第五章 语法制导翻译

学习重点

- 语法制导定义概念
- 利用语法制导定义构造语法树
- S-属性定义、L-属性定义
- 自顶向下计算属性
- 自底向上计算属性
- 递归方法计算属性
- 内存分配

概述

输入串 → 语法树 → 依赖图 → 语义规则执行顺序

- 语法制导定义
- 翻译模式
- “编程语言的翻译根据语法进行”
- “属性”，attribute
- 每个语法符号与若干属性相关联
- 翻译——指定属性的相互依赖关系
- 语义规则，semantic rule
- 语言规则的执行反映属性的相互关系

5.1 语法制导定义

- 扩充CFG
  - 语法符号←→**属性**——语法树节点，记录域
  - 产生式←→**语义规则**——语法树节点，用于计算属性
- 属性类型　　综合属性：孩子　　继承属性：父、兄弟
  - 综合，synthesized，根据孩子节点属性计算
  - 继承，inherited，由父、兄弟节点属性计算
- 依赖图，dependency graph
- 注释语法树：节点属性值计算完毕

  annotated parse tree, annotating, decorating

5.1.1 语法制导定义的形式

- 每个产生式$A \rightarrow \alpha$与一组语义规则相关联，每个语义规则具有如下形式：
  - $b = f(c_1, c_2, \ldots, c_k)$，两种可能情况
  - b为A的综合属性，$c_1, c_2, \ldots, c_k$为A、α中语法符号的属性
  - b为α中某个符号的继承属性，$c_1, c_2, \ldots, c_k$为A、α中语法符号的属性
  - b依赖$c_1, c_2, \ldots, c_k$
- **属性文法**：扩充了语法制导定义，无副作用

例5.1

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** $.lexval$

- digit.lexval：终结符只有综合属性，由词法分析器提供
- 开始符号通常没有继承属性

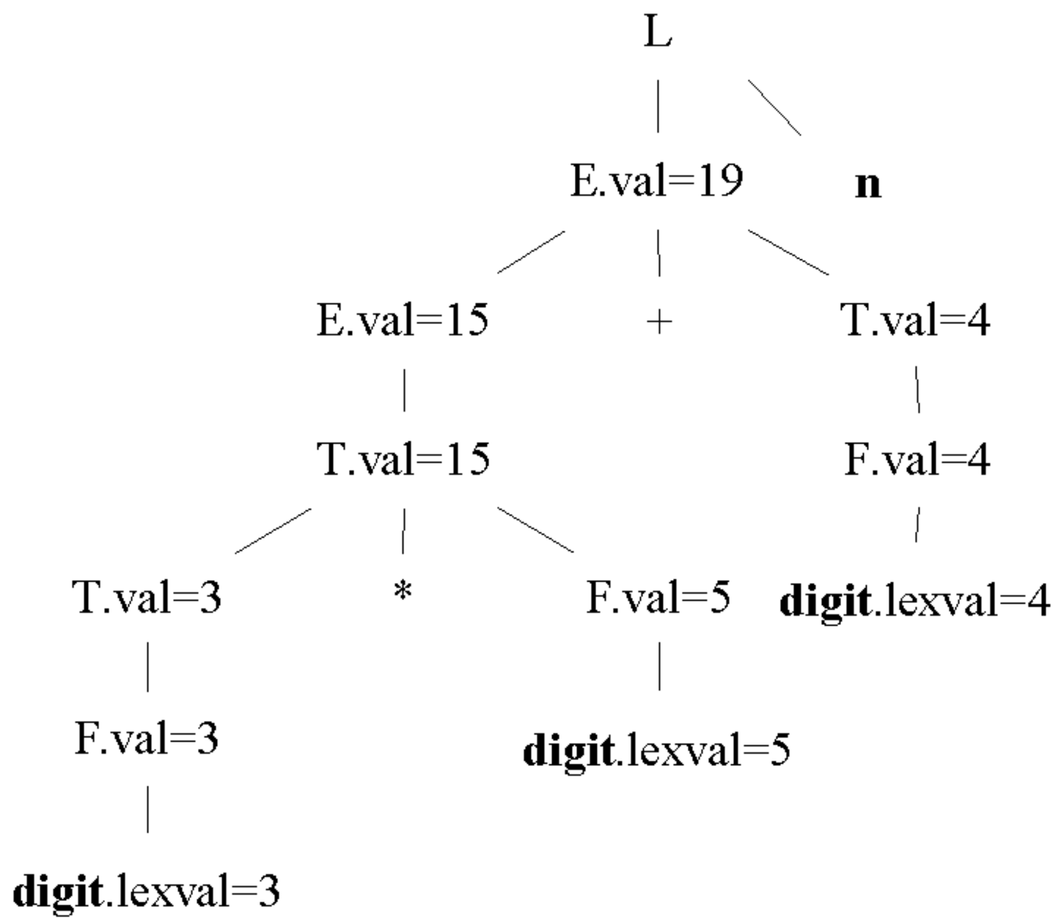
5.1.2 综合属性

- 只有综合属性：**S-属性定义**
- 语法树自底向上计算属性

根据孩子节点计算

自底向上

例5


L
E.val=19
n
E.val=15
+
T.val=4
T.val=15
F.val=4
T.val=3
*
F.val=5
digit.lexval=4
F.val=3
digit.lexval=5
digit.lexval=3

5.1.3 继承属性

- 表达程序语言结构在上下文中的相互依赖关系更加自然、方便

自顶向下计算

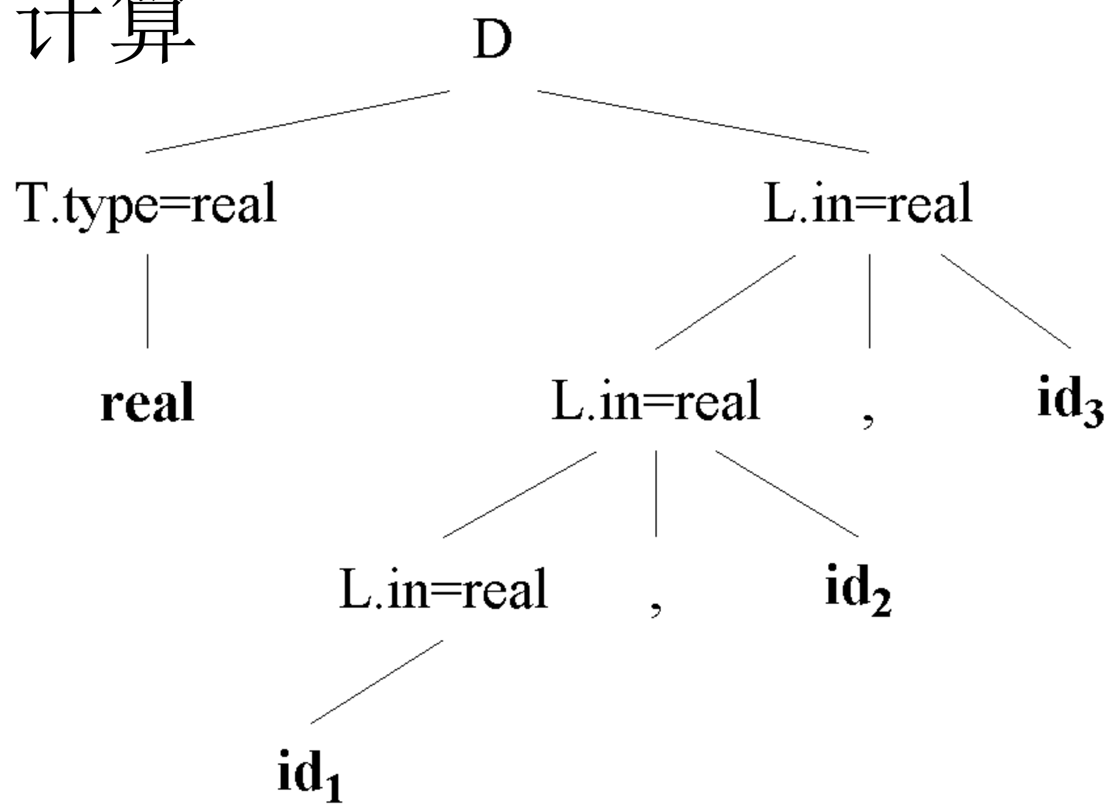
例5.3 变量定义

real id$_1$, id$_2$, id$_3$; 兄弟节点

产生式	语义规则
$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow$ **int**	$T.type = integer$
$T \rightarrow$ **real**	$T.type = real$
$L \rightarrow L_1$ **, id**	$L_1.in = L.in$ $addtype(\textbf{id}.entry, L.in)$
$L \rightarrow$ **id**	$addtype(\textbf{id}.entry, L.in)$

例5.3（续）

- 自顶向下计算



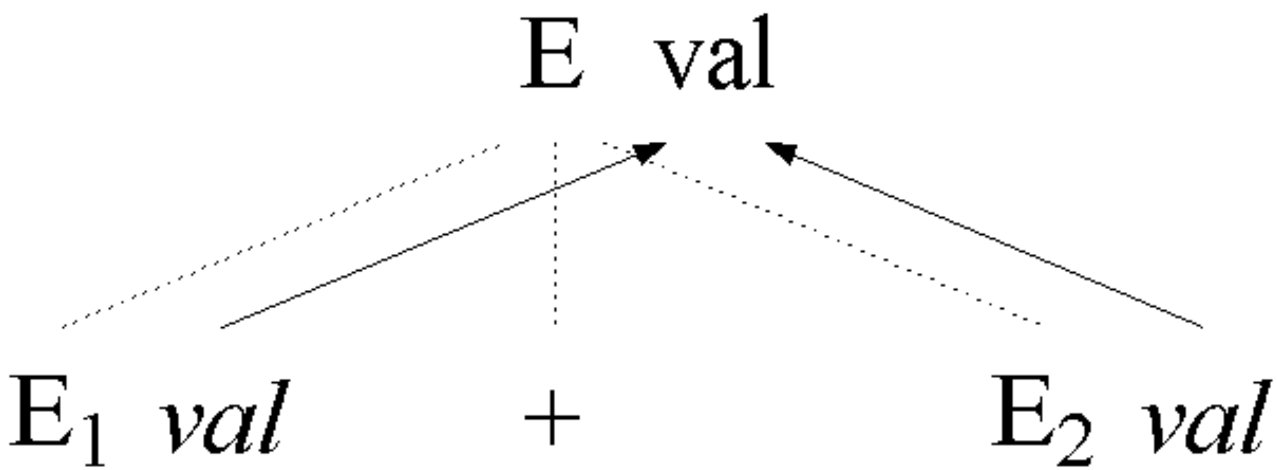
5.1.4 依赖图

- 属性b依赖属性c，则b应在c之后计算
- 有向图表示这种依赖关系——依赖图
- 构造方法：

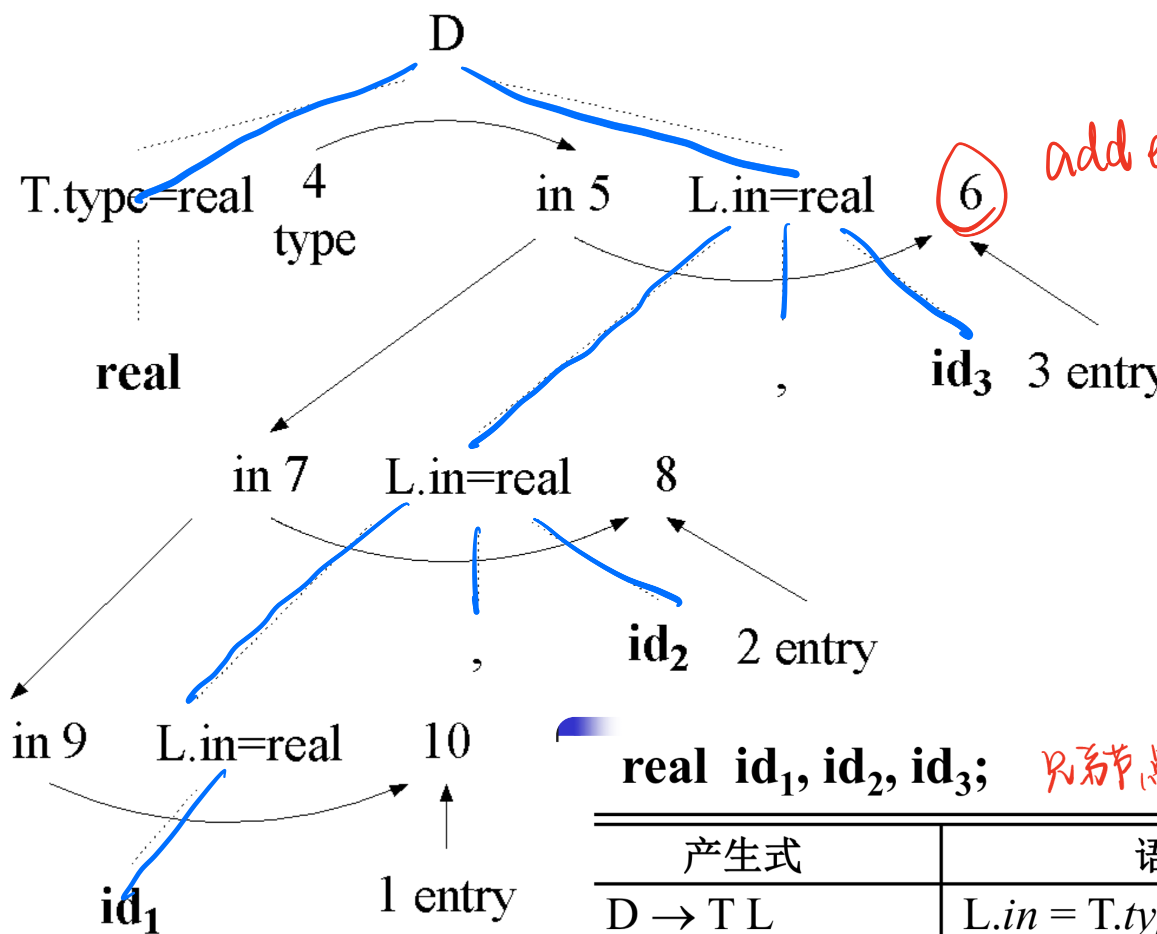
for 语法树中每个节点n **do**
    for n的每个语法符号的属性a **do**
        在依赖图中为a构造一个节点
for 语法树中每个节点n **do**
    for n使用的产生式的每个语义规则$b = f(c_1, c_2, \ldots, c_k)$ **do**
        for i = 1 to k **do**
            构造从c_i到b的一条边

