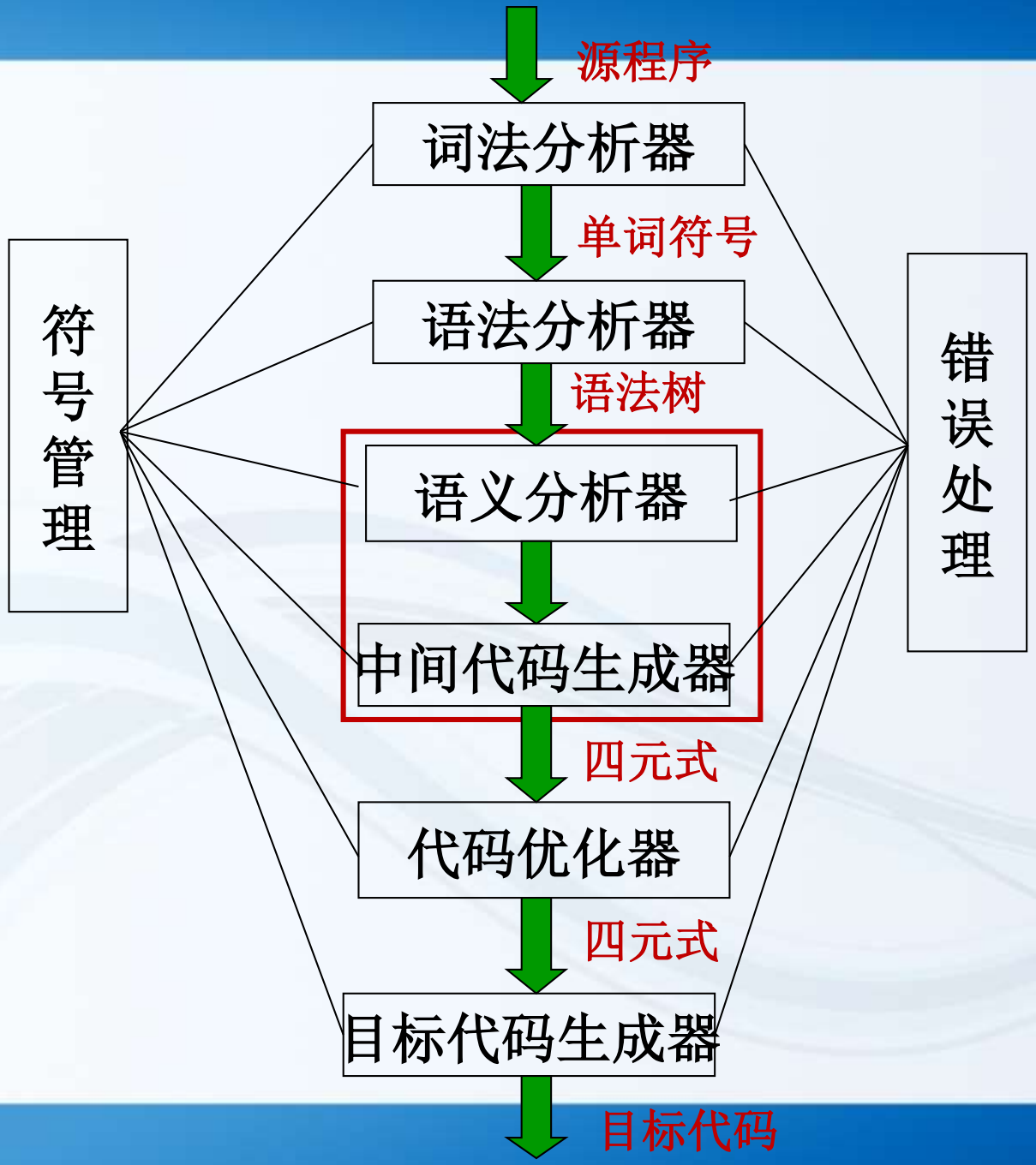
语法制导的翻译

编译程序总框


源程序
词法分析器
单词符号
语法分析器
语法树
语义分析器
中间代码生成器
四元式
代码优化器
四元式
目标代码生成器
目标代码
符号管理
错误处理

- 语法表述的是语言的形式，或者说是语言的样子和结构
- 程序设计语言中更重要的一个方面，是附着在语言结构上的语义
- 语义揭示了程序本身的涵义、施加于语言结构上的限制或者要执行的动作
- 语法正确的句子，它的语义可能存在问题
- “老鼠吃猫”、“草吃羊” 问题

语义分析的任务：

① 检查语言结构的语义是否正确

②执行所规定的语义动作

如：表达式的求值、符号表的填写、中间代码的生成

语法制导翻译的基本思想：

将语言结构的语义以属性(attribute)的形式赋予代表此结构的文法符号，如：

对于文法：

$E \rightarrow E_1+T$

$E \rightarrow T$

$T \rightarrow F$

$F \rightarrow digit$

digit·lexval为digit的属性；

F.val、T.val、E.val为文法符号F、T、E对应的属性值

而属性的计算以语义规则(semantic rules)的形式赋予由文法符号组成的产生式；

$F \rightarrow digit$	$F \cdot val = digit \cdot lexval$
$T \rightarrow F$	$T \cdot val = F \cdot val$
$E \rightarrow T$	$E \cdot val = T \cdot val$
$E \rightarrow E_1+T$	$E \cdot val = E_1 \cdot val+T \cdot val$

在语法分析推导或归约的每一步骤中，通过语义规则实现对属性的计算，以达到对语义的处理

即当归约（或推导）到某个产生式时，除了按照产生式进行相应的代换之外(语法分析)，还要按照产生式所对应的语义规则执行相应的语义动作，如计算表达式、查填符号表、产生中间代码(语义分析)

