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

中国科学技术大学

计算机科学与技术学院

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)

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

    可读性好，更适于描述规范
  - 翻译方案(translation scheme)

    $E \rightarrow E_1 + T$ $\quad\quad$ { print ‘+’ }

    陈述了实现细节(如语义规则的计算时机)
- 语法制导翻译的实现方法
  - 自上而下
  - 自下而上

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

4.1 语法制导的定义

- 定义
- 综合属性、继承属性
- 属性依赖图与属性的计算次序

语法制导定义

定义

- 基础的上下文无关文法
- 每个文法符号有一组属性
- 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，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，
- 综合属性(synthesized attribute)：如果b是A的属性，c_1，c_2，…，c_k是产生式右部文法符号的属性或A的其它属性
- 继承属性(inherited attribute)：如果b是右部某文法符号X的属性

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.$lexval$

L的匿名属性

对E加下标以区分不同的属性值

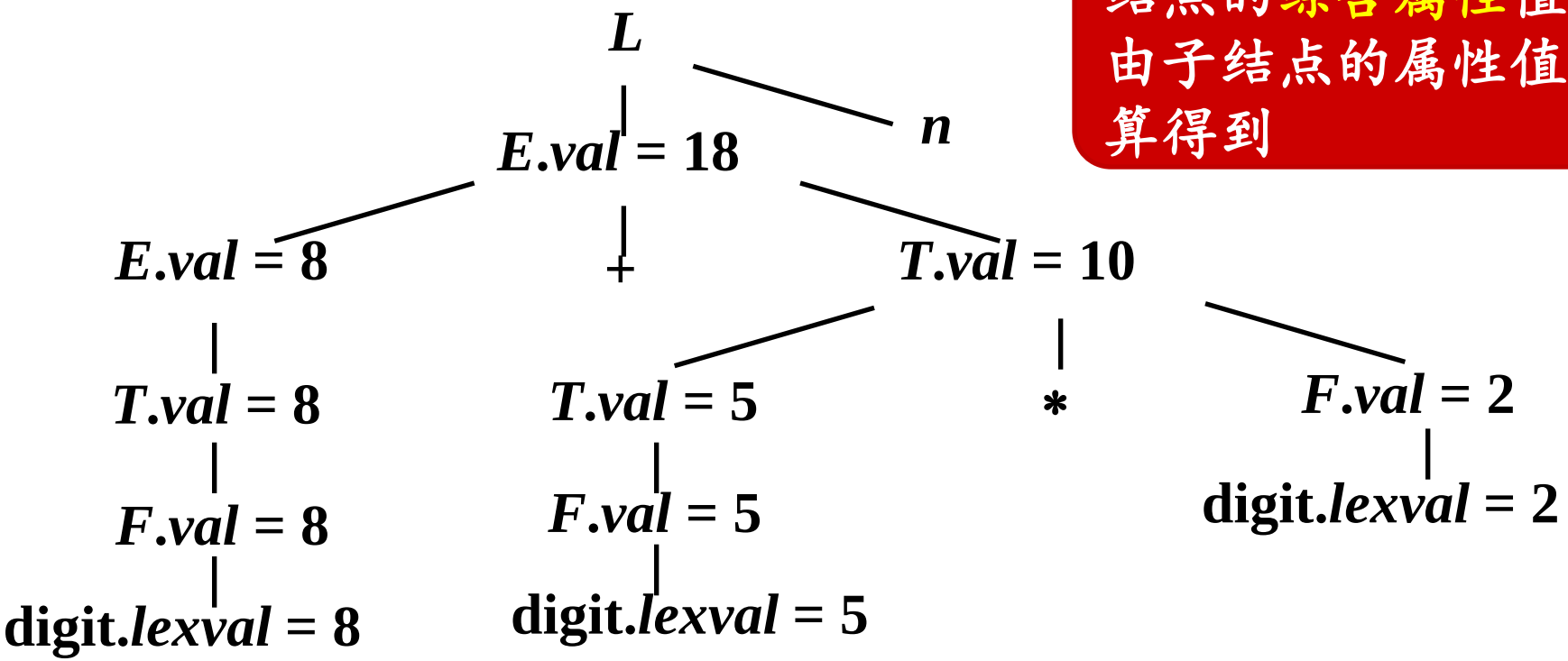
各文法符号的属性均是综合属性的语法制导定义——*S* 属性定义

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

继承属性举例

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，

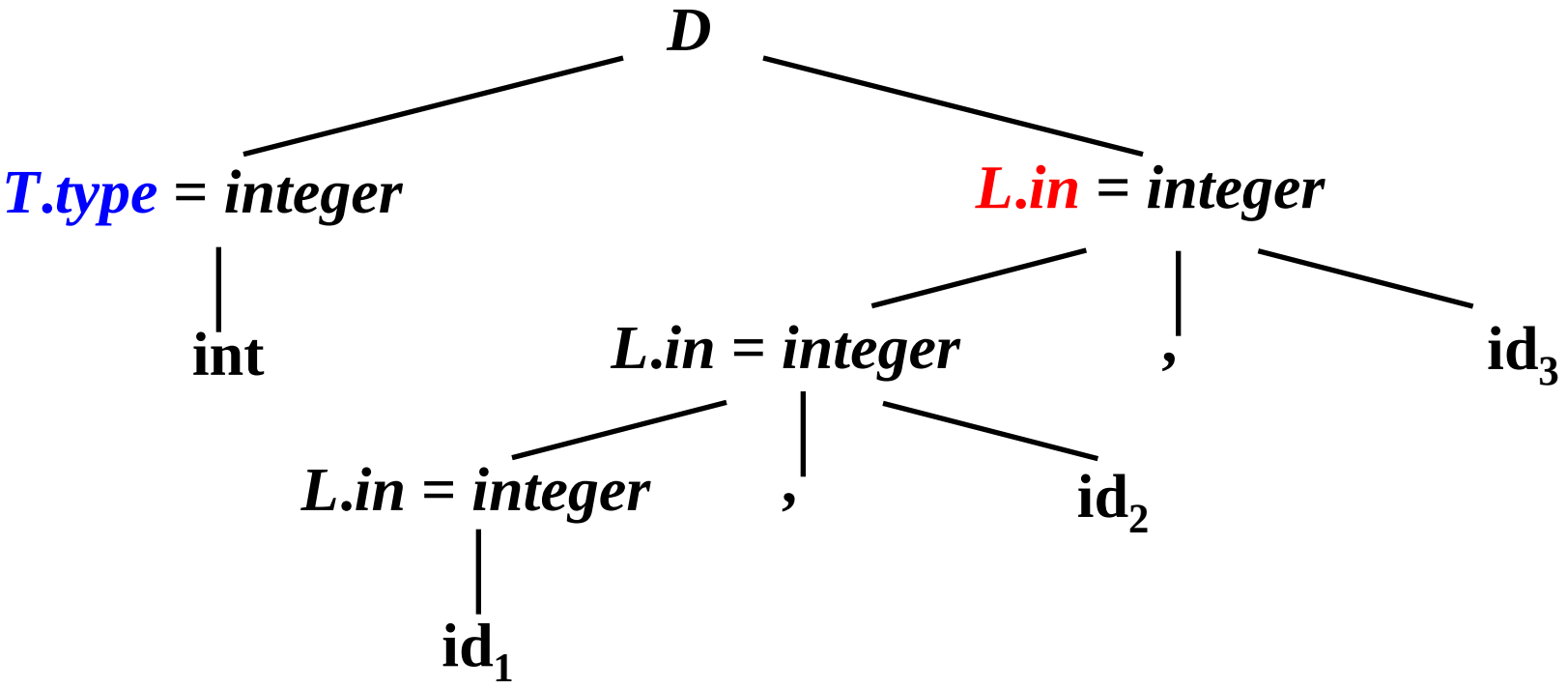
in – *L*的继承属性，把声明的类型传递给标识符列表

addType – 把类型信息加到符号表中的标识符条目里

含继承属性的注释分析树

int id_1, id_2, id_3

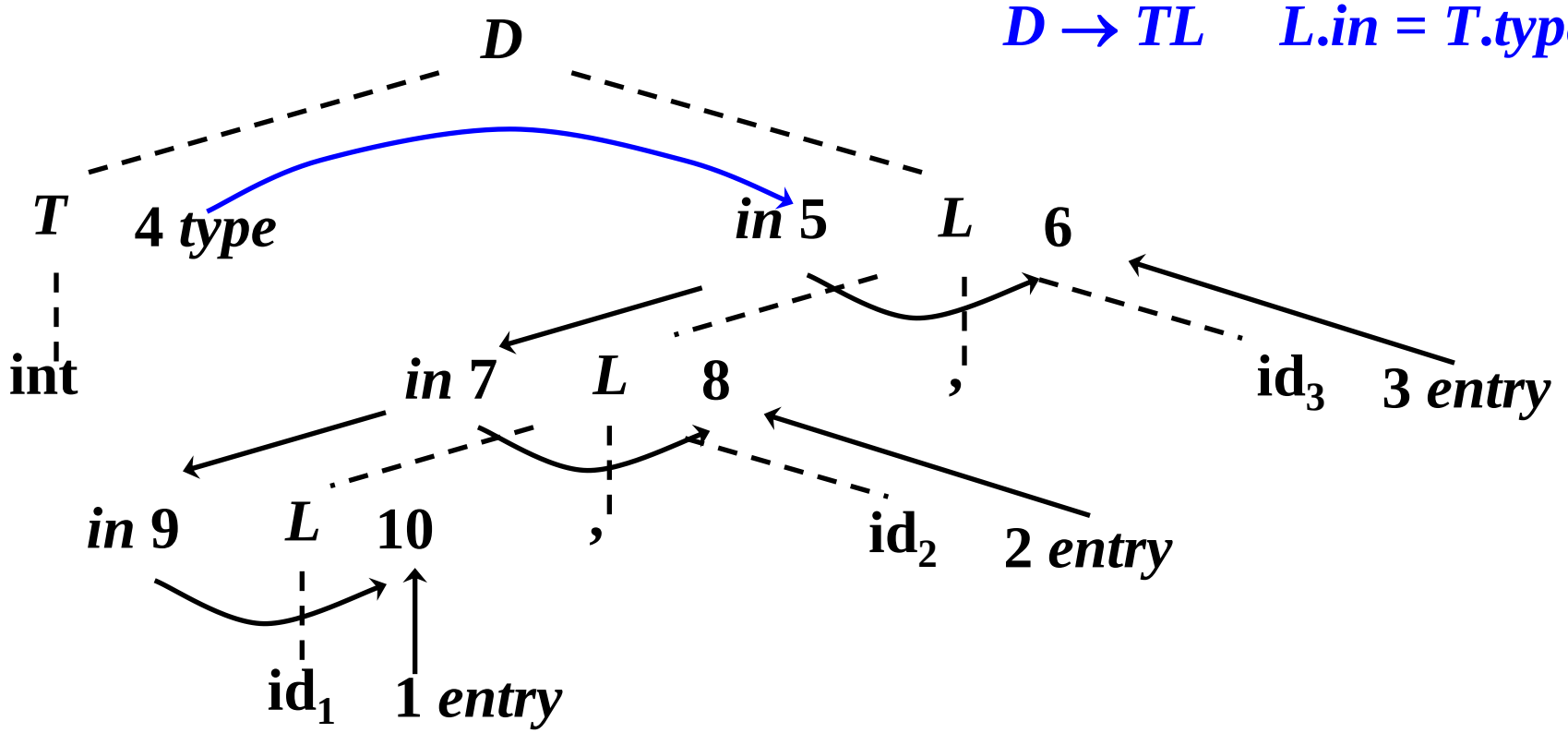
不能像综合属性那样自下而上标注继承属性



属性依赖图(dependence graph)

