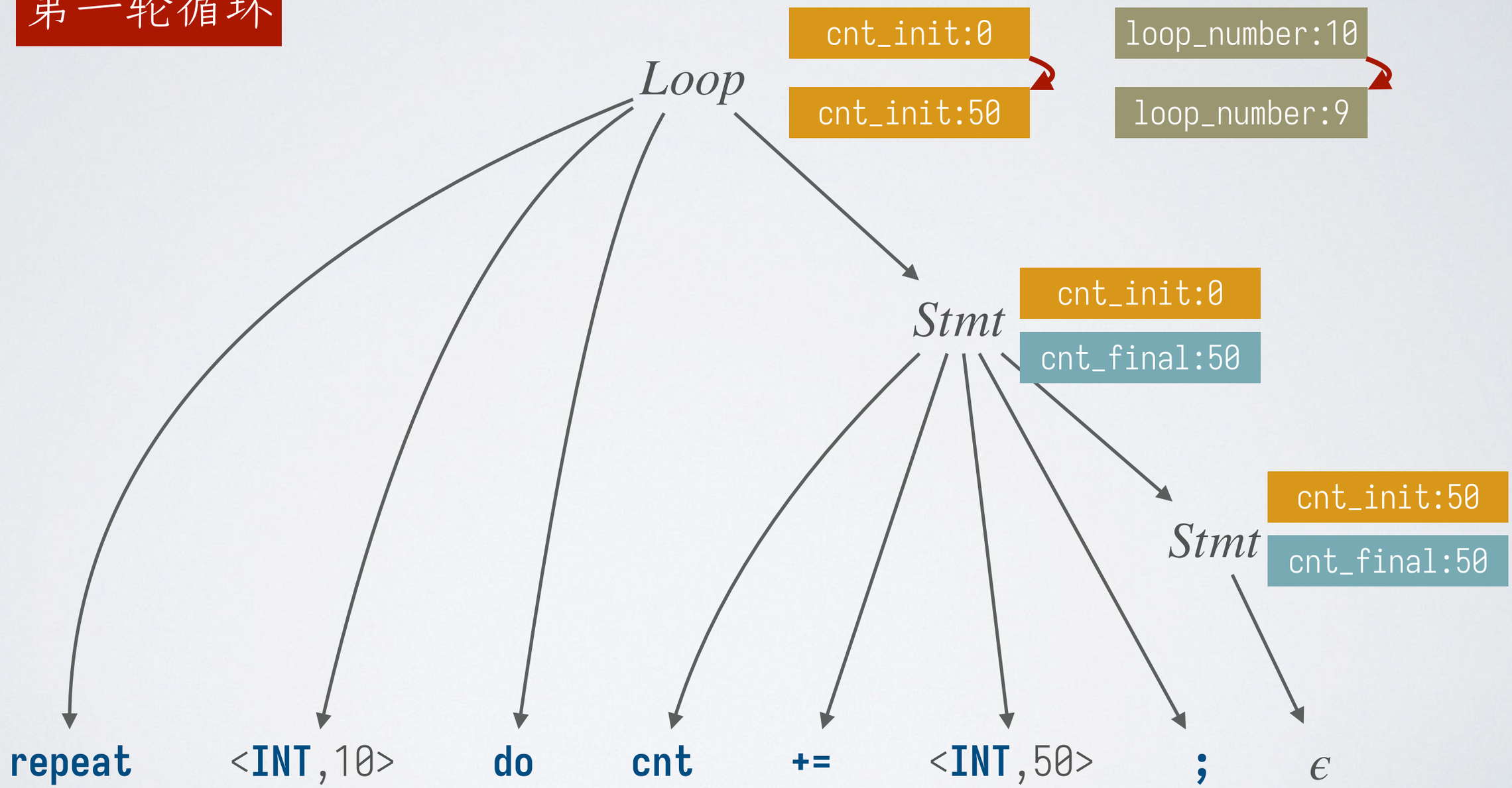
循环体执行结束后，更新 cnt 的值并准备进入下一轮循环

更新循环计数器，完成一轮循环

循环体中的语句依次对 cnt 的值进行更新

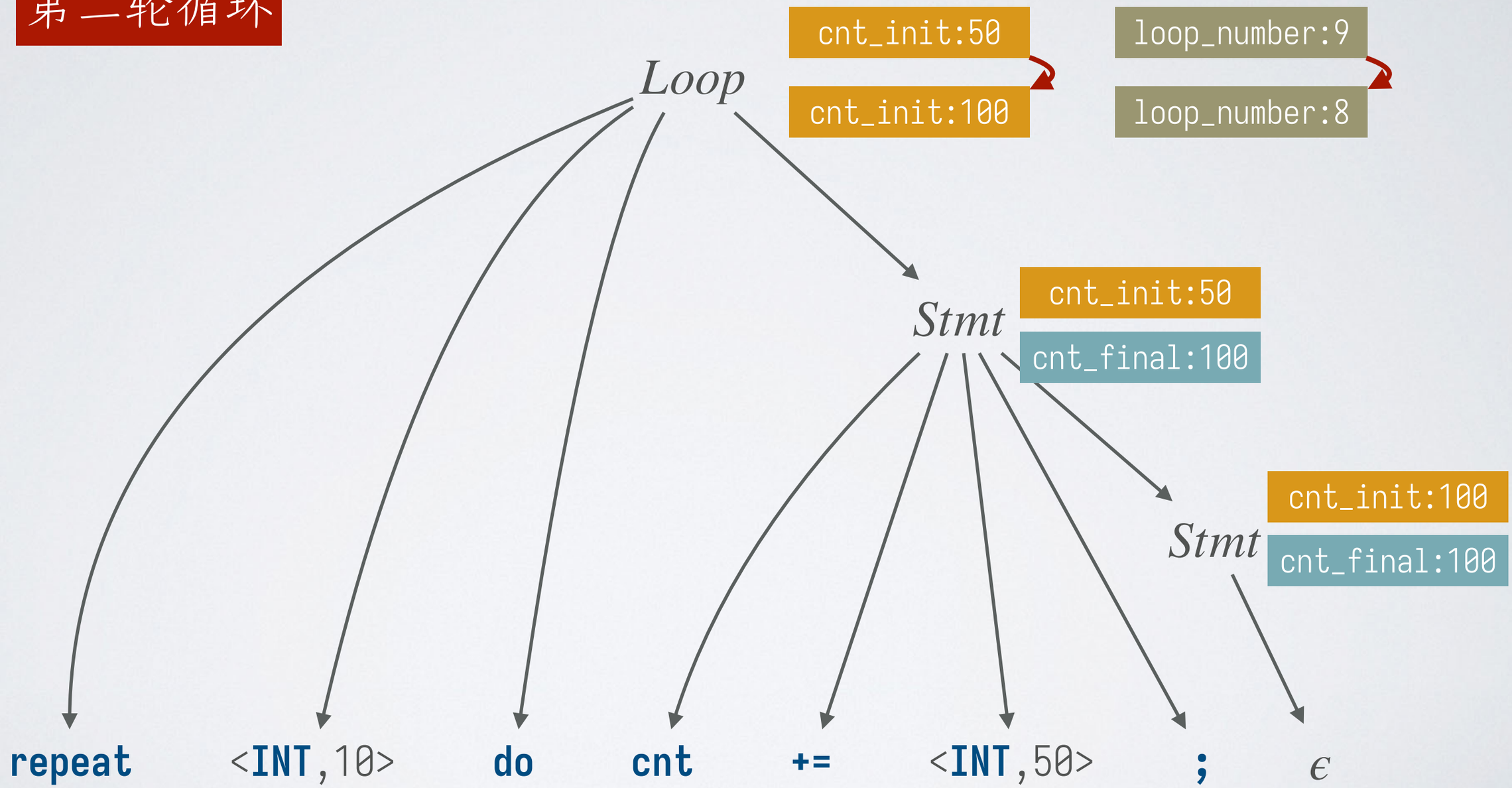
属性计算顺序：有环的情况

第一轮循环



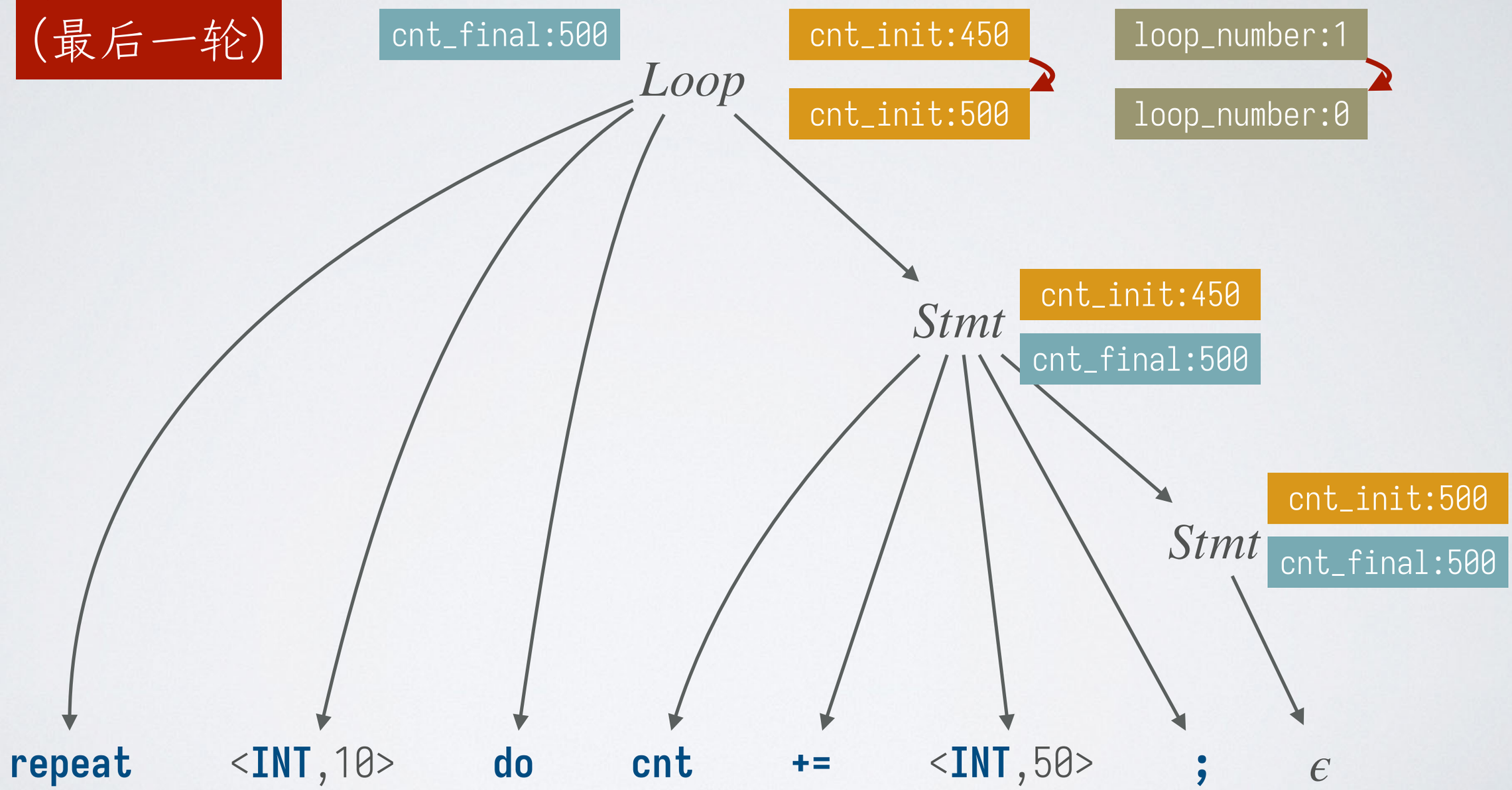
属性计算顺序：有环的情况

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
  - 进而作为可自动生成的语义分析的规约

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
- 容易结合自顶向下和自底向上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 $A \to X_1 X_2 \ldots X_k$，某个 X_i 的继承属性只依赖于 **X_i 的左边 $(X_1, X_2, \ldots, X_{i-1})$ 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
  - 部分 L 属性的文法也可以结合自底向上的语法分析进行实现

L 属性的文法示例

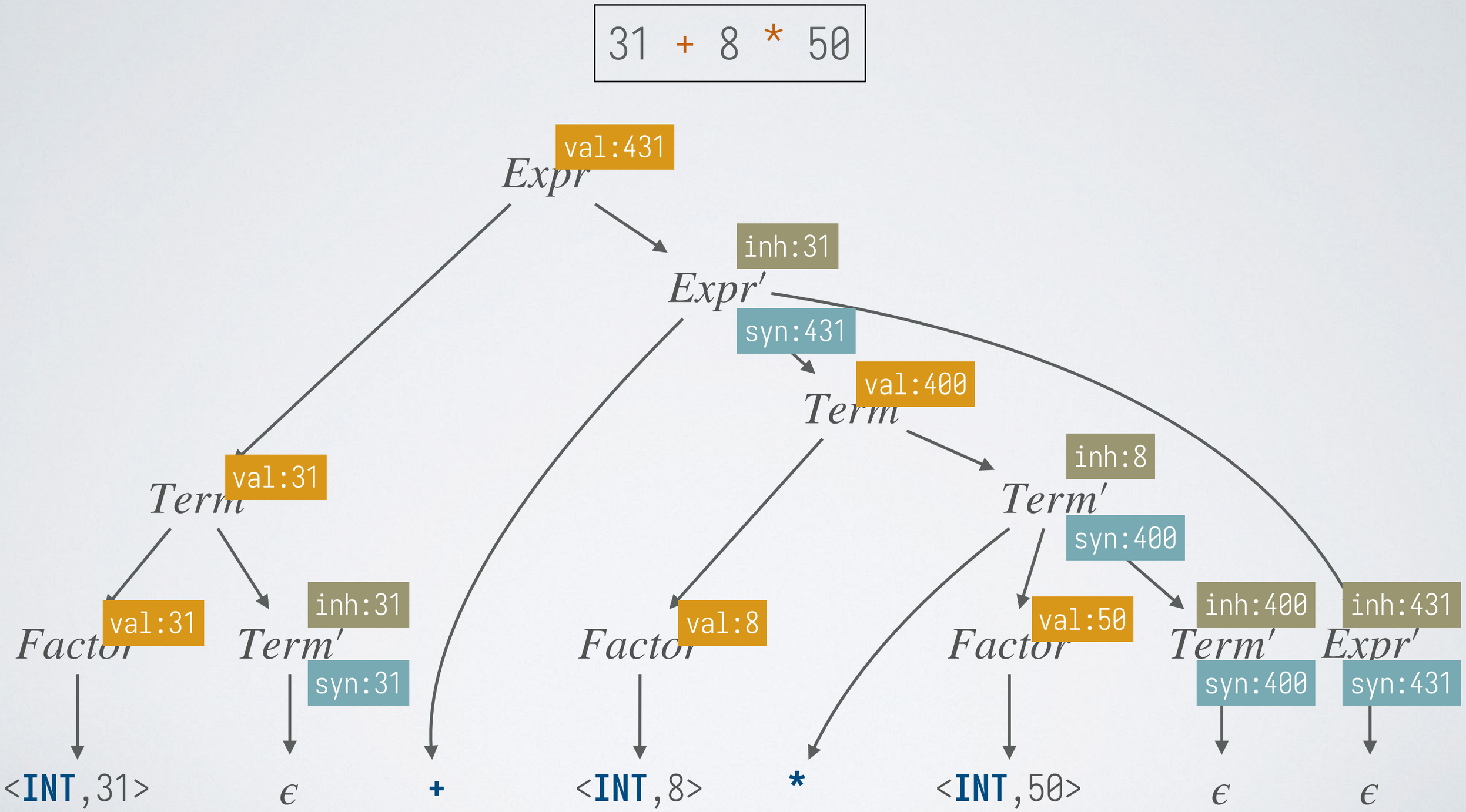
$$
\begin{aligned}
Expr &::= Term\ Expr' \\
Expr' &::= +\ Term\ Expr' \\
&\mid\ -\ Term\ Expr' \\
&\mid\ \epsilon \\
Term &::= Factor\ Term' \\
Term' &::= *\ Factor\ Term' \\
&\mid\ /\ Factor\ Term' \\
&\mid\ \epsilon \\
Factor &::= (\ Expr\ ) \\
&\mid\ \texttt{INT}
\end{aligned}
$$

继承属性 inh 记录左侧已经计算的值

综合属性 syn 记录在 inh 基础上计算的值

产生规则	属性计算规则
$Expr \to Term\ Expr'$	$Expr'.\texttt{inh} = Term.\texttt{val}$ $Expr.\texttt{val} = Expr'.\texttt{syn}$
$Expr' \to +\ Term\ Expr'_1$	$Expr'_1.\texttt{inh} = Expr'.\texttt{inh} + Term.\texttt{val}$ $Expr'.\texttt{syn} = Expr'_1.\texttt{syn}$
$Expr' \to \epsilon$	$Expr'.\texttt{syn} = Expr'.\texttt{inh}$
$Term \to Factor\ Term'$	$Term'.\texttt{inh} = Factor.\texttt{val}$ $Term.\texttt{val} = Term'.\texttt{syn}$
$Term' \to *\ Factor\ Term'_1$	$Term'_1.\texttt{inh} = Term'.\texttt{inh} \times Factor.\texttt{val}$ $Term'.\texttt{syn} = Term'_1.\texttt{syn}$
$Term' \to \epsilon$	$Term'.\texttt{syn} = Term'.\texttt{inh}$
$Factor \to (\ Expr\ )$	$Factor.\texttt{val} = Expr.\texttt{val}$
$Factor \to \texttt{INT}$	$Factor.\texttt{val} = \texttt{INT}.\texttt{intval}$

L 属性的文法示例


31 + 8 * 50
Expr
val:431
inh:31
Expr′
syn:431
Term
val:31
val:400
Term′
inh:8
Factor
val:31
Term′
inh:31
syn:31
Factor
val:8
Factor
val:50
Term′
inh:400
syn:400
Expr′
inh:431
syn:431
<INT,31>
ϵ
+
<INT,8>
*
<INT,50>
ϵ
ϵ

