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

语法制导的翻译 I

《编译原理和技术》

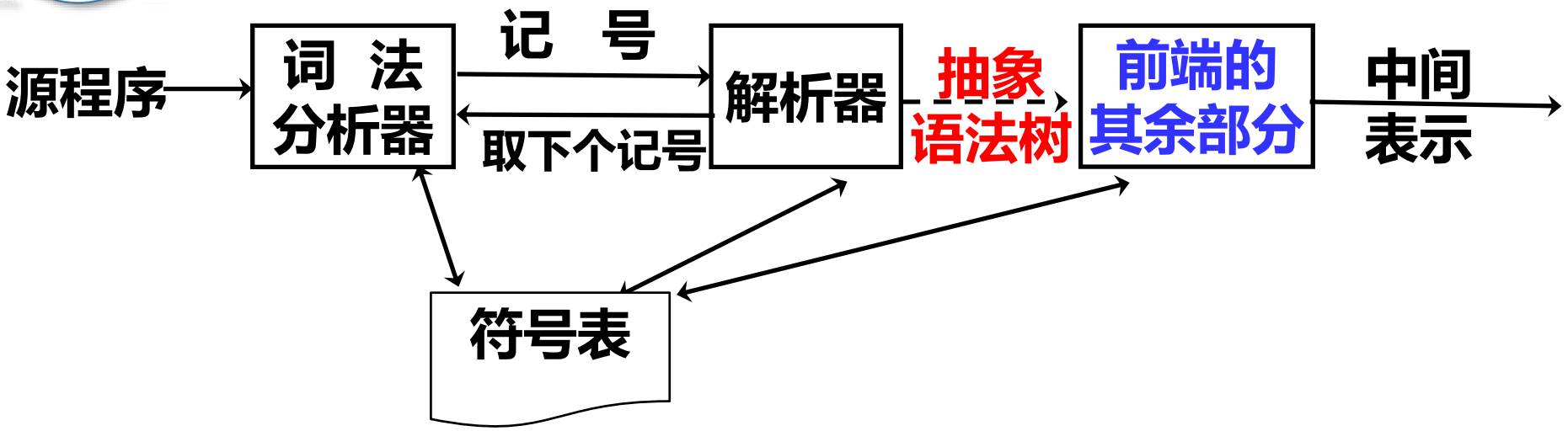
张昱

0551-63603804，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

- **语义描述举例：输出表达式的后缀形式**

  中缀式→后缀式

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

    $E \rightarrow E_1 + T$ $\quad\quad$ $E.code = E_1.code \,||\, T.code \,||\, \text{‘+’}$

    E或T的*code*属性表示E或T对应的后缀形式

    语法制导定义可读性好，更适于描述语义规范

  - **翻译方案(translation scheme)**

    $E \rightarrow E_1 + T$ $\quad\quad$ { 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$A \to \alpha$有一组形式为$b = f(c_1, c_2, \ldots, c_k)$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$c_1, c_2, \ldots, c_k$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，$c_1, c_2, \ldots, c_k$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Σ = {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

对*S*和*D*加下标以区分同类语法结构的不同实例

$S' \rightarrow S$	print(S.*val*)
$S \rightarrow D_1\ S_1\ D_2$	S.*val* = (D_1.*val* == D_2.*val*) and S_1.*val*
$S \rightarrow 2$	S.*val* = *true*
$D \rightarrow 0$	D.*val* = 0
$D \rightarrow 1$	D.*val* = 1

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
$L \rightarrow E$ 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*

换行标记

*L*的匿名属性

由词法分析给出

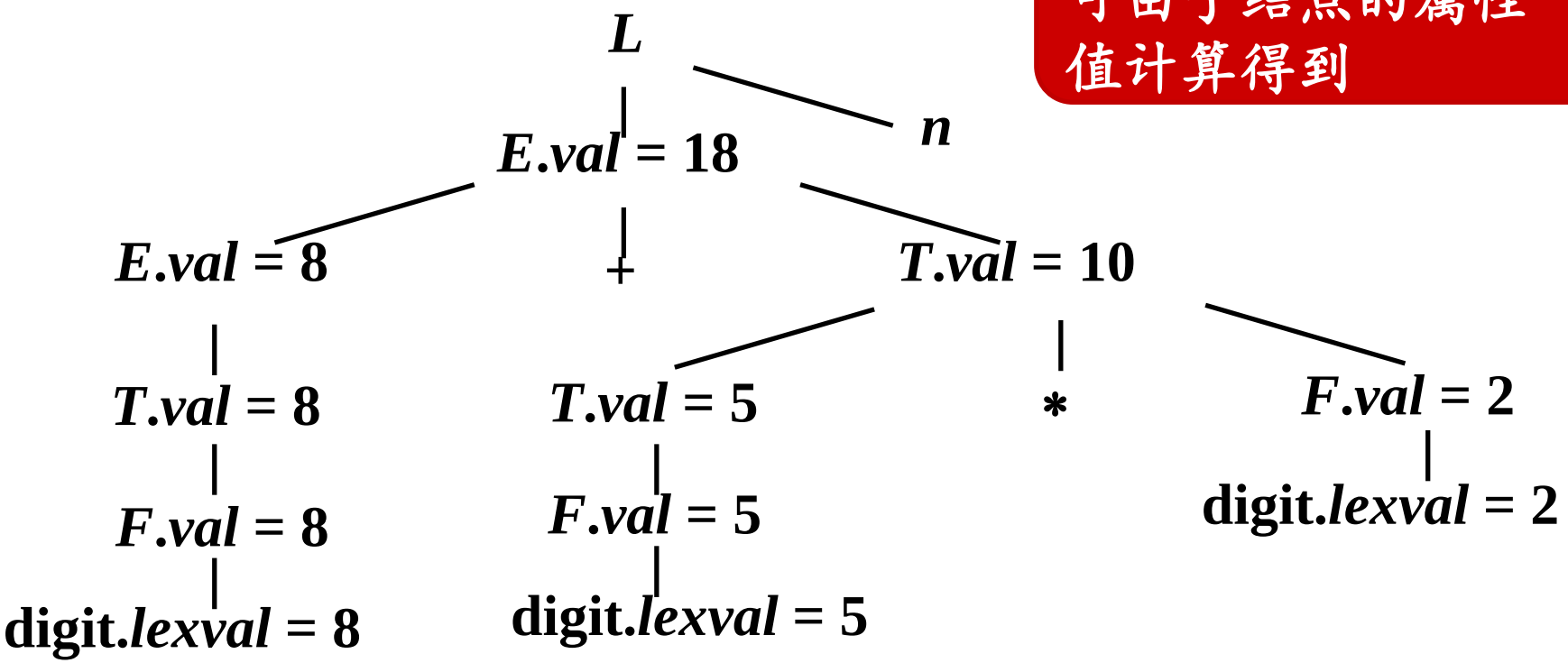
各文法符号的属性均是**综合属性**的语法制导定义——***S* 属性定义**

参见：bison-examples.tar.gz 中的config/expr1.y, expr.lex

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
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to$ digit	$F.val =$ digit.*lexval*

消除左递归

产 生 式	语 义 规 则
$T \to FW$	$T.val = F.val$ ？
$W \to * FW_1$	$W.val =$ ？ $*F.val$
$W \to \varepsilon$	
$F \to$ digit	$F.val =$ 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
- $W \to * FW_1$：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，$c_1, c_2, \ldots, c_k$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