int id_1, id_2, id_3

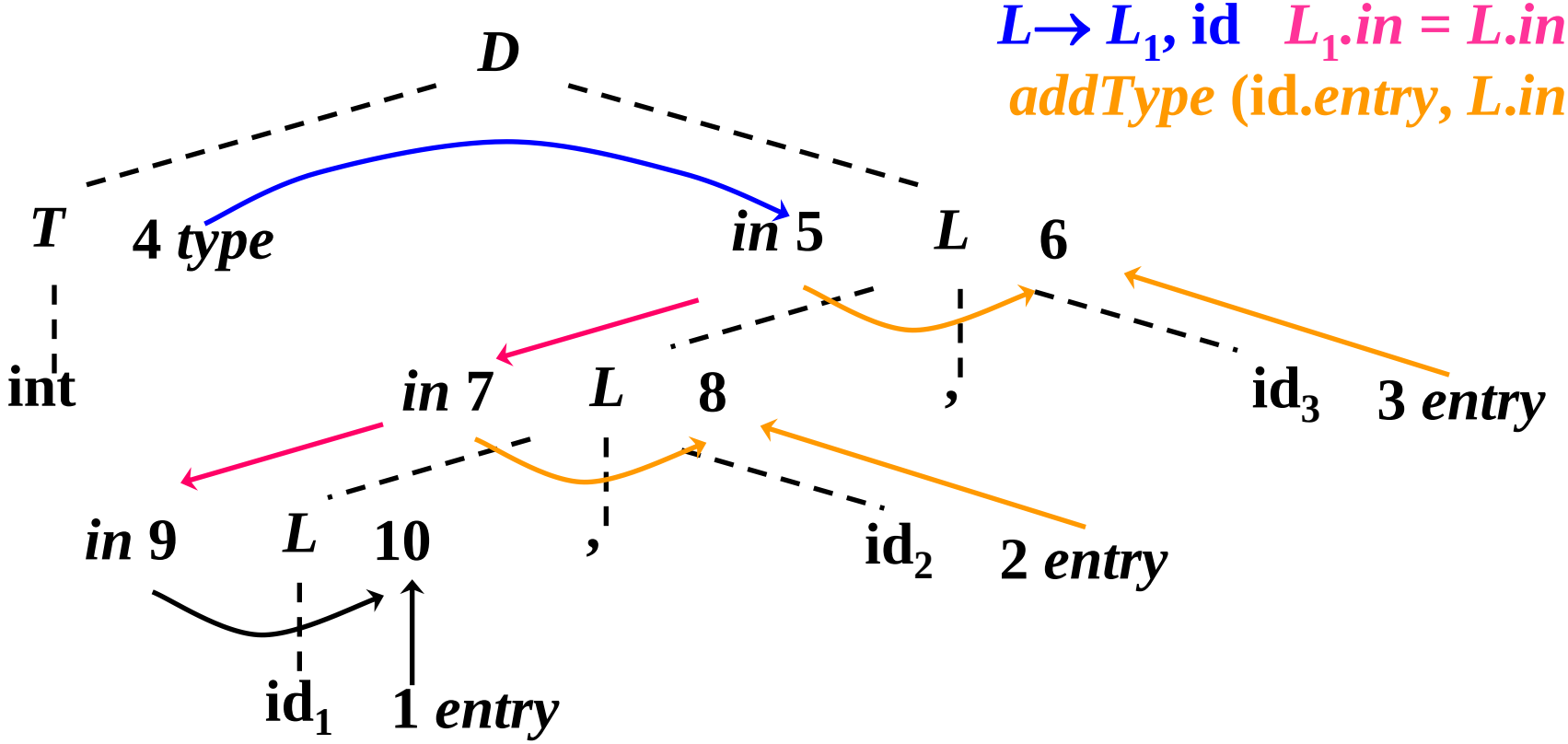
分析树（虚线）的依赖图（实线）



属性依赖图(dependence graph)

int id_1, id_2, id_3

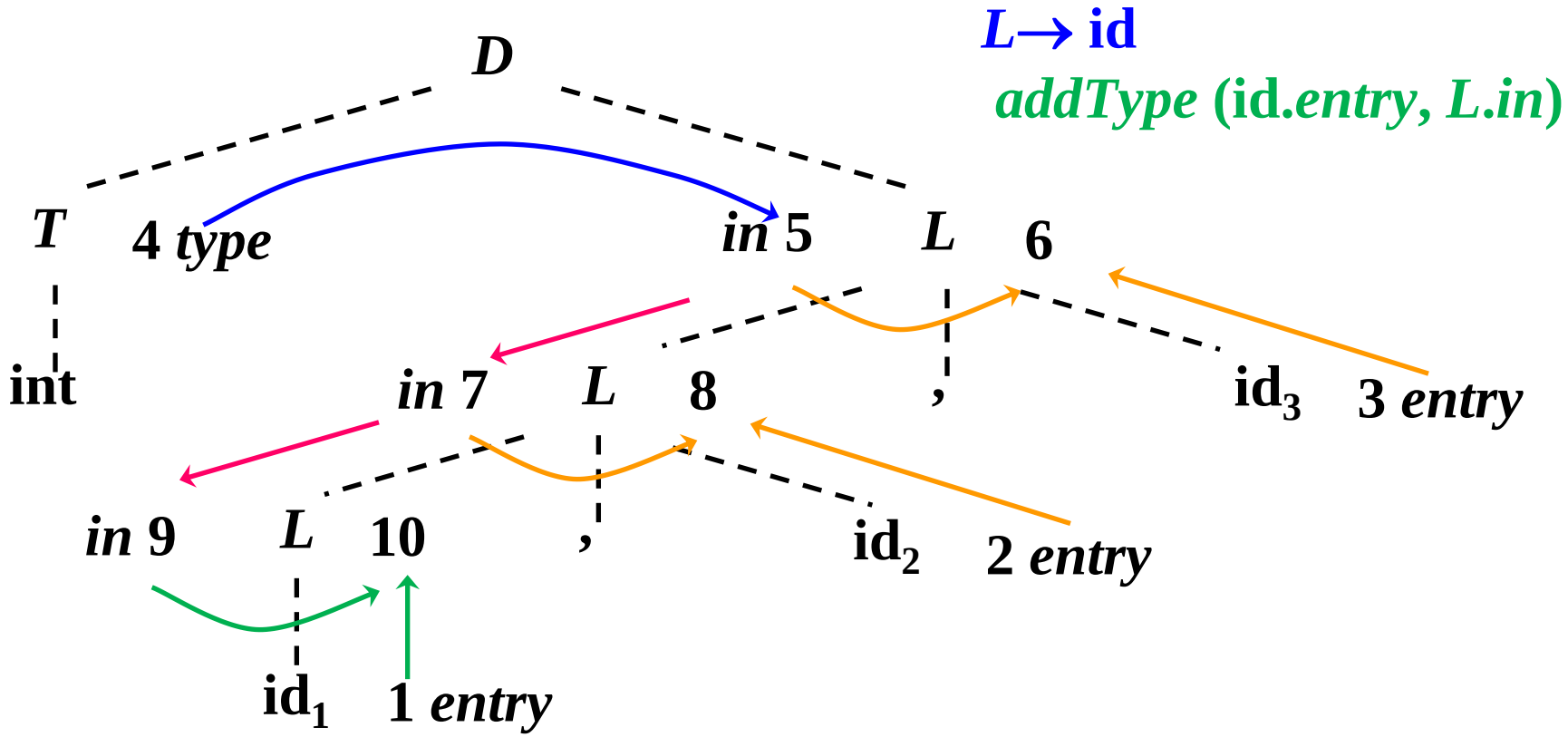
分析树（虚线）的依赖图（实线）



属性依赖图(dependence graph)

