



第六章 语法制导翻译与 属性文法

重点：语法制导翻译的基本思想，语法制导定义，翻译模式，自顶向下翻译，自底向上翻译。

难点：属性的意义，对综合属性、继承属性和固有属性的理解，属性计算，怎么通过属性来表达翻译。





第6章 语法制导翻译与属性文法

6.1 语法制导翻译概述

6.2 语法制导定义

6.3 属性计算

6.4 翻译模式

6.5 本章小结

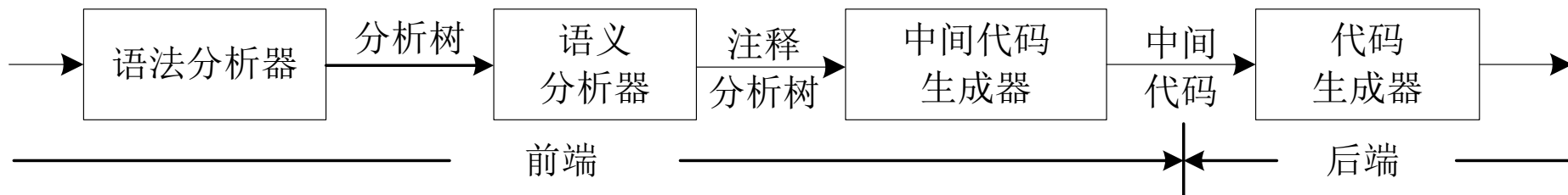


问题

- 为什么进行词法和语法分析?
- 用 $A \rightarrow \alpha$ 进行归约表达的是什么意思?
- 看: operand+term
- $E \rightarrow E_1 + T$
- E_1 的值+ T 的值的的结果作为 E 的值——即: 取来 E_1 的值和 T 的值做加法运算, 结果作为 E 的值
 - $E.val = E_1.val + T.val$

6.1 语法制导翻译概述

- 为了提高编译程序的可移植性，一般将编译程序划分为**前端**和**后端**。
 - 前端通常包括**词法分析、语法分析、语义分析、中间代码生成、符号表的建立**，以及**与机器无关的中间代码优化**等，它们的实现一般不依赖于具体的目标机器。
 - 后端通常包括**与机器有关的代码优化、目标代码的生成、相关的错误处理以及符号表的访问**等。





6.1 语法制导翻译概述

- 语义分析器的主要任务是检查各个语法结构的**静态语义**，即验证语法正确的程序结构**是否真正有意义**，也称为**静态语义分析**或**静态检查**
 - 类型检查：操作数和操作符的**类型是否相容**；
 - 控制流检查：控制流转向目标**地址是否合法**；
 - 惟一性检查：对象是否被**重复定义**；
 - 关联名检查：同一名字多次特定出现**是否一致**。



6.1 语法制导翻译概述

- 静态检查比较简单，可以和其他工作结合
- 将静态检查和中间代码生成结合到语法分析中进行的技術称为语法制导翻译 (syntax-directed translation)。



6.1 语法制导翻译概述

■ 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**。
- 一旦语法分析器识别出一个语法结构（例如 $E \rightarrow E+T$ ）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。



6.1 语法制导翻译概述

- 语法制导翻译的基本思想
 - 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**)。



6.1 语法制导翻译概述

- 语义子程序的功能

- 指明相应产生式中各个**文法符号的具体含义**，并规定了使用该产生式进行分析时所应采取的**语义动作**。
- 语义信息是通过**文法符号来携带和传递**的。



6.1 语法制导翻译概述

- 一个文法符号 X 所携带的**语义信息**称为 X 的语义属性，简称为**属性**
 - 它是根据翻译的需要设置的，主要用于描述语法结构的语义。
 - 文法符号的属性的计算规则称为**语义规则**
 - 一个变量的属性有类型、值和存储地址等

6.1 语法制导翻译概述

- 属性值的计算和产生式相关联，随着语法分析的进行，执行属性值的计算，完成语义分析和翻译的任务。
- 语法结构具有规定的语义
 - $E \rightarrow E_1 + E_2$ $E.val := E_1.val + E_2.val$
- 问题：如何根据被识别出的语法成分进行语义处理？
 - 亦即怎样将属性值的计算及翻译工作同产生式相关联？

6.1 语法制导翻译概述

■ 典型处理方法一：语法制导定义

- 通过将属性与文法符号关联、将**语义规则与产生式关联**来描述语言结构的翻译方案
- 对应每一个产生式编写一个语义子程序，当一个产生式获得**匹配**时，就**调用相应的语义子程序**来实现语义检查与翻译

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

- $T \rightarrow T_1 * F \{T.val := T_1.val * F.val\}$

- $F \rightarrow \text{digit} \{F.val := \text{digit.lexval}\}$

■ 适宜在完成**归约**的时候进行

6.1 语法制导翻译概述

■ 典型处理方法二：翻译模式

- 通过将属性与文法符号关联，并将**语义规则插入到产生式的右部**来描述语言结构的翻译方案
- 在产生式的右部的适当位置，插入相应的语义动作，**按照分析的进程，执行遇到的语义动作**
- $D \rightarrow T \{ L.inh := T.type \} L$
- $T \rightarrow \text{int} \{ T.type := \text{integer} \}$
- $T \rightarrow \text{real} \{ T.type := \text{real} \}$
- $L \rightarrow \{ L_1.inh := L.inh \} L_1, \text{id} \{ \text{addtype}(\text{id.entry}, L.inh) \}$
- $L \rightarrow \text{id} \{ \text{addtype}(\text{id.entry}, L.inh) \}$

■ 适宜在进行**推导**时完成



6.2 语法制导定义

- **语法制导定义是附带有属性和语义规则的上文无关文法**
 - **属性是与文法符号相关联的语义信息**
 - **语义规则是与产生式相关联的语义动作**



6.2 语法制导定义

- **语法制导定义是基于语言结构的语义要求设计的，类似于程序设计**
- **语法制导定义中的属性类似于程序中用到的数据结构，用于描述语义信息**
- **语义规则类似于计算，用于收集、传递和计算语义信息。**



6.2 语法制导定义

- 由于属性的计算在语法分析过程中进行，**属性**通常被保存在**分析树的相关节点**中
- 令 X 表示分析树中的某节点，则 $X.a$ 表示该文法符号的属性 a 的值



6.2 语法制导定义

- **综合属性**：节点的属性值是通过分析树中**该节点或其子节点的属性值**计算出来的
- **继承属性**：节点的属性值是由**该节点、该节点的兄弟节点或父节点的属性值**计算出来的
- **固有属性**：通过词法分析**直接得到**的属性
- **依赖图**：描述**属性之间依赖关系**的图，根据语义规则来构造
- **注释分析树**：节点带有属性值的分析树

6.2 语法制导定义

- 在语法制导定义中, $\forall A \rightarrow \alpha \in P$ 都有与之相关联的一套语义规则, 规则形式为

$$b := f(c_1, c_2, \dots, c_k),$$

f 是一个函数,

1. b 是 A 的一个综合属性并且 c_1, c_2, \dots, c_k 是 α 或 A 中的符号的属性

2. b 是 α 中某个符号的一个继承属性并且 c_1, c_2, \dots, c_k 是 A 或 α 中的任何文法符号的属性。

这两种情况都说属性 b 依赖于属性 c_1, c_2, \dots, c_k

例6.1 台式计算器的语法制导定义

产生式

$L \rightarrow En$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

计算 $6 + 7 * 8$

语义规则

$\text{print}(E.val)$

$E.val := E_1.val + T.val$

$E.val := T.val$

$T.val := T_1.val + F.val$

$T.val := F.val$

$F.val := E.val$

$F.val := \text{digit.lexval}$

6.2 语法制导定义

- 只含**综合属性**的语法制导定义称为 **S -属性定义**，通常在**自底向上**的分析方法计算属性值
- 在建立每一个**结点**处使用**语义规则**来计算综合属性值，即在用哪个产生式进行归约后，就执行那个产生式的 **S -属性定义**计算属性的值
- 没有副作用的语法制导定义又称为**属性文法**
- 属性文法的语义规则**单纯根据常数和和其它属性的值**来定义某个属性的值

6.2 语法制导定义

- 当分析树的结构同源代码的抽象语法不“匹配”时，**继承属性**将非常有用。
- 下面的例子可以说明怎样用继承属性来解决这种不匹配问题，产生不匹配的原因是因为**文法通常是为语法分析而不是为翻译设计的。**
- 例6.2
 - 考虑如何在自顶向下的分析过程中计算 $4*8*9$ 这样的表达式项
 - 消除左递归之后的算数表达式文法的一个子集：
$$T \rightarrow FT' \quad T' \rightarrow *FT_1' \quad T' \rightarrow \varepsilon \quad F \rightarrow \text{digit}$$