属性文法应用：翻译到中间表示

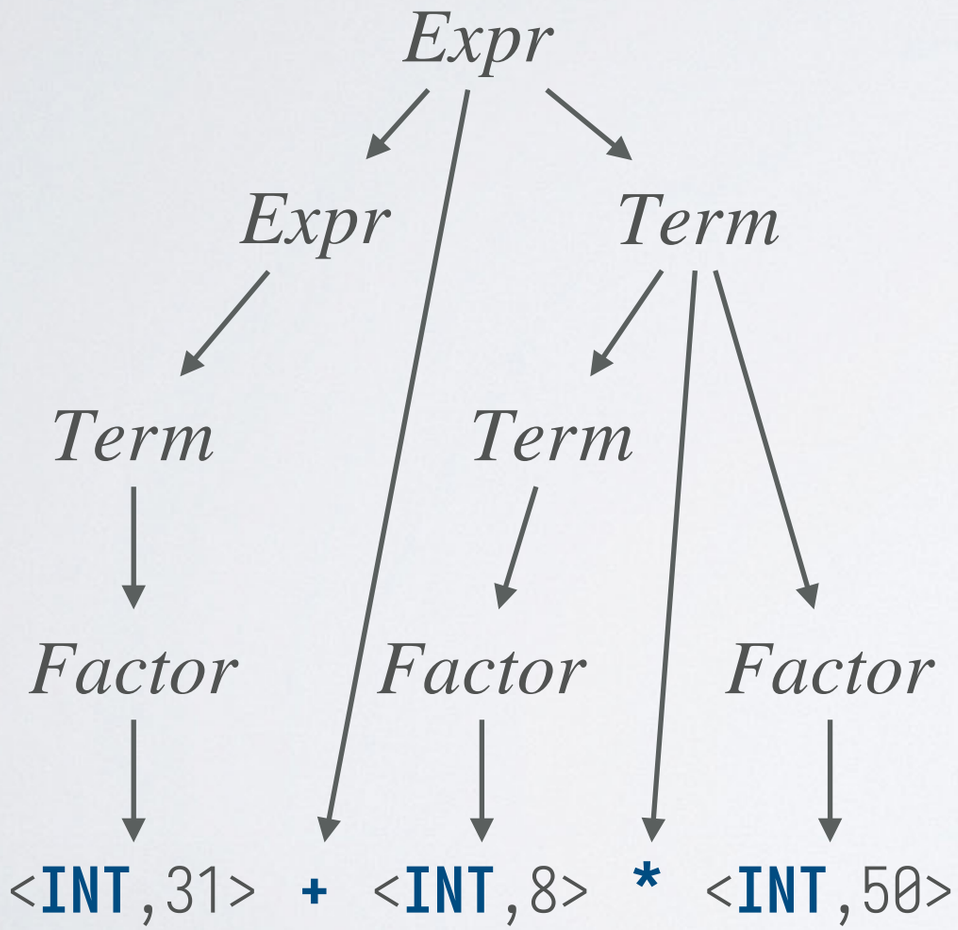
- **抽象语法树(Abstract Syntax Tree，AST)**
- 一种四则运算表达式 AST 的定义：

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

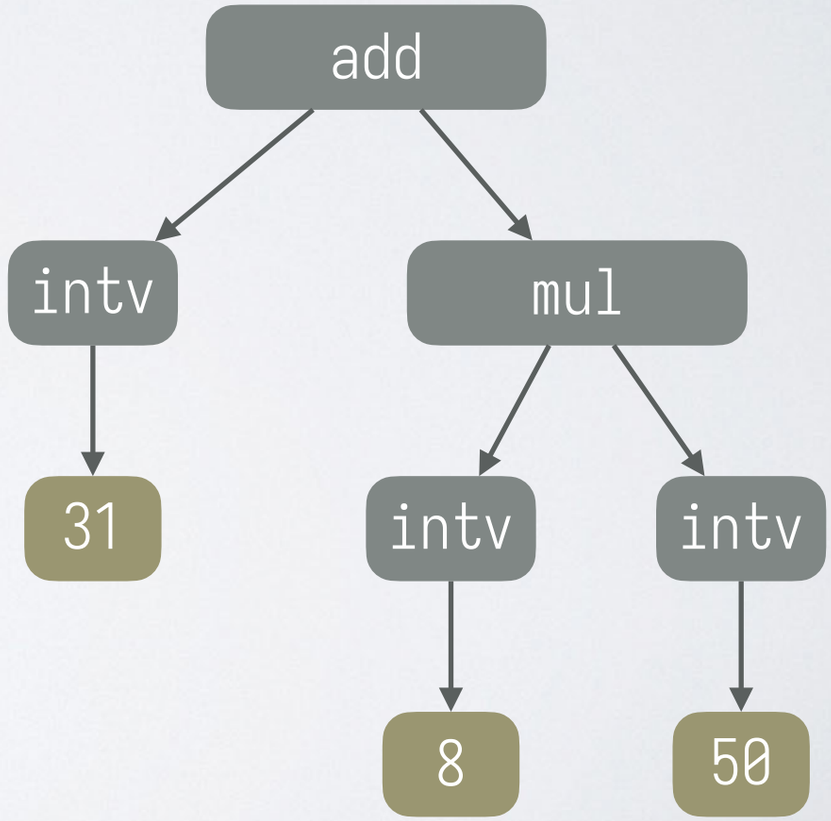
产生规则	属性计算规则
$Expr \rightarrow Expr_1 + Term$	$Expr.\texttt{ast} = \texttt{add}(Expr_1.\texttt{ast}, Term.\texttt{ast})$
$Expr \rightarrow Expr_1 - Term$	$Expr.\texttt{ast} = \texttt{sub}(Expr_1.\texttt{ast}, Term.\texttt{ast})$
$Expr \rightarrow Term$	$Expr.\texttt{ast} = Term.\texttt{ast}$
$Term \rightarrow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

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：$Factor ::= (\ Expr\ ) \mid \texttt{INT} \mid \texttt{FLOAT}$

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

由编程语言规定的类型计算规则

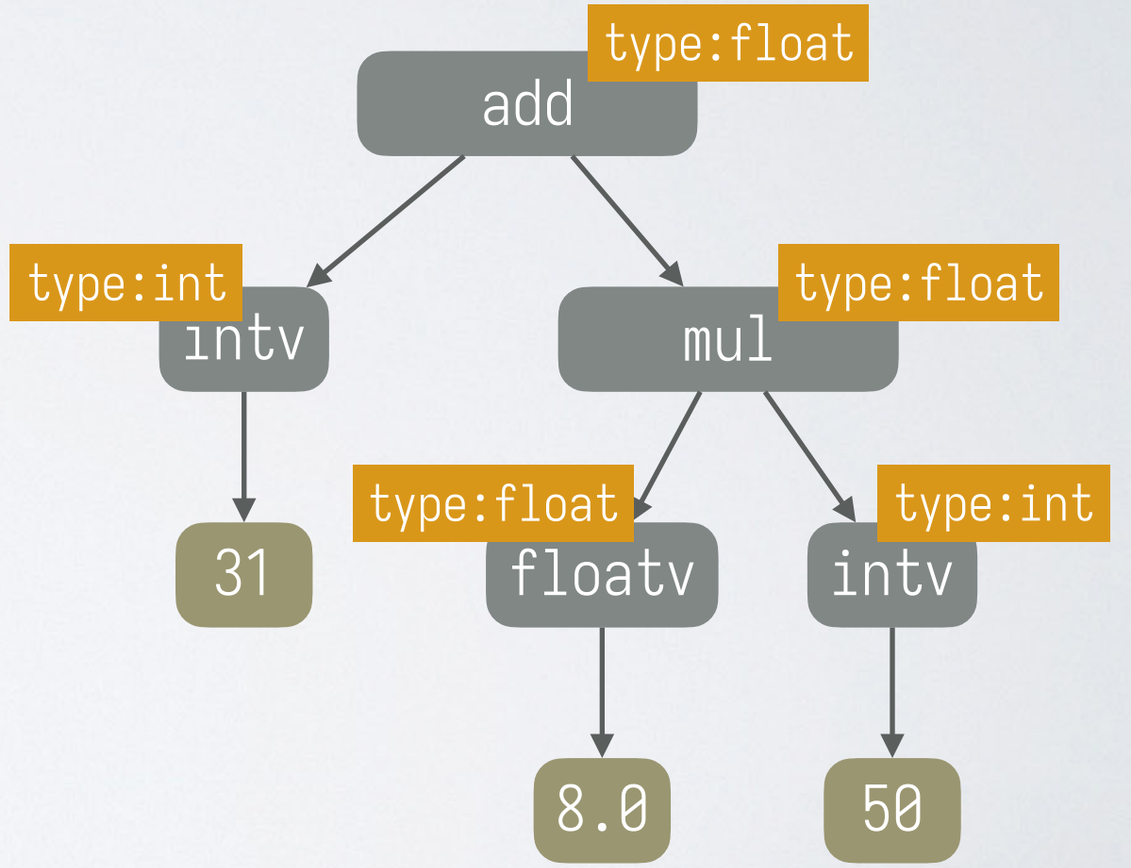
$\mathcal{F}_+$	int	float
int	int	float
float	float	float

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

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

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



属性文法应用：解释执行

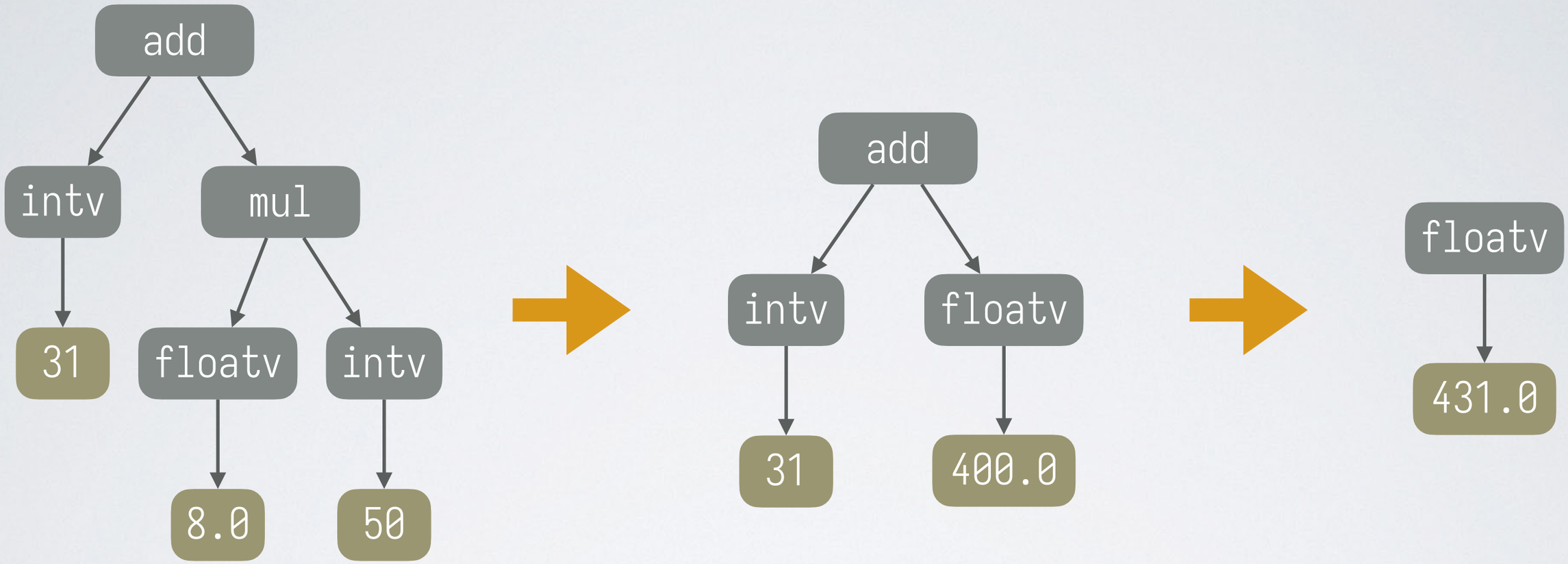
- 前面的简单循环的解释执行是通过存在依赖环的文法实现的
- 也可以通过属性文法定义一个**小步(small-step)运算**
  - 比如计算四则运算表达式(AST)一步运算后的结果表达式(AST)

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

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

$\mathscr{C}_+(E_1, E_2) =$
  if E_1 is not a value yet then
    $\mathtt{add}(E_1.\mathtt{next}, E_2)$
  else if E_2 is not a value yet then
    $\mathtt{add}(E_1, E_2.\mathtt{next})$
  else
    if both E_1 and E_2 are intv then
      $\mathtt{intv}(E_1.\mathtt{next} + E_2.\mathtt{next})$
    else
      $\mathtt{floatv}(E_1.\mathtt{next} + E_2.\mathtt{next})$

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)
  )
```

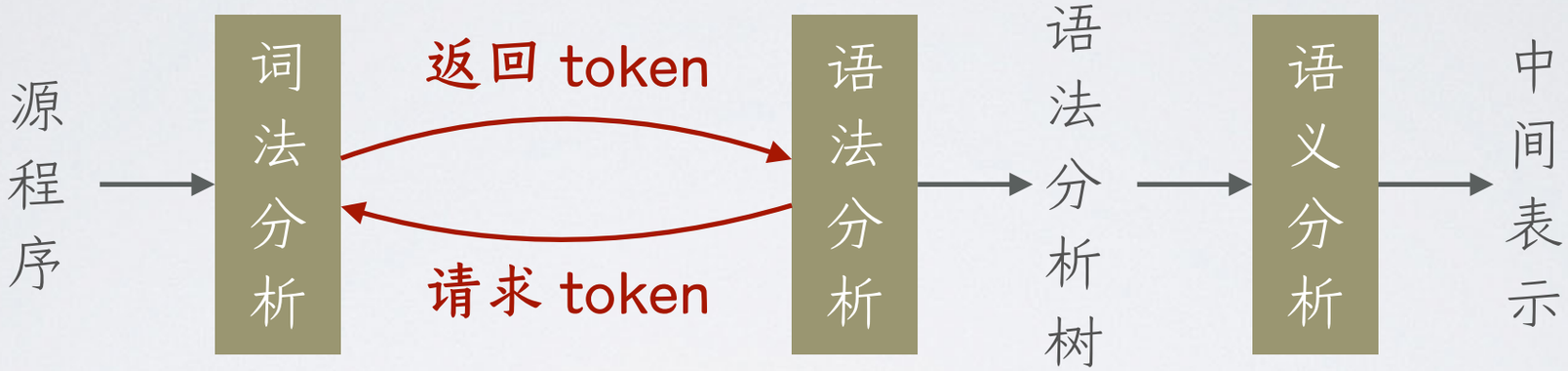
```
floatv(431.0)
```

主要内容

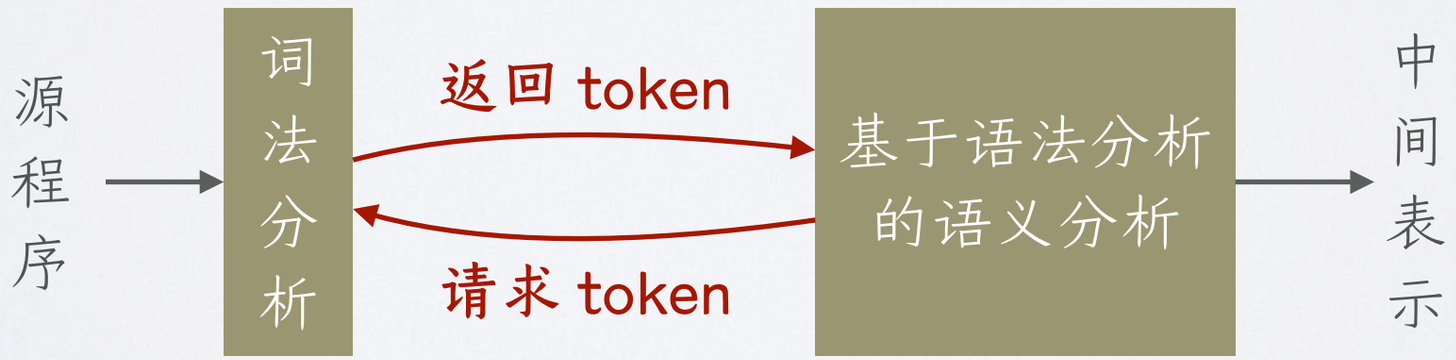
- 语义分析的作用
- 语义分析的规约
- **语义分析的手动实现**
- 语义分析的自动生成

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

$$
\begin{array}{ll}
 & S' \to S\ \texttt{EOF} \\
[1] & S \to \epsilon \\
[2] & S \to [\ S\ ]\ S
\end{array}
$$

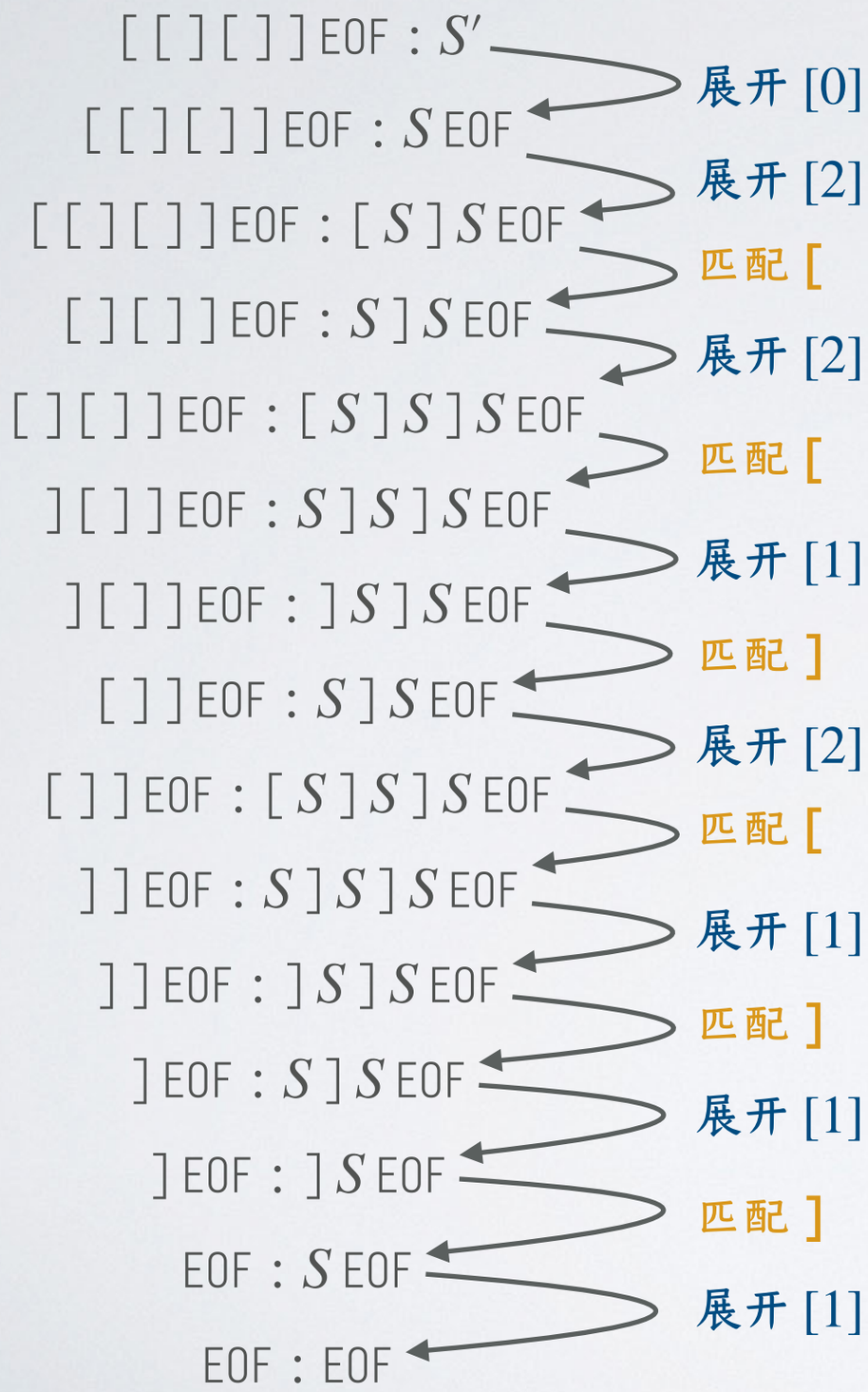
$$\text{FIRST}(S) = \{[, \epsilon\}$$

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

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

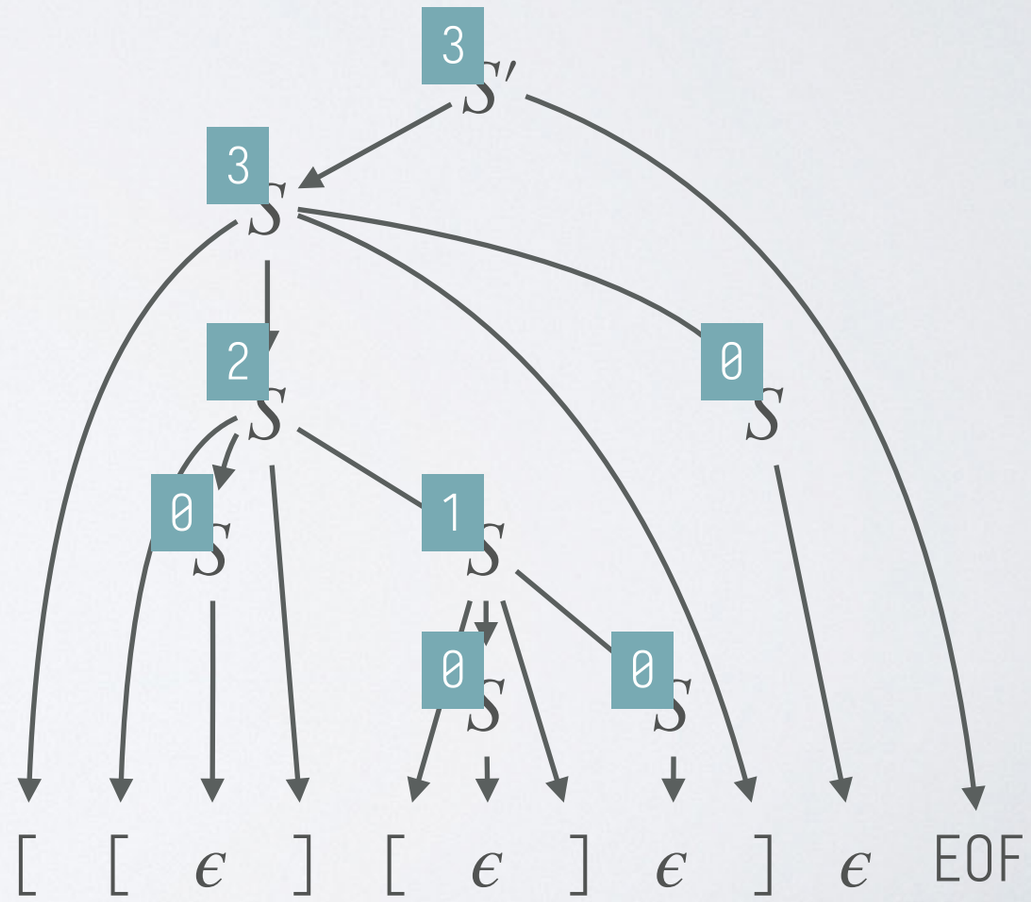
```
S() {
  if (token == LSQUARE) {
    token = next_token();
    if (S()) {
      if (token == RSQUARE) {
        token = next_token();
        if (S()) return true;
      }
    }
  } else if (token == EOF ||
             token == RSQUARE) {
    return true;
  }
  return false;
}
```

自顶向下的语义分析



综合属性 $S'.\text{cnt}$ 和 $S.\text{cnt}$ 记录匹配的括号对数

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



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**

$A ::= \alpha_1 \mid \ldots$

