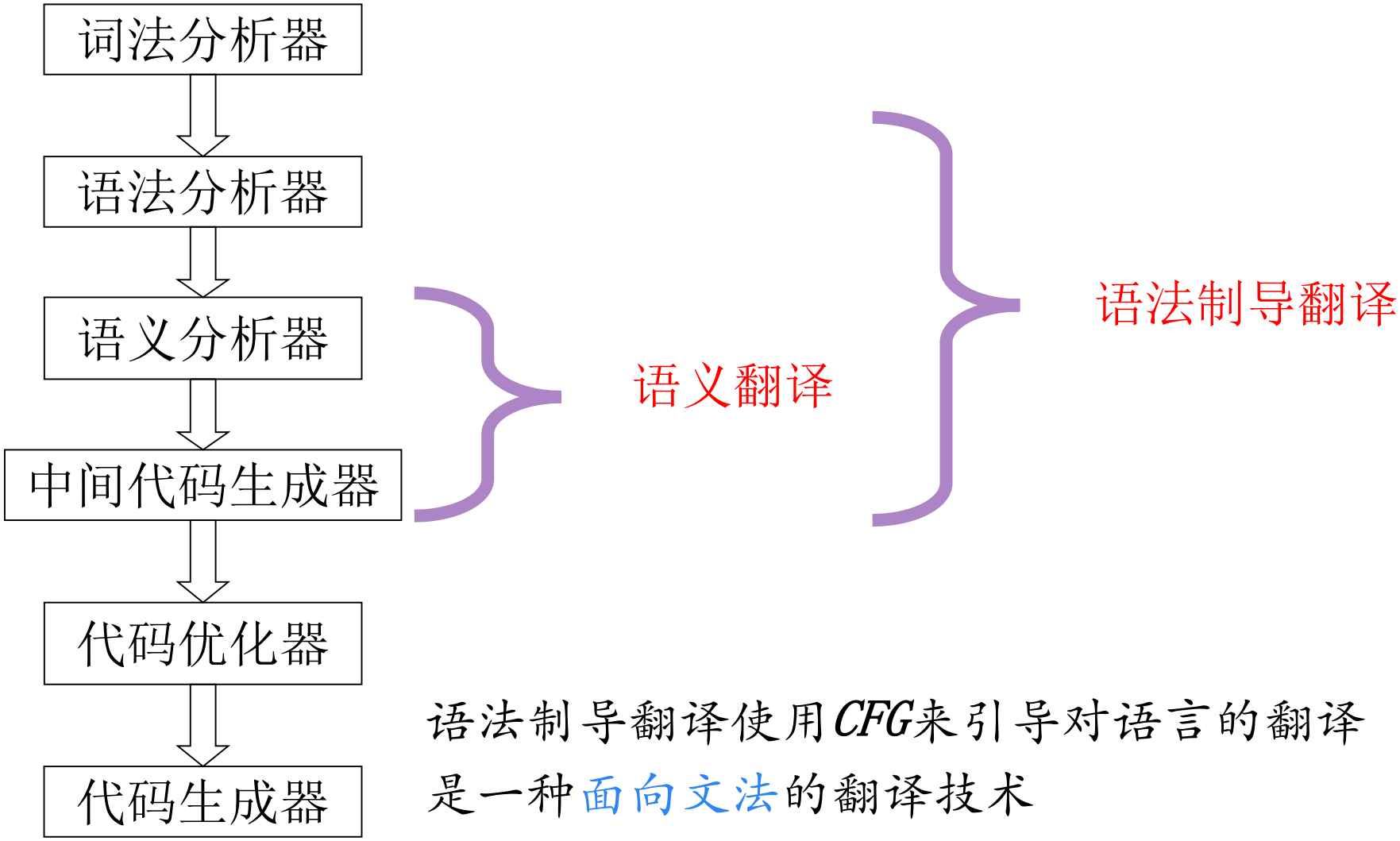
第五章 语法制导翻译



语法制导翻译使用*CFG*来引导对语言的翻译

是一种面向文法的翻译技术

学习内容

- 5.1 语法制导定义(SDD)
- 5.2 SDD的求值顺序
- 5.3 S属性与L属性
- 5.4 语法制导的翻译方案

概述

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

“老鼠吃猫” 问题

语法正确的句子，它的语义可能存在问题

概述

语义分析器的任务：

① 检查各个语法结构的静态语义 （静态语义分析或静态检查 ）

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

将静态检查和中间代码生成结合到语法分析中进行的技术称为语法制导翻译

语法制导翻译的基本思想

- 在进行语法分析的同时，完成相应的语义处理。也就是说，一旦语法分析器识别出一个语法结构就要立即对其进行翻译，翻译是根据语言的语义进行的，并通过调用事先为该语法结构编写的语义子程序来实现。
- 对文法中的每个产生式附加一个/多个语义动作(或语义子程序)，在语法分析的过程中，每当需要使用一个产生式进行推导或归约时，语法分析程序除执行相应的语法分析动作外，还要执行相应的语义动作(或调用相应的语义子程序)。

语法制导翻译

语法制导的翻译：一种形式化的语义描述方法，包括两种具体形式：

- 语法制导定义(Syntax-Directed Definitions, SDD)：定义翻译所必须的语义属性和语义规则，一般不涉及计算顺序。
- 翻译模式(translation schemes)： 给出语义规则的计算顺序。

语法制导定义

语法制导定义:

- 一个上下文无关文法。
- 每个属性与文法的一个终结符或非终结符相关联
- 每一个产生式和一个语义规则集合相关联。描述产生式中各文法符号的属性之间的依赖关系。通常用函数或程序语句的形式表示

语法制导定义

属性的抽象表示

例如：E.val（值）

E.type（类型）

E.code（代码序列）

E.place（存储空间）

语法制导定义

文法G：

- $E \rightarrow T^1+T^2 | T^1$ or T^2
- $T \rightarrow num | true | false$

语法制导定义

语义规则：

产生式	语义规则
$E \rightarrow T^1 + T^2$	$T^1.t=int$ AND $T^2.t=int$
$E \rightarrow T^1 or T^2$	$T^1.t=bool$ AND $T^2.t=bool$
$T \rightarrow num$	$T.t=int$
$T \rightarrow true$	$T.t=bool$
$T \rightarrow false$	$T.t=bool$

语法制导定义

语法制导定义：

每个文法产生式$A \to \alpha$有一组形式为$b = f(c_1, c_2, \cdots, c_k)$的语义规则，其中$f$是函数，$b$和$c_1, c_2, \cdots, c_k$是该产生式文法符号的属性

语义规则中的属性存在下述性质与关系

(1) 若b是A的属性，c1, c2, ..., ck是α中文法符号的属性，或者A的其它属性，则称b是A的综合属性。

(2) 若b是α中某文法符号X的属性，c1, c2, ..., ck是A的属性，或者是α中其它文法符号的属性，则称b是X的继承属性

语法制导定义

语法制导定义：

每个文法产生式$A \rightarrow \alpha$有一组形式为$b = f(c_1, c_2, \cdots, c_k)$的语义规则，其中$f$是函数，$b$和$c_1, c_2, \cdots, c_k$是该产生式文法符号的属性

语义规则中的属性存在下述性质与关系

(3) 称属性b依赖于属性c1, c2, ..., ck。实质上反映了属性计算的先后次序，即所有属性ci被计算之后才能计算属性b。

(4) 若语义规则的形式如下，则可将其想像为产生式左部文法符号A的一个虚拟属性。属性之间的依赖关系，在虚拟属性上依然存在。

f(c1, c2, ..., ck)

具有受控副作用的语义规则

受控副作用（controlled side effects）

- 副作用：一般属性值计算（基于属性值或常量进行的）之外的功能，如
  - 打印出一个结果
  - 将信息加入符号表

具有受控副作用的语义规则

- 例子：
  - 哑（虚）综合属性：L → E n　print(E.val)
  - 符号表操作：分析real id_1,id_2,id_3

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

语法制导定义

属性的特点：

- 一个结点的综合属性的值通过分析树中它的子结点的属性值和自己的属性值计算的；
- 继承属性的值由结点的父结点、兄弟结点来计算的；
- 非终结符号即可有综合属性也可有继承属性，但文法开始符号没有继承属性；
- 终结符号只有综合属性，由词法分析器提供，即记号的属性。
- 每个文法符号的综合属性和继承属性的交集为空；

语法制导定义

语法制导定义是一种接近形式化的语义描述方法。

语法制导定义的表示分两部分：

- 先针对语义为文法符号设置属性，
- 然后为每个产生式设置语义规则，来描述各属性间的关系。

一般将语法制导定义写成表格形式，每个文法规则用相应规则的语义规则列出。

语法制导定义

产生式	语义规则
产生式1	相关的属性规则
. .	. .
. .	. .
产生式n	相关的属性规则

语法制导定义

综合属性

分析树结点N上的非终结符号A的综合属性只能通过N的子结点或N本身的属性值来定义。

产生式的头一定是A.

语法树自底向上计算属性

终结符只有综合属性，由词法分析器提供

语法制导定义

继承属性

分析树结点N上的非终结符号B的继承属性只能通过N的父结点，N本身和N的兄弟结点上的属性值来定义。

产生式的体中必然包含符号B.