表6.3 为适于自顶向下分析的文法设计的语法制导定义

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

4*8*9的注释分析树

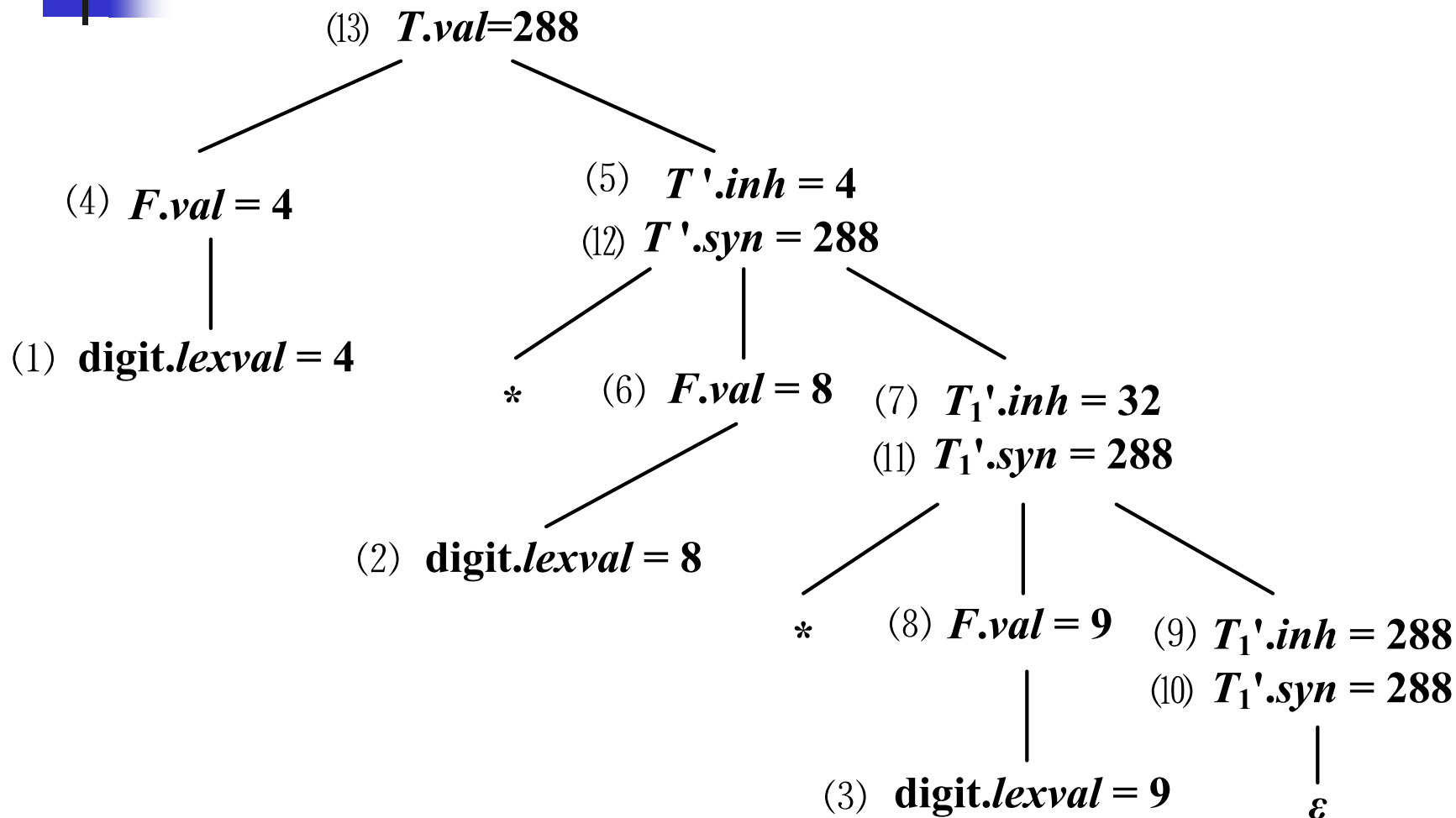


表6.3中语法制导定义对应的翻译模式

- 如果对 $4*8*9$ 进行自顶向下的语法制导翻译，则val的值的计算顺序为(1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)
- 根据上述对val值的计算顺序，可以将表6.3中的语法制导定义转换成如下的翻译模式
- $T \rightarrow F\{T'.inh := F.val\}T'\{T.val := T'.syn\}$
- $T' \rightarrow *F\{T_1'.inh := T'.inh \times F.val\}T_1'\{T'.syn := T_1'.syn\}$
- $T' \rightarrow \varepsilon\{T'.syn := T'.inh\}$
- $F \rightarrow \text{digit}\{F.val := \text{digit.lexval}\}$



6.3 属性计算

- 语义规则定义了属性之间的**依赖关系**，这种依赖关系将影响属性的**计算顺序**
- 为了确定分析树中各个属性的计算顺序，我们可以用**图**来表示属性之间的依赖关系，并将其称为**依赖图**

6.3.1 依赖图

- 依赖图其实就是一个有向图，用于描述分析树中节点的**属性和属性间的相互依赖关系**。
- 每个属性对应依赖图中的一个节点，如果**属性 b 依赖于属性 c** ，则从属性 c 的节点有一条有向边**指向属性 b** 的节点。
- 属性间的依赖关系是**根据相应的语义规则**确定的。
- 如果依赖图中**没有回路**，则利用它可以很方便地求出属性的计算顺序。



6.3.1 依赖图

依赖图的构造方法

for 分析树的每个节点 n do

for 与节点 n 对应的文法符号的每个属性 a do

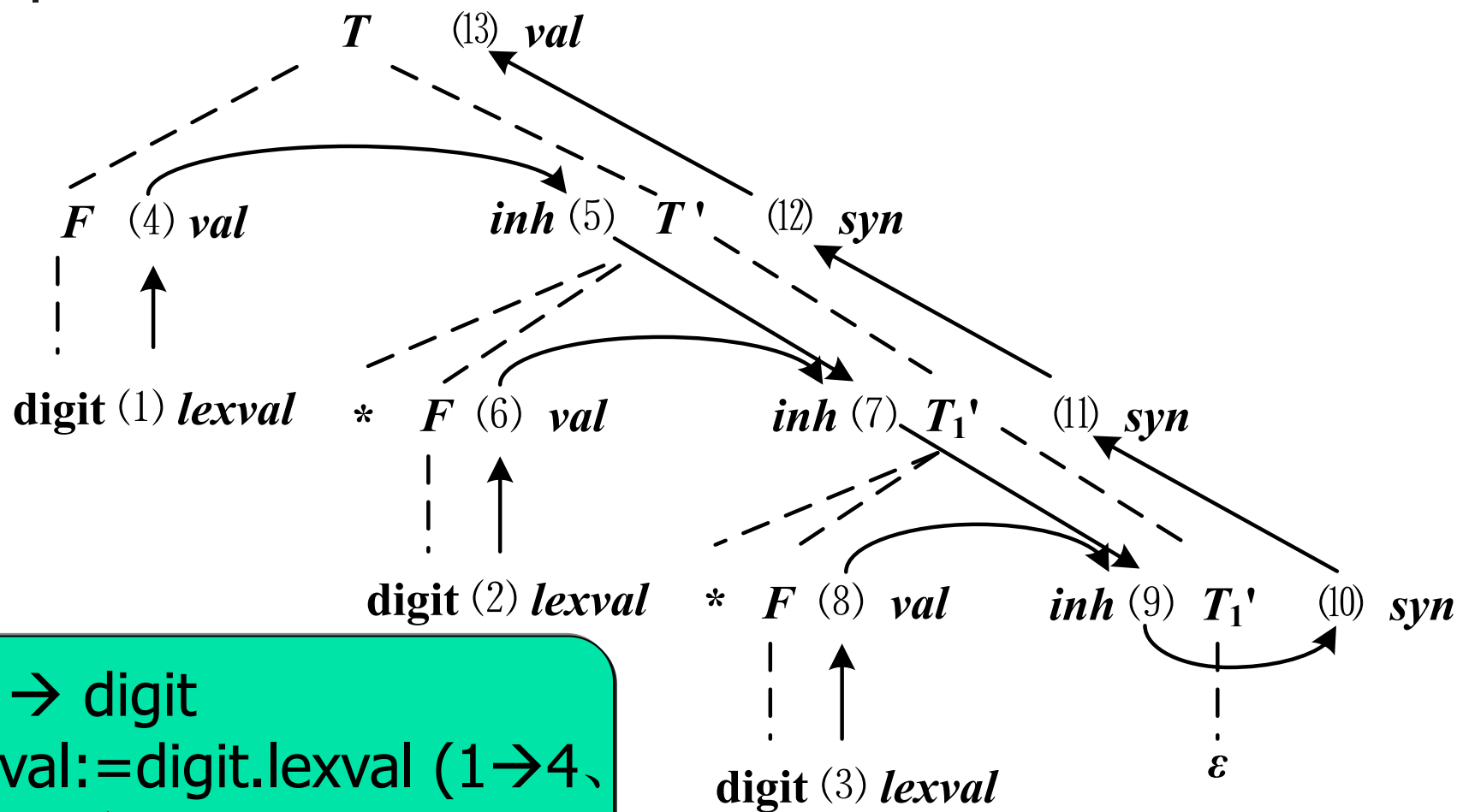
在依赖图中为 a 构造一个节点;

for 分析树的每个节点 n do

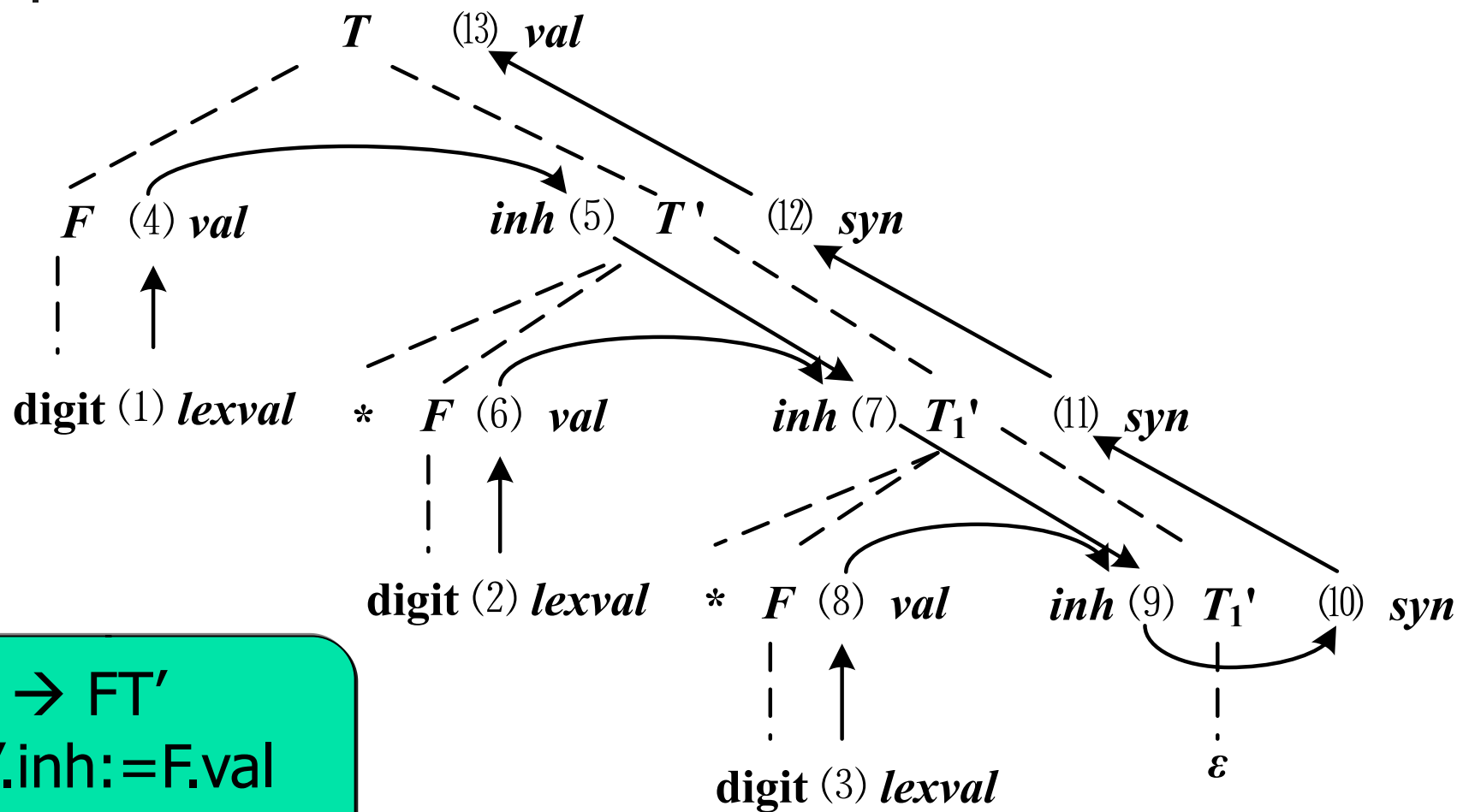
for 节点 n 所用产生式对应的每条语义规则 $b := f(c_1, c_2, \dots, c_k)$ do

