



中国科学技术大学  
University of Science and Technology of China

# 语法制导的翻译

《编译原理和技术》

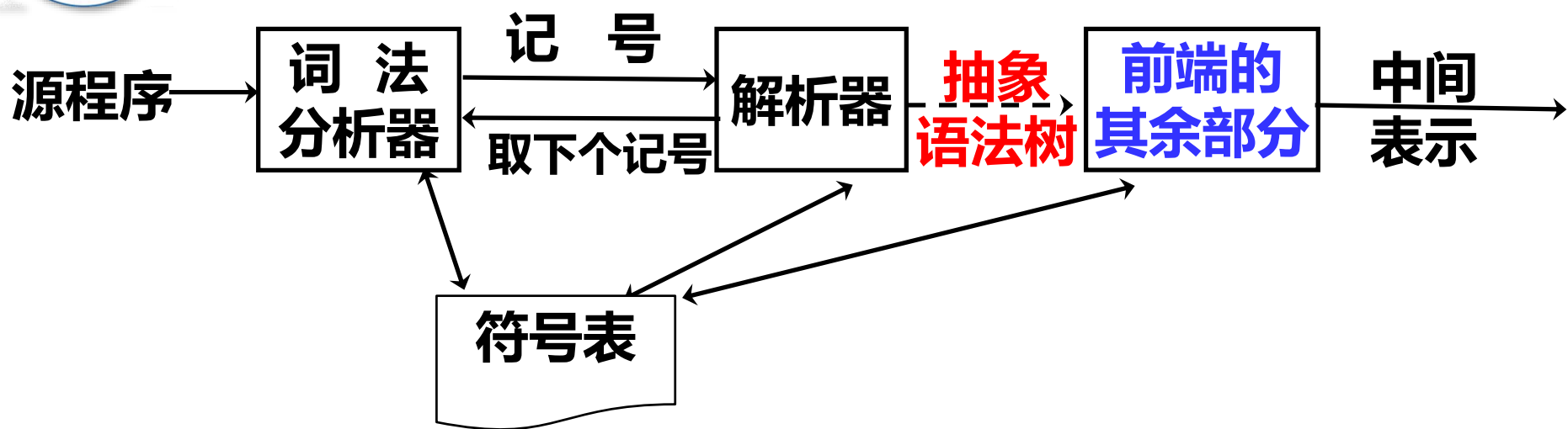
张昱

0551-63603804, [yuzhang@ustc.edu.cn](mailto:yuzhang@ustc.edu.cn)

中国科学技术大学  
计算机科学与技术学院



# 本章内容



## □ 语义的描述：语法制导的定义、翻译方案

### ■ 语法制导：syntax-directed

按语法结构来指导语义的定义和计算

### ■ 抽象语法树、注释分析树等

## □ 语法制导翻译的实现方法：自上而下、自下而上

### ■ 边语法分析边翻译



# 本章内容

## □ 语义描述的一种形式方法

### ■ 语法制导的定义(syntax-directed definition)

$$E \rightarrow E_1 + T$$

$$E.code = E_1.code \parallel T.code \parallel '+'$$

*code*属性表示表达式对应的后缀形式

语法制导定义可读性好，更适于描述规范

### ■ 翻译方案(translation scheme)

$$E \rightarrow E_1 + T$$

$$\{ \text{print '+'} \}$$

翻译方案陈述了实现细节(如语义规则的计算时机)

## □ 语法制导翻译技术可以用于

### ■ 构建抽象语法树、语义分析、中间代码生成等



中国科学技术大学  
University of Science and Technology of China

## 4.1 语法制导的定义

- 语法制导的定义
- 综合属性、继承属性
- 属性依赖图与属性的计算次序



# 语法制导的定义

## □ 语法制导的定义(Syntax-Directed Definition)

■ 基础的上下文无关文法

■ 每个文法符号有一组属性

用来表示语法成分  
对应的语义信息

■ 每个文法产生式  $A \rightarrow \alpha$  有一组形式为  $b = f(c_1, c_2, \dots, c_k)$  的  
语义规则

描述语义属性值的计算规则

其中  $f$  是函数,  $b$  和  $c_1, c_2, \dots, c_k$  是该产生式文法符号的属性

■  $b$  是综合属性(synthesized attribute): 如果  $b$  是  $A$  的属性,  
 $c_1, c_2, \dots, c_k$  是产生式右部文法符号的属性或  $A$  的其它属性

■  $b$  是继承属性(inherited attribute): 如果  $b$  是右部某文法符  
号  $X$  的属性



# 例题 1

下面是产生字母表  $\Sigma = \{0, 1, 2\}$  上数字串的一个文法:

$$S \rightarrow D S D \mid 2$$

$$D \rightarrow 0 \mid 1$$

写一个语法制导定义, 判断它接受的句子是否为回文数

$$S' \rightarrow S$$

$$S \rightarrow D_1 S_1 D_2$$

$$S \rightarrow 2$$

$$D \rightarrow 0$$

$$D \rightarrow 1$$

`print(S.val)`

$$S.val = (D_1.val == D_2.val) \text{ and } S_1.val$$

$$S.val = true$$

$$D.val = 0$$

$$D.val = 1$$

对  $S$  和  $D$  加下标

以区分同类语法  
结构的不同实例

$S.val$ :  $S$  对应的串是否是回文数

$D.val$ :  $D$  对应的串的数值

各文法符号的属性均是综合属性的语法制导定义—— $S$  属性定义



# 简单计算器的语法制导定义

产生式	语义规则
$L \rightarrow E \mathbf{n}$ <span>换行标记</span>	$\text{print}(E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit}.\text{lexval}$

$L$ 的匿名属性

由词法  
分析给出

各文法符号的属性均是**综合属性**的语法制导定义—— **$S$  属性定义**

参见: [bison-examples.tar.gz](http://bison-examples.tar.gz) 中的 `config/expr1.y`, `expr.lex`

张昱:《编译原理和技术》语法制导的翻译

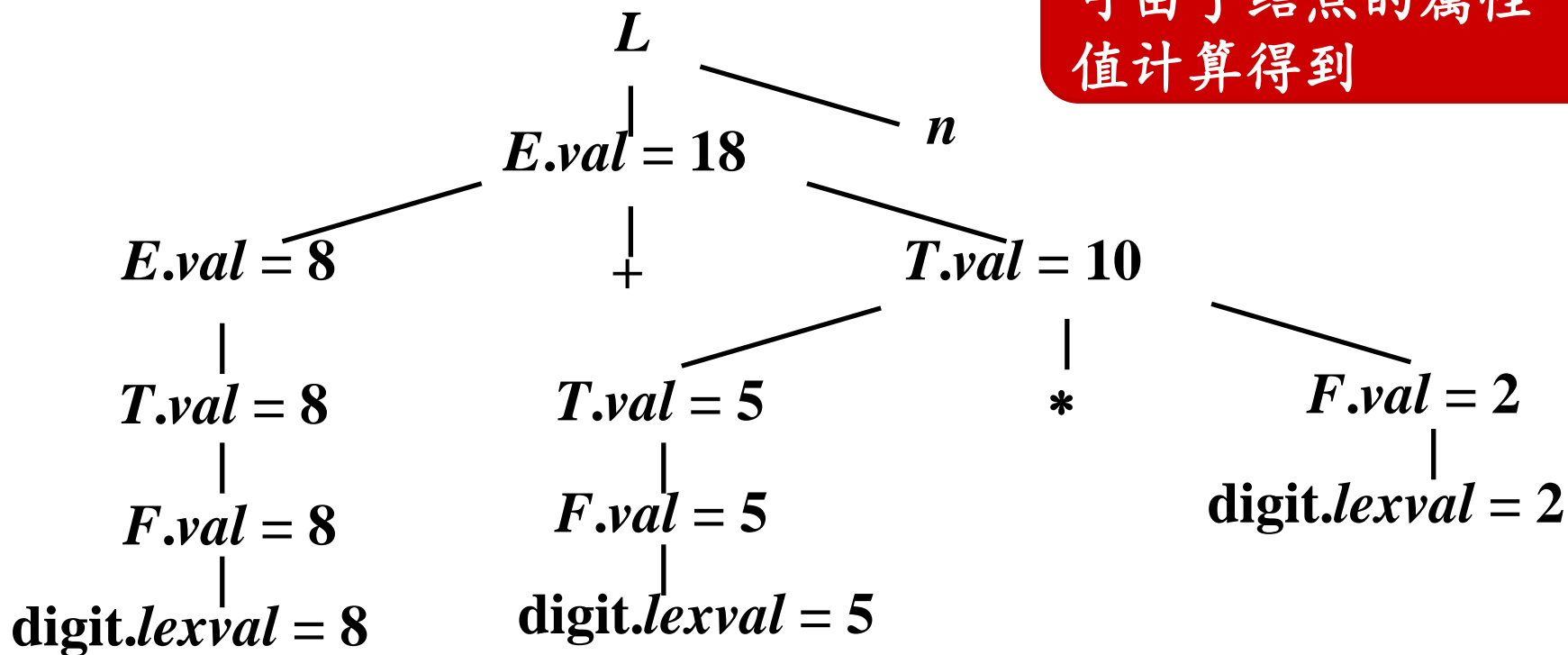


# 注释分析树 (annotated parse tree)

## □ 结点的属性值都标注出来的分析树

$8+5*2$  n (n为换行符)的注释分析树

结点的综合属性值  
可由子结点的属性  
值计算得到







# S属性定义的局限

## □ 考虑消除左递归后，表达式语言的 *LL* 文法

产生式	语义规则
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

消除  
左递归



产生式	语义规则
$T \rightarrow F W$	$T.val = F.val ?$
$W \rightarrow * F W_1$	$W.val = ? * F.val$
$W \rightarrow \epsilon$	
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

- $T$  对应的项中，第1个运算分量是  $F$ ，而运算符  $*$  和其第2个运算分量在  $W$  的子结构中
- $W \rightarrow * F W_1$ ：在分析  $W$  期间， $*$  的左运算分量不在  $W$  的子结构中



# 左递归的消除引起继承属性

- 继承属性(inherited attribute): 如果 $b$ 是右部某文法符号 $X$ 的属性,  $c_1, c_2, \dots, c_k$  是产生式右部文法符号的属性或 $A$ 的其它属性
- 为 $W$ 引入继承属性 $i$ , 继承运算符 $*$ 的左运算分量的属性为 $W$ 引入综合属性 $s$ , 它表示最终的计算结果

继承 $W$ 左边的文法符号 $F$ 的属性

产生式	语义规则
$T \rightarrow FW$	$W.i = F.val; T.val = W.s$
$W \rightarrow *FW_1$	$W_1.i = W.i * F.val; W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$
...	...

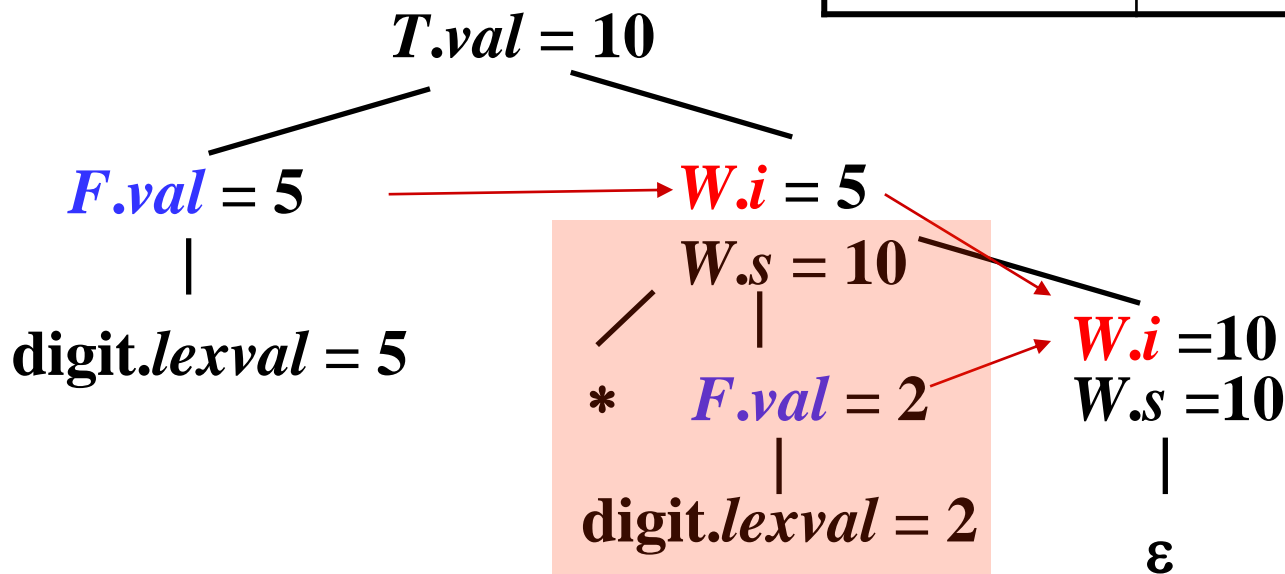


# 含继承属性的注释分析树

□  $5 * 2 \ n$  ( $n$ 为换行符)的注释分析树

□ 不能自下而上由子结点的属性值计算出父结点的属性值

产生式	语义规则
$T \rightarrow FW$	$W.i = F.val ; T.val = W.s$
$W \rightarrow * FW_1$	$W_1.i = W.i * F.val ; W.s = W_1.s$
$W \rightarrow \epsilon$	$W.s = W.i$
...	...