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

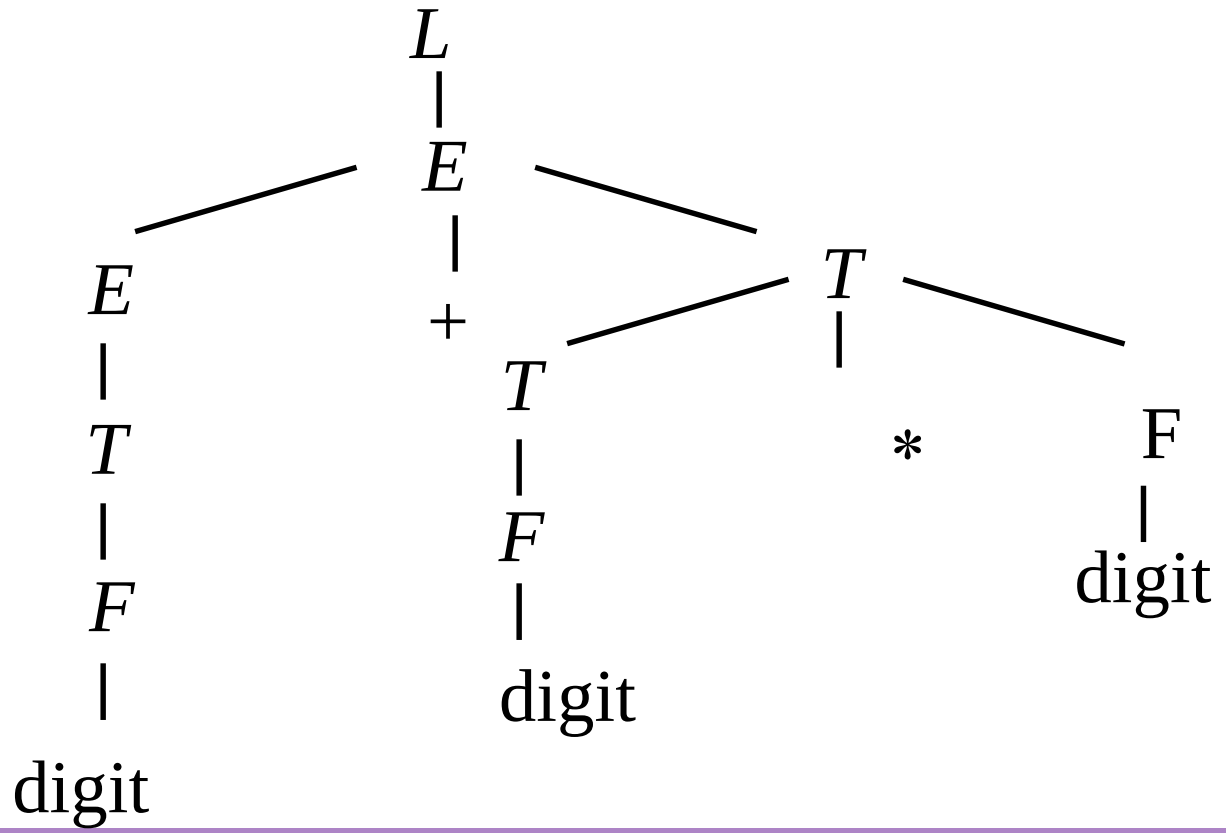
综合属性

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*

综合属性

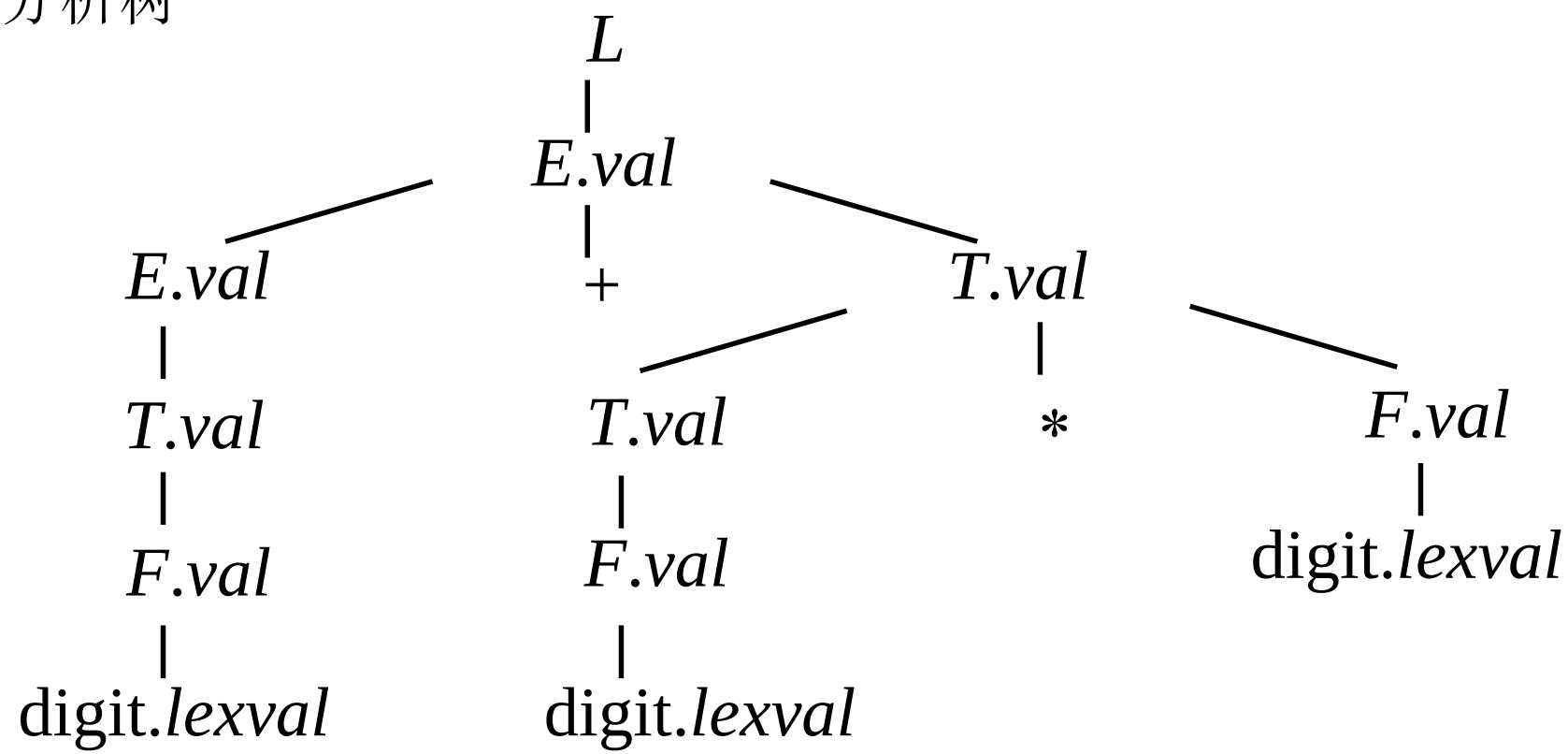
注释分析树：将属性附着在分析树对应文法符号上（属性作为分析树的注释）

8+5*2的分析树



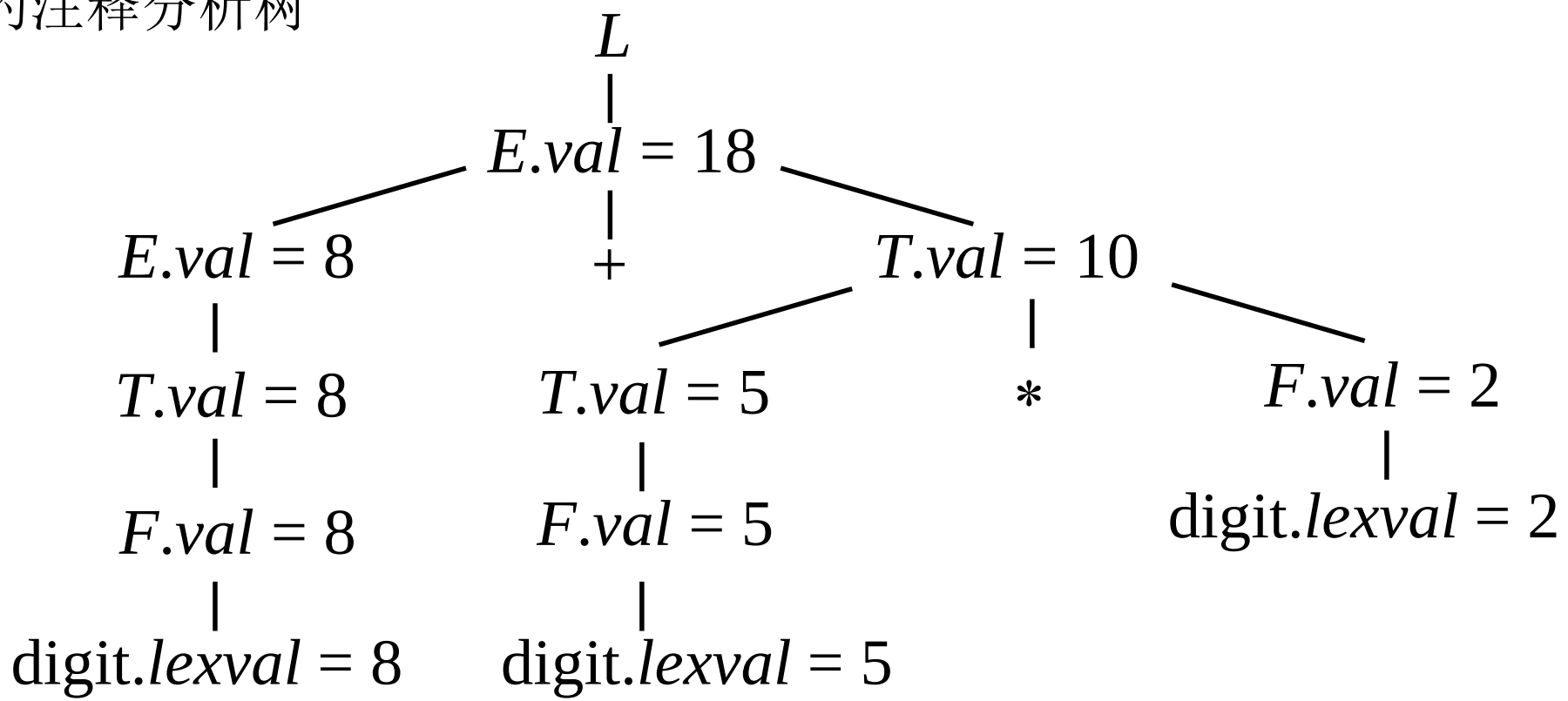
综合属性

- 注释分析树：将属性附着在分析树对应文法符号上（属性作为分析树的注释）
- 8+5*2的注释分析树



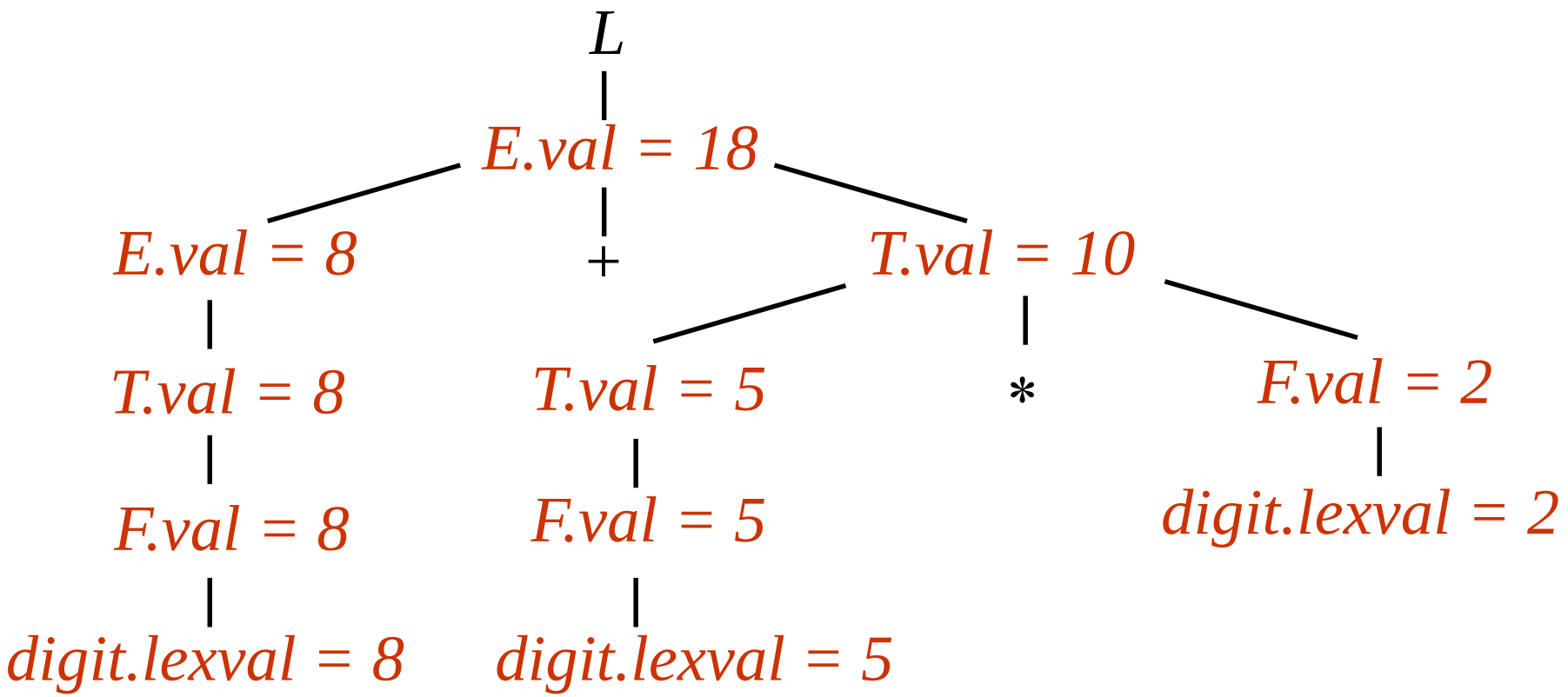
综合属性

- 注释分析树：将属性附着在分析树对应文法符号上（属性作为分析树的注释）
- 8+5*2的注释分析树



综合属性

分析树各结点属性的计算可以自下而上地完成



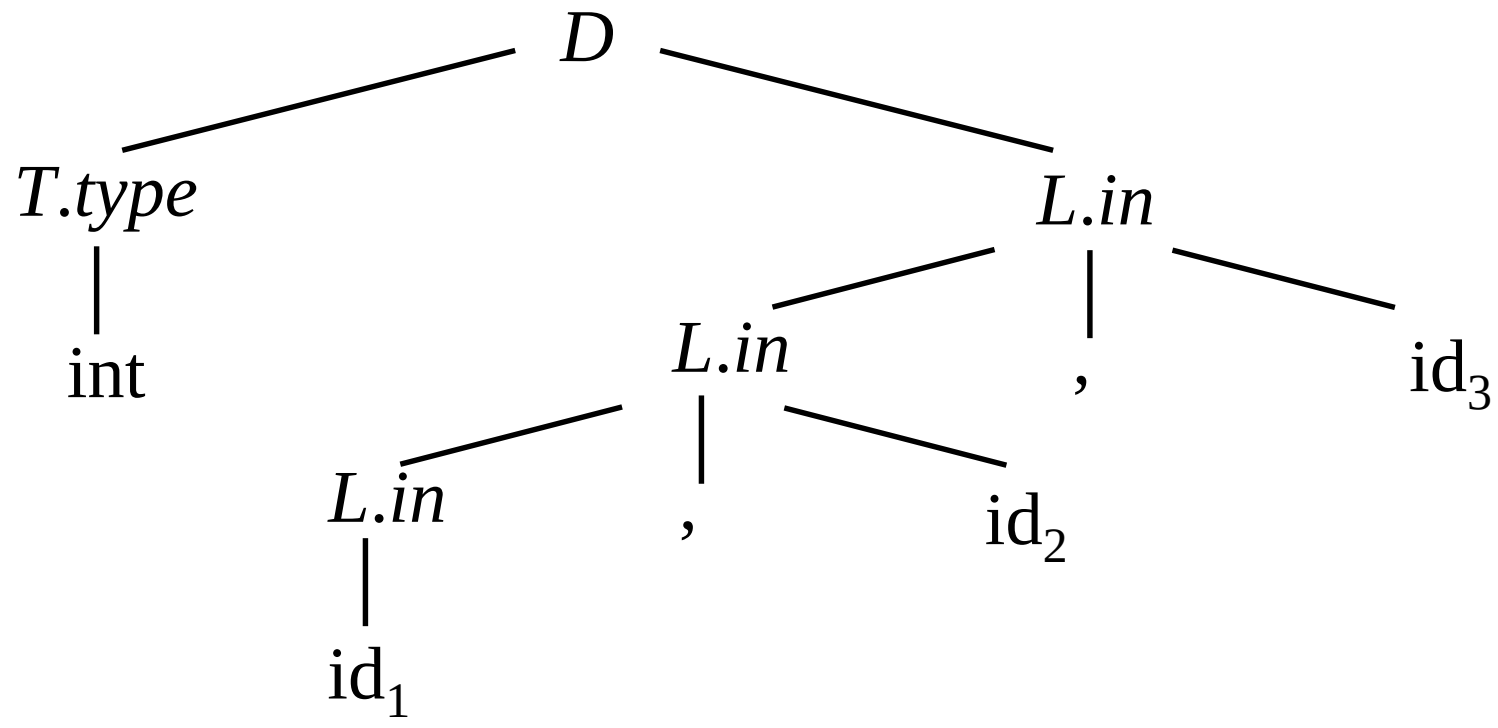
继承属性

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

继承属性

int id_1, id_2, id_3的注释分析树



SDD的求值顺序

输入符号串

→分析树

→依赖图

→语义规则的计算顺序

- 对单词符号串进行语法分析，构造语法分析树；
- 根据需要构造属性依赖图；
- 遍历语法树，并在语法树各结点处按语义规则进行计算；

构造属性依赖图

- 依赖图：表示文法符号属性之间依赖关系的有向图。
- 构造方法：

- X的每个属性都在依赖图中有一个结点
- X.c → 综合属性A.b，从X.c到A.b的边
- X.a → 继承属性B.c，从X.a到B.c的边

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, …, c_k) **do**
        for i = 1 to k **do**
            构造从c_i到b的一条边

属性依赖图

real id_1, id_2, id_3的分析树的依赖图

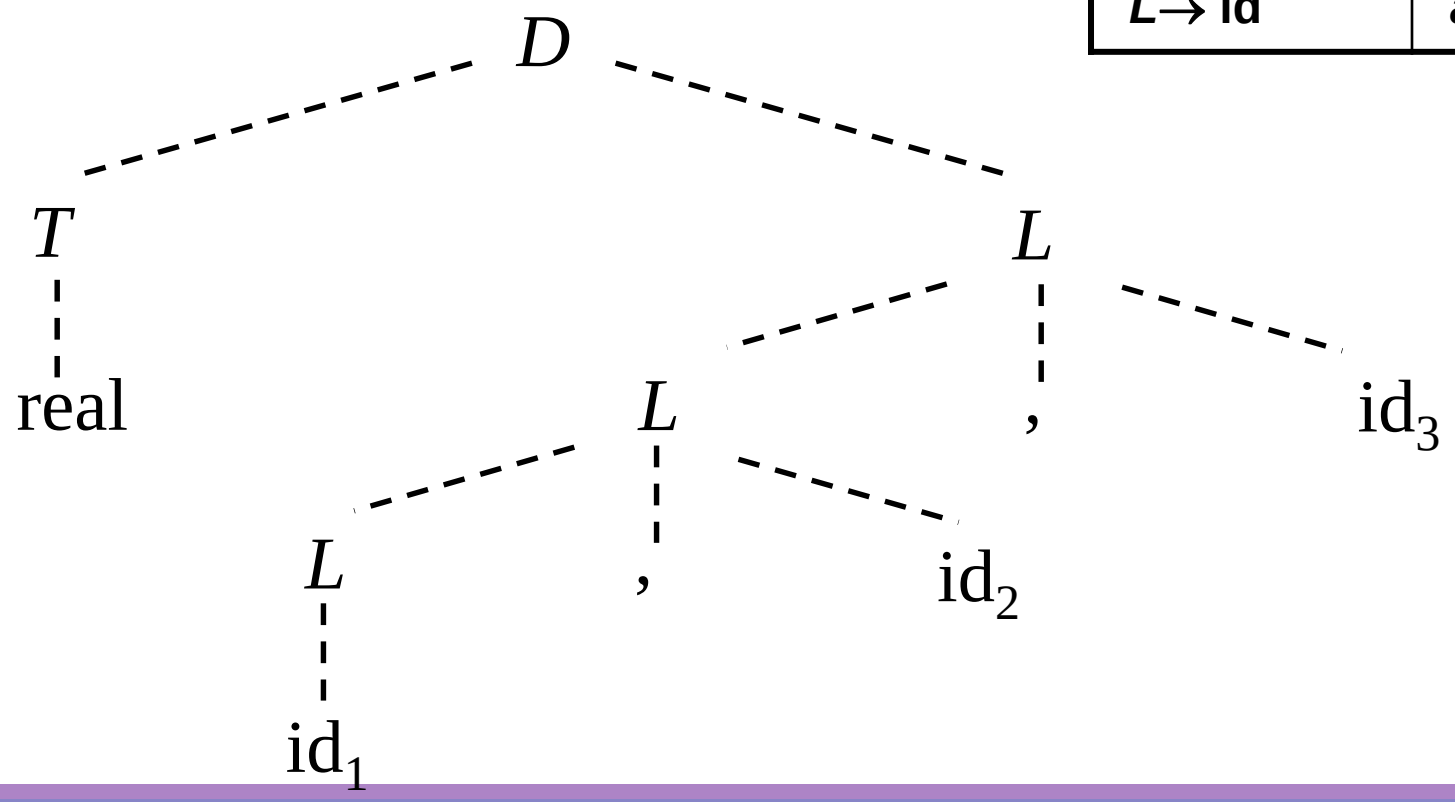
产　生　式	语　义　规　则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addType$(id.$entry$, $L.in$)
$L \rightarrow$ id	$addType$(id.$entry$, $L.in$)