例5.4

- $E \rightarrow E_1 + E_2,\quad E.val = E_1.val + E_2.val$



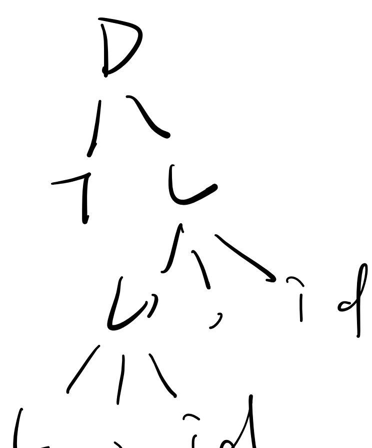
例5.5



add entry (id.entry, L.in).

real id$_1$, id$_2$, id$_3$; 兄弟节点

产生式	语义规则
D → T L	L.*in* = T.*type*
T → **int**	T.*type* = *integer*
T → **real**	T.*type* = *real*
L → L_1 **, id**	L_1.*in* = L.*in* *addtype*(**id**.*entry*, L.*in*)
L → **id**	*addtype*(**id**.*entry*, L.*in*)



5.1.5 计算顺序

- 拓扑排序，计算顺序满足依赖关系 $m_1, m_2, \ldots, m_k$，存在边 $m_i \rightarrow m_j$ ←→ $i < j$
- 计算 $b=(c_1, c_2, \ldots, c_k)$ 时，属性值 $c_1, c_2, \ldots, c_k$ 已经计算出来
- 文法→语法树→依赖图→拓扑排序→语义规则计算顺序→输入串翻译

例5.6

- 例5.5依赖图的边：小数字→大数字
- 按编号排列→满足要求的拓扑排序

a_4=real;

a_5=real;

addtype(id_3.entry, a_5); 6

a_7=a_5;

addtype(id_2.entry, a_7); 8

a_9=a_7;

addtype(id_1.entry, a_9); 10

从上往下传递

real

5.2 构造语法树

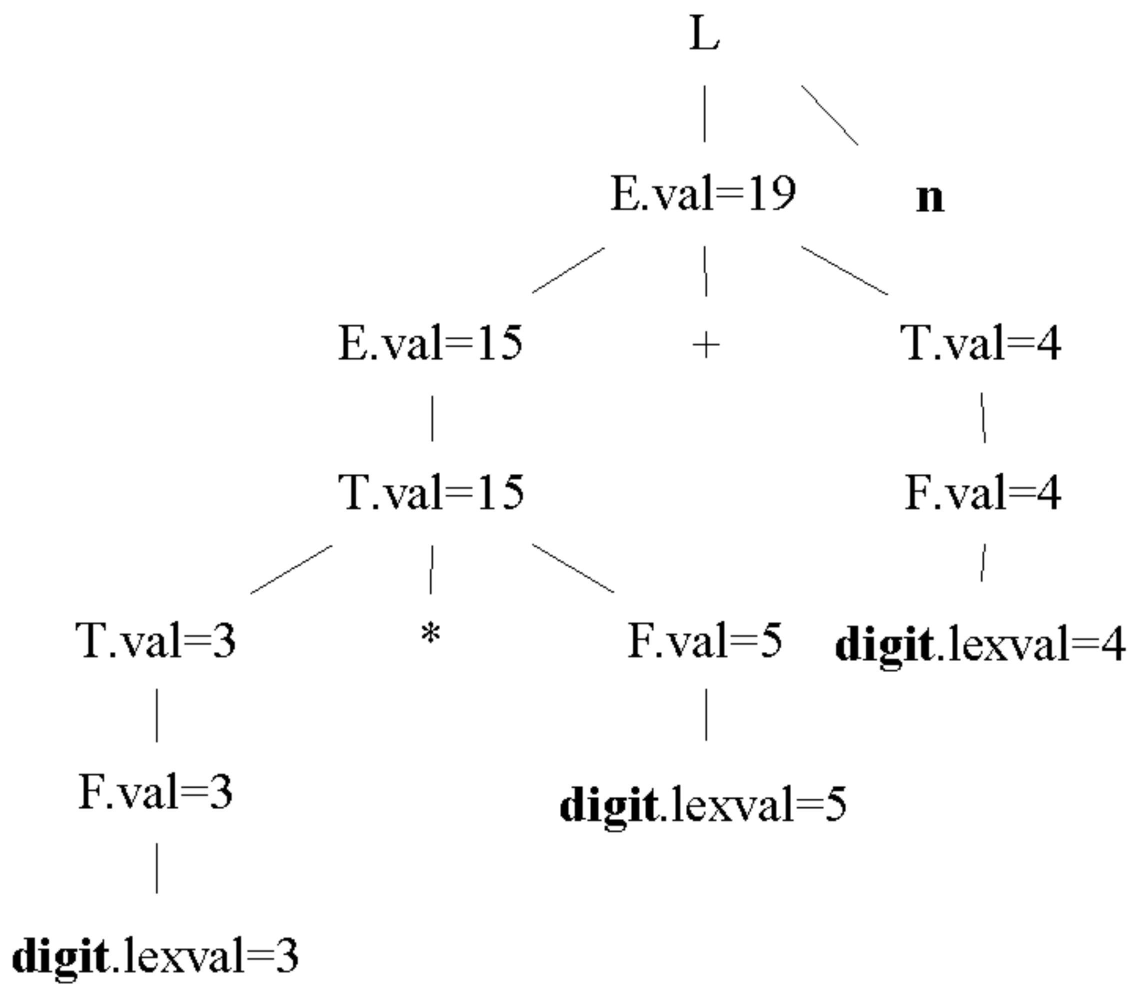
- 作为中间表示形式——分离分析与翻译
- 在进行语法分析的同时进行翻译存在缺陷：
  - 适合分析的文法可能未反映自然的语言结构
  - 分析顺序可能与翻译顺序不一致
- 利用语法制导翻译方法来构造语法树

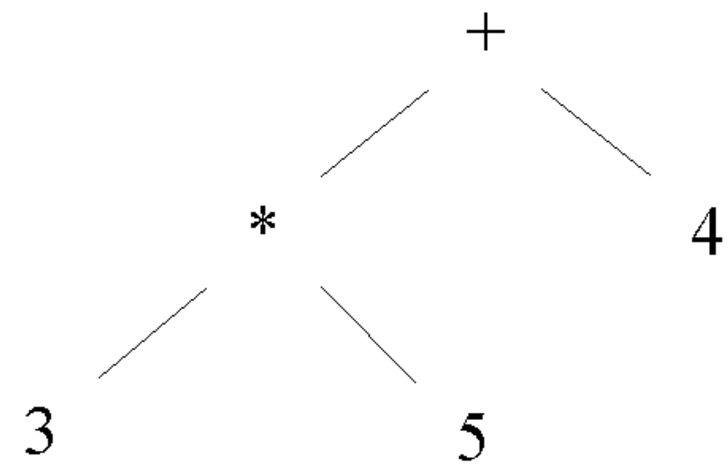
5.2.1 语法树

- （抽象）语法树，压缩形式
  - 关键字和运算符均在内部节点
  - 链式结构会被压缩

关键字与运算符

语法树压缩例


L
E.val=19
n
E.val=15
+
T.val=4
T.val=15
F.val=4
T.val=3
*
F.val=5
digit.lexval=4
F.val=3
digit.lexval=5
digit.lexval=3


+
*
4
3
5

5.2.2 表达式语法树的构造

- 与表达式翻译为后缀形式类似
- 数据结构：语法树每个节点用一个记录表示
  - 运算符节点记录格式：

```
{
  运算符
  指向运算对象节点1的指针
  执行运算对象节点2的指针
 …
}
```

辅助函数

mknode(op, left, right)：为运算符op创建语法树中节点，标记（运算符）为op，运算对象节点指针left和right

mkleaf(**id**, entry)：为标识符创建语法树节点，标记为**id**，另一个域为符号表项指针entry

mkleaf(**num**, val)：为运算数创建节点，标记为**num**，另一个域为数值

num，数值

例5.7 a-4+c的语法树

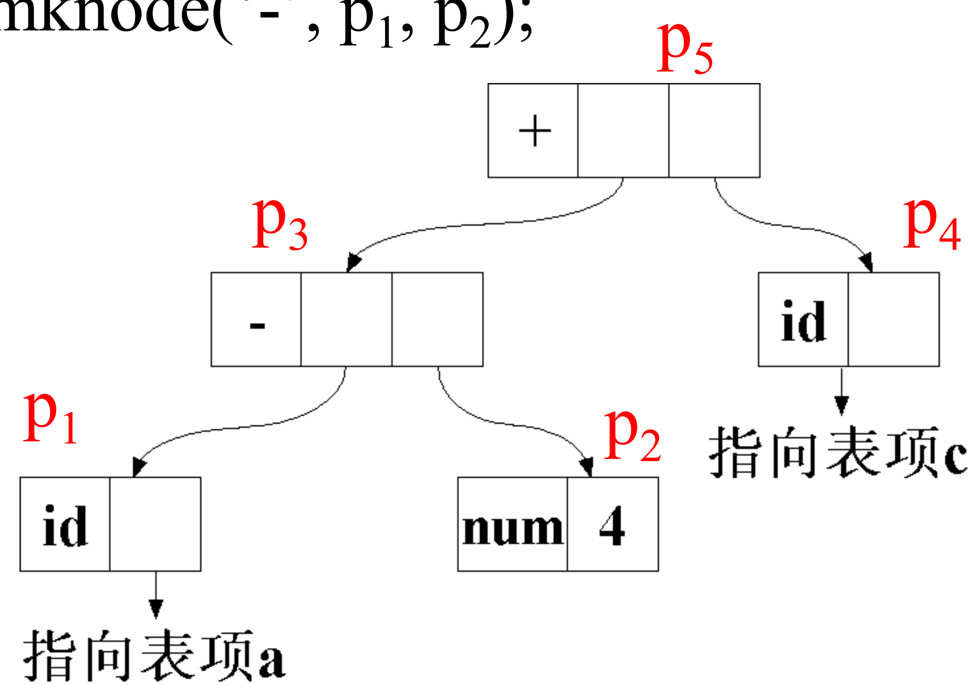
(1) p_1=mkleaf(**id**, entry_a);

(2) p_2=mkleaf(**num**, 4);

(3) p_3=mknode('-', p_1, p_2);

(4) p_4=mkleaf(**id**, entry_c);

(5) p_5=mknode('+', p_3, p_4);

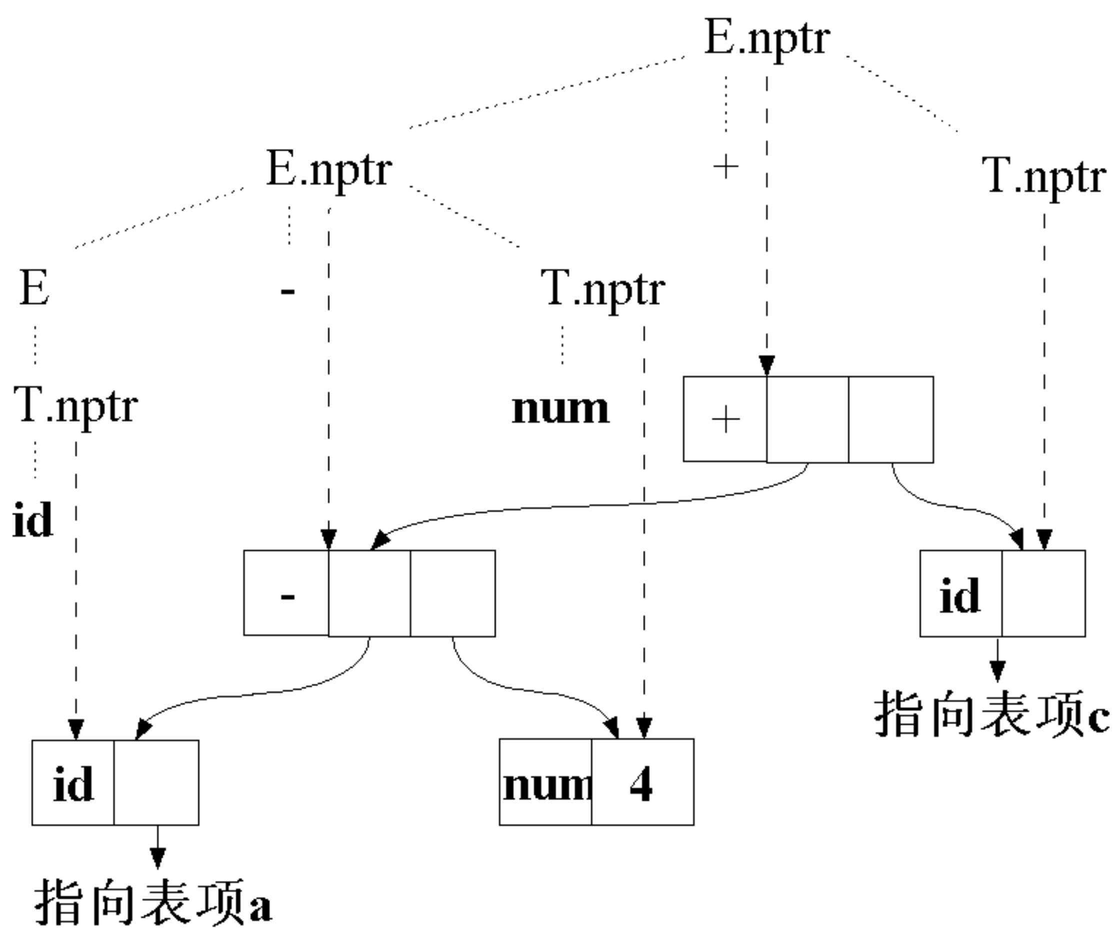


5.2.3 构造语法树的语法制导定义

- 例5.8

产生式	语义规则
$E \to E_1 + T$	$E.nptr = mknode(\text{"+"}, E_1.nptr, T.nptr)$
$E \to E_1 - T$	$E.nptr = mknode(\text{"-"}, E_1.nptr, T.nptr)$
$E \to T$	$E.nptr = T.nptr$
$T \to (E)$	$T.nptr = E.nptr$
$T \to$ **id**	$T.nptr = mkleaf(\mathrm{id}, \mathbf{id}.lexval)$
$T \to$ **num**	$T.nptr = mkleaf(\mathrm{num}, \mathbf{num}.val)$

例5.8（续）



5.2.4 用有向无环图表示表达式

- 公共子表达式用公共节点表示
- 可能出现一个节点有多个“父节点”的情况
- 构造方法
  - 类似语法树构造
  - 构造节点前检查是否已构造相同节点

图5.9

- a+a*(b-c)+(b-c)*d

(1) p_1=mkleaf(**id**, a);

(2) p_2=mkleaf(**id**, a)(=p_1);