分别构造从节点 c_1, c_2, \dots, c_k 到节点 b 的有向边;

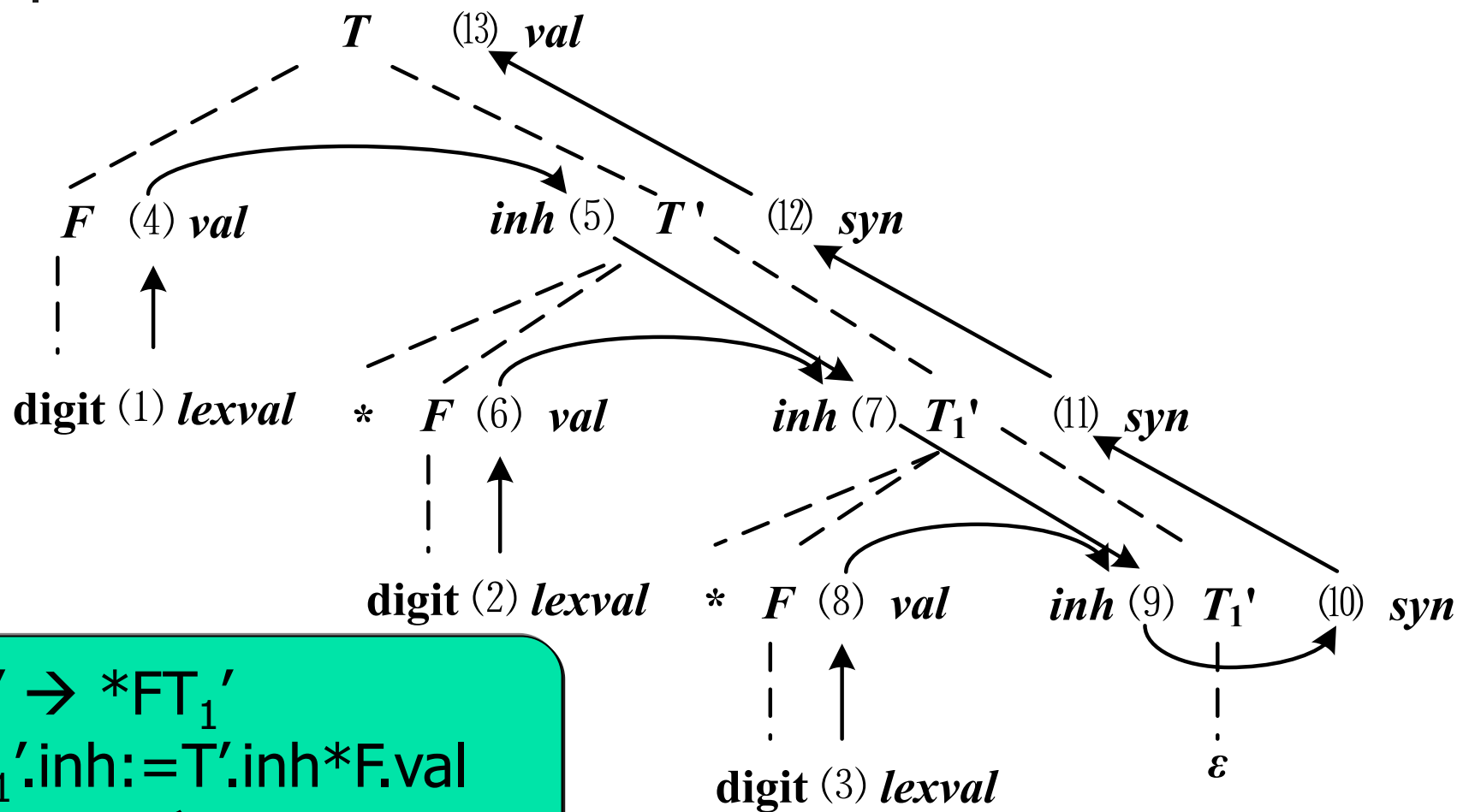
例6.3 图6.3中注释分析树的依赖图



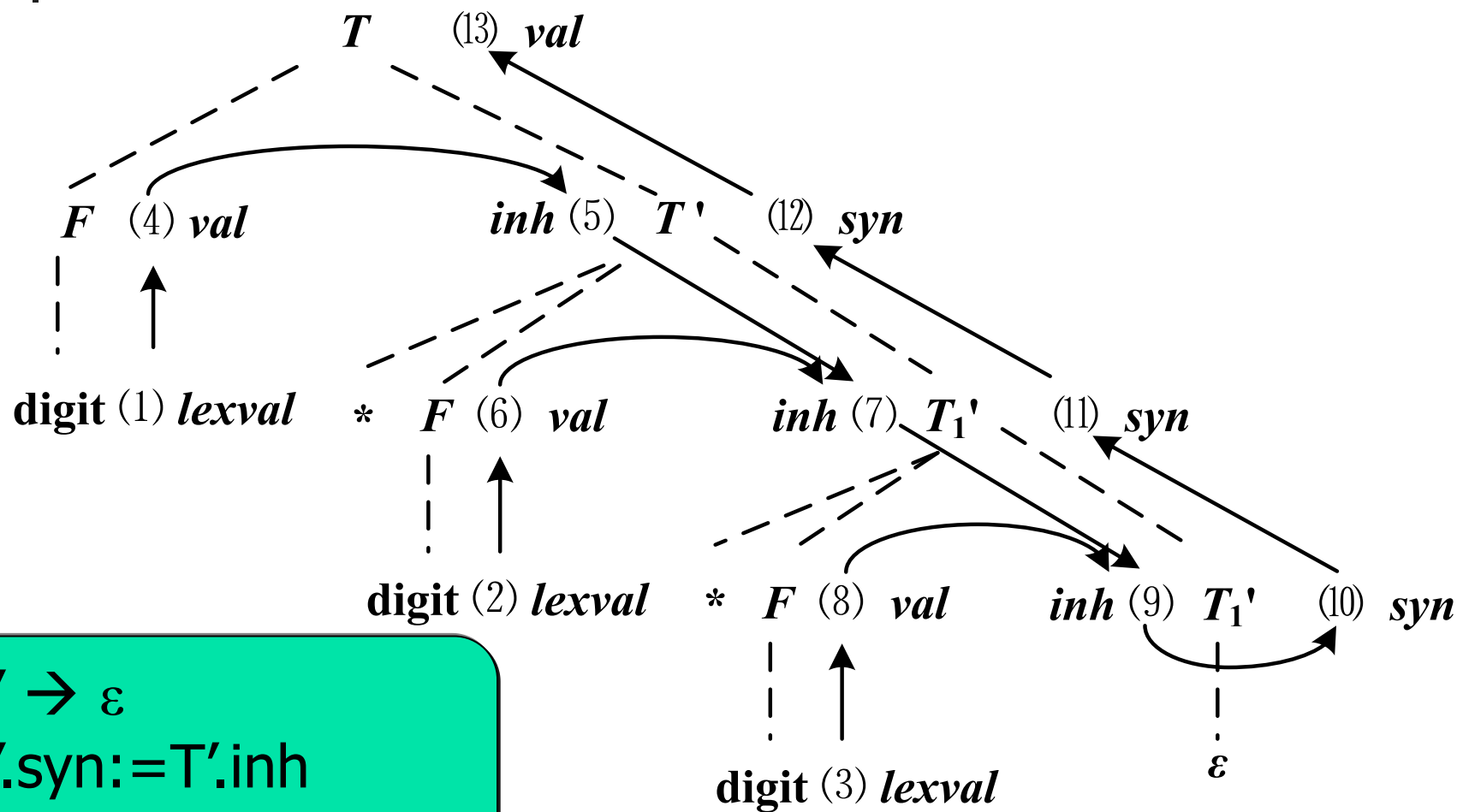
例6.3 图6.3中注释分析树的依赖图



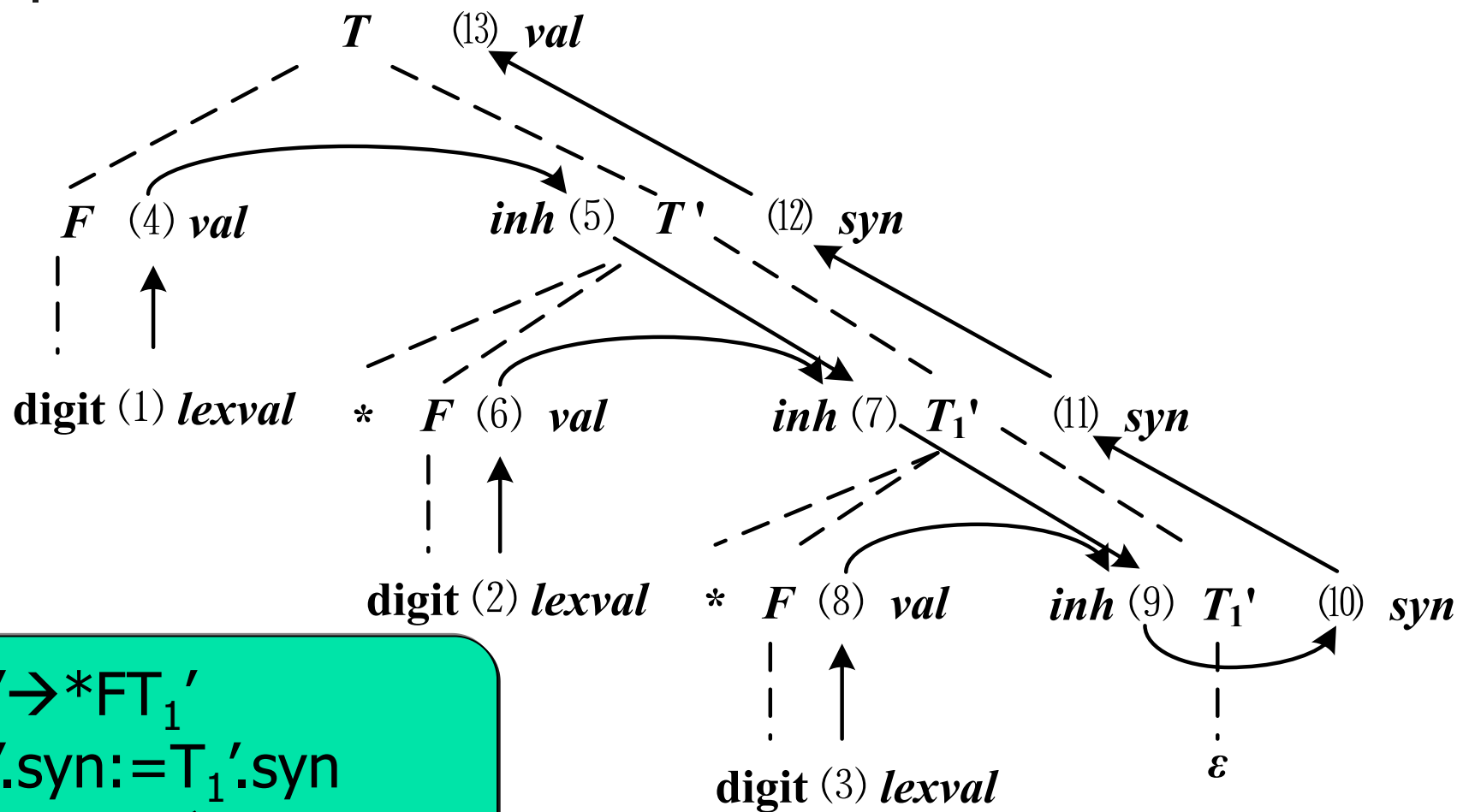
例6.3 图6.3中注释分析树的依赖图



例6.3 图6.3中注释分析树的依赖图

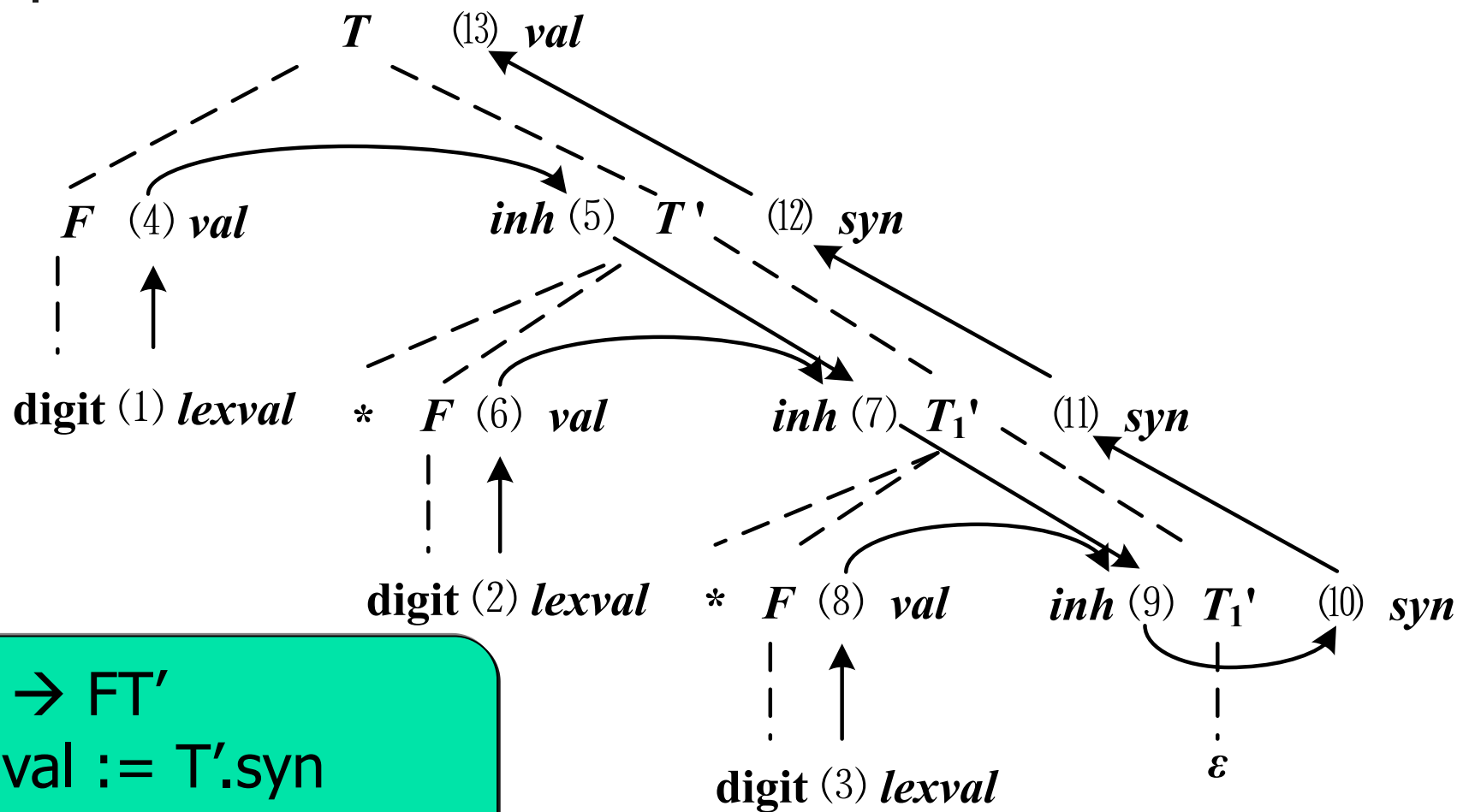


例6.3 图6.3中注释分析树的依赖图



$T' \rightarrow *FT_1'$
 $T'.syn := T_1'.syn$
 $(10 \rightarrow 11 \text{ 和 } 11 \rightarrow 12)$

例6.3 图6.3中注释分析树的依赖图



$T \rightarrow FT'$
 $T.val := T'.syn$
 $(12 \rightarrow 13)$