```
A() {
  if (token ∈ FIRST(α1)) {
    按照 α1 中符号依次进行 token 匹配或递归调用;
    按照 A → α1 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] $S' \to S\ \texttt{EOF}$	$S'.\texttt{cnt} = S.\texttt{cnt}$
[1] $S \to \epsilon$	$S.\texttt{cnt} = 0$
[2] $S \to [\ S_1\ ]\ S_2$	$S.\texttt{cnt} = 1 + S_1.\texttt{cnt} + S_2.\texttt{cnt}$

```
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

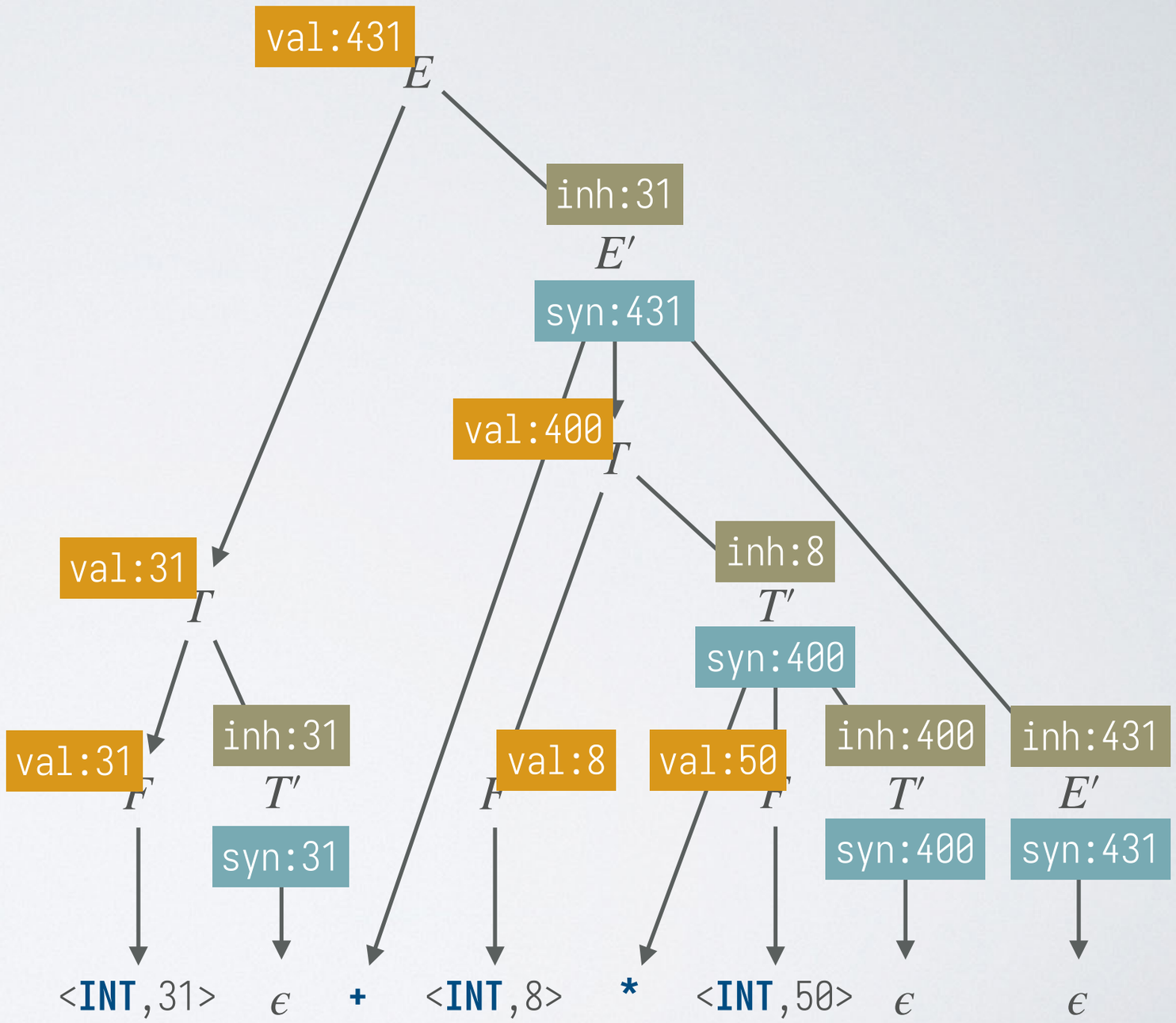
- 回顾：L 属性的文法中既有综合属性也有继承属性
  - 每个继承属性可以依赖左侧结点的属性和 parent 结点的继承属性
- **L 属性的文法的递归下降分析**
  - 每个非终结符号的过程**以其所有的继承属性为参数**
  - 每个非终结符号的过程**返回该符号所有的综合属性**

$A ::= \alpha_1 \mid \ldots$

```
A(A 的继承属性) {
  if (token ∈ FIRST(α1)) {
    按照 α1 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
$E \to T\,E'$	$E'.\mathrm{inh} = T.\mathrm{val}$ $E.\mathrm{val} = E'.\mathrm{syn}$
$E' \to +\,T\,E'_1$	$E'_1.\mathrm{inh} = E'.\mathrm{inh} + T.\mathrm{val}$ $E'.\mathrm{syn} = E'_1.\mathrm{syn}$
$E' \to \epsilon$	$E'.\mathrm{syn} = E'.\mathrm{inh}$
$T \to F\,T'$	$T'.\mathrm{inh} = F.\mathrm{val}$ $T.\mathrm{val} = T'.\mathrm{syn}$
$T' \to *\,F\,T'_1$	$T'_1.\mathrm{inh} = T'.\mathrm{inh} \times F.\mathrm{val}$ $T'.\mathrm{syn} = T'_1.\mathrm{syn}$
$T' \to \epsilon$	$T'.\mathrm{syn} = T'.\mathrm{inh}$
$F \to (\,E\,)$	$F.\mathrm{val} = E.\mathrm{val}$
$F \to \mathrm{INT}$	$F.\mathrm{val} = \mathrm{INT}.\mathrm{intval}$



L 属性的文法的递归下降实现

产生规则	属性计算规则
$E \to T\,E'$	$E'.\text{inh} = T.\text{val}$ $E.\text{val} = E'.\text{syn}$
$E' \to +\,T\,E'_1$	$E'_1.\text{inh} = E'.\text{inh} + T.\text{val}$ $E'.\text{syn} = E'_1.\text{syn}$
$E' \to \epsilon$	$E'.\text{syn} = E'.\text{inh}$
$T \to F\,T'$	$T'.\text{inh} = F.\text{val}$ $T.\text{val} = T'.\text{syn}$
$T' \to *\,F\,T'_1$	$T'_1.\text{inh} = T'.\text{inh} \times F.\text{val}$ $T'.\text{syn} = T'_1.\text{syn}$
$T' \to \epsilon$	$T'.\text{syn} = T'.\text{inh}$
$F \to (\,E\,)$	$F.\text{val} = E.\text{val}$
$F \to \text{INT}$	$F.\text{val} = \text{INT}.\text{intval}$

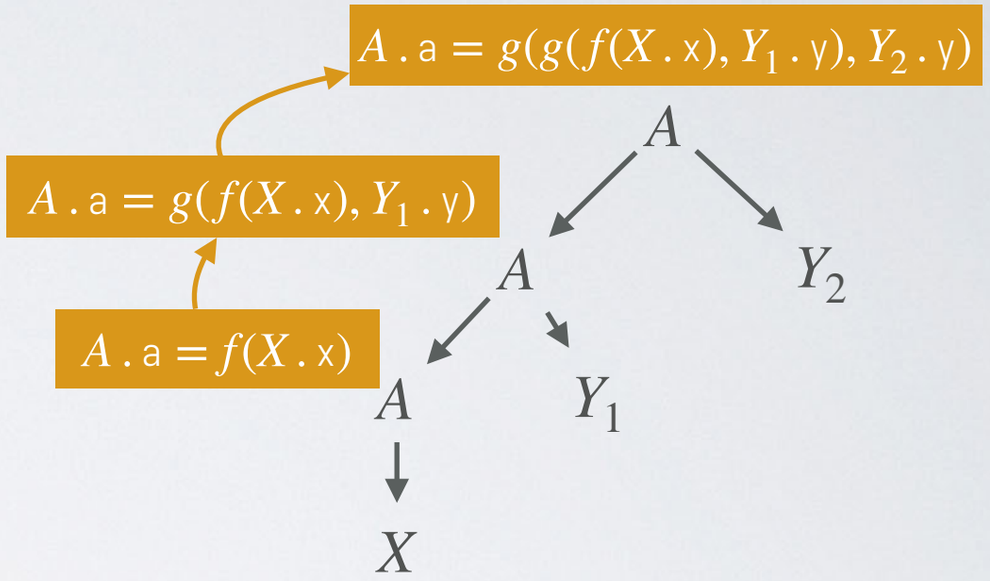
```
E() {
  if (token == LPAREN || token == INT) {
    T_val = T();
    E'_syn = E'(T_val);
    return E'_syn;
  }
  error();
}
```

```
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();
}
```

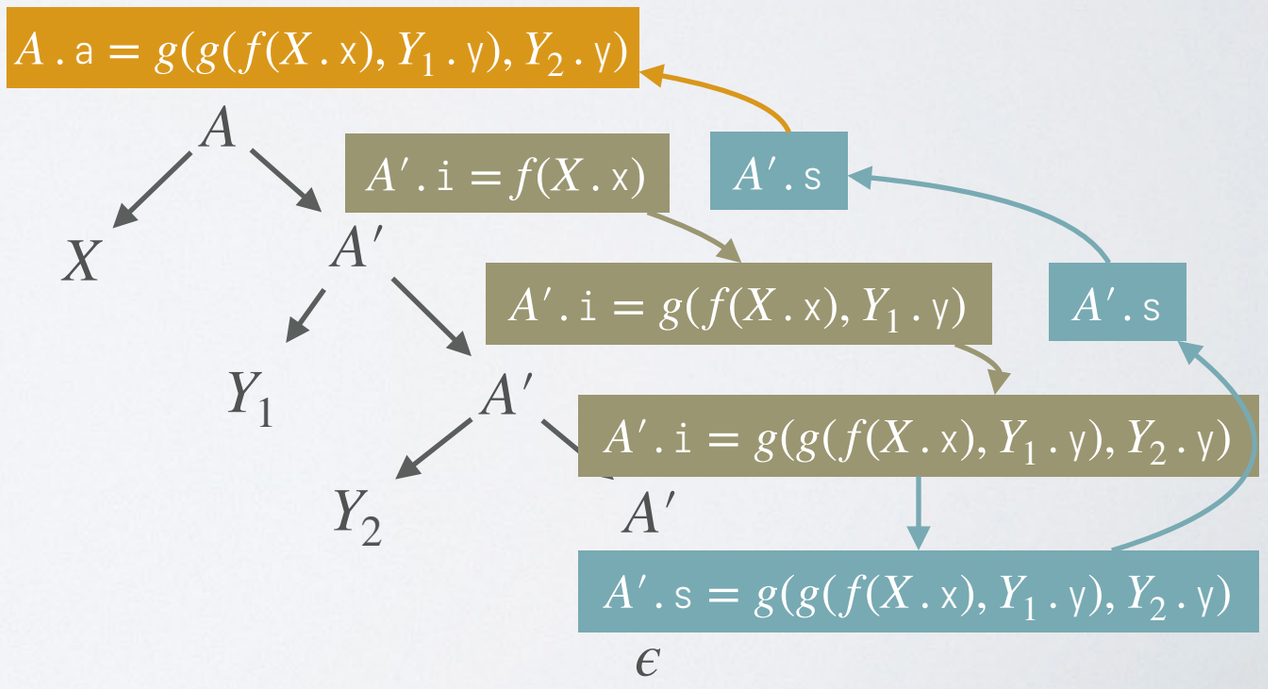
带左递归的 S 属性的文法

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



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



回顾：自底向上的语法分析

- **移进-归约分析**
  - 用一个分析栈记录移进的终结符号和完成归约的非终结符号
  - 根据分析栈顶的模式和向前看符号确定进行移进还是归约

```
top = 0;
token = next_token();
while (true) {
  if (top == 0) {
    switch (token) {
      case LSQUARE:
        top++; stack[top] = LSQUARE;
        token = next_token(); break;
      case RSQUARE:
        return false;
      case EOF:
        top++; stack[top] = S; break;
    }
  } else if (top == 1 && stack[top] == S) {
    switch (token) {
      case LSQUARE: return false;
      case RSQUARE: return false;
      case EOF: return true;
    }
  } else if ......
}
```

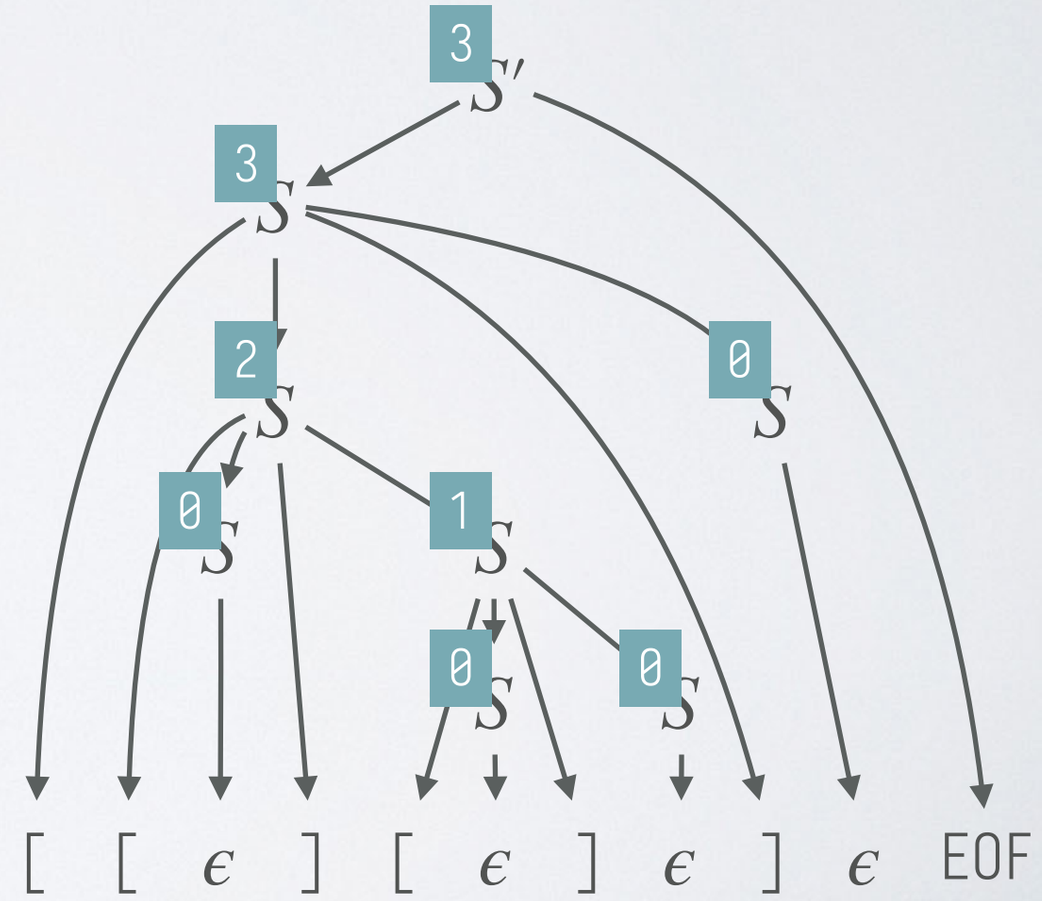
	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]S$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

自底向上的语义分析

综合属性 $S'.\mathrm{cnt}$ 和 $S.\mathrm{cnt}$ 记录匹配的括号对数

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



S 属性的文法的移进-归约实现

- 回顾：S 属性的文法中只有综合属性
  - 信息的流向自底向上，与语法分析的方向一致
- **S 属性的文法的移进-归约分析**
  - 在分析栈中记录每个非终结符号的综合属性
  - 在进行归约时，计算被归约符号的综合属性

```
.....
} else if (根据分析栈顶和向前看要按照 A → B1 B2 ... Bk 进行归约) {
  按照栈顶 k 个符号及其记录的综合属性计算 A 的综合属性;
  top = top - k + 1; stack[top] = A;
  在 stack[top] 的位置上记录之前算出的综合属性;
} else ......
```