int id_1, id_2, id_3

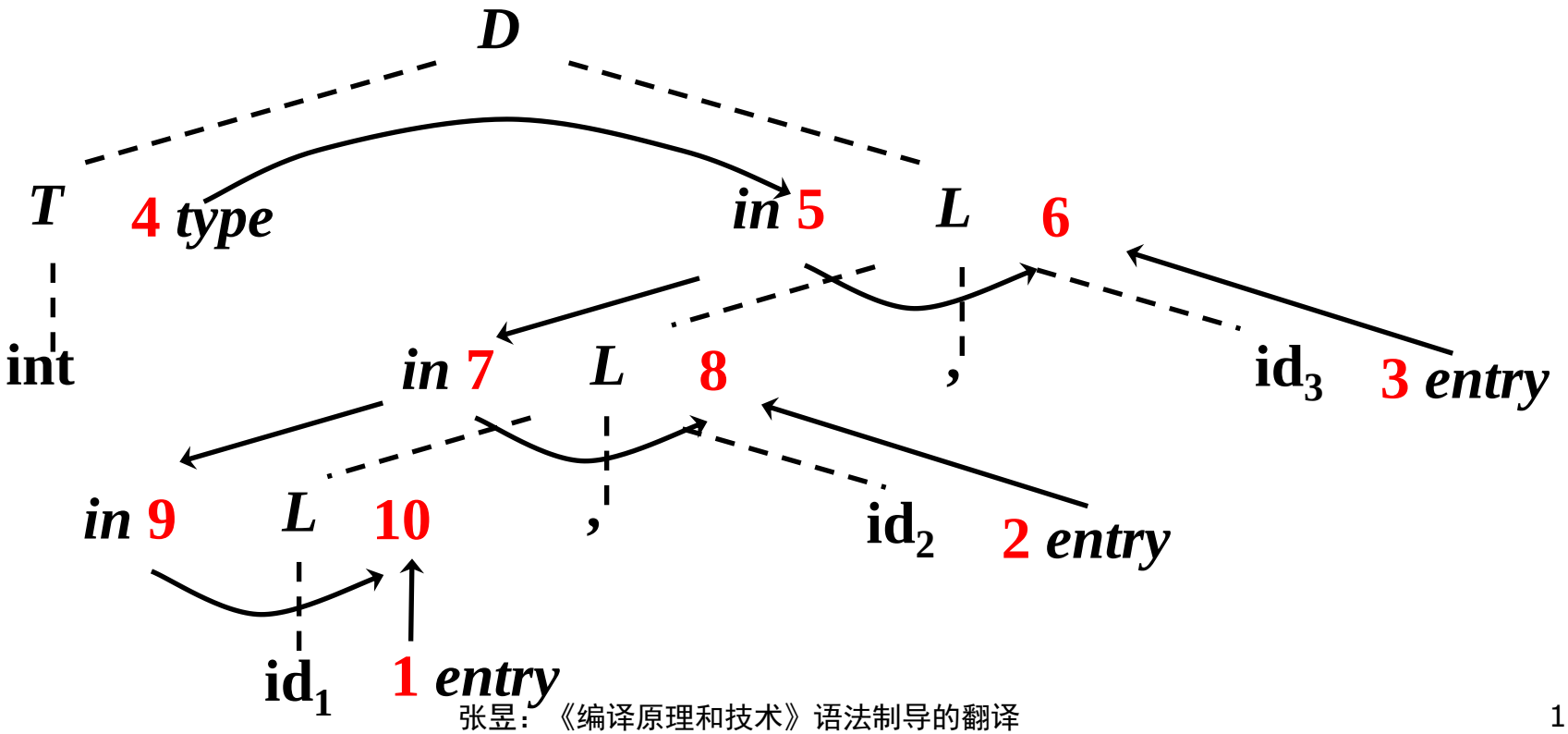
分析树（虚线）的依赖图（实线）



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

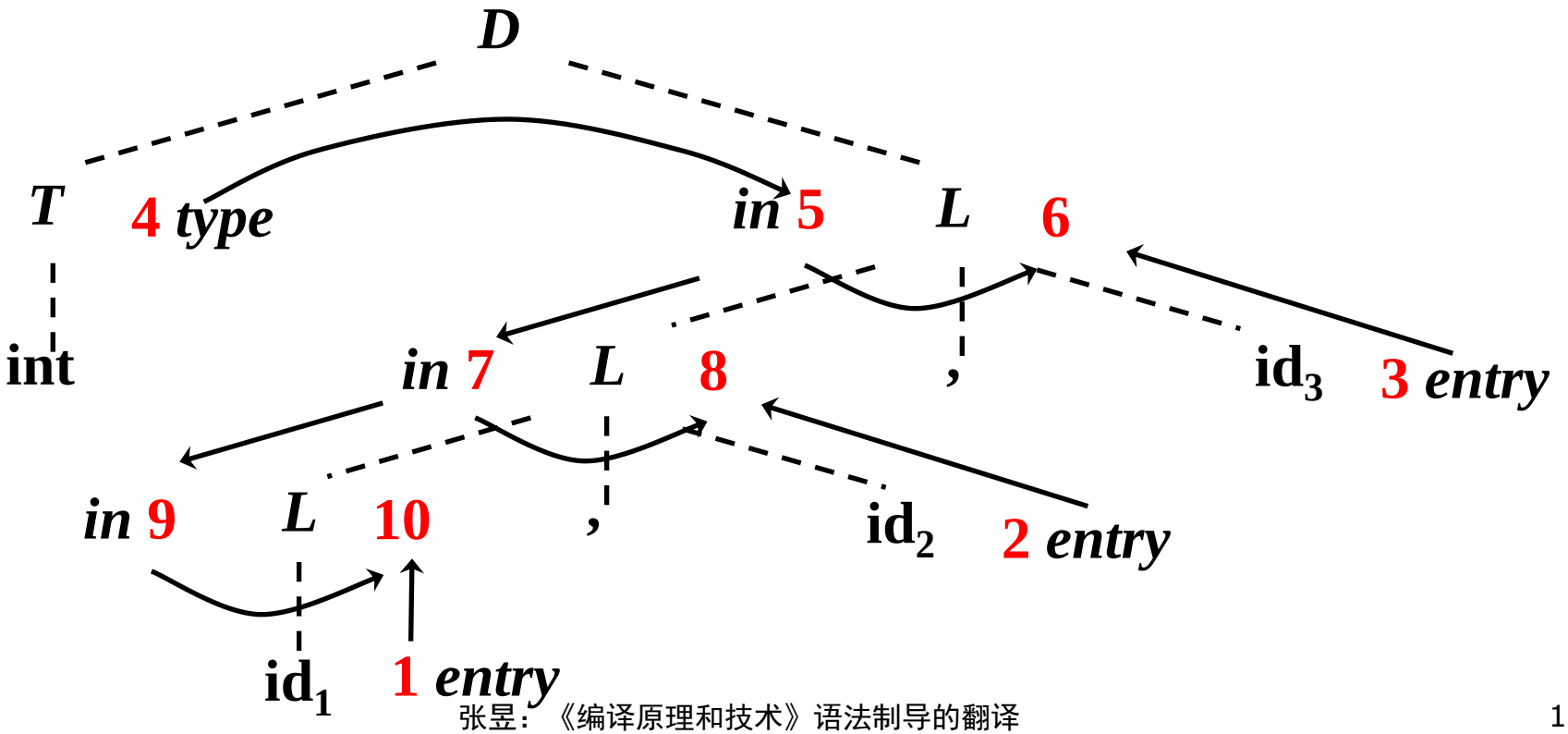
例 1，2，3，4，5，6，7，8，9，10



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



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

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
  - 翻译方案

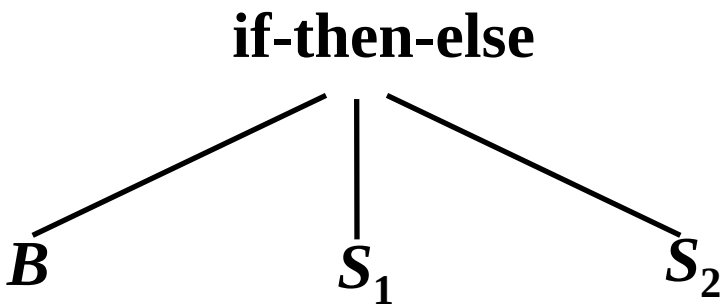
语法树(syntax tree)