6.3.2 属性的计算顺序

■ 拓扑排序

- 一个无环有向图的**拓扑排序**是图中结点的任何顺序 m_1, m_2, \dots, m_k , 如果 $m_i \rightarrow m_j$ 是依赖图中 m_i 到 m_j 的一条边, 那么序列中 m_i 必须出现在 m_j 的前面。



6.3.2 属性的计算顺序

■ 拓扑排序

- 若依赖图中有环，不存在拓扑排序，属性的计算存在循环依赖关系。
- 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。

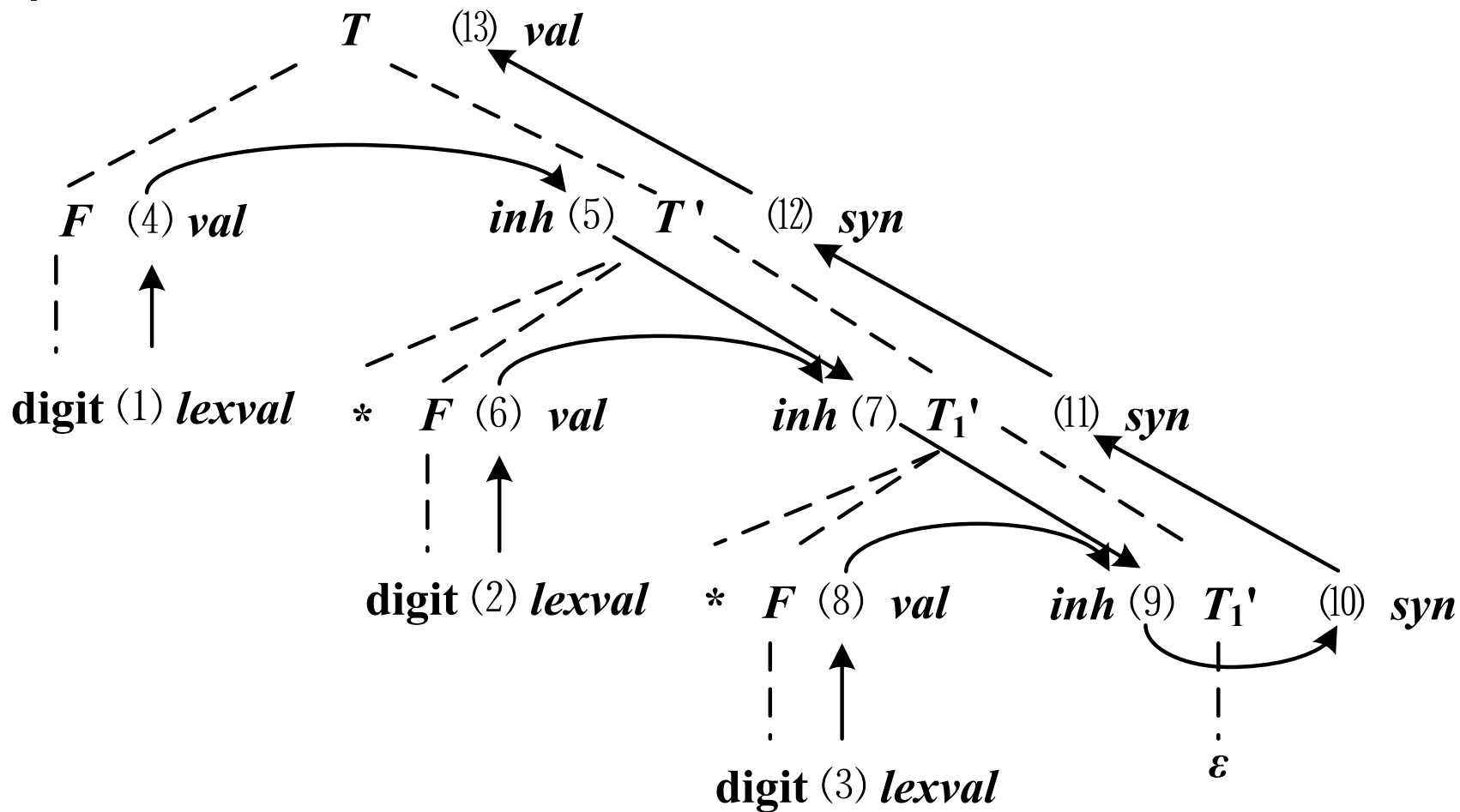


6.3.2 属性的计算顺序

- 无前驱的顶点优先拓扑排序方法
 - 在依赖图中寻找一个入度为0的节点，输出该节点
 - 删除该节点的所有出边
 - 在剩下的图中继续寻找入度为0的节点，直到输出图中的所有节点为止
 - 没有回路的依赖图中总能找到一个入度为0的节点。
 - **反证法**：如果在图中不存在入度为0的节点，那么从节点 i 走到 i 的前驱，继续到 i 的前驱的前驱，总能回到 i 本身，与图中不存在环矛盾