S 属性的文法的移进-归约实现

```
top = 0;
token = next_token();
while (true) {
  if (top == 0) {
    switch (token) {
      case LSQUARE:
        top++; stack[top] = LSQUARE;
        token = next_token(); break;
      case RSQUARE:
        return false;
      case EOF:
        top++; stack[top] = S;
        stack[top].cnt = 0; break;
    }
  } else if (top == 1 && stack[top] == S) {
    switch (token) {
      case LSQUARE: return false;
      case RSQUARE: return false;
      case EOF: return true;
    }
  } else if ......
}
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[\,S$	错误	移进	错误
$\beta[\,S\,]$	移进	归约 [1]	归约 [1]
$\beta[\,S\,]\,S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

```
.......
} else if (top >= 1 && stack[top] == LSQUARE) {
  switch (token) {
    case LSQUARE:
      top++; stack[top] = LSQUARE;
      token = next_token(); break;
    case RSQUARE:
      top++; stack[top] = S;
      stack[top].cnt = 0; break;
    case EOF:
      return false;
  }
} else if (top >= 2 && stack[top] == S &&
           stack[top - 1] == LSQUARE) {
  switch (token) {
    case LSQUARE: return false;
    case RSQUARE:
      top++; stack[top] = RSQUARE;
      token = next_token(); break;
    case EOF: return false;
  }
} else if ......
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

```
.......
} else if (top >= 3 &&
           stack[top] == RSQUARE &&
           stack[top - 1] == S &&
           stack[top - 2] == LSQUARE) {
  switch (token) {
    case LSQUARE:
      top++; stack[top] = LSQUARE;
      token = next_token(); break;
    case RSQUARE:
      top++; stack[top] = S;
      stack[top].cnt = 0; break;
    case EOF:
      top++; stack[top] = S;
      stack[top].cnt = 0; break;
  }
} else if ......
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

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

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[S$	错误	移进	错误
$\beta[S]$	移进	归约 [1]	归约 [1]
$\beta[S]S$	错误	归约 [2]	归约 [2]

S 属性的文法的移进-归约实现

```
  ......
} else if (top >= 4 && stack[top] == S &&
           stack[top - 1] == RSQUARE &&
           stack[top - 2] == S &&
           stack[top - 3] == LSQUARE) {
  switch (token) {
    case LSQUARE:
      return false;
    case RSQUARE:
      tmp = 1 + stack[top - 2].cnt +
            stack[top].cnt;
      top = top - 3; stack[top] = S;
      stack[top].cnt = tmp; break;
    case EOF:
      tmp = 1 + stack[top - 2].cnt +
            stack[top].cnt;
      top = top - 3; stack[top] = S;
      stack[top].cnt = tmp; break;
    }
  } else return false;
}
```

	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

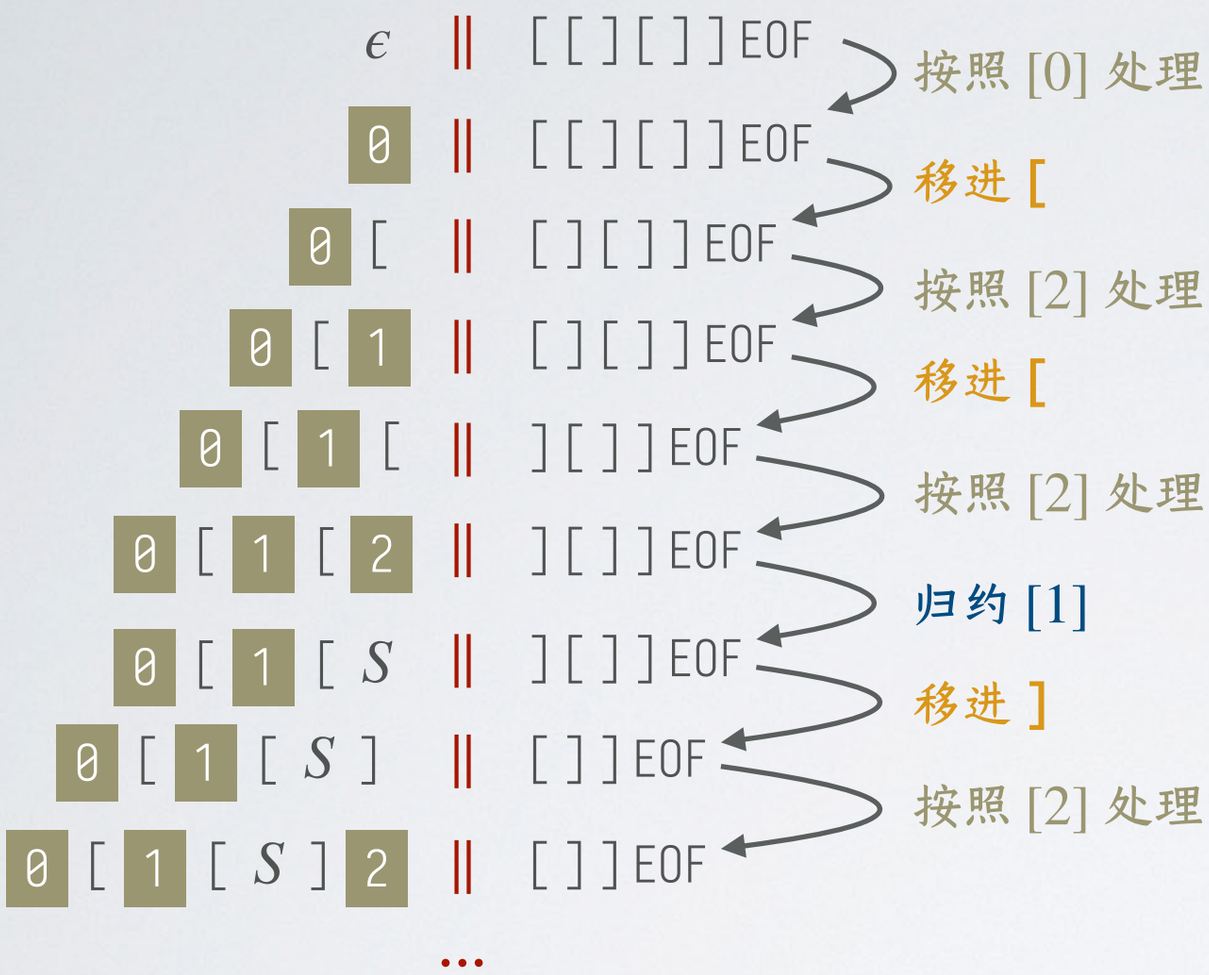
产生规则	属性计算规则
[0] $S' \to S\,\mathrm{EOF}$	$S'.\mathrm{cnt} = S.\mathrm{cnt}$
[1] $S \to \epsilon$	$S.\mathrm{cnt} = 0$
[2] $S \to [\,S_1\,]\,S_2$	$S.\mathrm{cnt} = 1 + S_1.\mathrm{cnt} + S_2.\mathrm{cnt}$

分析栈 \ 向前看	[	]	EOF
ϵ	移进	错误	归约 [1]
S	错误	错误	接受
$\beta[$	移进	归约 [1]	错误
$\beta[\,S$	错误	移进	错误
$\beta[\,S\,]$	移进	归约 [1]	归约 [1]
$\beta[\,S\,]\,S$	错误	归约 [2]	归约 [2]

L 属性的文法的移进-归约实现

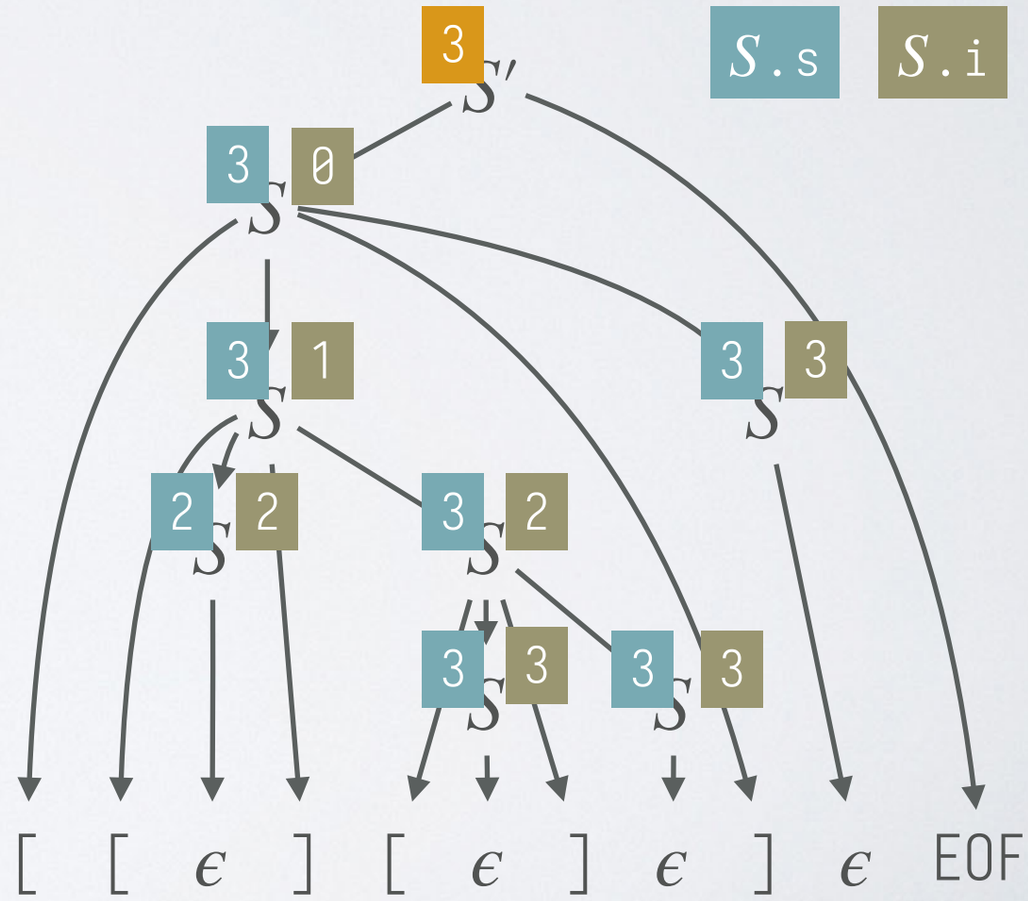
- 回顾：L 属性的文法中既有综合属性也有继承属性
  - 每个继承属性可以依赖左侧结点的属性和 parent 结点的继承属性
  - **问题：继承属性的信息流向可以自顶向下，与语法分析的方向不一致**
- **如果文法本身是 LL 的呢？**
  - 从左往右处理 token 时，根据向前看符号就能确定展开规则
  - 自底向上分析在移进符号时，有足够信息获得这个展开规则
- 以 LL 文法为基础的 L 属性的文法可以通过移进-归约实现
  - **合理安排分析栈**，需要继承属性时可以在分析栈的特定位置找到

自底向上的带继承属性的语义分析



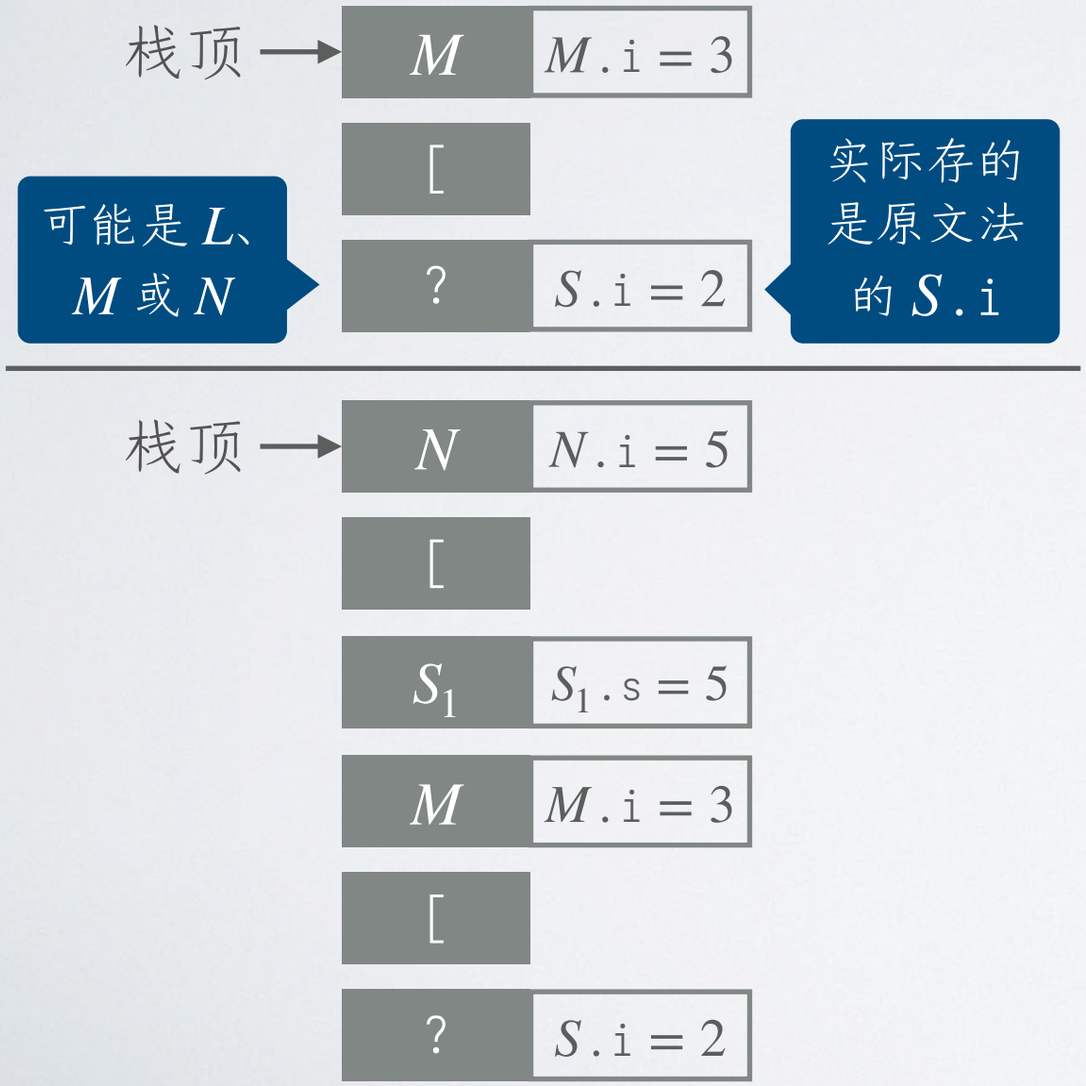
产生规则	属性计算规则
[0] $S' \to S$ EOF	$S.\mathrm{i} = 0$ $S'.\mathrm{s} = S.\mathrm{s}$
[1] $S \to \epsilon$	$S.\mathrm{s} = S.\mathrm{i}$
[2] $S \to [\ S_1\ ]\ S_2$	$S_1.\mathrm{i} = S.\mathrm{i} + 1$　　$S_2.\mathrm{i} = S_1.\mathrm{s}$ $S.\mathrm{s} = S_2.\mathrm{s}$

即将展开非终结符号 S 时，在分析栈中(将来进行归约的位置的下方)存储 S 的继承属性



L 属性的文法的移进-归约实现

- 安排分析栈使得在**归约位置减 1 处**总能找到需要的继承属性
- 技巧：引入**标记**非终结符号(如下面的 L、M、N)



产生规则	属性计算规则
$S' \to L\,S\,\mathrm{EOF}$	$S'.\mathrm{s} = S.\mathrm{s}$
$S \to \epsilon$	$S.\mathrm{s} =$ 栈中归约位置减 1 处$.\mathrm{i}$
$S \to [\,M\,S_1\,]\,N\,S_2$	$S.\mathrm{s} = S_2.\mathrm{s}$
$L \to \epsilon$	$L.\mathrm{i} = 0$
$M \to \epsilon$	$M.\mathrm{i} =$ 栈中归约位置减 2 处$.\mathrm{i} + 1$
$N \to \epsilon$	$N.\mathrm{i} =$ 栈中归约位置减 2 处$.\mathrm{s}$

L 属性的文法的移进-归约实现

- 在 LL 文法中插入这些标记非终结符号(如下面的 L、M、N)后，得到的文法是一个 LR 文法，可以用移进-归约实现