# 属性依赖图(dependence graph)

## □ 属性依赖图

- 描述分析树中结点的属性间依赖关系的有向图
- 顶点为属性：对应分析树中每个文法符号 $X$ 的每个属性 $a$
- 弧为属性间依赖关系：如果属性 $X.a$ 的值依赖于属性 $Y.b$ 的值，则存在从 $Y.b$ 的顶点指向 $X.a$ 的顶点的弧



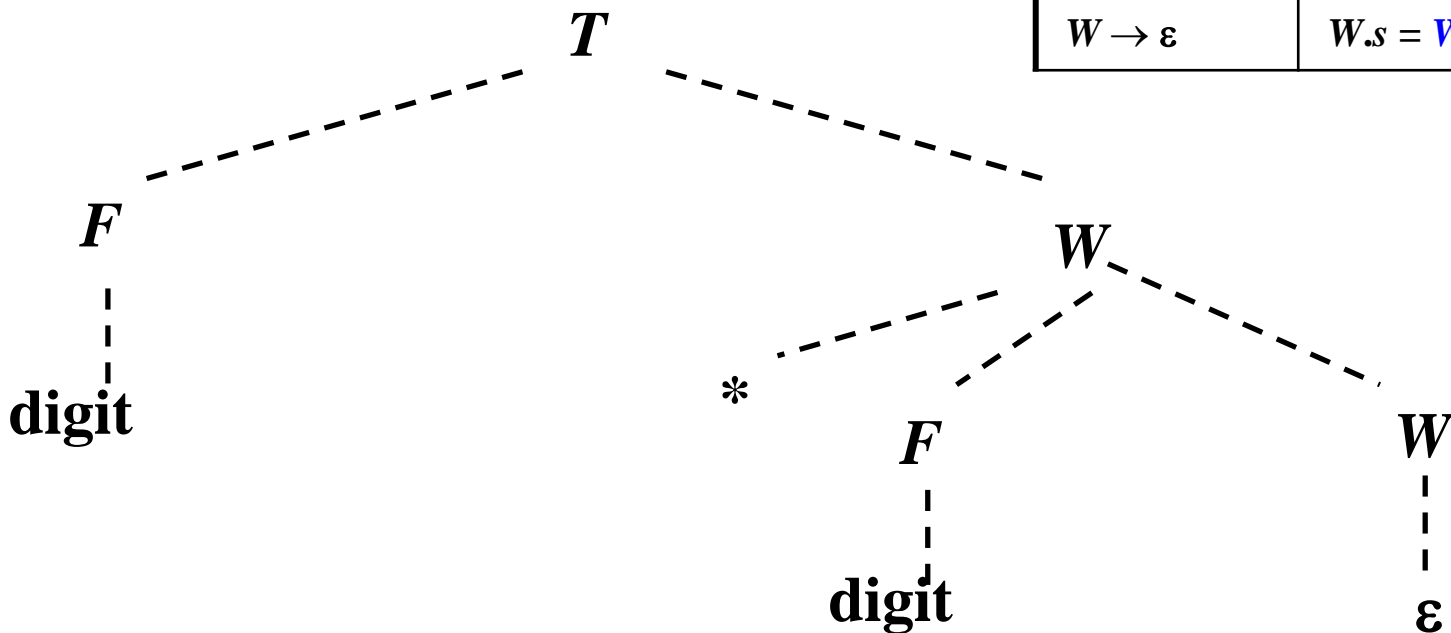
# 属性依赖图(dependence graph)

## □ 属性依赖图

■ 描述分析树中结点的属性间依赖关系的有向图

## □ $5*2\ n$ ( $n$ 为换行符)

分析树 (虚线)



产生式	语义规则
$T \rightarrow FW$	$W.i = F.val ; T.val = W.s$
$W \rightarrow *FW_1$	$W_1.i = W.i * F.val ; W.s = W_1.s$
$W \rightarrow \epsilon$	$W.s = W.i$



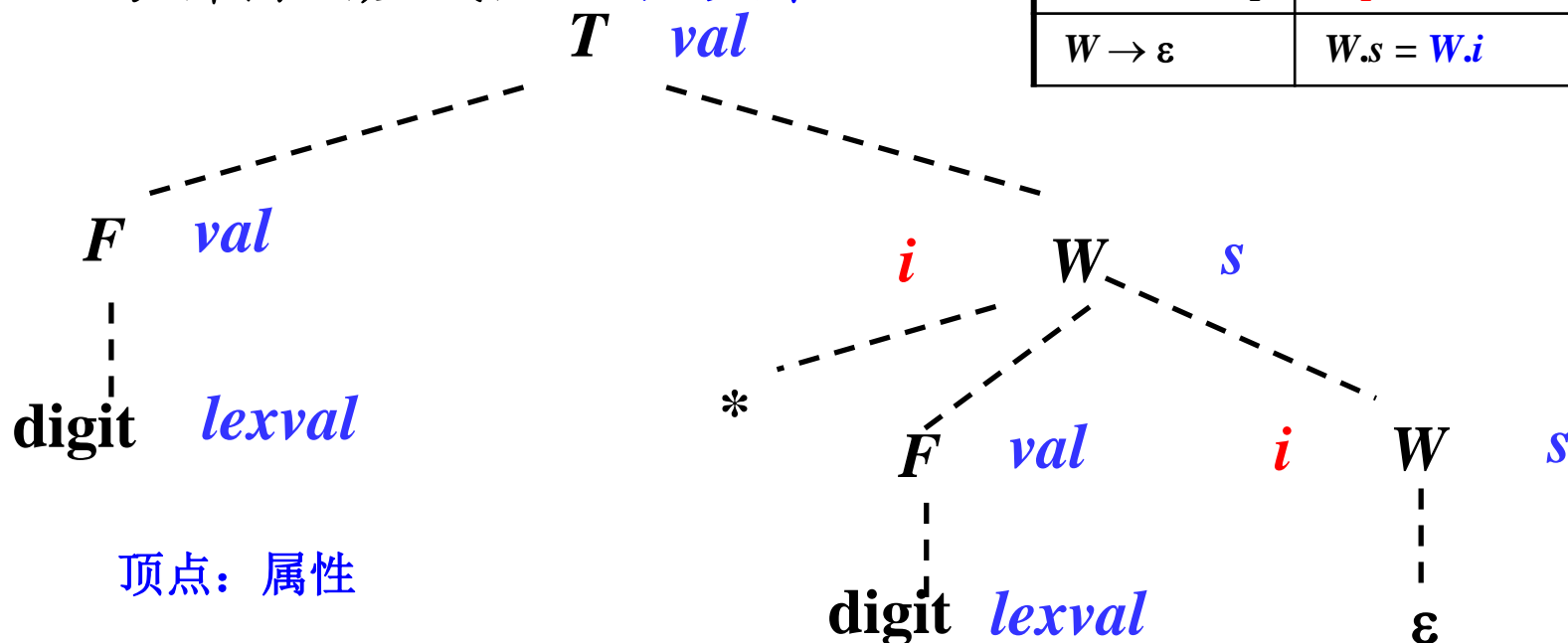
# 属性依赖图(dependence graph)

## □ 属性依赖图

■ 描述分析树中结点的属性间依赖关系的有向图

## □ $5*2\ n$ ( $n$ 为换行符)

分析树（虚线）的依赖图



产生式	语义规则
$T \rightarrow FW$	$W.i = F.val ; T.val = W.s$
$W \rightarrow *FW_1$	$W_1.i = W.i * F.val ; W.s = W_1.s$
$W \rightarrow \epsilon$	$W.s = W.i$

顶点：属性



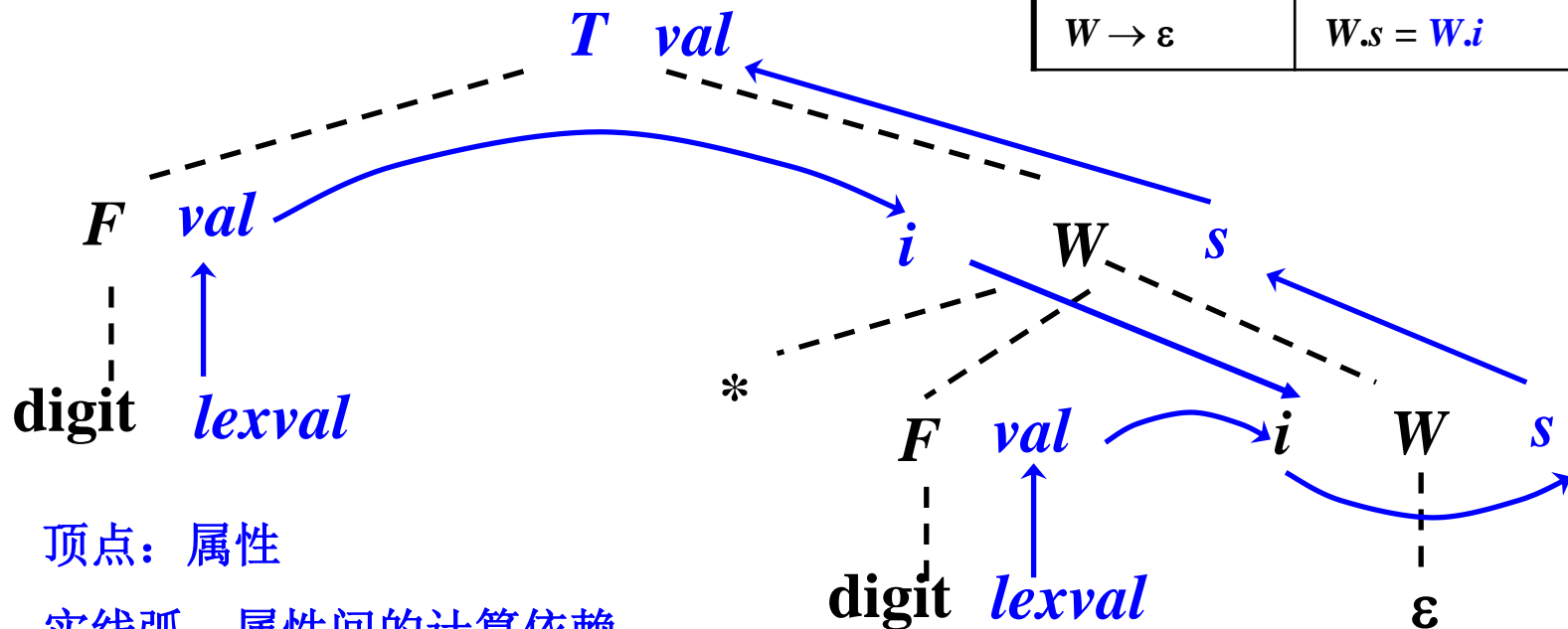
# 属性依赖图(dependence graph)

## □ 属性依赖图

■ 描述分析树中结点的属性间依赖关系的有向图

## □ 5\*2 n (n为换行符)

分析树（虚线）的依赖图



产生式	语义规则
$T \rightarrow FW$	$W.i = F.val ; T.val = W.s$
$W \rightarrow *FW_1$	$W_1.i = W.i * F.val ; W.s = W_1.s$
$W \rightarrow \epsilon$	$W.s = W.i$

顶点：属性

实线弧：属性间的计算依赖



## ■ 描述分析树中结点的属性间依赖关系的有向图

### 分析树（虚线）的依赖图（实线）

**拓扑排序(topological sort)**

是DAG的结点的一种排序 $m_1, \dots, m_k$ , 若有 $m_i$ 到 $m_j$ 的边, 则在排序中 $m_i$ 生于 $m_j$ 之前

张昱：《编译原理和技术》语法制导的翻译





# 属性计算次序

## ■ 属性计算次序

- 1) 构造输入的分析树
- 2) 构造属性依赖图
- 3) 对结点进行拓扑排序
- 4) 按拓扑排序的次序计算属性



# $S$ 属性定义和 $L$ 属性定义

## □ $S$ 属性定义

仅使用综合属性的语法制导定义

## □ $L$ 属性定义 (属性信息自左向右流动)

如果每个产生式  $A \rightarrow X_1 \dots X_{j-1} X_j \dots X_n$  的每条语义规则计算的属性是  $A$  的综合属性, 或者是  $X_j$  的继承属性, 但它仅依赖:

- 该产生式中  $X_j$  左边符号  $X_1, X_2, \dots, X_{j-1}$  的属性;
- $A$  的继承属性

可以按边分析边翻译的方式计算继承属性

## □ $S$ 属性定义是 $L$ 属性定义



# 语义规则的计算方法

## □ 分析树方法

刚才介绍的方法，动态确定计算次序，效率低

——概念上的一般方法

## □ 基于规则的方法

（编译器实现者）静态确定（编译器设计者提供的）语义规则的  
计算次序

——适用于手工构造的方法

## □ 忽略规则的方法

（编译器实现者）事先确定属性的计算策略（如边分析边计算），  
（编译器设计者提供的）语义规则必须符合所选分析方法的限制

——适用于自动生成的方法



# L 属性定义的另一个例子

int id, id, id

产 生 式	语 义 规 则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow \text{int}$	$T.type = \text{integer}$
$T \rightarrow \text{real}$	$T.type = \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in = L.in;$ $\text{addType}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{addType}(\text{id.entry}, L.in)$