语法分析—建立语法分析树

语义分析---遍历语法分析树

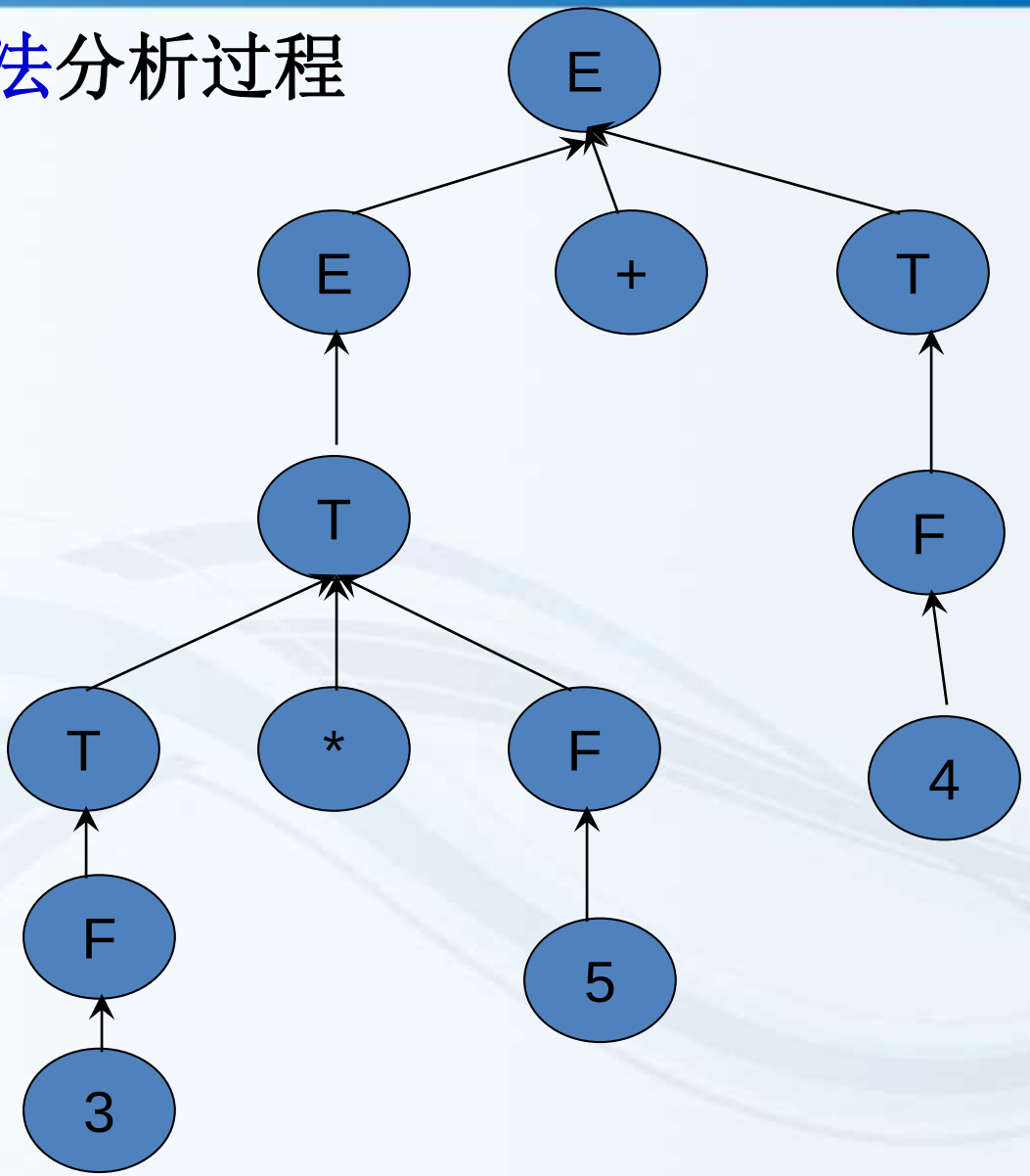
语法制导翻译---建立与遍历同时完成

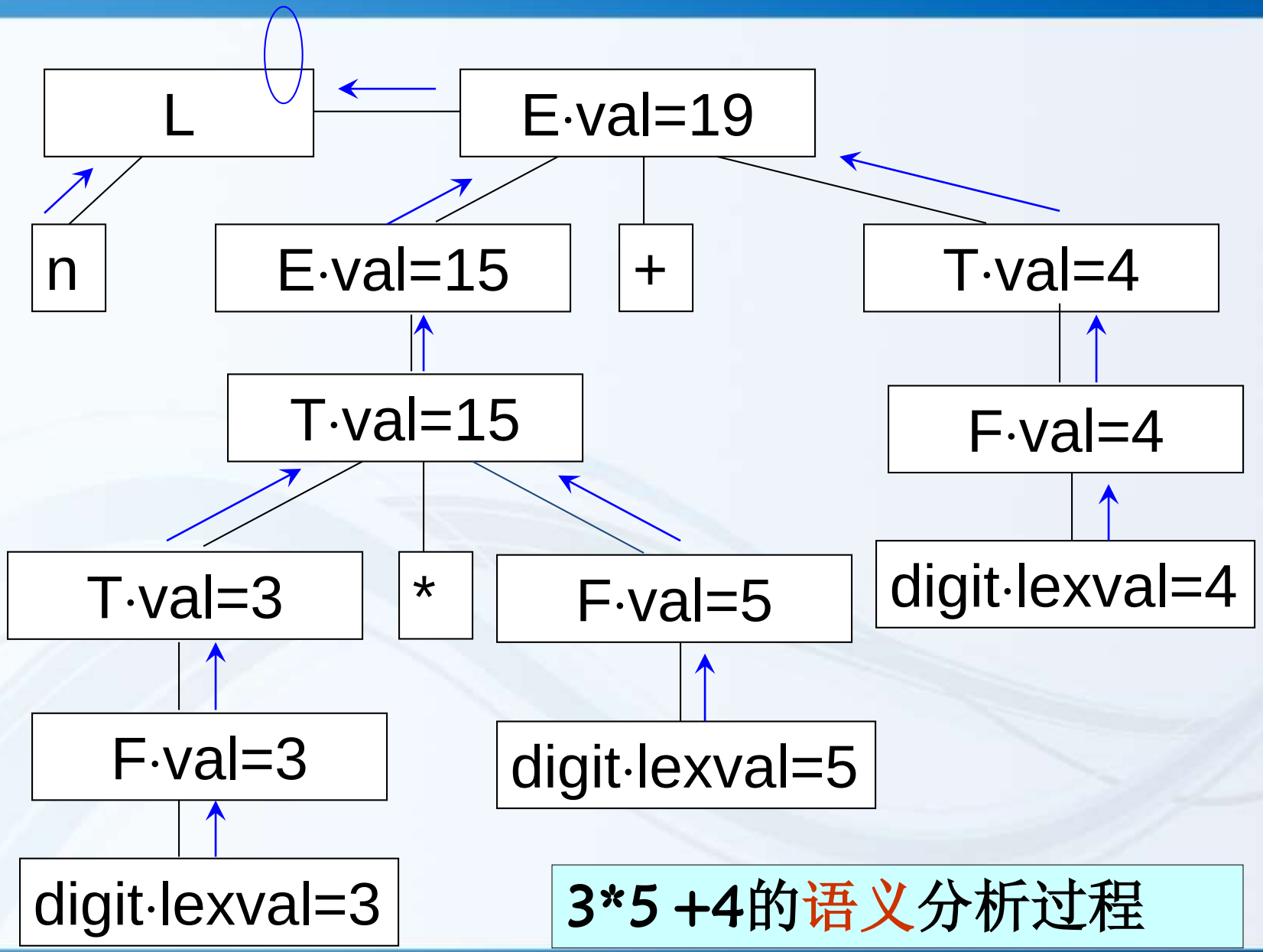
语法制导翻译是目前最常用的语义分析技术

例 3*5 +4的分析过程

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

3*5 +4的语法分析过程




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

3*5 +4的语义分析过程

语法制导翻译

- 采用文法来指导对程序的翻译过程。
- 通过向一个文法的产生式附加一些规则或程序片段来实现。
- 翻译方案和语法制导定义分别描述了两种不同的附加程序片段的方法。

语法制导定义

- 让每个文法符号与一个属性集合相关联
  - 属性：文法符号的一些特征。如：

    表达式的属性可以有类型、值等。
- 让每个产生式与一组语义规则相关联
  - 语义规则将计算与该产生式的文法符号相关联的属性的值

语法制导定义（例）

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

语法制导翻译方案

- 常简称为翻译方案
- 将语义动作直接嵌入到产生式规则中
  - 语义动作：嵌入到产生式内的程序片段
  - 翻译方案指定了语义动作的执行顺序。
- 画出一个翻译方案的语法分析树时，要同时描述语义信息：为每个语义动作分别构造一个额外的子结点，并用虚线将它和该产生式头部对应的结点相连。

翻译方案（例）

$expr \rightarrow expr_1 + term$ { print('+') }

$expr \rightarrow expr_1 - term$ { print('-') }

$expr \rightarrow term$

$term \rightarrow 0$ { print ('0') }

$term \rightarrow 1$ { print('1') }

. . .

$term \rightarrow 9$ { print ('9') }

5.1语法制导定义 (*Syntax Directed Definition*, SDD)