- 语法树是分析树的浓缩表示

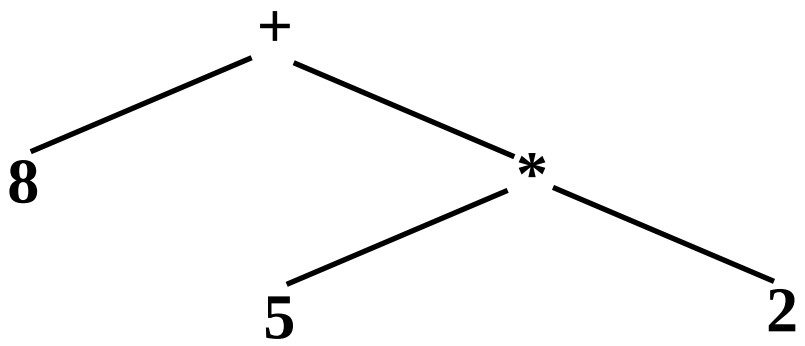
  每个结点表示一个语法构造，算符和关键字是语法树中的内部结点

  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

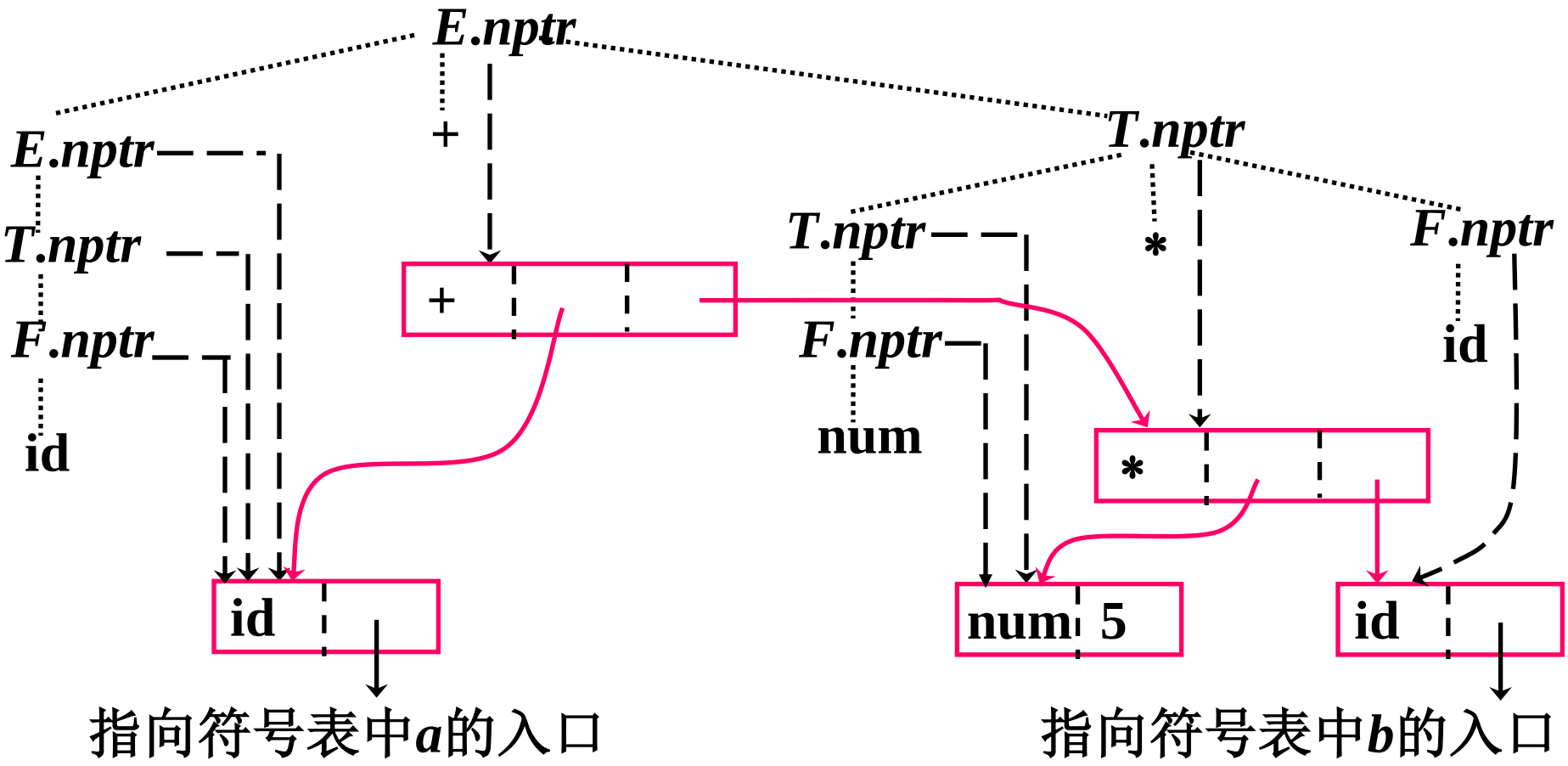
构造语法树的语法制导定义

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）

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

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

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

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

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

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

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

综合属性的计算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

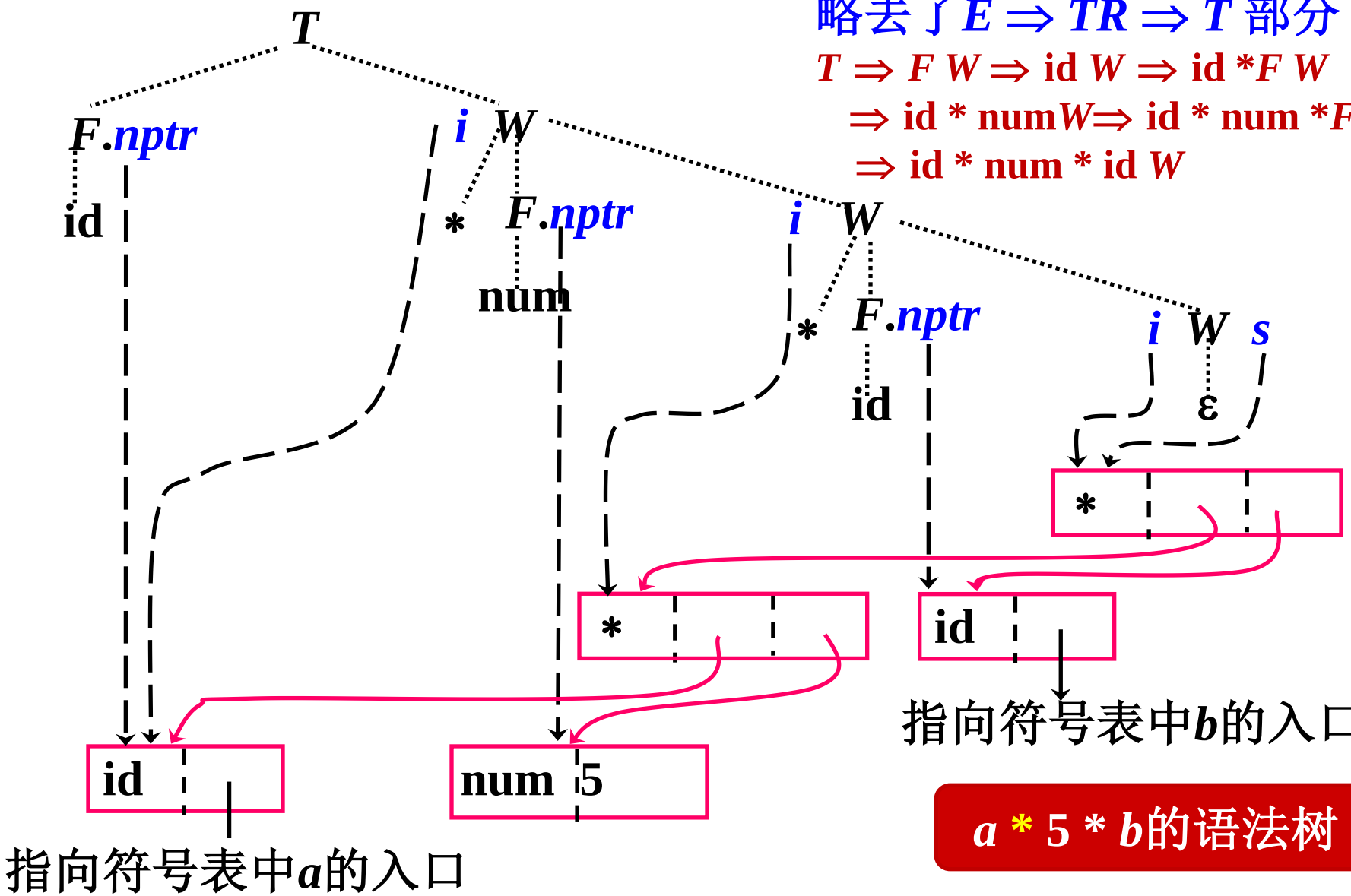
表达式语言的 *LL*文法

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

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

$E \rightarrow T$ $\quad \{R.i = T.nptr\}$ $\qquad T + T + T + \ldots$

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

$R \rightarrow +$

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

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

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

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

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

$W \rightarrow *$

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

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

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

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

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

$S' \rightarrow S$	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

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。

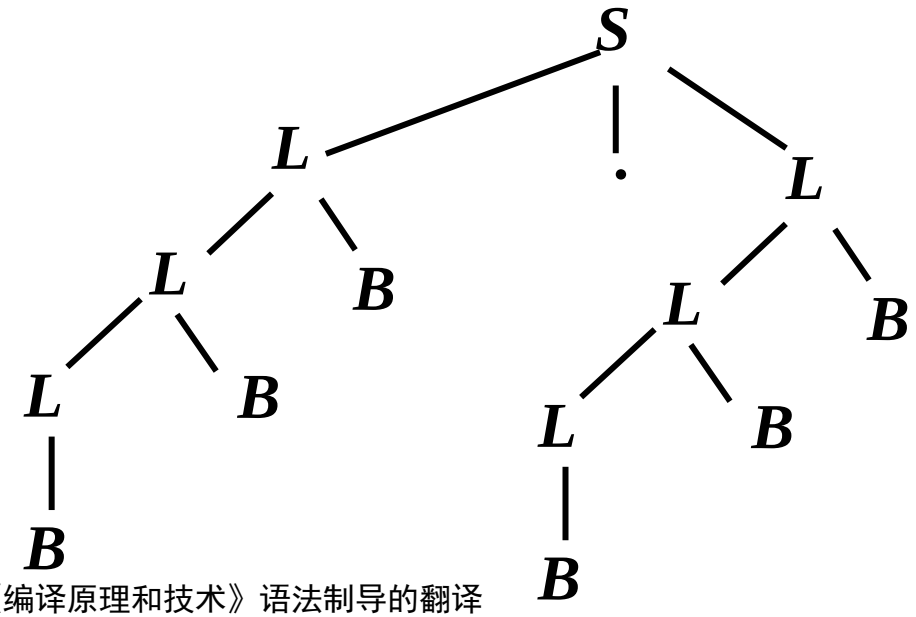
例如，输入101.101时，S.*val* = 5.625（可以修改文法）

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

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$



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。

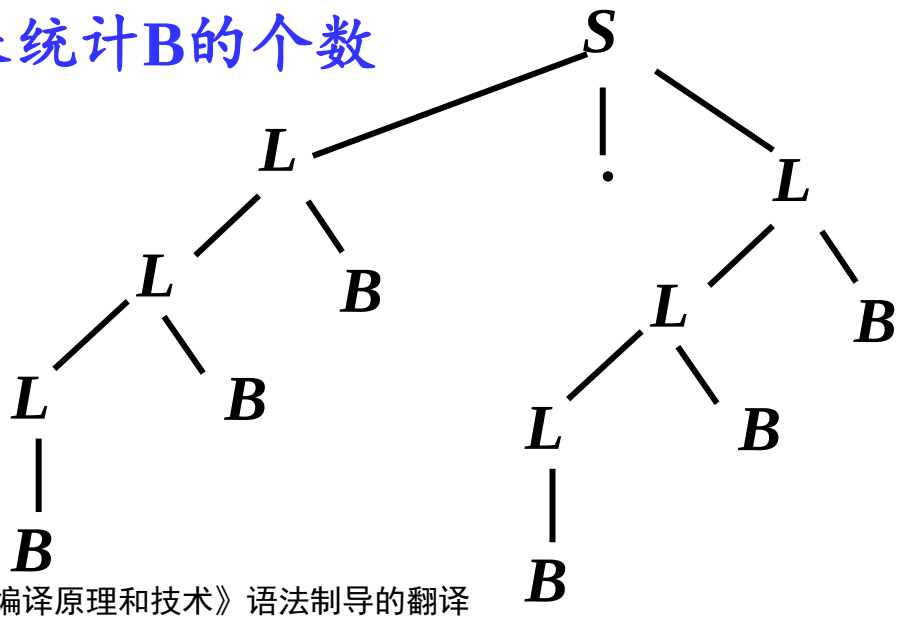
例如，输入101.101时，S.*val* = 5.625（可以修改文法）

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）

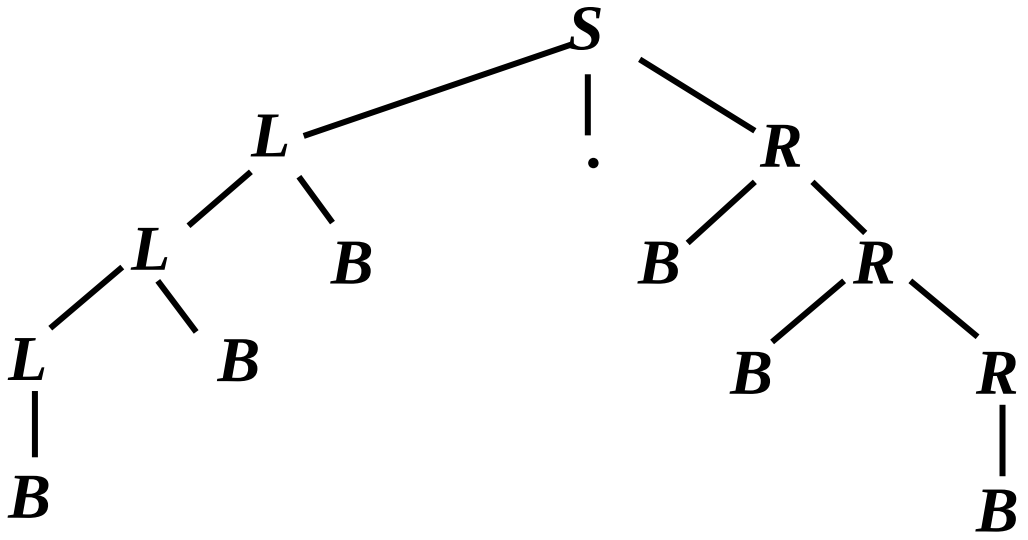
更清楚的办法是将文法改成下面的形式

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

$L \rightarrow L\ B \mid B$

$R \rightarrow B\ R \mid B$

$B \rightarrow 0 \mid 1$



例 题 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

$E.code = E_1.code \| \text{“+”} \| \text{“(”} \| T.code \| \text{“)”}$

else

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

$E.op = plus$

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

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

4.3 自上而下计算

- S属性定义、L属性定义
- 翻译方案
- 预测翻译器的设计
- 用综合属性代替继承属性

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**

L属性定义举例

变量类型声明的语法制导定义

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	$addType(\text{id}.entry, L.in)$