(3) p_3=mkleaf(**id**, b);

(4) p_4=mkleaf(**id**, c);

(5) p_5=mknode('-', p_3, p_4);

(6) p_6=mknode('*', p_2, p_5);

(6) p_7=mknode('+', p_1, p_6);

(8) p_8=mkleaf(**id**, b)(=p_3);

(9) p_9=mkleaf(**id**, c)(=p_4);

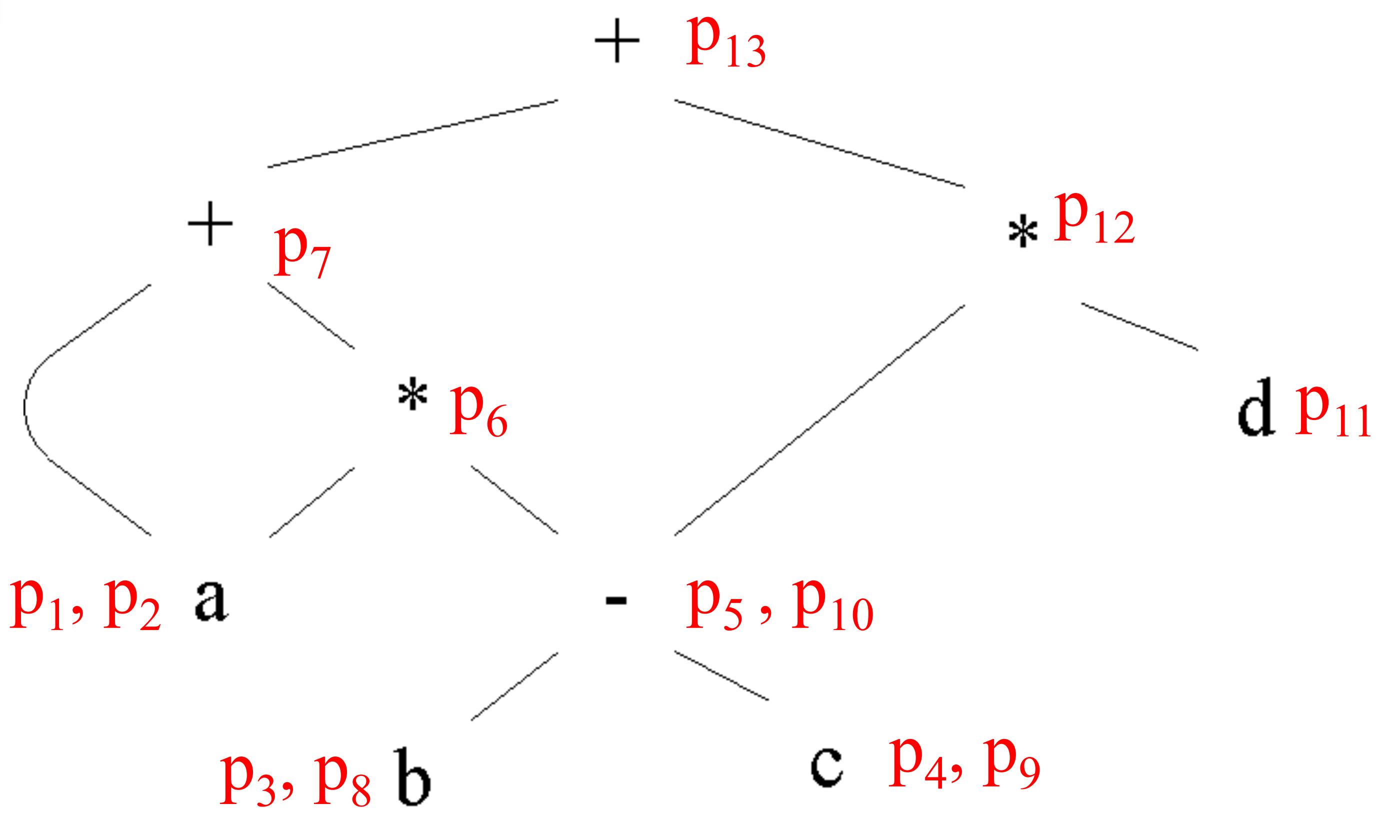
(10) p_{10}=mknode('-', p_8, p_9)(=p_5);

(11) p_{11}=mkleaf(**id**, d);

(12) p_{12}=mknode('*', p_{10}, p_{11});

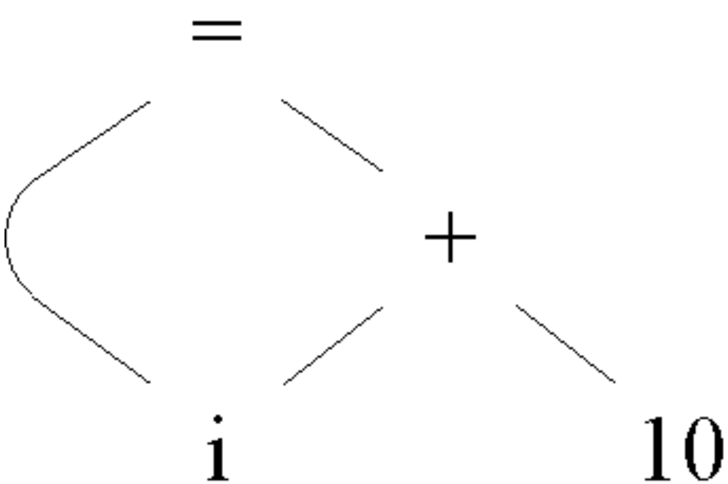
(13) p_{13}=mknode('+', p_7, p_{12});

例5.9（续）



DAG的实现

- i = i + 10



id	指向i的符号表项	
num	10	
+	1	2
=	1	3
……		

算法5.1 构造DAG节点

输入：标记（操作符）op，节点l，节点r

输出：具有<op, l, r>形式的节点

方法：

在创建节点前，搜索节点数组

寻找标记为op，左孩子l，右孩子r的节点m

若找到，直接返回m

否则，创建新节点，标记为op，左孩子为l，右孩子为r，将该节点返回

- 搜索方法：hash技术…

TinyC中的语法树

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3
typedef struct treeNode
   {
     struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
   } TreeNode;
```

TinyC中的语法树——辅助函数

```
TreeNode * newStmtNode(StmtKind kind)
{
  TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
  int i;
  if (t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->nodekind = StmtK;
    t->kind.stmt = kind;
    t->lineno = lineno;
  }
  return t;
}
```

TinyC中的语法树——辅助函数

```
TreeNode * newExpNode(ExpKind kind)
{
  TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
  int i;
  if (t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->nodekind = ExpK;
    t->kind.exp = kind;
    t->lineno = lineno;
    t->type = Void;
  }
  return t;
}
```

TinyC中的语法树——表达式

```
TreeNode * simple_exp(void)
{
  TreeNode * t = term();
  while ((token==PLUS) || (token==MINUS))
  {
    TreeNode * p = newExpNode(OpK);
     if (p!=NULL) {
      p->child[0] = t;
      p->attr.op = token;
      t = p;
      match(token);
      t->child[1] = term();
    }
  }
  return t;
}
```

TinyC中的语法树——IF

```
TreeNode * if_stmt(void)
{
  TreeNode * t = newStmtNode(IfK);
  match(IF);
  if (t!=NULL) t->child[0] = exp();
  match(THEN);
  if (t!=NULL) t->child[1] = stmt_sequence();
  if (token==ELSE) {
    match(ELSE);
    if (t!=NULL) t->child[2] = stmt_sequence();
  }
  match(END);
  return t;
}
```

TinyC中的语法树——Yacc

```
…
#define YYSTYPE TreeNode *
…
simple_exp  : simple_exp PLUS term
                { $$ = newExpNode(OpK);
                  $$->child[0] = $1;
                  $$->child[1] = $3;
                  $$->attr.op = PLUS;
                }
            | simple_exp MINUS term
                { $$ = newExpNode(OpK);
                  $$->child[0] = $1;
                  $$->child[1] = $3;
                  $$->attr.op = MINUS;
                }
            | term { $$ = $1; }
            ;
```

TinyC中的语法树——Yacc

```
if_stmt     : IF exp THEN stmt_seq END
               { $$ = newStmtNode(IfK);
                 $$->child[0] = $2;
                 $$->child[1] = $4;
               }
            | IF exp THEN stmt_seq ELSE stmt_seq END
               { $$ = newStmtNode(IfK);
                 $$->child[0] = $2;
                 $$->child[1] = $4;
                 $$->child[2] = $6;
               }
```

5.3 自底向上计算S-属性定义

- 与LR(1)分析器结合
  - 在栈中保存语法符号的属性值
  - 归约时，利用栈中语法符号（产生式右部）属性值计算新的（左部符号的）综合属性值

自底向上计算S-属性定义示例

	state	val
	…	…
	X	X.x
	Y	Y.y
top →	Z	Z.z
	…	…

$$A \rightarrow XYZ \Rightarrow$$

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

val[ntop]=f(val[top-2], val[top-1], val[top])

	state	val
	…	…
top →	A	A.a
	…	…

例5.10

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

例5.10（续）

输入	状态	val	归约用产生式
3*5+4n	-	-	
*5+4n	3	3	
*5+4n	F	3	F-->**digit**
*5+4n	T	3	T-->F
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	T * F	3 _ 5	F-->**digit**
+4n	T	15	T-->T * F
+4n	E	15	E-->T
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	F-->**digit**
n	E + T	15 _ 4	T-->F
n	E	19	E-->E + T
	E n	19 _	
	L	19	L-->E **n**

5.4 L-属性定义

- 语法分析同时进行翻译，深度优先顺序