*type* – *T*的综合属性

*in* – *L*的继承属性，把声明的类型传递给标识符列表

*addType* – 把类型信息加到符号表中的标识符条目里

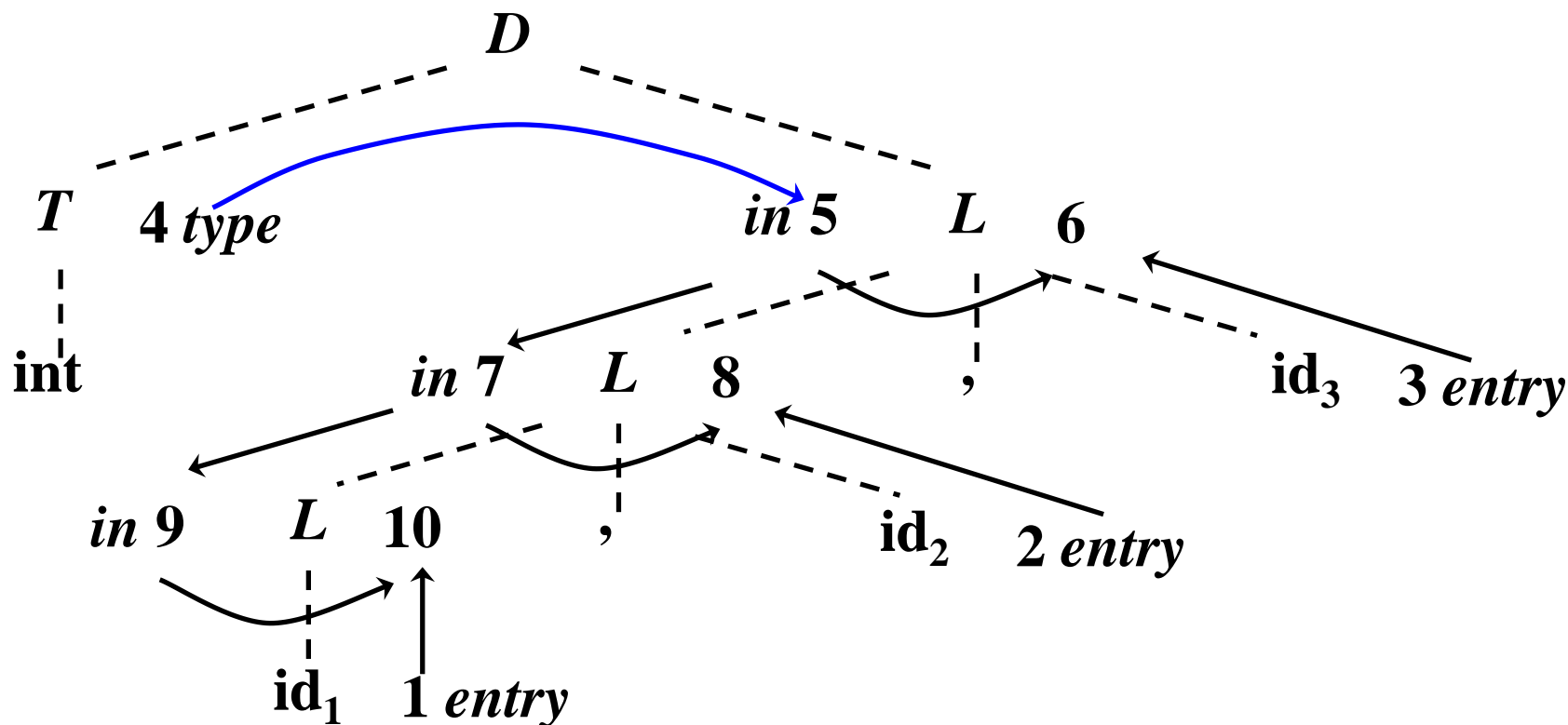


# 属性依赖图(dependence graph)

$\text{int id}_1, \text{id}_2, \text{id}_3$

分析树（虚线）的依赖图（实线）

$D \rightarrow TL \quad L.in = T.type$



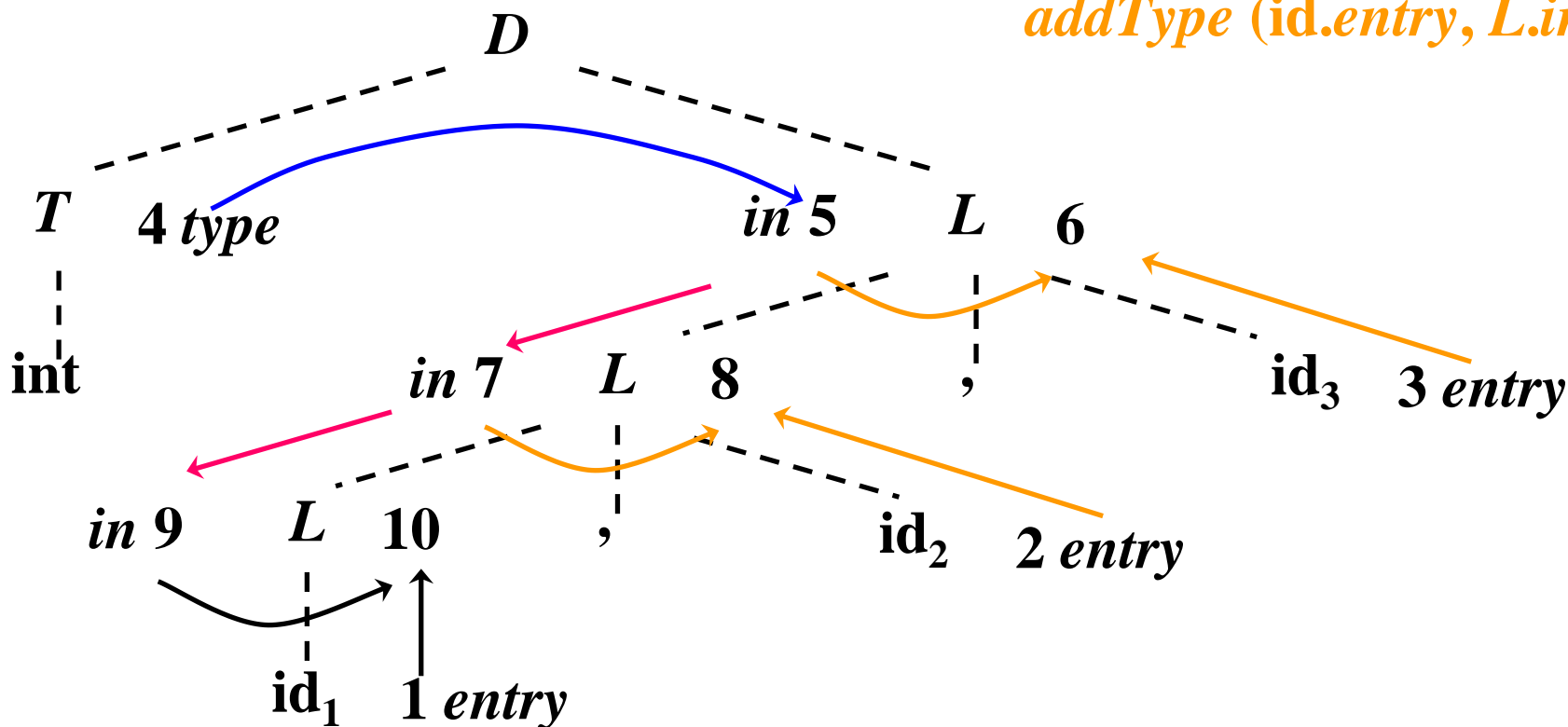


# 属性依赖图(dependence graph)

int id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub>

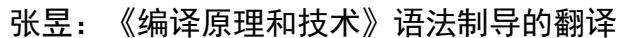
分析树（虚线）的依赖图（实线）

$L \rightarrow L_1, id$   $L_1.in = L.in;$   
 $addType(id.entry, L.in)$





### 分析树（虚线）的依赖图（实线）

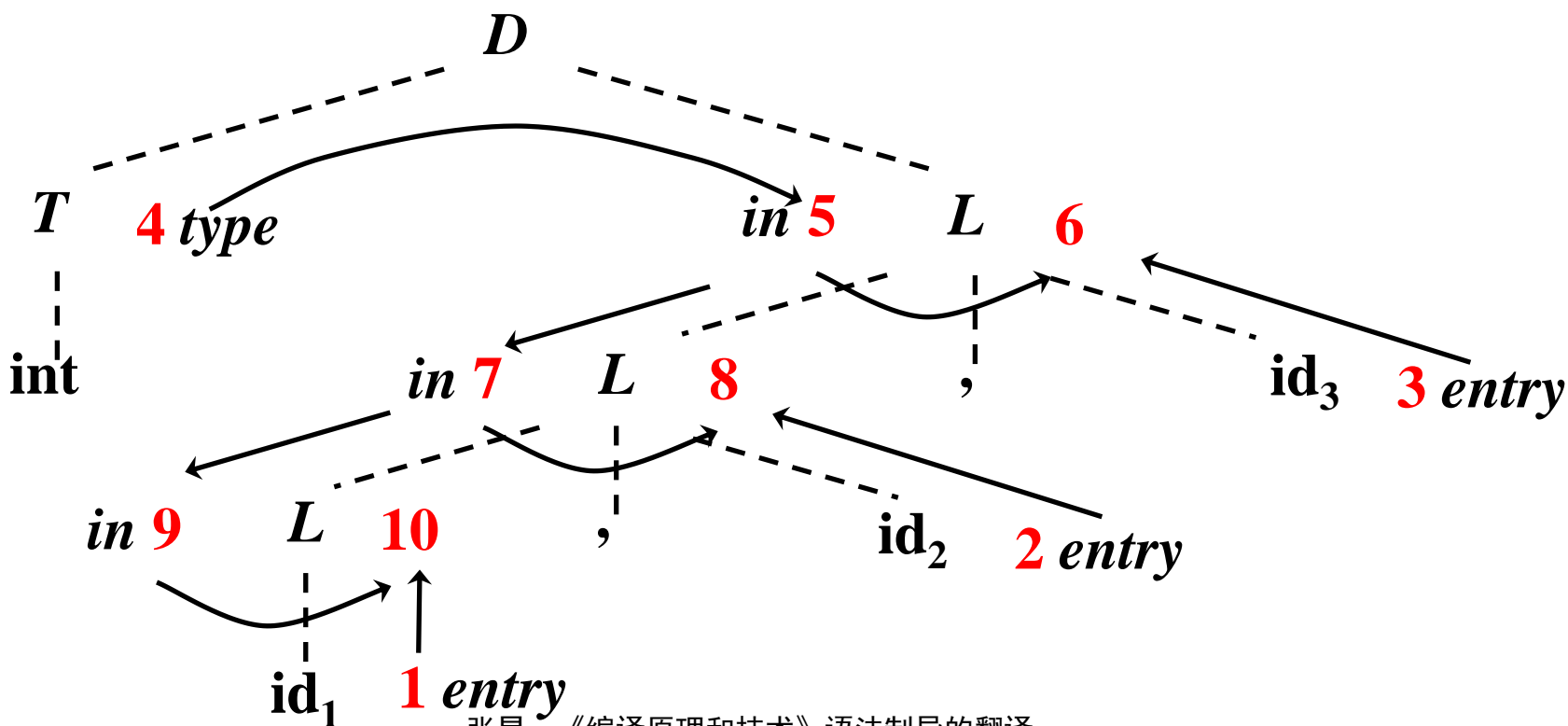




# 属性计算次序

- 拓扑排序(topological sort): 是DAG的结点的一种排序  $m_1, \dots, m_k$ , 若有  $m_i$  到  $m_j$  的边, 则在排序中  $m_i$  先于  $m_j$

例 1, 2, 3, 4, 5, 6, 7, 8, 9, 10







## 4.2 语法树及其构造

- 语法树
- 语法树的构造（文法对构造的影响）
  - 语法制导定义 vs. 翻译方案
  - 自上而下计算 vs. 自下而上计算



# 语法树(syntax tree)

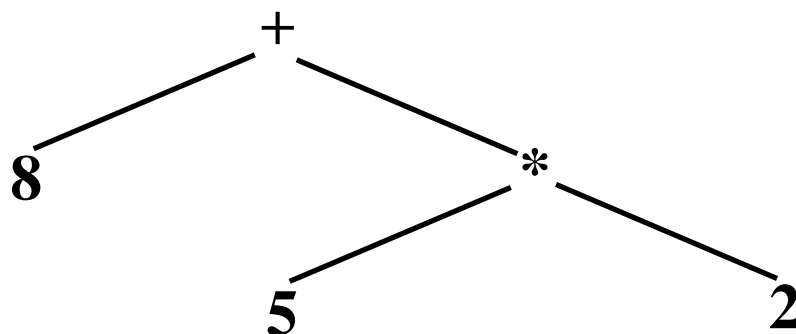
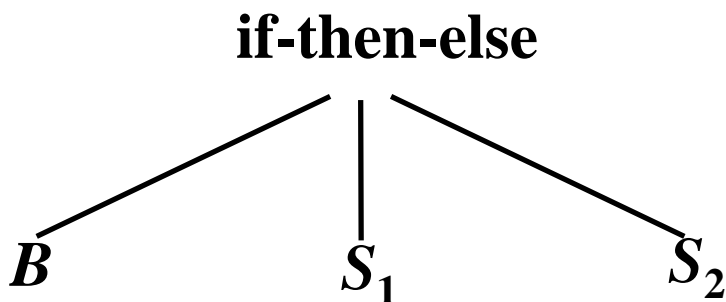
## □ 语法树是分析树的浓缩表示