属性依赖图

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

real id_1, id_2, id_3

第一步：画语法树

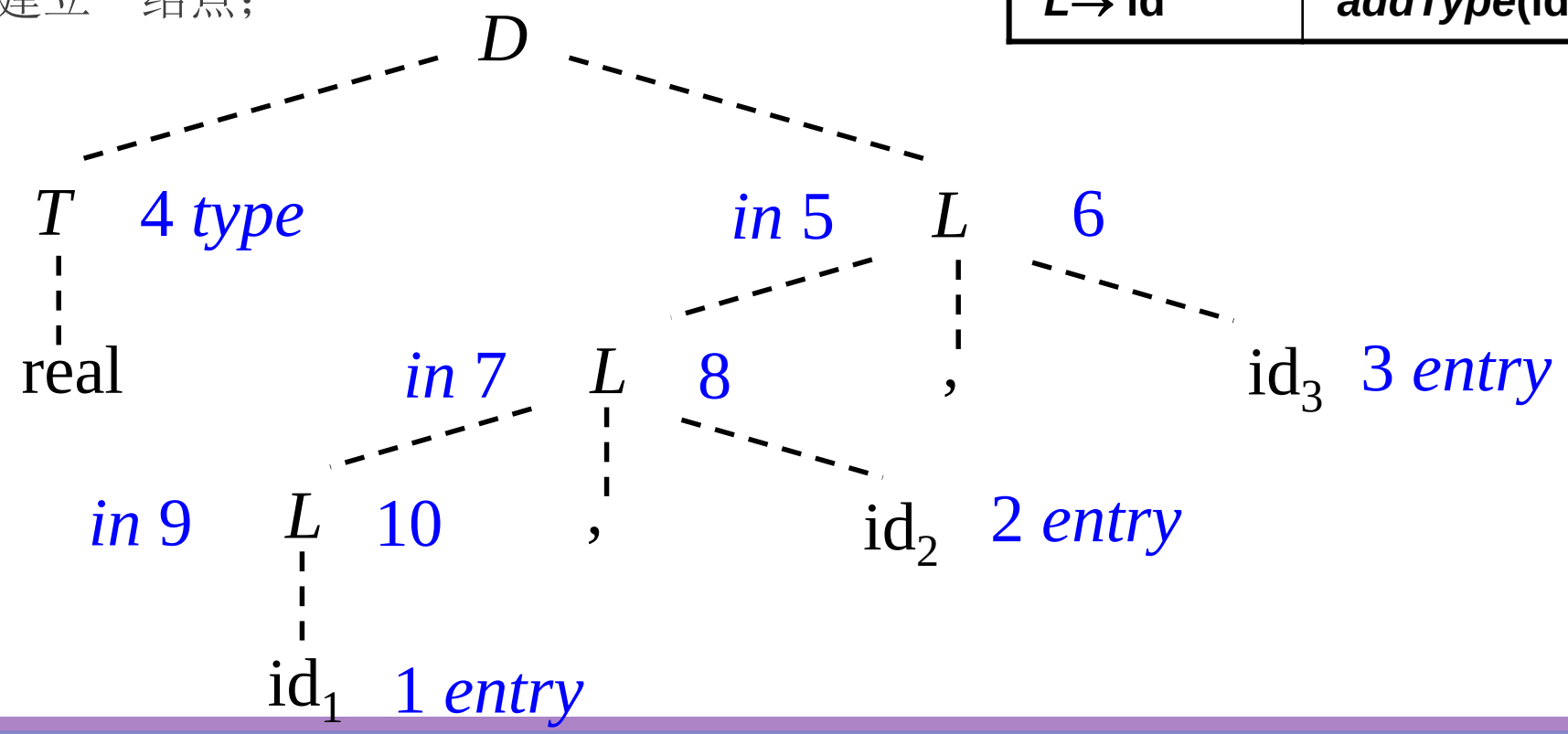


属性依赖图

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

real id_1, id_2, id_3

第二步：每一属性建立一结点；



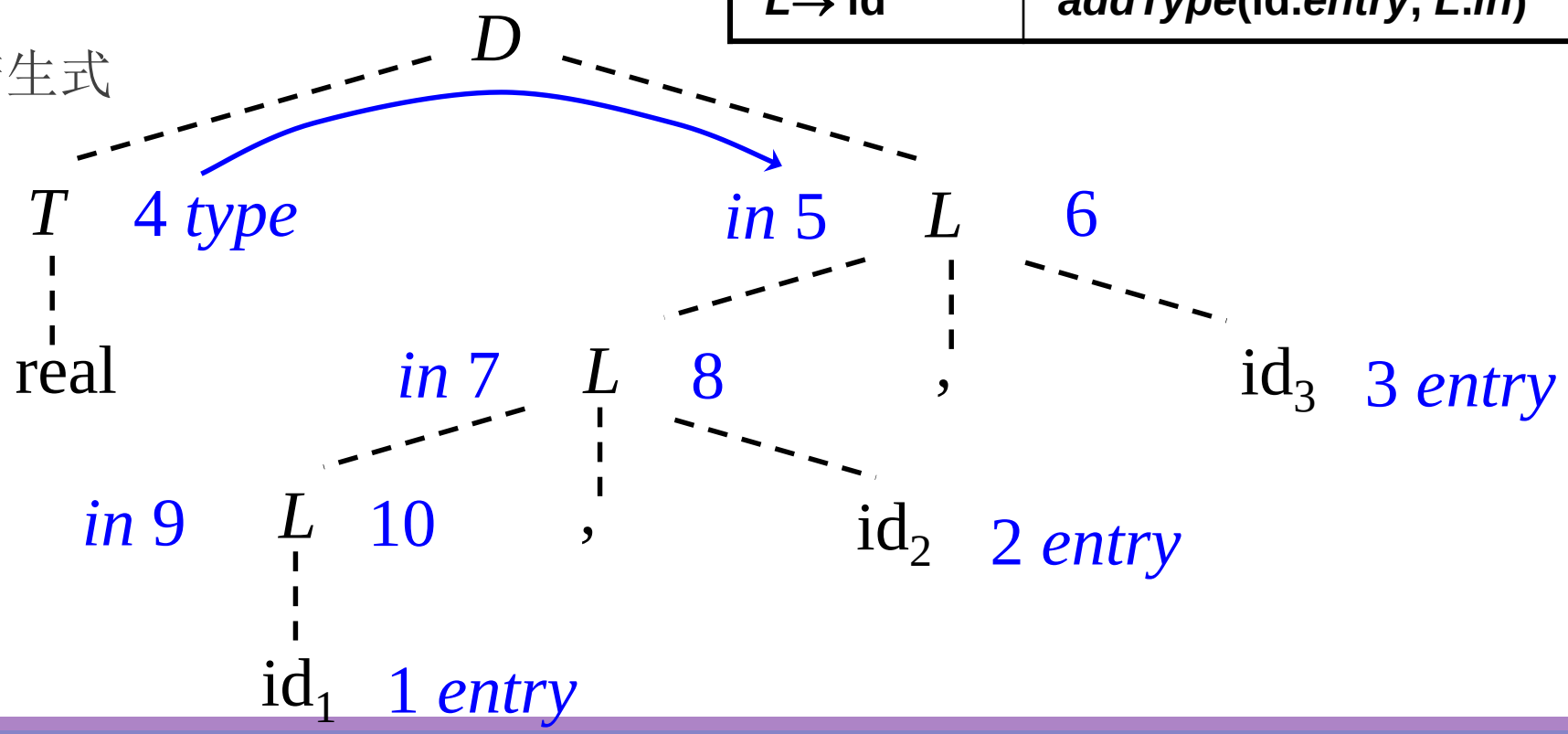
结点6、8、10是为*addtype*过程引入的虚属性

属性依赖图

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

real id_1, id_2, id_3

第三步：为每一产生式构造有向边；

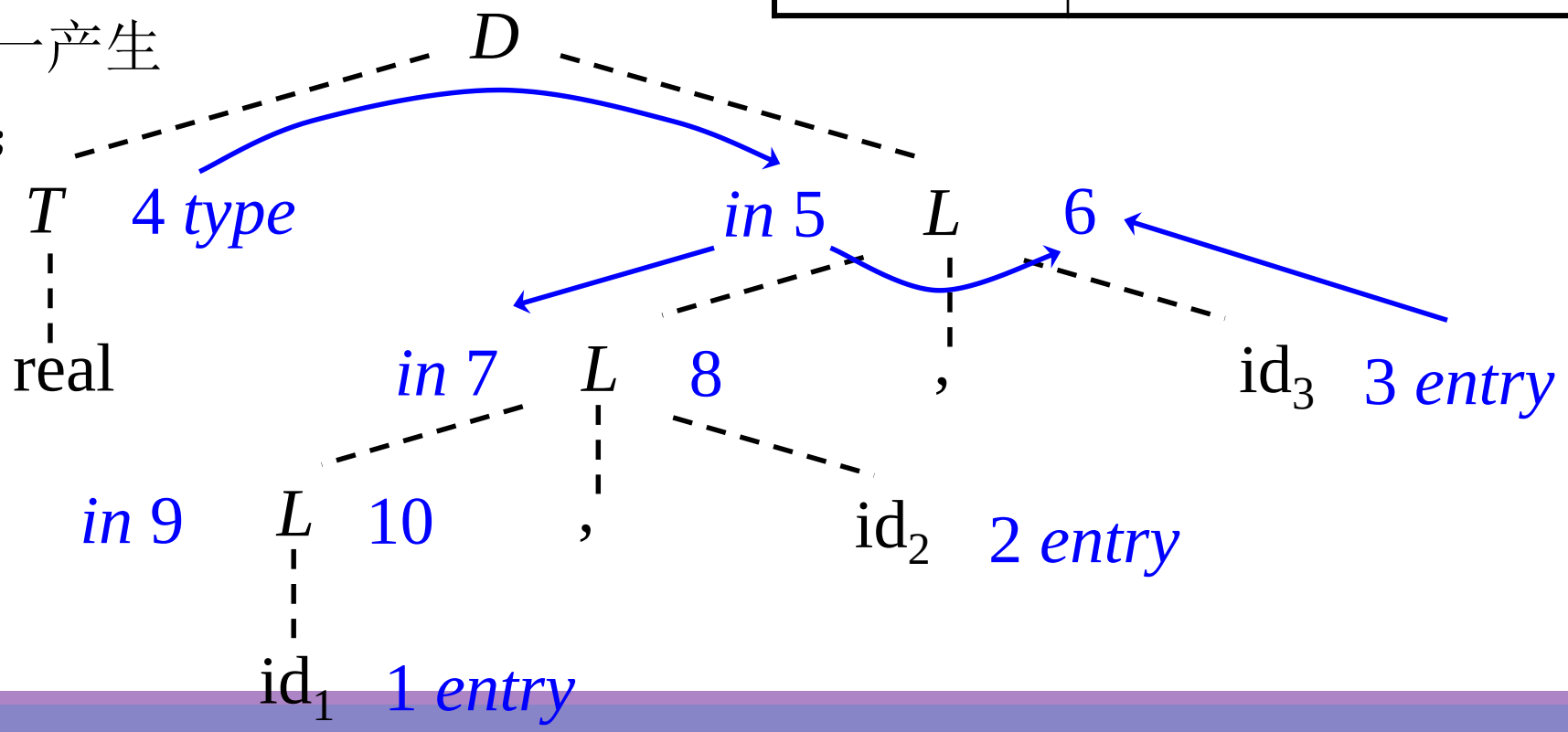


属性依赖图

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

real id_1, id_2, id_3

第三步：为每一产生式构造有向边；

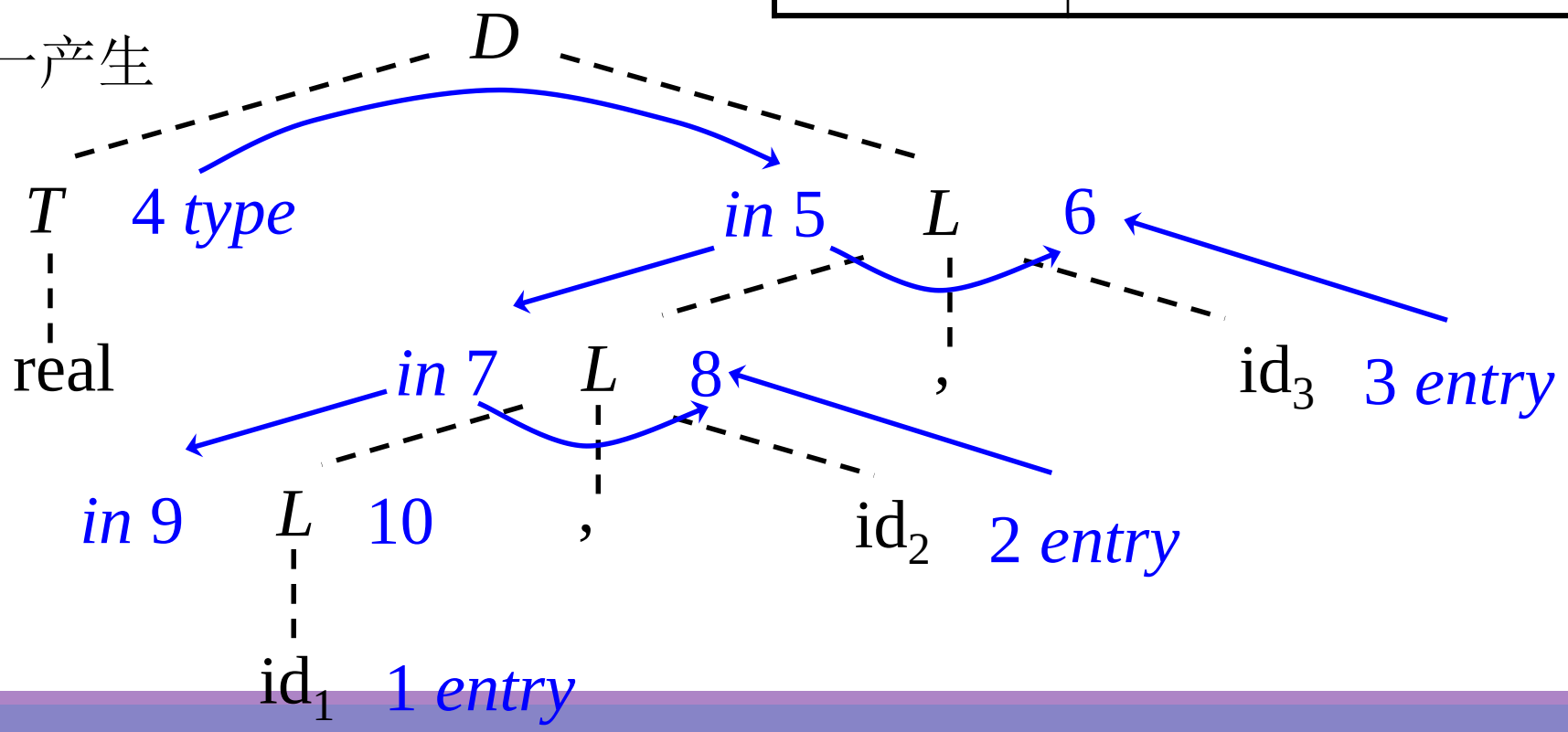


属性依赖图

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

real id_1, id_2, id_3

第三步：为每一产生式构造有向边；

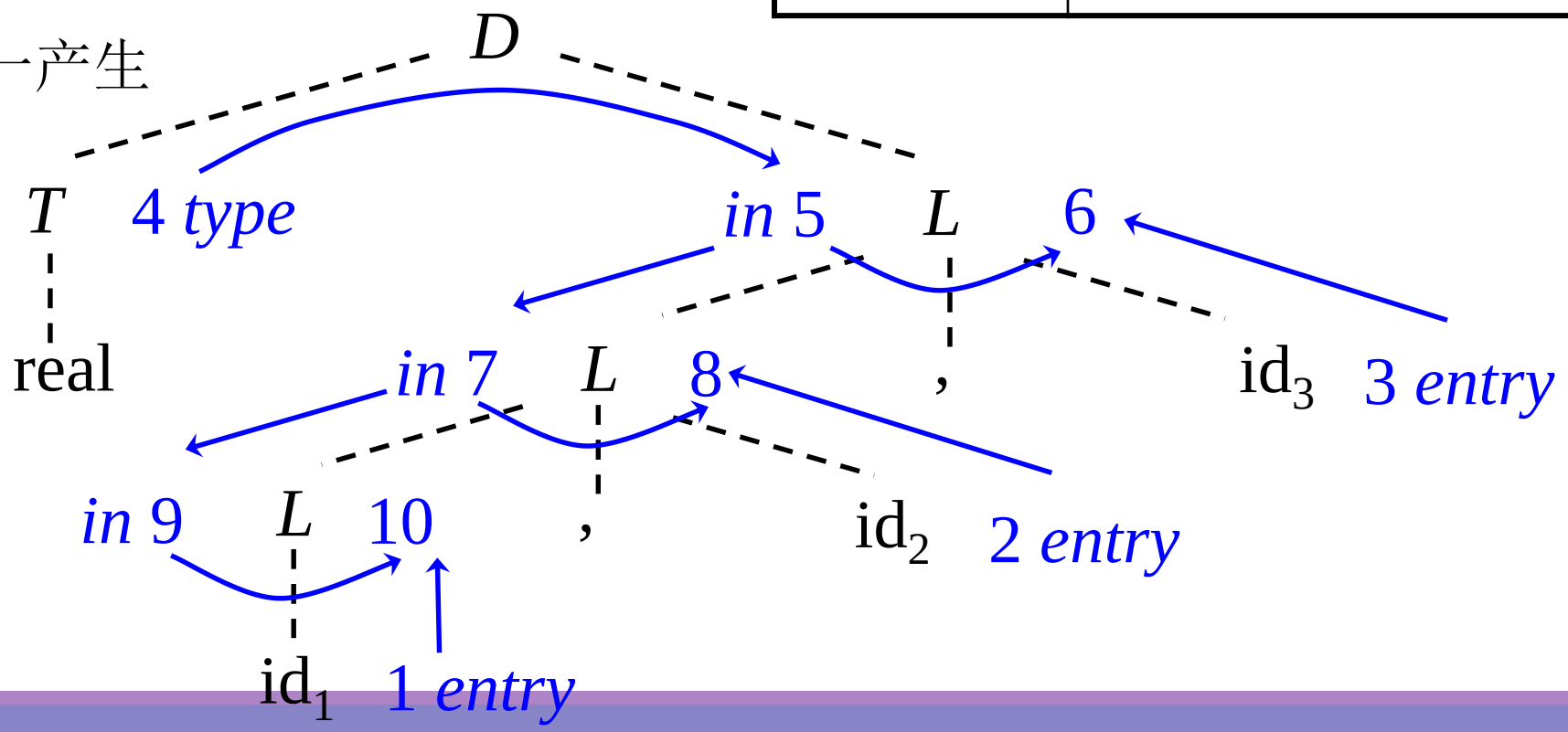


属性依赖图

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

real id_1, id_2, id_3

第三步：为每一产生式构造有向边；



计算语义规则

拓扑排序

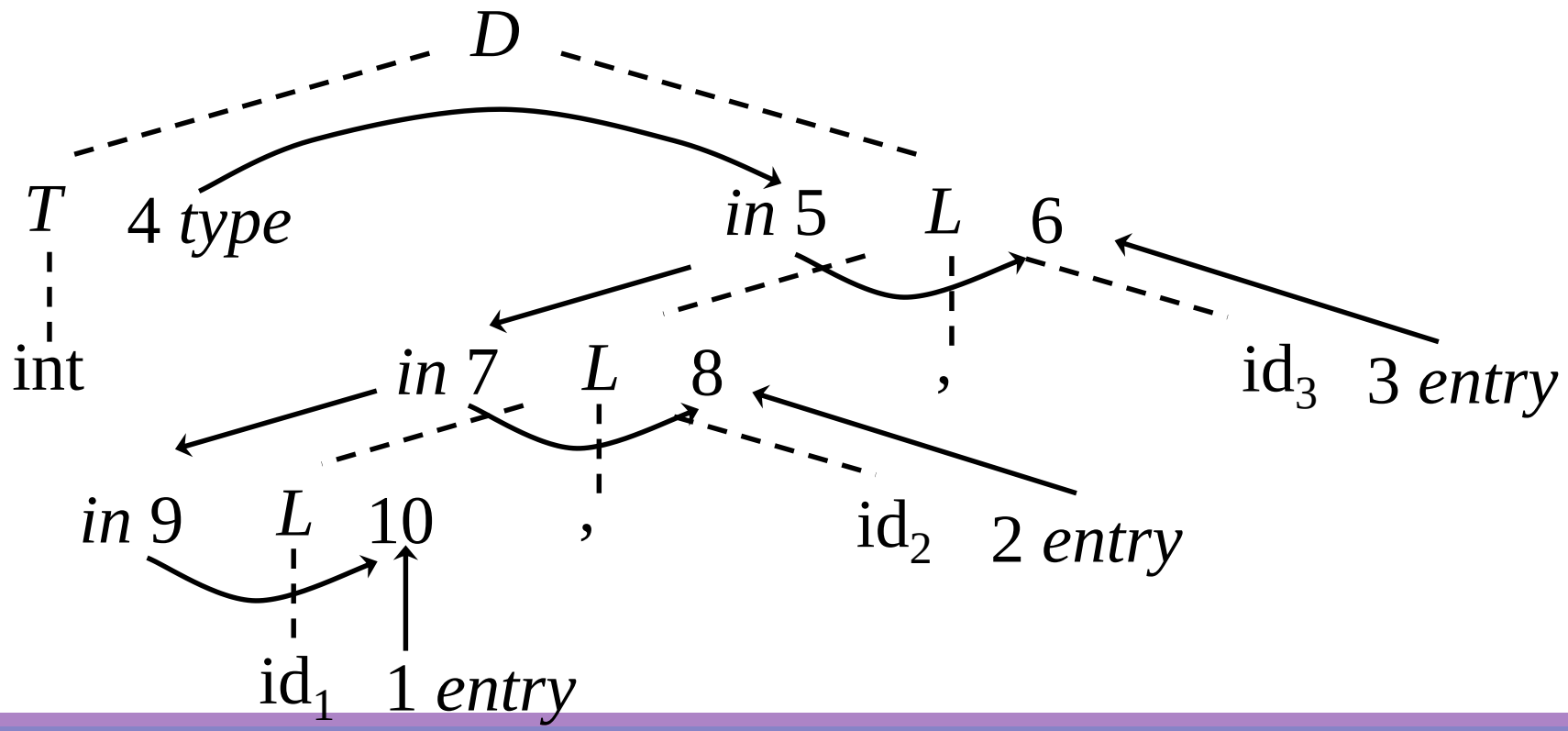
一个有向非循环图的拓扑排序是图中结点的任何顺序m_1，m_2，…，m_k，使得边必须是从序列中前面的结点指向后面的结点，也就是说，如果 $m_i \rightarrow m_j$ 是 m_i 到 m_j 的一条边,那么在序列中 m_i 必须出现在 m_j 的前面。

若依赖图中无环，则存在一个拓扑排序，它就是属性值的计算顺序。

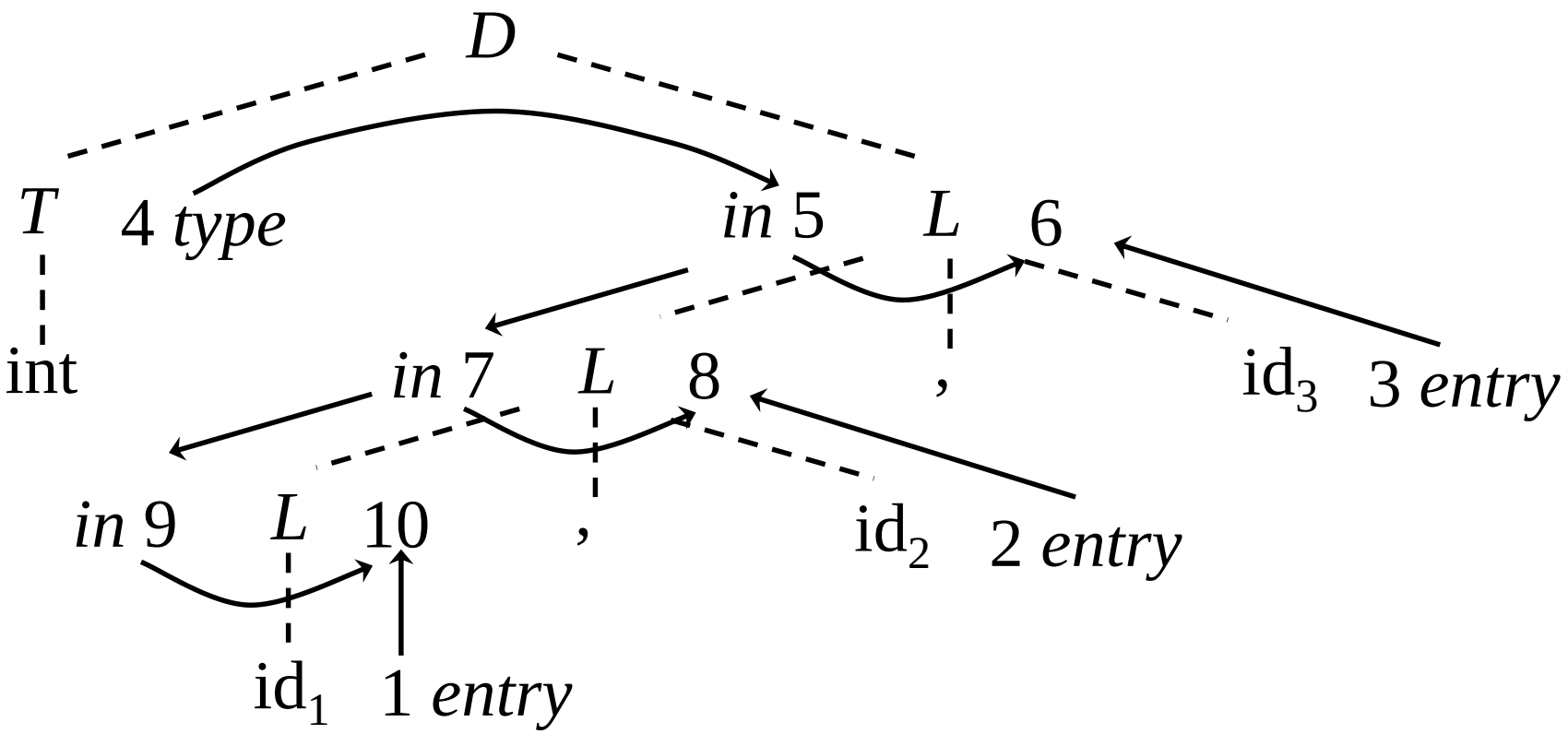
计算语义规则

拓扑排序：结点的一种排序，使得边只会从该次序中先出现的结点到后出现的结点

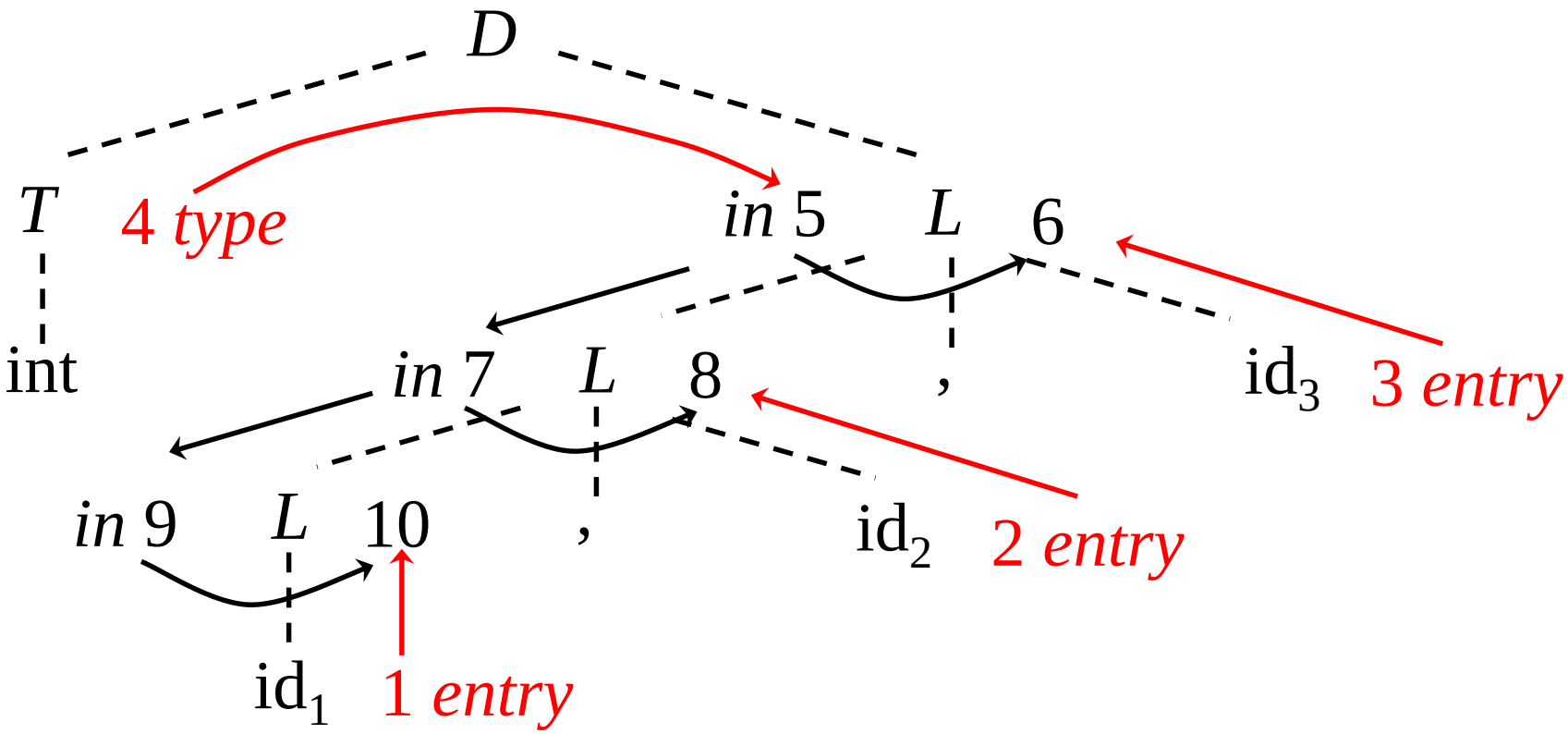
例：1，2，3，4，5，6，7，8，9，10



计算语义规则

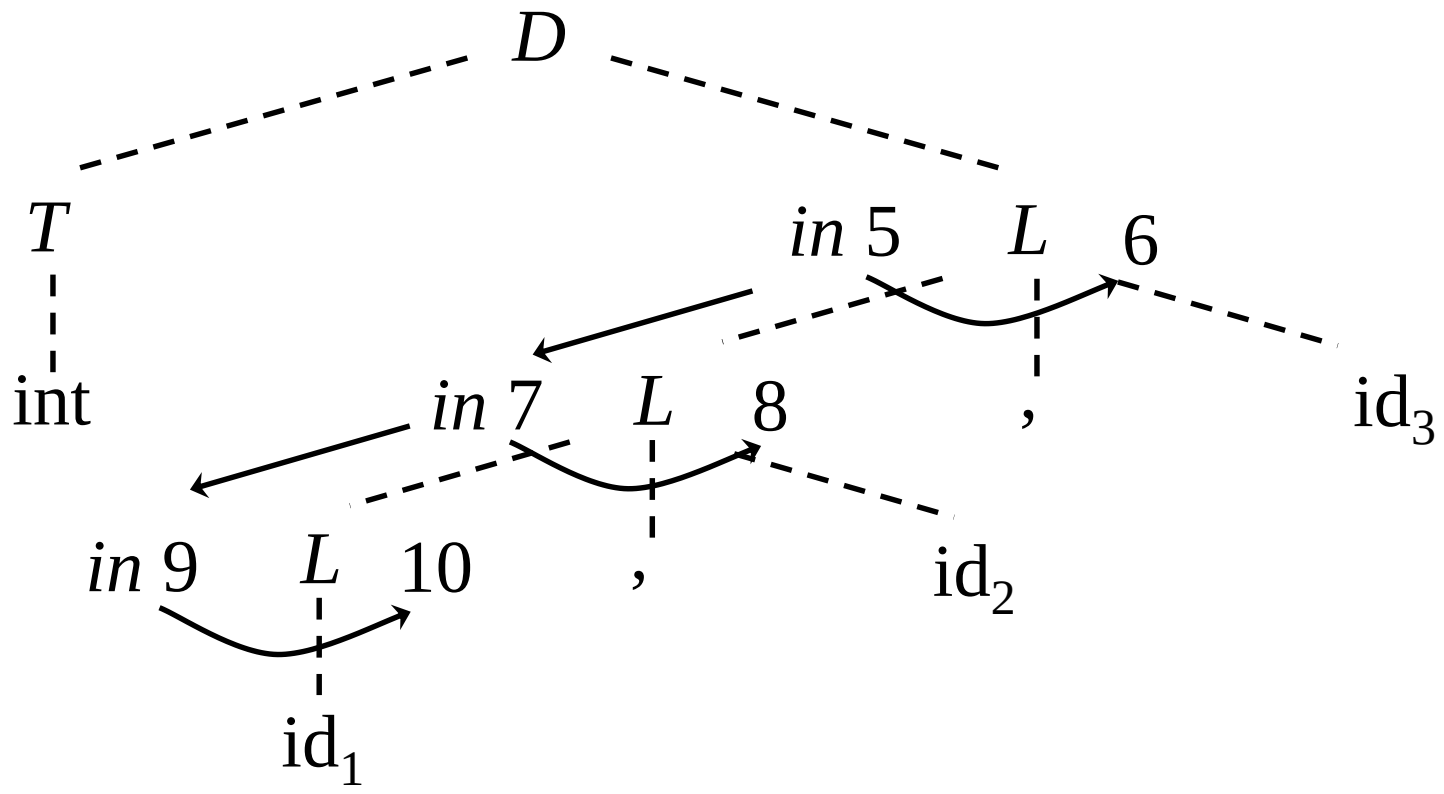

D
T
4 type
in 5
L
6
int
in 7
L
8
,
id3
3 entry
in 9
L
10
,
id2
2 entry
id1
1 entry

计算语义规则


D
T
4 type
in 5
L
6
int
in 7
L
8
,
id3
3 entry
in 9
L
10
,
id2
2 entry
id1
1 entry