```
procedure dfsvisit(n:node)
{   for n的每个孩子，由左至右 do
    {
          计算m的继承属性;（利用n的继承属性和m的左兄弟的属性）
          dfsvisit(m);
    }
    计算n的综合属性;
}
```

- L-属性定义
  - 可由深度优先计算
  - 包括所有基于LL(1)文法的语法制导定义

5.4.1 L-属性定义

- 一个语法制导定义，若满足如下条件，则称之为L-属性定义：
  - 对产生式$A \rightarrow X_1 \ldots X_n$，$X_j (1 \le j \le n)$的每个继承属性应仅仅依赖于
    1. X_j左边符号X_1、X_2、…、X_{j-1}的属性（继承属性和综合属性）
    2. A的继承属性
- 所有S-属性定义都是L-属性定义

例5.11

产生式	语义规则
$A \to LM$	$L.i = f_1(A.i)$ $M.i = f_2(L.s)$ $A.s = f_3(M.s)$
$A \to QR$	$R.i = f_4(A.i)$ $Q.i = f_5(R.s)$ $A.s = f_6(Q.s)$

不是L-属性定义

5.4.2 翻译模式

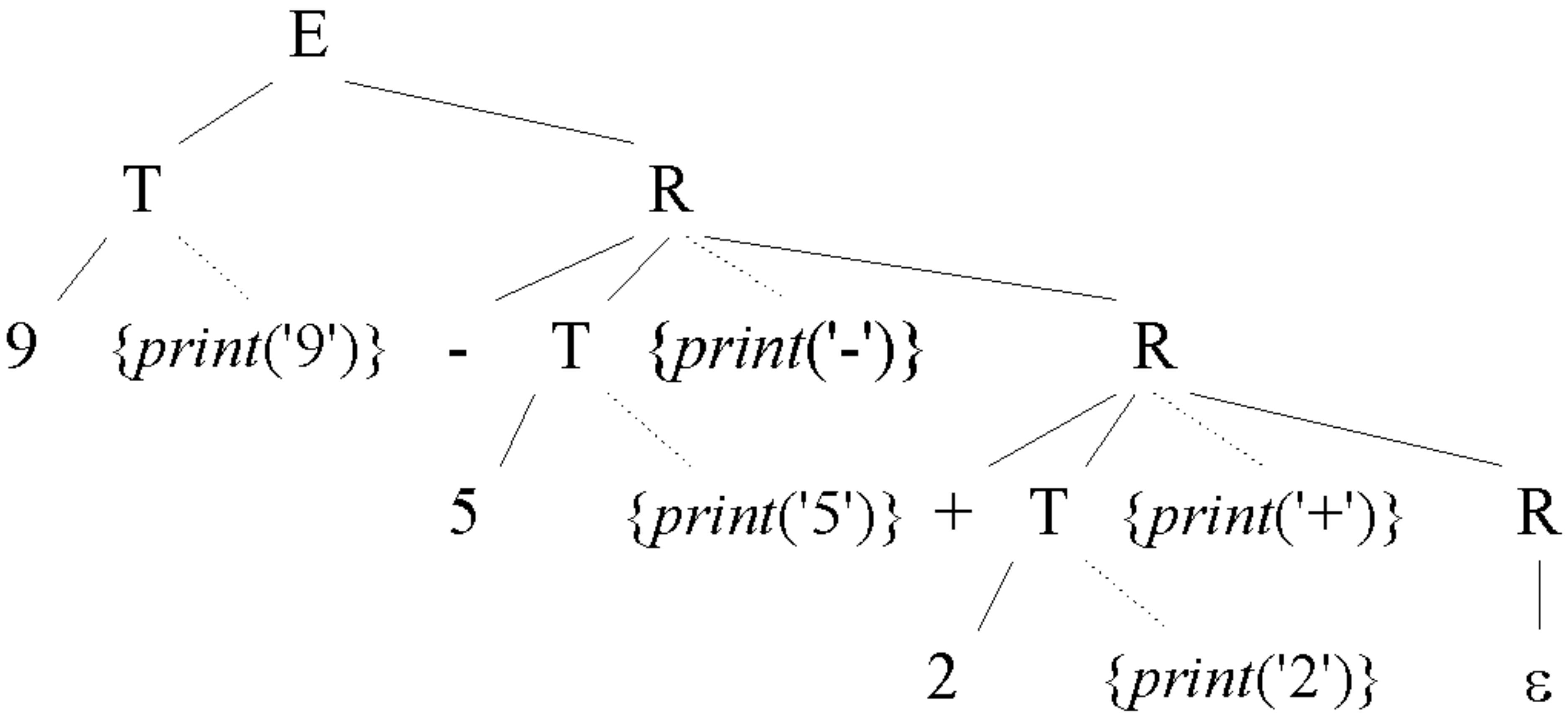
- 语义动作嵌入产生式
- 指明计算顺序
- 例5.12

$E \rightarrow T\ R$

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

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

例5.12（续）



设计翻译模式

- 以语法制导定义为基础
- 注意L-属性定义所带来的限制，确保不会违反属性计算的依赖关系
- 最简单情况：只有综合属性
- 既有综合属性，又有继承属性
  1. 右部符号的继承属性必须在符号之前的语义动作中进行计算
  2. 动作不能引用它右边符号的综合属性
  3. 左部NT 的综合属性，必须在其依赖的所有属性计算完毕后，才能计算。一般置于右部的末尾

例

$S \rightarrow A_1A_2 \ \{ A_1.in = 1; A_2.in = 2; \}$

$A \rightarrow a \ \{ \mathit{print}(A.in); \}$

- 产生式1不满足第一条要求，A_1，A_2的继承属性在符号之后进行计算。
  当利用产生式2进行归约时，A的继承属性还未得到，失败
- L-属性定义总能构造出符合3条要求的翻译模式

例5.13

- 数学格式化语言EQN
- **text**——“文本框”， **sub**——下标
- E **sub** 1 .val➔E_1.val

例5.13 语法制导定义

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**$.h \times B.ps$

例5.13 翻译模式

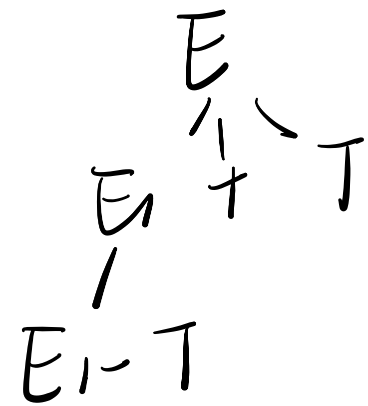
$S \rightarrow$	$\{\ B.ps = 10;\ \}$
$\quad B$	$\{\ S.ht = B.ht;\ \}$
$B \rightarrow$	$\{\ B_1.ps = B.ps;\ \}$
$\quad B_1$	$\{\ B_2.ps = B.ps;\ \}$
$\quad B_2$	$\{\ B.ht = \max(B_1.ht, B_2.ht);\ \}$
$B \rightarrow$	$\{\ B_1.ps = B.ps;\ \}$
$\quad B_1$	
$\quad$ **sub**	$\{\ B_2.ps = shrink(B.ps);\ \}$
$\quad B_2$	$\{\ B.ht = disp(B_1.ht, B_2.ht);\ \}$
$B \rightarrow$ **text**	$\{\ B.ht = \textbf{text}.h \times B.ps;\ \}$

5.5 自顶向下计算L-属性定义

- 在预测分析法过程中计算L-属性定义
- 预测分析法
  - 文法无左递归，提取了左公因子
  - 每个NT 构造一个函数，分析其所有产生式

5.5.1 消除左递归（例5.14）

$E \rightarrow E_1 + T$	左递归	$\{E.val = E_1.val + T.val\}$
$E \rightarrow E_1 - T$		$\{E.val = E_1.val - T.val\}$
$E \rightarrow T$		$\{E.val = T.val\}$
$T \rightarrow (E)$		$\{T.val = E.val\}$
$T \rightarrow$ **num**		$\{T.val =$ **num**$.val\}$



$E \rightarrow E \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

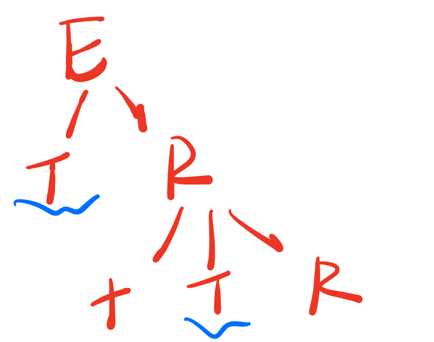
- 消除左递归

$E \rightarrow TR$	$\{E.val = ?\}$
$R \rightarrow +TR$	$\{?\}$
$R \rightarrow -TR$	$\{?\}$
$R \rightarrow \varepsilon$	$\{?\}$
$T \rightarrow (E)$	$\{T.val = E.val\}$
$T \rightarrow$ **num**	$\{T.val =$ **num**$.val\}$

$E \rightarrow TR$

$R \rightarrow \alpha R \mid \varepsilon$

$R \rightarrow +TR \mid \varepsilon$



解决方法：改写文法

- $A \rightarrow A\alpha \mid \beta$ 改写为
- $A \rightarrow \beta R$
  $R \rightarrow \alpha R \mid \varepsilon$

$\beta \mid \alpha\alpha\alpha\alpha\alpha$

左递归.

2个运算对象在2个产生式里
不好联系

例 $\alpha\alpha\alpha\varepsilon$.

例5.14（续）

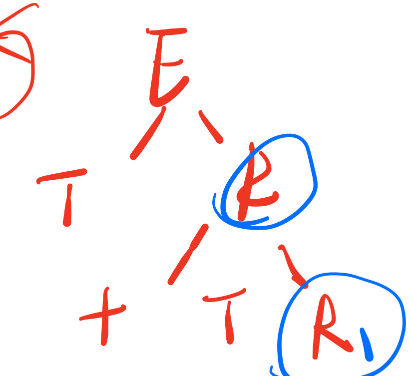
① R.i 自顶向下传递
② R.s 自底向上传递.

- 继承属性——R.i，综合属性——R.s
  将T的值向下传递，计算结果向上传递

保存右边的属性

$E \to T$	$\{R.i = T.val\}$	R	$\{E.val = R.s\}$
$R \to + T$	$\{R_1.i = R.i + T.val\}$	R_1	$\{R.s = R_1.s\}$