每个结点表示一个语法构造，算符和关键字是语法树中的内部结点

举例：

if  $B$  then  $S_1$  else  $S_2$

$8 + 5 * 2$



语法制导翻译可以基于分析树，也可以基于语法树



# 语法制导定义: 构造语法树

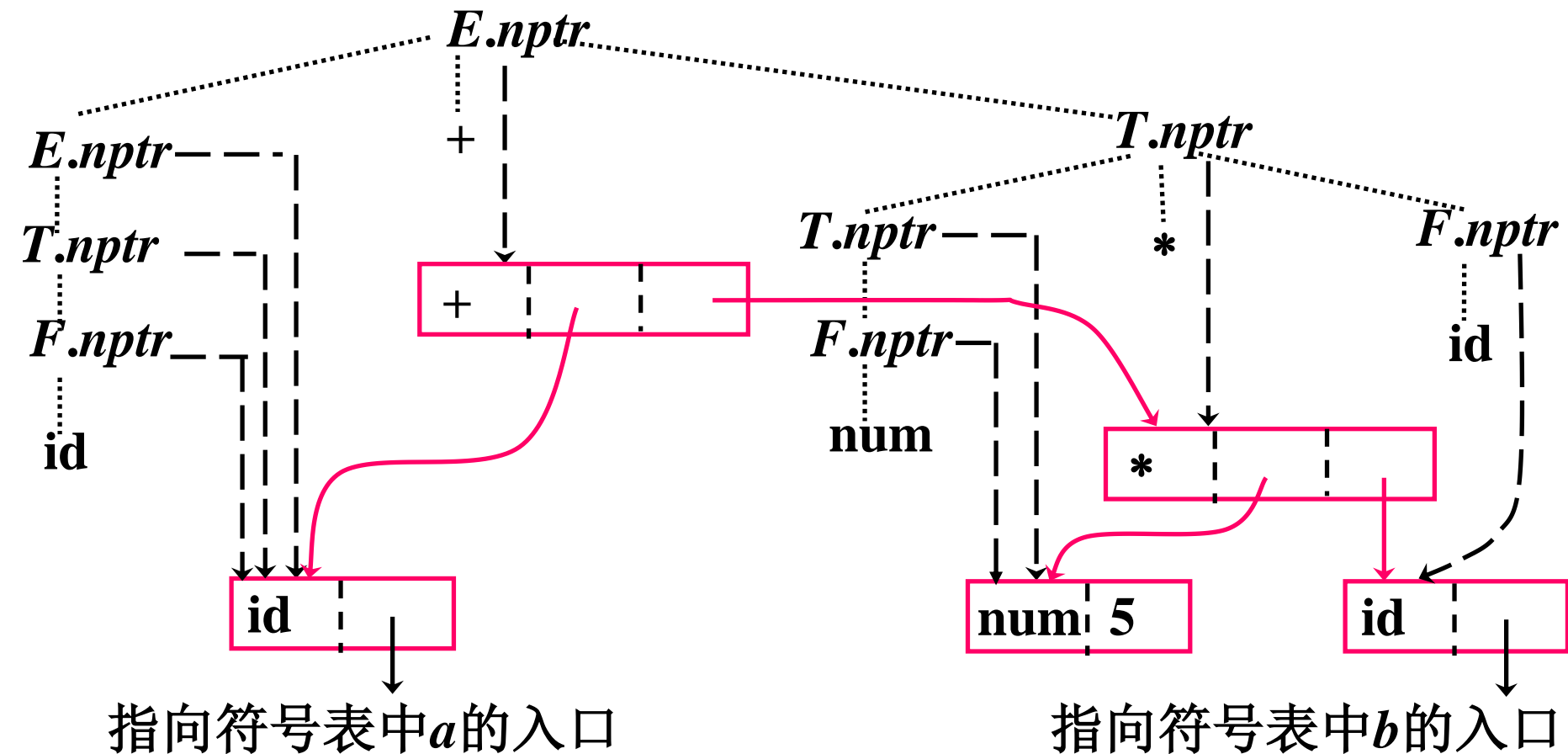
产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr = mkNode( '+', E_1.nptr, T.nptr )$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mkNode( '*', T_1.nptr, F.nptr )$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow id$	$F.nptr = mkLeaf( id, id.entry )$
$F \rightarrow num$	$F.nptr = mkLeaf( num, num.val )$

参见: [bison-examples.tar.gz](http://bison-examples.tar.gz) 中的 `config/asgn2ast.y`, `asgn.lex`



# 语法树的构造

## $a+5*b$ 的语法树的构造





# 翻译方案

## □ 构造语法树的翻译方案（左递归文法）

语义动作，  
置于花括号中

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

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

$T \rightarrow T_1 * F \quad \{T.nptr = mkNode( '*', T_1.nptr, F.nptr ) \}$

$T \rightarrow F \quad \{T.nptr = F.nptr \}$

$F \rightarrow (E) \quad \{F.nptr = E.nptr \}$

$F \rightarrow id \quad \{F.nptr = mkLeaf( id, id.entry ) \}$

$F \rightarrow num \quad \{F.nptr = mkLeaf( num, num.val ) \}$

综合属性的计算规则置于产生式右部的右边，表示识别出右部后计算



# 左递归的消除引起继承属性

## 表达式语言的 $LL$ 文法

产生式	语义规则
$E \rightarrow T R$	$R.i = T.nptr ; E.nptr = R.s$
$R \rightarrow + T R_1$	$R_1.i = mkNode ( '+', R.i, T.nptr ); R.s = R_1.s$
$R \rightarrow \varepsilon$	$R.s = R.i$
$T \rightarrow F W$	$W.i = F.nptr ; T.nptr = W.s$
$W \rightarrow * F W_1$	$W_1.i = mkNode ( '*', W.i, F.nptr ); W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$
...	...



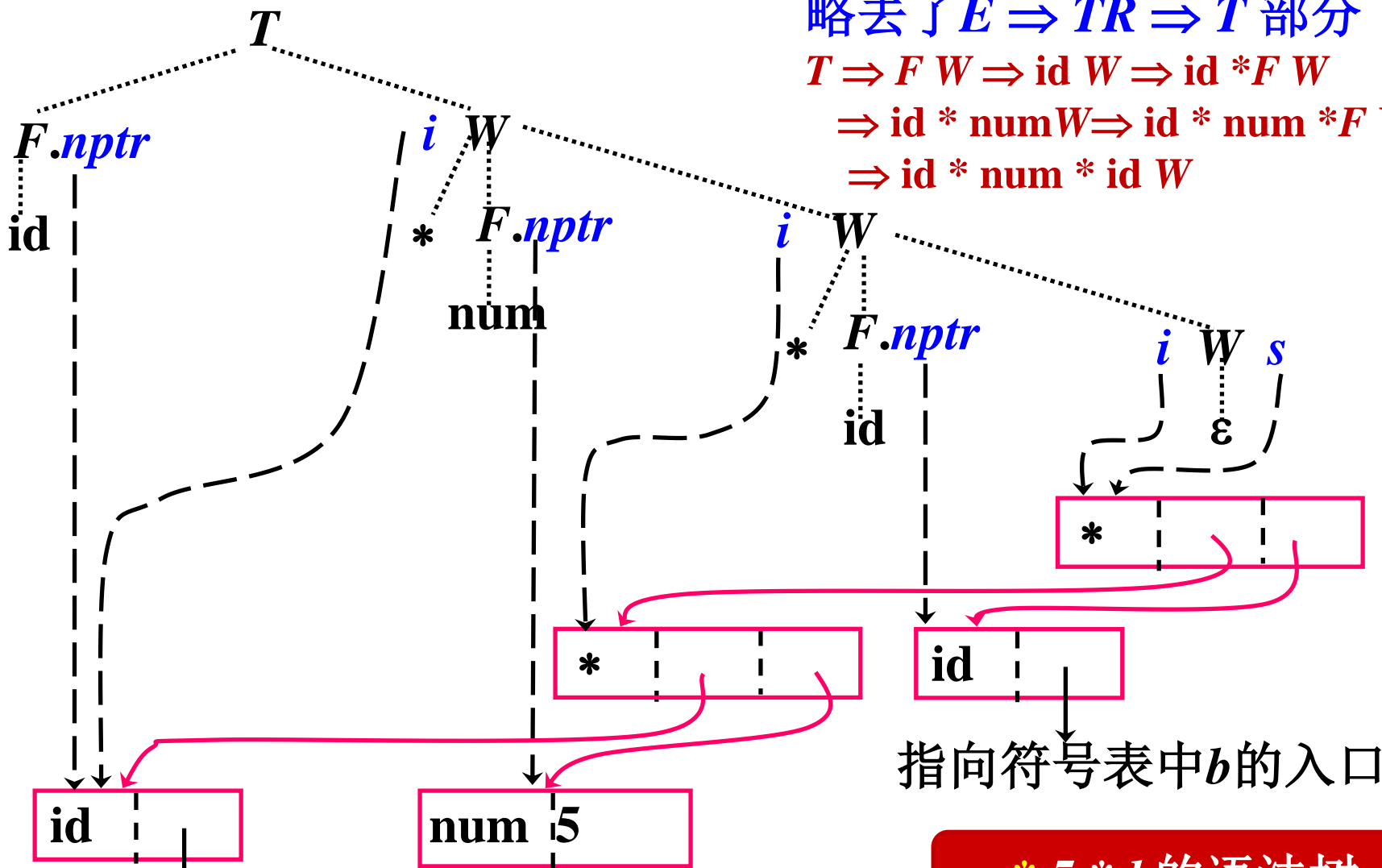
# 语法树的构造 (LL文法)

略去了  $E \Rightarrow TR \Rightarrow T$  部分

$T \Rightarrow F W \Rightarrow \text{id } W \Rightarrow \text{id } * F W$

$\Rightarrow \text{id } * \text{num } W \Rightarrow \text{id } * \text{num } * F W$

$\Rightarrow \text{id } * \text{num } * \text{id } W$



指向符号表中  $b$  的入口

指向符号表中  $a$  的入口

$a * 5 * b$  的语法树



# 翻译方案

$$E \rightarrow T \quad \{R.i = T.nptr\}$$
$$R \quad \{E.nptr = R.s\}$$
$$T + T + T + \dots$$
$$R \rightarrow +$$
$$T \quad \{R_1.i = mkNode ( '+', R.i, T.nptr )\}$$
$$R_1 \quad \{R.s = R_1.s\}$$
$$R \rightarrow \varepsilon \quad \{R.s = R.i\}$$
$$T \rightarrow F$$
$$W \quad \{W.i = F.nptr\}$$
$$\quad \{T.nptr = W.s\}$$
$$W \rightarrow *$$
$$F \quad \{W_1.i = mkNode ( '*', W.i, F.nptr )\}$$
$$W_1 \quad \{W.s = W_1.s\}$$
$$W \rightarrow \varepsilon \quad \{W.s = W.i\}$$

继承属性的计算嵌在产生式右部的某文法符号之前，表示在分析该文法符号之前计算

$F$  的产生式部分不再给出





# Lab: ParseTree => AST

## □ AST的定义

### ■ Node

#### Public Member Functions

```
virtual void accept(Visitor &visitor) = 0;
```

#### Public Attributes

```
Position loc;
```

## □ 访问者Visitor

### ■ Visitor

```
struct Node;
```

```
struct InitVal: Node;  
struct Assembly : Node;
```

```
struct GlobalDef : virtual Node;
```

```
struct FuncDef : GlobalDef;  
struct VarDef : Stmt, GlobalDef;
```

```
struct Stmt : virtual Node;
```

```
struct VarDef : Stmt, GlobalDef;  
struct AssignStmt : Stmt;  
struct ReturnStmt : Stmt;  
struct BlockStmt : Stmt;  
struct EmptyStmt : Stmt;  
struct ExprStmt : Stmt;  
struct IfStmt : Stmt;  
struct WhileStmt : Stmt;  
struct BreakStmt : Stmt;  
struct ContinueStmt : Stmt;
```

```
struct Expr : Node;
```

```
struct CondExpr : Expr;
```

```
struct UnaryCondExpr : CondExpr;  
struct BinaryCondExpr : CondExpr;
```

```
struct AddExpr : Expr;
```

```
struct BinaryExpr : AddExpr;  
struct UnaryExpr : AddExpr;  
struct LVal : AddExpr;  
struct Literal : AddExpr;  
struct FuncCallStmt : AddExpr;
```

```
struct FuncParam : Node;
```

```
struct FuncParamList : Node;
```



# Lab: ParseTree => AST

## □ 访问者Visitor

```
class Visitor
{
public:
    virtual void visit(Assembly &node) = 0;
    virtual void visit(FuncDef &node) = 0;
    virtual void visit(BinaryExpr &node) = 0;
    virtual void visit(UnaryExpr &node) = 0;
    virtual void visit(LVal &node) = 0;
    virtual void visit(Literal &node) = 0;
    virtual void visit(ReturnStmt &node) = 0;
    virtual void visit(VarDef &node) = 0;
    virtual void visit(AssignStmt &node) = 0;
    virtual void visit(FuncCallStmt &node) = 0;
    virtual void visit(BlockStmt &node) = 0;
    virtual void visit(EmptyStmt &node) = 0;
    virtual void visit(ExprStmt &node) = 0;
    virtual void visit(FuncParam &node) = 0;
    virtual void visit(FuncFParamList &node) = 0;
    virtual void visit(IfStmt &node) = 0;
    virtual void visit(WhileStmt &node) = 0;
    virtual void visit(BreakStmt &node) = 0;
    virtual void visit(ContinueStmt &node) = 0;
    virtual void visit(UnaryCondExpr &node) = 0;
    virtual void visit(BinaryCondExpr &node) = 0;
    virtual void visit(InitVal &node) = 0;
};
```

```
struct Node;
```

```
struct InitVal: Node;
struct Assembly : Node;
```

```
struct GlobalDef : virtual Node;
```

```
    struct FuncDef : GlobalDef;
    struct VarDef : Stmt, GlobalDef;
```

```
struct Stmt : virtual Node;
```

```
    struct VarDef : Stmt, GlobalDef;
    struct AssignStmt : Stmt;
    struct ReturnStmt : Stmt;
    struct BlockStmt : Stmt;
    struct EmptyStmt : Stmt;
    struct ExprStmt : Stmt;
    struct IfStmt : Stmt;
    struct WhileStmt : Stmt;
    struct BreakStmt : Stmt;
    struct ContinueStmt : Stmt;
```

```
struct Expr : Node;
```

```
    struct CondExpr : Expr;
```

```
        struct UnaryCondExpr : CondExpr;
        struct BinaryCondExpr : CondExpr;
```

```
struct AddExpr : Expr;
```

```
    struct BinaryExpr : AddExpr;
    struct UnaryExpr : AddExpr;
    struct LVal : AddExpr;
    struct Literal : AddExpr;
    struct FuncCallStmt : AddExpr;
```

```
struct FuncParam : Node;
```

```
struct FuncFParamList : Node;
```