- 也叫属性文法，在上下文无关文法的基础上，为每个文法符号（终结符或非终结符）配备若干相关的“值”，（称为属性）：
  - 属性(*Attributes*) 代表文法符号相关信息，如类型、值、代码序列、符号表内容等
  - 属性可以计算和传递
  - 语义规则 (*Semantic Rules*)：对于文法的每个产生式都配备了一组属性的计算规则.

$E \rightarrow E_1 + T$　　$E.code = E_1.code \,||\, T.code \,||\, '+'$

例

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

语法制导定义(II)

- 属性分为综合属性和继承属性

  综合属性：“自下而上”传递信息

  继承属性：“自上而下”传递信息

- 规则$A\rightarrow\alpha$的语义规则可表示为$b=f(c_1,\ldots,c_n)$，f是一个函数
  - b是综合属性：如果b是A的属性，而$c_1\ldots c_n$是产生式右边α中的文法符号的属性,或者A的其它属性
  - b是继承属性：如果b是α中某个符号的属性，而$c_1\ldots c_n$是A或α中的文法符号的属性
  - 这两种情况下，都说属性b依赖于属性c1，c2，…，ck。

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

由词法分析器提供

说明：

- 终结符只有综合属性，由语法分析器提供，如：

  F → digit

  digit.*lexval*

- 非终结符既可有综合属性，也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值（即预先给出值）

  L → E n

- 对出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个计算规则。属性计算规则中只能使用相应产生式中的文法符号的属性。如：

  $F \rightarrow digit$

  $F.val = digit.lexval$

- 出现在产生式右边的综合属性和出现在产生式左边的继承属性不由产生式所给的属性计算规则进行计算，由其他产生式的属性计算规则或者由属性计算的参数提供
- 语义规则所描述的工作包括属性计算、静态语义检查、符号表操作、代码生成等