$R \to - T$	$\{R_1.i = R.i - T.val\}$	R_1	$\{R.s = R_1.s\}$
$R \to \varepsilon$	$\{R.s = R.i\}$		
$T \to (E)$	$\{T.val = E.val\}$		
$T \to \textbf{num}$	$\{T.val = \textbf{num}.val\}$		

看到ε停止
开始传递综合属性



计算结果

$E \to E_1 + T \quad \{E.val = E_1.val + T.val\}$

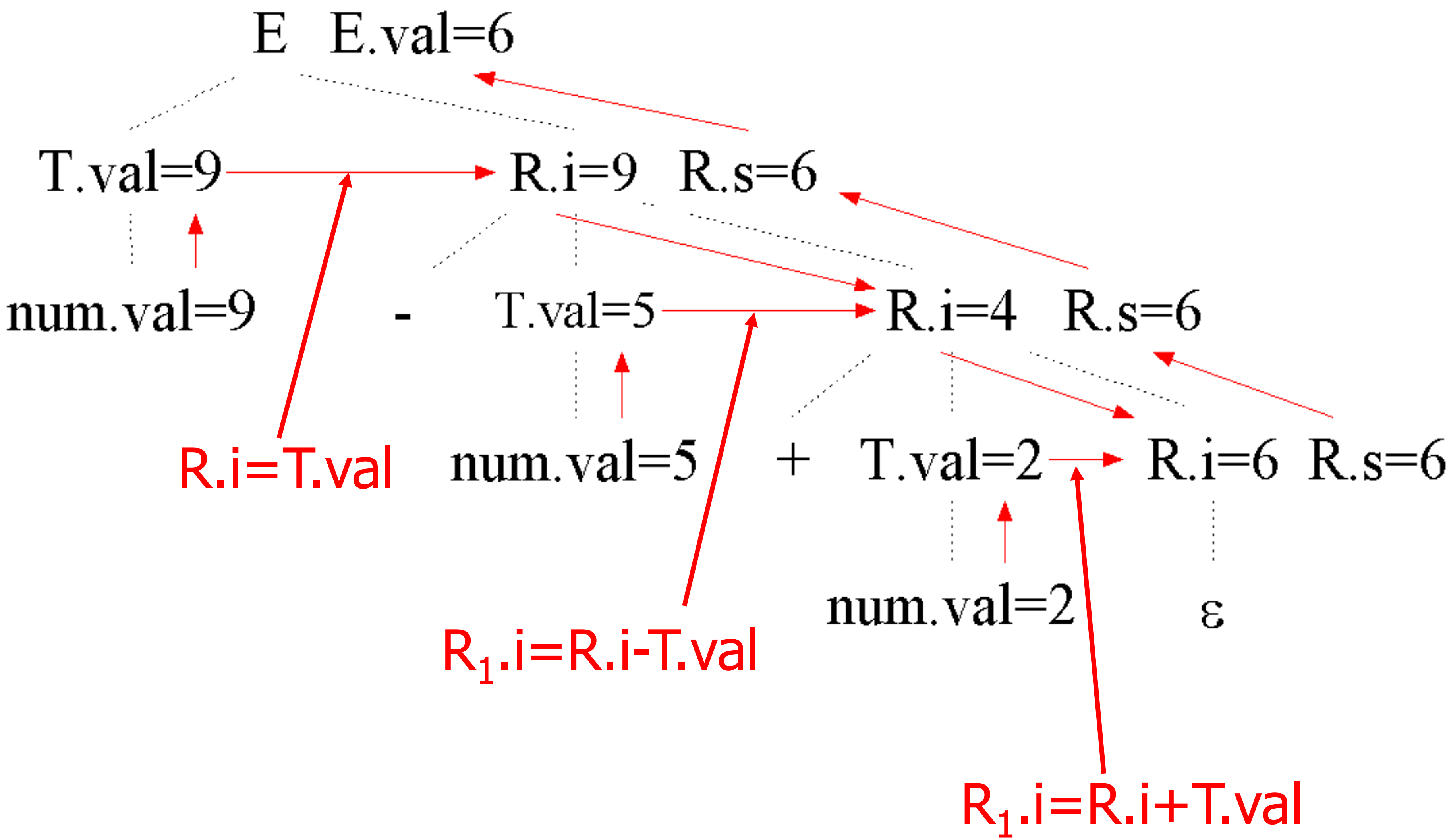
$E \to E_1 - T \quad \{E.val = E_1.val - T.val\}$

$E \to T \quad \{E.val = T.val\}$

$T \to (E) \quad \{T.val = E.val\}$

$T \to \textbf{num} \quad \{T.val = \textbf{num}.val\}$

例5.14（续）


E E.val=6
T.val=9
R.i=9 R.s=6
num.val=9
-
T.val=5
R.i=4 R.s=6
R.i=T.val
num.val=5
+
T.val=2
R.i=6 R.s=6
num.val=2
ε
R1.i=R.i-T.val
R1.i=R.i+T.val

消除左递归一般性方法

$A \to A_1\ Y$ | X $\{A.a = g(A_1.a\ ,\ Y.y)\ \}$

$A \to X\ \{A.a = f(X.x)\ \}$

A → XR1

R → YR1

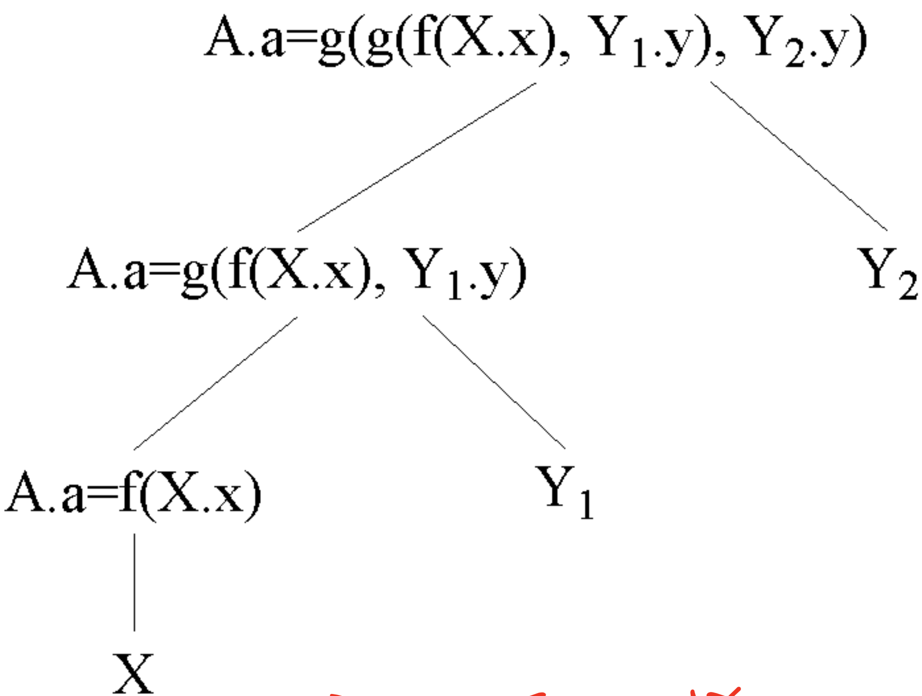
- 消除左递归→

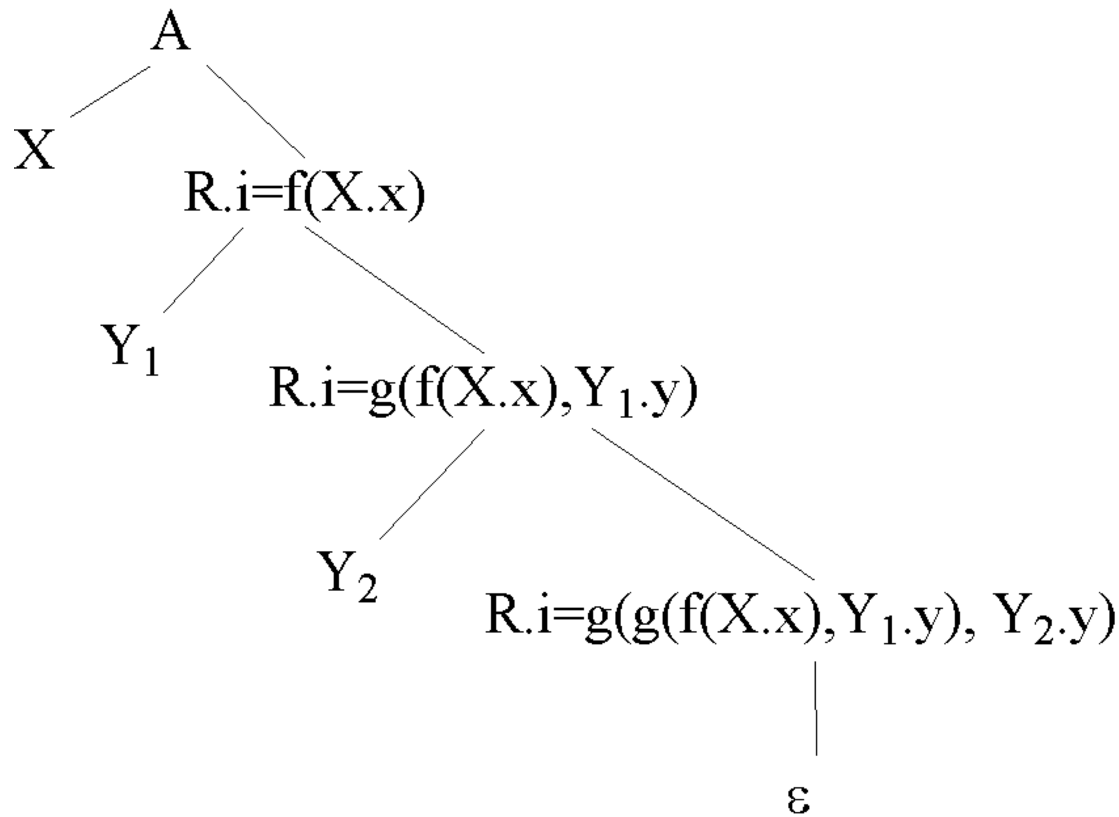
$A \to X\ \{R.i = f(X.x)\}\quad R\ \ \{A.a = R.s)\ \}$

$R \to Y\ \{R_1.i = g(R.i,\ Y.y)\}\ R_1\ \{R.s = R_1.s)\ \}$

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

左递归/非左递归比较





倾斜方向不一样

例5.15 创建语法树

$E \to E_1 + T$	$\{E.nptr = \text{mknode}(\text{"+"}, E_1.nptr, T.nptr)\}$
$E \to E_1 - T$	$\{E.nptr = \text{mknode}(\text{"-"}, E_1.nptr, T.nptr)\}$
$E \to T$	$\{E.nptr = T.nptr\}$
$T \to (E)$	$\{T.nptr = E.nptr\}$
$T \to \mathbf{id}$	$\{T.nptr = \text{mkleaf}(\mathbf{id}, \mathbf{id}.entry)\}$
$T \to \mathbf{num}$	$\{T.nptr = \text{mkleaf}(\mathbf{num}, \mathbf{num}.entry)\}$

△ 学习&理解

例5.15 消除左递归后

$E \rightarrow T \; \{ R.i = T.nptr \} \; R \; \{ E.nptr = R.s \}$

$R \rightarrow + \; T \; \{ R_1.i = \text{mknode}(\text{"+"}, R.i, T.nptr) \} \; R_1 \; \{ R.s = R_1.s \}$

$R \rightarrow - \; T \; \{ R_1.i = \text{mknode}(\text{"-"}, R.i, T.nptr) \} \; R_1 \; \{ R.s = R_1.s \}$

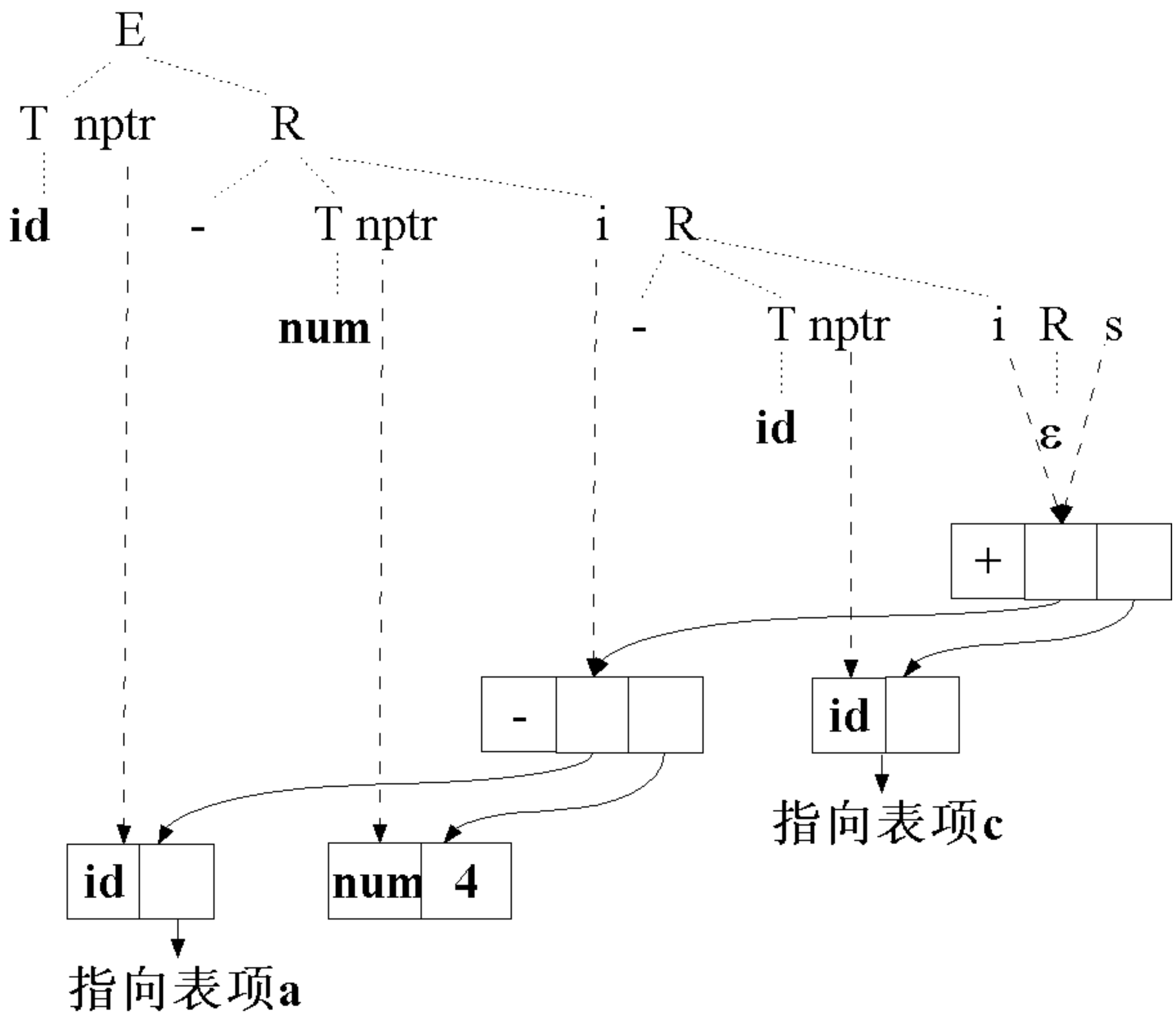
$R \rightarrow \varepsilon \quad \{R.s = R.i\}$

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

$T \rightarrow \mathbf{id} \quad \{T.nptr = \text{mkleaf}(\mathbf{id}, \mathbf{id}.entry)\}$

$T \rightarrow \mathbf{num} \quad \{T.nptr = \text{mkleaf}(\mathbf{num}, \mathbf{num}.entry)\}$

例5.15 创建语法树（续）



5.5.2 构造预测翻译器

算法5.2

输入：语法制导翻译模式，文法适用于预测分析

输出：语法制导翻译器的代码

方法

1. 对每个NT A构造一个函数，计算综合属性
   每个继承属性做为一个形式参数
   假定每个NT只有一个综合属性
   产生式中语法符号的属性——函数局部变量
2. 参照2.4节，A的代码根据当前输入符号确定使用哪个产生式

算法5.2 构造预测翻译器（续）

3. 每个产生式对应代码构造如下：由左至右依次考虑产生式右部的T、NT和语义动作
   i. 终结符X的综合属性x
      - 保存到对应的局部变量
      - 调用match匹配X
      - 输入指针前移

算法5.2 构造预测翻译器（续）

ii. NT B

- 在其后生成一条赋值语句$c=B(b_1, b_2, \ldots, b_k)$
- $b_1, b_2, \ldots, b_k$——B的继承属性对应变量
- c——B的综合属性对应变量

iii. 语义动作

- 代码片断复制到翻译器相应位置
- 对属性的引用➔对相应变量的引用

例5.16

$E \rightarrow T$	$\{ R.i = T.nptr \}$
$\quad R$	$\{ E.nptr = R.s \}$
$R \rightarrow$ **addop**	
$\quad T$	$\{ R_1.i =$ mknode(**addop**.*lex* , $R.i$, $T.nptr$)$\}$
$\quad R_1$	$\{ R.s = R_1.s \}$
$R \rightarrow \varepsilon$	$\{R.s = R.i\}$
$T \rightarrow (E)$	$\{T.nptr = E.nptr\}$
$T \rightarrow$ **id**	$\{T.nptr =$ mkleaf(**id**, **id**.*entry*)$\}$
$T \rightarrow$ **num**	$\{T.nptr =$ mkleaf(**num**, **num**.*entry*)$\}$

例5.16（续）