## 例题 2

为下面文法写一个语法制导的定义，用S的综合属性 $val$ 给出下面文法中S产生的二进制数的值。

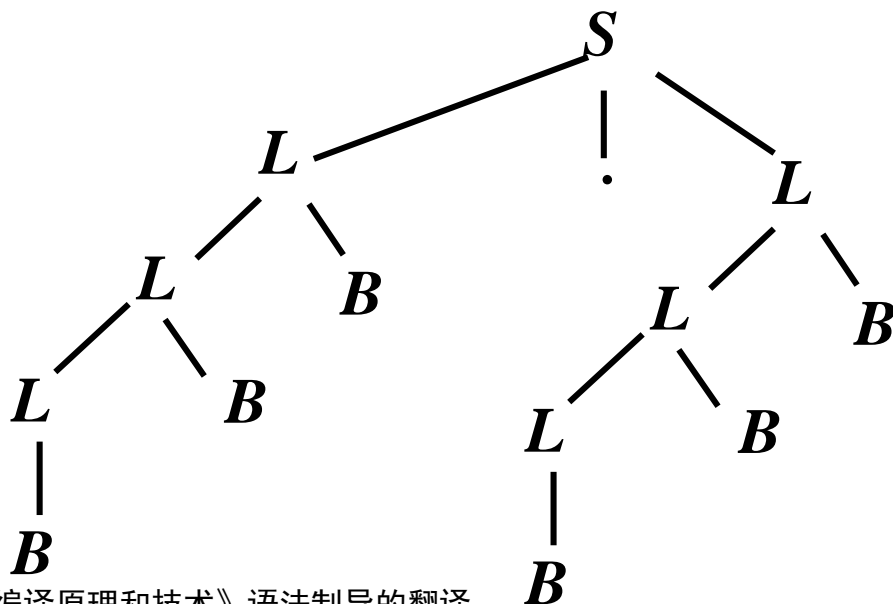
例如，输入101.101时， $S.val = 5.625$ （可以修改文法）

若按 $2^2 + 0 + 2^0 + 2^{-1} + 0 + 2^{-3}$ 来计算，该文法对小数点左边部分的计算不利，因为需要继承属性来确定每个B离开小数点的距离

$S \rightarrow L . L \mid L$

$L \rightarrow L B \mid B$

$B \rightarrow 0 \mid 1$





## 例题 2

为下面文法写一个语法制导的定义，用S的综合属性 $val$ 给出下面文法中S产生的二进制数的值。

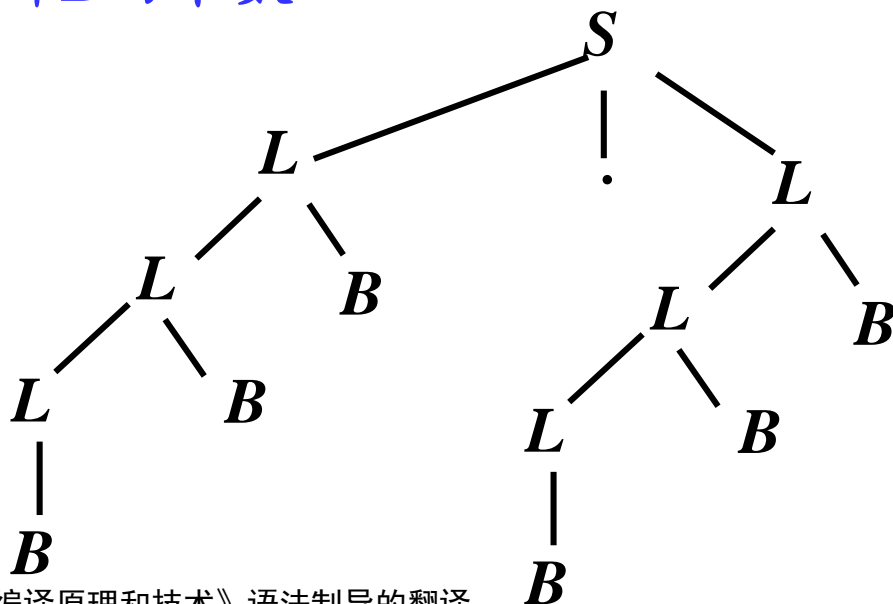
例如，输入101.101时， $S.val = 5.625$ （可以修改文法）

若小数点左边按 $(1 \times 2 + 0) \times 2 + 1$ 计算。该办法不能直接用于小数点右边，需改成 $((1 \times 2 + 0) \times 2 + 1)/2^3$ ，这时需要综合属性来统计B的个数

$S \rightarrow L . L \mid L$

$L \rightarrow L B \mid B$

$B \rightarrow 0 \mid 1$





## 例题 2

为下面文法写一个语法制导的定义，用S的综合属性 $val$ 给出下面文法中S产生的二进制数的值。

例如，输入101.101时， $S.val = 5.625$ （可以修改文法）

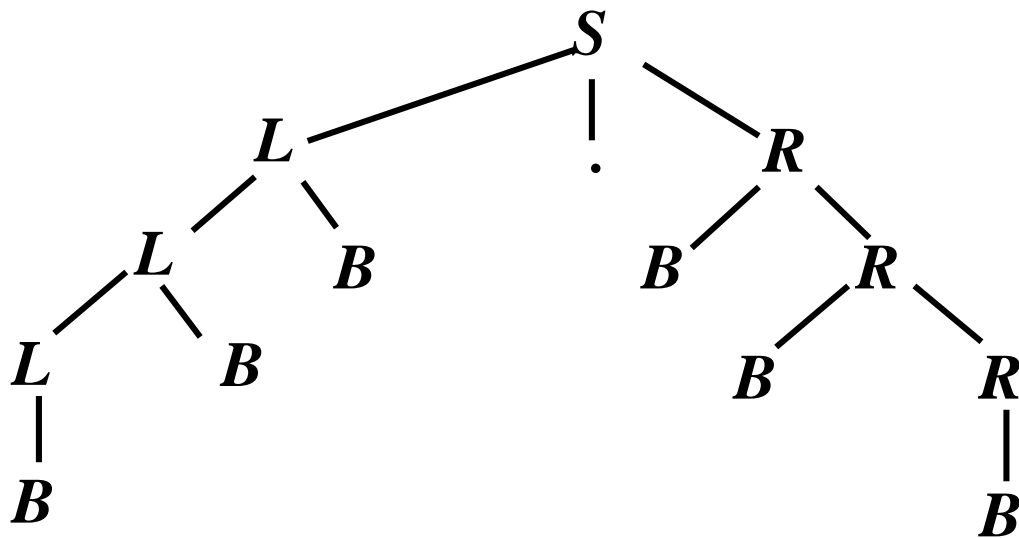
更清楚的办法是将文法改成下面的形式

$S \rightarrow L . R \mid L$

$L \rightarrow L B \mid B$

$R \rightarrow B R \mid B$

$B \rightarrow 0 \mid 1$





## 例题 2

为下面文法写一个语法制导的定义，用S的综合属性 $val$ 给出下面文法中S产生的二进制数的值。

例如，输入101.101时， $S.val = 5.625$ （可以修改文法）

$S \rightarrow L . R$	$S.val = L.val + R.val$
$S \rightarrow L$	$S.val = L.val$
$L \rightarrow L_1 B$	$L.val = L_1.val \times 2 + B.val$
$L \rightarrow B$	$L.val = B.val$
$R \rightarrow B R_1$	$R.val = R_1.val / 2 + B.val / 2$
$R \rightarrow B$	$R.val = B.val / 2$
$B \rightarrow 0$	$B.val = 0$
$B \rightarrow 1$	$B.val = 1$



## 例题 3

给出把中缀表达式翻译成没有冗余括号的中缀表达式的语法制导定义。例如, 因为+和\*是左结合,

$((a * (b + c)) * (d))$  可以重写成  $a * (b + c) * d$

### 两种方法:

- 先把括号都去掉, 然后在必要的地方再加括号
- 去掉表达式中的冗余括号, 保留必要的括号



## 例 题 3

□ 先把括号都去掉，然后在必要的地方再加括号

$S' \rightarrow E \quad \text{print} ( E.code )$

$E \rightarrow E_1 + T$

if  $T.op == plus$  then

$E.code = E_1.code \parallel "+" \parallel "(" \parallel T.code \parallel ")"$

else

$E.code = E_1.code \parallel "+" \parallel T.code;$

$E.op = plus$

$E \rightarrow T \quad E.code = T.code; E.op = T.op$





## 例题 3

- 先把括号都去掉，然后在必要的地方再加括号

$T \rightarrow T_1 * F$

if ( $F.op == plus$ ) or ( $F.op == times$ ) then

if  $T_1.op == plus$  then

$T.code = "(" || T_1.code || ")" || "*" || "(" || F.code || ")"$

else

$T.code = T_1.code || "*" || "(" || F.code || ")"$

else if  $T_1.op = plus$  then

$T.code = "(" || T_1.code || ")" || "*" || F.code$

else

$T.code = T_1.code || "*" || F.code;$

$T.op = times$



## 例 题 3

□ 先把括号都去掉，然后在必要的地方再加括号

$T \rightarrow F$

$T.code = F.code; T.op = F.op$

$F \rightarrow id$

$F.code = id.lexeme; F.op = id$

$F \rightarrow ( E )$

$F.code = E.code; F.op = E.op$



## 例题 3

### □ 去掉表达式中的冗余括号，保留必要的括号

- 给 $E$ ， $T$ 和 $F$ 两个继承属性 $left\_op$ 和 $right\_op$ 分别表示左右两侧算符的优先级
- 给它们一个综合属性 $self\_op$ 表示自身主算符的优先级
- 再给一个综合属性 $code$ 表示没有冗余括号的代码
- 分别用1和2表示加和乘的优先级，用3表示 $id$ 和 $(E)$ 的优先级，用0表示左侧或右侧没有运算对象的情况



## 例题 3

$S' \rightarrow E$

$E. left\_op = 0; E. right\_op = 0; print ( E. code )$

$E \rightarrow E_1 + T$

$E_1. left\_op = E. left\_op; E_1. right\_op = 1;$

$T. left\_op = 1; T. right\_op = E. right\_op;$

$E.code = E_1.code // “+” // T. code ; E. self\_op = 1;$

$E \rightarrow T$

$T. left\_op = E. left\_op;$

$T. right\_op = E. right\_op;$

$E. code = T. code; E. self\_op = T. self\_op$



# 例题 3

$T \rightarrow T_1 * F \quad \dots$

$T \rightarrow F \quad \dots$

$F \rightarrow \text{id}$

$F. \text{code} = id. \text{lexeme}; F. \text{self\_op} = 3$



## 例题 3

$F \rightarrow ( E )$

$E. left\_op = 0; E. right\_op = 0;$

$F. self\_op =$

if ( $F. left\_op < E. self\_op$ ) and

$(E. self\_op \geq F. right\_op)$

then  $E. self\_op$  else 3

$F. code =$

if ( $F. left\_op < E. self\_op$ ) and

$(E. self\_op \geq F. right\_op)$

then  $E. code$  else “(” ||  $E. code$  || “)”



中国科学技术大学  
University of Science and Technology of China

## 4.3 自上而下计算

- 翻译方案
- 预测翻译器的设计
- 用综合属性代替继承属性



# 翻译方案—内嵌不传播的动作

**例** 把有加和减的中缀表达式翻译成后缀表达式

如果输入是 $8+5-2$ ，则输出是 $8\ 5\ +\ 2\ -$

$$E \rightarrow T R$$
$$R \rightarrow \text{addop } T \{\textit{print}(\text{addop.lexeme})\} R_1 \mid \varepsilon$$
$$T \rightarrow \text{num } \{\textit{print}(\text{num.val})\}$$
$$E \Rightarrow T R \Rightarrow \text{num } \{\textit{print}(8)\} R$$
$$\Rightarrow \text{num } \{\textit{print}(8)\} \text{addop } T \{\textit{print}(+)\} R$$
$$\Rightarrow \text{num } \{\textit{print}(8)\} \text{addop num } \{\textit{print}(5)\} \{\textit{print}(+)\} R$$
$$\dots \{\textit{print}(8)\} \{\textit{print}(5)\} \{\textit{print}(+)\} \text{addop } T \{\textit{print}(-)\} R$$
$$\dots \{\textit{print}(8)\} \{\textit{print}(5)\} \{\textit{print}(+)\} \{\textit{print}(2)\} \{\textit{print}(-)\}$$





# 翻译方案—内嵌有信息传播

$$\begin{array}{lll} E \rightarrow T & \{R.i = T.nptr\} & T + T + T + \dots \\ & R & \\ & \{E.nptr = R.s\} & \\ R \rightarrow + & & \\ & T & \\ & \{R_1.i = mkNode ( '+', R.i, T.nptr )\} & \\ & R_1 & \\ & \{R.s = R_1.s\} & \\ R \rightarrow \varepsilon & \{R.s = R.i\} & \\ T \rightarrow F & \{W.i = F.nptr\} & \\ & W & \\ & \{T.nptr = W.s\} & \\ W \rightarrow * & & \\ & F & \\ & \{W_1.i = mkNode ( '*', W.i, F.nptr )\} & \\ & W_1 & \\ & \{W.s = W_1.s\} & \\ W \rightarrow \varepsilon & \{W.s = W.i\} & \end{array}$$

继承属性的计算嵌在产生式右部的某文法符号之前，表示在分析该文法符号之前计算

$F$  的产生式部分不再给出



# 预测翻译器的设计

**方法：**将预测分析器的构造方法推广到翻译方案的实现 (*LL*文法)

**产生式 $R \rightarrow +TR \mid \varepsilon$ 的分析过程**

```
void R( ) {  
    if (lookahead == '+' ) {  
        match ( '+' ); T( ); R( );  
    }  
    else if (lookahead == ')' || lookahead == '$' ) ;  
    else error( );  
}
```