例：考虑非终结符A、B和C，A有一个继承属性a和一个综合属性b，B有综合属性c，C有继承属性d，产生式

$A \rightarrow BC$有规则

C.d=B.c+1

A.b=A.a+B.c

而属性A.a和B.c在其它地方计算

综合属性

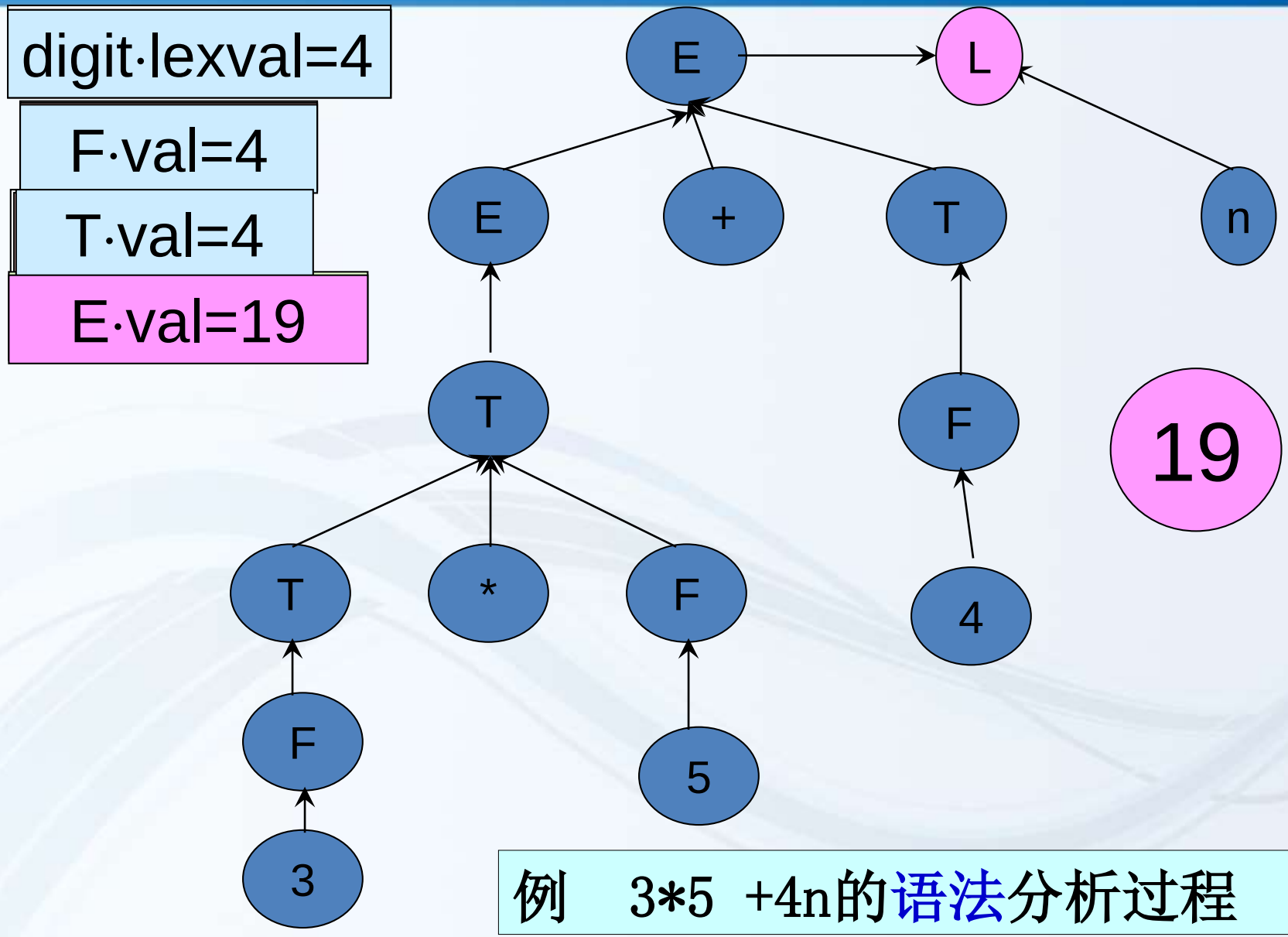
- 在语法树中，一个结点的综合属性的值可由子结点和自身的属性值计算得到

- S-属性定义(文法):只使用综合属性的语法制导定义。

- 利用S-属性文法进行语义分析时,结点属性值的计算正好和自底向上分析建立分析树结点同步进行。

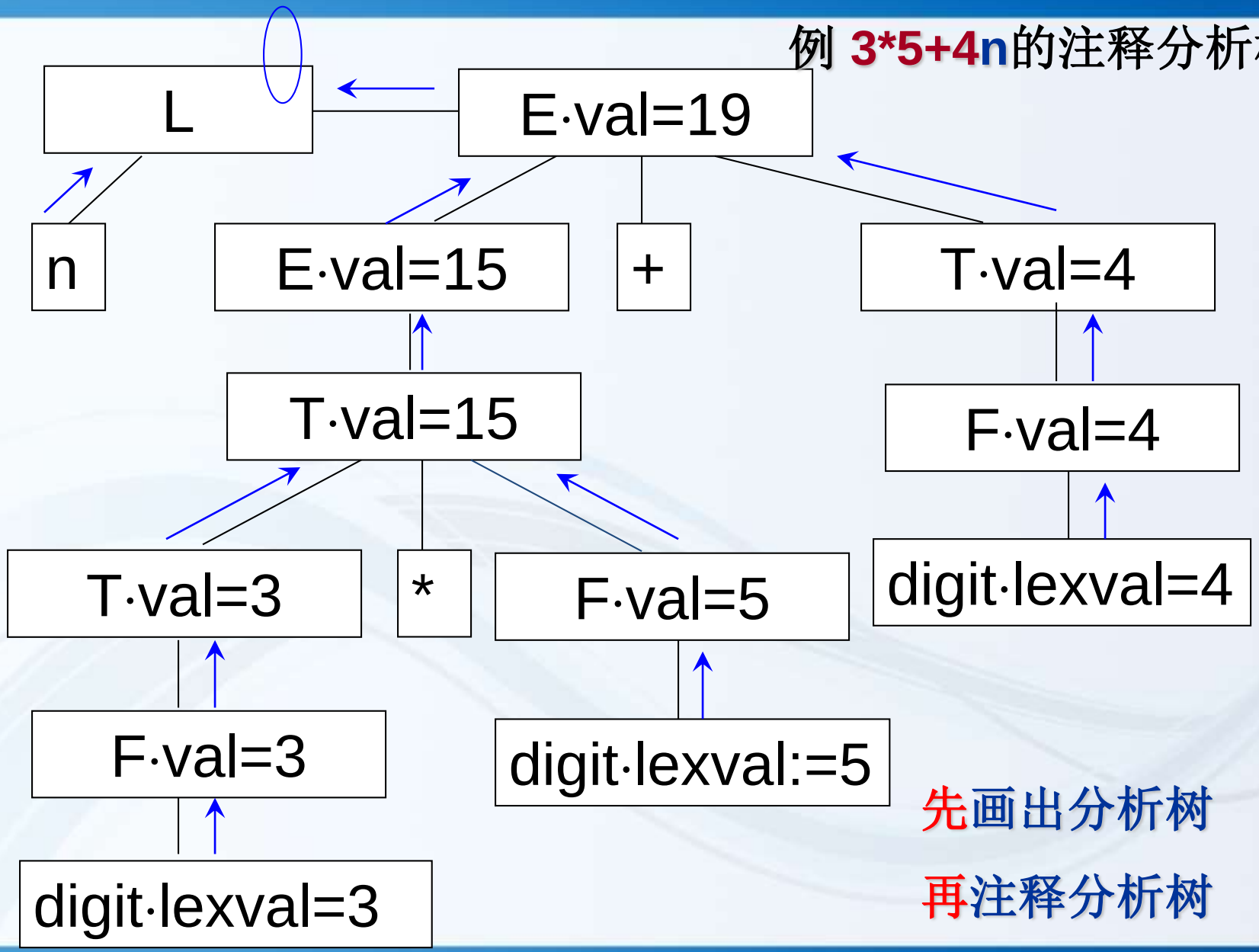
例：

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



例　3*5 +4n的语法分析过程

例 3*5+4n的注释分析树



先画出分析树

再注释分析树

注释分析树

- 在结点上标记了属性值的语法分析树。
  - 注释方法：先构建分析树，然后注释。

    如果只存在综合属性，则对分析树进行一次后序遍历，可以算出各个结点的属性值。

继承属性

- 在语法树中，一个结点的继承属性的值由父节点、其兄弟结点和自身的某些属性确定
- 用继承属性来表示程序设计语言结构中的上下文依赖关系很方便
- 结点属性值的计算使用自顶向下分析语法分析。
  - ✓ 计算方法：采用深度优先遍历方式计算

分析树的遍历

❖ 利用*dfvisit*来计算属性值

```
procedure dfvisit(n:node)
{
    for each child m of n from left to right do   //遍历子树
    {
        计算m的继承属性值;
        dfvisit(m)          //处理m子树内部的语义
    }
    计算n的综合属性值    //访问根
}
```

L属性定义 *L-Attributed Definitions*

- 与S属性定义类似，也是一种SDD
- 一个语法制导定义是L属性定义，如果 $\forall A \rightarrow X_1 X_2 \ldots X_n \in P$,其每一个语义规则中的每一个属性都是一个综合属性，或是$X_j(1 \le j \le n)$的一个继承属性，这个继承属性仅依赖于
  1、产生式中X_j的左边符号$X_1, X_2, \ldots X_{j-1}$的属性；
  2、A的继承属性。
- S属性定义也是L属性定义