```
node main()
{
  return E();
}
```

例5.16（续）

$E \rightarrow T \; \{ R.i = T.nptr \}$

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

```
node TD_E()
{
   tnptr = T();
   ri = tnptr;
   rs = R(ri);
   enptr = rs;
   return enptr;
}
```

```
node TD_E()
{
   nptr = T();
   return R(nptr);
}
```

例5.16（续）

```
node T()
{if lookahead = '(' then
    { match('(');
      nptr=E(); match(')'); }
 else if lookahead = 'id'
      then { match('id');
             label = lexval;
              nptr=mkleaf(label);}
 else if lookahead = 'num'
      then { match('num');
             label = lexval;
              nptr=mkleaf(label);}
 else error; return nptr}
```

例5.16（续）

R→ **addop**

T {$R_1.i$ = mknode(**addop**.*lex* , R.*i* , T.*nptr*)}

R_1 {R.*s* = R_1.*s* }

```
node R(i:node)
{if lookahead='addop'
     then
     { match('addop');
       addoplex=lexval;
       nptr = T();
       i = mknode(addoplex,i,nptr);
       s = R(i); }
  else { s=i }
 return s
}
```

5.6 自底向上计算L-属性定义

- 自顶向下方法缺点：文法必须适用于预测分析法
- 自底向上方法
  - 适用基于LL(1)文法的L-属性定义
  - 适用很多基于LR(1)的L-属性定义
  - 基本思想：用额外的栈保存属性，归约时更新栈中属性值

5.6.1 消除翻译模式的嵌入动作

- 第一个障碍
  - 自底向上分析方法，通过移进在在栈顶形成句柄，然后归约
  - 在移进过程中根本不知道将来会形成哪个产生式，嵌入动作无法执行
- 解决方法：加入“标记”（**marker**）NT——消除嵌入语义动作

加入 marker NT

消除嵌入语义动作

加标记NT例

E→ T R

R→ + T { print(‘+’) } R | - T { print(‘-’) } R | ε

T→ **num** { print(**num**.*val*)}

- 改写为

E→ T R

R→ + T M R | - T N R | ε

T→ **num** { print(**num**.*val*)}

M→ ε { print(‘+’) }

N→ ε { print(‘-’) }

消除嵌入至内部的产生式

- 属性值的计算只会在归约时进行

形成句柄后再执行语义动作

5.6.2 分析栈中的继承属性

- 第二个障碍
  - 用C→XYZ归约，此时若需要C的继承属性，从哪里获得？
  - C的继承属性依赖于其父亲和左兄弟➔寻找C在右部的那些产生式！
  - 左边兄弟节点在已在栈中！父节点呢？

A→BC ?

C→XYZ

栈中的继承属性

- 考虑A→ XY
  - 假定X具有综合属性X.s
  - 由5.3节方法，X.s与X一起保存在栈中
  - 当进行子树Y的分析前，X.s必在栈中
  - 若Y的继承属性Y.i=X.s，则可直接使用X.s
  - “拷贝规则”

例5.17

产生式	语义动作
$D \rightarrow T\ L$	$L.\mathit{in} = T.\mathit{type}$
$T \rightarrow$ **int**	$T.\mathit{type} = \mathit{integer}$
$T \rightarrow$ **real**	$T.\mathit{type} = \mathit{real}$
$L \rightarrow$	$L_1.\mathit{in} = L.\mathit{in}$
L_1 **, id**	$\mathit{addtype}(\textbf{id}.\mathit{entry}, L.\mathit{in})$
$L \rightarrow$ **id**	$\mathit{addtype}(\textbf{id}.\mathit{entry}, L.\mathit{in})$

例5.17（续）

STACK	Input	Action
$	real a,b,c$	
$[real, ‘real’]	a,b,c$	T → real，修改*type*
$[T, ‘real’]	a,b,c$	
$[T, ‘real’] [id , ‘a’]	b,c$	L → id，需要L.in
$[T, ‘real’] [L, …]	,b,c$	
$[T, ‘real’] [L, …] ,	b,c$	
$[T, ‘real’] [L, …] , [id , ‘b’]	,c$	L → L, id，需要L.in
$[T, ‘real’] [L, …] ,	c$	
$[T, ‘real’] [L, …] , [id , ‘c’]	$	L → L, id，需要L.in

例5.17（续）

- 对关于L的产生式归约时，需用L的继承属性in进行计算，而L.in = T.type
- 此时，产生式右部位于栈顶，而T恰位于它们下面（左面）
- 因此，在归约时，T.type的位置可知，可避免属性的简单复制

例5.17（续）

产生式	代码片断
$D \to T\ L$	
$T \to$ **int**	$\text{val}[ntop] = integer$
$T \to$ **real**	$\text{val}[ntop] = real$
$L \to L_1$ **, id**	$addtype($**id**$.entry, \text{val}[top\text{-}3])$
$L \to$ **id**	$addtype($**id**$.entry, \text{val}[top\text{-}1])$

T.type→L.in

父节点的左兄弟
继承属性

$L \to L_1 , id$

L_2 , id

id

5.6.3 模拟继承属性的计算

- 例5.18：不能预测属性值栈中位置的文法

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

- C通过拷贝规则继承了A的综合属性
- B是否在栈中？——不知道！
- A.s(C.i)在top-1？ top-2？
- 如何解决？——利用marker改写文法

例5.18（续）

通过改写 使 C 的继承属性直接依赖于紧邻的兄弟节点

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

- 也可用于非拷贝规则（更复杂的引用）

$S \rightarrow aAC$	$C.i = f(A.s)$

改写为

$S \rightarrow aANC$	$N.i = A.s$; $C.i = N.s$
$N \rightarrow \varepsilon$	$N.s = f(N.i)$

例5.19

产生式	语义规则
$S \to LB$	$B.ps = L.s$ $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$
$B \to B_1MB_2$	$B_1.ps = B.ps$ $M.i = B.ps$ $B_2.ps = M.s$ $B.ht = max(B_1.ht, B_2.ht);$
$B \to B_1$ **sub** $N\ B_2$	$B_1.ps = B.ps;$ $N.i = B.ps;$ $B_2.ps = N.s;$ $B.ht = disp(B_1.ht, B_2.ht);$
$B \to$ **text**	$B.ht =$ **text**$.h \times B.ps$
$M \to \varepsilon$	$M.s = M.i$
$N \to \varepsilon$	$N.s = shrink(N.i)$

s、ht等综合属性保存在栈中
继承属性不保存

例5.19（续）

产生式	代码片断
$S \rightarrow LB$	val[ntop] = val[top];
$L \rightarrow \varepsilon$	val[ntop] = 10
$B \rightarrow B_1MB_2$	val[ntop] = max(val[top-2], val[top])
$B \rightarrow B_1$ **sub** N B_2	val[ntop] = disp(val[top-3], val[top])
$B \rightarrow$ **text**	val[ntop] = val[top]*val[top-1]
$M \rightarrow \varepsilon$	val[ntop] = val[top-1]
$N \rightarrow \varepsilon$	val[ntop] = shrink(val[top-2])

M.s=M.i(B.ps, L.s)
为什么从top-1获得？

B.ht=text.h*B.ps
B.ps为什么总可以从top-1获得？

算法5.3：自底向上翻译

输入：LL(1)文法及L-属性定义

输出：一个分析器，可在分析栈中计算属性值

方法：

分析X_j之前，$X_j.i$已经在栈中

1. 假定每个NT A有一个继承属性A.i，每个语法符号X有一个综合属性X.s
2. 若X为T，X.s为词法值，保存在属性栈val中
3. 对每个产生式$A\rightarrow X_1\ldots X_n$，引入n个标记NT $M_1, \ldots, M_n$ ➔ 产生式变为$A\rightarrow M_1X_1\ldots M_nX_n$
4. 综合属性$X_j.s$保存在val栈，与X_j相关联
5. 继承属性$X_j.i$（M_j的综合属性s）也保存在val中，但与M_j相关联

算法5.3：（续）

A之前的标记NT M（A在右部的产生式），M.s(A.i)已在栈中

6. 分析过程中，A.i在栈中始终紧挨在M_1之下。
7. 考虑分析过程中的两种情况：

一、归约为M_j

- 可知它所属产生式➔可知计算$X_j.i$（$M_j.s$）所需属性值位置
- A.i－val[top-2j+2]，$X_1.i(M_1.s)$－val[top-2j+3]，$X_1.s$－val[top-2j+4]，…

二、归约为其他NT

- 仅需计算A.s🡸A.i和X_j的属性的位置是可知

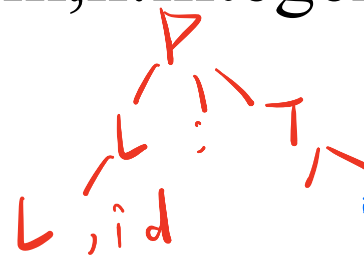
注意两点：

1. 若X_j无继承属性，则不需要M_j
2. 若$X_1.i$存在，但$X_1.i$=A.i，则不需要M_1

5.6.4 用综合属性代替继承属性

- 改写语法，如Pascal变量定义 m,n:integer;

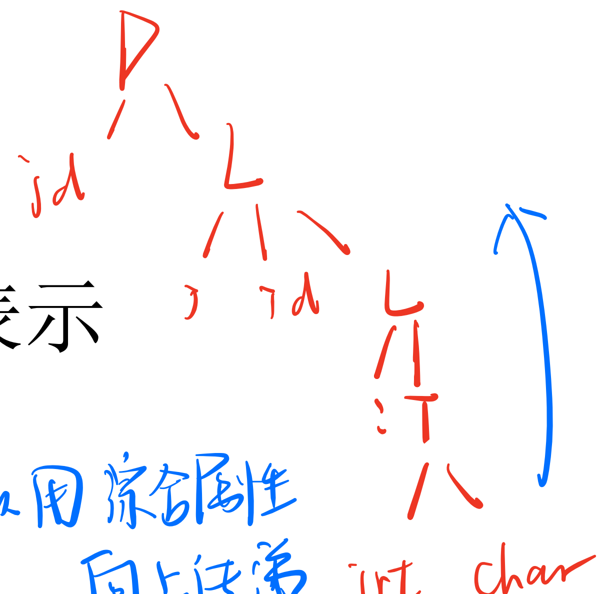
D → L : T
T → **integer** | **char**
L → L , **id** | **id**



类型与变量
不在一棵子树
要用继承属性

- 变量由L产生，但类型不在L的子树中——需用继承属性。改写为

D → **id** L
L → , **id** L | : T
T → **integer** | **char**



- 类型即可用L的一个综合属性表示

可以用综合属性
向上传递

5.6.5 一个难实现的语法制导定义

- 自底向上翻译适用于LL(1)文法
- 可扩展到某些LR(1)文法，但不是全部

产生式	代码片断
$S \rightarrow L$	$L.count = 0$
$L \rightarrow L_1\ 1$	$L_1.count = L.count + 1$
$L \rightarrow \varepsilon$	$print(L.count)$

- $L \rightarrow \varepsilon$中的L继承了由S产生的1的个数，但$L \rightarrow \varepsilon$第一个归约，无法得知count值。

在Yacc中计算属性

- $A \rightarrow X \quad Y \quad \{ A.a = f(X.x, Y.y); \}$
  $\$\$ \quad \$1 \quad \$2 \quad \{ \$\$ = f(\$1, \$2); \}$
  栈：…, [X, $1], [Y, $2]→…, [A, $$]
- YYSTYPE：属性类型
- 继承属性如何获得？
- …, $-2, $-1, $0——
  栈中XYZ之下符号（A的兄弟）的属性！
- 小心使用

在Yacc中计算属性（续）

产生式	语义动作
$D \rightarrow T\ L$	$L.\mathit{in} = T.\mathit{type}$
$T \rightarrow$ **int**	$T.\mathit{type} = \mathit{integer}$
$T \rightarrow$ **real**	$T.\mathit{type} = \mathit{real}$
$L \rightarrow$	$L_1.\mathit{in} = L.\mathit{in}$
L_1 **, id**	$\mathit{addtype}($**id**$.\mathit{entry}, L.\mathit{in})$
$L \rightarrow$ **id**	$\mathit{addtype}($**id**$.\mathit{entry}, L.\mathit{in})$

T : int { $$ = $1; }

L : id { addtype($1, $0); }

T.type

T : int { curr_type = $1; }

L : id { addtype($1, curr_type);

推荐方法

不考

5.7 递归方法计算属性

- 遍历语法分析树时计算属性，适用那些不能在语法分析同时计算属性的情况
- NT←→函数
- 依次访问NT 对应节点的孩子节点

5.7.1 由左至右的访问顺序

- 每个NT 对应相似的递归函数
- 在对应节点，由产生式确定其孩子节点
- 对应节点、继承属性——参数
- 综合属性——返回值

例5.20

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

例5.20 B对应函数

```
B(n, ps)
{
    var ps1, ps2, ht1, ht2;
    switch (节点n所用产生式)
    {
        case "B → B1B2":
                ps1 = ps;
                ht1 = B(child(n, 1), ps1);
                ps2 = ps;
                ht2 = B(child(n, 2), ps2);
                return max(ht1, ht2);
```

例5.20 B对应函数

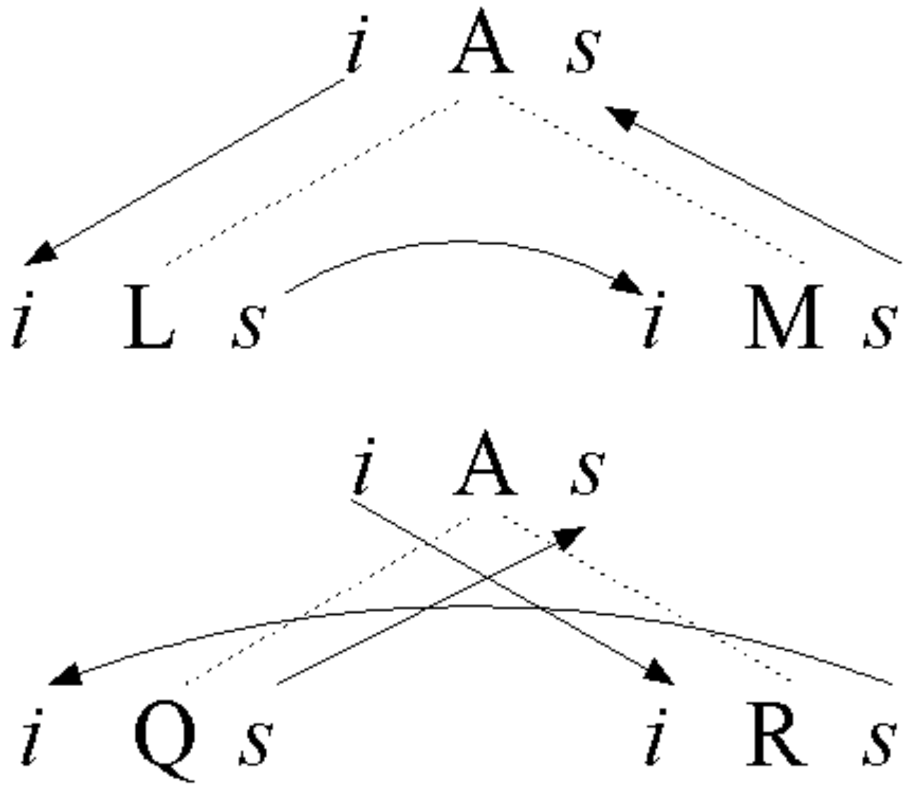
```
case "B1 sub B2":
        ps1 = ps;
        ht1 = B(child(n, 1), ps1);
        ps2 = shrink(ps);
        ht2 = B(child(n, 3), ps2);
        return disp(ht1, ht2);
case "B → text":
        return ps*text.h;
default:
        error();
    }
}
```

5.7.2 其他访问顺序（例5.21）

产生式	语义规则
$A \to LM$	$L.i = l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \to QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

- 第一个产生式需要由左至右顺序访问
- 第二个需要由右至左

例5.21（续）



例5.21（续）

```
A(n, ai)
{
    switch (节点n所用产生式)
    {
        case "A → LM":
                li = l(ai);
                ls = L(child(n, 1), li);
                mi = m(ls);
                ms = M(child(n, 2), mi);
                return f(ms);
```

例5.21（续）

```
case "A → QR":
        ri = r(ai);
        rs = R(child(n, 2), ri);
        qi = q(rs);
        qs = Q(child(n, 1), qi);
        return f(qs);
default:
        error();
    }
}
```

5.8 编译时内存空间分配

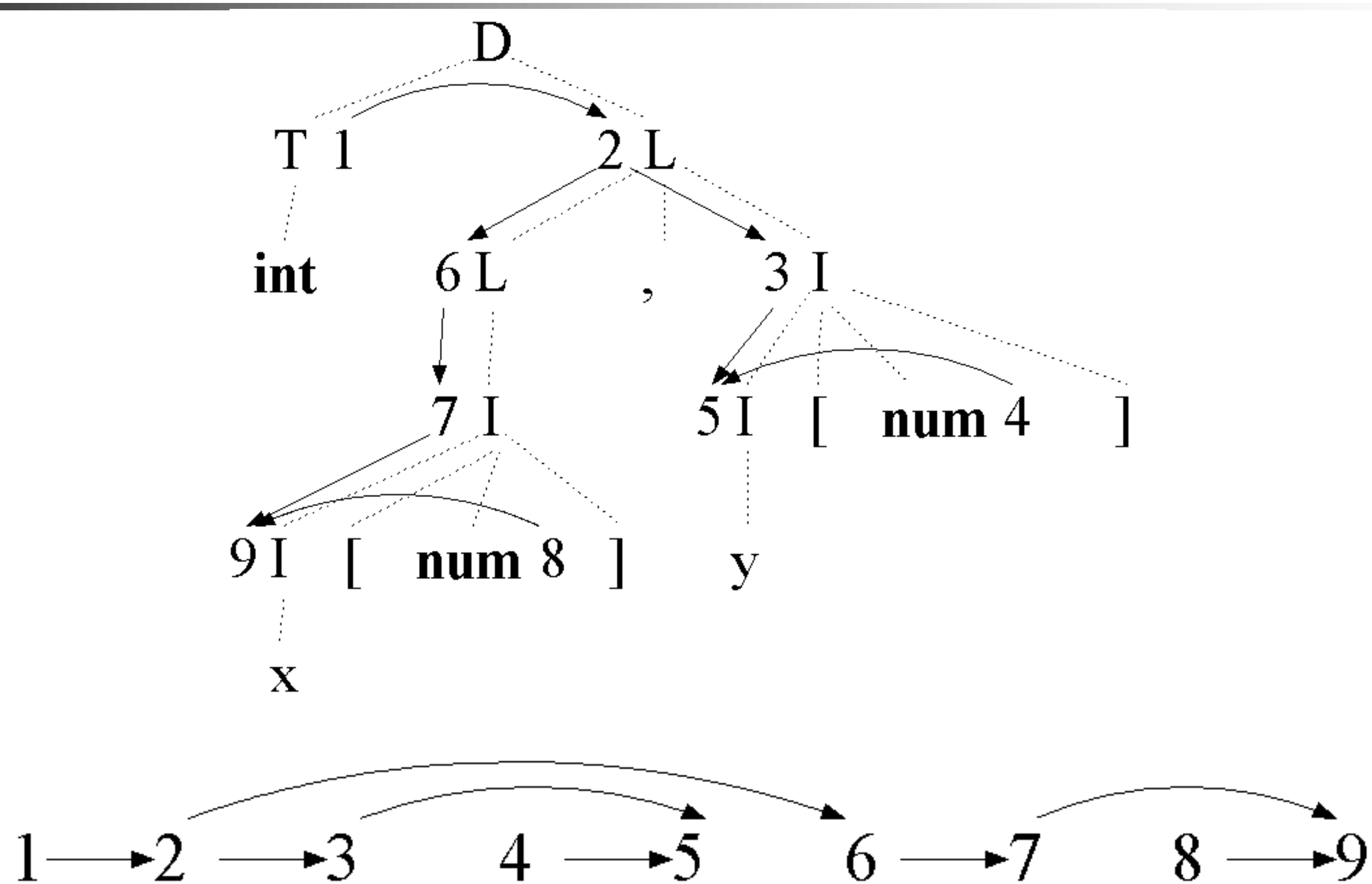
- 依赖图→属性计算顺序→内存分配
- 深度优先顺序→属性生存期（它第一次被计算——依赖它的其他属性都计算完毕）
- 只在属性生存期内为其分配内存

例5.22

产生式	代码片断
$D \to T\ L$	$L.in = T.type$
$T \to$ **int**	$T.type = integer$
$T \to$ **real**	$T.type = real$
$L \to L_1\ ,\ I$	$L_1.in = L.in$ $I.in = L.in$
$L \to I$	$I.in = L.in$
$I \to I_1$ [**num**]	$I_1.in = array($**num**$.\mathrm{val}, I.in)$
$I \to$ **id**	$addtype($**id**$.\mathrm{entry}, I.in)$

int x[3], y[5];

例5.22（续）

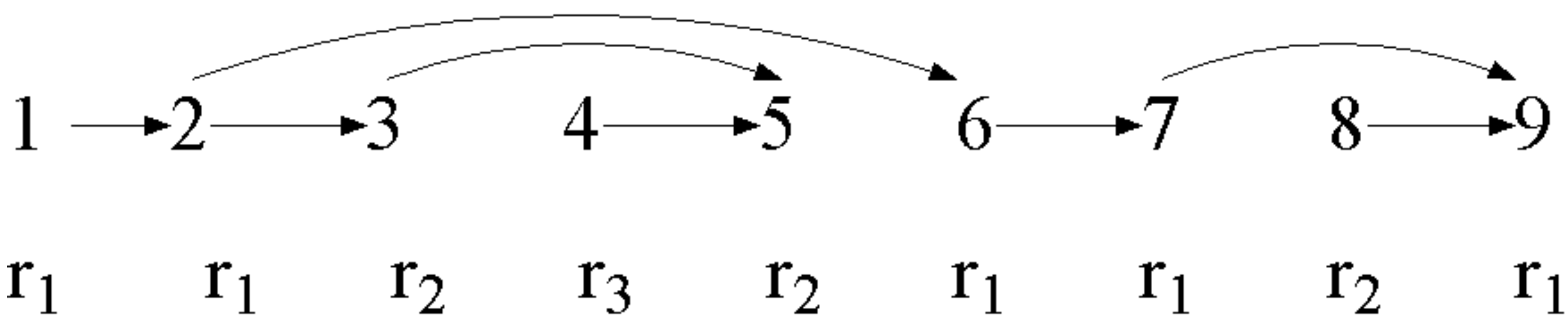


5.8.1 在编译时分配空间

- 内存分配算法
- 寄存器池：$r_1, r_2, \ldots$