# 预测翻译器的设计

```
syntaxTreeNode * R (syntaxTreeNode * i) {
```

```
//继承属性作为参数,综合属性为返回值
```

```
syntaxTreeNode *nptr, *i1, *s1, *s;
```

```
char addoplexeme;
```

```
if (lookahead == '+' ) {
```

```
    addoplexeme = lexval;
```

```
    match('+'); nptr = T();
```

```
    i1 = mkNode(addoplexeme, i , nptr);
```

```
    s1 = R (i1); s = s1;
```

```
}
```

```
else if (lookahead == ')' || lookahead == '$') s = i;
```

```
else error();
```

```
return s;
```

```
}
```

```
void R() {  
    if (lookahead == '+') {  
        match ('+'); T(); R();  
    }  
    else if (lookahead == ')' || lookahead == '$') ;  
    else error();  
}
```

*R* : *i*, *s*

*T* : *nptr*

*+* : *addoplexeme*



# 非L属性定义

例 Pascal的声明, 如  $m, n : \text{integer}$

$$D \rightarrow L : T \qquad L.in = T.type$$
$$T \rightarrow \text{integer} \mid \text{char} \qquad T.type = \dots$$
$$L \rightarrow L_1, \text{id} \mid \text{id} \qquad L_1.in = L.in, \dots$$

该语法制导定义 **非L属性定义**

信息从右向左流, 归约从左向右, 两者不一致



# 非L属性定义：改写文法

例 Pascal的声明，如  $m, n : \text{integer}$

$D \rightarrow L : T$        $L.in = T.type$  (非L属性定义)

$T \rightarrow \text{integer} \mid \text{char}$        $T.type = \dots$

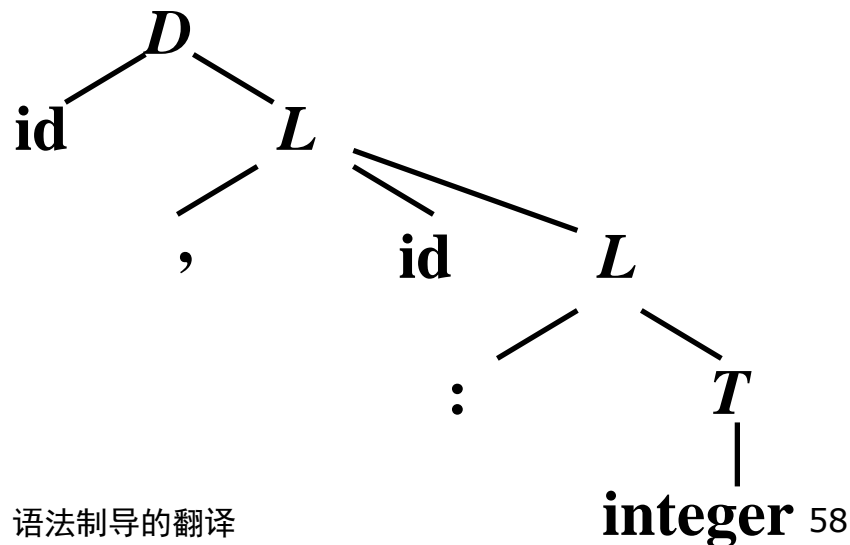
$L \rightarrow L_1, \text{id} \mid \text{id}$        $L_1.in = L.in, \dots$

等所需信息获得后再归约，改成从右向左归约

$D \rightarrow \text{id } L$  (S属性定义)

$L \rightarrow , \text{id } L \mid : T$

$T \rightarrow \text{integer} \mid \text{char}$





# 用综合属性代替继承属性

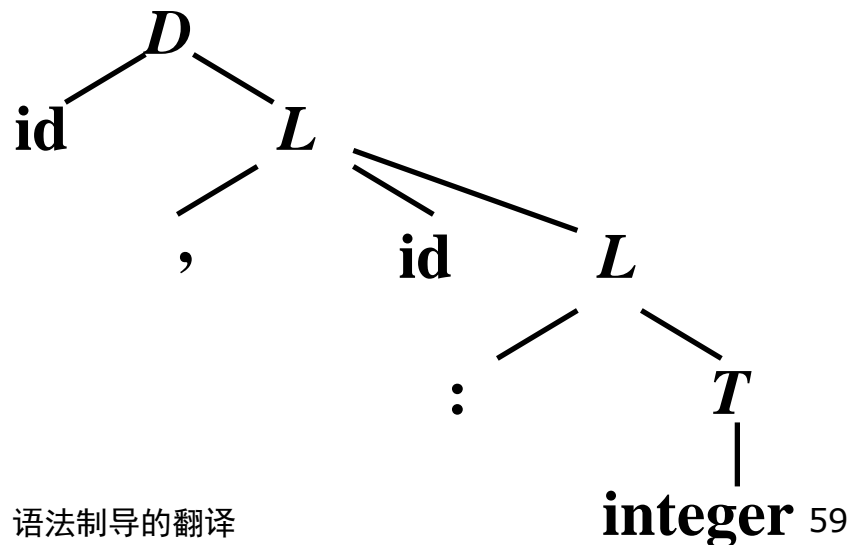
$D \rightarrow \text{id } L \quad \{ \text{addtype}(\text{id. entry}, L.\text{type}) \}$

$L \rightarrow , \text{id } L_1 \quad \{ L.\text{type} = L_1.\text{Type};$   
 $\quad \text{addtype}(\text{id. entry}, L_1.\text{type}) \}$

$L \rightarrow : T \quad \{ L.\text{type} = T.\text{type} \}$

$T \rightarrow \text{integer} \quad \{ T.\text{type} = \text{integer} \}$

$T \rightarrow \text{real} \quad \{ T.\text{type} = \text{real} \}$





## 4.4 自下而上计算

- 综合属性的计算
- 删除翻译方案中嵌入的动作
- 继承属性的计算



## S属性定义举例

### $a+5*b$ 的语法树的构造







# S属性的自下而上计算

## □ 边分析边计算

LR分析器的栈增加一个域来保存综合属性值

*top* →

...	...
<i>Z</i>	<i>Z.z</i>
<i>Y</i>	<i>Y.y</i>
<i>X</i>	<i>X.x</i>
...	...

↑

若产生式 $A \rightarrow XYZ$ 的语义规则是

$$A.a = f(X.x, Y.y, Z.z),$$

那么归约后:

*top* →

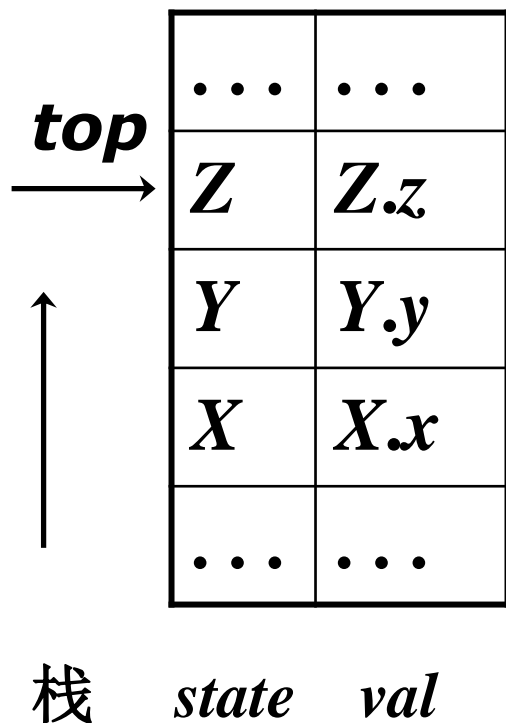
...	...
<i>A</i>	<i>A.a</i>
...	...

栈    *state*    *val*



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



产生式	语义规则
$L \rightarrow E n$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

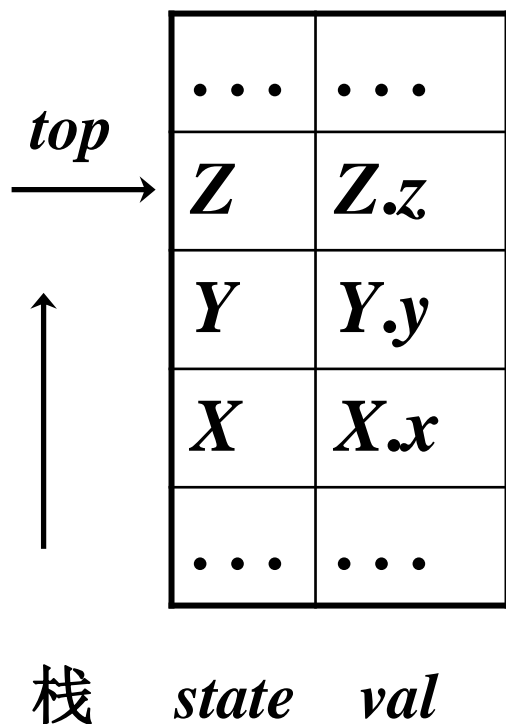
参见: [bison-examples.tar.gz](http://bison-examples.tar.gz) 中的 `config/expr1.y`, `expr.lex`

张昱: 《编译原理和技术》语法制导的翻译



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码

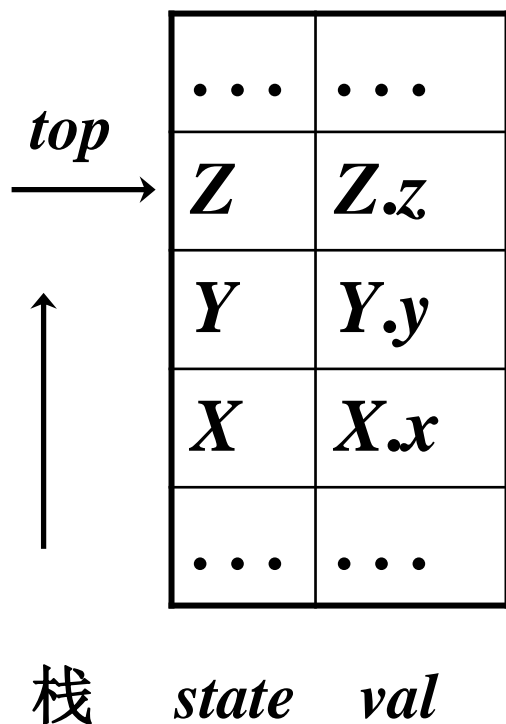


产生式	代码段
$L \rightarrow E n$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



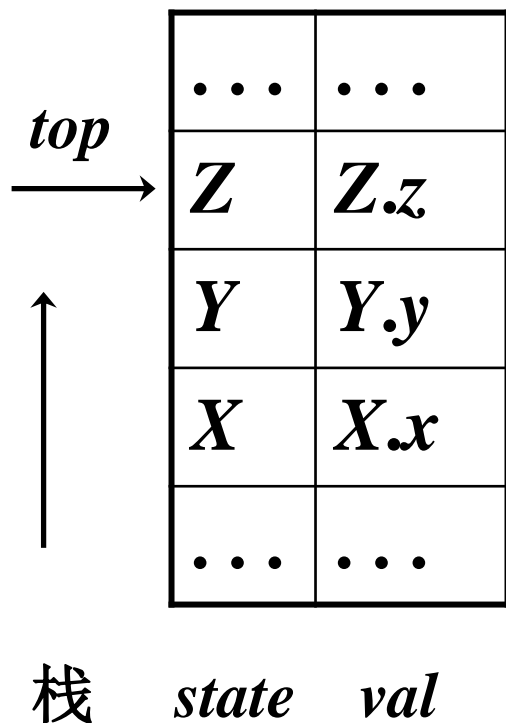
产生式	代码段
$L \rightarrow E n$	$print(val[top-1])$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



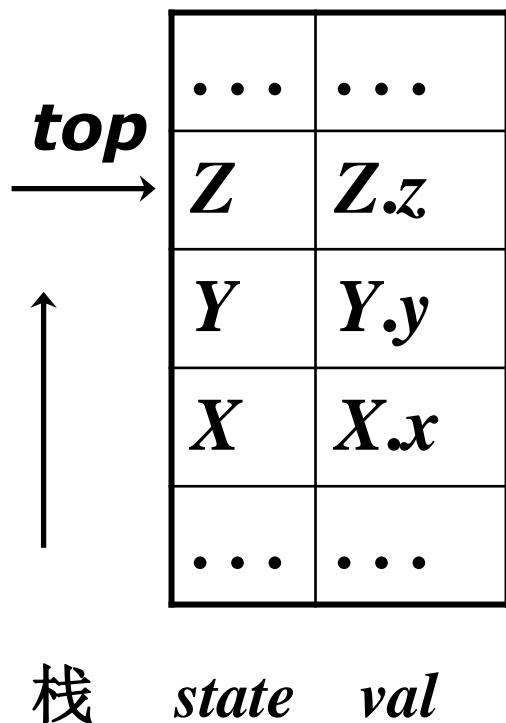
产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



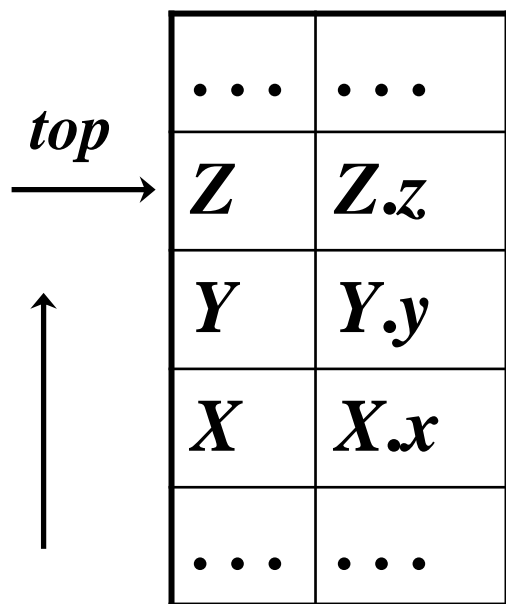
产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