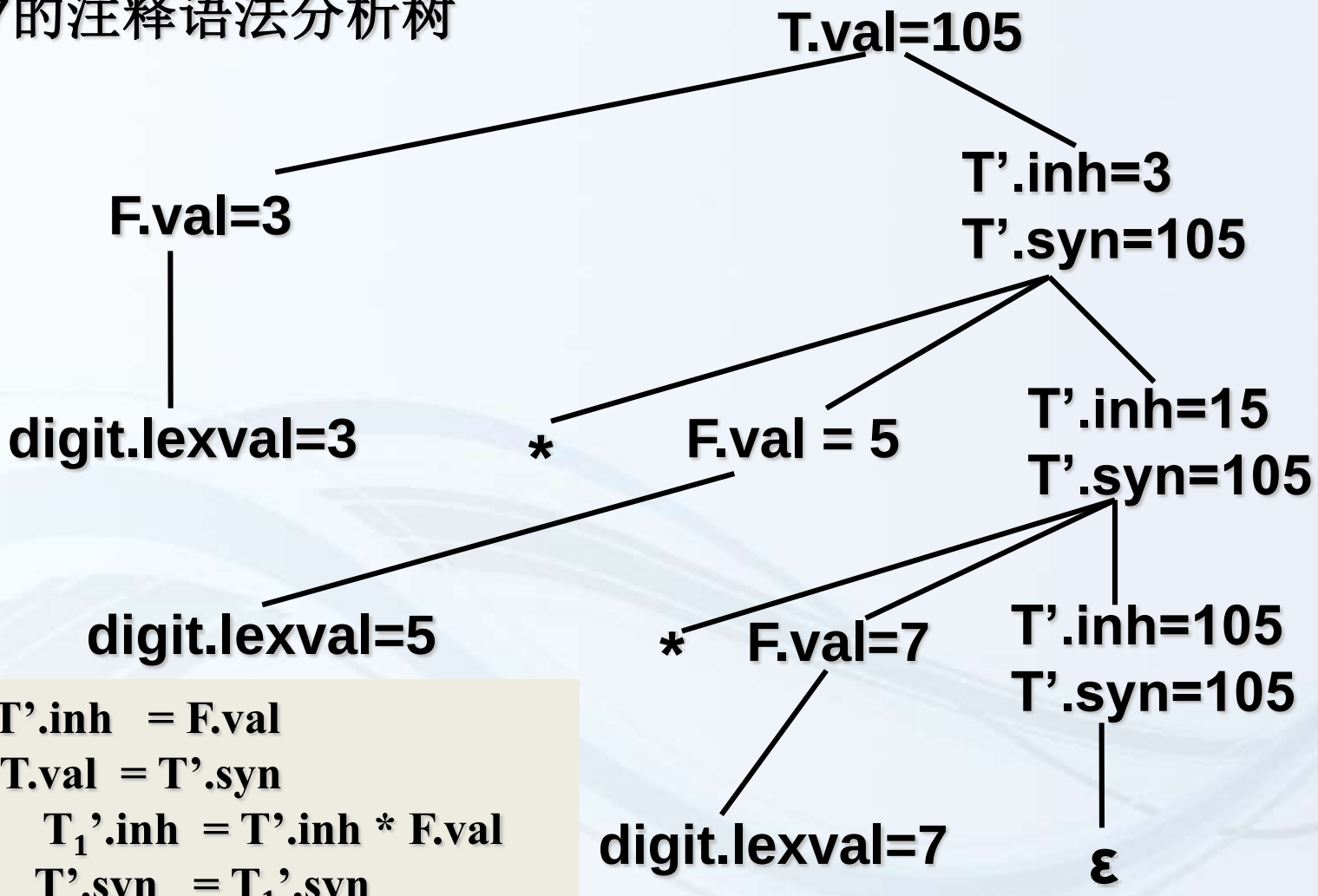
例:一个不是L属性定义的SDD

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

例

产生式	语义规则
$T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

例 3*5*7的注释语法分析树



$T \rightarrow FT'$ $T'.inh = F.val$
$T.val = T'.syn$

$T' \rightarrow *FT_1'$ $T_1'.inh = T'.inh * F.val$
$T'.syn = T_1'.syn$

$T' \rightarrow \varepsilon$ $T'.syn = T'.inh$

$F \rightarrow digit$ $F.val = digit.lexval$

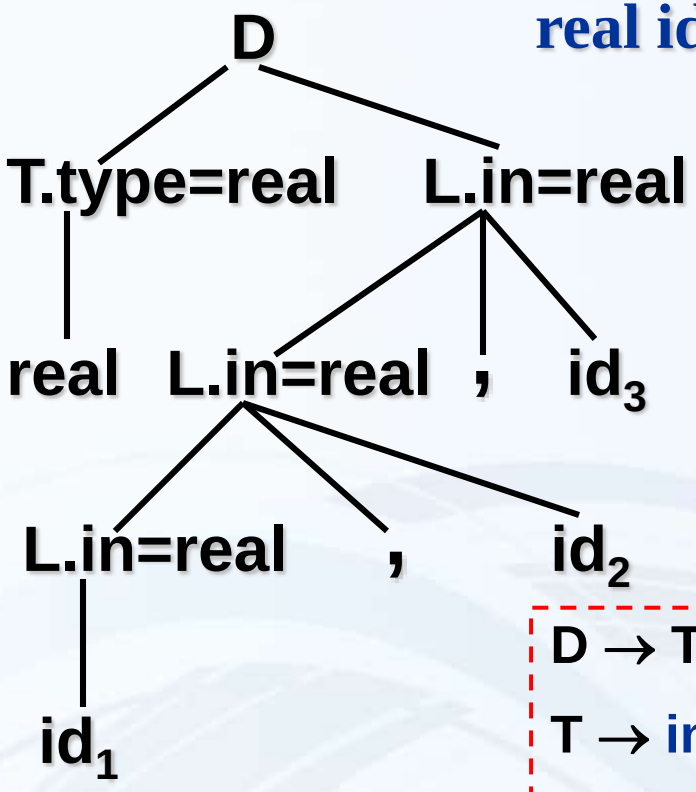
例（P202）

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

过程 ***addtype***(…)修改符号表：

在符号表相应项目插入类型信息，如***real***, ***int***

real id_1, id_2, id_3的注释语法分析树



…	…
id3	real
id2	real
id1	real

$D \rightarrow T\ L$	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(id.*entry*, L.*in*)
$L \rightarrow$ id	*addtype*(id.*entry*, L.*in*)