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 –

$E \rightarrow T\ R$

$R \rightarrow$ addop T {*print*(addop.*lexeme*)} $R_1 \mid \varepsilon$

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

$E \Rightarrow T\ R \Rightarrow$ num {*print* (8)} R

$\Rightarrow$ num {*print*(8)}addop T {*print* (+)}R

$\Rightarrow$ num {*print*(8)}addop num{*print*(5)}{*print* (+)}R

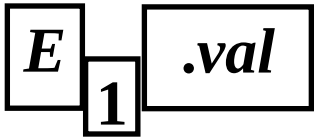
… {*print*(8)}{*print*(5)}{*print*(+)}addop T {*print*(–)}R

… {*print*(8)}{*print*(5)}{*print*(+)}{*print*(2)}{*print*(–)}

L属性定义的自上而下计算

例 数学排版语言EQN

E sub 1 .val



$S \rightarrow B$

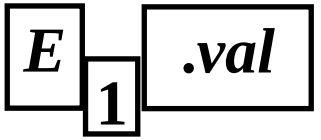
$B \rightarrow B_1\ B_2$

$B \rightarrow B_1$ sub B_2

$B \rightarrow$ text

数学排版语言EQN

语法制导定义　*E* sub 1 .val



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

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

数学排版语言EQN

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

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

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

B的继承属性ps的计算位于B的左边

数学排版语言EQN

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

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

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

S 的综合属性 ht 的计算放在S 的产生式右部的末端

数学排版语言EQN

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

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

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

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

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

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

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

B_1

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

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

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

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 : integer

$D \rightarrow L : T$	$L.in = T.type$
$T \rightarrow$ integer | char	$T.type = \ldots$
$L \rightarrow L_1$, id | id	$L_1.in = L.in, \ldots$

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 : integer

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

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

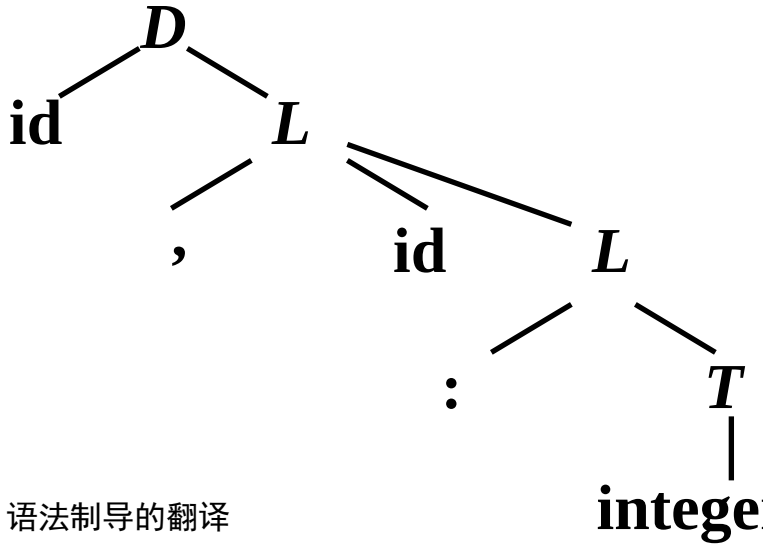
$L \rightarrow L_1, \text{id} \mid \text{id}$ $\quad L_1.in = L.in, \ldots$

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

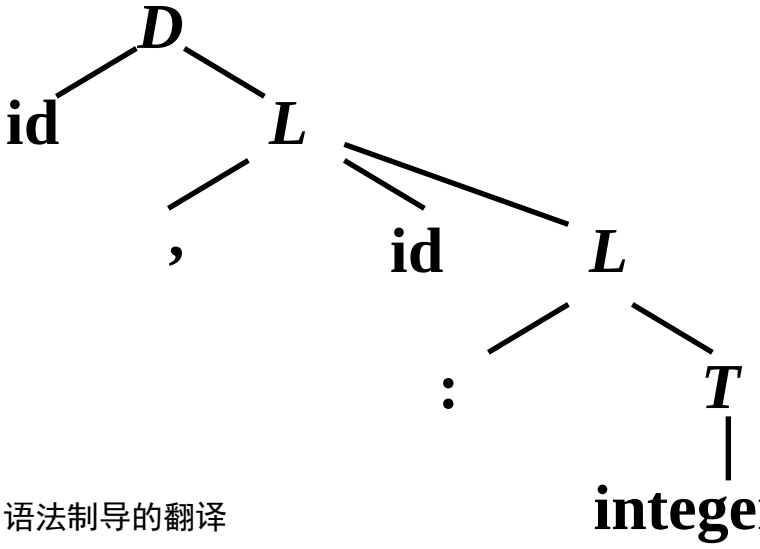
$D \rightarrow$ id L { *addtype* (id. *entry*, *L. type*)}

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

$L \rightarrow$: T {*L. type* = *T. type*}

$T \rightarrow$ integer {*T. type* = *integer*}

$T \rightarrow$ real {*T. type* = *real*}



Lab1-3: ParseTree =>AST

- AST的定义
  - syntax_tree_node

Public Member Functions

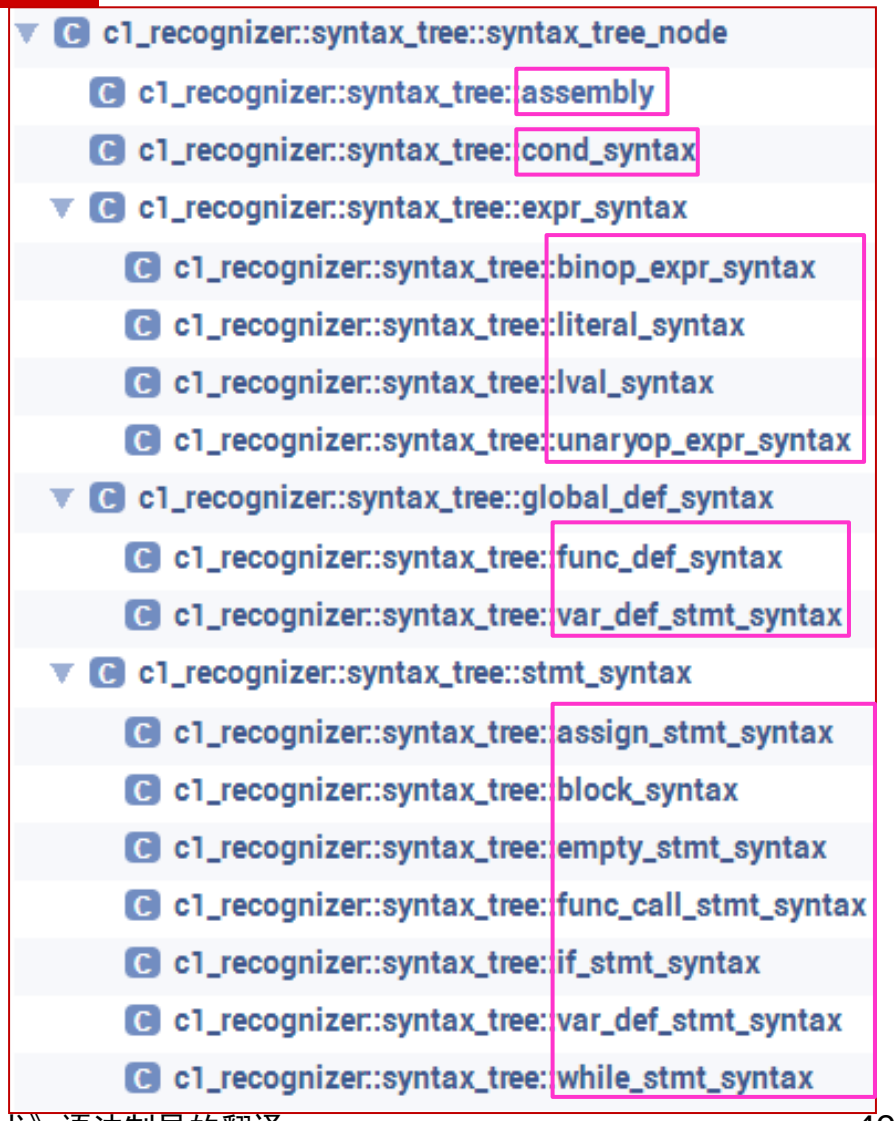
virtual void **accept** (syntax_tree_visitor &visitor)=0

Public Attributes

int **line**

int **pos**

- 访问者
  - syntax_tree_visitor



Lab1-3: ParseTree =>AST

AST的定义syntax_tree_node

```
virtual void accept (syntax_tree_visitor &visitor)=0
```

访问者syntax_tree_visitor

```
virtual void visit (assembly &node)=0
virtual void visit (func_def_syntax &node)=0
virtual void visit (cond_syntax &node)=0
virtual void visit (binop_expr_syntax &node)=0
virtual void visit (unaryop_expr_syntax &node)=0
virtual void visit (lval_syntax &node)=0
virtual void visit (literal_syntax &node)=0
virtual void visit (var_def_stmt_syntax &node)=0
virtual void visit (assign_stmt_syntax &node)=0
virtual void visit (func_call_stmt_syntax &node)=0
virtual void visit (block_syntax &node)=0
virtual void visit (if_stmt_syntax &node)=0
virtual void visit (while_stmt_syntax &node)=0
virtual void visit (empty_stmt_syntax &node)=0
```

- c1_recognizer::syntax_tree::syntax_tree_node
  - c1_recognizer::syntax_tree::assembly
  - c1_recognizer::syntax_tree::cond_syntax
  - c1_recognizer::syntax_tree::expr_syntax
    - c1_recognizer::syntax_tree::binop_expr_syntax
    - c1_recognizer::syntax_tree::literal_syntax
    - c1_recognizer::syntax_tree::lval_syntax
    - c1_recognizer::syntax_tree::unaryop_expr_syntax
  - c1_recognizer::syntax_tree::global_def_syntax
    - c1_recognizer::syntax_tree::func_def_syntax
    - c1_recognizer::syntax_tree::var_def_stmt_syntax
  - c1_recognizer::syntax_tree::stmt_syntax
    - c1_recognizer::syntax_tree::assign_stmt_syntax
    - c1_recognizer::syntax_tree::block_syntax
    - c1_recognizer::syntax_tree::empty_stmt_syntax
    - c1_recognizer::syntax_tree::func_call_stmt_syntax
    - c1_recognizer::syntax_tree::if_stmt_syntax
    - c1_recognizer::syntax_tree::var_def_stmt_syntax
    - c1_recognizer::syntax_tree::while_stmt_syntax