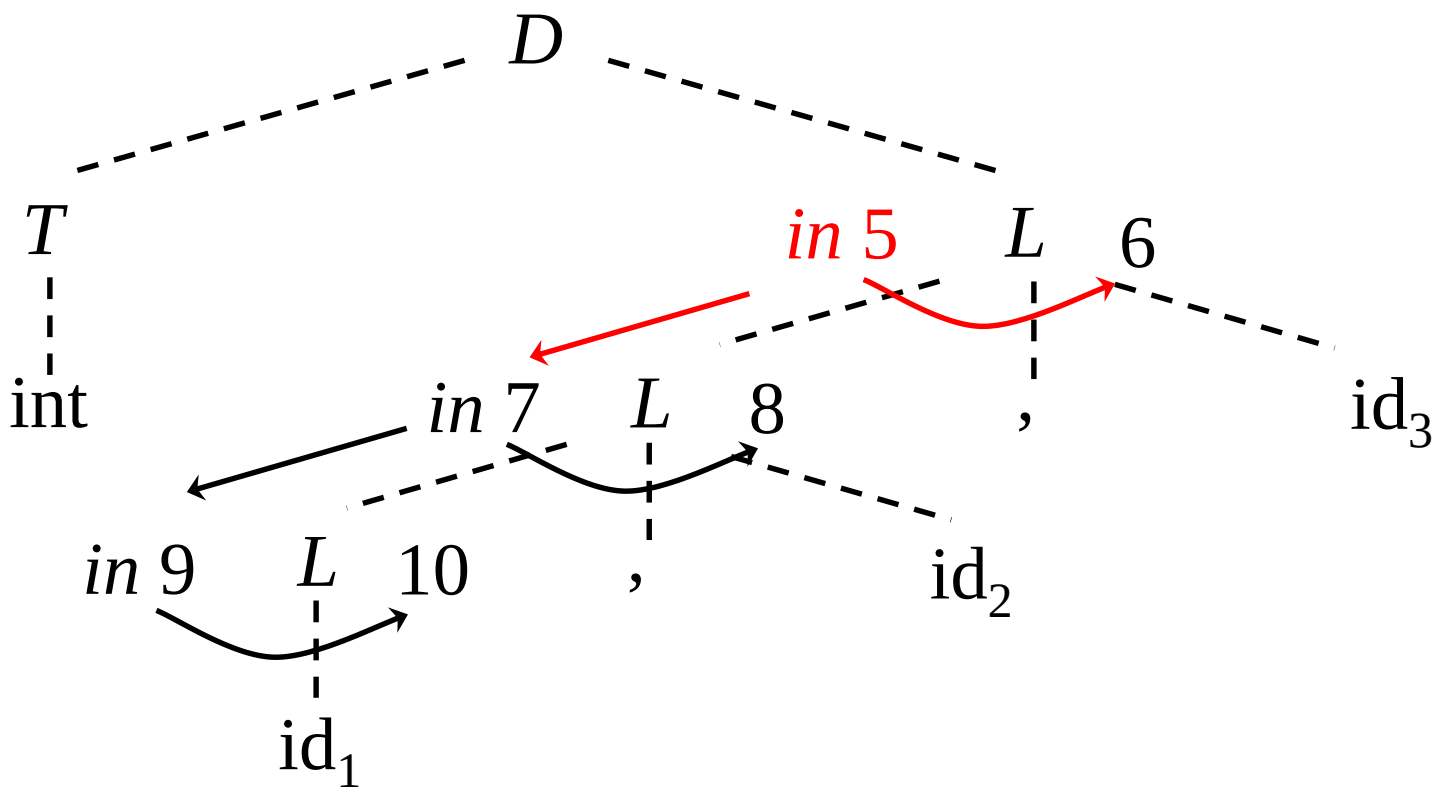
计算语义规则

例：1，2，3，4



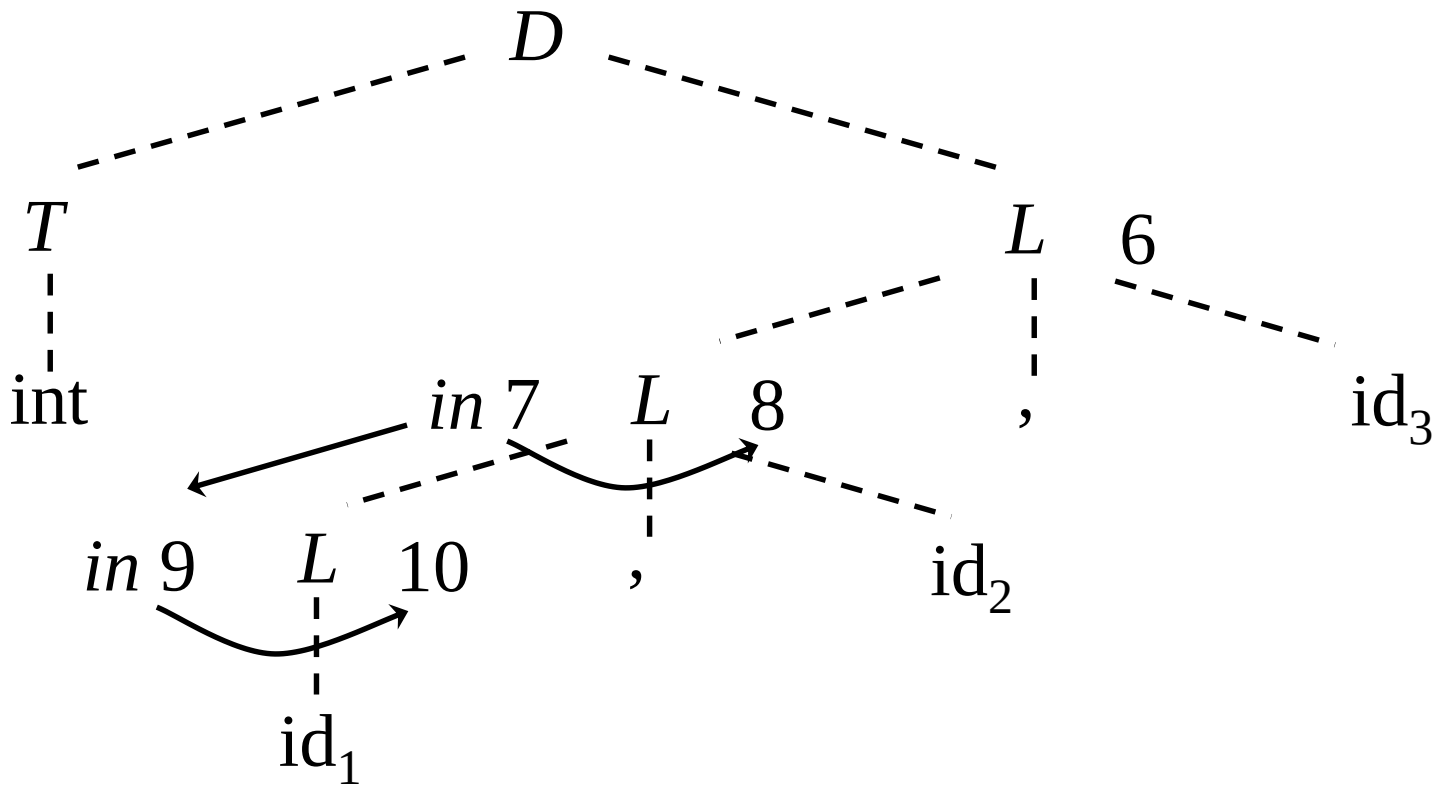
计算语义规则

例：1，2，3，4



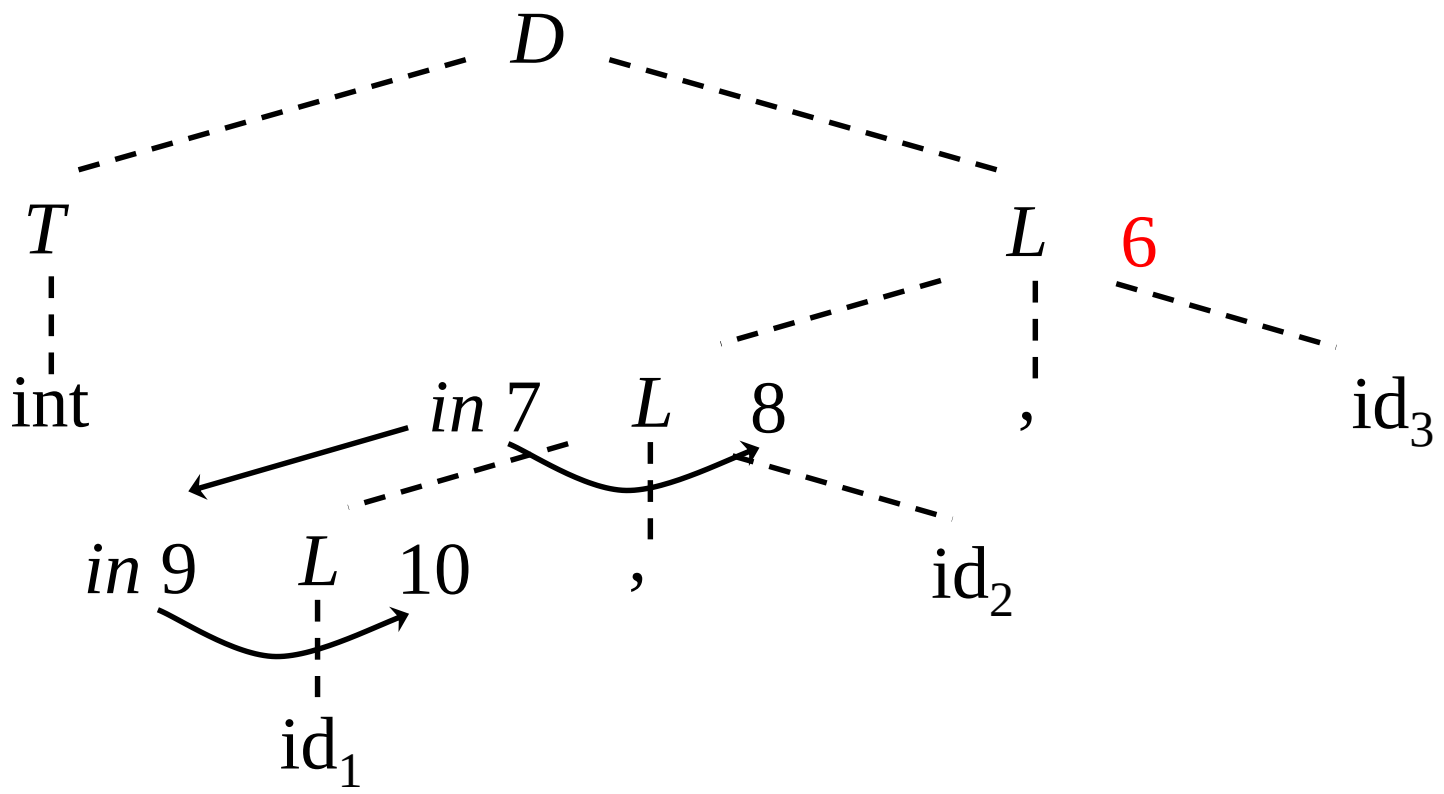
计算语义规则

例：1，2，3，4, 5



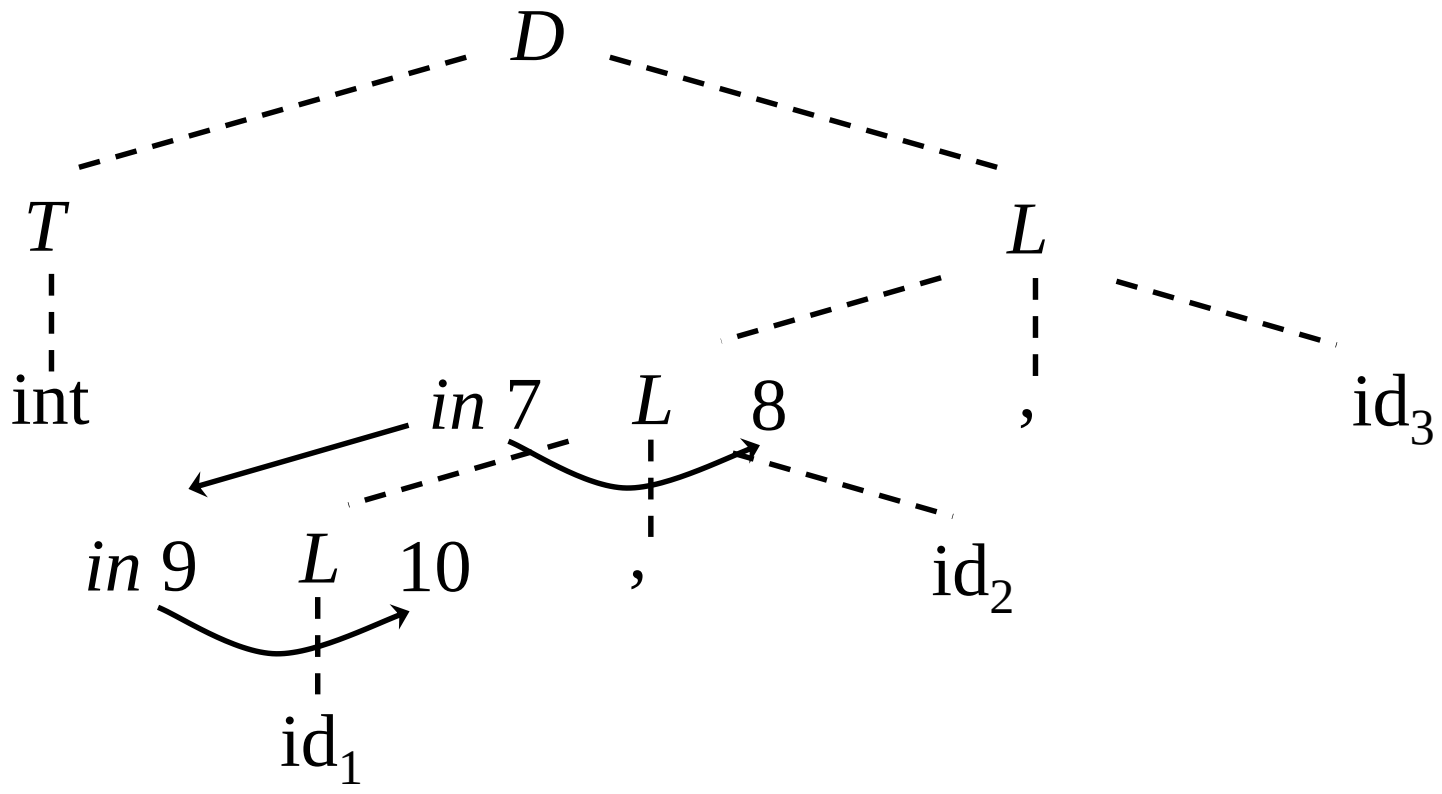
计算语义规则

例：1，2，3，4, 5



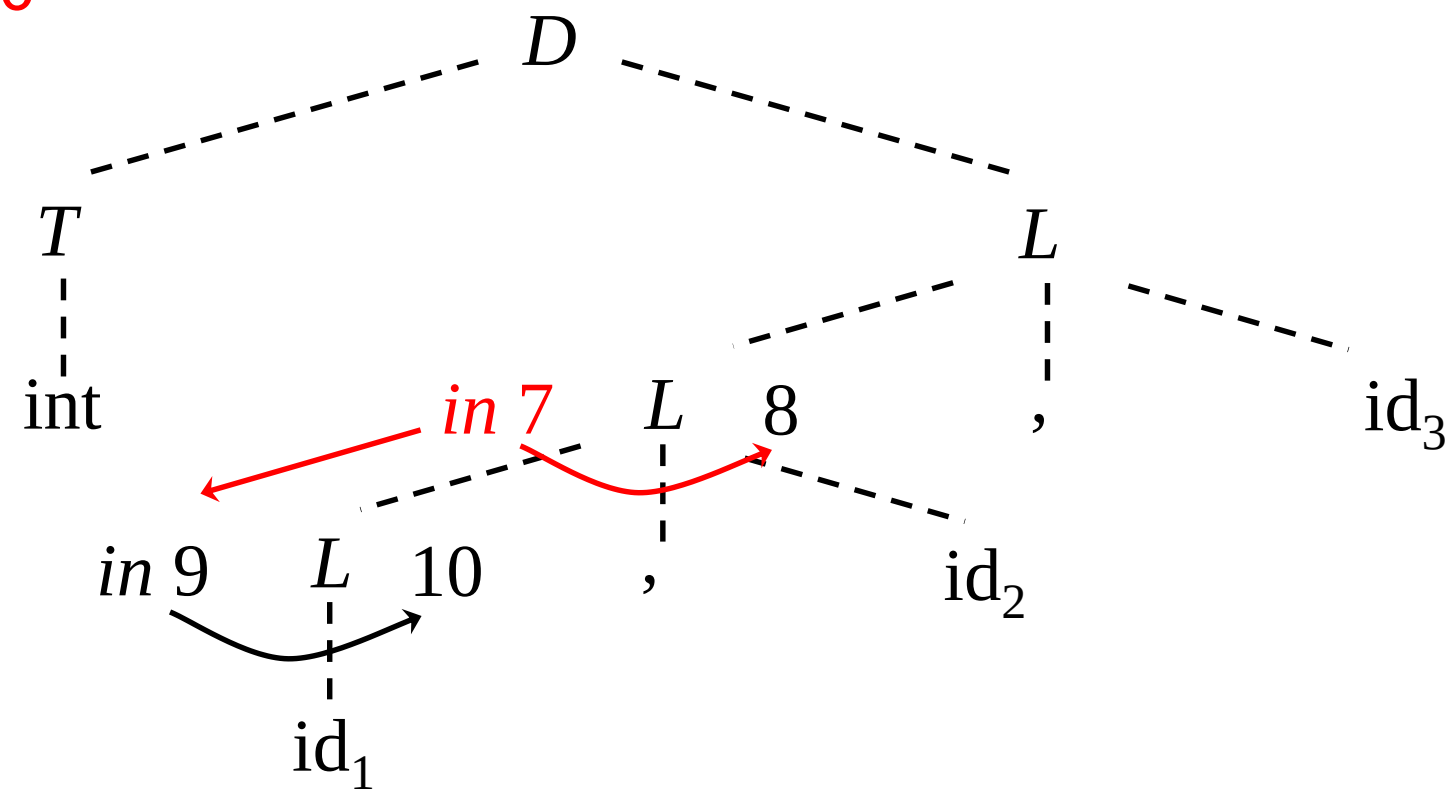
计算语义规则

例：1，2，3，4, 5, 6



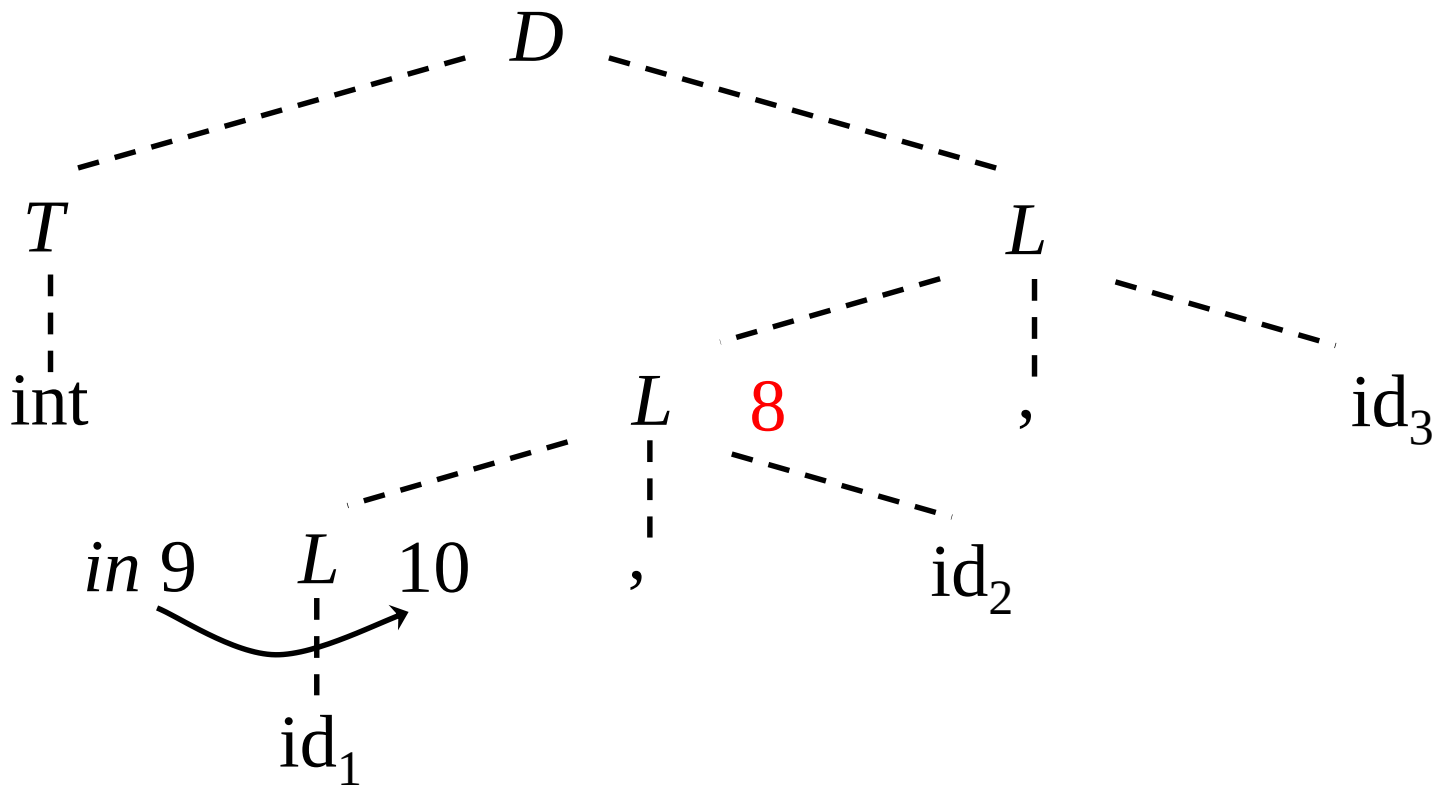
计算语义规则

例：1，2，3，4, 5, 6



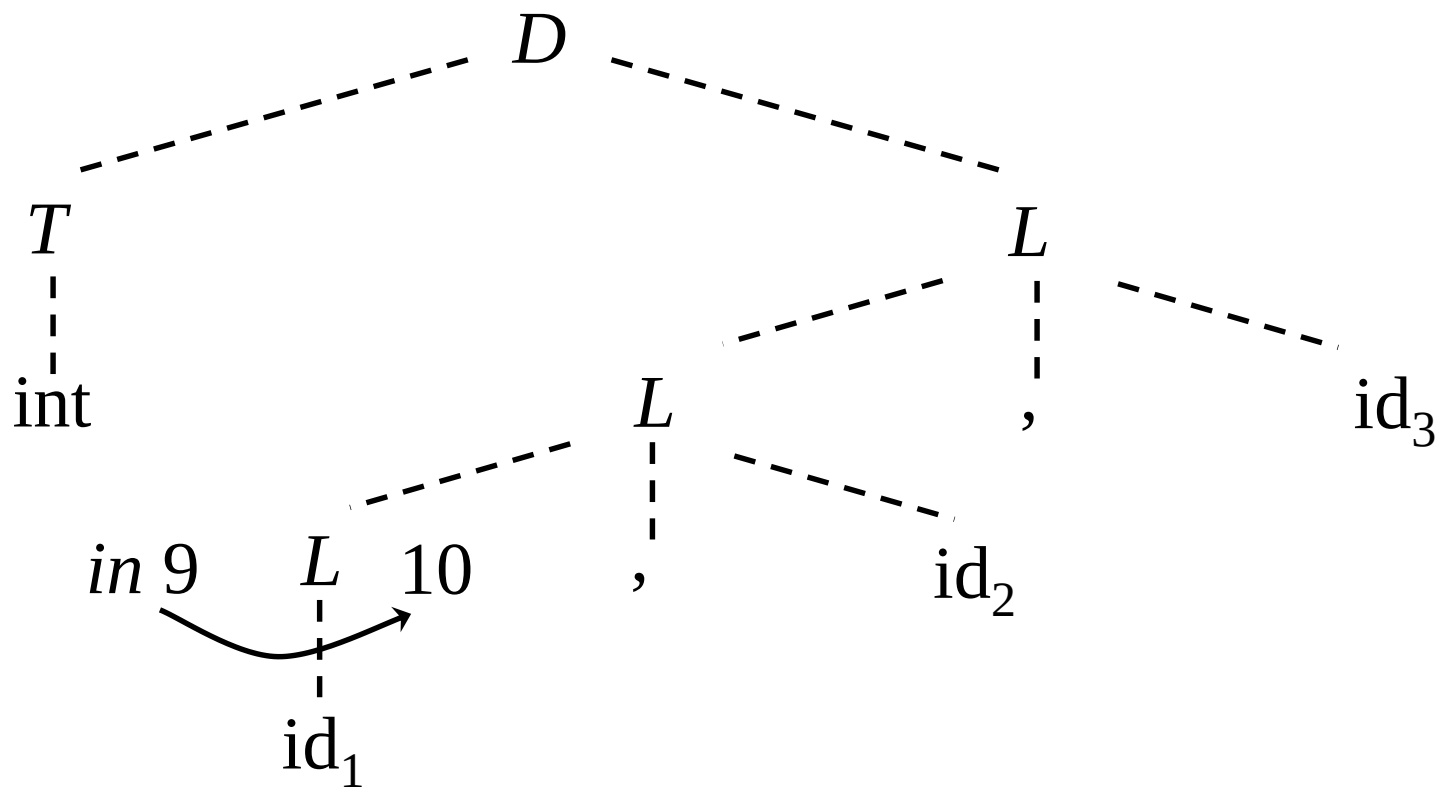
计算语义规则

例：1，2，3，4, 5, 6, 7



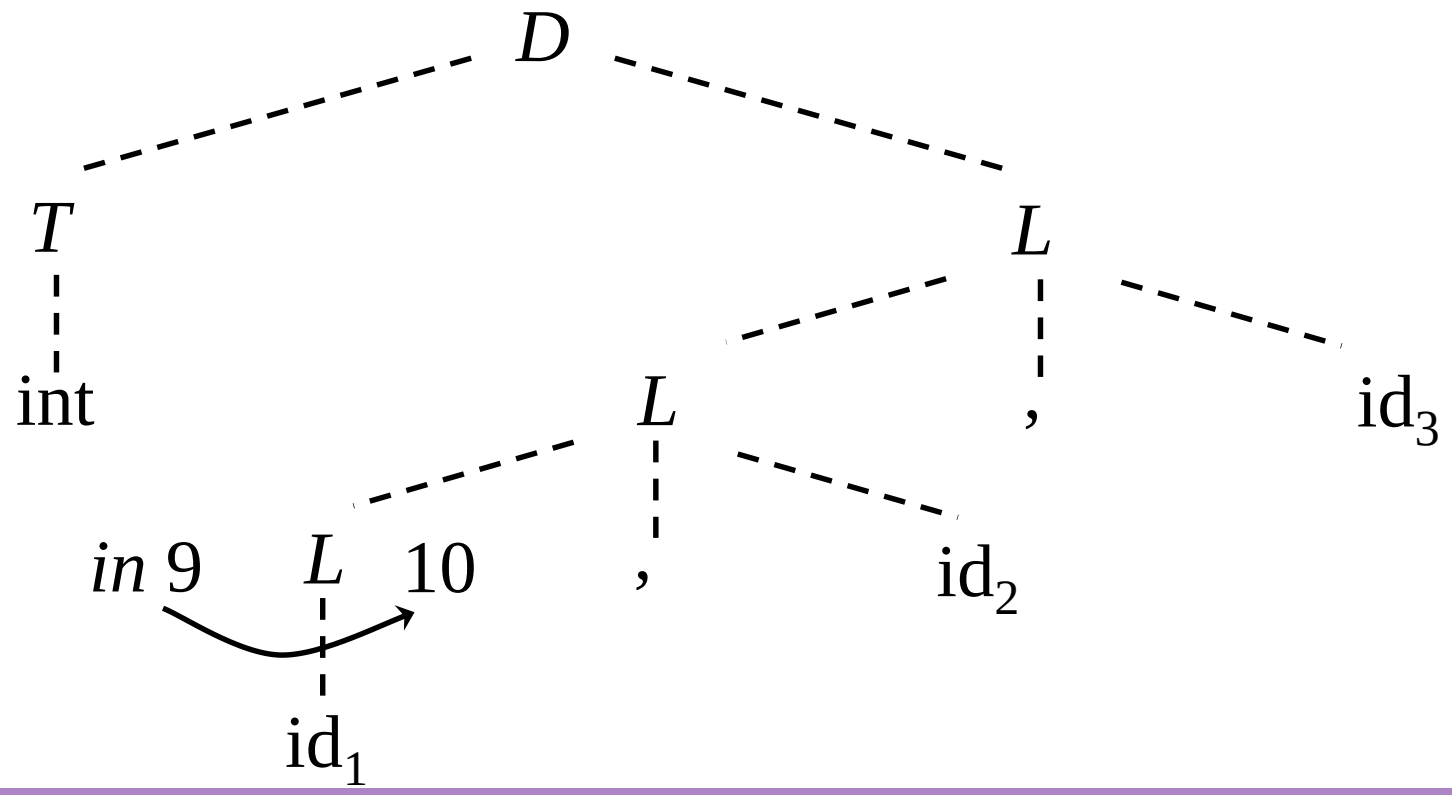
计算语义规则

例：1，2，3，4, 5, 6, 7, 8

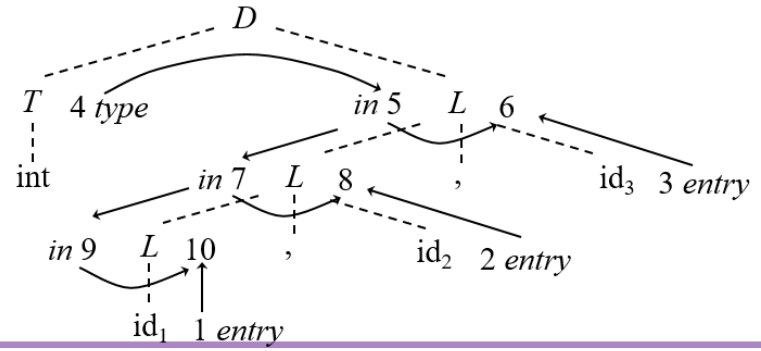


计算语义规则

例：1，2，3，4, 5, 6, 7, 8 , 9, 10

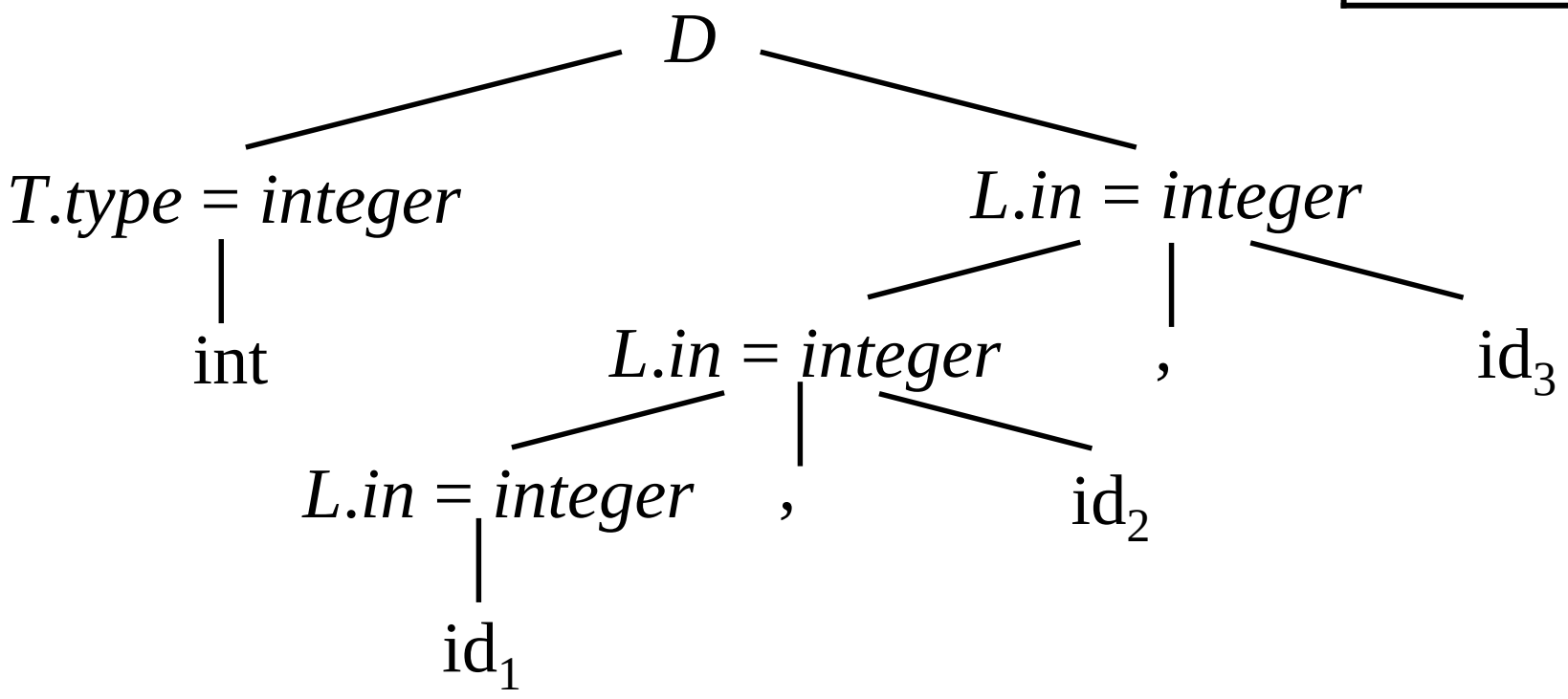


属性计算次序



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

int id_1, id_2, id_3的注释分析树



S属性与L属性

S属性定义：

只使用综合属性的语法制导定义。

利用S-属性定义进行语义分析时，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。

综合属性从下到上包括自身，其属性可从后代和自身的其它属性计算得到

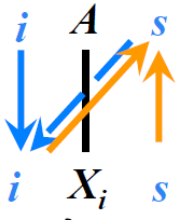
S属性与L属性

L属性的定义

限制：在一个产生式所关联的各个属性之间，依赖图的边总是从左到右，而不能从右到左

每个属性是综合属性或满足以下条件的继承属性

- 对于任意产生式规则$A \rightarrow X_1 \ldots X_n$，附加的计算$X_i.i$的语义规则只能使用：
  - A的继承属性
  - X_i左边的符号$X_1, X_2, \ldots, X_{i-1}$的属性
  - X_i的属性，并且由X_i的全部属性组成的依赖图中不存在环。



对综合属性没有限制。显然，所有的S属性定义均为L属性定义

例：L属性的实例

产生式	语义规则
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$

产生式	语义规则
$A \rightarrow B\ C$	$A.s = B.b$ $B.i = f(C.c, A.s)$

文法符号*B*的继承属性依赖于它右边文法符号*C*的属性。

语法制导定义来构造语法树

S属性定义——自底向上

L属性定义——自顶向下

抽象语法树的构造

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

抽象语法树的构造

结点 = 一个程序构造

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

指向运算对象节点2的指针

…

}

抽象语法树的构造

叶子结点：附加的域存放词法值。构造函数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

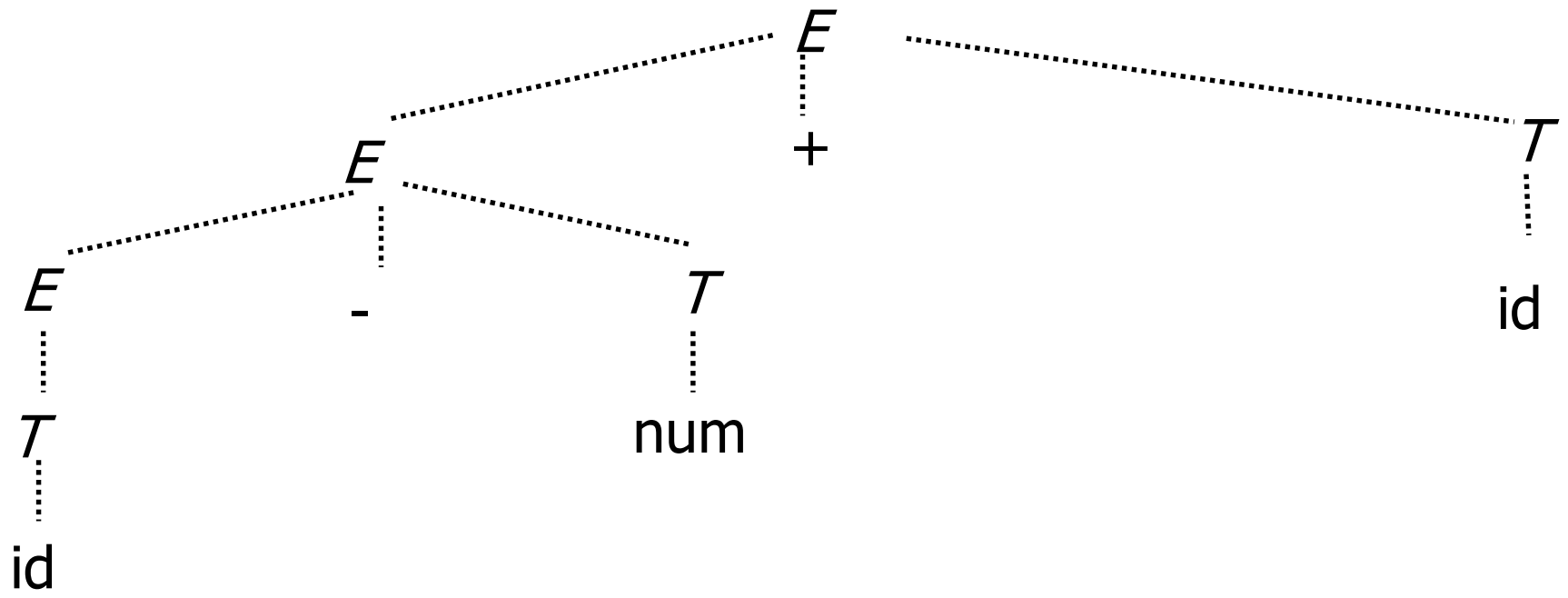
内部结点：附加字段个数=子结点个数。构造函数mknode(op，c1，c2,…,ck)：为运算符op创建语法树中节点，标记（运算符）为op，其余k个字段的值为c1,…,ck.

例：S属性定义 构造语法树

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

例：S属性定义 构造语法树

a-4+c的语法分析树



例：S属性定义 构造语法树

产生式	语义规则
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow (E)$	T.*nptr* = E.*nptr*
$T \rightarrow$ **id**	T.*nptr* = *mkleaf*(id, **id**.*lexval*)
$T \rightarrow$ **num**	T.*nptr* = *mkleaf*(num, **num**.*val*)

a-4+c的抽象语法树

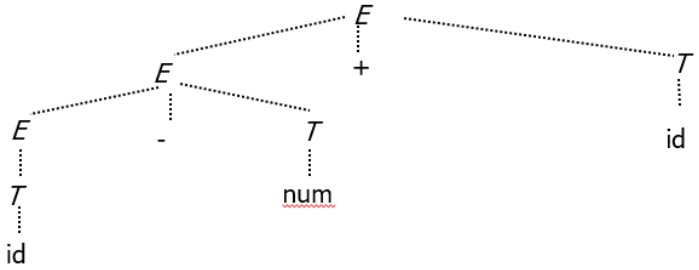
(1) p_1=mkleaf(**id**, entry_a);

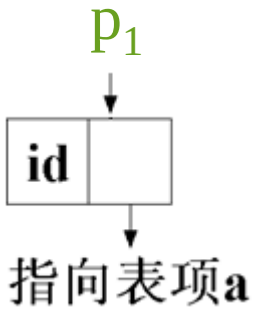
(2) p_2=mkleaf(**num**, 4);