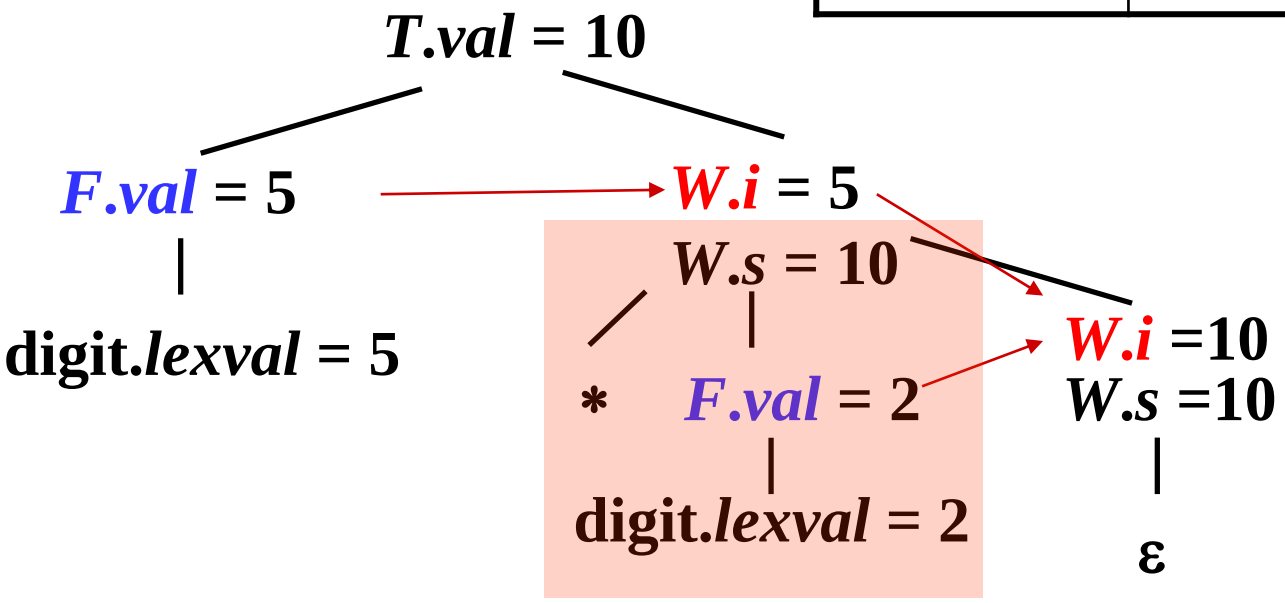
继承W左边的文法符号F的属性

产 生 式	语 义 规 则
$T \to FW$	$W.i = F.val$; $T.val = W.s$
$W \to *FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \to \varepsilon$	$W.s = W.i$
…	…

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
$T \rightarrow FW$	$W.i = F.val$; $T.val = W.s$
$W \rightarrow *\ FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$
…	…



属性依赖图(dependence graph)

属性依赖图

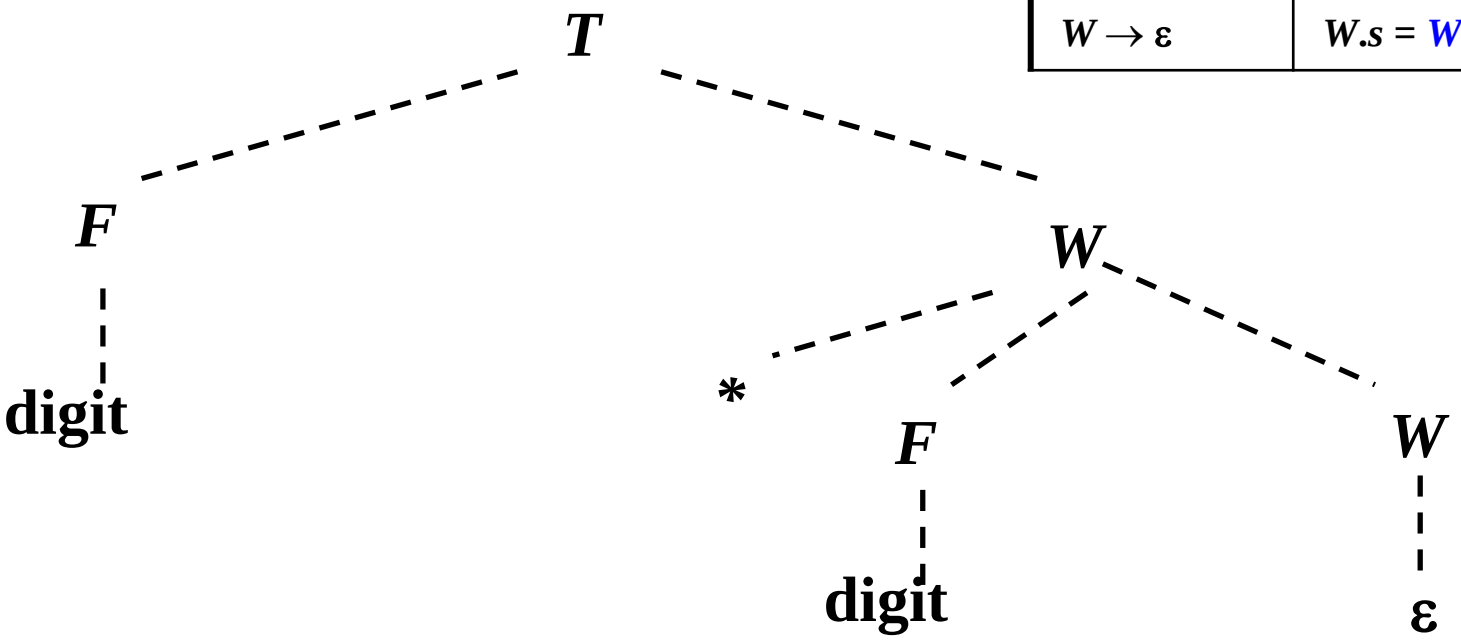
- 描述分析树中结点的属性间依赖关系的有向图
- 顶点为属性：对应分析树中每个文法符号X的每个属性a
- 弧为属性间依赖关系：如果属性$X.a$的值依赖于属性$Y.b$的值，则存在从$Y.b$的顶点指向$X.a$的顶点的弧

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
$T \rightarrow FW$	$W.i = F.val$; $T.val = W.s$
$W \rightarrow * FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$