```
for m1, m2, …mN中每个节点m {
    for 计算m时生存期结束的每个节点n
            标记n的寄存器;
    if (寄存器r被标记) {
            去除r的标记;
            将r分配给m，计算m;
            将标记的寄存器放回寄存器池;
    } else  从寄存器池为m分配一个寄存器，计算m;
    if (m的生存期已经结束)
            将m的寄存器放回寄存器池;
}
```

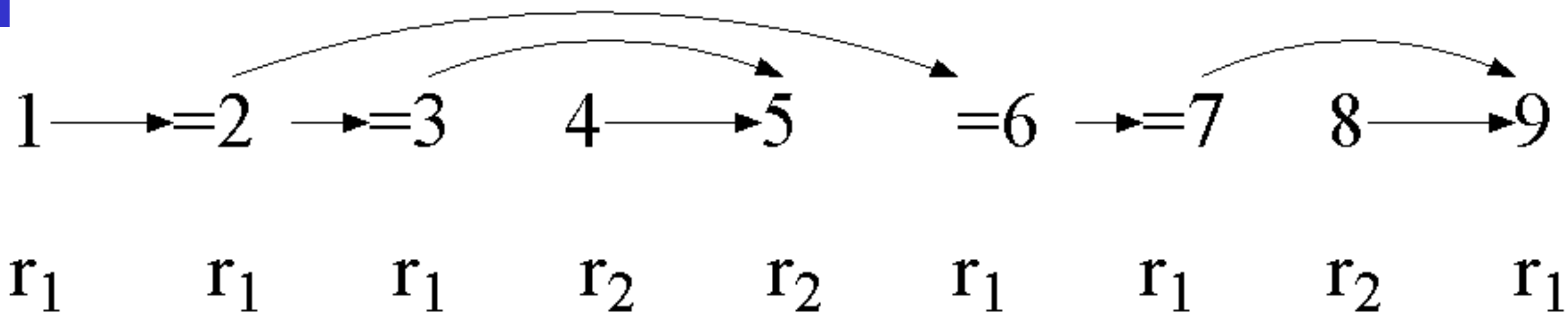
算法应用实例



5.8.2 避免属性拷贝

- b=c：b的值在c的寄存器中，不再分配
- 在节点m
  - 首先检查是否是拷贝规则
  - 若是，m值已经在某个寄存器中，m加入该寄存器的等价类
  - 只有当寄存器等价类中所有属性的生存期均结束，才可释放该寄存器

例5.24：例5.23采用新方法



$r_1 = \text{integer};$

$r_2 = 5;$

$r_2 = \text{array}(r_2, r_1);$

$\text{addtype}(y, r_2);$

$r_2 = 3;$

$r_2 = \text{array}(r_2, r_1);$

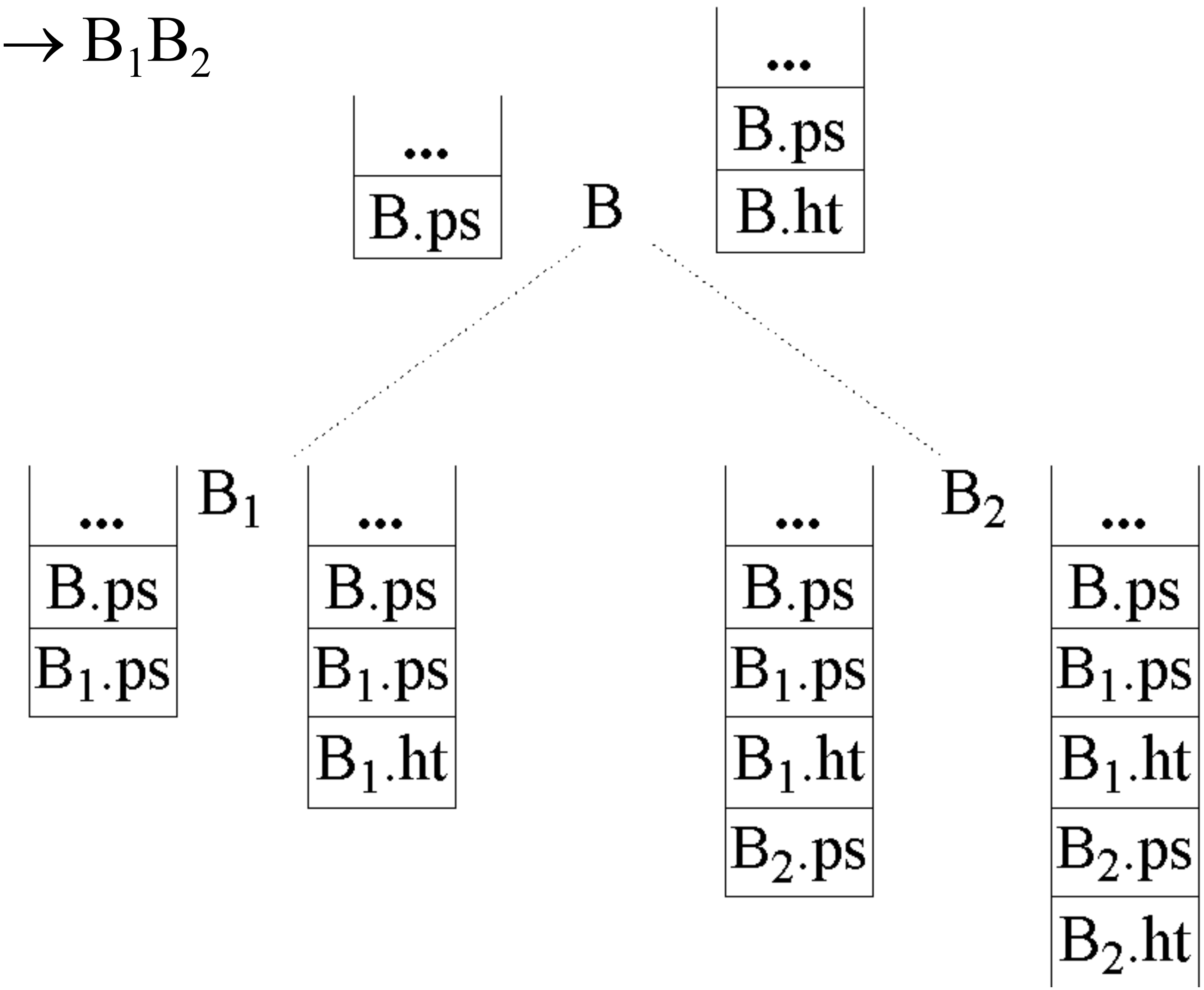
$\text{addtype}(x, r_2);$

5.9 在构造编译器时分配内存

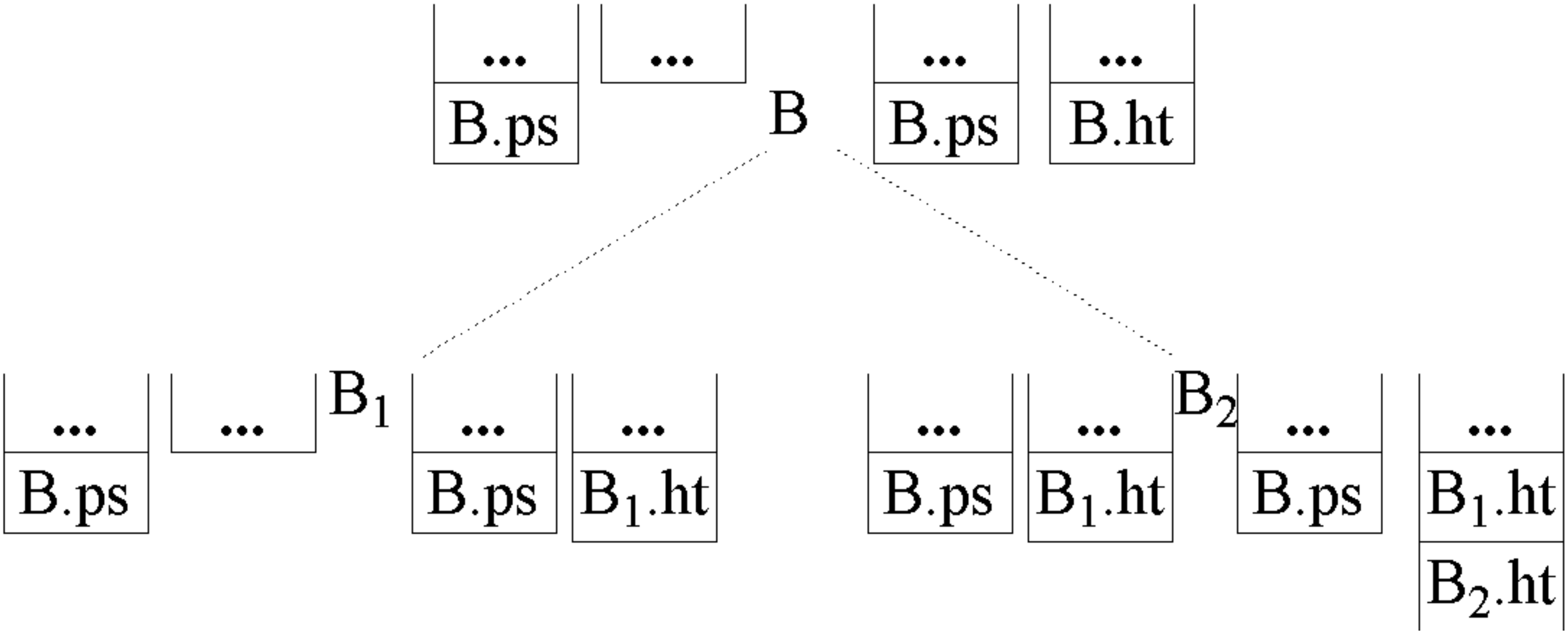
- 多栈避免属性拷贝
- 5.9.1 预测生存期
  - 对特定遍历顺序，可预测属性生存期
  - A→BC，深度优先遍历：子树B－子树C－A，返回A后，B、C的生存期即可结束
  - c的生存期包含在b内，则栈中c在b之上
  - A的继承属性入栈——计算B的继承属性并入栈——访问子树B，返回B的综合属性，入栈——对C同样进行上两个步骤——I(A), I(B), S(B), I(C), S(C)——计算A的综合属性，B、C生命期结束——I(A), S(A)

例5.25

$B \rightarrow B_1 B_2$



例5.26 多栈避免拷贝



例5.26（续）

$S \to$	{ push(10, ps); }
B	
$B \to B_1$	
B_2	{ h2 = top(ht); pop(ht); h1 = top(ht); pop(ht); push(max(h1, h2), ht); }
$B \to B_1$	
sub	{ push(shrink(top(ps)), ps); }
B_2	{ h2 = top(ht); pop(ht); h1 = top(ht); pop(ht); push(max(h1, h2), ht); }
$B \to$ **text**	{ push(**text**.h*top(ps), ht); }

例5.27 中间代码生成

- E **and** F，短路求值
- E.true(false)：E为真（假）时跳转目标

产生式	语义规则
$E \rightarrow E_1$ **and** E_2	$E_1.\mathit{true} = \mathit{newlabel}$ $E_1.\mathit{false} = E.\mathit{false}$ $E_2.\mathit{true} = E.\mathit{true}$ $E_2.\mathit{false} = E.\mathit{false}$ $E.\mathit{code} = E_1.\mathit{code} \mathbin{\|}$ $\mathit{gen}(\text{‘}\textbf{label}\text{’}E_1.\mathit{true}) \mathbin{\|} E_2.\mathit{code}$
$E \rightarrow$ **id**	$E.\mathit{code} = \mathit{gen}(\text{‘}\textbf{if}\text{’}\ \textbf{id}.\mathit{place}$ $\text{‘}\textbf{goto}\text{’}\ E.\mathit{true}) \mathbin{\|} \mathit{gen}(\text{‘}\textbf{goto}\text{’}\ E.\mathit{false})$

例5.27（续）翻译模式

产生式	语义动作
$E \rightarrow$	$\{ E_1.\mathit{true} = \mathit{newlabel};$ $E_1.\mathit{false} = E.\mathit{false}; \}$
E_1	
and	$\{ \mathit{emit}($‘**label**’ $E_1.\mathit{true});$ $E_2.\mathit{true} = E.\mathit{true};$ $E_2.\mathit{false} = E.\mathit{false}; \}$
E_2	
$E \rightarrow$ **id**	$\{ \mathit{emit}($‘**if**’ **id**$.\mathit{place}$ ‘**goto**’ $E.\mathit{true});$ $\mathit{emit}($‘**goto**’ $E.\mathit{false}); \}$

例5.27（续）栈实现

产生式	语义动作
$E \rightarrow$	{ *push*(*newlabel*, *true*); }
E_1	
and	{ *emit*(‘**label**’ *top*(*true*)); *pop*(*true*); }
E_2	
$E \rightarrow$ **id**	{ *emit*(‘**if**’ **id**.*place* ‘**goto**’ top(*true*)); *emit*(‘**goto**’ *top*(*false*)); }

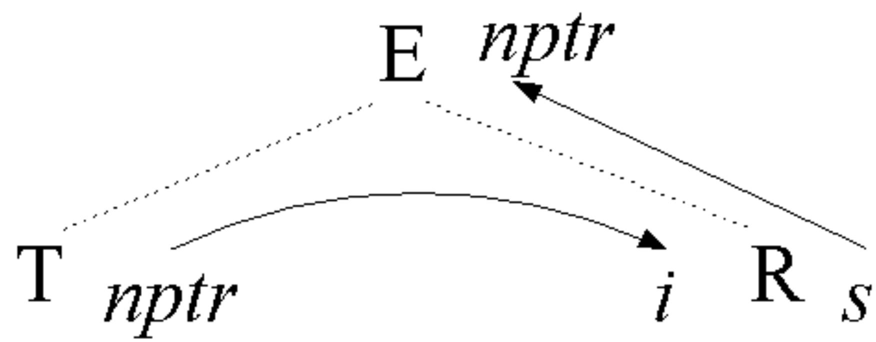
5.9.2 非重叠生存期（例5.28）

- 每个push紧接一个pop→只需一个寄存器

产生式	语义规则
$E \to T\ R$	$R.i = T.nptr$
	$E.nptr = R.s$
$R \to$ **addop** $T\ R_1$	$R_1.i = mknode(\textbf{addop}.lexeme, R.i, T.nptr)$
	$R.s = R_1.s$
$R \to \varepsilon$	$R.s = R.i$
$T \to$ **num**	$T.nptr = mkleaf(\textbf{num}, \textbf{num}.val)$

例5.28（续）

- 考虑扩展下面的依赖图
  - R→ε，R.*i*→R.*s*拷贝，可用一个寄存器
  - R→**addop** T R_1，计算R_1.*i*时R.*i*生存期结束，可用同一寄存器，由归纳假设，子树R如何扩展，R_1总可与最初的R共用同一寄存器
  - 而R.*s*为R_1.*s*的拷贝，可共用一个寄存器



例5.28（续）

$E \rightarrow T \ \{ r = T.\mathit{nptr} \}$

$\quad R \ \{ E.\mathit{nptr} = r \}$

$R \rightarrow$ **addop**

$\quad T \ \{ r = \text{mknode}(\textbf{addop}.\mathit{lex}, r, T.\mathit{nptr}) \}$

$\quad R_1$

$R \rightarrow \varepsilon$

$T \rightarrow$ **num** $\quad \{ T.\mathit{nptr} = \text{mkleaf}(\textbf{num}, \textbf{num}.\mathit{entry}) \}$

例5.28（续）