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

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

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





例：S属性定义 构造语法树

产生式	语义规则
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow (E)$	T.*nptr* = E.*nptr*
$T \rightarrow$ **id**	T.*nptr* = *mkleaf*(id, **id**.*lexval*)
$T \rightarrow$ **num**	T.*nptr* = *mkleaf*(num, **num**.*val*)

a-4+c的抽象语法树

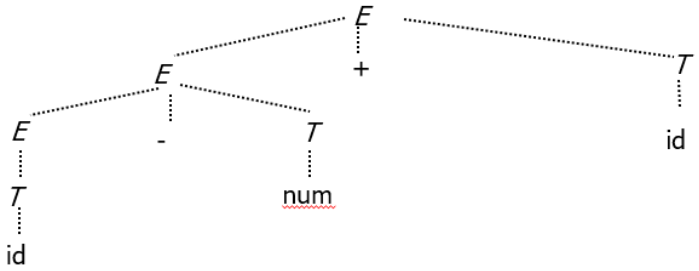
(1) p_1=mkleaf(**id**, entry_a);

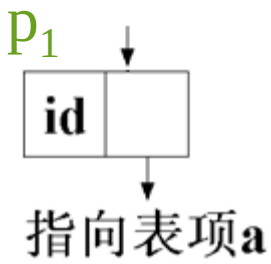
(2) p_2=mkleaf(**num**, 4);

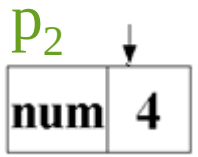
(3) p_3=mknode('-', p_1, p_2);

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

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







例：S属性定义 构造语法树

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

a-4+c的抽象语法树

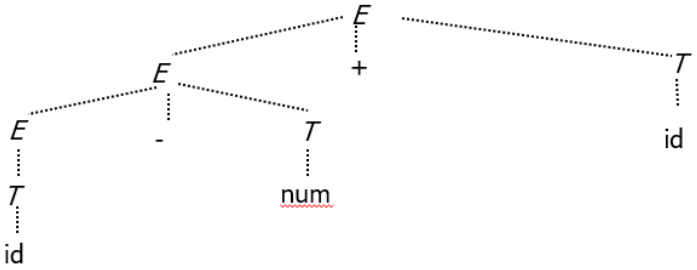
(1) p_1=mkleaf(**id**, entry_a);

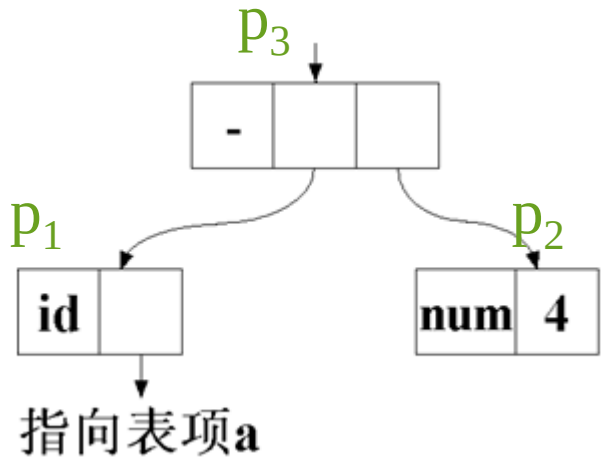
(2) p_2=mkleaf(**num**, 4);

(3) p_3=mknode(‘-’, p_1, p_2);

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

(5) p_5=mknode(‘+’, p_3, p_4);





例：S属性定义 构造语法树

产生式	语义规则
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow (E)$	T.*nptr* = E.*nptr*
$T \rightarrow$ **id**	T.*nptr* = *mkleaf*(id, **id**.*lexval*)
$T \rightarrow$ **num**	T.*nptr* = *mkleaf*(num, **num**.*val*)

a-4+c的抽象语法树

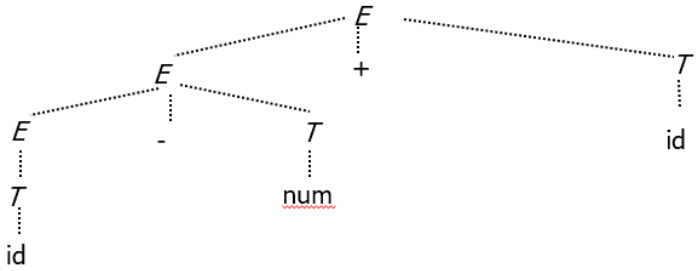
(1) p_1=mkleaf(**id**, entry_a);

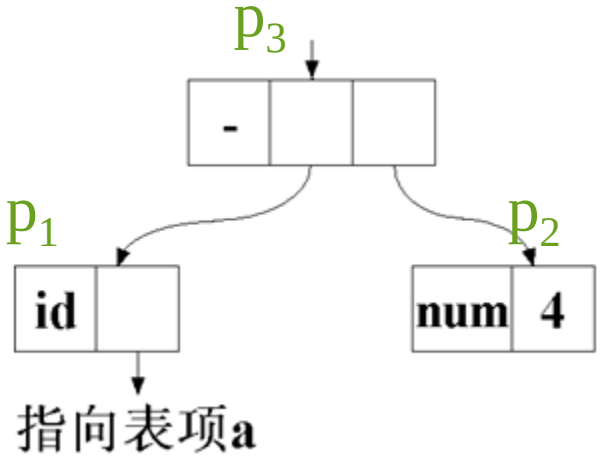
(2) p_2=mkleaf(**num**, 4);

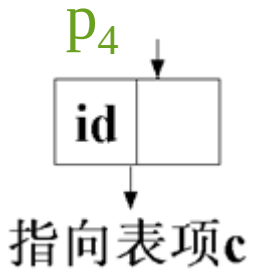
(3) p_3=mknode('-', p_1, p_2);

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

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







例：S属性定义 构造语法树

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

a-4+c的抽象语法树

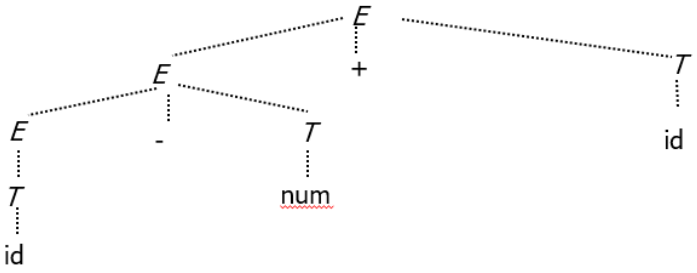
(1) p_1=mkleaf(**id**, entry_a);

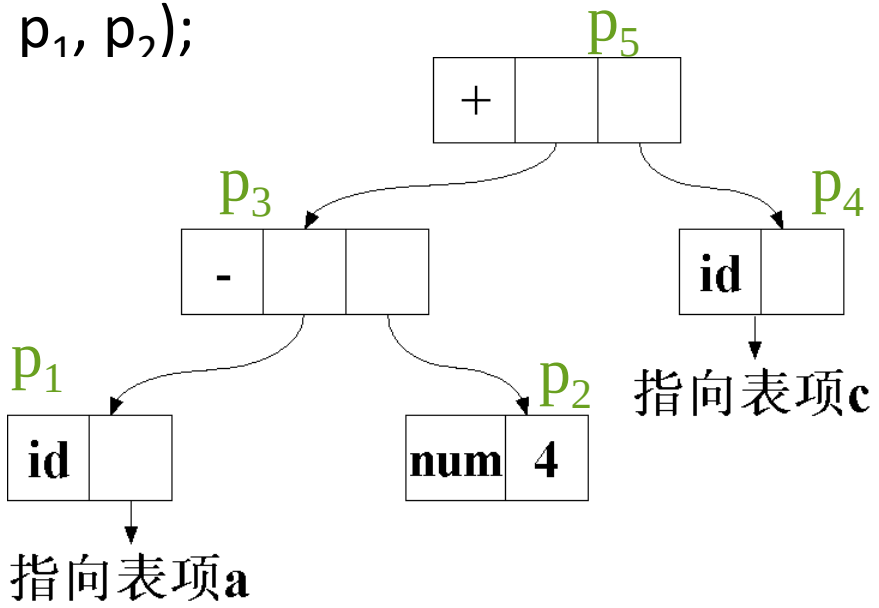
(2) p_2=mkleaf(**num**, 4);

(3) p_3=mknode(‘-’, p_1, p_2);

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

(5) p_5=mknode(‘+’, p_3, p_4);

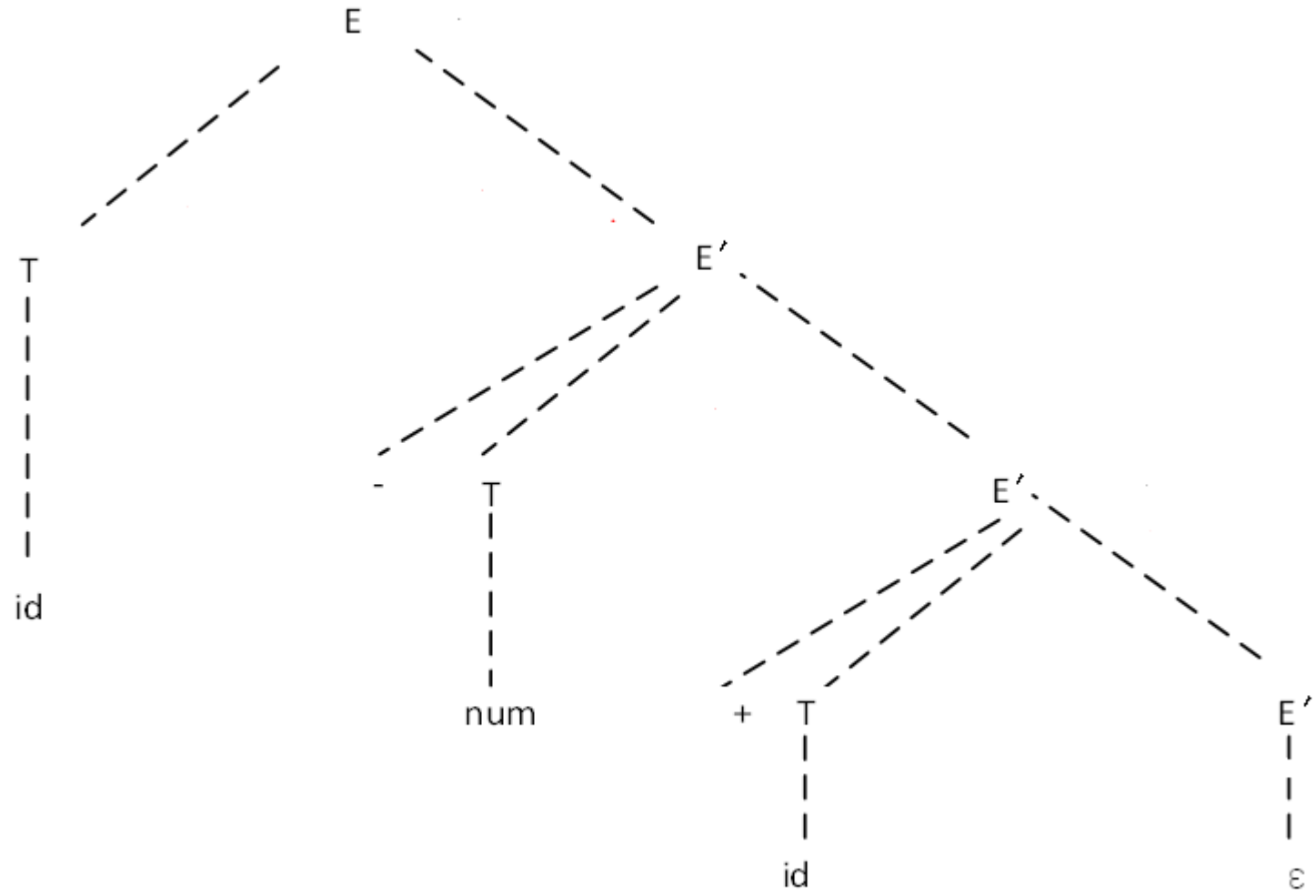




例：L属性定义 构造语法树

产生式	语义规则
$E \rightarrow T\ E'$	
$E' \rightarrow +\ T\ E_1'$	
$E' \rightarrow -\ T\ E_1'$	
$E' \rightarrow\ \varepsilon$	
$T \rightarrow (E)$	
$T \rightarrow \mathbf{id}$	
$T \rightarrow \mathbf{num}$	