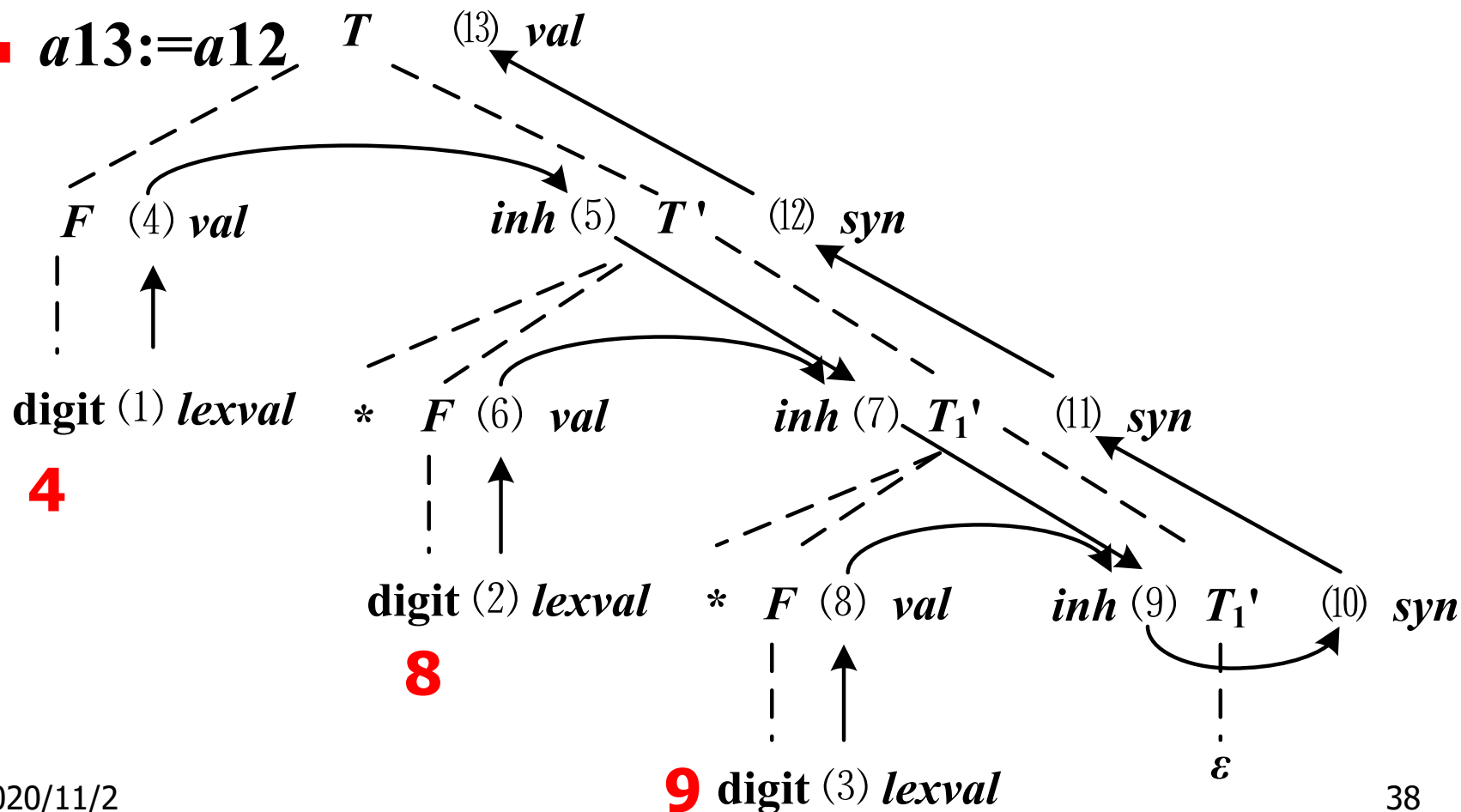
下图的拓扑排序为:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13



■ 根据拓扑排序得到的翻译程序

■ $a4:=4$ $a5:=a4$ $a6:=8$
 ■ ~~$a7:=a5 \times a6$~~ ~~$a8:=9$~~ $a9:=a7 \times a8$
 ■ $a10:=a9$ $a11:=a10$ $a12:=a11$
 ■ $a13:=a12$





6.3.2 属性的计算顺序

- 上述属性计算方法又称为**分析树法**
 - 在编译时需要显式地构造分析树和依赖图
 - 优点：可以多次遍历分析树，从而使得属性的计算不依赖于所采用的语法分析方法以及属性间严格的依赖关系。
 - 缺点：编译的时空效率比较低，而且如果分析树的依赖图中存在回路的话，这种方法将会失效。



6.3.2 属性的计算顺序

- 给定语法制导定义，要**预先判断是否存在一棵其依赖图不含回路**的分析树是比较困难的
- 可采用**保证属性计算顺序**的语法制导定义来实现翻译，可以肯定其对应分析树的依赖图中不出现回路
- S-属性定义和L-属性定义

6.3.3 S-属性定义

- 定义6.1 . **只含综合属性**的语法制导定义称为S-属性定义。
- 如果某个语法制导定义是S-属性定义，则可以按照**自下而上的顺序**来计算分析树中节点的属性。
- 一种简单的属性计算方法是对分析树进行**后根遍历**，在最后一次遍历节点 N 时计算与节点 N 相关联的属性。
 - $postorder(N) \{$
 - for N 的每个子节点 M (从左到右) $postorder(M);$
 - 计算与节点 N 相关联的属性;
 - $\}$



6.3.4 L-属性定义

- S-属性定义的属性自下而上传播
- 如果增加**从左向右**的传播方向，就得到L-属性定义

6.3.4 L-属性定义

- 定义6.2 一个语法制导定义被称为L-属性定义，当且仅当它的每个属性或者是**综合属性**，或者是满足如下条件的**继承属性**：

设有产生式 $A \rightarrow X_1 X_2 \dots X_n$ ，右部符号 X_i ($1 \leq i \leq n$) 的**继承属性**只依赖于下列属性：

- (1) A 的继承属性；
- (2) 产生式中 X_i 左边的符号 X_1, X_2, \dots, X_{i-1} 的综合属性或继承属性；
- (3) X_i 本身综合属性或继承属性，但前提是 X_i 的属性不能在依赖图中形成回路。



6.3.4 L-属性定义

L-属性定义示例

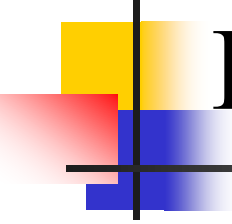
产生式	语义规则
$T \rightarrow FT'$	$T'.inh := F.val$ $T.val := T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh := T'.inh \times F.val$ $T'.syn := T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$



6.3.4 L-属性定义

不是L-属性定义的语法制导定义

产生式	语义规则
$A \rightarrow BC$	$A.syn := B.b$ $B.inh := f(C.c, A.syn)$



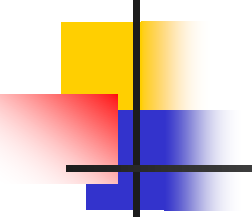
L-属性定义中的属性计算

```
visit(N) {  
    for N的每个子节点M(从左到右) {  
        计算节点M的继承属性;  
        visit (M);  
    }  
    计算节点N的综合属性;  
};
```



6.3.5 属性计算示例

- 以**抽象语法树**为例讨论如何在语法制导定义中进行属性计算



6.3.5 属性计算示例

- **抽象语法树**是把分析树中对语义无关紧要的成分去掉，是**分析树**的抽象形式。
- 抽象语法树是常用的一种**中间表示**形式，把语法分析和翻译分开，原因在于
 - 适于语法分析的文法可能不反映语言成分的自然层次结构；
 - 语法分析方法的限制，对分析树结点的访问顺序和翻译需要的访问顺序不一致。

6.3.5 属性计算示例

- **构造表达式的抽象语法树使用的函数**

1. *mknnode*(*op*, *left*, *right*) 建立一个标记为*op*的运算符结点，两个域*left*和*right*分别是指向左右运算对象的指针。
2. *mkleaf*(*id*, *entry*) 建立一个标记为*id*的标识符结点，其域*entry*是指向该标识符在符号表中相应表项的指针。
3. *mkleaf*(*num*, *val*) 建立一个标记为*num*的数结点，其域*val*用于保存该数的值。