栈      *state*    *val*

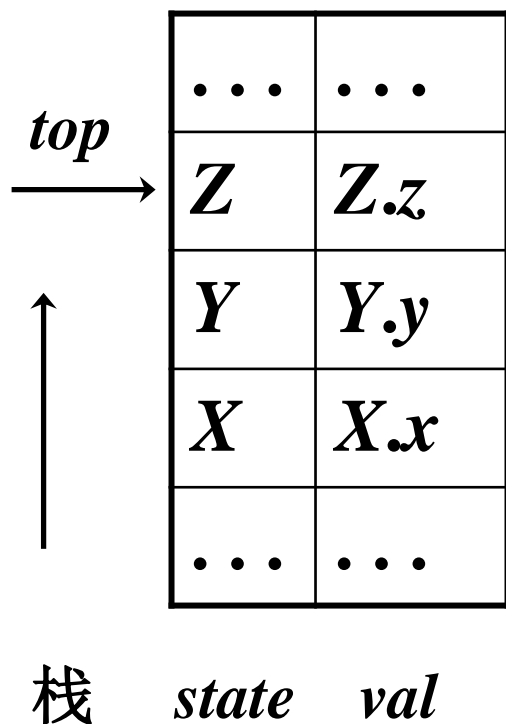
产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[top-2] = val[top-2] \times val[top];$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[top-2] = val[top-2] \times val[top];$
$T \rightarrow F$	
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

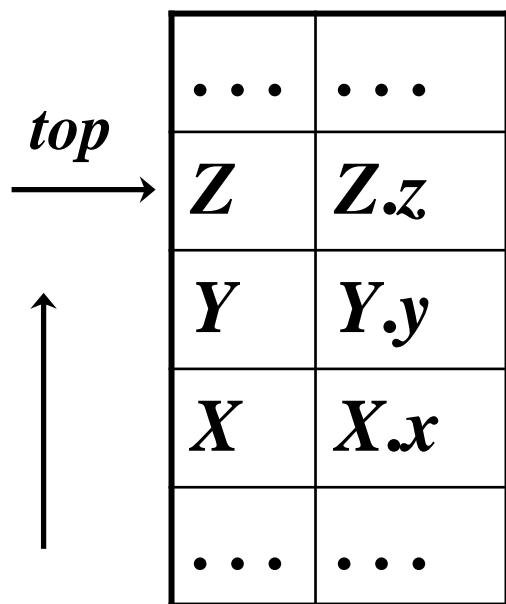
注：栈顶位置指示器top的修改由原来的分析程序在语义动作执行后去做





# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



栈     *state*     *val*

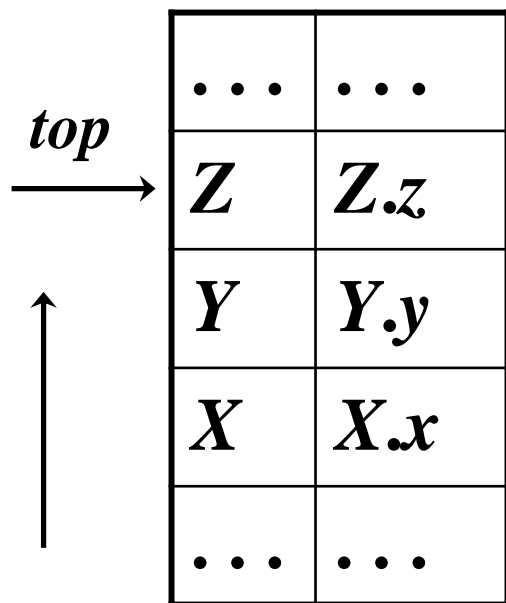
产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[top-2] = val[top-2] \times val[top];$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[top-2] = val[top-1];$
$F \rightarrow digit$	$F.val = digit.lexval$

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# 自下而上的翻译

## 例 简单计算器的语法制导定义改成栈操作代码



栈     $state$      $val$

产生式	代码段
$L \rightarrow E n$	$print(val[top-1]);$
$E \rightarrow E_1 + T$	$val[top-2] = val[top-2] + val[top];$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[top-2] = val[top-2] \times val[top];$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[top-2] = val[top-1];$
$F \rightarrow digit$	

注：栈顶位置指示器 $top$ 的修改由原来的分析程序在语义动作执行后去做



# Bison 举例 bison-examples: config/expr.y

```
%{
#include <stdio.h>
#include <math.h>
%}

%union {
    float val;
}

%token NUMBER
%token PLUS MINUS MULT DIV EXPON
...
%left MINUS PLUS
%left MULT DIV
%right EXPON

%type <val> exp NUMBER
%%
```

各种语义值类型和域  
置于共用体中

声明exp、Number  
的语义值是在val域

```
input : | input line
      ;

...

exp  : NUMBER          { $$ = $1;      }
    | exp PLUS exp     { $$ = $1 + $3; }
    | exp MINUS exp    { $$ = $1 - $3; }
    | exp MULT exp     { $$ = $1 * $3; }
    | exp DIV exp      { $$ = $1 / $3; }
    | MINUS exp %prec MINUS { $$ = -$2; }
    | exp EXPON exp    { $$ = pow($1,$3); }
    | LB exp RB        { $$ = $2;      }
      ;

%%

yyerror(char *message)
{ printf("%s\n",message); }

int main(int argc, char *argv[])
{ yyparse(); return(0); }
```



# $L$ 属性的自下而上计算

## 在自下而上分析的框架中实现 $L$ 属性定义的方法

- 它能实现**任何**基于LL(1)文法的 $L$ 属性定义
- 也能实现**许多**（但不是所有的）基于LR(1)的 $L$ 属性定义



# 删除翻译方案中嵌入的动作

## □ 中缀表达式翻译成后缀表达式

$$E \rightarrow T R$$

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

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

在文法中加入产生 $\varepsilon$ 的标记非终结符，让每个嵌入动作由不同的标记非终结符 $M$ 代表，并把该动作放在产生式 $M \rightarrow \varepsilon$ 的右端 （继承属性 $\Rightarrow$ 综合属性）

$$E \rightarrow T R$$

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

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

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

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

YACC会按这种方法来处理输入的文法，即为嵌入的语义动作引入 $\varepsilon$ 产生式



# L属性的自下而上计算

bison-examples: config/exprL.y

input : ...

```
| input{ lineno ++; printf("Line %d:\t", lineno);} line { printf("*"); } ;
```

\$\$表示LHS符号的语义值, \$1, \$2...依次为RHS中符号的语义值, 本例中line的语义值通过\$3 来引用

src/exprL.tab.c

```
case 4:
/* Line 1806 of yacc.c */
#line 36 "config/exprL.y"
{ printf("*"); }
break;
```

**yyreduce:**

/\* yyn is the number of a rule to reduce with. \*/

...

YY\_REDUCE\_PRINT (yyn);

switch (yyn) { ...

case 3:

/\* Line 1806 of yacc.c \*/

#line 32 "config/exprL.y"

{ lineno ++;

printf("Line %d:\t", lineno);

} break;

产生式编号



中国科学技术大学  
University of Science and Technology of China

## 4.4 自下而上计算

- 综合属性的计算
- 删除翻译方案中嵌入的动作
- 继承属性的计算



# 继承属性在分析栈中

## 情况1 属性位置可预测

例  $\text{int } p, q, r$

$D \rightarrow T \quad \{L.in = T.type\}$

$L$

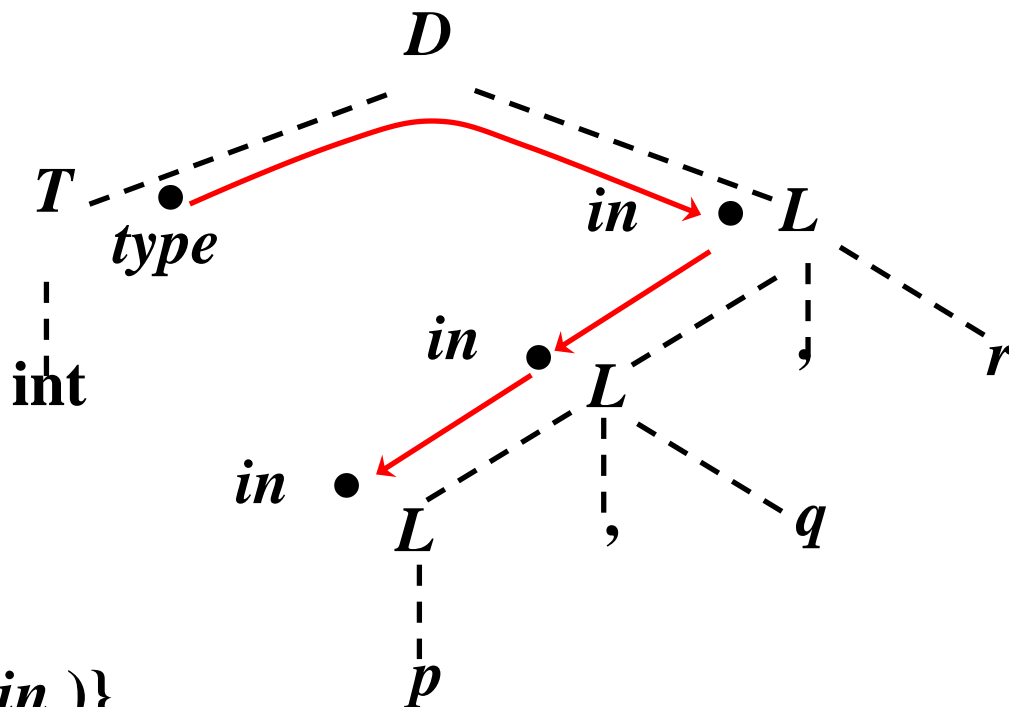
$T \rightarrow \text{int} \quad \{T.type = \text{integer}\}$

$T \rightarrow \text{real} \quad \{T.type = \text{real}\}$

$L \rightarrow \quad \{L_1.in = L.in\}$

$L_1, \text{id} \quad \{\text{addtype}(\text{id.entry}, L.in)\}$

$L \rightarrow \text{id} \quad \{\text{addtype}(\text{id.entry}, L.in)\}$



继承属性值已在分析栈中





# 继承属性在分析栈中

## 情况1 属性位置可预测

例 `int p, q, r`

$D \rightarrow T \quad \{ \cancel{L.in} = \cancel{T.type} \}$   
 $L$

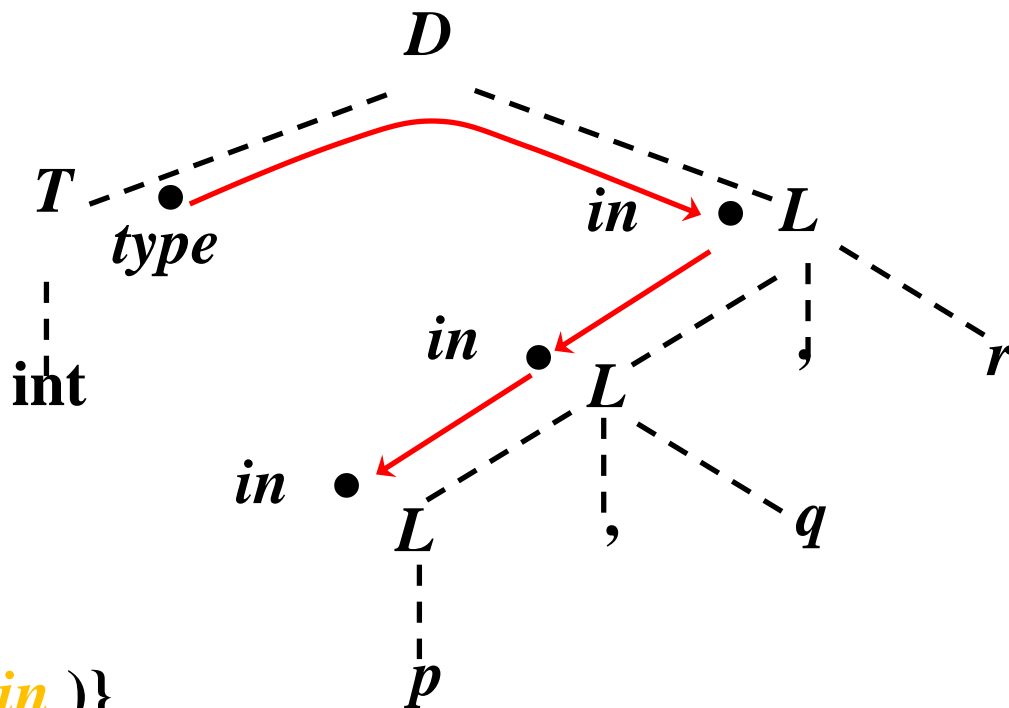
$T \rightarrow \text{int} \quad \{ T.type = integer \}$