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

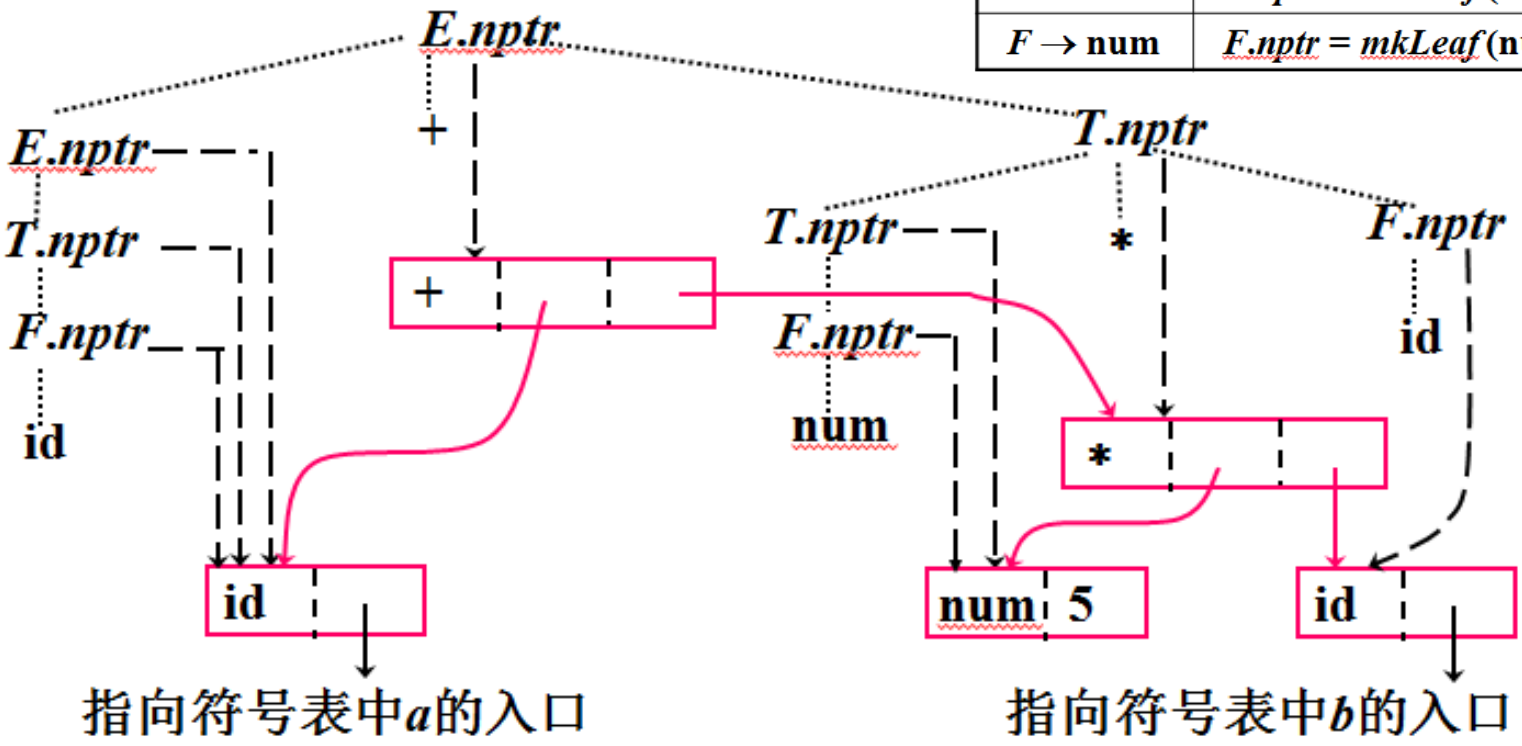
4.4 自下而上计算

- 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

S属性定义举例

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*)

a+5**b*的语法树的构造



S属性的自下而上计算

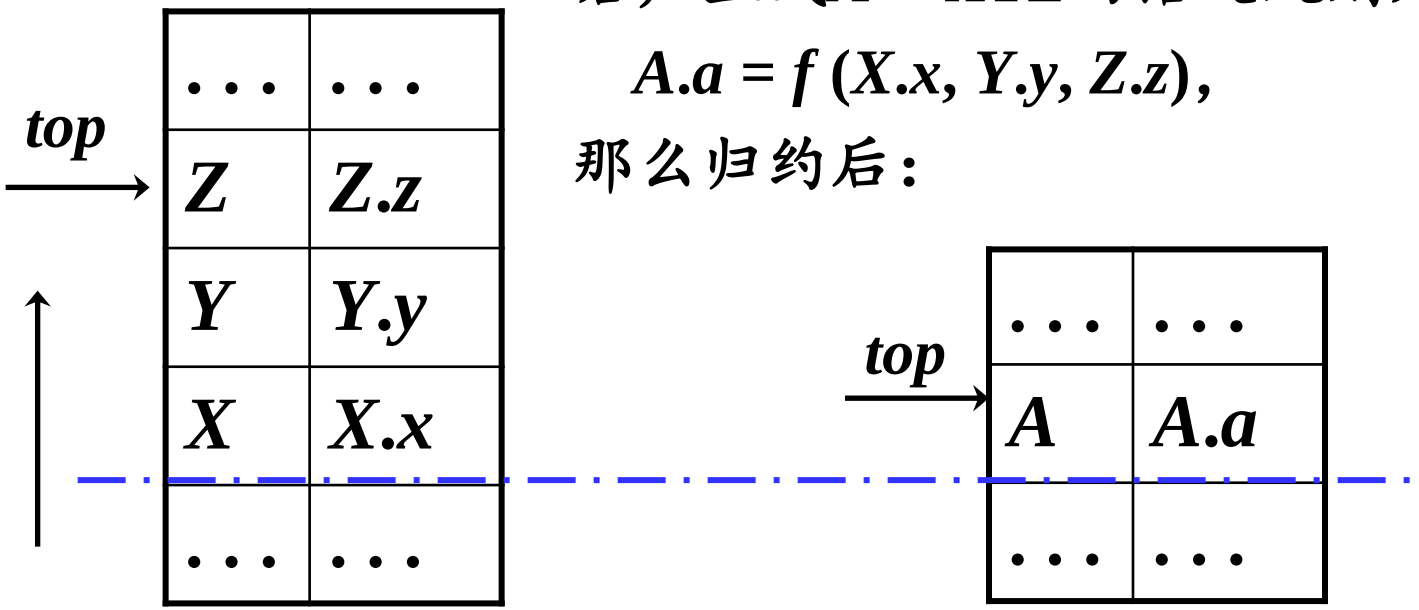
边分析边计算

LR分析器的栈增加一个域来保存综合属性值

若产生式$A \rightarrow XYZ$的语义规则是

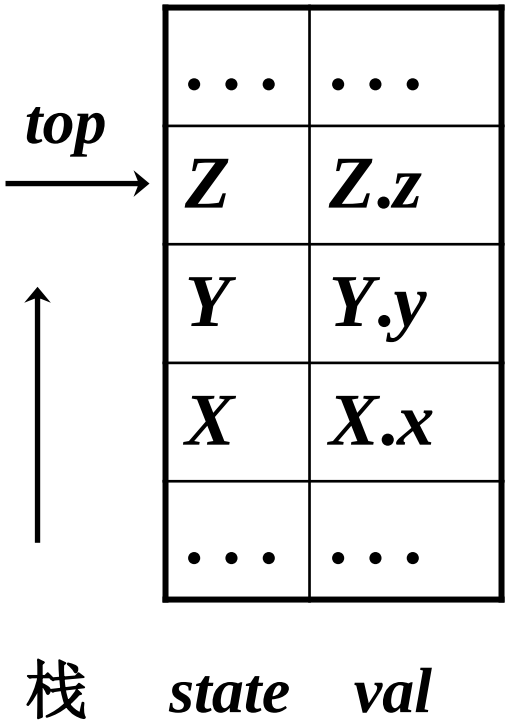
$A.a = f(X.x, Y.y, Z.z)$,

那么归约后：



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

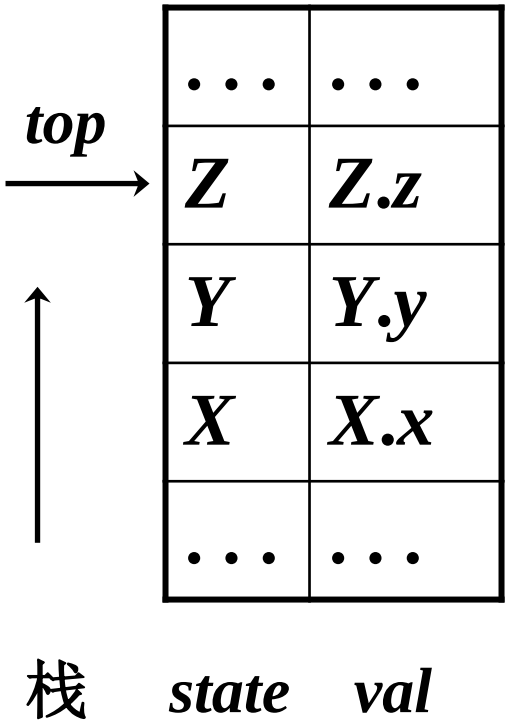


产 生 式	语 义 规 则
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.*lexval*

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

自下而上的翻译

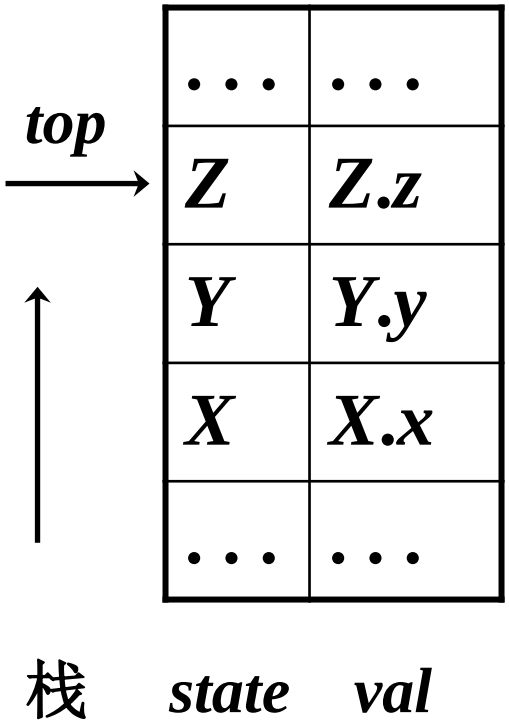
例　简单计算器的语法制导定义改成栈操作代码



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

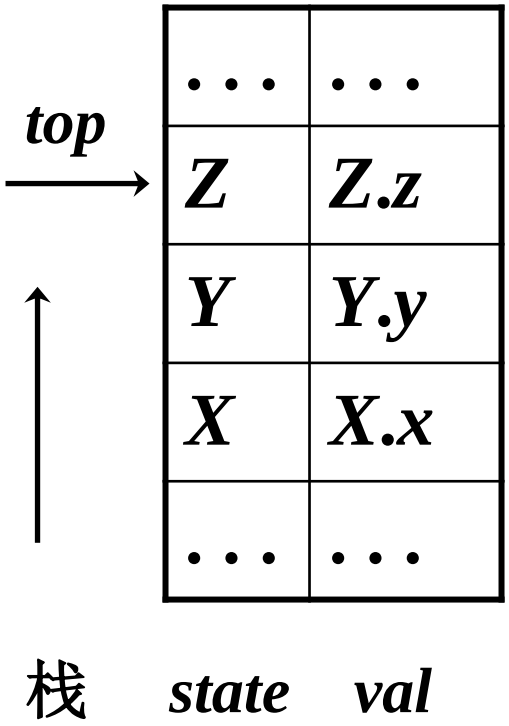


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1])
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.*lexval*