```
top++; stack[top] = S;
stack[top].s = stack[top - 1].i;
```

```
top++; stack[top] = M;
stack[top].i = stack[top - 2].i + 1;
```

```
top++; stack[top] = N;
stack[top].i = stack[top - 2].s;
```

产生规则	属性计算规则
$S' \to L\,S\,\text{EOF}$	$S'.\text{s} = S.\text{s}$
$S \to \epsilon$	$S.\text{s} =$ 栈中归约位置减 1 处$.\text{i}$
$S \to [\,M\,S_1\,]\,N\,S_2$	$S.\text{s} = S_2.\text{s}$
$L \to \epsilon$	$L.\text{i} = 0$
$M \to \epsilon$	$M.\text{i} =$ 栈中归约位置减 2 处$.\text{i} + 1$
$N \to \epsilon$	$N.\text{i} =$ 栈中归约位置减 2 处$.\text{s}$

- 注意：严格来说这种属性计算规则并不规范，但可以纳入后面会讲的语法制导的翻译方案中

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
- **语义分析的自动生成**

从规约到实现的转换

- **语义分析的规约：**属性文法
  - 也称为语法制导定义(Syntax-Directed Definition, SDD)
- **语义分析的实现：**
  - 独立于语法分析实现：在语法分析树上进行遍历
  - 基于语法分析进行实现：S 属性的文法、L 属性的文法
- **语义分析的自动生成：**
  - 自动化 S 或 L 属性的文法到实现的转换
  - 基于表驱动的语法分析进行实现
  - **问题：**属性文法中给出的属性计算规则不一定与实际计算顺序一致

语法制导的翻译方案

- Syntax-Directed Translation，SDT

- 与属性文法相比，SDT 把属性计算规则改写为计算属性值的**程序片段**，并用花括号 { } 括起来，插入到产生规则**右侧**的任何合适的位置上

- 本质是一种对**语法分析**和**语义动作**交错的表示方法
  - 理解为深度优先遍历语法分析树时，何时执行语义动作

- 与属性文法相比，SDT 明确了属性值的计算顺序

SDT 示例：四则运算表达式

- **后缀翻译方案**：每个语义动作都在产生规则的最后
- 对于 S 属性的文法，SDT 看起来跟属性文法差不多

产生规则	语义动作
$Expr \rightarrow Expr_1 + Term$	{ $Expr$.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 * $Factor$.val; }
$Term \rightarrow Term_1 / Factor$	{ $Term$.val = $Term_1$.val / $Factor$.val; }
$Term \rightarrow Factor$	{ $Term$.val = $Factor$.val; }
$Factor \rightarrow (Expr)$	{ $Factor$.val = $Expr$.val; }
$Factor \rightarrow$ INT	{ $Factor$.val = INT.intval; }

SDT 示例：四则运算表达式

- 语义动作可以出现在产生规则中间，比如 $B \to X \{ a \} Y$
- 对于 L 属性的文法，SDT 在中间插入的动作可以操作继承属性

产生规则	语义动作
$Expr \to Term$ $Expr'$	$\{ Expr'.\texttt{inh} = Term.\texttt{val}; \}$ $\{ Expr.\texttt{val} = Expr'.\texttt{syn}; \}$
$Expr' \to +\ Term$ $Expr'_1$	$\{ Expr'_1.\texttt{inh} = Expr'.\texttt{inh} + Term.\texttt{val}; \}$ $\{ Expr'.\texttt{syn} = Expr'_1.\texttt{syn}; \}$
$Expr' \to \epsilon$	$\{ Expr'.\texttt{syn} = Expr'.\texttt{inh}; \}$
$Term \to Factor$ $Term'$	$\{ Term'.\texttt{inh} = Factor.\texttt{val}; \}$ $\{ Term.\texttt{val} = Term'.\texttt{syn}; \}$
$Term' \to *\ Factor$ $Term'_1$	$\{ Term'_1.\texttt{inh} = Term'.\texttt{inh} * Factor.\texttt{val}; \}$ $\{ Term'.\texttt{syn} = Term'_1.\texttt{syn}; \}$
$Term' \to \epsilon$	$\{ Term'.\texttt{syn} = Term'.\texttt{inh}; \}$
$Factor \to (\ Expr\ )$	$\{ Factor.\texttt{val} = Expr.\texttt{val}; \}$
$Factor \to \texttt{INT}$	$\{ Factor.\texttt{val} = \texttt{INT}.\texttt{intval}; \}$

也可以写在一行上：

$E \to T \{ E'.\texttt{inh} = T.\texttt{val}; \} E' \{ E.\texttt{val} = E'.\texttt{syn}; \}$

回顾：按照深度优先遍历语法分析树的顺序插入语义动作

从结点 E 遍历其孩子结点，先访问结点 T，结束后用综合属性 $T.\texttt{val}$ 设置下一个孩子结点的继承属性 $E'.\texttt{inh}$，然后访问结点 E'，结束后把综合属性 $E'.\texttt{syn}$ 作为结点 E 的综合属性返回

SDT 示例：四则运算表达式

- 语义动作也可以不计算属性，而是直接产生**副作用**
- 下面的 SDT 把中缀表达式转换成等价的前缀表达式输出
  - 比如输入 9+5*2，输出 + 9 * 5 2

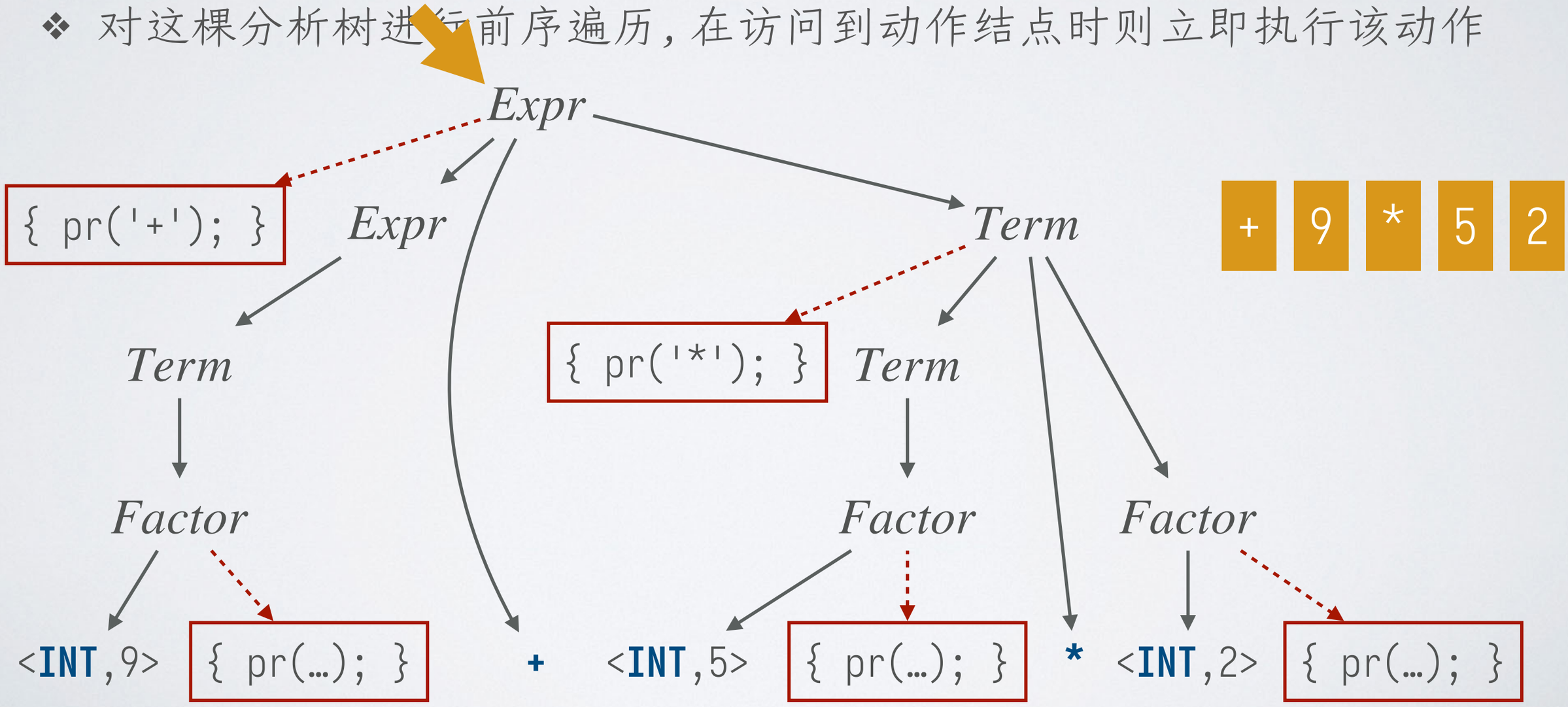
产生规则	语义动作
$Expr \rightarrow$	{ print('+'); }
$Expr_1 + Term$	
$Expr \rightarrow Term$	
$Term \rightarrow$	{ print('*'); }
$Term_1 * Factor$	
$Term \rightarrow Factor$	
$Factor \rightarrow (Expr)$	
$Factor \rightarrow$ INT	{ print(INT.lexeme); }

也可以写在一行上：

$Expr \rightarrow$ { print('+'); } $Expr_1 + Term$

嵌入语义动作的语法分析树

- **SDT 的通用实现方法：**
  - 忽略语义动作，建立语法分析树
  - 对每个内部结点 N，把它对应产生规则中的动作作为 N 的孩子结点加入
  - 对这棵分析树进行前序遍历，在访问到动作结点时则立即执行该动作



基于语法分析实现 SDT

- 回顾：
  - **LL 文法可以通过递归下降法实现**
    - 支持 S 或 L 属性的文法
  - **LR 文法可以通过移进-归约法实现**
    - 支持 S 属性的文法
    - 支持以 LL 文法为基础的 L 属性的文法

- 回顾：**LL 文法和 LR 文法都可以用表驱动的语法分析实现**

- **基于语法分析实现 SDT：**
  - 把 S 或 L 属性的文法改写为带语义动作的 SDT
  - 把 SDT 中的语义动作嵌入表驱动的语法分析中

回顾：表驱动的 LL(1) 分析

S'	‖	[ ] [ ] EOF
EOF S	‖	[ ] [ ] EOF
EOF S] S [	‖	[ ] [ ] EOF
EOF S] S	‖	[ ] [ ] EOF
EOF S] S] S [	‖	[ ] [ ] EOF
EOF S] S] S	‖	] [ ] EOF
EOF S] S]	‖	] [ ] EOF
EOF S] S	‖	[ ]] EOF
EOF S] S [	‖	[ ]] EOF
EOF S] S] S	‖	]] EOF
EOF S] S]	‖	]] EOF
EOF S] S	‖	] EOF
EOF S]	‖	] EOF
EOF S	‖	EOF
EOF	‖	EOF

[0]	$S' \to S$ EOF
[1]	$S \to \epsilon$
[2]	$S \to [\ S\ ]\ S$

使用一个栈来记录待匹配的终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta S'$	展开 [0]	错误	展开 [0]
βS	展开 [2]	展开 [1]	展开 [1]
β [	匹配	错误	错误
β]	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

回顾：表驱动的 LL(1) 分析