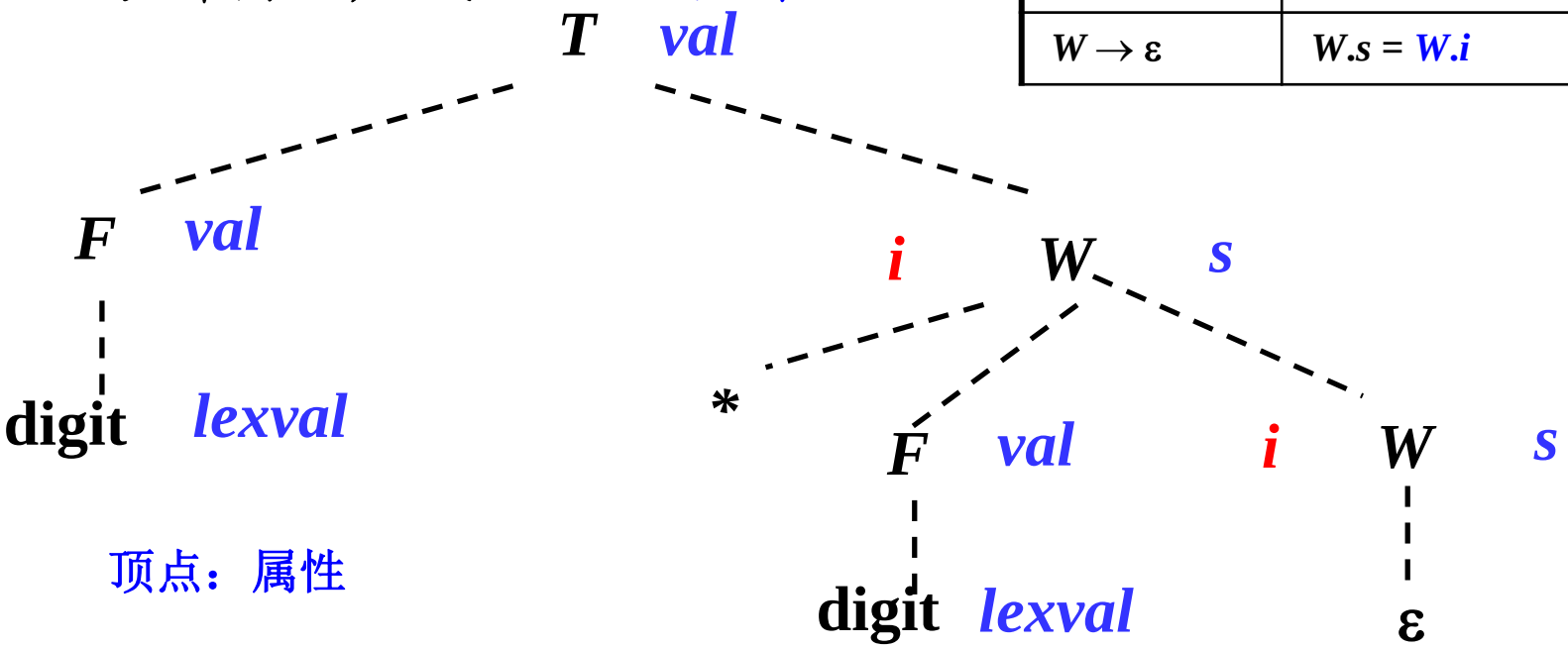


属性依赖图(dependence graph)

- 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)

产 生 式	语 义 规 则
$T \to FW$	$W.i = F.val$; $T.val = W.s$
$W \to * FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \to \varepsilon$	$W.s = W.i$

分析树（虚线）的依赖图



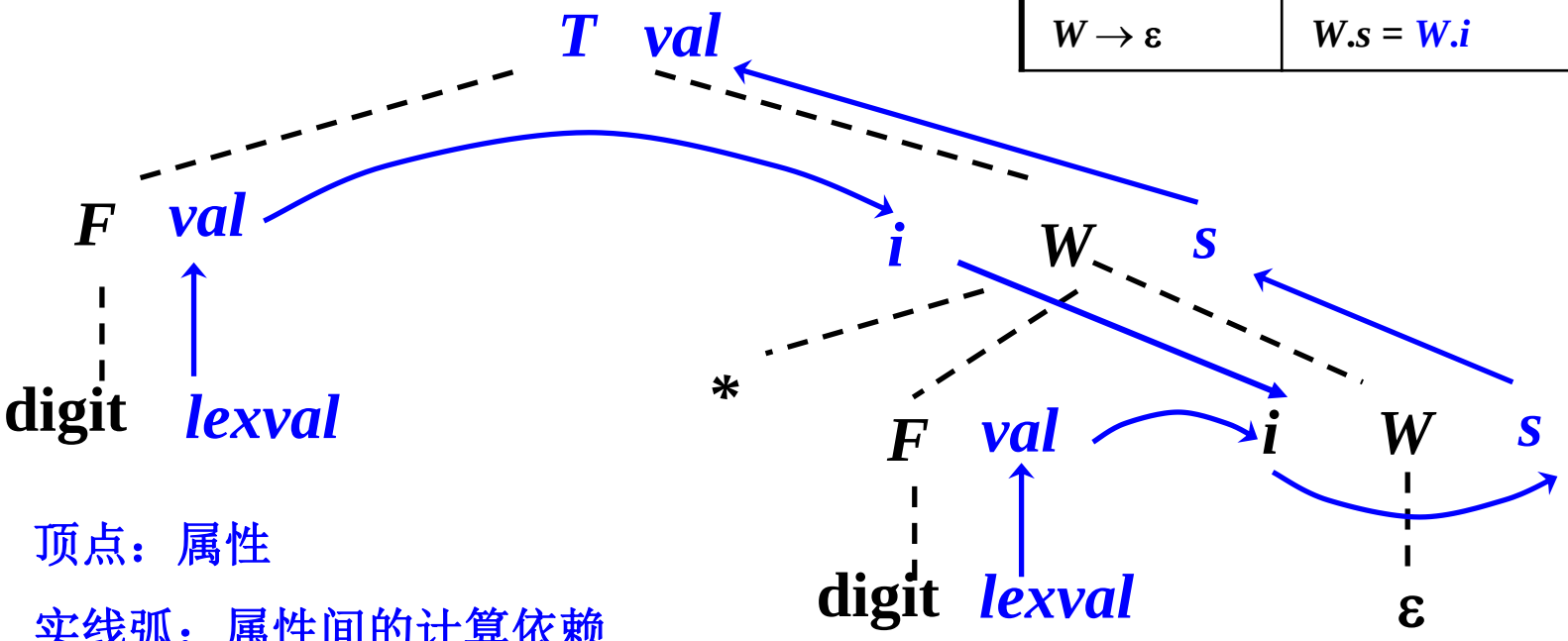
顶点：属性

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
$T \rightarrow FW$	$W.i = F.val$; $T.val = W.s$
$W \rightarrow * FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$