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

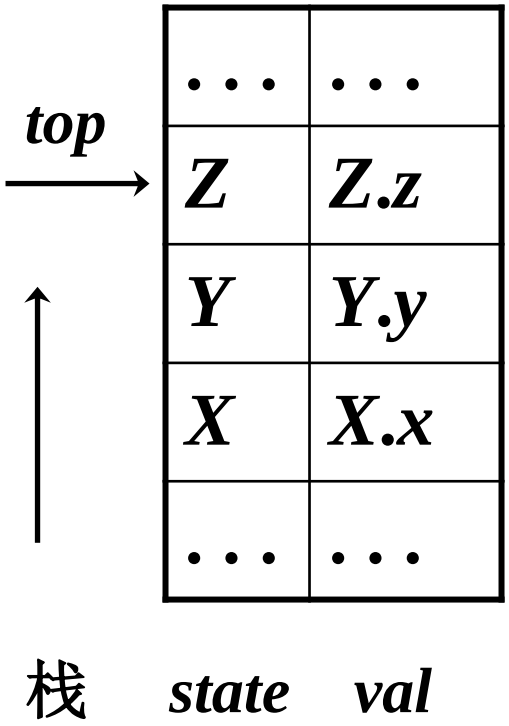


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1]);
$E \rightarrow E_1 + T$	*val*[*top*–2] = *val* [*top*–2] +*val*[*top*];
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	*T.val* = T_1.*val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ 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

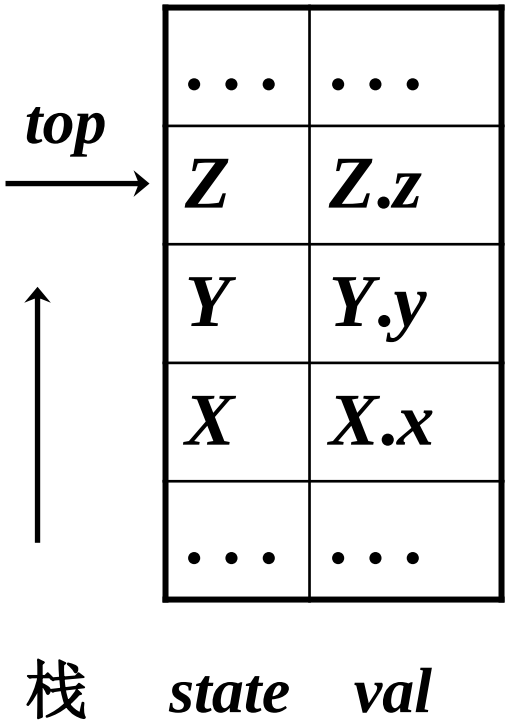


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1]);
$E \rightarrow E_1 + T$	*val*[*top*–2] = *val* [*top*–2] +*val*[*top*];
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*T.val* = T_1*.val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = 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

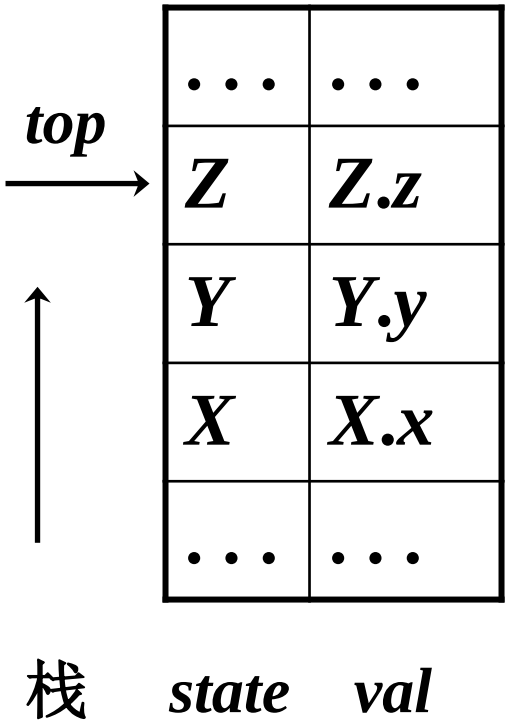


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1]);
$E \rightarrow E_1 + T$	*val*[*top*–2] = *val* [*top*–2] +*val*[*top*];
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*top*–2] = *val* [*top*–2] × *val*[*top*];
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ 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

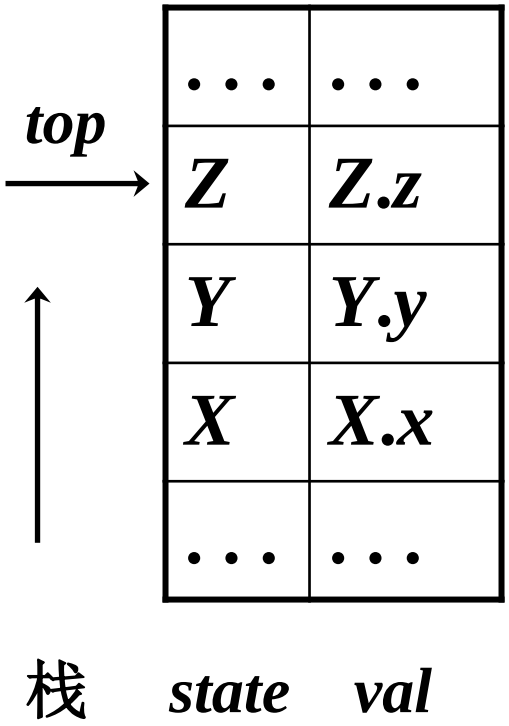


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1]);
$E \rightarrow E_1 + T$	*val*[*top*–2] = *val* [*top*–2] +*val*[*top*];
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*top*–2] = *val* [*top*–2] × *val*[*top*];
$T \rightarrow F$	
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*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

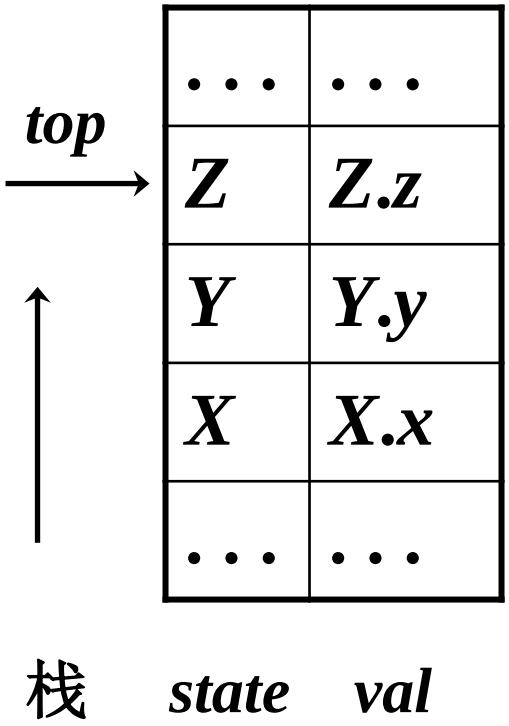


产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val*[*top*–1]);
$E \rightarrow E_1 + T$	*val*[*top*–2] = *val* [*top*–2] +*val*[*top*];
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*top*–2] = *val* [*top*–2] × *val*[*top*];
$T \rightarrow F$	
$F \rightarrow (E)$	*val*[*top*–2] = *val* [*top*–1] ;
$F \rightarrow$ 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



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] × *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
%%
```

```
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

$E \rightarrow T\ R$

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

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

在文法中加入产生ε的标记非终结符，让每个嵌入动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=>综合属性）

$E \rightarrow T\ R$

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

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

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

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

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ε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

```
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;
```

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

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 int p, q, r

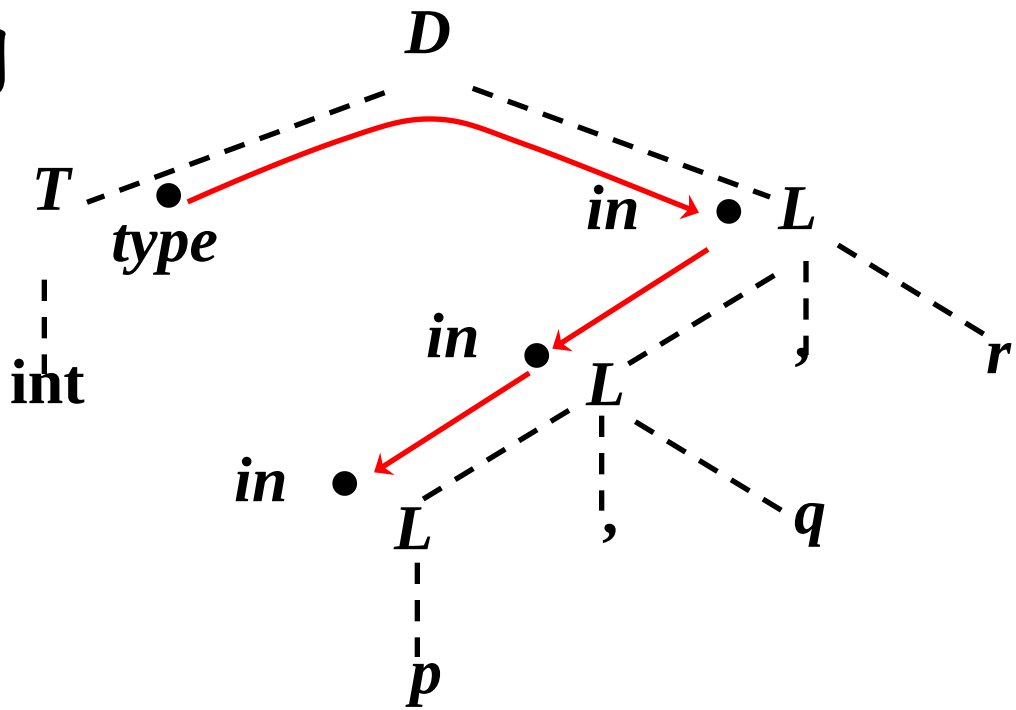
$D \rightarrow T$ {*L.in* = *T.type*}
L

$T \rightarrow$ int {*T. type* = *integer*}

$T \rightarrow$ real {*T. type* = *real*}

$L \rightarrow$ {L_1.*in* = *L.in* }
L_1, id {*addtype* (id.*entry*, *L.in*)}

$L \rightarrow$ id {*addtype* (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