虚综合属性

依赖图（*Dependency Graph*）

- 在一棵语法树中的结点的综合属性和继承属性之间的相互依赖关系可以用依赖图（有向图）来描述。
- 为每一个包含过程调用的语义规则引用一个虚的综合属性b（哑综合属性），这样把每一个语义规则都写成以下形式：

$$b = f(c_1, \ldots, c_k)$$

- 依赖图中为每一个属性设置一个结点，如果属性b依赖于c，则从属性c的结点有一条有向边连到属性b的结点。

❖ 依赖图指出语法树中各结点属性值的计算顺序。

- 特点：先有文法，后确定语义计算顺序

❖ 构建依赖图：

for 分析树中的每个结点n do　　//构建结点

　for 结点n对应文法符号的每个属性a do

　　在依赖图中为a构建一个结点

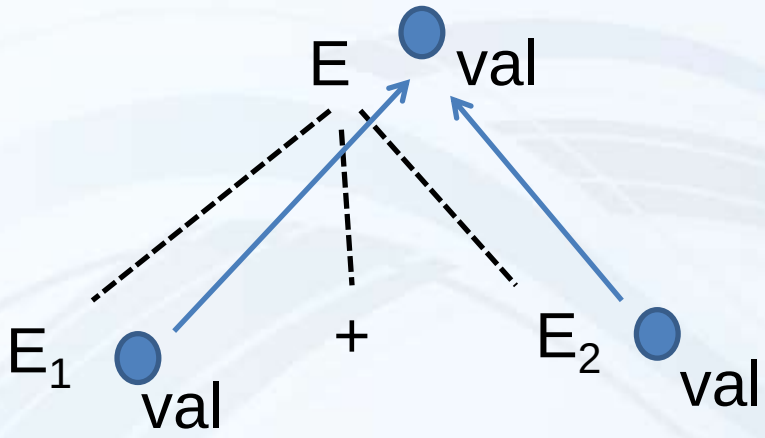
for 分析树中的每个结点n do　　//标记依赖

　for 结点n的产生式的每条语义规则$b=f(c_1,c_2,\ldots,c_k)$ do

　　for(i=1 to k) do

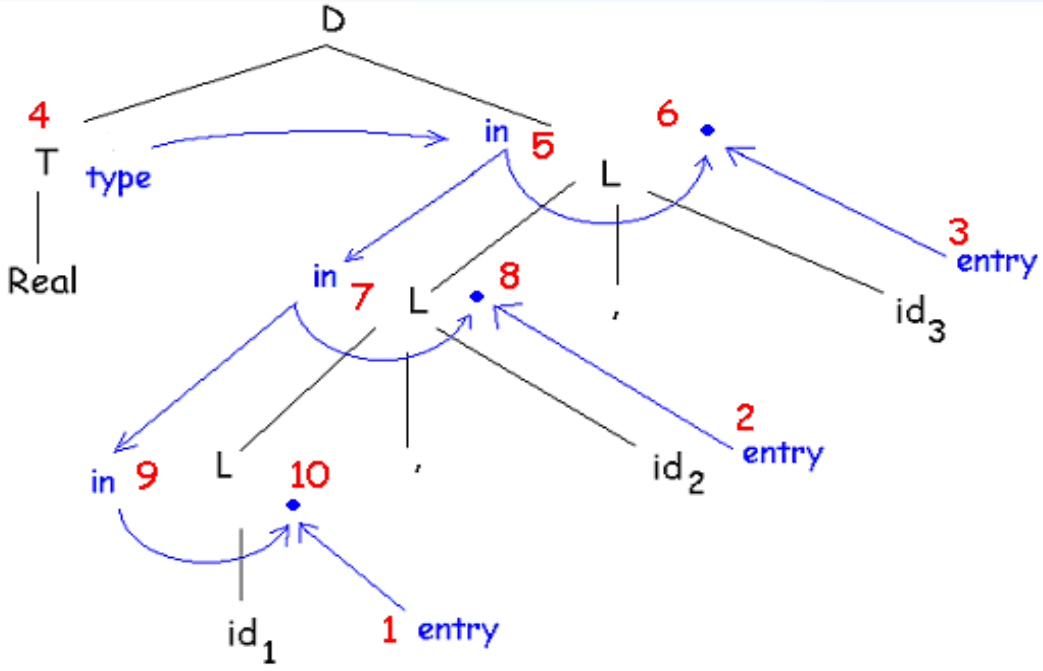
　　从结点c_i到结点b构造一条有向边

$E \rightarrow E_1 + E_2$ $E.val = E_1.val + E_2.val$



- 如果一个SDD中不存在属性之间的循环依赖关系（即依赖图中不存在环），则该文法是良定义的，依赖图存在拓扑排序
- 拓扑排序：对于节点$m_1,m_2,\ldots,m_k$，若$m_i \rightarrow m_j$是从m_i到m_j的边，那么在此排序中m_i先于m_j，有向图变成一个线性排序
- 一个依赖图的任何拓扑排序都给出一棵语法树中结点的语义规则计算的有效顺序
- 属性文法说明的翻译是很精确的
  - ✓ 基础文法用于建立输入符号串的语法分析树
  - ✓ 根据语义规则建立依赖图
  - ✓ 从依赖图的拓扑排序中，可以得到计算语义规则的顺序

句子real id_1, id_2, id_3带注释语法分析树的依赖图



Production	Semantic Rules
$D \rightarrow T\ L$	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*(**id**.entry, L.in)
$L \rightarrow$ **id**	• := *addtype*(**id**.entry, L.in)

dummy attributes (nodes 6,8,10)

```
a4 := real;
a5 := a4;
addtype(id3.entry, a5);
a7 := a5;
addtype(id2.entry, a7);
a9 := a7;
addtype(id1.entry, a9);
```