```
token = next_token();
top = 1; stack[top] = S′;
while (true) {
  if (top == 1 &&
      stack[top] == EOF &&
      token == EOF) {
    return true;
  } else if (top > 0 && stack[top] == token) {
    top--; token = next_token();
  } else if (top > 0 &&
      table[stack[top]][token] ==
      A → B1 B2 … Bk) {
    top--;
    for (int i = k; i >= 1; i--) {
      if (Bi != ϵ) {
        top++; stack[top] = Bi;
      }
    }
  } else {
    return false;
  }
}
```

[0]	$S' \to S\,\mathrm{EOF}$
[1]	$S \to \epsilon$
[2]	$S \to [\,S\,]\,S$

使用一个栈来记录待匹配的终结符号和非终结符号

分析栈 \ 向前看	[	]	EOF
$\beta\,S'$	展开 [0]	错误	展开 [0]
$\beta\,S$	展开 [2]	展开 [1]	展开 [1]
$\beta\,[$	匹配	错误	错误
$\beta\,]$	错误	匹配	错误
EOF	错误	错误	接受
ϵ	错误	错误	错误

以 LL 文法为基础的表驱动语义分析

- 对于规则 $B \to X\ \{\ a\ \}\ Y$，在自顶向下的语法分析的过程中，执行动作 a 的时机为：
  - 如果 Y 是非终结符号，则在**展开 Y 之前**执行
  - 如果 Y 是终结符号，则在**匹配 Y 之前**执行

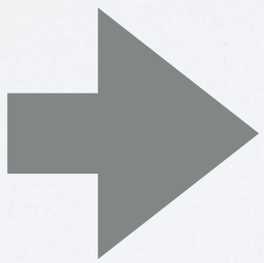
- **如何在表驱动的语法分析中支持语义动作？**

- 需要解决两方面的问题：
  - 语义动作可能操作尚未分析的符号的继承属性
  - 展开规则时，需要把语义动作存放在分析栈上

以 LL 文法为基础的表驱动语义分析

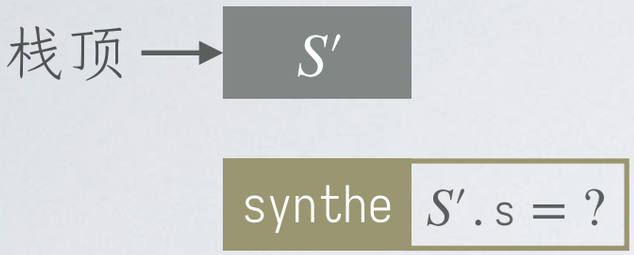
- 对于 S 或 L 属性的 LL 文法，转换为 SDT 时：
  - 计算某个非终结符号 A 的继承属性的动作插入到产生规则中**紧靠在 A 的本次出现之前的位置**
  - 计算产生规则左侧的符号的综合属性的动作插入到**该产生规则的最右侧**

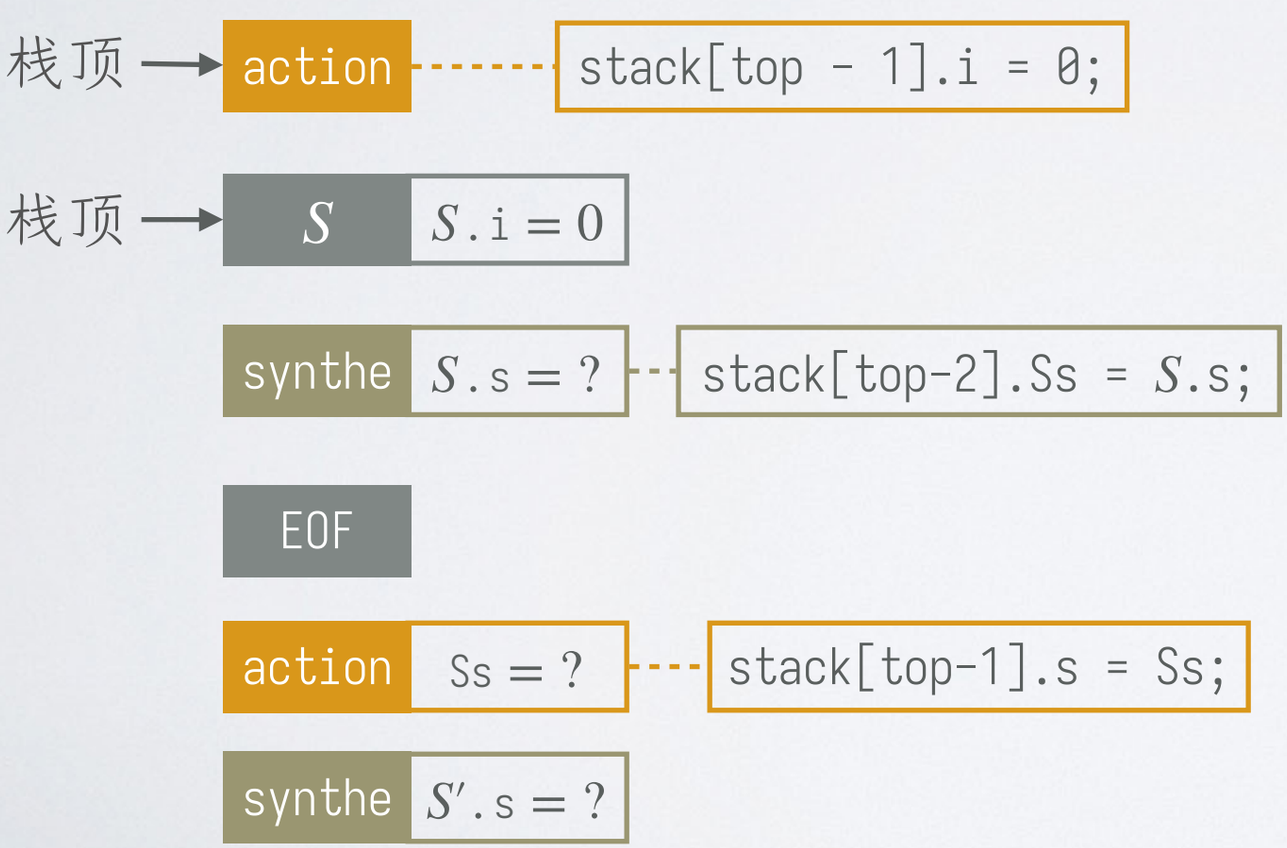
产生规则	属性计算规则
[0] $S' \to S$ EOF	$S.\mathrm{i} = 0$ $S'.\mathrm{s} = S.\mathrm{s}$
[1] $S \to \epsilon$	$S.\mathrm{s} = S.\mathrm{i}$
[2] $S \to [\ S_1\ ]\ S_2$	$S_1.\mathrm{i} = S.\mathrm{i} + 1$ $S_2.\mathrm{i} = S_1.\mathrm{s}$ $S.\mathrm{s} = S_2.\mathrm{s}$



产生规则	语义动作
[0] $S' \to$ S EOF	{ $S.\mathrm{i} = 0;$ } { $S'.\mathrm{s} = S.\mathrm{s};$ }
[1] $S \to \epsilon$	{ $S.\mathrm{s} = S.\mathrm{i};$ }
[2] $S \to [$ $S_1\ ]$ S_2	{ $S_1.\mathrm{i} = S.\mathrm{i} + 1;$ } { $S_2.\mathrm{i} = S_1.\mathrm{s};$ } { $S.\mathrm{s} = S_2.\mathrm{s};$ }

以 LL 文法为基础的表驱动语义分析

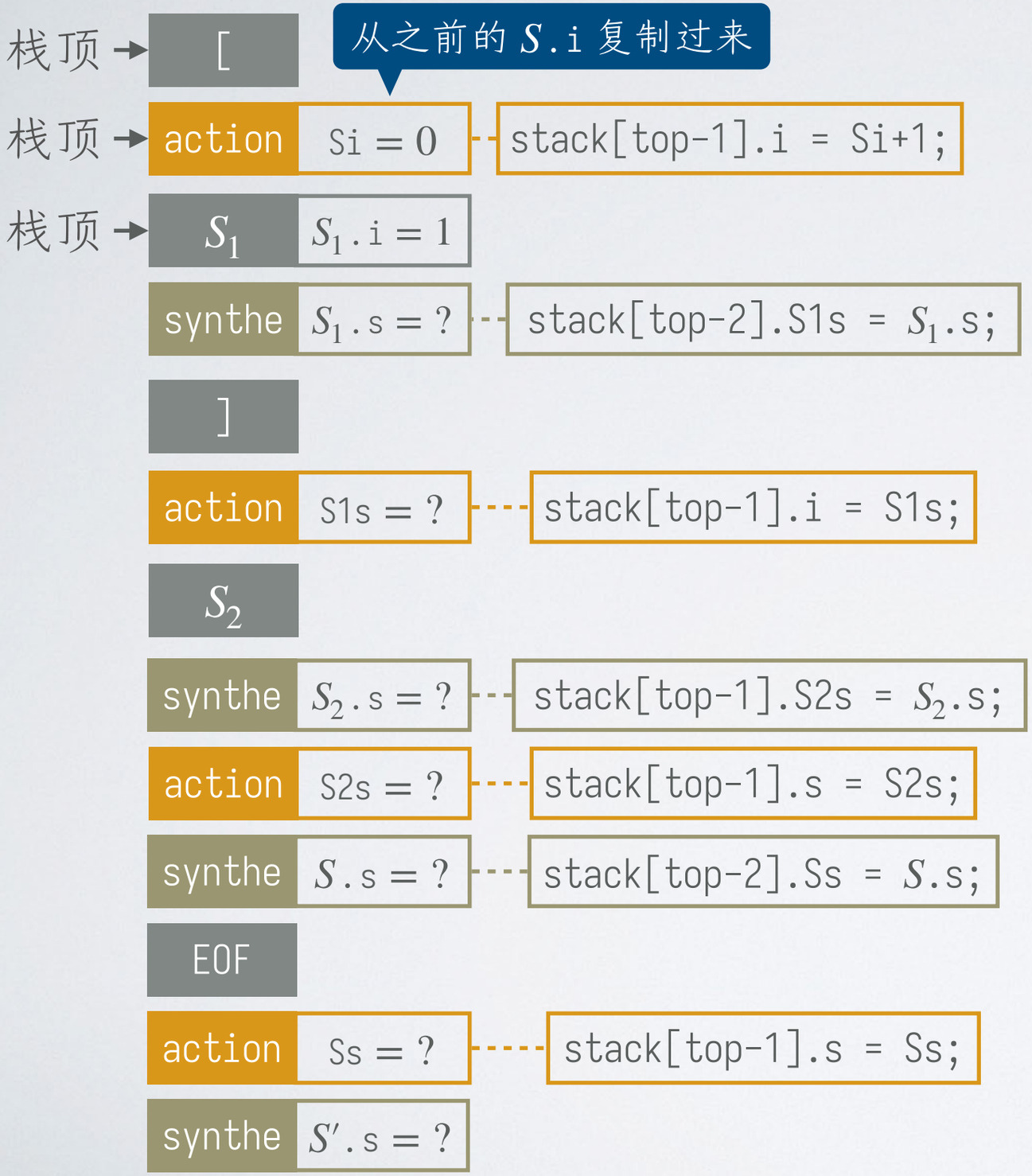




- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow S\ \mathrm{EOF}$	{ $S.\mathrm{i} = 0;$ } { $S'.\mathrm{s} = S.\mathrm{s};$ }
[1] $S \rightarrow \epsilon$	{ $S.\mathrm{s} = S.\mathrm{i};$ }
[2] $S \rightarrow [\ S_1\ ]\ S_2$	{ $S_1.\mathrm{i} = S.\mathrm{i} + 1;$ } { $S_2.\mathrm{i} = S_1.\mathrm{s};$ } { $S.\mathrm{s} = S_2.\mathrm{s};$ }

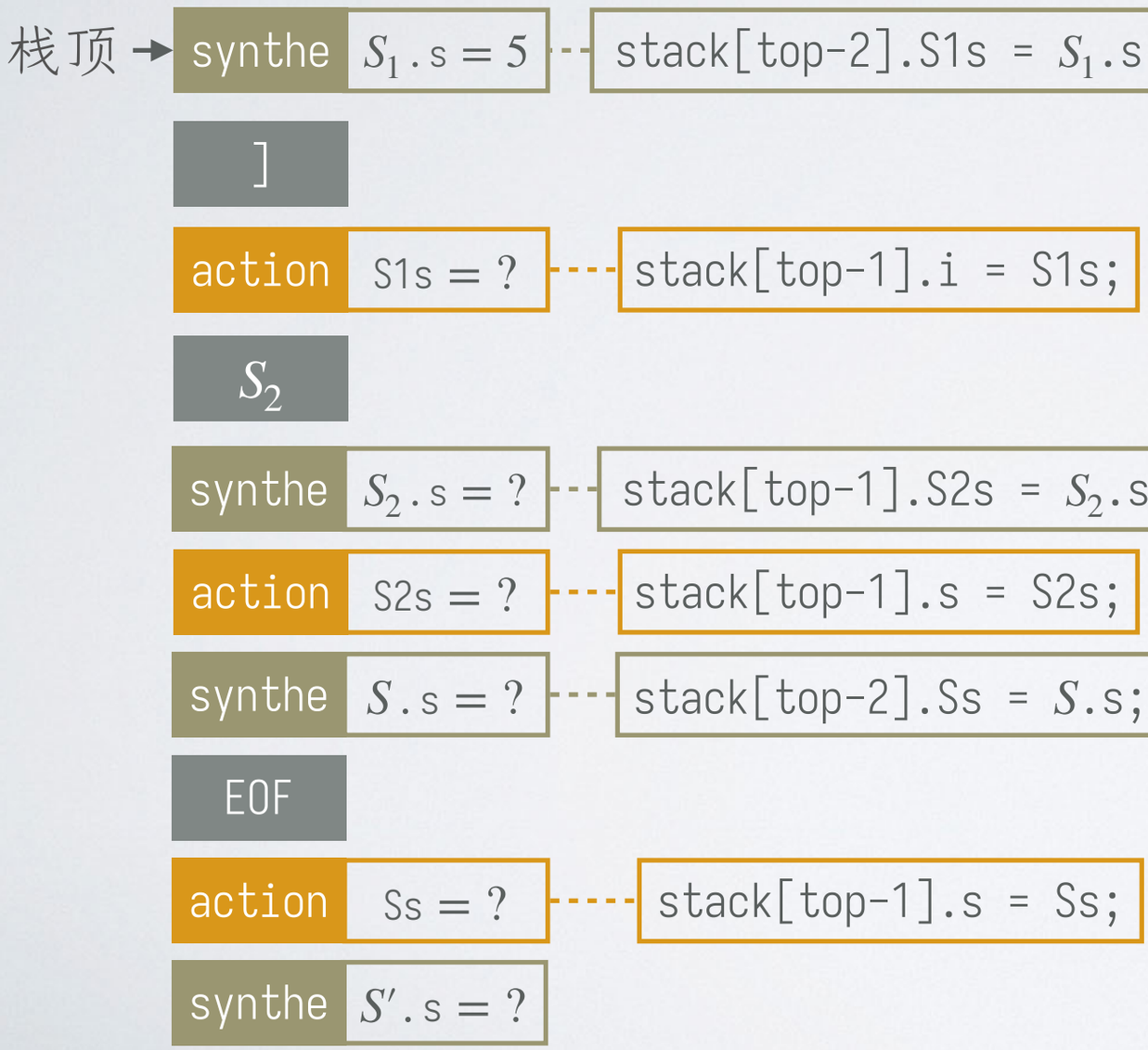
以 LL 文法为基础的表驱动语义分析



- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$ S EOF	{ S.i = 0; } { S'.s = S.s; }
[1] $S \rightarrow \epsilon$	{ S.s = S.i; }
[2] $S \rightarrow$ [S_1] S_2	{ S_1.i = S.i + 1; } { S_2.i = S_1.s; } { S.s = S_2.s; }

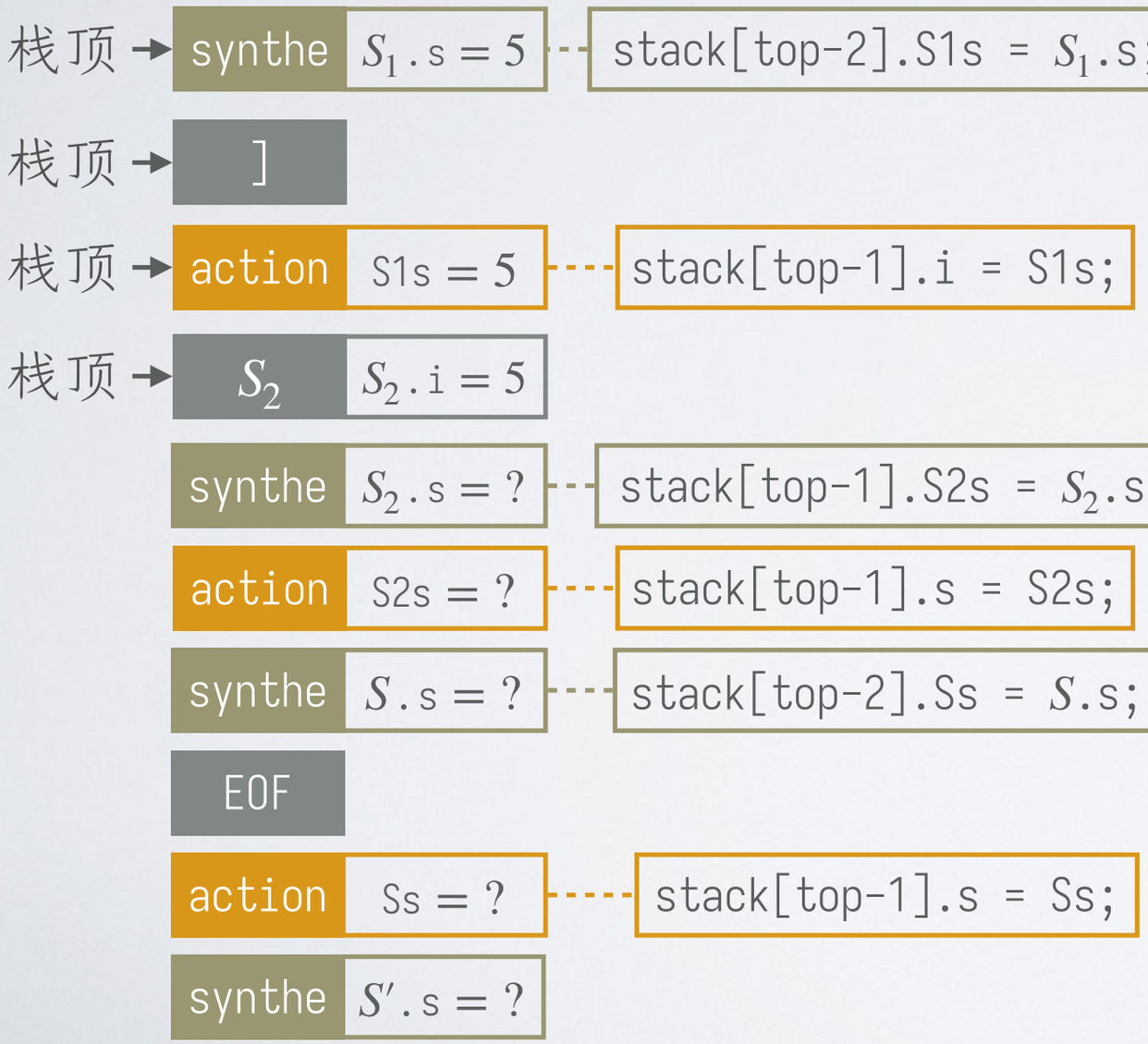
以 LL 文法为基础的表驱动语义分析



- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$	{ S.i = 0; }
S EOF	{ S'.s = S.s; }
[1] $S \rightarrow \epsilon$	{ S.s = S.i; }
[2] $S \rightarrow$ [	{ S_1.i = S.i + 1; }
S_1]	{ S_2.i = S_1.s; }
S_2	{ S.s = S_2.s; }

以 LL 文法为基础的表驱动语义分析

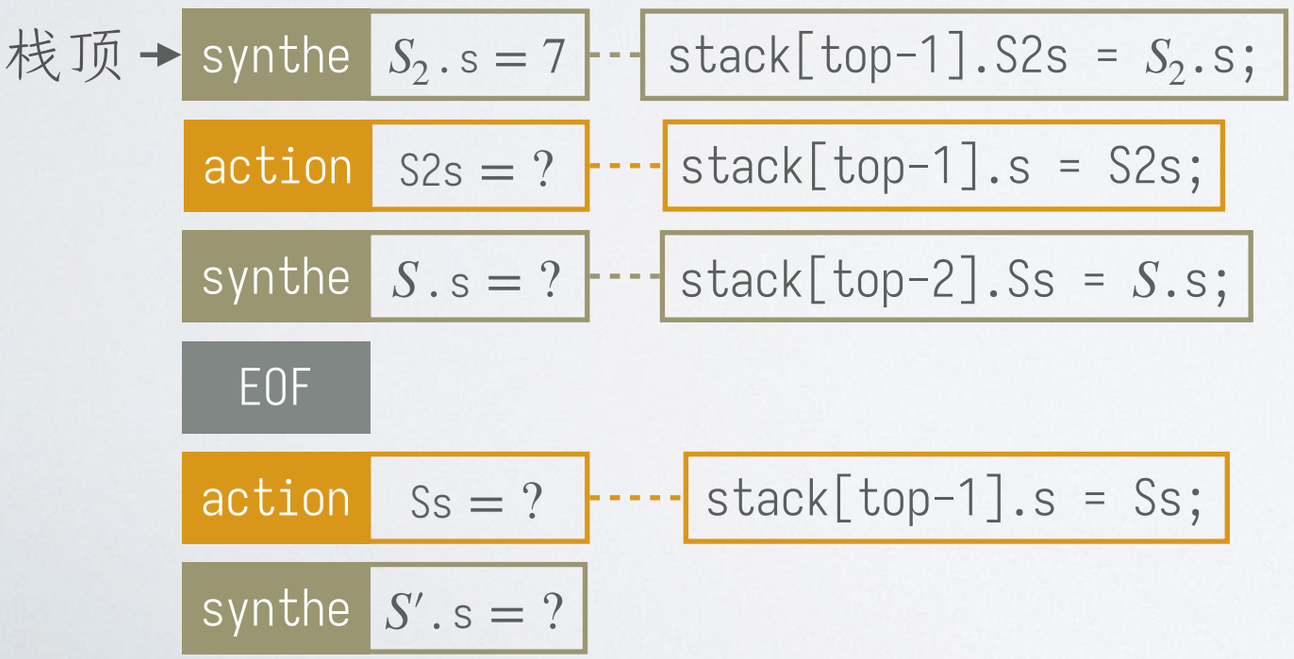


- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$ S EOF	{ S.i = 0; } { S'.s = S.s; }
[1] $S \rightarrow \epsilon$	{ S.s = S.i; }
[2] $S \rightarrow$ [S_1] S_2	{ S_1.i = S.i + 1; } { S_2.i = S_1.s; } { S.s = S_2.s; }

以 LL 文法为基础的表驱动语义分析

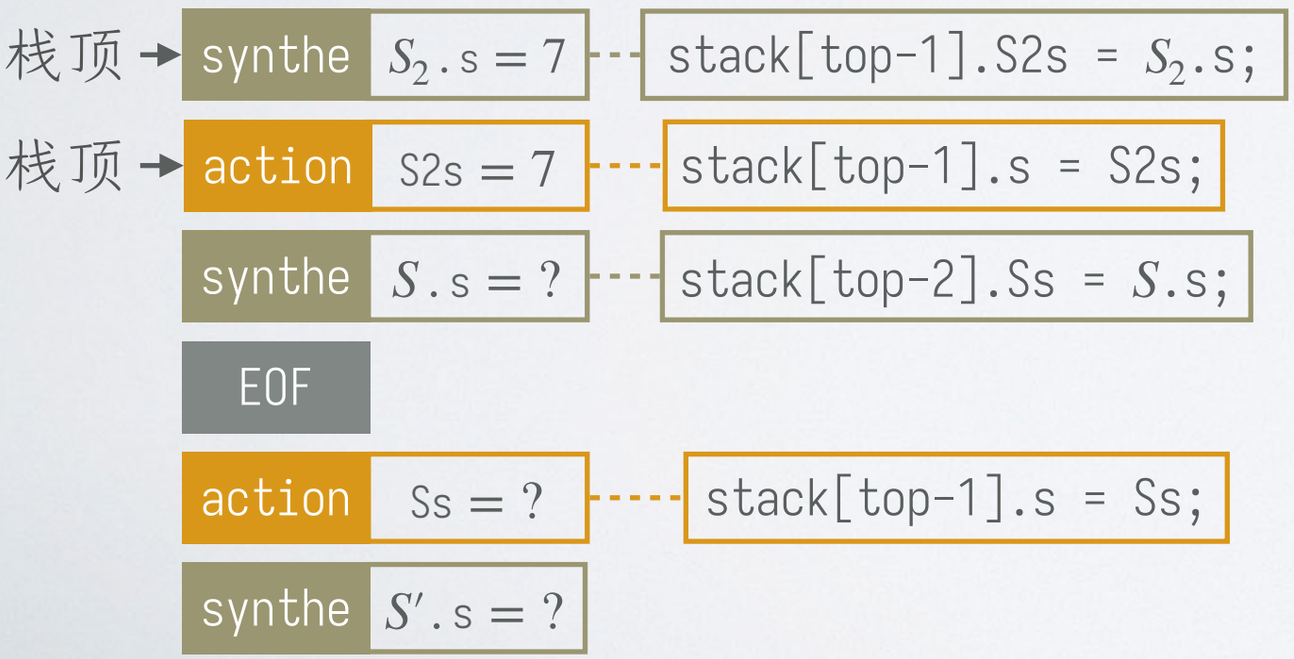
- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$	{ S.i = 0; }
S EOF	{ S'.s = S.s; }
[1] $S \rightarrow \epsilon$	{ S.s = S.i; }
[2] $S \rightarrow$ [	{ S_1.i = S.i + 1; }
S_1]	{ S_2.i = S_1.s; }
S_2	{ S.s = S_2.s; }

以 LL 文法为基础的表驱动语义分析

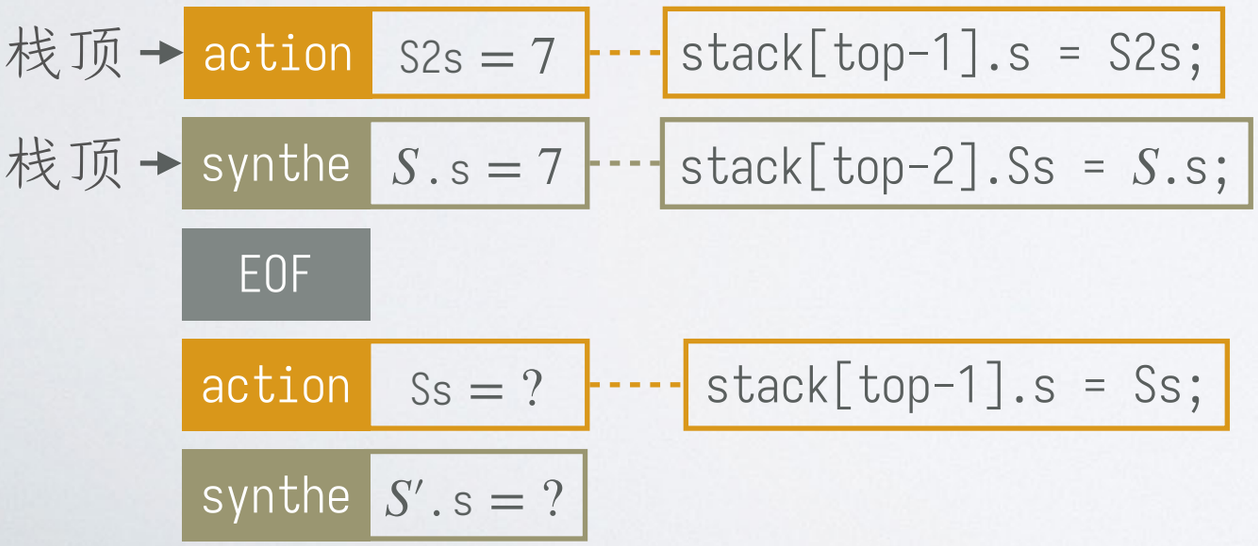
- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$	{ S.i = 0; }
S EOF	{ S'.s = S.s; }
[1] $S \rightarrow \epsilon$	{ S.s = S.i; }
[2] $S \rightarrow$ [	{ S_1.i = S.i + 1; }
S_1]	{ S_2.i = S_1.s; }
S_2	{ S.s = S_2.s; }

以 LL 文法为基础的表驱动语义分析

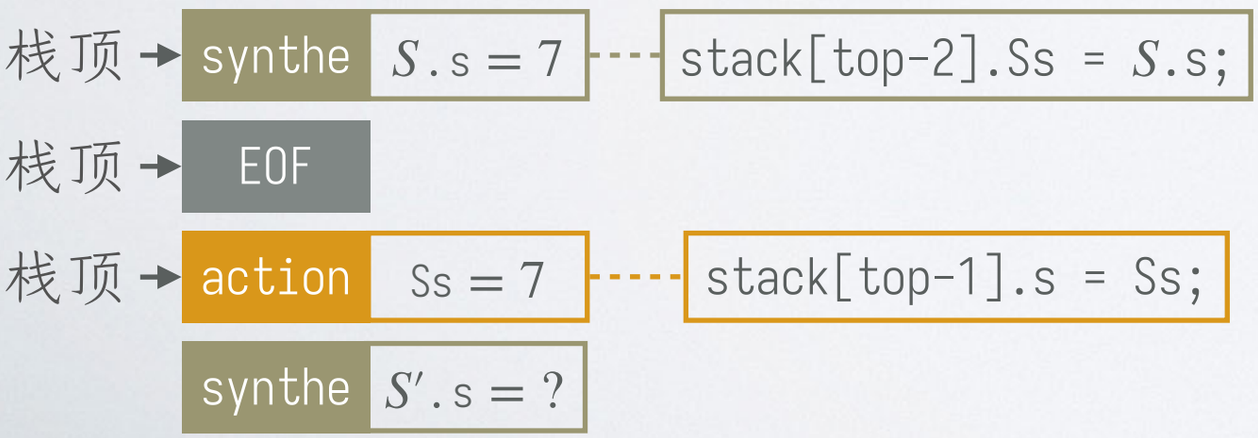
- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的



产生规则	语义动作
[0] $S' \rightarrow$ S EOF	{ $S.\mathrm{i} = 0;$ } { $S'.\mathrm{s} = S.\mathrm{s};$ }
[1] $S \rightarrow \epsilon$	{ $S.\mathrm{s} = S.\mathrm{i};$ }
[2] $S \rightarrow [$ S_1] S_2	{ $S_1.\mathrm{i} = S.\mathrm{i} + 1;$ } { $S_2.\mathrm{i} = S_1.\mathrm{s};$ } { $S.\mathrm{s} = S_2.\mathrm{s};$ }

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

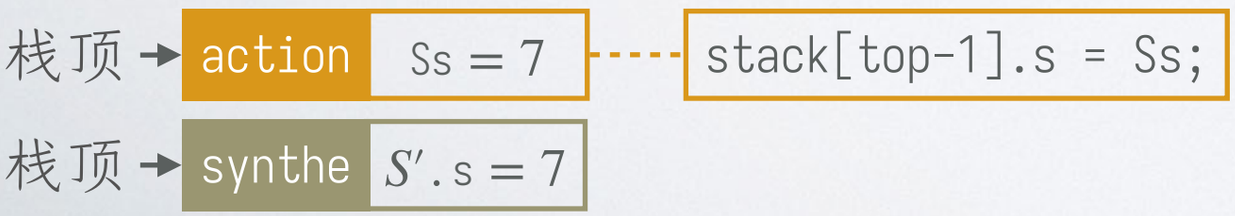


产生规则	语义动作
[0] $S' \rightarrow$	{ $S.\mathrm{i} = 0;$ }
S EOF	{ $S'.\mathrm{s} = S.\mathrm{s};$ }
[1] $S \rightarrow \epsilon$	{ $S.\mathrm{s} = S.\mathrm{i};$ }
[2] $S \rightarrow$ [	{ $S_1.\mathrm{i} = S.\mathrm{i} + 1;$ }
S_1]	{ $S_2.\mathrm{i} = S_1.\mathrm{s};$ }
S_2	{ $S.\mathrm{s} = S_2.\mathrm{s};$ }

以 LL 文法为基础的表驱动语义分析

- 语法分析栈中需要保存**综合记录**
  - 保存非终结符号的综合属性值
  - 它总是紧靠在对应的非终结符号的下方
- 语法分析栈中需要保存**动作记录**
  - 保存该动作需要的其它地方计算的属性值
  - 保存该动作的代码，可以实现为函数指针
  - 栈中符号相对位置是可以静态确定的

产生规则	语义动作
[0] $S' \rightarrow$	{ $S.\mathrm{i} = 0;$ }
S EOF	{ $S'.\mathrm{s} = S.\mathrm{s};$ }
[1] $S \rightarrow \epsilon$	{ $S.\mathrm{s} = S.\mathrm{i};$ }
[2] $S \rightarrow$ [	{ $S_1.\mathrm{i} = S.\mathrm{i} + 1;$ }
S_1]	{ $S_2.\mathrm{i} = S_1.\mathrm{s};$ }
S_2	{ $S.\mathrm{s} = S_2.\mathrm{s};$ }



LL(1) 文法的语义分析驱动程序

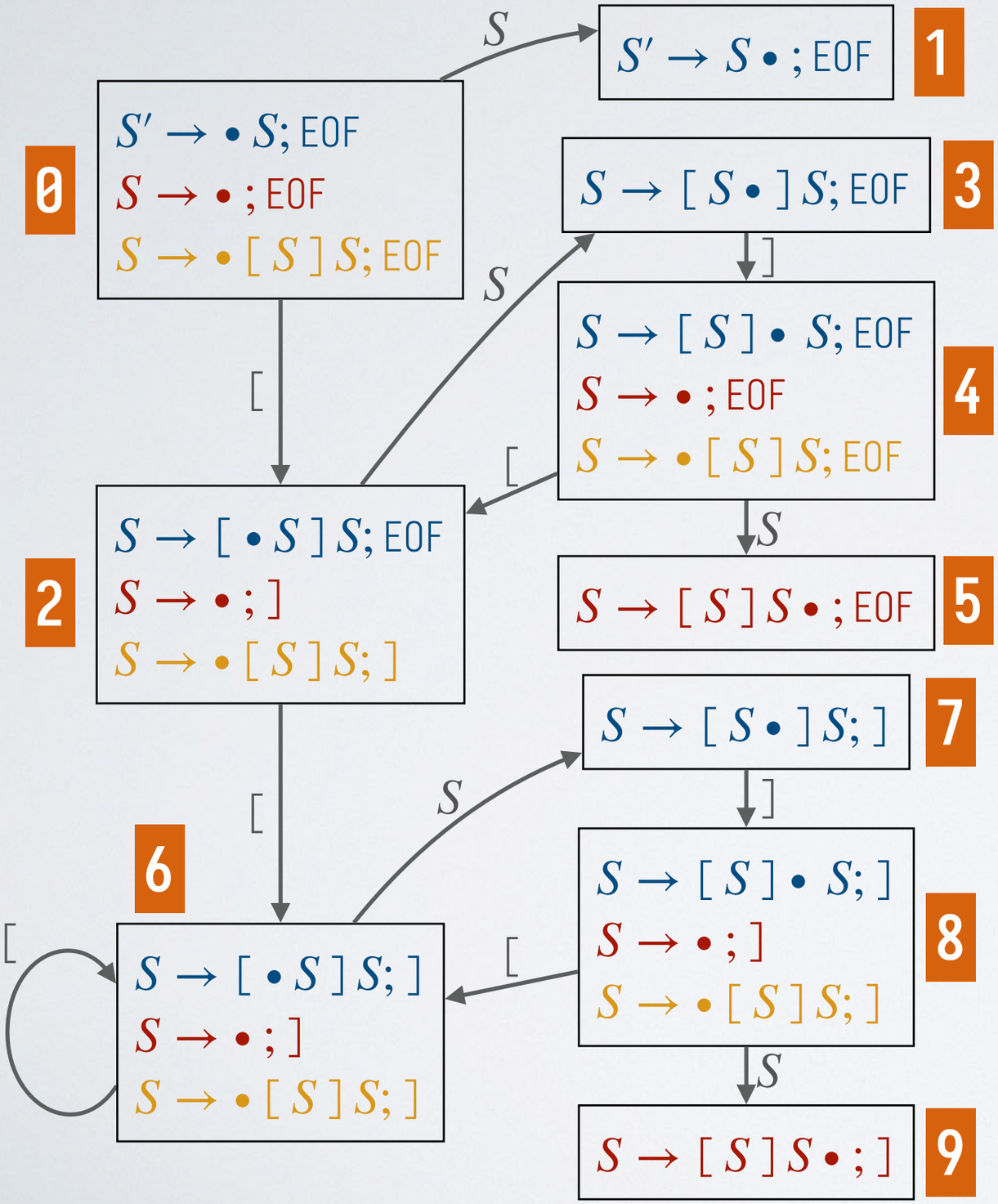
```
token = next_token();
top = 1; stack[top] = S′;
while (true) {
  if (top > 0 && stack[top] 是动作/综合记录) {
    stack[top].action_func();
    top--;
  } else if (top > 0 && stack[top] 是非终结符号 && table[stack[top]][token] == A → B1 B2 ... Bk) {
    top--;
    for (int i = k; i >= 1; i--) {
      if (Bi != ε) {
        if (Bi 是非终结符号) {
          top++; stack[top] = Bi 的综合记录;
          top++; stack[top] = Bi; }
        else if (Bi 是语义动作 act) { top++; stack[top] = act 的动作记录; }
        else { top++; stack[top] = Bi; }
      }
    }
  } else ......
}
```

记录中保存的动作代码(函数指针)可以操作全局变量 stack 和 top

存储 B_i 的综合属性，如果后面的语义动作用到这些属性，在对应的动作代码中复制这些属性到相应的位置

存储 act 会用到的属性，并在动作代码中使用这些属性完成动作的计算，把结果复制到相应的位置

回顾：表驱动的 LR(1) 分析



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

	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

回顾：表驱动的 LR(1) 分析

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

				ACTION			GOTO
				状态 \ 向前看	[	]	EOF	S
0	‖	[[ ] [ ]] EOF		0	移进 2		归约 [1]	1
0 2	‖	[ ] [ ]] EOF		1			接受	
0 2 6	‖	] [ ]] EOF		2	移进 6	归约 [1]		3
0 2 6 7	‖	] [ ]] EOF		3		移进 4		
0 2 6 7 8	‖	[ ]] EOF		4	移进 2		归约 [1]	5
0 2 6 7 8 6	‖	]] EOF		5			归约 [2]	
0 2 6 7 8 6 7	‖	]] EOF		6	移进 6	归约 [1]		7
0 2 6 7 8 6 7 8	‖	] EOF		7		移进 8		
0 2 6 7 8 6 7 8 9	‖	] EOF		8	移进 6	归约 [1]		9
0 2 6 7 8 9	‖	] EOF		9		归约 [2]		
0 2 3	‖	] EOF						
0 2 3 4	‖	EOF						
0 2 3 4 5	‖	EOF						
0 1	‖	EOF						

回顾：表驱动的 LR(1) 分析