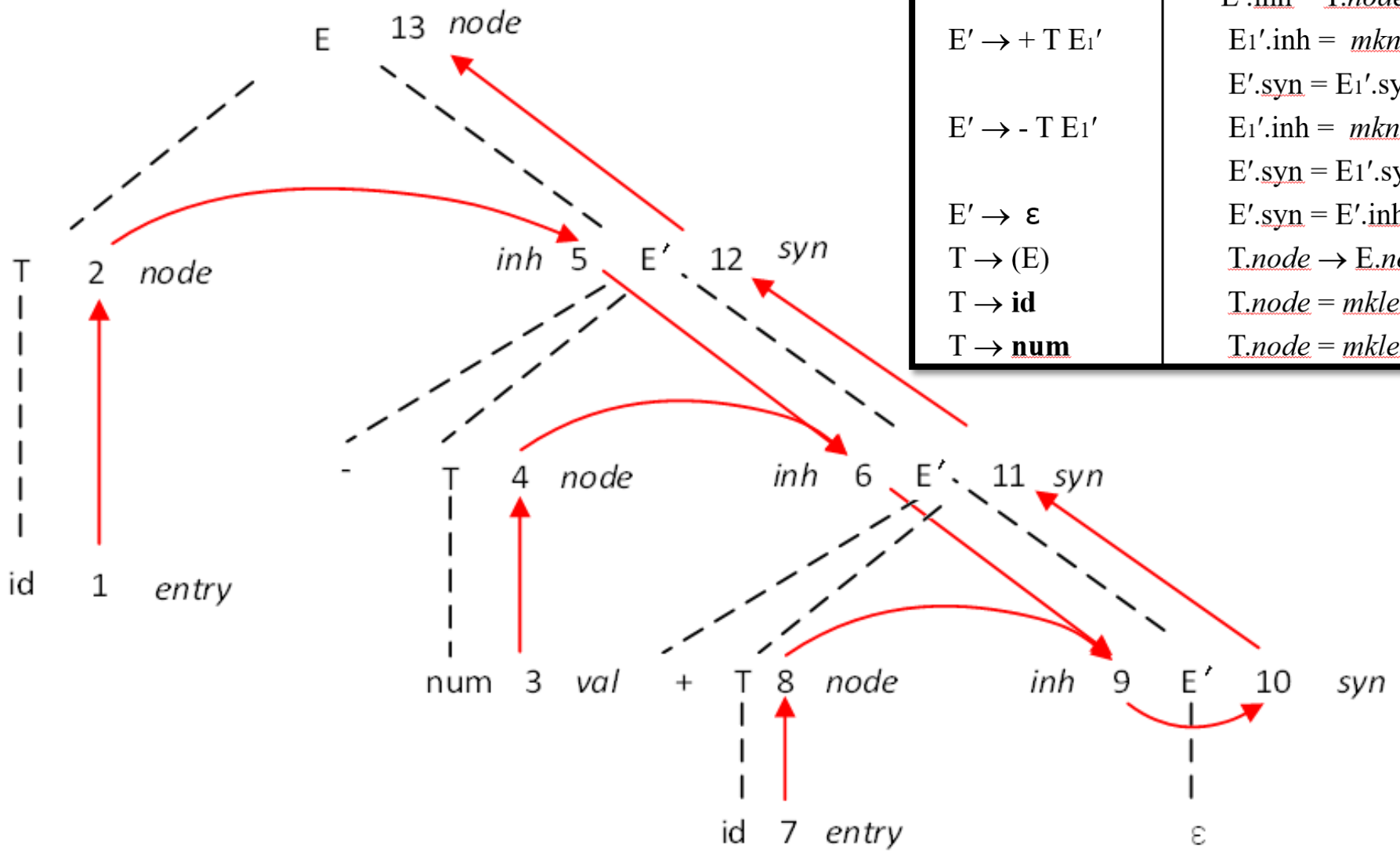
a - 4 + c的分析树



例：L属性定义 构造语法树

产生式	语义规则
$E \rightarrow T\ E'$	E.*node* = E′.*syn*
	E′.inh = T.*node*
$E' \rightarrow +\ T\ E_1'$	E_1'.inh = *mknode*(“+”, E′.*inh*, T.*node*)
	E′.syn = E_1'.syn
$E' \rightarrow -\ T\ E_1'$	E_1'.inh = *mknode*(“-”, E′.inh, T.*node*)
	E′.syn = E_1'.syn
$E' \rightarrow \varepsilon$	E′.syn = E′.inh
$T \rightarrow (E)$	T.*node* → E.*node*
$T \rightarrow$ **id**	T.*node* = *mkleaf*(id, **id**.*lexval*)
$T \rightarrow$ **num**	T.*node* = *mkleaf*(num, **num**.*val*)

a - 4 + c的依赖图



产生式	语义规则
E → T E′	E.*node* = E′.*syn* E′.inh = T.*node*
E′ → + T E₁′	E₁′.inh = *mknode*(“+”, E′.*inh*, T.*node*) E′.syn = E₁′.syn
E′ → - T E₁′	E₁′.inh = *mknode*(“-”, E′.inh, T.*node*) E′.syn = E₁′.syn
E′ → ε	E′.syn = E′.inh
T → (E)	T.*node* → E.*node*
T → **id**	T.*node* = *mkleaf*(id, **id**.*lexval*)
T → **num**	T.*node* = *mkleaf*(num, **num**.*val*)

语法制导的翻译方案

语法制导的翻译方案（Syntax-Directed Translation Scheme，SDT）是SDD的实现。

SDT是拓广的CFG，在文法中嵌入语义动作，语义动作写在花括号“{}”里，出现在产生式右部适当位置。

当归约出产生式右部的某个非终结符号后，就执行紧接在该非终结符号右边的语义动作。

SDD vs SDT

语义规则

产 生 式	语 义 规 则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

语义动作

$E \rightarrow E_1 + T \{E.val = E_1.val + T.val\}$

语法制导的翻译方案

任何SDT可以通过先建立语法分析树，然后前序遍历执行语义动作。

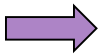
SDT能够在语法分析的同时进行，无需构造好分析树。这里讨论两类：

1. 基础文法是LR，SDD是S-attributed；
2. 基础文法是LL，SDD是L-attributed。

将S-SDD转换为SDT

将一个**S-SDD**转换为**SDT**的方法：将每个语义动作都放在产生式的最后

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



$L \rightarrow E$ n { *print* (*E.val*); }

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$;}

$E \rightarrow T$ { $E.val = T.val$;}

$T \rightarrow T_1 * F$ { $T.val = T_1.val \times F.val$;}

$T \rightarrow F$ { $T.val = F.val$; }

$F \rightarrow (E)$ { $F.val = E.val$; }

$F \rightarrow$ digit { F.val = digit.*lexval*; }

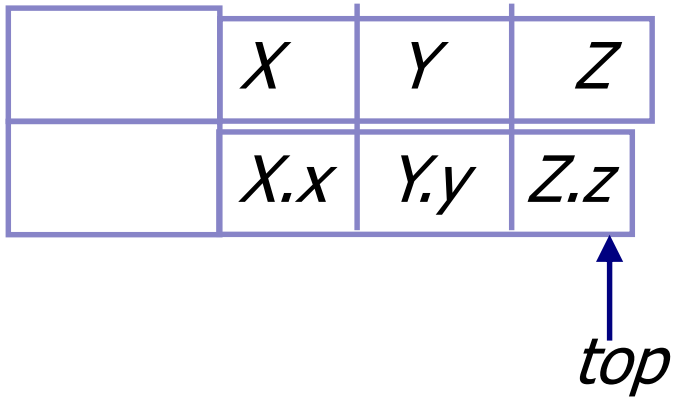
S属性定义的SDT 实现

如果一个**S-SDD**的基本文法可以使用**LR**分析技术，那么它的**SDT**可以在**LR**语法分析过程中实现

S-SDD的SDT的语法分析实现

$A \rightarrow X\ Y\ Z$

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



文法符号

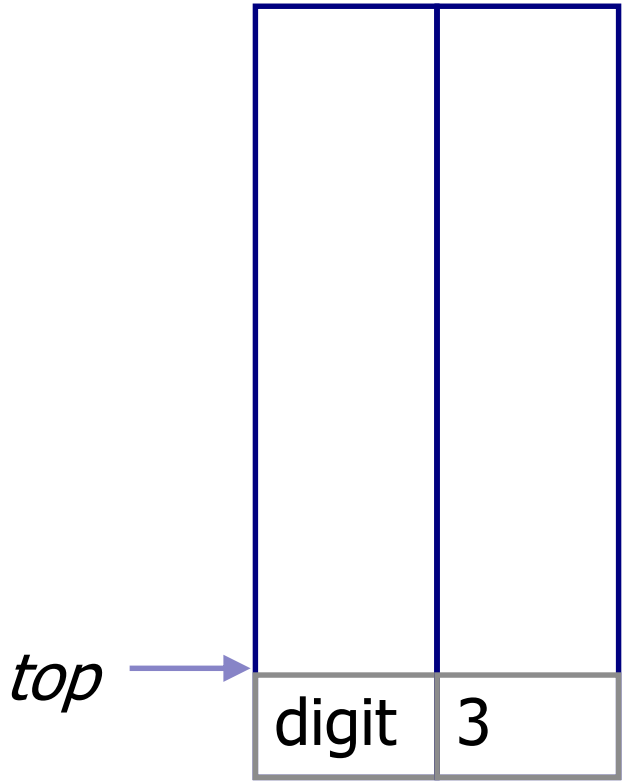
综合属性

stack[top-2].symb =A

stack[top-2].val = *f*(stack[top-2].val, stack[top-1].val, stack[top].val)

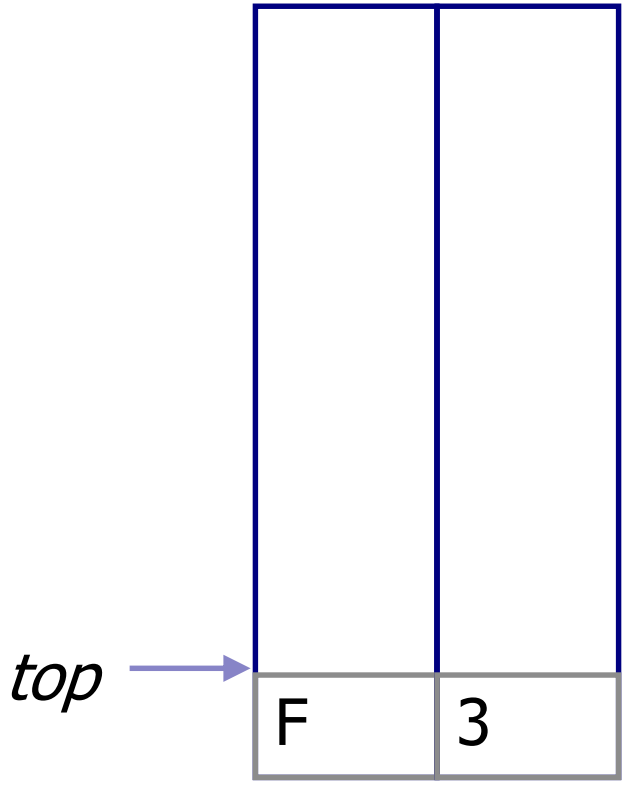
top =top-2;

3*5 n的分析计算过程



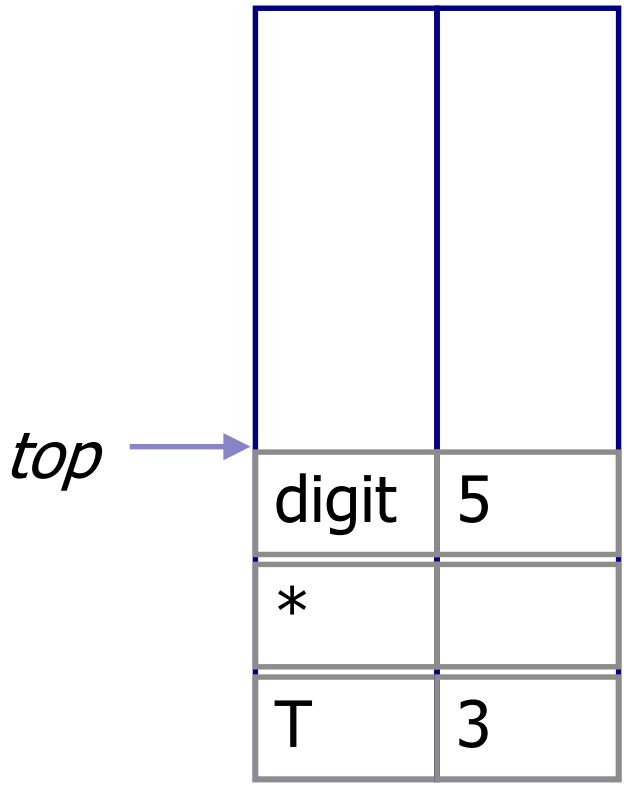
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*); *top* = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*; *top* = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*; *top* = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ; *top* = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



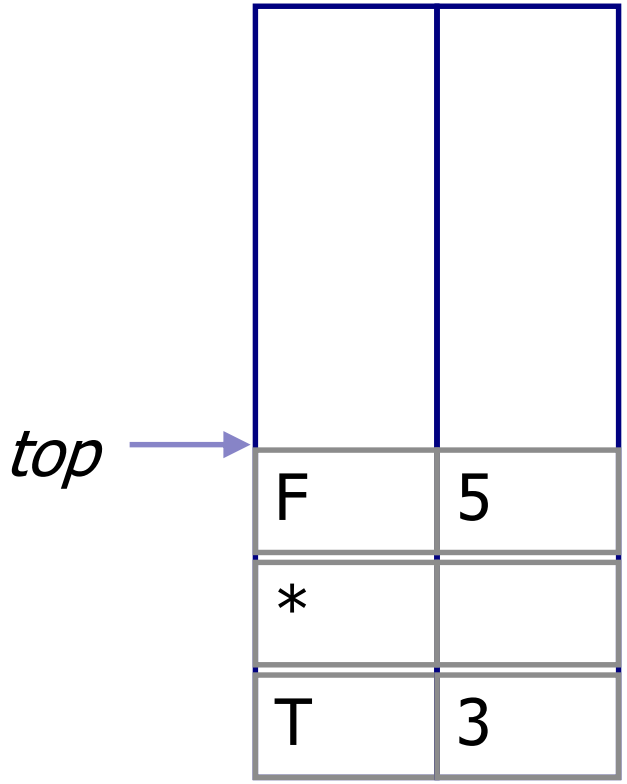
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*); *top* = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*; *top* = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*; *top* = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ; *top* = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



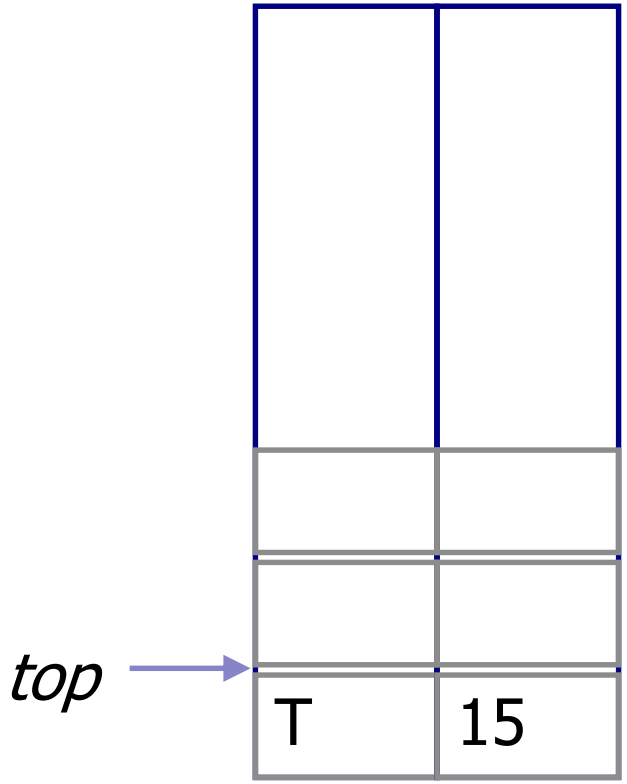
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*);
	top = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*;
	top = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*;
	top = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ;
	top = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



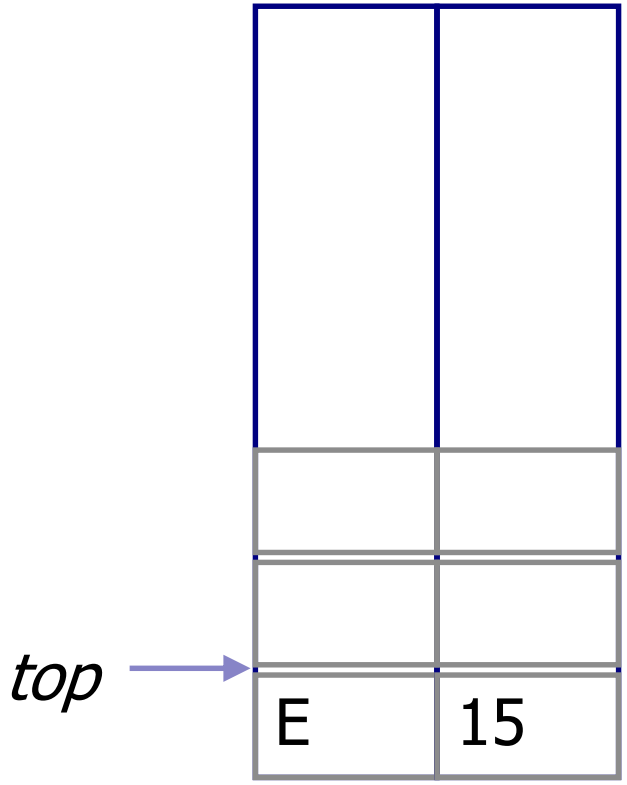
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*);
	top = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*;
	top = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*;
	top = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ;
	top = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



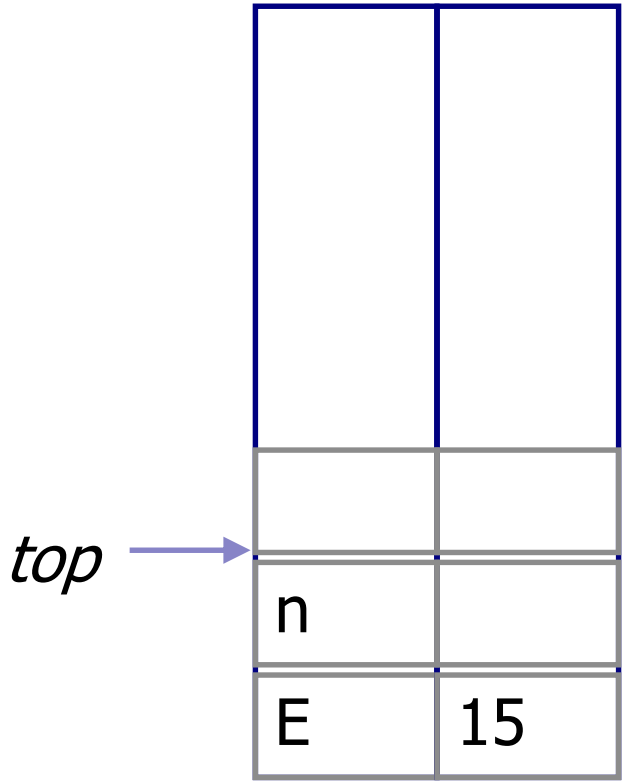
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*);
	top = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*;
	top = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*;
	top = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ;
	top = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



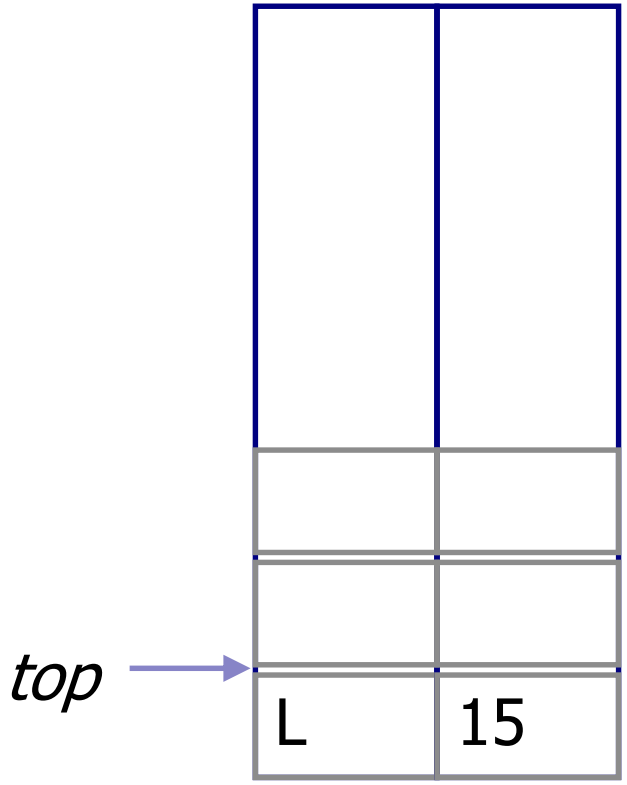
$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*); *top* = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*; *top* = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*; *top* = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ; *top* = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*); *top* = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*; *top* = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*; *top* = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ; *top* = *top* -2;}
$F \rightarrow$ digit	

3*5 n的分析计算过程



$L \rightarrow E$ n	{ *print* (*stack*[*top*-1].*val*);
	top = *top* -1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* + *stack*[*top*].*val*;
	top = *top* -2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2] .*val* × *stack*[*top*].*val*;
	top = *top* -2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1] .*val* ;
	top = *top* -2;}
$F \rightarrow$ digit	

产生式内部带语义动作的SDT

$B \rightarrow X\ \{a\}\ Y$

- LR：归约出X之后立即执行动作$\{a\}$
- LL：试图展开Y之前执行动作$\{a\}$

例，中缀到后缀转换的SDT

$L \rightarrow E$ **n**

$E \rightarrow E_1 + T$ {*print*(‘+’)}

$E \rightarrow T$

$T \rightarrow T_1 * F$ {*print*(‘*’)}

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow$ **digit** {*print*(**digit**.*lexval*)}