$T \rightarrow \text{real} \quad \{ T.type = real \}$

$L \rightarrow \quad \{ \cancel{L_1.in} = \cancel{L.in} \}$

$L_1, \text{id} \quad \{ addtype(\text{id.entry}, L.in) \}$

$L \rightarrow \text{id} \quad \{ addtype(\text{id.entry}, L.in) \}$



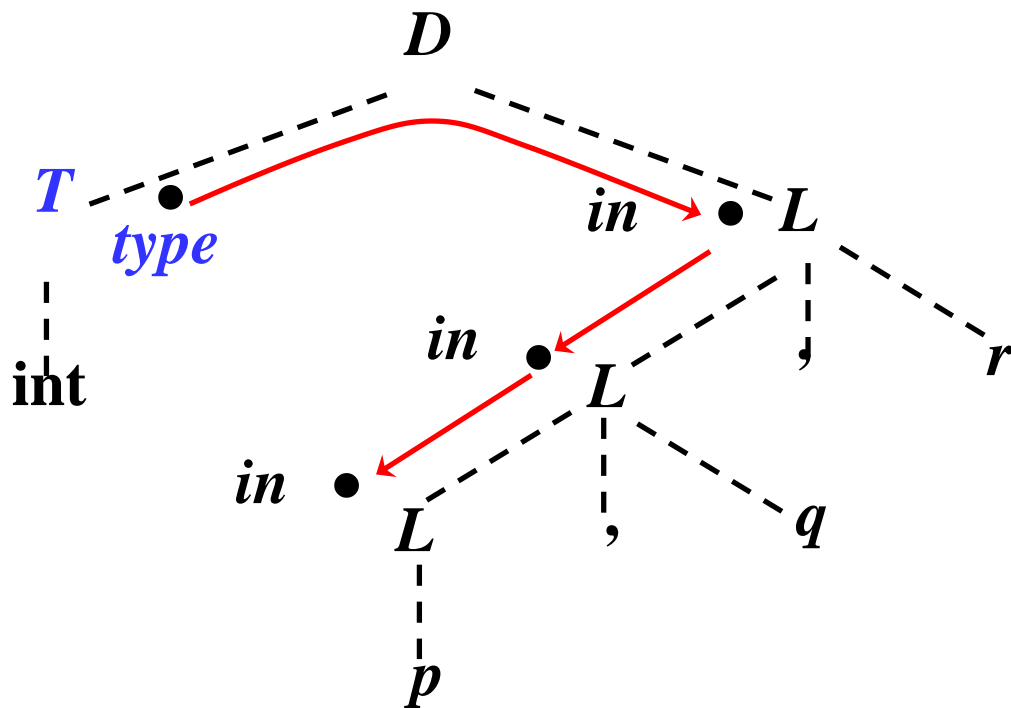
略去继承属性的计算  
引用继承属性的地方改成  
引用其他符号的综合属性



# 继承属性在分析栈中

## 情况1 属性位置可预测

产生式	代码段
$D \rightarrow TL$	
$T \rightarrow \text{int}$	$val[top] = integer$
$T \rightarrow \text{real}$	$val[top] = real$
$L \rightarrow L_1, id$	$addType(val[top],$ $val[top-3]);$
$L \rightarrow id$	$addType(val[top],$ $val[top-1]);$



略去继承属性的计算  
引用继承属性的地方改成  
引用其他符号的综合属性



# YACC中的继承属性定义

## 在内嵌动作代码中设置该文法符号的语义值

bison-examples: config/exprL1.y

line : ...

| NUMBER { ...

指明lineno的语义值  
所在的共用体的域

**\$<val>lineno** = \$1; // val是%union中声明的语义值类型

// **\$<val>\$** = \$1; // 该语义动作代码未指定名字时

...

} **[lineno]**

给内嵌语义动作对应的  
标记非终结符命名

exp EOL { ...

printf("Line %d: %g\n", (int) **\$<val>lineno**, \$3);

...

}



# YACC中的继承属性定义

在内嵌代码中使用存储在栈中任意固定相对位置的语义值

bison-examples: [config/midrule.y](#)

```
exp: a_1 a_2 { $<val>$ = 3; } { $<val>$ = $<val>3 + 1; } a_5
    sum_of_the_five_previous_values
    {
        USE (($1, $2, $<foo>3, $<foo>4, $5));
        printf ("%d\n", $6);
    }
```

```
sum_of_the_five_previous_values:
{
    $$ = $<val>0 + $<val>-1 + $<val>-2 + $<val>-3 + $<val>-4;
}
```

$\$<val>0$ 、 $\$<val>-1$ 、 $\$<val>-2$ 、 $\$<val>-3$ 、 $\$<val>-4$ 分别表示栈中 $a_5$ 、 $\{ \$<val>$ = \$<val>3 + 1; \}$ 、 $\{ \$<val>$ = 3; \}$ 、 $a_2$ 、 $a_1$ 文法符号的语义值



# 继承属性在分析栈中

## 情况2 属性位置不可预测

$$S \rightarrow aAC \quad C.i = A.s$$

$$S \rightarrow bA\textcolor{violet}{B}C \quad C.i = A.s$$

$$C \rightarrow c \quad C.s = g(C.i)$$

*B可能在, 也可能不在A和C之间, C.i的值有2种可能*

□ 增加标记非终结符, 使得位置可以预测

$$S \rightarrow aAC \quad C.i = A.s$$

$$S \rightarrow bAB\textcolor{violet}{M}C \quad \textcolor{violet}{M}.i = A.s; \textcolor{violet}{C}.i = \textcolor{violet}{M}.s$$

$$C \rightarrow c \quad C.s = g(C.i)$$

$$\textcolor{violet}{M} \rightarrow \varepsilon \quad \textcolor{violet}{M}.s = \textcolor{violet}{M}.i$$

继承属性值  
已在分析栈中



# 模拟继承属性的计算

## □ 继承属性是综合属性的函数

$$S \rightarrow aAC \quad \textcolor{red}{C.i} = \textcolor{blue}{f(A.s)}$$

$$C \rightarrow c \qquad C.s = g(C.i)$$

继承属性不直接等于某个综合属性

## □ 增加标记非终结符，把 $f(A.s)$ 的计算移到对标记非终结符归约时进行

$$S \rightarrow aA\textcolor{red}{N}C$$

$$\textcolor{red}{N.i} = A.s; \textcolor{red}{C.i} = N.s$$

$$\textcolor{red}{N} \rightarrow \varepsilon$$

$$\textcolor{red}{N.s} = \textcolor{red}{f(N.i)}$$

$$C \rightarrow c$$

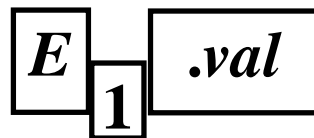
$$C.s = g(C.i)$$



# $L$ 属性定义的自下而上计算

例 数学排版语言EQN

$E \text{ sub } 1 \text{ .val}$



$S \rightarrow B$

$B \rightarrow B_1 B_2$

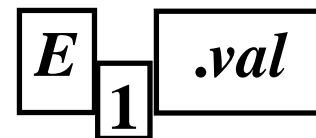
$B \rightarrow B_1 \text{ sub } B_2$

$B \rightarrow \text{text}$



# 数学排版语言EQN

语法制导定义  $E \text{ sub } 1 \text{ .val}$



$ps$ -point size (**L属性**);  $ht$ -height(**S属性**)

产生式	语义规则
$S \rightarrow B$	$B.ps = 10; S.ht = B.ht$
$B \rightarrow B_1 B_2$	$B_1.ps = B.ps; B_2.ps = B.ps;$ $B.ht = \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps = B.ps; B_2.ps = \text{shrink}(B.ps);$ $B.ht = \text{disp}(B_1.ht, B_2.ht)$
$B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$





# 数学排版语言EQN

$S \rightarrow \{B.ps = 10\}$

$B \quad \{S.ht = B.ht\}$

$B \rightarrow \{B_1.ps = B.ps\}$

$B_1 \quad \{B_2.ps = B.ps\}$

$B_2 \quad \{B.ht = \max(B_1.ht, B_2.ht)\}$

$B \rightarrow \{B_1.ps = B.ps\}$

$B_1$

$\text{sub} \quad \{B_2.ps = \text{shrink}(B.ps)\}$

$B_2 \quad \{B.ht = \text{disp}(B_1.ht, B_2.ht)\}$

$B \rightarrow \text{text} \quad \{B.ht = \text{text.h} \times B.ps\}$

产生式	语义规则
$S \rightarrow B$	$B.ps = 10; S.ht = B.ht$
$B \rightarrow B_1 B_2$	$B_1.ps = B.ps; B_2.ps = B.ps;$ $B.ht = \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps = B.ps; B_2.ps = \text{shrink}(B.ps);$ $B.ht = \text{disp}(B_1.ht, B_2.ht)$
$B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$



# EQN：自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$B.ps = L.s; S.ht = B.ht$
$L \rightarrow \epsilon$	$L.s = 10$ 将 $B.ps$ 存入栈中，便于引用
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \epsilon$	$M.s = M.i$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \epsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$B.ps = L.s; S.ht = B.ht$
$L \rightarrow \epsilon$	$L.s = 10$ 将 $B.ps$ 存入栈中, 便于引用
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \epsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \epsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$



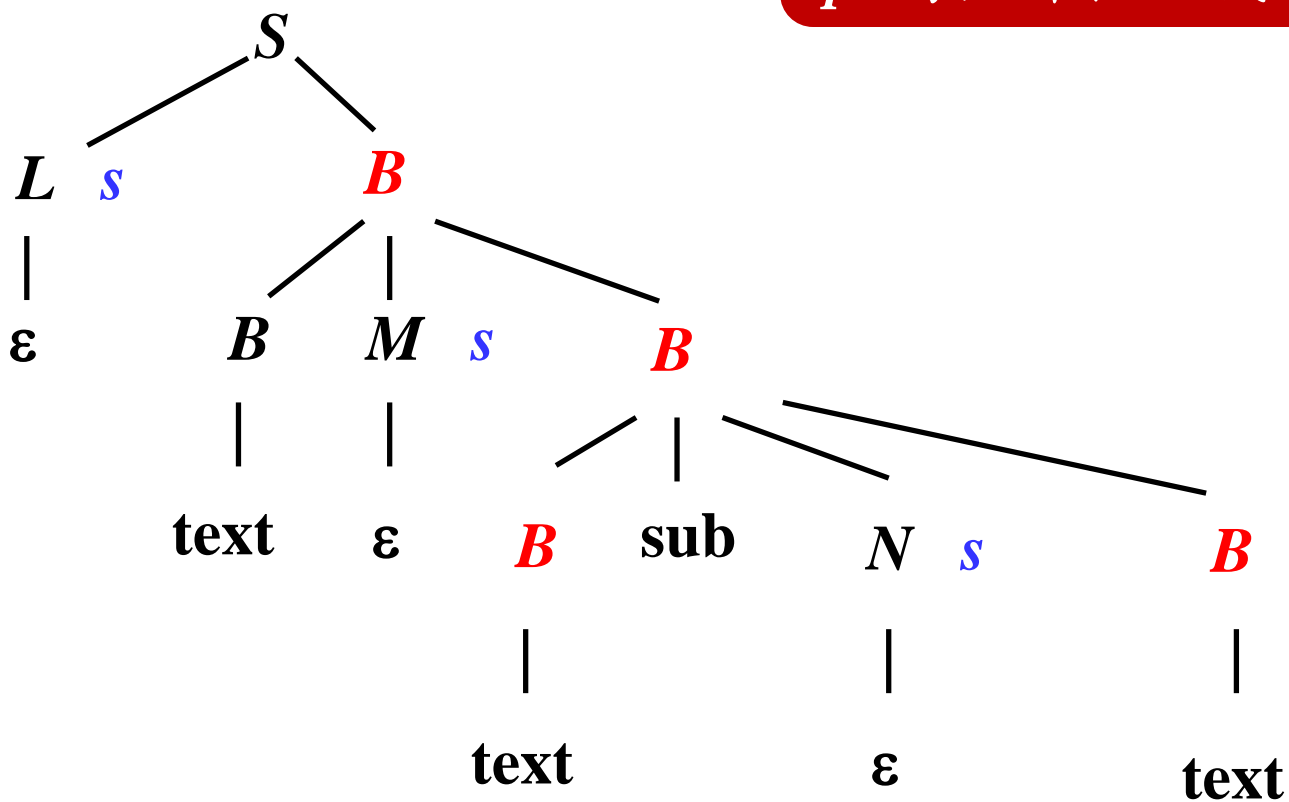
# EQN：自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$B.ps = L.s; S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$ 将 $B.ps$ 存入栈中，便于引用
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$ 兼有计算功能
$B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$



# EQN: 自下而上计算的实现

在text归约成 $B$ 时， $B$ 的 $ps$ 属性都在次栈顶位置





# EQN：自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$B.ps = L.s; S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

继承属性的值等于栈中某个综合属性的值，因此栈中只保存综合属性的值



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \max(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$B.ps = L.s; S.ht = B.ht$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$B_1.ps = B.ps; M.i = B.ps;$ $B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$L.s = 10$





# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = \max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$B_1.ps = B.ps; M.i = B.ps; B_2.ps = M.s; B.ht = \max(B_1.ht, B_2.ht)$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = \max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps = B.ps; N.i = B.ps;$ $B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$M.i = B.ps; M.s = M.i$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = \max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1 \text{ sub } NB_2$	$val[top-3] = \text{disp}(val[top-3], val[top])$
$N \rightarrow \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$B_1.ps = B.ps; N.i = B.ps; B_2.ps = N.s; B.ht = \text{disp}(B_1.ht, B_2.ht)$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = \max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1 \text{ sub } NB_2$	$val[top-3] = \text{disp}(val[top-3], val[top])$
$N \rightarrow \varepsilon$	$val[top+1] = \text{shrink}(val[top-2])$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h \times B.ps$

$$B.ht = \text{text}.h \times B.ps$$



# EQN: 自下而上计算的实现

产生式	语义规则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = \max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1 \text{ sub } NB_2$	$val[top-3] = \text{disp}(val[top-3], val[top])$
$N \rightarrow \varepsilon$	$val[top+1] = \text{shrink}(val[top-2])$
$B \rightarrow \text{text}$	$val[top] = val[top] \times val[top-1]$

$N.i = B.ps; N.s = \text{shrink}(N.i)$



中国科学技术大学  
University of Science and Technology of China

下期预告：语义分析