```
token = next_token();
top = 1; stack[top] = I_0;
while (true) {
  state = stack[top];
  if (ACTION[state][token] == "归约 A → α") {
    top = top - |α|;
    state = stack[top];
    top++; stack[top] = GOTO[state][A];
  } else if (ACTION[state][token] == "移进 I_j") {
    top++; stack[top] = I_j;
    token = next_token();
  } else if (ACTION[state][token] == "接受") {
    return true;
  } else {
    return false;
  }
}
```

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

ACTION　　GOTO

状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

- 对于规则 $B \to X \{ a \} Y$，在自底向上的语法分析的过程中，执行动作 a 的时机为：
  - 当 X 的此处出现位于分析栈的栈顶时，立即执行动作 a

- **如何知道是在进行 $B \to XY$ 的推导？**

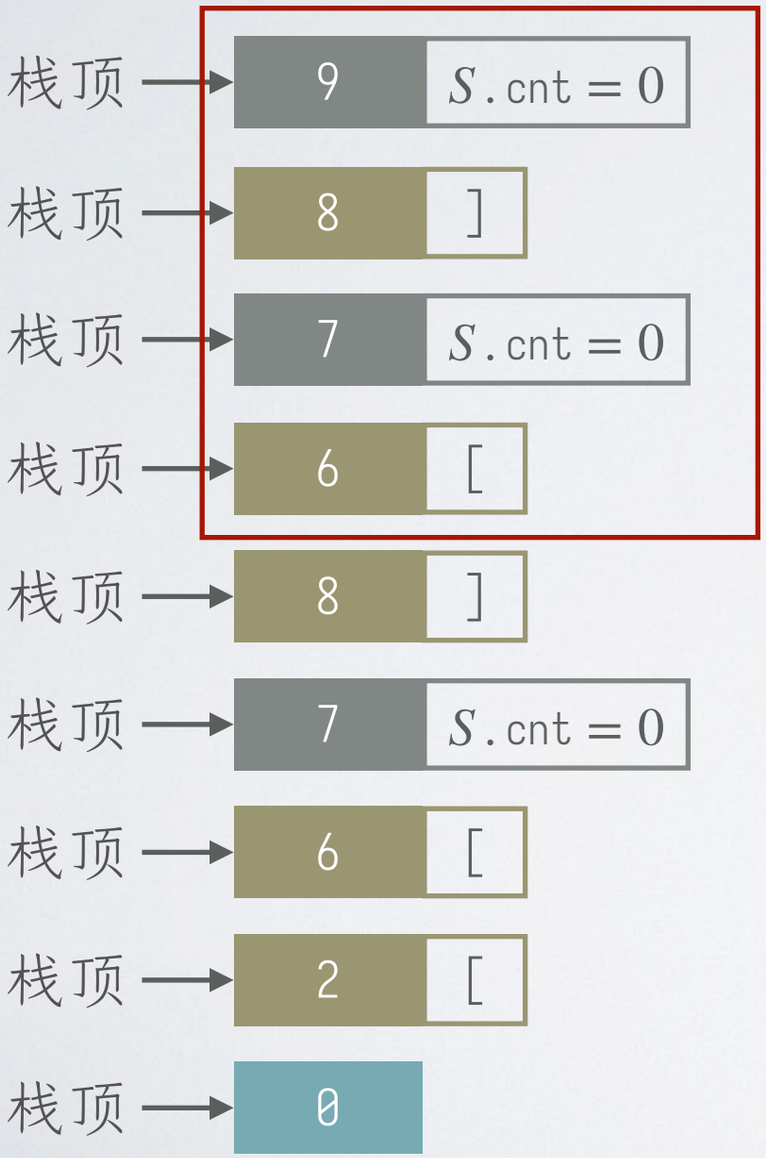
- 一般而言，在进行归约时，才能确定使用的推导规则

- **解决方案：**要求 SDT 的语义动作都出现在产生规则的最右侧
  - 语义动作只会在归约的时候执行
  - **适用于 S 属性的文法**

- 在分析栈中除了记录自动机状态，还需记录各符号的属性值

以 LR 文法为基础的表驱动语义分析

[[ ] [ ]] EOF



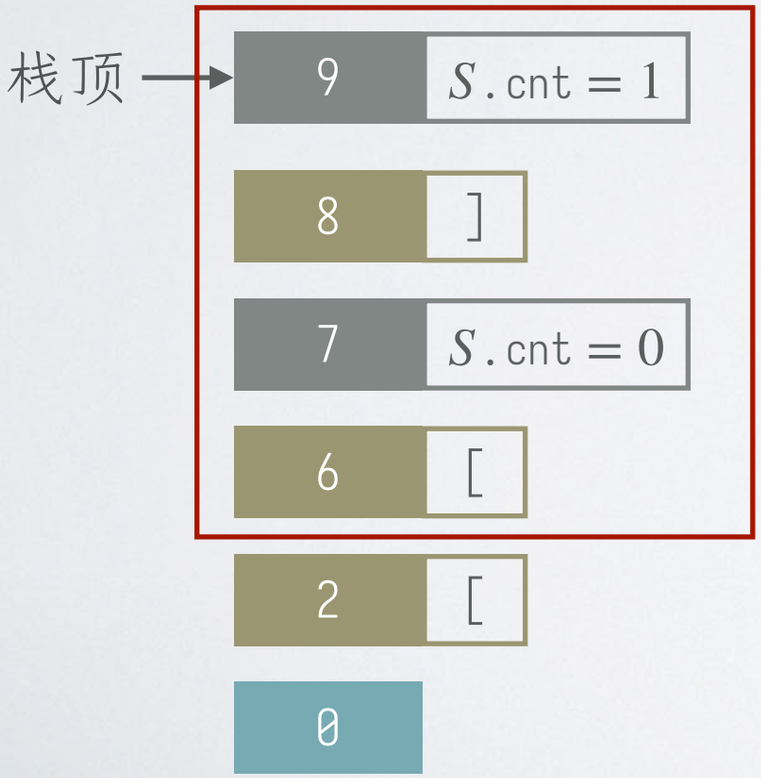
	$S' \to S\,\mathrm{EOF}$	$\{\ S'.\mathrm{cnt} = S.\mathrm{cnt};\ \}$
[1]	$S \to \epsilon$	$\{\ S.\mathrm{cnt} = 0;\ \}$
[2]	$S \to [\ S_1\ ]\ S_2$	$\{\ S.\mathrm{cnt} = 1 + S_1.\mathrm{cnt} + S_2.\mathrm{cnt};\ \}$

	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[ ] [ ]] EOF

	$S' \to S\,\texttt{EOF}$	$\{\ S'.\texttt{cnt} = S.\texttt{cnt};\ \}$
[1]	$S \to \epsilon$	$\{\ S.\texttt{cnt} = 0;\ \}$
[2]	$S \to [\ S_1\ ]\ S_2$	$\{\ S.\texttt{cnt} = 1 + S_1.\texttt{cnt} + S_2.\texttt{cnt};\ \}$

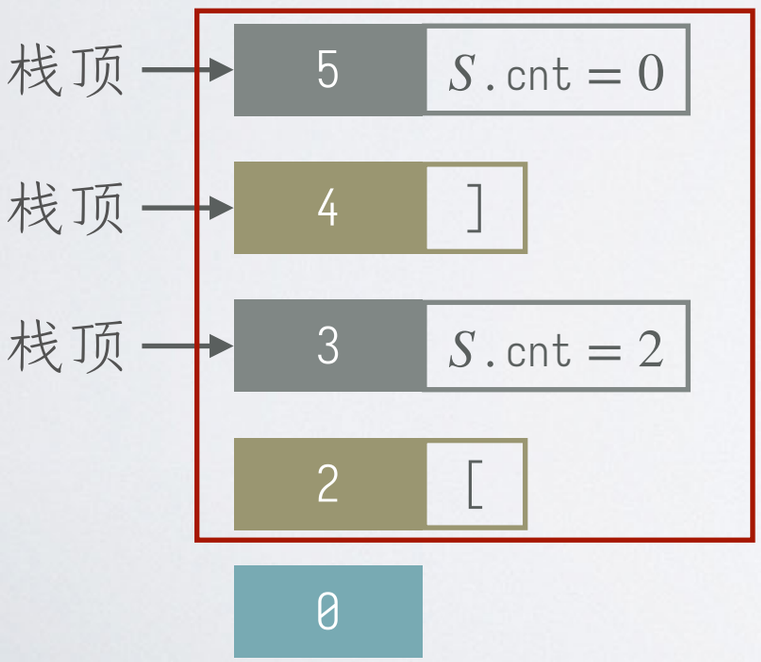


	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[ ] [ ]] EOF

	$S' \to S\,\texttt{EOF}$	$\{\ S'.\texttt{cnt} = S.\texttt{cnt};\ \}$
[1]	$S \to \epsilon$	$\{\ S.\texttt{cnt} = 0;\ \}$
[2]	$S \to [\ S_1\ ]\ S_2$	$\{\ S.\texttt{cnt} = 1 + S_1.\texttt{cnt} + S_2.\texttt{cnt};\ \}$



	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[ ] [ ]] EOF

	$S' \to S\,\texttt{EOF}$	$\{\ S'.\texttt{cnt} = S.\texttt{cnt};\ \}$
[1]	$S \to \epsilon$	$\{\ S.\texttt{cnt} = 0;\ \}$
[2]	$S \to [\ S_1\ ]\ S_2$	$\{\ S.\texttt{cnt} = 1 + S_1.\texttt{cnt} + S_2.\texttt{cnt};\ \}$

	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

以 LR 文法为基础的表驱动语义分析

[[ ] [ ]] EOF

	$S' \to S\,\texttt{EOF}$	$\{\ S'.\texttt{cnt} = S.\texttt{cnt};\ \}$
[1]	$S \to \epsilon$	$\{\ S.\texttt{cnt} = 0;\ \}$
[2]	$S \to [\ S_1\ ]\ S_2$	$\{\ S.\texttt{cnt} = 1 + S_1.\texttt{cnt} + S_2.\texttt{cnt};\ \}$

	ACTION			GOTO
状态 \ 向前看	[	]	EOF	S
0	移进 2		归约 [1]	1
1			接受	
2	移进 6	归约 [1]		3
3		移进 4		
4	移进 2		归约 [1]	5
5			归约 [2]	
6	移进 6	归约 [1]		7
7		移进 8		
8	移进 6	归约 [1]		9
9		归约 [2]		

LR(1) 文法的语义分析驱动程序

自动机状态