计算属性值过程(依赖图方法）

- 创建分析树（自顶向下或自底向上）
- 构建依赖图（基于分析树和语义规则）
- 对依赖图，进行拓扑排序
- 计算属性值

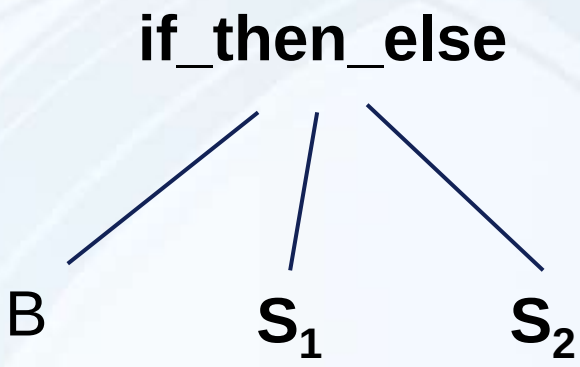
计算属性值过程(一遍扫描的处理方法）

- 是在语法分析的同时计算属性值
  - ✓所采用的语法分析方法
  - ✓属性的计算次序
- L-属性定义适合于一遍扫描的自上而下分析
- S-属性定义适合于一遍扫描的自下而上分析

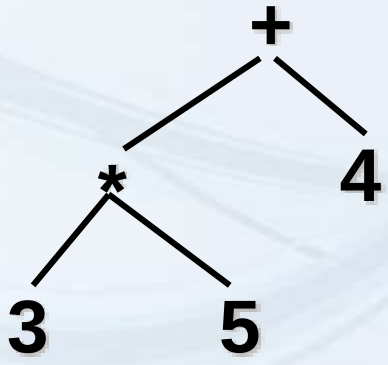
抽象语法树

- 在语法分析树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称为抽象语法树（**Abstract Syntax Tree**）

如：**S→if B then S_1 else S_2**

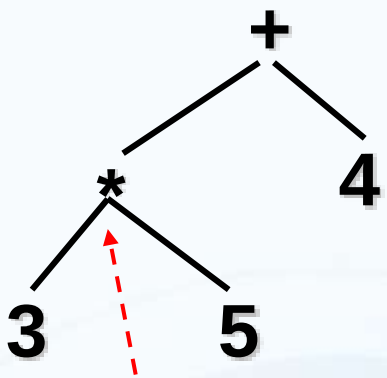


3*5+4

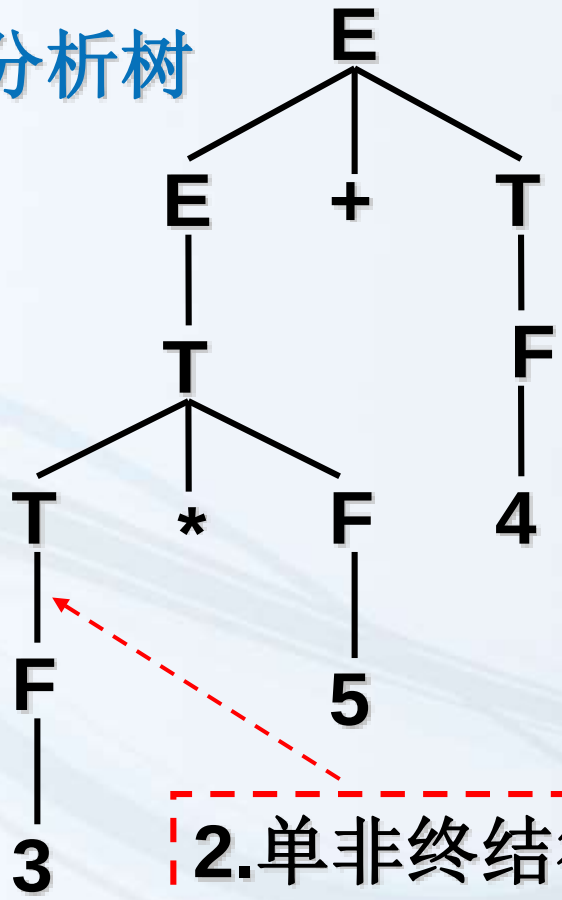


构建语法树（例）

语法树



分析树



1.运算符作为内部结点

2.单非终结符产生式链可能消失

建立表达式的抽象语法树

- 两个建立结点的函数（返回值为指向新建立结点的指针）：
  - ***Node***(op,c_1,c_2,…,c_k) 建立内部结点，op为运算符，其余k个字段的值为指向其它结点的指针.
  - ***Leaf***(op,val) 建立叶结点，op为记号，*val*为指向符号表相关项目的指针。

建立抽象语法树的语义规则

❖ 应用综合属性 ***node*** (语法树的结点)

产生式	语义规则
$E \rightarrow E_1 + T$	E.*node* = new *node*(“+”,E_1.*node* ,T.*node*)
$E \rightarrow E_1 - T$	E.*node* = new *node*(“-”,E_1. *node* ,T. *node*)
$E \rightarrow T$	E.*node* = T.*node*
$T \rightarrow (E)$	T.*node* = E.*node*
$T \rightarrow id$	T.*node* = new *leaf*(id, id.*lexval*)
$T \rightarrow num$	T.*node* = new *leaf*(num, num.*val*)

a-4+c的抽象语法树

产生式	语义规则
$E \rightarrow E_1 + T$	E.*node* = new *node*(“+”,E_1.*node* ,T.*node*)
$E \rightarrow E_1 - T$	E.*node* = new *node*(“-”,E_1. *node* ,T. *node*)
$E \rightarrow T$	E.*node* = T.*node*
$T \rightarrow (E)$	T.*node* = E.*node*
$T \rightarrow id$	T.*node* = new *leaf*(id, id.*lexval*)
$T \rightarrow num$	T.*node* = new *leaf*(num, num.*val*)