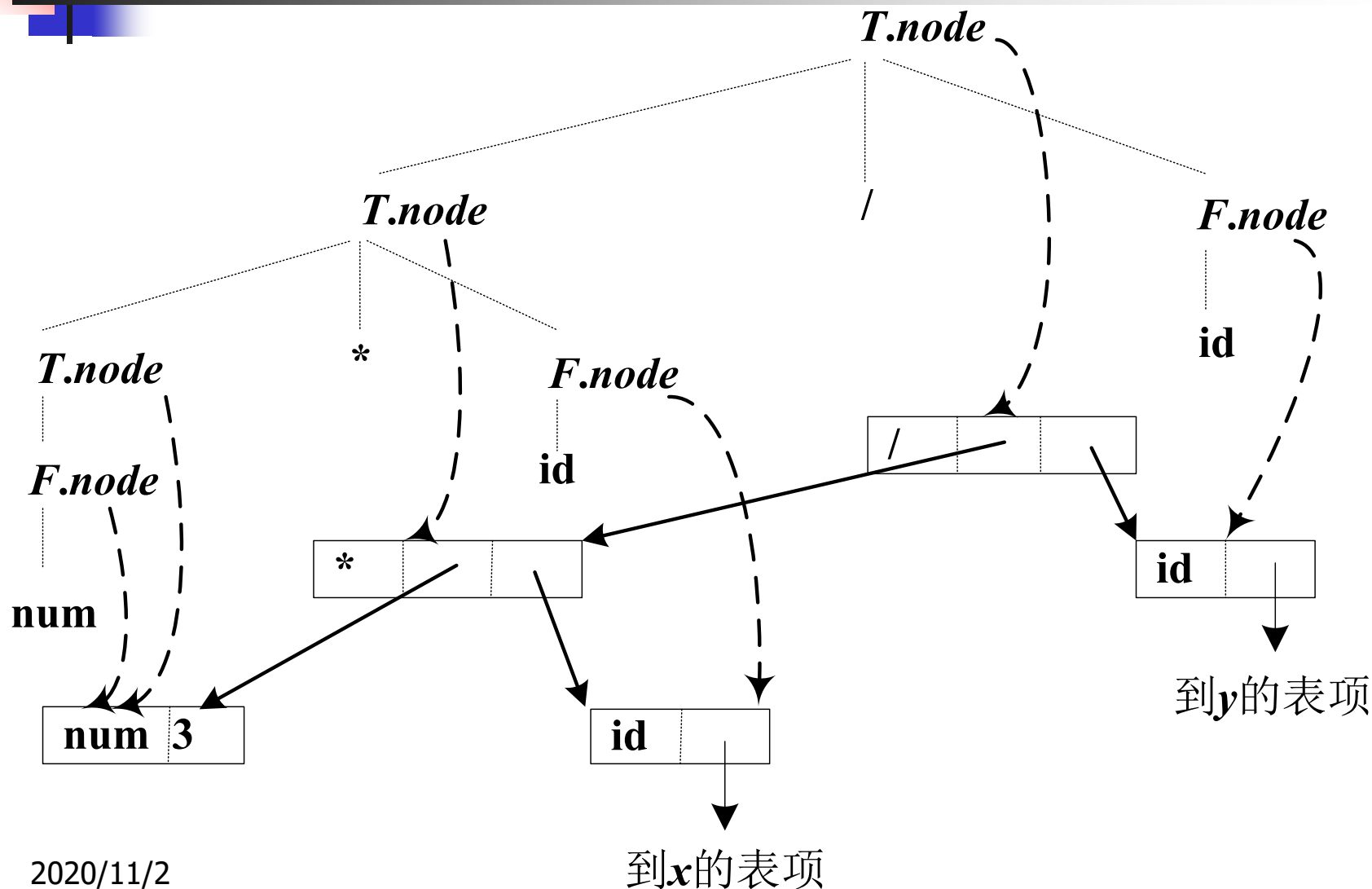


6.3.5 属性计算示例

构造表达式的抽象语法树的语法制导定义

产生式	语义规则
(1) $T \rightarrow T_1 * F$	$T.node := mknode('*', T_1.node, F.node)$
(2) $T \rightarrow T_1 / F$	$T.node := mknode('/', T_1.node, F.node)$
(3) $T \rightarrow F$	$T.node := F.node$
(4) $F \rightarrow (E)$	$F.node := E.node$
(5) $F \rightarrow id$	$F.node := mkleaf(id, id.entry)$
(6) $F \rightarrow num$	$F.node := mkleaf(num, num.val)$

图6.5 $3 * x / y$ 的语法树的构造



3*x/y的抽象语法树的构造步骤

```
p1 := mkleaf(num, 3);  
p2 := mkleaf(id, entry-x);  
p3 := mknode('*', p1, p2);  
p4 := mkleaf(id, entry-y);  
p5 := mknode('/', p3, p4);
```

图6.5的语法树是**自底向上**构造的，对于那些为便于进行**自顶向下**分析而设计的文法来说，使用同样的步骤照样可以建立图6.5中的抽象语法树。当然，分析树的结构可能大不相同，而且可能需要引入**继承属性**来传递语义信息。

在自顶向下分析过程中构造语法树

产生式	语义规则
(1) $T \rightarrow FT'$	$T.node := T'.syn$ $T'.inh := F.node$
(2) $T' \rightarrow *FT_1'$	$T_1'.inh := mknode('*', T'.inh, F.node)$ $T'.syn := T_1'.syn$
(3) $T' \rightarrow /FT_1'$	$T_1'.inh := mknode('/', T'.inh, F.node)$ $T'.syn := T_1'.syn$
(4) $T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
(5) $F \rightarrow (E)$	$F.node := E.node$
(6) $F \rightarrow id$	$F.node := mkleaf(id, id.entry)$
(7) $F \rightarrow num$	$F.node := mkleaf(num, num.val)$

The diagram illustrates the construction of a parse tree for the expression "num * id / id ε" using LR(0) items and transitions. The tree structure shows the derivation from the start symbol T (13) node through various LR(0) items and transitions (F, inh, T' , syn) to the final expression.

The expression is "num * id / id ε".

The LR(0) items and transitions are:

- T (13) *node*
- F (2) *node*
- inh (3) T' (12) *syn*
- F (5) *node*
- inh (6) T' (11) *syn*
- F (8) *node*
- inh (9) T' (10) *syn*

The transitions are:

- F (2) *node* → F (5) *node*
- inh (3) T' (12) *syn* → inh (6) T' (11) *syn*
- inh (6) T' (11) *syn* → inh (9) T' (10) *syn*

The expression is "num * id / id ε".



6.4 翻译模式

- **翻译模式**是语法制导定义的一种**实现**形式。其中属性与文法符号相关联，**语义规则**(也称为**语义动作**)用**花括号 { }**括起来，可插入到产生式右部任何合适的位置。
- 这是一种语法分析和语义动作交错的表示法，它表达在按**深度优先**遍历分析树的过程中何时执行语义动作。
- 翻译模式给出了使用语义规则进行计算的**时机**,可看成是分析过程中翻译的注释。

例6.10 一个简单的翻译模式

将中缀表达式翻译成后缀表达式：

$$E \rightarrow TR$$

$$R \rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R_1 | \varepsilon$$

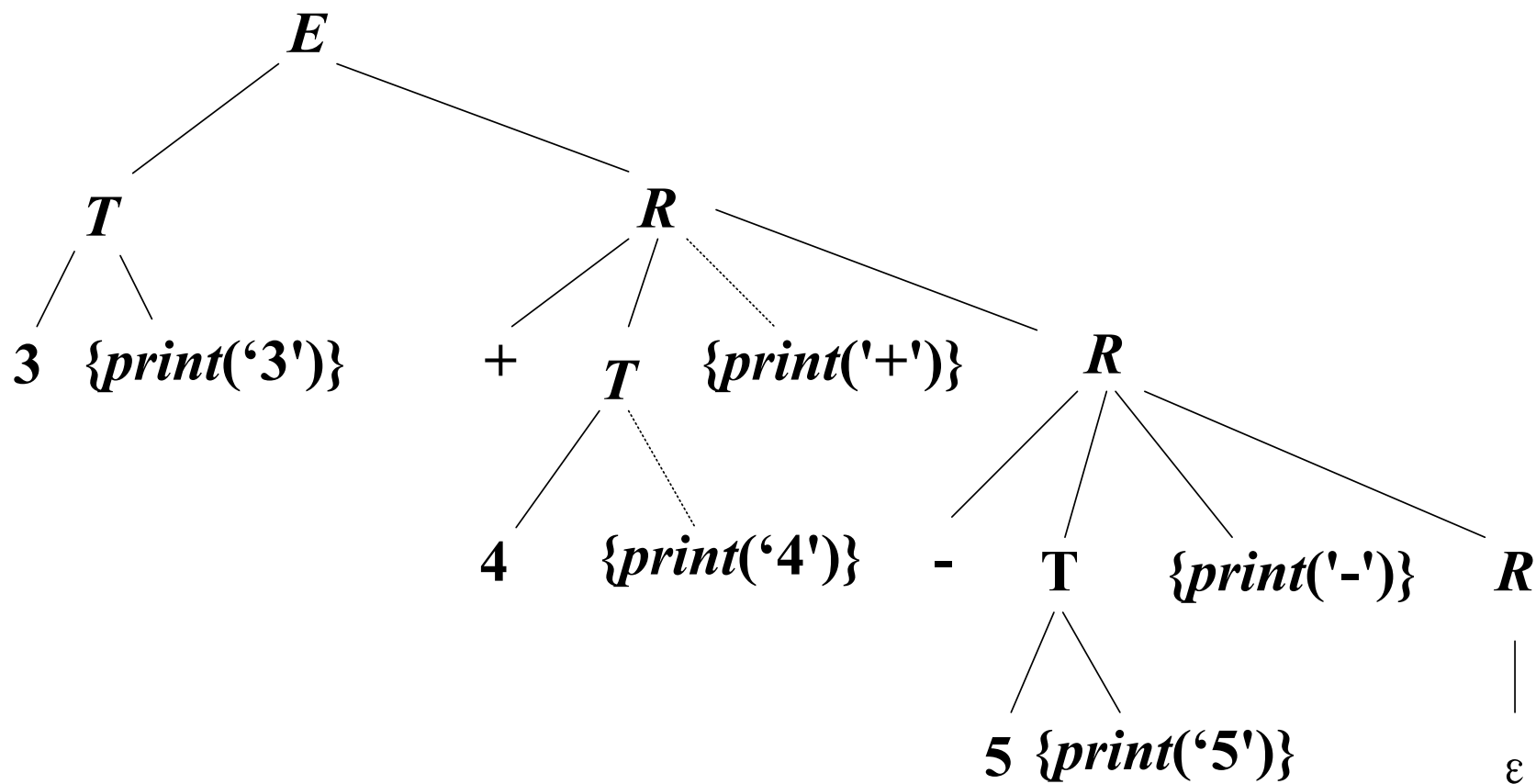
$$T \rightarrow \text{num} \{ \text{print}(\text{num.val}) \}$$

把语义动作看成终结符号，输入3+4-5,其分析树如图6.8，当按**深度优先遍历**，执行遍历中访问的语义动作，将输出

3 4 + 5 -

它是输入表达式3+4-5的后缀式。

图6.8 3+4-5的带语义动作的分析树





6.4 翻译模式

- 翻译模式的设计的前提

保证语义动作**不会引用还没计算出来的属性值**



6.4 翻译模式

1. 只需要综合属性的情况

为每一个语义规则建立一个赋值动作，并把该动作放在相应的**产生式右部的末尾**。

例如： $T \rightarrow T_1 * F$ $T.val := T_1.val * F.val$

转换成：

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



6.4 翻译模式

2. 既有综合属性又有继承属性

- 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。
- 一个动作不能引用这个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放在产生式右部的末尾。

6.4 翻译模式

下面的翻译模式不满足要求:

$$S \rightarrow A_1 A_2 \quad \{A_1 \cdot in := 1; \quad A_2 \cdot in := 2\}$$
$$A \rightarrow a \quad \{ \text{print}(A \cdot in) \} \quad /* A \cdot in \text{尚无定义} */$$

修改方法

$$S \rightarrow \{A_1 \cdot in := 1\} A_1 \{A_2 \cdot in := 2\} A_2;$$
$$A \rightarrow a \quad \{ \text{print}(A \cdot in) \}$$

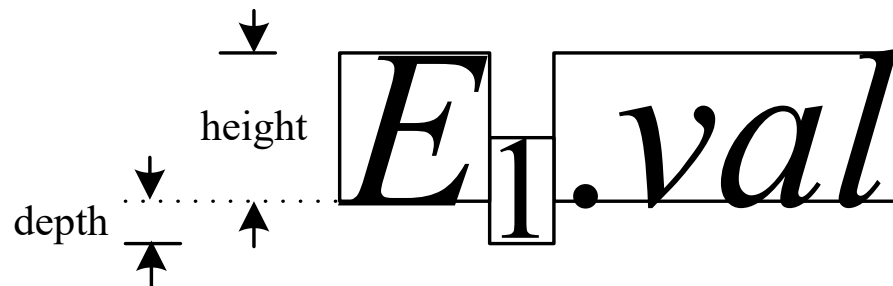
6.4 翻译模式

例6.11 从 L -属性制导定义建立一个满足上面要求的翻译模式。

使用文法产生的语言是数学排版语言EQN

$E \text{ sub } 1 \cdot val$

排版结果



B 表示盒子，下面给出对盒子排版的简单文法

- (1) $B \rightarrow B_1 B_2$ 代表两个相邻盒子的并列，且 B_1 位于 B_2 的左边。
- (2) $B \rightarrow B_1 \text{ sub } B_2$ 代表盒子 B_1 后随下标盒子 B_2 ，下标盒子 B_2 以较小的字体和较低的位置出现。
- (3) $B \rightarrow (B_1)$ 代表一个由括号括起来的盒子 B_1 ，主要是为了对多个盒子或下标进行分组。
- (4) $B \rightarrow \text{text}$ 代表文本字符串 $B_1 \text{ sub } B_2 B_3$ 组成的串。