顶点：属性

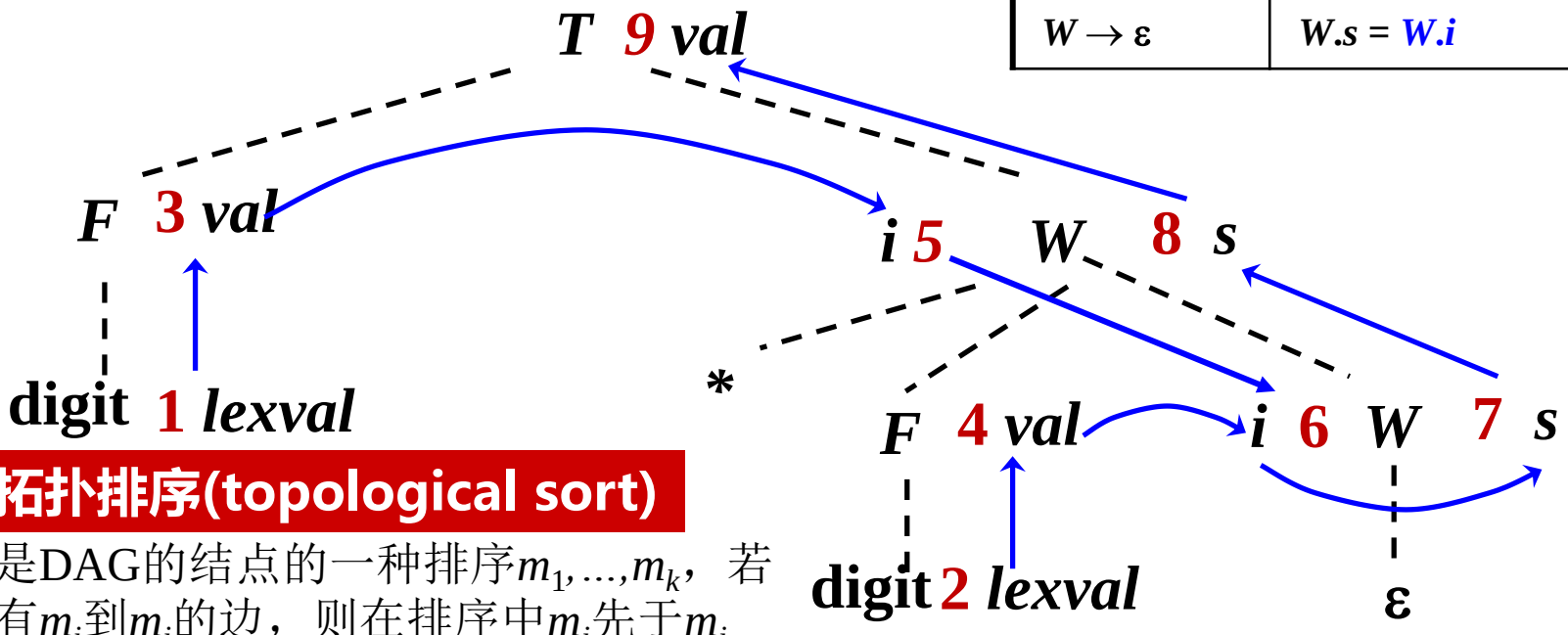
实线弧：属性间的计算依赖

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（实线）

产 生 式	语 义 规 则
$T \to FW$	$W.i = F.val$; $T.val = W.s$
$W \to *\ FW_1$	$W_1.i = W.i * F.val$; $W.s = W_1.s$
$W \to \varepsilon$	$W.s = W.i$



拓扑排序(topological sort)

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

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 $A \rightarrow X_1 \ldots X_{j-1} X_j \ldots X_n$ 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 $X_1, X_2, \ldots, X_{j-1}$ 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
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addType(\text{id}.entry, L.in)$
$L \rightarrow$ id	$addType(\text{id}.entry, L.in)$

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

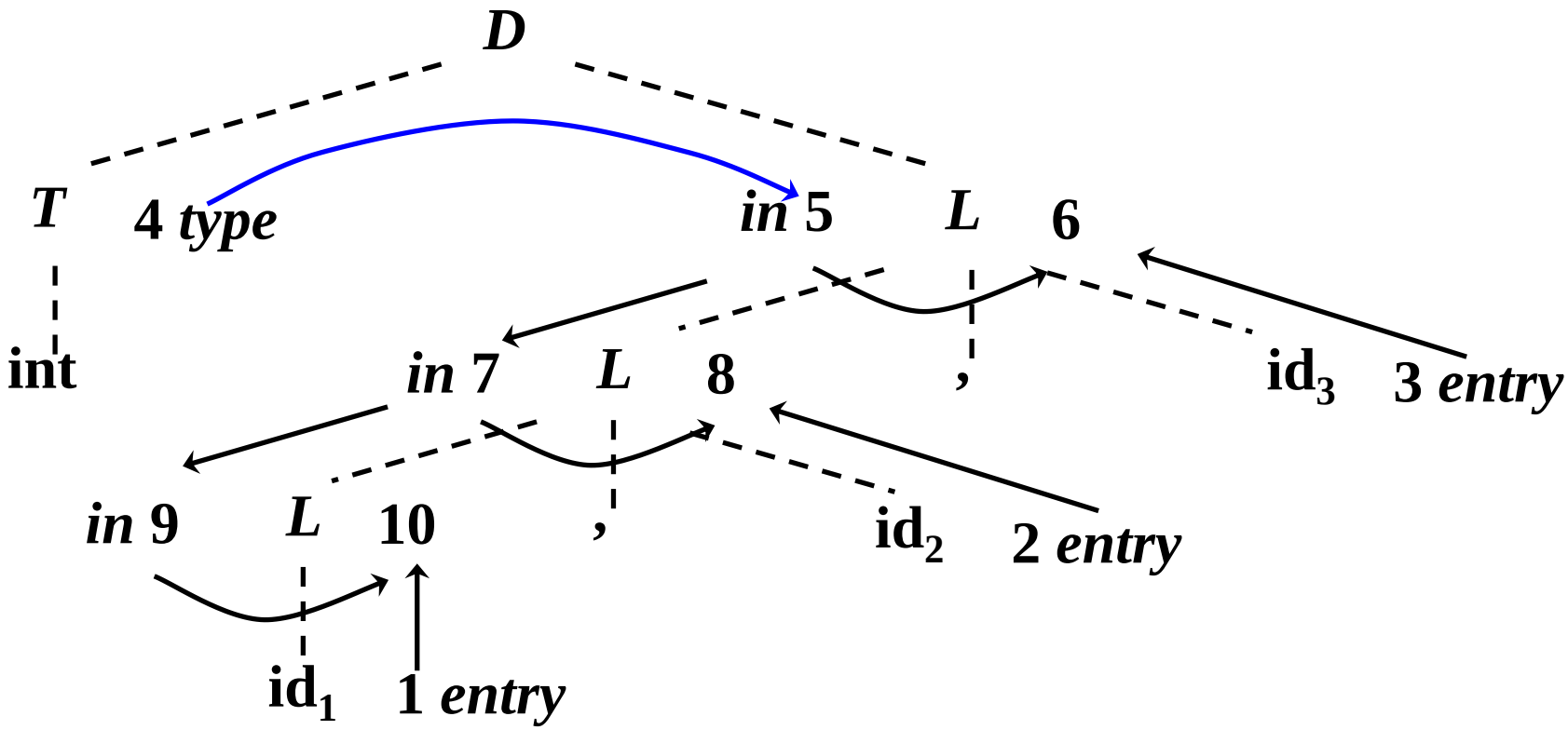
addType – 把类型信息加到符号表中的标识符条目里

属性依赖图(dependence graph)