```
struct syntax_tree_node *r;
E()
{
    r = T(); R(); return r;
}
R()
{
    char addoplexeme;
    if (lookahead = addop) {
            addoplexeme = lexval;   match(addop);
            r = mknode(addoplexeme, r, T());
            R();
    }
}
```

5.10 语法制导定义分析

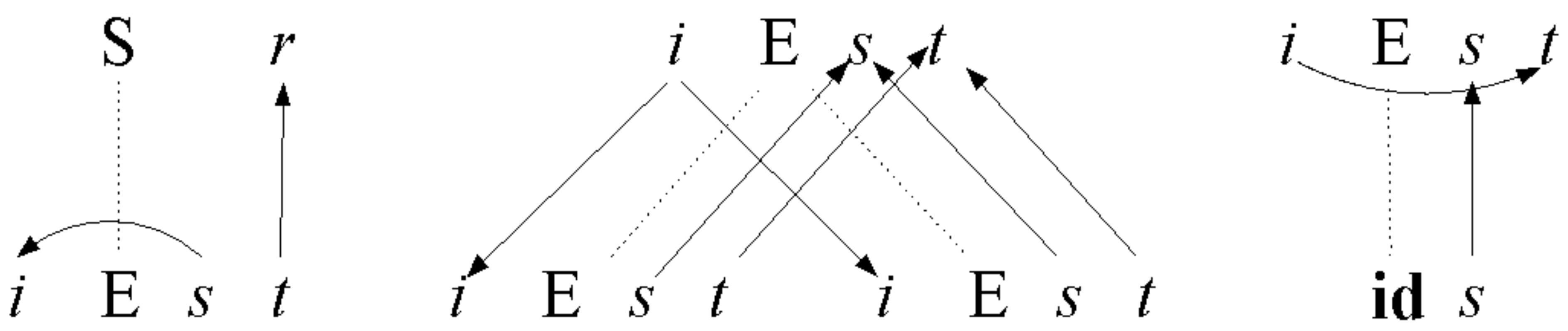
- 5.7节：多个递归函数协同计算属性
- 一次深度优先无法进行完整翻译——每个综合属性由一个单独函数计算
- 例5.29
  - 一个“重载”标识符可以有一组可能的类型
  - 表达式有一组可能类型
  - 通过上下文确定每个子表达式的类型
  - 自底向上计算可能类型集合，自顶向下确定一个最终类型

例5.29（续）

产生式	语义规则
$S \rightarrow E$	$E.i = g(E.s)$
	$S.r = E.t$
$E \rightarrow E_1 E_2$	$E.s = fs(E_1.s, E_2.s)$
	$E_1.i = fi1(E.i)$
	$E_2.i = fi2(E.i)$
	$E.t = ft(E_1.t, E_2.t)$
$E \rightarrow \mathbf{id}$	$E.s = \mathbf{id}.s$
	$E.t = h(E.i)$

- s——可能类型集合
  t——根据上下文确定的最终类型

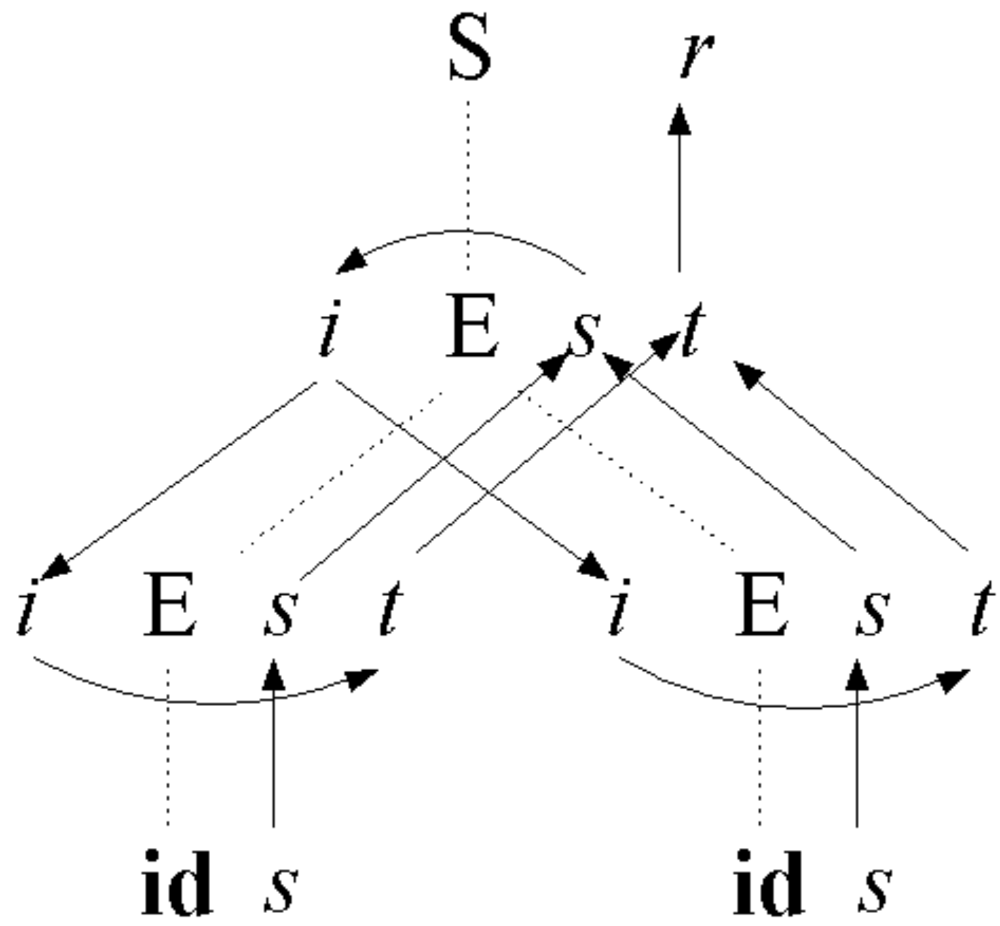
例5.29（续）



5.10.1 递归计算属性

- 语法树的依赖图——小依赖图（对应产生式的语义规则）组成
- 产生式p的依赖图D_p仅依赖p的语义规则
- “局部依赖关系”
- 多趟扫描
- 递归函数（计算综合属性）取继承属性作为参数
- A.a依赖A.b——A.b作为A.a函数的参数

例5.30



例5.30（续）

```
Es(n)
{
    switch (节点n使用的产生式) {
        case 'E → E1 E2':
            s1 = Es(child(n, 1));
            s2 = Es(child(n, 2));
            return fs(s1, s2);
        case 'E → id':
            return id.s;
        default:
            error();
    }
}
```

例5.30（续）

```
Et(n, i)
{
    switch (节点n使用的产生式) {
        case 'E → E1 E2':
            i1 = fi1(i);  t1 = Et(child(n, 1), i1);
            i2 = fi2(i); t2 = Es(child(n, 2), i2);
            return ft(t1, t2);
        case 'E → id':
            return h(i);
        default:
            error();
    }
}
```

例5.30（续）

```
Sr(n)
{
    s = Es(child(n, 1));
    i = g(s);
    t = Et(child(n, 1), i);
}
```

5.10.2 强无环语法制导定义

- 上述方法可用于强无环语法制导定义（strongly noncircular）
  - 不同节点相同NT 属性的计算可按照相同（局部）顺序
  - 根据此顺序确定选择哪些继承属性作为综合属性计算函数的参数

定义

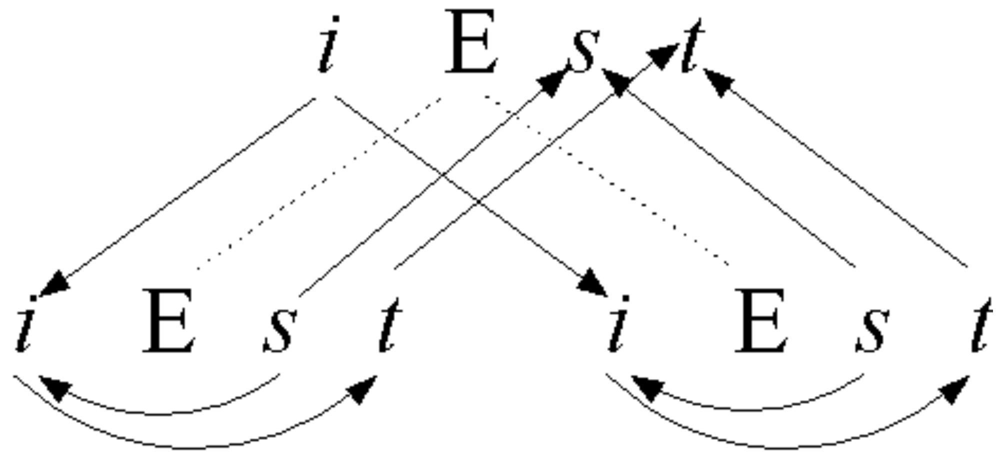
- 节点n，对应的NT A
  - 依赖图中路径，n的一个属性→其他节点属性→n的另一个属性
  - 路径位于A之下？继承属性→综合属性
- 令产生式p右部NT 为$A_1, A_2, \ldots, A_n$
  RA_j为A_j的属性的局部顺序
- $D_p[RA_1, RA_2, \ldots, RA_n]$：按如下顺序扩展$D_p$得到的图
  - 若顺序RA_j中$A_j.b$位于$A_j.c$之前，添加$A_j.b$到$A_j.c$的一条边

定义（续）

- 一个语法制导定义如果满足如下条件，则称之为“强无环的”
  - 每个NT A，存在其属性的局部顺序RA使得：对每个形如$A \rightarrow A_1, A_2, \ldots, A_n$的产生式p，满足

1. $D_p[RA_1, RA_2, \ldots, RA_n]$为无环图
2. 若$D_p[RA_1, RA_2, \ldots, RA_n]$存在A.b到A.c的边，则顺序RA中A.b在A.c之前

例5.31



- $E \rightarrow E_1\ E_2$
- 设定RE：s→i→t， RE_1、RE_2与RE相同
- $D_p[RE_1, RE_2]$如上图所示
- 唯一路径i→t，与RE不矛盾

5.10.3 检测回路（例5.32）

产生式	语义规则
$S \rightarrow A$	$A.i = c$
$A \rightarrow 1$	$A.s = f(A.i)$
$A \rightarrow 2$	$A.s = \mathrm{d}$

- 路径与产生式有关：$A \rightarrow 1$，s依赖i，否则不依赖
- 为获得完整依赖关系，需保存所有可能局部顺序集合

回路检测算法

- 局部顺序→有向无环图，检测DAG
  - 产生式p： $A \rightarrow X_1X_2\ldots X_n$，依赖图$D_p$
  - D_j为X_j的DAG
  - 将D_j中边b→a暂时加入依赖图D_p
  - 若结果依赖图存在回路，则语法制导定义是有回路的
  - 否则，图中路径形成产生式左部NT 属性的新的DAG，将它加入$\mathcal{F}(A)$

算法描述

for 每个语法符号X

$\mathcal{F}$(X)仅包含一个图：节点为X的属性，无任何边

repeat

change = **false**

for 产生式p： $A \rightarrow X_1X_2\ldots X_n$ {

for dag $G_1 \in \mathcal{F}(X_1), \ldots\ G_k \in \mathcal{F}(X_k)$ {

$D = D_p$;

for G_j中边b→c

在D中添加b→c的边;

if D包含回路

失败，语法制导定义包含回路

算法描述（续）

```
        else {
                G=包含A的属性，无边的图;
                for A的每对属性b、c
                        if D中包含边b→c
                                添加边b→c到G;
                if G不在𝓕(A)中 {
                        将G加入𝓕(A);
                        change = true;
                }
        }
    }
  }
} until change = false;
```

分析

- 运行时间与$\mathcal{F}$(A)大小成指数关系
- 改进
  - dag集$\mathcal{F}$(A)→单一dag F(X)——$\mathcal{F}$(A)图的并集
  - 最坏情况估计
  - 充分但不必要：F(X)无环→语法制导定义无环；F(X)有回路$\nrightarrow$语法制导定义有回路