中缀到前缀转换的SDT

$L \rightarrow E$ **n**

$E \rightarrow \{print(\text{‘+’})\}\ E_1 + T$

$E \rightarrow T$

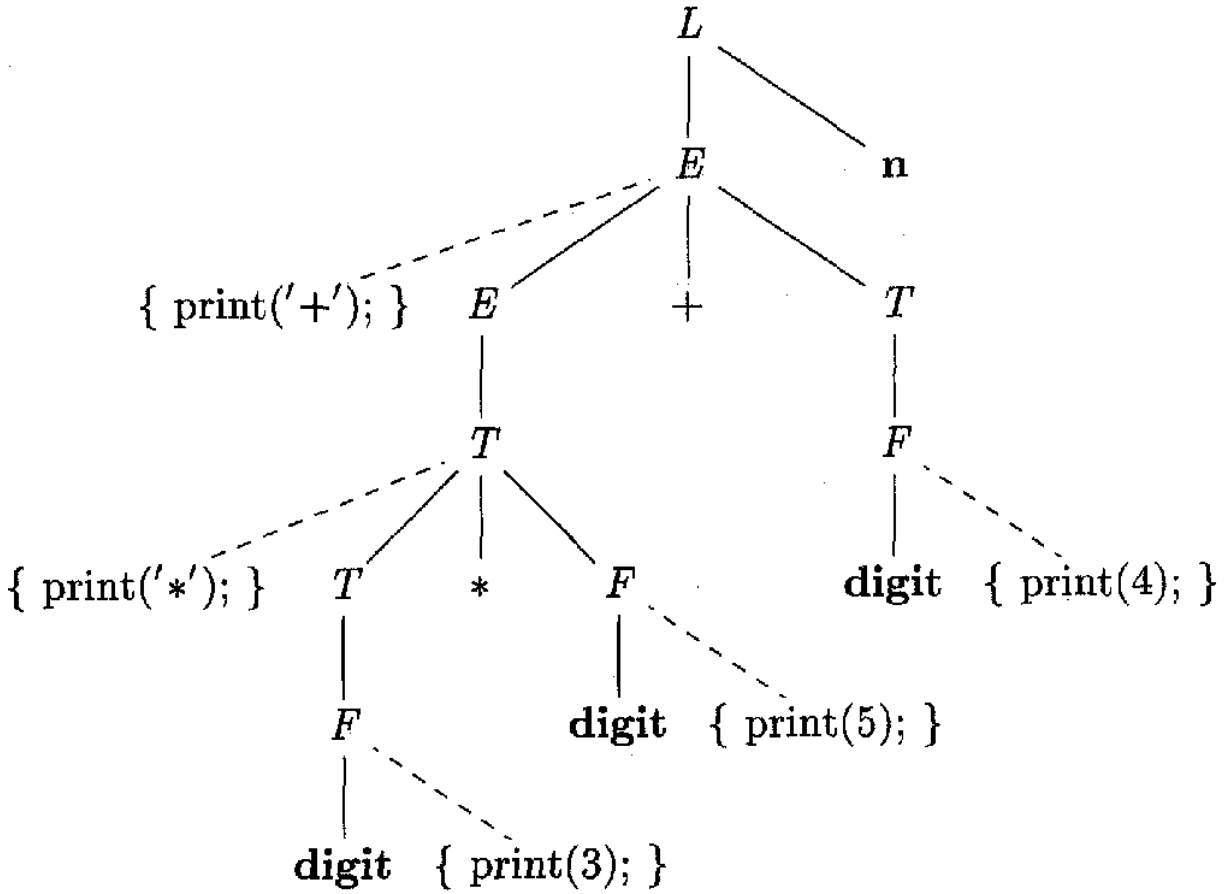
$T \rightarrow \{print(\text{‘*’})\}\ T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow$ **digit** $\{print(\textbf{digit}.lexval)\}$

嵌入动作的分析树：3*5+4



从SDT中删除左递归

带左递归的文法的翻译模式例子

$E \rightarrow E_1 + T \quad \{ print(\text{‘+’}); \}$

$E \rightarrow T$

转换为：

$E \rightarrow TR$

$R \rightarrow + T \{ print(\text{‘+’}); \} R$

$R \rightarrow \varepsilon$

例：一个一般化的例子

$A \rightarrow A_1 Y \qquad \{ A.a = g(A_1.a,\ Y.y) \}$

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

消除左递归后：

$A \rightarrow X R$

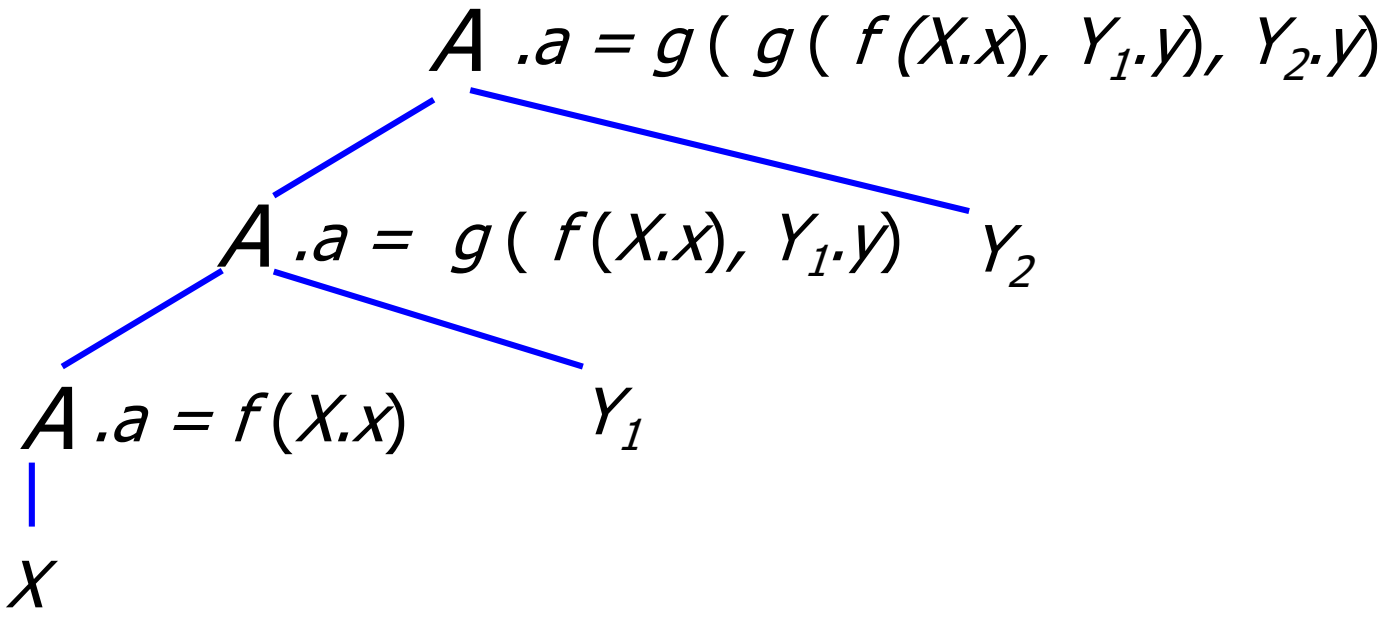
$R \rightarrow Y R_1$

$R \rightarrow \varepsilon$

消除一个后缀SDT中的左递归

$A \rightarrow A_1 Y$ $\quad$ $\{ A.a = g(A_1.a, Y.y) \}$

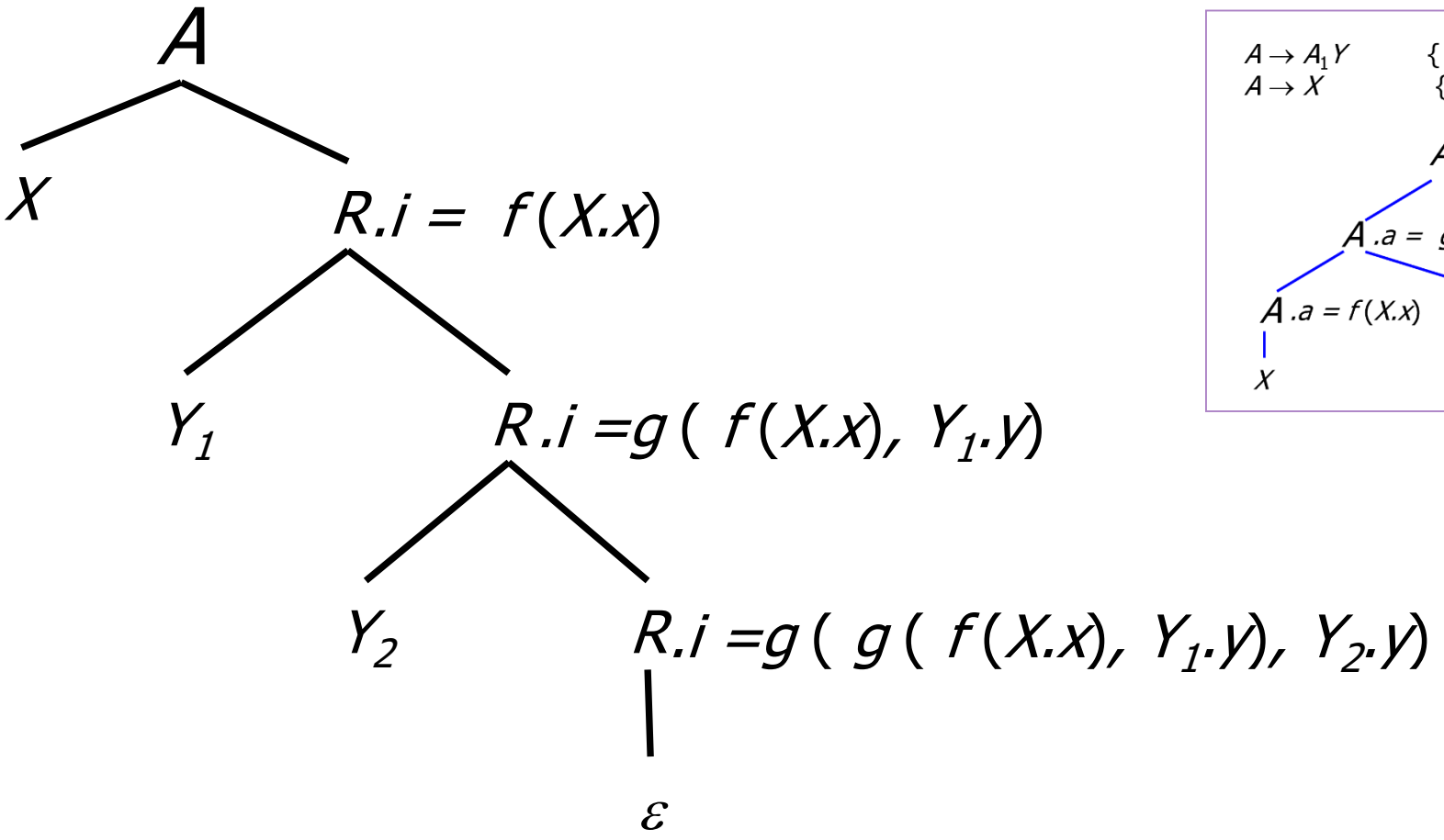
$A \rightarrow X$ $\quad$ $\{ A.a = f(X.x) \}$

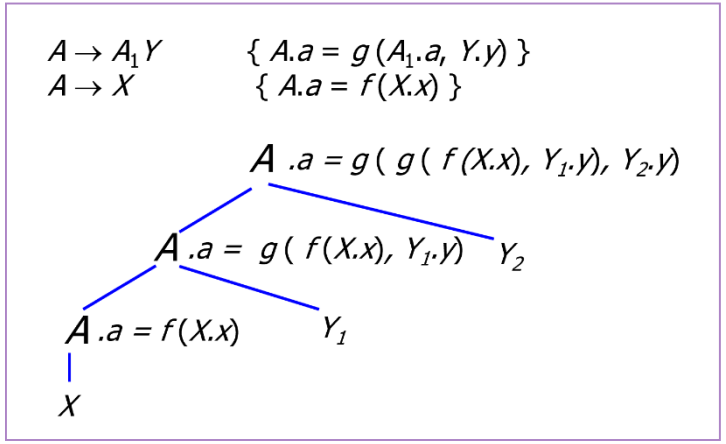


$A \rightarrow X \ \{ R.i = f(X.x) \} \ R \ \{ A.a = R.s \}$

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

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





L属性定义的SDT

如果一个L-SDD的基本文法可以使用LL分析技术

1) $T \rightarrow FT'$
2) $T' \rightarrow *FT_1'$
3) $T' \rightarrow \varepsilon$
4) $F \rightarrow digit$

- 那么它的SDT可以在LL或LR语法分析过程中实现

- 在非递归的预测分析过程中进行语义翻译
- 在递归的预测分析过程中进行语义翻译
- 在LR分析过程中进行语义翻译

L属性定义的SDT-1

自顶向下的方式

将动作附加到语法分析树，前序遍历执行动作

重点：

确定语义动作的执行时机

如何将语义动作嵌入产生式右部的适当位置？

L属性的SDD转换为一个SDT的规则

1. 只有综合属性的情况：将计算综合属性的语义规则作为产生式右边末尾的动作。例如：

产生式	语义规则
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$

产生式	语义动作
$T \rightarrow T_1 * F$	$\{T.val = T_1.val * F.val\}$

L属性的SDD转换为一个SDT的规则

2. 同时存在综合属性和继承属性，建立翻译模式的必要条件：

- 产生式右部符号的继承属性必须在这个符号以前的动作中计算出来；
- 一个动作不能引用该动作右边符号的综合属性；
- 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可以放在产生式右端的末尾。

L属性的SDD转换为一个SDT的规则

例：翻译模式

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

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

不符合条件(1)。

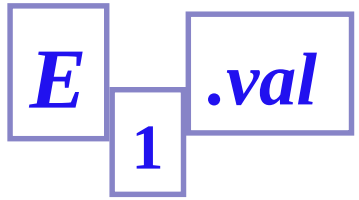
若改写成

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

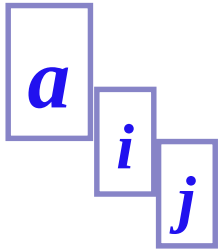
$A \rightarrow a$ $\{print(A.in)\}$

则就符合条件(1)。

例：数学排版语言的翻译模式片断

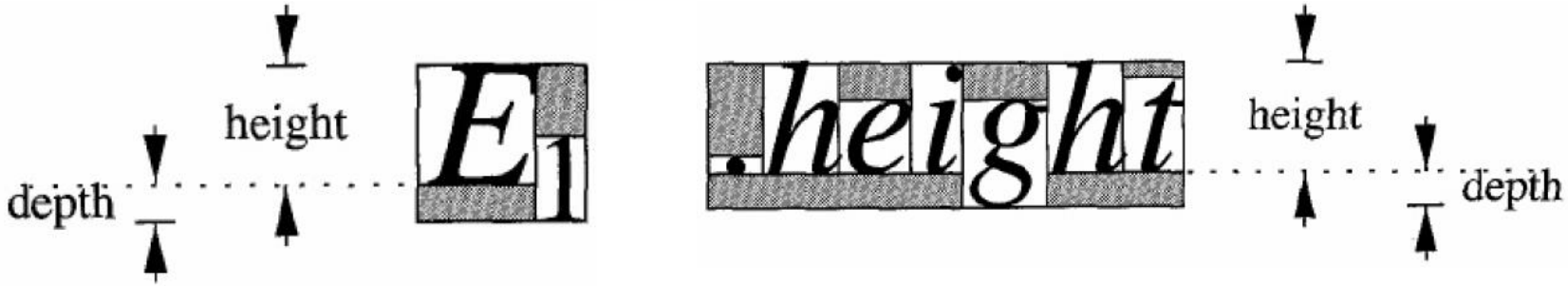


上图可以用命令：E **sub** 1.val 描述。



上图可以用命令：a **sub** i **sub** j 描述。

例：数学排版语言的翻译模式。



- 综合属性：

  ht：描述*height*

  dp：描述*depth*

- 继承属性*ps*：*point size*，设定字体大小。

  Assume *ps* = 10。

  下标的大小按比例缩小，位置下降。

方框排版的语法制导定义

产生式	语义规则
$S \rightarrow B$	$B.ps = 10$
$B \rightarrow B_1 B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = max(B_1.ht, B_2.ht)$ $B.dp = max(B_1.dp, B_2.dp)$
$B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 * B.ps$ $B.ht = max(B_1.ht, B_2.ht - 0.25 * B.ps)$ $B.dp = max(B_1.dp, B_2.dp + 0.25 * B.ps)$
$B \rightarrow (B_1)$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
$B \rightarrow$ **text**	$B.ht = getHt(B.ps$, **text**$.lexval)$ $B.dp = getDp(B.ps$, **text**$.lexval)$

方框排版的SDT

S	→		{ B.ps = 10; }
		B	
B	→		{ B_1.ps = B.ps; }
		B_1	{ B_2.ps = B.ps; }
		B_2	{ B.ht = max(B_1.ht, B_2.ht);
			B.dp = max(B_1.dp, B_2.dp); }
B	→		{ B_1.ps = B.ps; }
		B_1 **sub**	{ B_2.ps = B.ps × 70%; }
		B_2	{ B.ht = max(B_1.ht, B_2.ht – B.ps × 25%);
			B.dp = max(B_1.dp, B_2.dp + B.ps × 25%); }
B	→	**(**	{ B_1.ps = B.ps; }
		B_1 **)**	{ B.ht = B_1.ht;
			B.dp = B_1.dp; }
B	→	**text**	{ B.ht = getHight(B.ps, **text**.lexval);
			B.dp = getDepth(B.ps, **text**.lexval); }

在非递归的预测分析过程中SDT

非递归的自顶向下语法分析过程

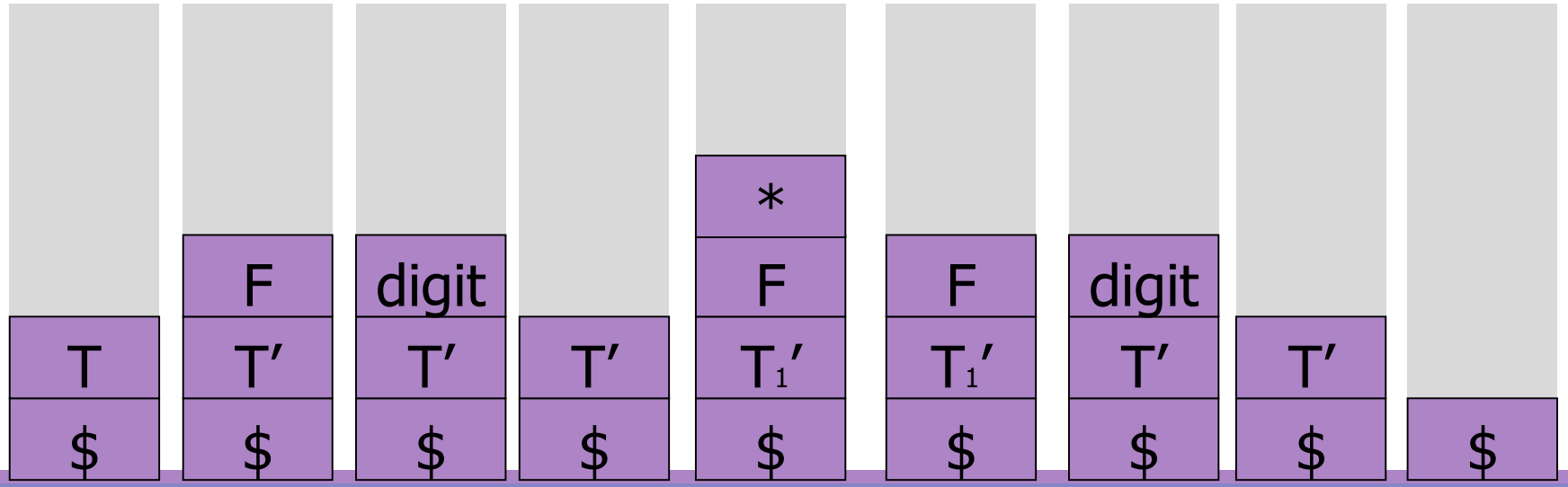
1) T→ FT′

2) T′→ *FT$_1$′

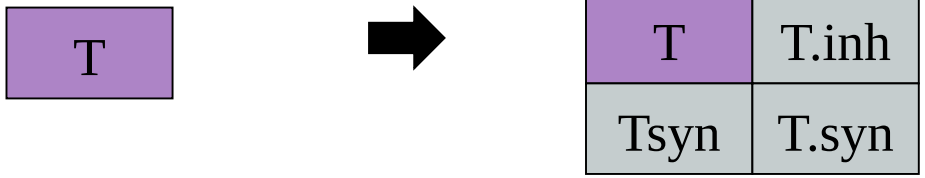
3) T′→ ε

4) F → digit

3*5$



在非递归的预测分析过程中SDT



在非递归的预测分析过程中SDT

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$
2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

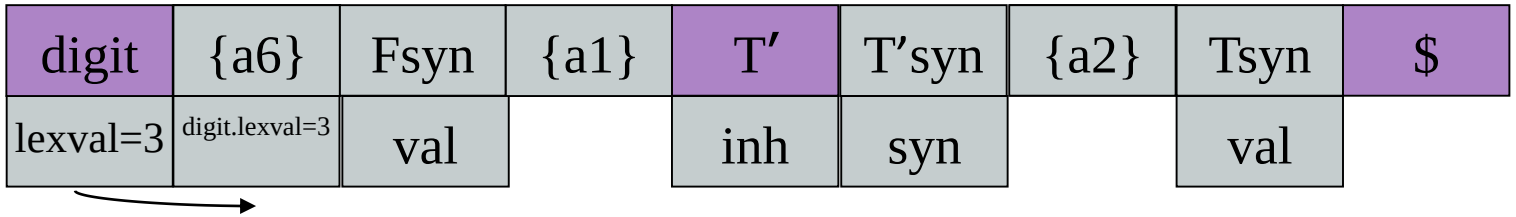
1) $T \rightarrow F \{ a_1 \}\ T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \}\ T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow \text{digit} \{ a_6 \}$

a_1 : $T'.inh = F.val$
a_2: $T.val = T'.syn$
a_3: $T_1'.inh = T'.inh \times F.val$
a_4 : $T'.syn = T_1'.syn$
a_5 : $T'.syn = T'.inh$
a_6 : $F.val = \text{digit}.lexval$

在非递归的预测分析过程中SDT

1) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	a_1 : $T'.inh = F.val$
2) $T' \rightarrow *F\ \{a_3\}\ T_1'\ \{a_4\}$	a_2: $T.val = T'.syn$
3) $T' \rightarrow \varepsilon\ \{a_5\}$	a_3: $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val =$ digit.$lexval$

3*5$

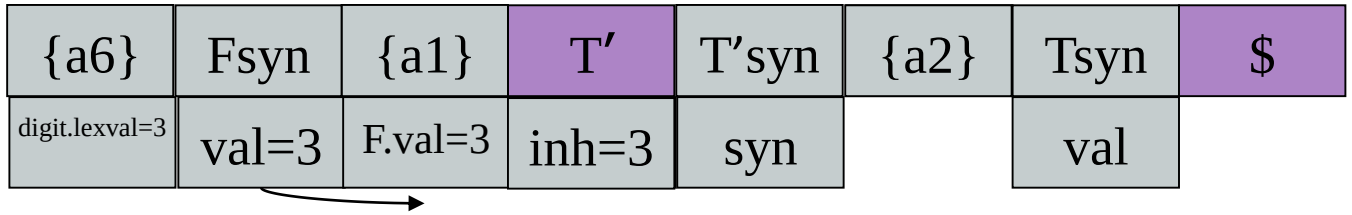


在非递归的预测分析过程中SDT

1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow \text{digit} \{ a_6 \}$

a_1 : $T'.inh = F.val$
a_2: $T.val = T'.syn$
a_3: $T_1'.inh = T'.inh \times F.val$
a_4 : $T'.syn = T_1'.syn$
a_5 : $T'.syn = T'.inh$
a_6 : $F.val = \text{digit}.lexval$

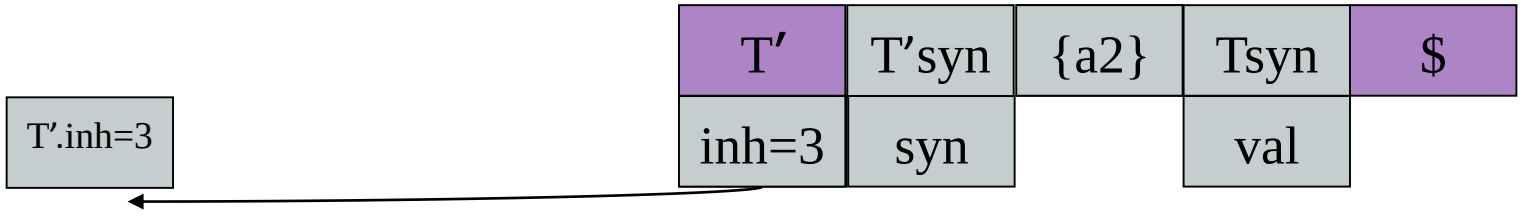
3*5$



在非递归的预测分析过程中SDT

1) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	a_1 : $T'.inh = F.val$
2) $T' \rightarrow *F\ \{a_3\}\ T_1'\ \{a_4\}$	a_2: $T.val = T'.syn$
3) $T' \rightarrow \varepsilon\ \{a_5\}$	a_3: $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val =$ digit.*lexval*