int id_1, id_2, id_3

分析树（虚线）的依赖图（实线）

$D \rightarrow TL$　　$L.in = T.type$

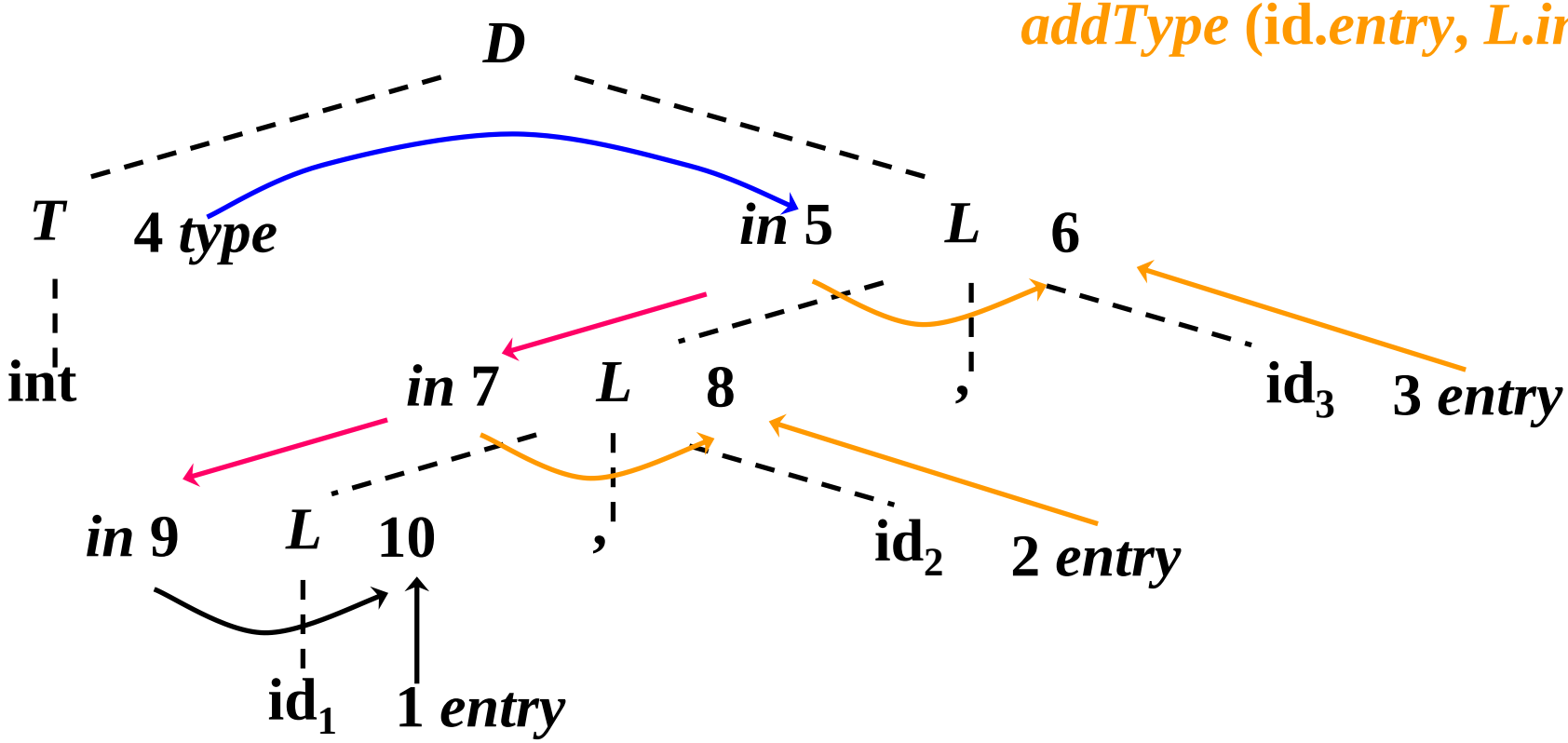


属性依赖图(dependence graph)

int id_1, id_2, id_3

分析树（虚线）的依赖图（实线）

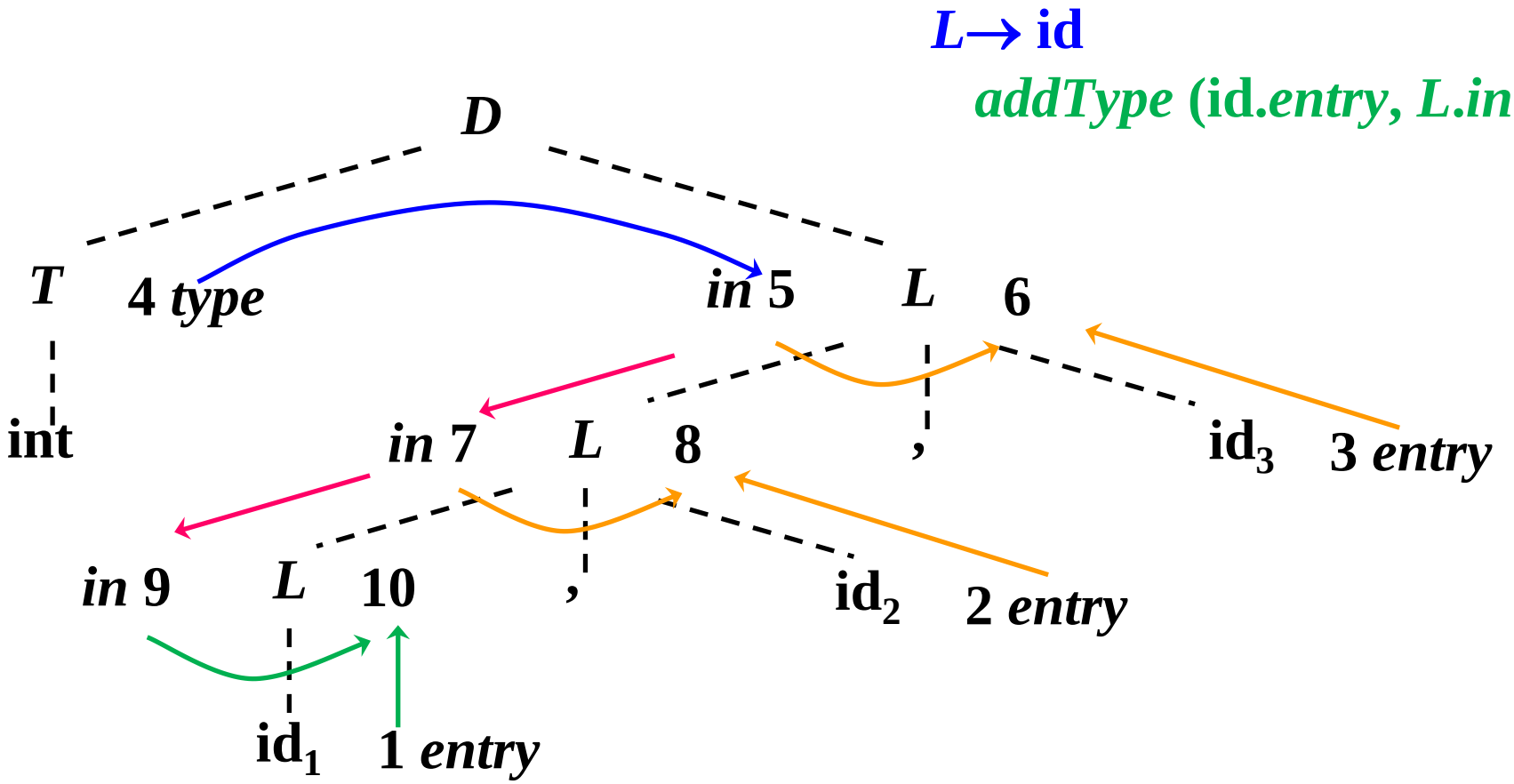
$L \rightarrow L_1$, id　$L_1.in = L.in$;
addType (id.*entry*, *L.in*)