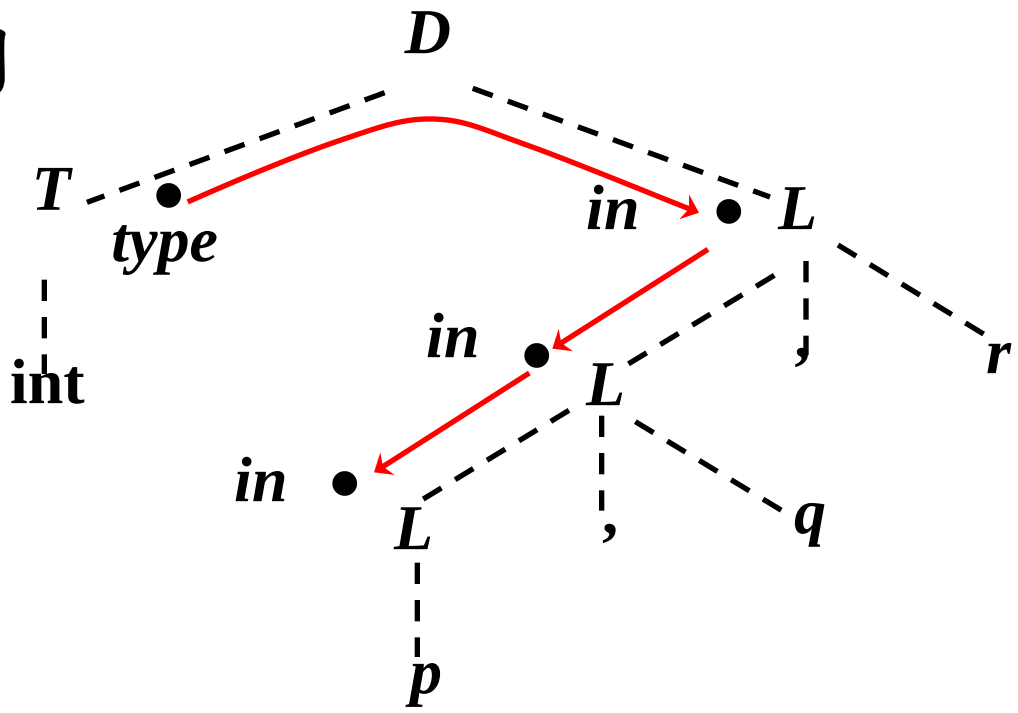
$D \rightarrow T$ {~~L.in = T.type~~}
L

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

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

$L \rightarrow$ {~~L_1.in = L.in~~}
L_1, id {addtype (id.entry, L.in)}

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

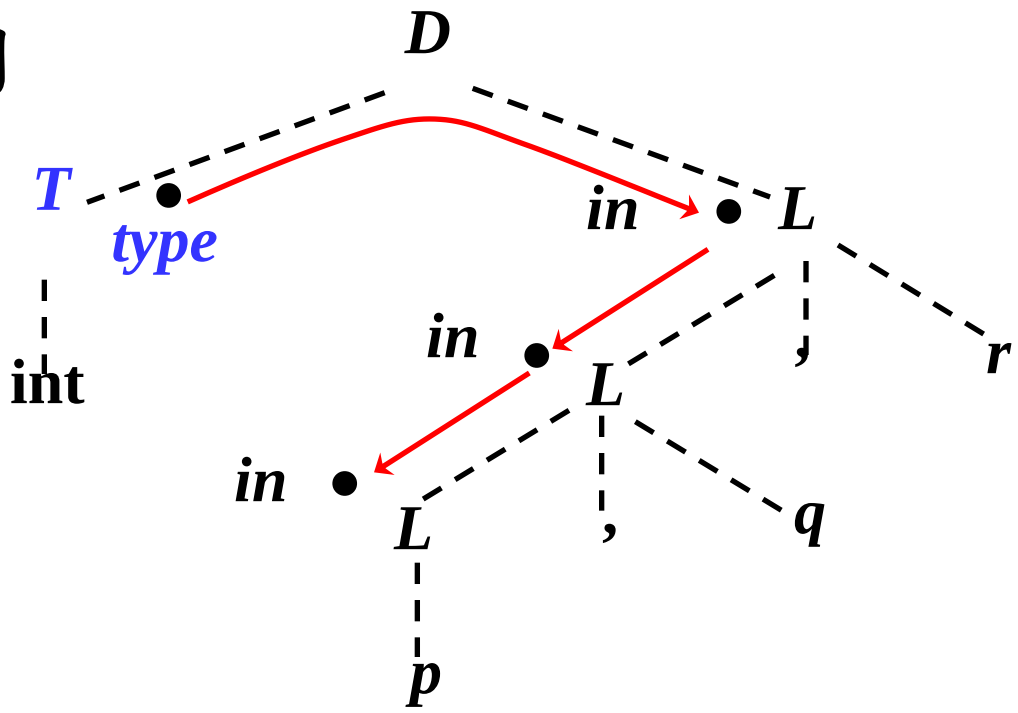


略去继承属性的计算
引用继承属性的地方改成
引用其他符号的综合属性

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
$D \rightarrow TL$	
$T \rightarrow$ int	*val*[*top*] = *integer*
$T \rightarrow$ 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 { …
                  $<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]
       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

继承属性值已在分析栈中

情况2 属性位置不可预测

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABC$	$C.i = A.s$
$C \rightarrow c$	$C.s = g(C.i)$

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

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABMC$	$M.i = A.s;\ C.i = M.s$
$C \rightarrow c$	$C.s = g(C.i)$
$M \rightarrow \varepsilon$	$M.s = M.i$

模拟继承属性的计算

- 继承属性是综合属性的函数

$S \rightarrow aAC$　　$C.i = f(A.s)$

$C \rightarrow c$　　$C.s = g(C.i)$

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

$S \rightarrow aANC$　　$N.i = A.s;\ C.i = N.s$

$N \rightarrow \varepsilon$　　$N.s = f(N.i)$

$C \rightarrow c$　　$C.s = g(C.i)$

数学排版语言EQN

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

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

EQN：自下而上计算的实现

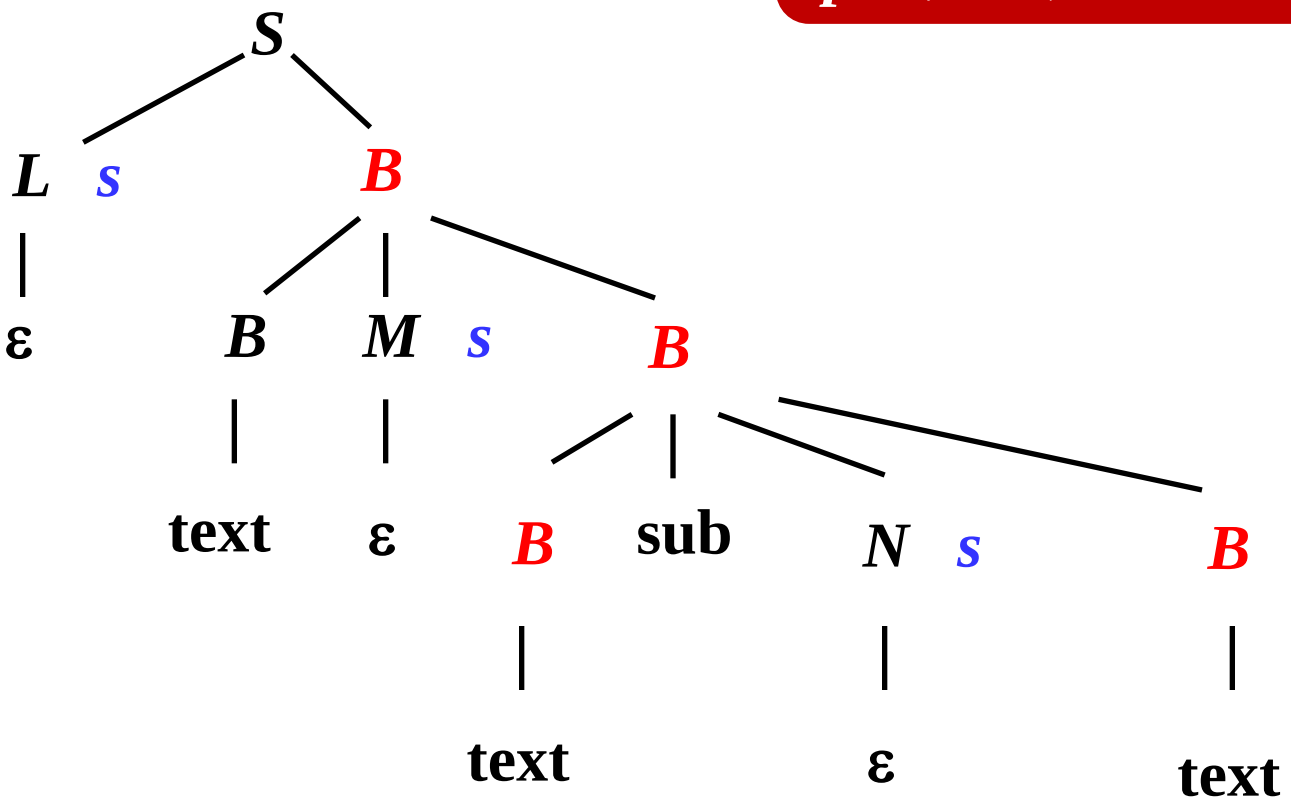
产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$; $B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ 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
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$ 兼有计算功能
$B \rightarrow$ text	$B.ht =$ 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
$S \rightarrow LB$	$B.ps = L.s;\ S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ 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
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s;B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ 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
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ 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
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ 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
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ 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
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top])$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

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

EQN：自下而上计算的实现

产 生 式	语 义 规 则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top])$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$B.ht = text.h \times B.ps$

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

EQN：自下而上计算的实现

产 生 式	语 义 规 则
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top])$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$B.ht =$ text.$h \times val[top-1]$

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