3*5$



在非递归的预测分析过程中SDT

1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow \text{digit} \{ a_6 \}$

a_1 : $T'.inh = F.val$
a_2: $T.val = T'.syn$
a_3: $T_1'.inh = T'.inh \times F.val$
a_4 : $T'.syn = T_1'.syn$
a_5 : $T'.syn = T'.inh$
a_6 : $F.val = \text{digit}.lexval$

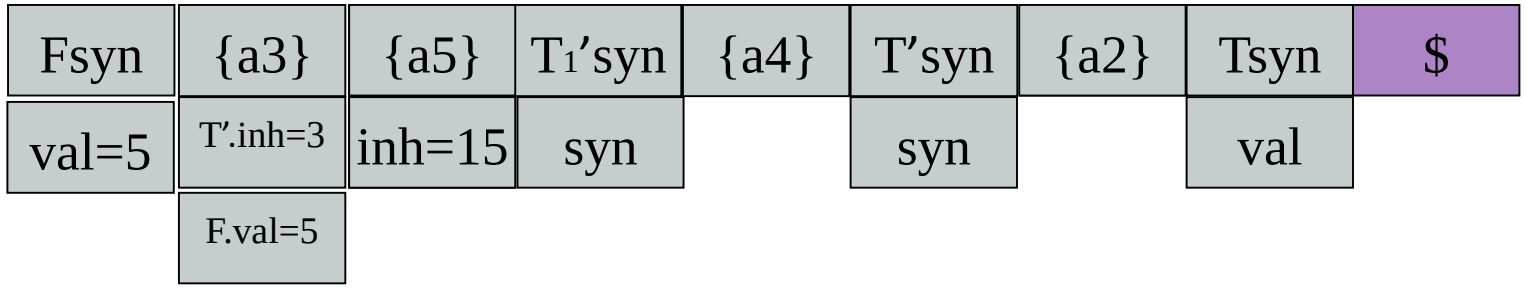
3*5$

digit	{a6}	Fsyn	{a3}	T1′	T1′syn	{a4}	T′syn	{a2}	Tsyn	$
lexval=5	digit.lexval=5	val=5	T′.inh=3	inh	syn		syn		val	

在非递归的预测分析过程中SDT

1) $T \rightarrow F \{a_1\}\ T' \{a_2\}$	a_1 : $T'.inh = F.val$
2) $T' \rightarrow *F \{a_3\}\ T_1' \{a_4\}$	a_2: $T.val = T'.syn$
3) $T' \rightarrow \varepsilon \{a_5\}$	a_3: $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val =$ digit.*lexval*

3*5$



在非递归的预测分析过程中SDT

1) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	a_1 : $T'.inh = F.val$
2) $T' \rightarrow *F\ \{a_3\}\ T_1'\ \{a_4\}$	a_2: $T.val = T'.syn$
3) $T' \rightarrow \varepsilon\ \{a_5\}$	a_3: $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val =$ digit.*lexval*

3*5$

{a5}	T1’syn	{a4}	T’syn	{a2}	Tsyn	$
inh=15	syn=15	T1’syn=15	syn=15	T’syn=15	val=15	

在非递归的预测分析过程中SDT

分析栈中的每一个记录都对应着一段执行代码

- 综合记录出栈时，要将综合属性值复制给后面特定的语义动作
- 变量展开时（即变量本身的记录出栈时），如果其含有继承属性，则要将继承属性值复制给后面特定的语义动作

在递归的预测分析过程中SDT

使用递归下降的语法分析器

- 每个非终结符号对应一个函数
- 函数的参数接受继承属性
- 返回值包含了综合属性

在递归下降语法分析过程中SDT

在函数体中：

- 首先选择适当的产生式
- 使用局部变量来保存属性
- 对于产生式体中的终结符号，读入符号并获取其（经词法分析得到的）综合属性
- 对于非终结符号，使用适当的方式调用相应函数，并记录返回值。

递归下降语法分析过程SDT

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow$ digit $\{ F.val =$ digit.lexval $\}$

```
T'syn T' (token, T'inh)
{ D: Fval, T1'inh, T1'syn;
if token="*" then
{ Getnext(token);
Fval=F(token);
T1'inh= T'inh × Fval;
Getnext(token);
T1'syn=T1'(token, T1'inh);
T'syn=T1'syn;
return T'syn;
}
else if token= " $" then
{ T'syn= T'inh;
return T'syn;
}
else Error;
}
```

递归下降语法分析过程SDT

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$
2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

```
Tval T(token)
{
D:  Fval, T'inh, T'syn;
Fval = F(token);
T'inh = Fval;
Getnext(token);
T'syn = T'(token, T'inh);
Tval = T'syn;
return Tval;
}
```

递归下降语法分析过程SDT

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$
2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

```
Fval F(token)
{
if token ≠ digit then Error;
Fval=token.lexval;
return Fval;
}
```

递归下降语法分析过程SDT

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$
2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

```
Desent()
{
D：  Tval；
Getnext(token)；
Tval = T(token)；
if token ≠ “$” then Error；
return ；
}
```

L属性定义的SDT-2

自底向上方式

以LL文法为基础的L属性SDD可以在LR语法分析过程中实现

- S-属性的自底向上计算
  - 综合属性——在归约时执行计算动作
- L-属性的自底向上计算
  - 语义动作不在最右——内嵌的语义动作
  - 继承属性的计算——关键问题
  - 解决方法——引入**标记**

L属性的自底向上实现

方法：

- 首先构造出L属性SDD的SDT，即在非终结符号前计算其继承属性
- 对于每个语义动作a，引入标记非终结符号M，并且 M→ε{a’}

例：

A → { B.i=f(A.i); } B C

引入标记非终结符号M后

A→ M B C

M→ ε {M.i=A.i； M.s=f(M.i);}

L属性的自底向上实现

例：A → { B.i=f(A.i); } B C
引入标记非终结符号M后
A→ M B C
M→ {M.i=A.i； M.s=f(M.i);}

目的:
使所有嵌入的动作都出现在产生式的末尾,可以自下而上处理继承属性

- a’的构造方法如下：
  - 将a中需要的A或者其它属性作为M的继承属性进行拷贝
  - 按照a中的规则计算各个属性，作为M的综合属性
  - a’必须设法找到相应的属性，因为产生式M→ε中没有A等符号。

L属性的自底向上实现

如何找到A.i？

- 设法使得在即将把BC归约到A时，A的继承属性存放在分析栈中BC的下方。
- 当执行到M的归约时，A.i的值存放在M的下方。(如果产生式体为K M B C，那么M的下方为K，K的下方存放A.i）
- M.s即B.i，存放在M所在的位置，即将来归约到B时，B.i存放在归约位置的下方。

自底向上实现L属性SDT的例子

1) $T \rightarrow F \{ T'.inh = F.val \}\ T' \{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \}\ T_1' \{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

1) $T \rightarrow F\ M\ T' \{ T.val = T'.syn \}$

   $M \rightarrow \varepsilon \{ M.i = F.val;\ M.s = M.i \}$

2) $T' \rightarrow *F\ N\ T_1' \{ T'.syn = T_1'.syn \}$

   $N \rightarrow \varepsilon \{ N.i1 = T'.inh;$

   $N.i2 = F.val;$

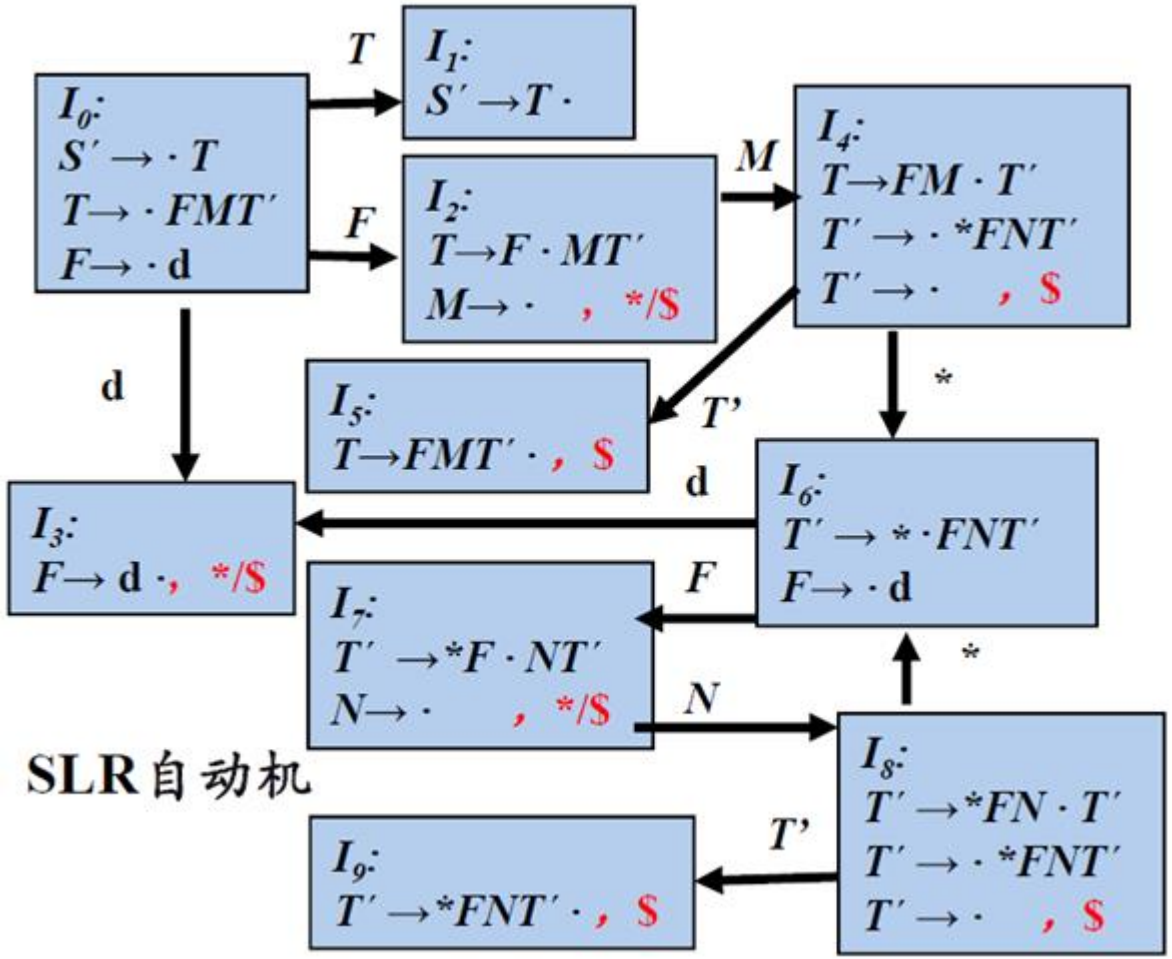
   $N.s = N.i1 \times N.i2 \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

修改***SDT***，(标记非终结符)
所有语义动作都
位于产生式末尾

T'inh = M.s



SLR自动机

) $T \rightarrow F\ M\ T'$ { $T.val = T'.syn$ }

$M \rightarrow \varepsilon$ {$M.i = F.val$; $M.s = M.i$ }

) $T' \rightarrow *F\ N\ T_1'$ { $T'.syn = T_1'.syn$ }

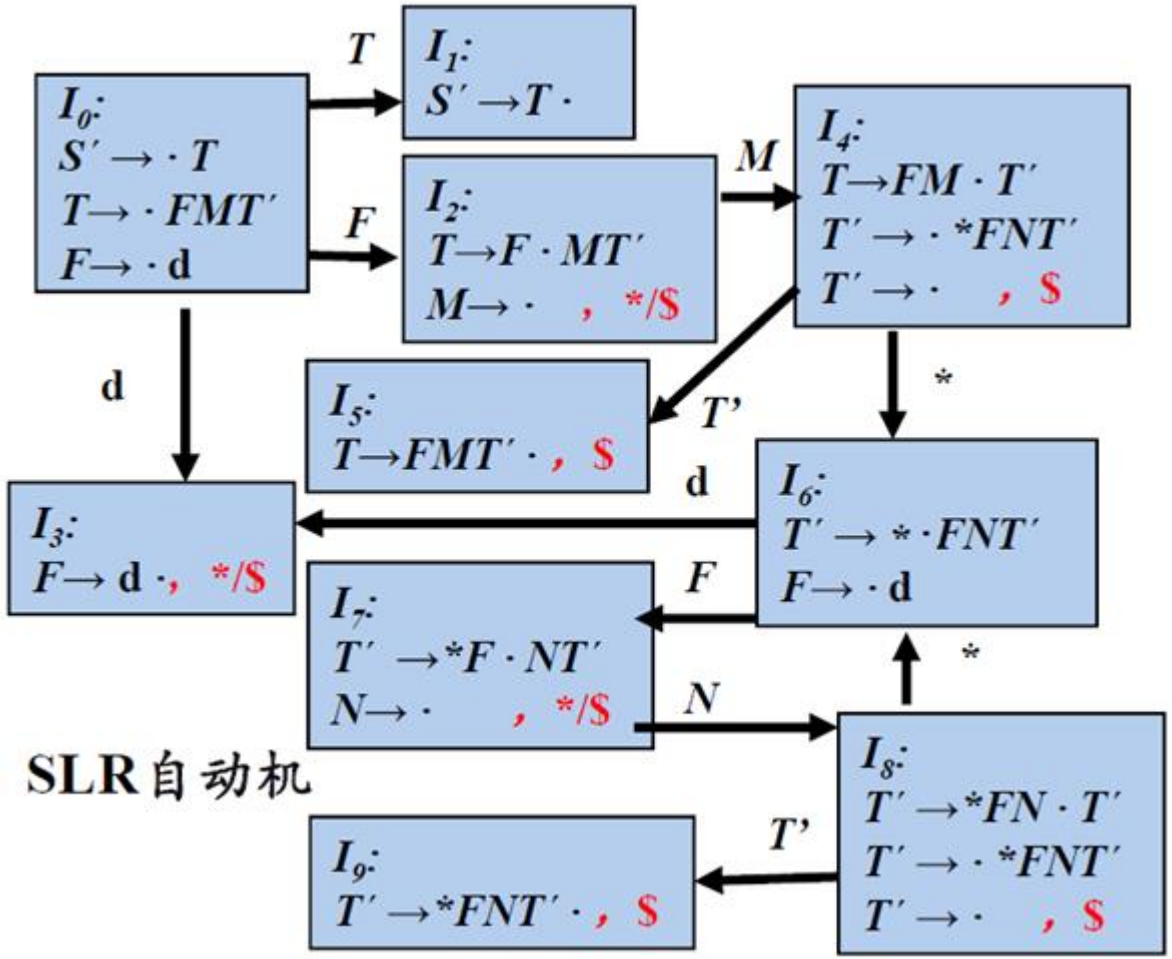
$N \rightarrow \varepsilon$ {$N.i1 = T'.inh$;
$N.i2 = F.val$;
$N.s = N.i1 \times N.i2$ }

) $T' \rightarrow \varepsilon$ { $T'.syn = T'.inh$ }

) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }

3*5$

0	2	4	6	7	8	9
$	F	M	*	F	N	T'
	3	$T'inh=3$		5	$T_1'inh=15$	$syn=15$



1) $T \rightarrow F\ M\ T'$ { $T.val = T'.syn$ }

$M \rightarrow \varepsilon$ {$M.i = F.val$; $M.s = M.i$ }

2) $T' \rightarrow *F\ N\ T_1'$ { $T'.syn = T_1'.syn$ }

$N \rightarrow \varepsilon$ {$N.i1 = T'.inh$;

$N.i2 = F.val$;

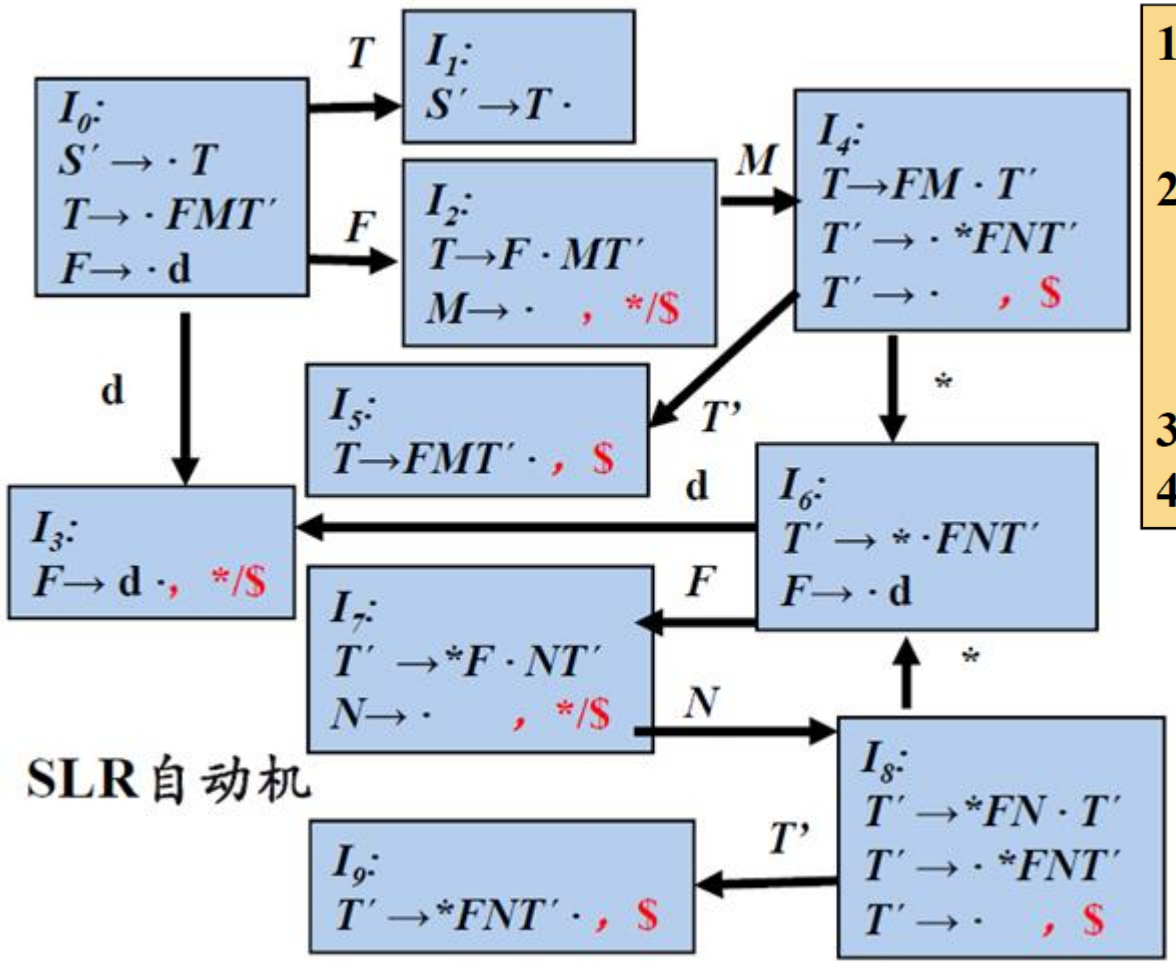
$N.s = N.i1 \times N.i2$ }

3) $T' \rightarrow \varepsilon$ { $T'.syn = T'.inh$ }

4) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }

3*5$

0	2	4	5
$	F	M	T'
	3	$T'inh=3$	$syn=15$



1) $T \rightarrow F\ M\ T'$ { $T.val = T'.syn$ }

$M \rightarrow \varepsilon$ {$M.i = F.val$; $M.s = M.i$ }

2) $T' \rightarrow *\ F\ N\ T_1'$ { $T'.syn = T_1'.syn$ }

$N \rightarrow \varepsilon$ {$N.i1 = T'.inh$;
$N.i2 = F.val$;
$N.s = N.i1 \times N.i2$ }

3) $T' \rightarrow \varepsilon$ { $T'.syn = T'.inh$ }

4) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }

3*5$

0	1
$	*T*
	val=15

自底向上实现L属性SDT

1) $T \to F\ M\ T'$ { $T.val = T'.syn$ }
   $M \to \varepsilon$ { $M.i = F.val$; $M.s = M.i$ }
2) $T' \to *\ F\ N\ T_1'$ { $T'.syn = T_1'.syn$ }
   $N \to \varepsilon$ { $N.i1 = T'.inh$;
   $N.i2 = F.val$;
   $N.s = N.i1 \times N.i2$ }
3) $T' \to \varepsilon$ { $T'.syn = T'.inh$ }
4) $F \to$ digit { $F.val$ = digit.$lexval$ }

将语义动作改写为可执行的栈操作

1) $T \to F\ M\ T'$ {stack[top-2]. val = stack[top].syn; top = top-2;}

$M \to \varepsilon$ {stack[top+1]. T'inh = stack[top].val; top = top+1;}

2) $T' \to *\ F\ N\ T_1'$ {stack[top-3]. syn = stack[top].syn; top = top-3;}

$N \to \varepsilon$ {stack[top+1]. T'inh = stack[top-2]. T´inh × stack[top].val;

top = top+1;}

3) $T' \to \varepsilon$ {stack[top+1].syn = stack[top]. T'inh; top = top+1;}

4) $F \to$ digit {stack[top].val = stack[top]. lexval;}

例题1. 为文法

$$S \to (\ L\ ) \mid a$$

$$L \to L\ ,\ S \mid S$$

写一个语法制导定义，它输出括号的对数。

$S' \to S$	$print\ (S.\ num)$
$S \to (\ L\ )$	$S.\ num = L.num + 1$
$S \to a$	$S.\ num = 0$
$L \to L_1\ ,\ S$	$L.\ num = L_1.\ num + S.\ num$
$L \to S$	$L.\ num = S.num$

例题2. 为文法

$$S \to (L) \mid a$$

$$L \to L , S \mid S$$

写一个翻译方案，它输出每个a的嵌套深度。例如，对于$(a , (a , a))$，输出的结果是1 2 2。

$S' \to \{S.depth = 0\}\ S$

$S \to \{L.depth = S.depth + 1\}\ (L)$

$S \to a\ \{print\ (S.depth)\}$

$L \to \{L_1.depth = L.depth\}\ L_1 ,$

$\{S.depth = L.depth\}\ S$

$L \to \{S.depth = L.depth\}\ S$

例题3. 为文法 $S \to (\ L\ ) \mid a$

$L \to L\ ,\ S \mid S$

写一个翻译方案，它打印出每个a在句子中是第几个字符。例如，当句子是$(\ a\ ,\ (\ a\ ,\ (\ a\ ,\ a\ )\ ,\ (a)\ )\ )$时，打印的结果是2 5 8 10 14。

$S' \to \{S.\ in = 0\ \}\ S$

$S \to \{L.\ in = S.\ in + 1\ \}\ (\ L\ )$

$\{S.\ out = L.\ out + 1\ \}$

$S \to a\ \{S.\ out = S.\ in + 1;\ print\ (S.\ out)\ \}$

$L \to \{L_1.\ in = L.\ in\ \} L_1\ ,$

$\{S.\ in = L_1.\ out + 1\ \}\ S$

$\{L.\ out = S.\ out\ \}$

$L \to \{S.\ in = L.\ in\ \} S\ \{L.\ out = S.\ out\ \}$