属性依赖图(dependence graph)

int id_1, id_2, id_3

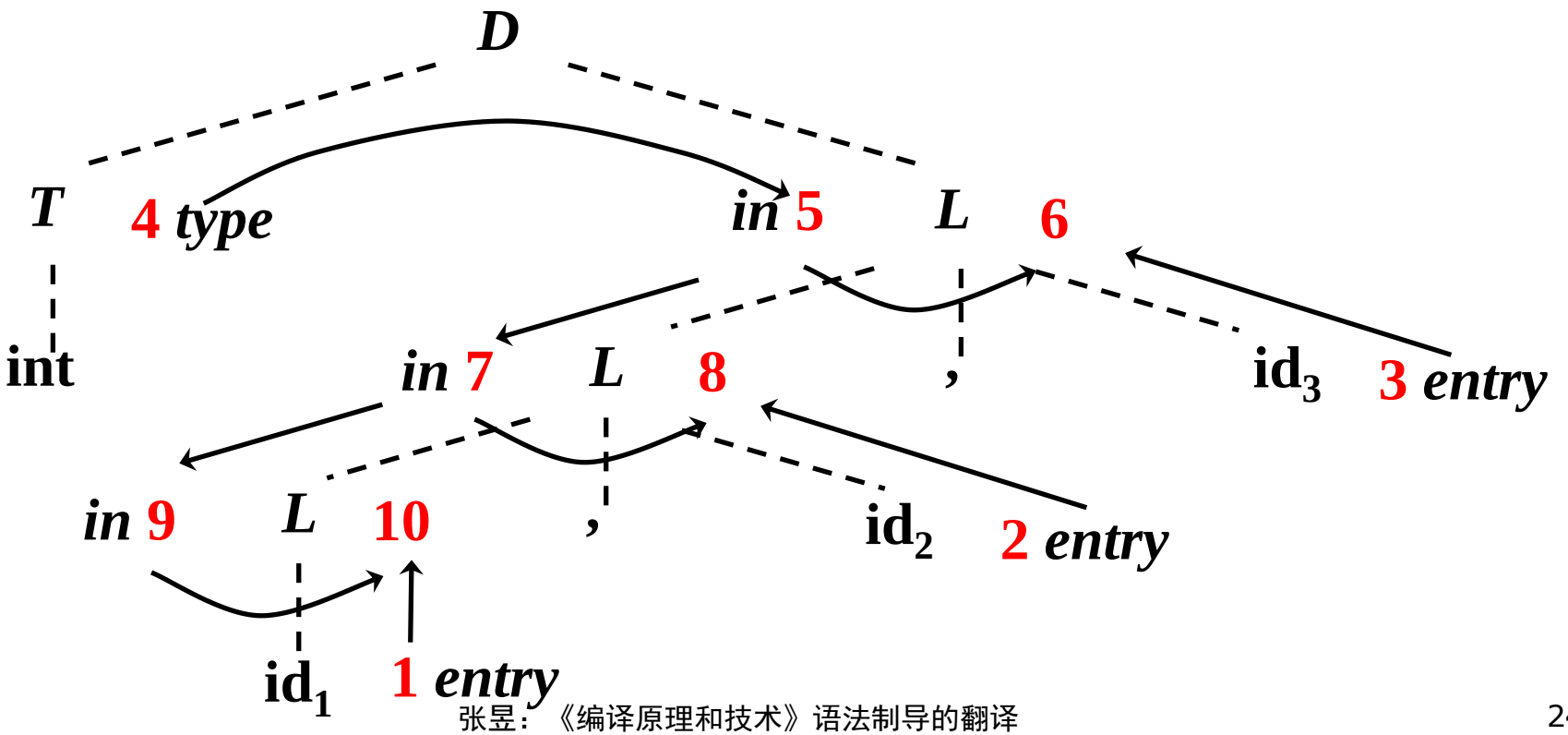
分析树（虚线）的依赖图（实线）



属性计算次序

- 拓扑排序(topological sort)：是DAG的结点的一种排序 $m_1,\ldots,m_k$，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



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

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)