该文法是二义性的文法

将“并列”和“下标”看成是左结合的，并令“下标”的优先级高于“并列”的话，则可以对该文法所描述的语言进行自底向上的语法分析。

属性设置

(1) point size用于表示盒子中文本的尺寸。如果标准盒子的尺寸为 ps ，则下标盒子的尺寸为 $0.7 \times ps$ 。属性 $B.ps$ 表示盒子 B 的尺寸，该属性是继承属性。

(2) 每个盒子都有一个基线(baseline)属性，该属性决定了盒子底部与基线之间的距离。

公式的高度取决于公式中各个盒子的值。

(3) $B.h$ 表示盒子 B 的高度，该属性是继承属性。
(4) $B.dp$ 表示盒子 B 的深度depth，该属性也是继承属性。

公式的深度取决于公式中各个盒子的深度。

表6.7 对盒子进行排版的语法制导定义

产生式	语义规则
(1) $S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$ $S.dp := B.dp$
(2) $B \rightarrow B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := \max(B_1.ht, B_2.ht)$ $B.dp := \max(B_1.dp, B_2.dp)$
(3) $B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps := B.ps$ $B_2.ps := 0.7 \times B.ps$ $B.ht := \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp := \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
(4) $B \rightarrow (B_1)$	$B_1.ps := B.ps$ $B.ht := B_1.ht$ $B.dp := B_1.dp$
(5) $B \rightarrow \text{text}$ 2020/11/2	$B.ht := \text{getheight}(B.ps, \text{text.lexval})$ $B.dp := \text{getdepth}(B.ps, \text{text.lexval})$

从表6.7构造的翻译模式

$S \rightarrow \{B.ps := 10\} B \{S.ht := B.ht; S.dp := B.dp\}$

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

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

$B \rightarrow \{B_1.ps := B.ps\} B_1 \text{sub} \{B_2.ps := 0.7 \times B.ps\}$

$B_2 \{B.ht := \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$

$B.dp := \max(B_1.dp, B_2.dp + 0.25 \times B.ps); \}$

$B \rightarrow (\{B_1.ps := B.ps\} B_1) \{B.ht := B_1.ht; B.dp := B_1.dp; \}$

$B \rightarrow \text{text} \{B.ht := \text{getheight}(B.ps, \text{text.lexval});$

$B.dp := \text{getdepth}(B.ps, \text{text.lexval}) \}$

6.4.2 S-属性定义的自底向上计算

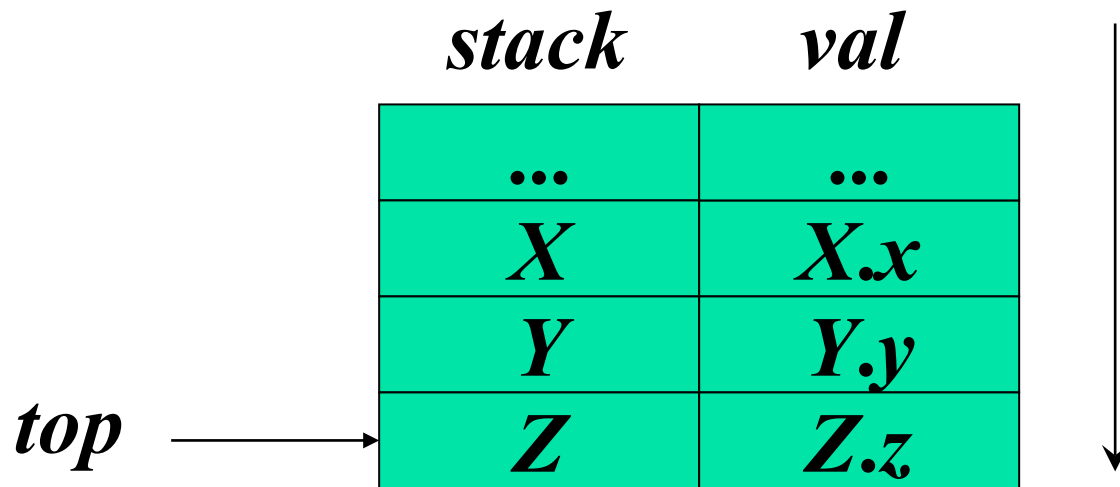
- 只含S-属性定义的实现比较简单
- 将每个语义规则放到相应产生式的末尾即可得到一个翻译模式（后缀式语法制导翻译模式）
- 对产生式归约时执行语义动作


6.4.2 S-属性定义的自底向上计算

- 可在LR分析过程中实现
- LR分析器可将文法符号的综合属性保存在栈中
- 进行归约时新的综合属性由栈内正在归约的产生式右部符号的属性来计算

6.4.2 S-属性定义的自底向上计算

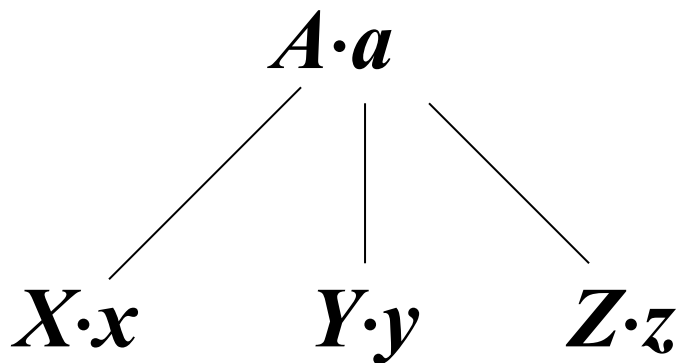
- 下图为一个带有综合属性值域的分析栈：

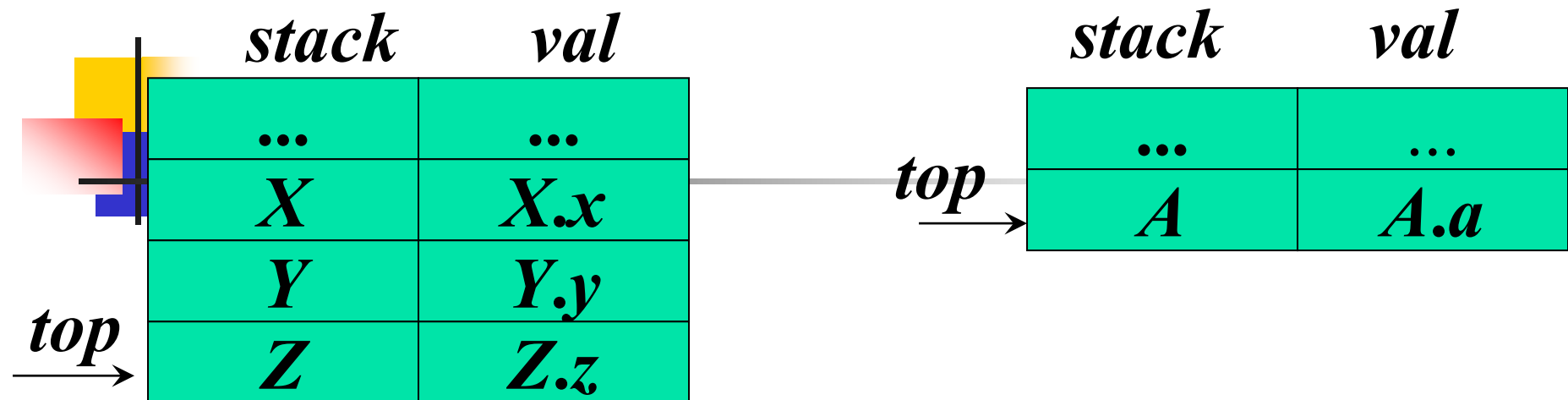



$$\forall A \rightarrow \alpha \quad b := f(c_1, c_2, \dots, c_k)$$

**b 是 A 的综合属性, $c_i (1 \leq i \leq k)$ 是 α 中符号的属性。
综合属性的值是在自底向上的分析过程中, 每次归约之前进行计算的。**

$$A \rightarrow XYZ \quad A \cdot a := f(X \cdot x, Y \cdot y, Z \cdot z)$$





实现时，将定义式 $A.a := f(X.x, Y.y, Z.z)$ (抽象) 变成 $stack[\textcolor{red}{ntop}].val := f(stack[top-2].val, stack[top-1].val, stack[top].val)$ (具体可执行代码)。

在执行代码段之前执行：

$\textcolor{red}{ntop} := top - r + 1$ —— $\textcolor{red}{r}$ 是句柄的长度

执行代码段后执行： $top := ntop;$

例6.14 用LR分析器实现台式计算器

——与表6.2对比

$L \rightarrow En\{\text{print}(\text{stack}[\text{top}-1].\text{val}); \text{top} := \text{top}-1;\}$

$E \rightarrow E_1 + T\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-2].\text{val} + \text{stack}[\text{top}].\text{val};$
 $\text{top} := \text{top}-2;\}$

$E \rightarrow T$

$T \rightarrow T_1 * F\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-2].\text{val} \times \text{stack}[\text{top}].\text{val};$
 $\text{top} := \text{top}-2;\}$

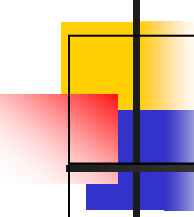
$T \rightarrow F$

$F \rightarrow (E)\{\text{stack}[\text{top}-2].\text{val} := \text{stack}[\text{top}-1].\text{val}; \text{top} := \text{top}-2;\}$

$F \rightarrow \text{digit}$

表6.8 翻译输入 $6+7*8n$ 上的移动序列

输入	<i>stack</i>	<i>val</i>	使用的产生式
$6+7*8n$	-	-	
$+7*8n$	6	6	
$+7*8n$	F	6	$F \rightarrow \text{digit}$
$+7*8n$	T	6	$T \rightarrow F$
$+7*8n$	E	6	$E \rightarrow T$
$7*8n$	$E+$	$6+$	
$*8n$	$E+7$	$6+7$	