```
token = next_token();
top = 1; stack[top].state = I_0;
while (true) {
  state = stack[top].state;
  if (ACTION[state][token] == "归约 A → α { act }") {
    根据 stack[top - |α| + 1] 到 stack[top] 的属性完成 act 动作;
    top = top - |α|;
    state = stack[top].state;
    top++; stack[top].state = GOTO[state][A];
    stack[top].symbol = A; 根据前面的计算结果保存 A 的属性值;
  } else if (ACTION[state][token] == "移进 I_j") {
    top++; stack[top].state = I_j;
    stack[top].symbol = token;
    token = next_token();
  } else if (ACTION[state][token] == "接受") {
    根据 stack[top] 的属性计算 S' 的属性值并存放于 stack[top];
    return true;
  } else return false;
}
```

显式操作分析栈的 SDT

- 基于 LR 文法的语义分析所使用的 SDT 可以显式操作分析栈

产生规则	语义动作
[0] $S' \to S$ EOF	{ S'.cnt = S.cnt; }
[1] $S \to \epsilon$	{ S.cnt = 0; }
[2] $S \to [\ S_1\ ]\ S_2$	{ S.cnt = 1 + S_1.cnt + S_2.cnt; }

产生规则	语义动作
[0] $S' \to S$ EOF	{ stack[top-1].cnt = stack[top-1].cnt; top = top-1; }
[1] $S \to \epsilon$	{ stack[top+1].cnt = 0; top = top+1; }
[2] $S \to [\ S_1\ ]\ S_2$	{ stack[top-3].cnt = 1 + stack[top-2].cnt + stack[top].cnt; top = top-3; }

此处假设 EOF 也像其它 token 一样移进了分析栈

回顾：L 属性的文法的移进-归约实现

- 在 LL 文法中插入标记非终结符号(如下面的 L、M、N)后，得到的文法是一个 LR 文法，可以用移进-归约实现语义分析

```
top++; stack[top] = S;
stack[top].s = stack[top - 1].i;
```

```
top++; stack[top] = M;
stack[top].i = stack[top - 2].i + 1;
```

```
top++; stack[top] = N;
stack[top].i = stack[top - 2].s;
```

产生规则	属性计算规则
$S' \to L\,S\,\mathrm{EOF}$	$S'.\mathrm{s} = S.\mathrm{s}$
$S \to \epsilon$	$S.\mathrm{s} =$ 栈中归约位置减 1 处$.\mathrm{i}$
$S \to [\,M\,S_1\,]\,N\,S_2$	$S.\mathrm{s} = S_2.\mathrm{s}$
$L \to \epsilon$	$L.\mathrm{i} = 0$
$M \to \epsilon$	$M.\mathrm{i} =$ 栈中归约位置减 2 处$.\mathrm{i} + 1$
$N \to \epsilon$	$N.\mathrm{i} =$ 栈中归约位置减 2 处$.\mathrm{s}$

- **可以通过显式操作分析栈的 SDT 来进行实现**

L 属性的文法的移进-归约实现

文法本身是 LL 的

产生规则	属性计算规则
[0] $S' \to S\ \texttt{EOF}$	$S.\texttt{i} = 0$ $S'.\texttt{s} = S.\texttt{s}$
[1] $S \to \epsilon$	$S.\texttt{s} = S.\texttt{i}$
[2] $S \to [\ S_1\ ]\ S_2$	$S_1.\texttt{i} = S.\texttt{i} + 1$ $S_2.\texttt{i} = S_1.\texttt{s}$ $S.\texttt{s} = S_2.\texttt{s}$

产生规则	语义动作
$S' \to L\ S\ \texttt{EOF}$	{ stack[top-2].s = stack[top-1].s; top = top-2; }
$S \to \epsilon$	{ stack[top+1].s = stack[top].i; top = top+1; }
$S \to [\ M\ S_1\ ]\ N\ S_2$	{ stack[top-5].s = stack[top].s; top = top-5; }
$L \to \epsilon$	{ stack[top+1].i = 0; top = top+1; }
$M \to \epsilon$	{ stack[top+1].i = stack[top-1].i + 1; top = top+1; }
$N \to \epsilon$	{ stack[top+1].i = stack[top-1].s; top = top+1; }

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
- 语义分析的自动生成
- One More Thing

类型化的语言

- **类型化的语言：**变量都被给定类型的语言
  - 表达式、语句等语法构造的类型都是静态确定的
  - 例如，类型 bool 的变量 x 和 y 在程序运行时的值只能是布尔值
    - 从而表达式 x && y 总是良定义的

- **非类型化的语言：**不限制变量值范围的语言
  - 一个运算可以作用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未明确定义的结果
  - 例如：Python、JavaScript 等语言

- **Gradual Typing：**混合静态类型和动态类型
  - 平衡语言的**安全性**和**灵活性**

类型表达式

- 类型自身也有结构
- **类型表达式(type expression)** 用来表示程序中变量、常量、表达式、语句等语言成分的类型
  - 基本类型: bool、char、int、float、void 等
  - 面向对象语言中的类名
  - 数组类型: *array* 作用于一个数字和一个类型表达式
  - 记录类型: *record* 作用于字段名和相应的类型
  - 函数类型: 从类型 s 到类型 t 的函数表示为 $s \rightarrow t$
  - 笛卡尔积: $s \times t$ 可以用来表示类型的元组(例如函数参数)
  - 取值为类型表达式的变量

类型表达式的例子

- C 语言中的结构体:

```
struct {
  int no;
  char name[20];
}
```

- 它对应的类型表达式为:
  - $record(\{no : int, name : array(20,char)\})$
  - 或者(如果不考虑字段名)
  - $int \times array(20,char)$

类型等价

- **类型等价(type equivalence)**：两个类型的值的集合相等并且作用在其上的运算集合相等
  - 类型等价具有对称性
- 类型等价主要可以分为两种：
  - **按名字等价**：两个类型名字相同，或者被定义成等价的两个名字
    - 比如：C++ 和 Java 中的类定义
  - **按结构等价**：两个类型的结构完全相同，但是名字不一定相同(或者不一定有名字)
    - 比如：Standard ML 中的记录类型

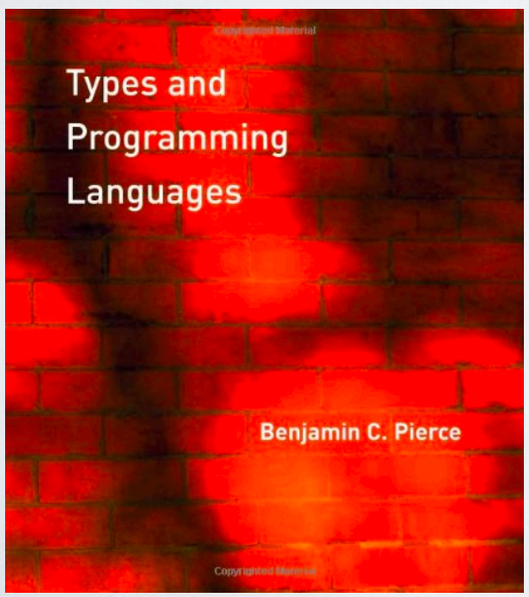
$$\{x : int, y : int\} \equiv \{y : int, x : int\}$$

  - 按名字等价一定是按结构等价的

类型检查

- **类型系统(type system)：**
  - 给程序的每一个组成部分赋予一个类型表达式
  - 通过一组逻辑规则来表示这些类型表达式必须满足的条件
  - **健全性(soundness)**：保证某类类型错误不会在运行时发生

- **「类型系统是指一种根据程序所计算的值进行分类从而证明某些(不良)程序行为不会发生的轻量级形式化方法」**



Benjamin C. Pierce, **Types and Programming Languages**

类型检查规则分类

- **类型综合(type synthesis)**
  - 根据子表达式的类型构造出表达式的类型

    如果　f 的类型为 $s \to t$ 且 x 的类型为 s

    那么　表达式 $f(x)$ 的类型为 t

- **类型推导(type inference)**
  - 根据语言结构的使用方式来确定该结构的类型

    如果　$f(x)$ 是一个表达式

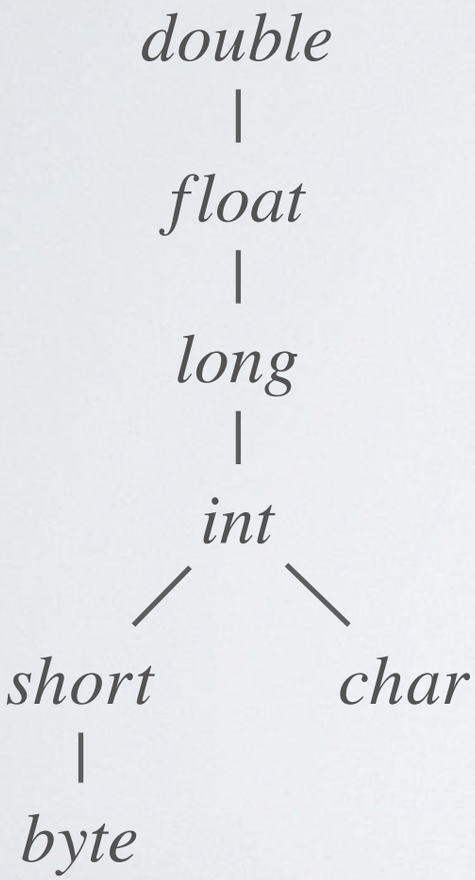
    那么　对于某些类型 α 和 β, f 的类型是 $\alpha \to \beta$ 且 x 的类型为 α

  - α 和 β 可以是类型表达式中的类型变量

- **Bidirectional Typing**：结合类型综合和推导
  - 有确定的类型信息时，通过类型综合传播信息
  - 出现未知(或未写明的)类型时，进行类型推导

类型的拓宽和窄化

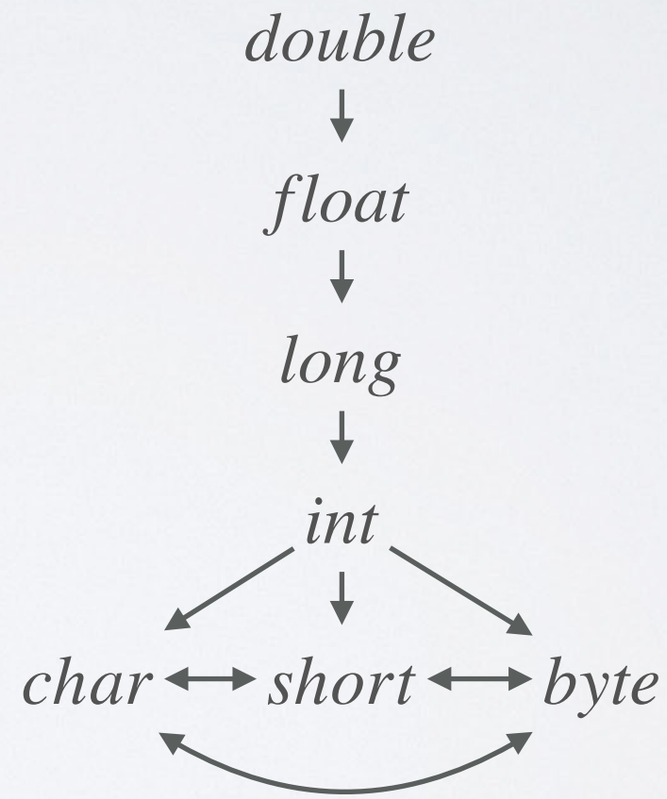
- Java 的类型转换规则：
  - **拓宽(widening)**：保持转换前的原有数值范围信息(但可能**丢失精度**)
  - **窄化(narrowing)**：转换可能丢失数值范围信息



拓宽

```
byte b = 4;
short s = b;
int i = s;
long l = i;
float f = l;
double d = f;
```

隐式转换



窄化

```
double d = 4;
float f = (float)d;
long l = (long)f;
int i = (int)l;
short s = (short)i;
byte b = (byte)s;
```

显式转换

函数和运算符的重载

- 根据符号所在的上下文不同，被**重载(overloaded)**的符号可能有不同的含义

- 针对重载函数的类型综合规则：

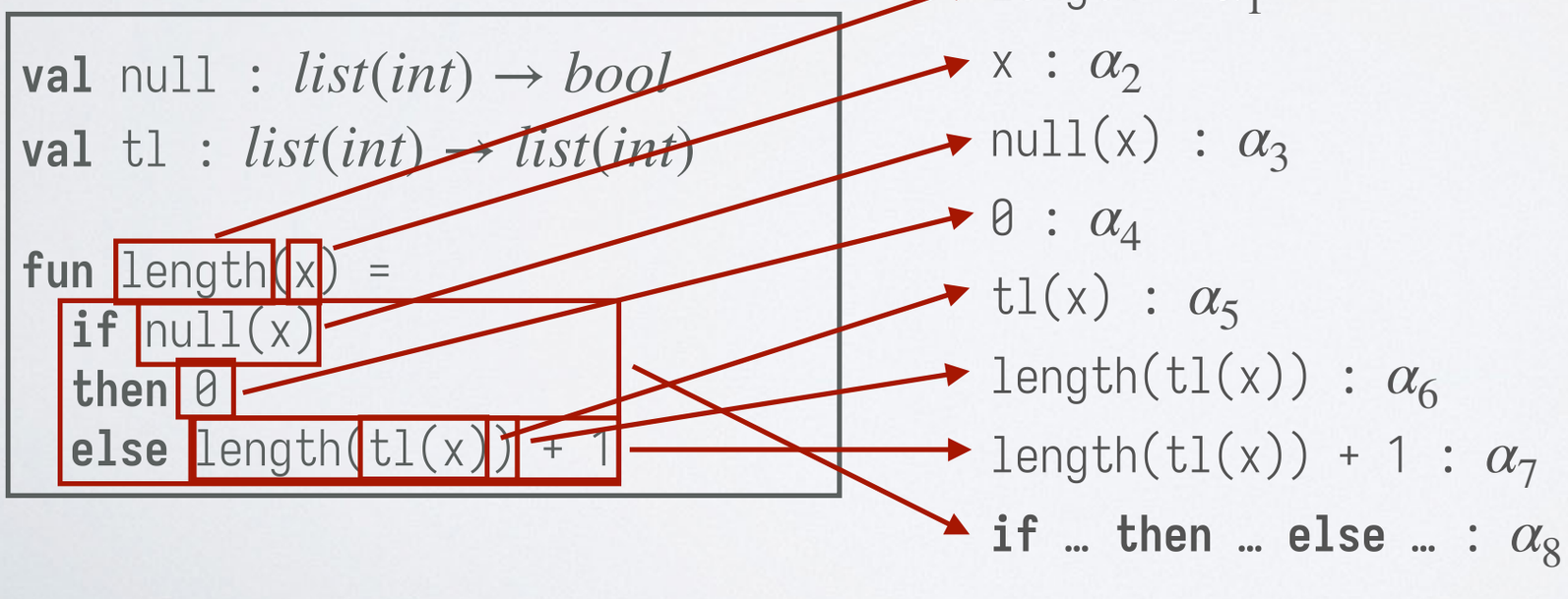
  如果 f **可能的类型为** $s_i \to t_i\ (1 \le i \le n)$，**其中** $s_i \ne s_j\ (i \ne j)$

  且 x **的类型为** $s_k\ (1 \le k \le n)$

  那么 **表达式** $f(x)$ **的类型为** t_k

类型推导

- **问题：**类型化的语言可能要求程序员**写很多的类型标注**
  - 比如 C++ 和 Java 等语言
- **类型推导**可以一定程度上降低程序员的负担
  - 比如 C++ 中的 auto，Rust、Haskell 等语言中不同程度的自动推导
- 类型推导的一般方法：**解方程**



$$\alpha_1 = list(int) \to int$$

↑ 解

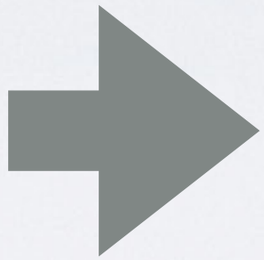
$$\alpha_2 \to \alpha_3 = list(int) \to bool$$
$$\alpha_4 = int$$
$$\alpha_2 \to \alpha_5 = list(int) \to list(int)$$
$$\alpha_5 \to \alpha_6 = \alpha_1$$
$$\alpha_6 = int$$
$$\alpha_7 = int$$
$$\alpha_4 = \alpha_7$$
$$\alpha_8 = \alpha_4$$
$$\alpha_1 = \alpha_2 \to \alpha_8$$

参数多态

- **参数多态(parametric polymorphism)**：函数可以在不同类型(通过类型变量来刻画)的参数上运行

```
val null : list(int) → bool
val tl : list(int) → list(int)

fun length(x) =
  if null(x)
  then 0
  else length(tl(x)) + 1
```



```
val null : ∀β . list(β) → bool
val tl : ∀β . list(β) → list(β)

fun length(x) =
  if null(x)
  then 0
  else length(tl(x)) + 1
```

- 在类型推导中，多态函数每次应用可能作用于不同的类型

length : α_1

x : α_2

null(x) : α_3

0 : α_4

tl(x) : α_5

length(tl(x)) : α_6

length(tl(x)) + 1 : α_7

if … then … else … : α_8

方程

$$\alpha_2 \to \alpha_3 = list(\beta_1) \to bool$$
$$\alpha_2 \to \alpha_5 = list(\beta_2) \to list(int)$$

解

$$list(\beta_1) = \alpha_2$$
$$list(\beta_2) = \alpha_2$$
$$\alpha_1 = \alpha_2 \to int$$

泛化

length : $\forall \beta_1 . list(\beta_1) \to int$

历史：从 B 语言到 C 语言

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- **无类型**
- **性能差**

历史：从 B 语言到 C 语言

```
printn(n, b) {
   extrn putchar;
   auto a;

   a = n / b;
   if (a)
      printn(a, b);
   putchar(n % b + '0');
}
```

B 语言

函数参数是**无类型**的
函数也没有返回类型

历史：从 B 语言到 C 语言

```
printn(n, b) {
  extrn putchar;
  auto a;
  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

auto 关键字仅表示为
变量 a 预留存储空间

B 语言中只有一种类型：
所有数据都是字(word)

历史：从 B 语言到 C 语言

```
printn(n, b) {
   extrn putchar;
    auto a;

   a = n / b;
   if (a)
      printn(a, b);
   putchar(n % b + '0');
}
```

B 语言

调用的外部函数**没有函数签名**

编译时**无法检查**函数调用
是否传入了正确的参数

运行时检查降低了程序性能

历史：从 B 语言到 C 语言

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

```
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

历史：从 B 语言到 C 语言

```c
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

- 1972 年
- 贝尔实验室
- 通用编程语言，常被用来开发操作系统、设备驱动、协议栈等
- **静态类型**
- **性能优越**

类型系统的优点

- **检查错误**
  - 函数调用不符合签名，对浮点数进行按位与操作，…
- **提供文档**
  - 函数的类型标注提供了基本的函数文档
- **编译优化**
  - 去除不必要的运行时检查，优化数据结构的内存布局，…
- **语言安全**
  - Rust 提供的内存安全，TypeScript 提供的针对 JavaScript 的安全，…
- **抽象机制**
  - 带类型的泛型编程，面向对象编程，函数式编程，…

类型系统与语言设计

语言设计应当同类型系统设计并行进行且互相协同

- 不含类型系统的语言在设计时容易引入**难以进行类型检查**的编程特性
- 带类型语言的**语法可能更复杂**，需要在设计中考虑类型标注的清晰性

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语法分析进行同步实现
- 语义分析的自动生成基于语法制导的翻译方案
  - 以 LL 文法为基础的 L 属性的文法
  - 以 LR 文法为基础的 S 属性的文法

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？
- 语义分析和语法分析结合，有什么优点？有什么缺点？
- 常见的语法分析生成工具（如 YACC）支持什么样的语法制导的翻译方案？为什么？