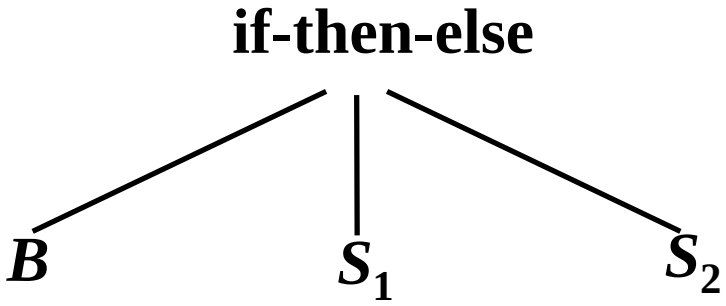
- 语法树是分析树的浓缩表示

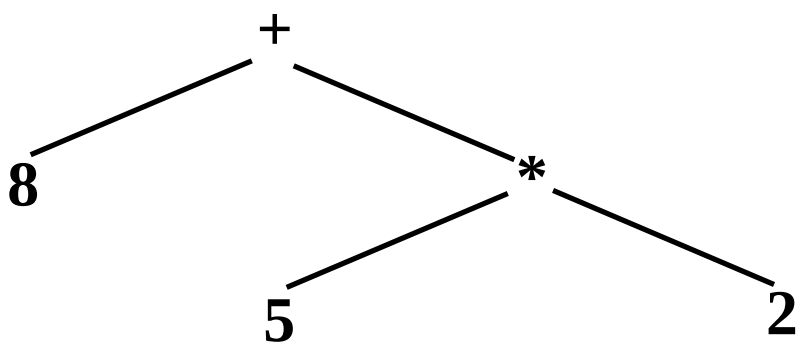
  每个结点表示一个语法构造，算符和关键字是语法树中的内部结点

  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

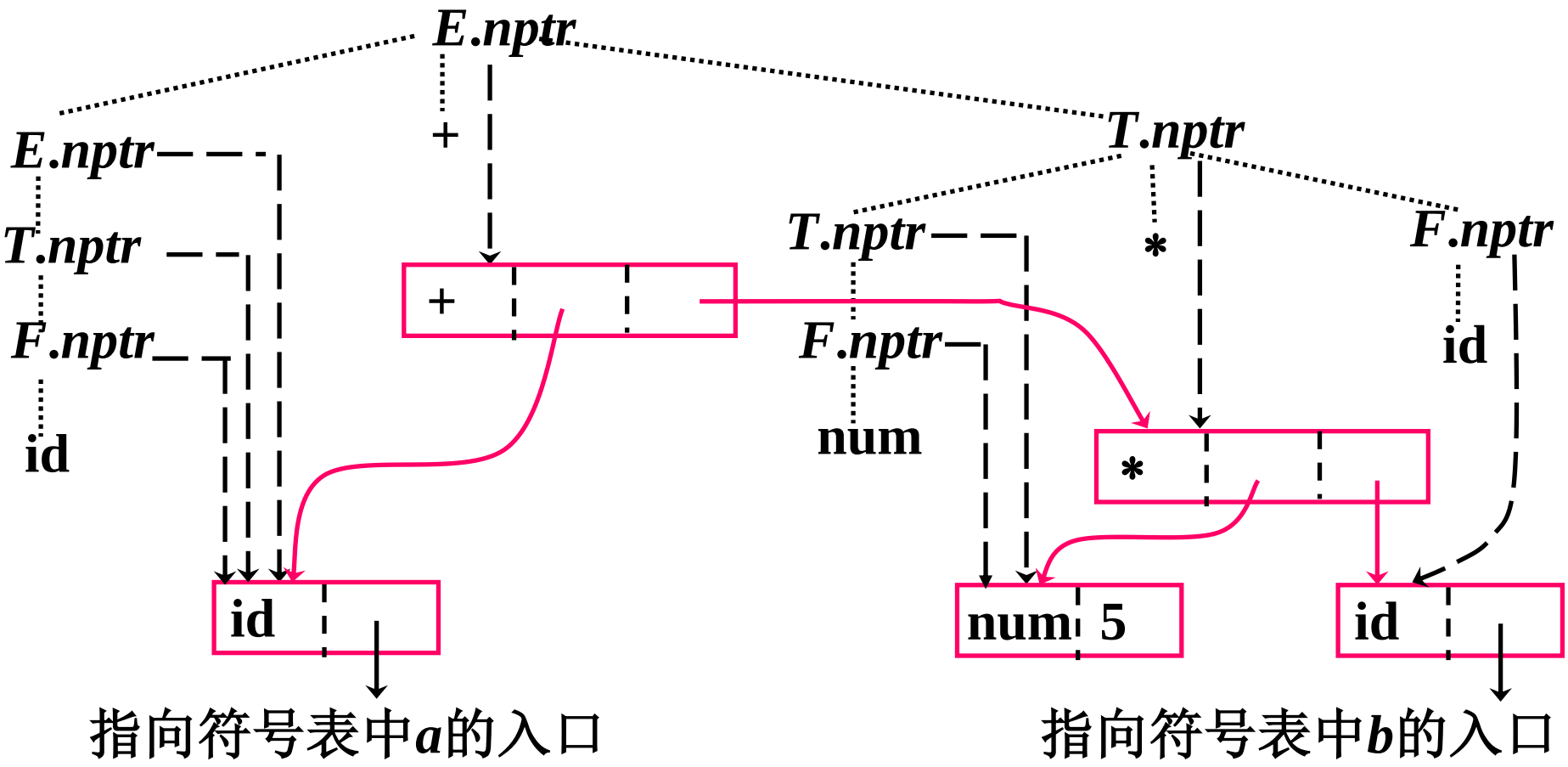
语法制导定义: 构造语法树

产　生　式	语　义　规　则
$E \rightarrow E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow \text{id}$	$F.nptr = mkLeaf\ (\text{id}, \text{id}.entry)$
$F \rightarrow \text{num}$	$F.nptr = mkLeaf\ (\text{num}, \text{num}.val)$

参见：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

$E \rightarrow E_1 + T$	$\{E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)\}$
$E \rightarrow T$	$\{E.nptr = T.nptr\}$
$T \rightarrow T_1 * F$	$\{T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)\}$
$T \rightarrow F$	$\{T.nptr = F.nptr\}$
$F \rightarrow (E)$	$\{F.nptr = E.nptr\}$
$F \rightarrow \text{id}$	$\{F.nptr = mkLeaf(\text{id}, \text{id}.entry)\}$
$F \rightarrow \text{num}$	$\{F.nptr = mkLeaf(\text{num}, \text{num}.val)\}$

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

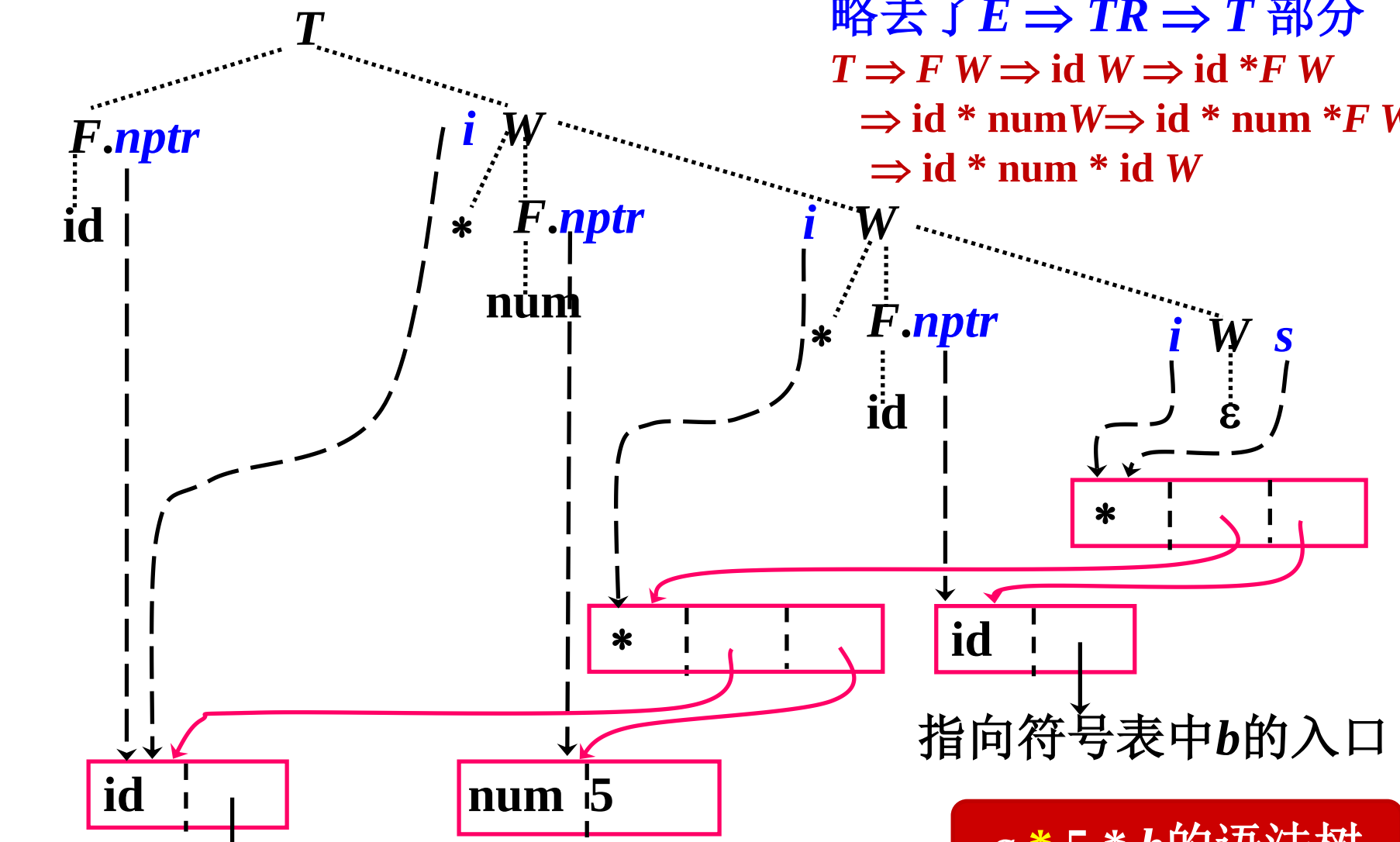
表达式语言的 *LL*文法

产 生 式	语 义 规 则
$E \rightarrow T\ R$	$R.i = T.nptr\ ;\ E.nptr = R.s$
$R \rightarrow +\ TR_1$	$R_1.i = mkNode\ (\ '+',\ R.i,\ T.nptr);\ R.s = R_1.s$
$R \rightarrow \varepsilon$	$R.s = R.i$
$T \rightarrow FW$	$W.i = F.nptr\ ;\ T.nptr = W.s$
$W \rightarrow *\ FW_1$	$W_1.i = mkNode\ (\ '+',\ W.i,\ F.nptr);\ W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$
…	…

语法树的构造（LL文法）

略去了$E \Rightarrow TR \Rightarrow T$部分

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



a * 5 * b的语法树

翻译方案

$E \to T$ $\quad \{R.i = T.nptr\}$

$\quad R$ $\quad \{E.nptr = R.s\}$

$R \to +$

$\quad T$ $\quad \{R_1.i = mkNode(\text{'+'}, R.i, T.nptr)\}$

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

$R \to \varepsilon$ $\quad \{R.s = R.i\}$

$T \to F$ $\quad \{W.i = F.nptr\}$

$\quad W$ $\quad \{T.nptr = W.s\}$

$W \to *$

$\quad F$ $\quad \{W_1.i = mkNode(\text{'*'}, W.i, F.nptr)\}$

$\quad W_1$ $\quad \{W.s = W_1.s\}$

$W \to \varepsilon$ $\quad \{W.s = W.i\}$

$T + T + T + \ldots$

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 FuncFParamList : 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。

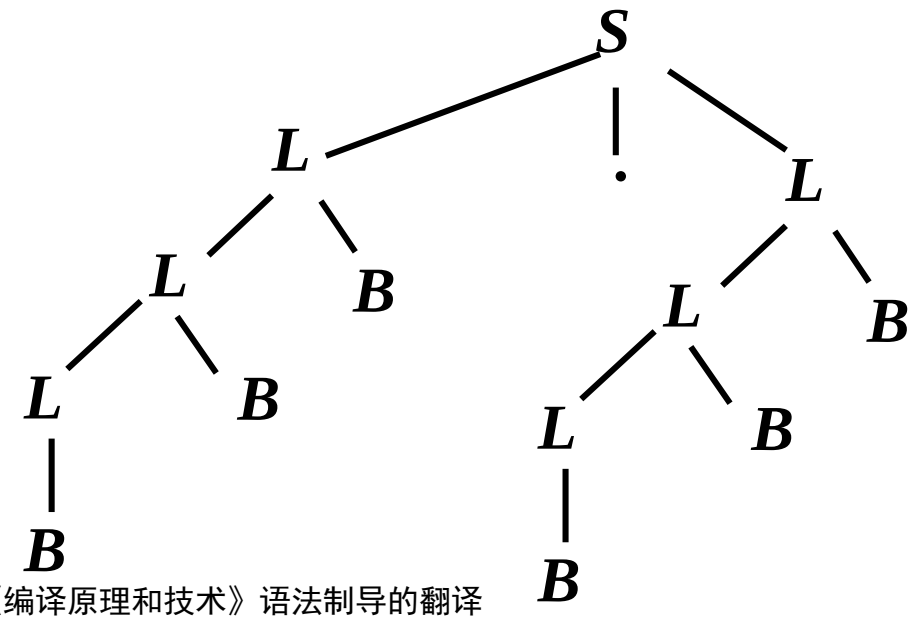
例如，输入101.101时，S.*val* = 5.625（可以修改文法）

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。

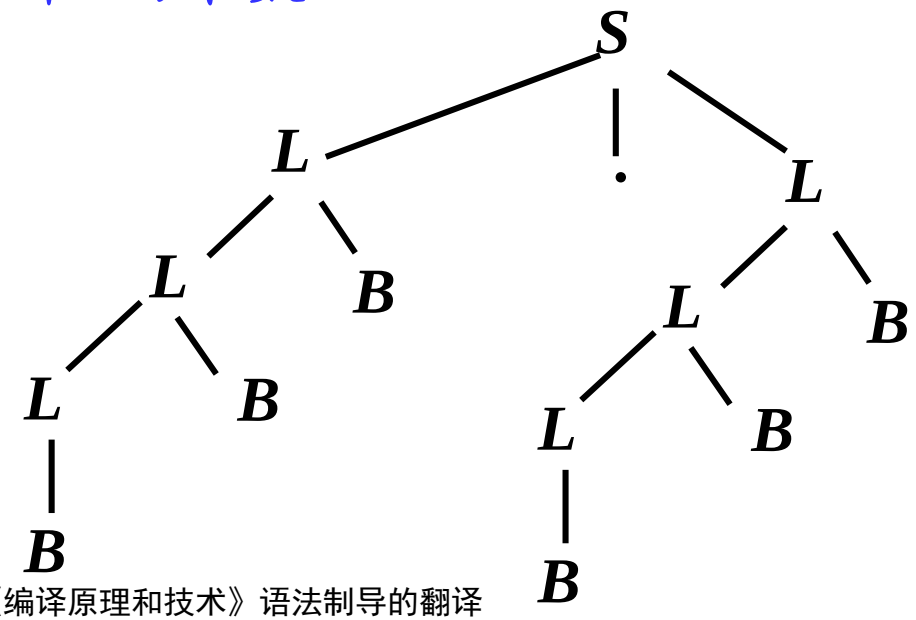
例如，输入101.101时，S.*val* = 5.625（可以修改文法）

若小数点左边按$(1\times2+0)\times2+1$计算。该办法不能直接用于小数点右边，需改成$((1\times2+0)\times2+1)/2^3$，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）

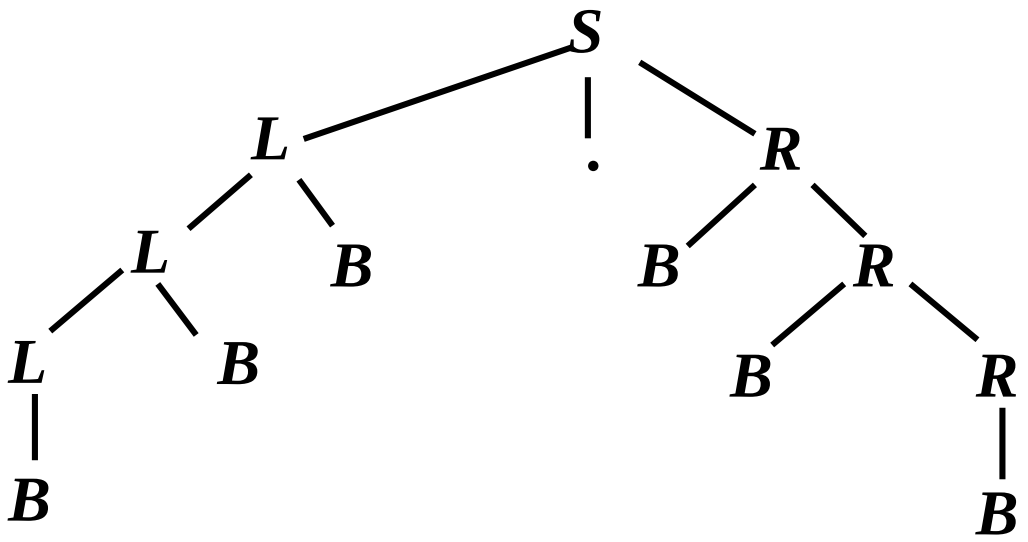
更清楚的办法是将文法改成下面的形式

$S \rightarrow L\ .\ R \mid L$

$L \rightarrow L\ B \mid B$

$R \rightarrow B\ R \mid B$

$B \rightarrow 0 \mid 1$



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

$S' \rightarrow E$ *print* (E. *code*)

$E \rightarrow E_1 + T$

if T. *op* == *plus* then

E.*code* =E_1.*code*||“+”||“(”||T.*code*||“)”

else

E. *code* = E_1. *code* || “+” || T. *code*;

E. *op* = *plus*

$E \rightarrow T$ 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

$S' \rightarrow E$

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

$E \rightarrow E_1 + T$

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

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

$E.code = E_1.code \| \text{“+”} \| T.code\ ;\ E.self\_op = 1;$

$E \rightarrow T$

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

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

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

例 题 3

$T \rightarrow T_1 * F$ . . .

$T \rightarrow F$ . . .

$F \rightarrow \mathbf{id}$

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

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* >= *F. right_op*)

then *E. self_op* else 3

F. code =

if (*F. left_op* < *E. self_op*) and

(*E. self_op* >= *F. right_op*)

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

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

下期预告：语义计算

- 自上而下计算
- 自下而上计算