$*8n$	$E+F$	$6+7$	$F \rightarrow \text{digit}$
$*8n$	$E+T$	$6+7$	$T \rightarrow F$
$8n$	$E+T^*$	$6+7^*$	
n	$E+T^*8$	$6+7^*8$	
n	$E+T^*F$	$6+7^*8$	$F \rightarrow \text{digit}$
n	$E+T$	$6+56$	$T \rightarrow T^*F$
n	E	62	$E \rightarrow E+T$
	En	62	
	L	62	$L \rightarrow En$

6.4.2 S-属性定义的自底向上计算

小结

采用自底向上分析，例如 LR 分析，首先给出 S -属性定义，然后，把 S -属性定义变成可执行的代码段，这就构成了翻译程序。

随着语法分析的进行，归约前调用相应的语义子程序，完成翻译的任务。

6.4.3 L-属性定义的自顶向下翻译

- **用翻译模式构造自顶向下的翻译。**

1. 从翻译模式中消除左递归

对于一个翻译模式，若采用自顶向下分析，必须消除左递归和提取左公因子，在改写基础文法时考虑属性值的计算。

为了方便讨论，这里只考虑L-属性定义只包含综合属性的情况，并且已经转化为翻译模式。



6.4.3 L-属性定义的自顶向下翻译

- 只有简单语义动作时的左递归消除：将语义动作看作终结符

只有简单语义动作时的左递归消除

- 例6.15 考虑如下将中缀表达式翻译为后缀表达式的翻译模式中的两个产生式：

$$E \rightarrow E_1 + T \text{ \{print(' + '); \}}$$

$$E \rightarrow T$$



$$E \rightarrow TR$$

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

$$R \rightarrow \varepsilon$$



S-属性定义的左递归消除

- 设有如下左递归翻译模式：

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

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

假设每个非终结符都有一个综合属性，用相应的小写字母表示， g 和 f 是任意函数。

- 消除左递归后，文法转换成

$$A \rightarrow X R$$

$$R \rightarrow Y R | \varepsilon$$

输入: XY_1Y_2

$$A \cdot a = g(g(f(X \cdot x), Y_1 \cdot y), Y_2 \cdot y)$$

$$A \cdot a = g(f(X \cdot x), Y_1 \cdot y)$$

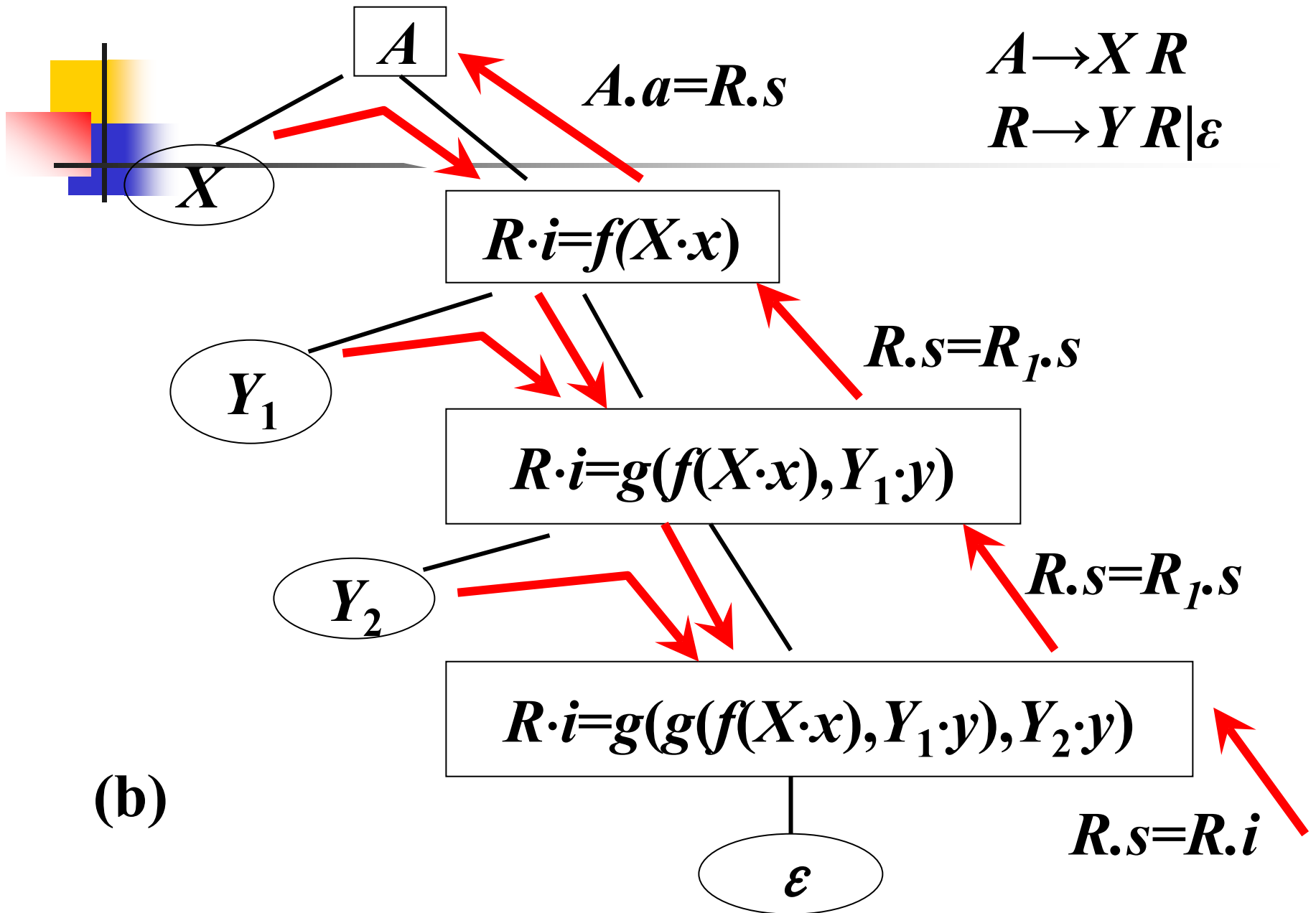
$$A \cdot a = f(X \cdot x)$$

X

Y_2

Y_1

$$\begin{aligned} A &\rightarrow A_1 Y \quad \{A.a := g(A_1.a, Y.y)\} \\ A &\rightarrow X \quad \{A.a := f(X.x)\} \end{aligned} \quad (a)$$



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

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

S-属性定义的左递归消除

- 再考虑语义动作，翻译模式变为：

$$A \rightarrow X \quad \{R.i := f(X.x)\}$$

$$R \quad \{A.a := R.s\}$$

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

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

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

引入继承属性 $R.i$ 来收集应用函数 g 的计算结果。其初始值为 $A.a := f(X.x)$

经过转换的翻译模式使用 R 的继承属性 i 和综合属性 s 。转换前后的结果是一样的，

引入综合属性 $R.s$ 在 R 结束生成 Y 时复制 $R.i$ 的值。

2. L -属性定义的递归下降翻译法

L -属性定义的递归下降翻译器的构造:

1. 对每个非终结符 A 构造一个函数 A , 将非终结符 A 的各个**继承属性**当作函数 A 的**形式参数**, 而将非终结符 A 的**综合属性集**当作函数 A 的**返回值**, 为了便于讨论, 假设每个非终结符只具有一个**综合属性**。
2. 在函数 A 的过程体中, 不仅要进行语法分析, 而且要处理相应的语义属性。函数 A 的代码首先根据当前输入确定用哪个产生式展开 A , 然后按照3中所给的方法对各产生式进行编码。



2. L -属性定义的递归下降翻译法

3. 与每个产生式对应的程序代码的工作过程为：按照从左到右的次序，依次对产生式右部的记号、非终结符和语义动作执行如下的动作：

- 1) 对带有综合属性 x 的符号 X ，将 x 的值保存在 $X.x$ 中，并生成一个函数调用来匹配 X ，然后继续读入下一个输入符号；
- 2) 对非终结符 B ，生成一个右部带有函数调用的赋值语句 $c := B(b_1, b_2, \dots, b_k)$ ，其中， b_1, b_2, \dots, b_k 是代表 B 的继承属性的变量， c 是代表 B 的综合属性的变量；
- 3) 对于语义动作，将其代码复制到语法分析器中，并将对属性的引用改为对相应变量的引用。



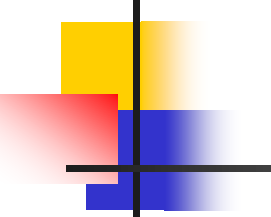
例 6.16

```
function  $T$ : $\uparrow$ syntax_tree_node;  
function  $T'$ ( $inh$ :  $\uparrow$ syntax_tree_node): $\uparrow$ syntax_tree_node;  
function  $F$ : $\uparrow$ syntax_tree_node;  
  
function  $T$ : $\uparrow$ syntax_tree_node; ( $T \rightarrow FT'$ )  
  var  $node, syn$ :  $\uparrow$ syntax_tree_node;  
begin  
   $node := F$ ;  
   $syn := T'(node)$ ;  
  return  $syn$   
end;
```

```

function  $T'$ (inh:  $\uparrow$ syntax_tree_node): $\uparrow$ syntax_tree_node;
var node,inh1,syn1:  $\uparrow$ syntax_tree_node; oplexeme:char;
begin if lookahead = '*' then begin
    /* 匹配产生式  $T' \rightarrow *FT'*$  */
    oplexeme := lexval;
    match('*'); node := F;
    inh1:=mknode('*', inh, node);
    syn1 :=  $T'$ (inh1); syn := syn1
end
else if lookahead = '/' then begin
    /* 匹配产生式  $T' \rightarrow /FT'*$  */
    oplexeme := lexval;
    match('/'); node := F;
    inh1:=mknode('/', inh, node);
    syn1 :=  $T'$ (inh1); syn := syn1
end else syn := inh; return syn end;

```



```
function  $F$ :  $\uparrow$ syntax_tree_node;  
var  $node$ :  $\uparrow$ syntax_tree_node;  
begin  if  $lookahead = '('$  then begin  
        /* 匹配产生式  $F \rightarrow (E)$  */  
         $match('(')$ ;   $node := E$ ;  
         $match('')$   
    end  
    else if  $lookahead = id$  then begin  
        /* 匹配产生式  $F \rightarrow id$  */  
         $match(id)$ ;   $node := mkleaf(id, id.entry)$   end  
    else if  $lookahead = num$  then begin  
        /* 匹配产生式  $F \rightarrow num$  */  
         $match(num)$ ;   $node := mkleaf(num, num.val)$   
    end  return  $node$   
end;
```



3. L -属性定义的 $LL(1)$ 翻译法

- 预先在源文法中的相应位置上嵌入语义动作符号(每个对应一个语义子程序), 用于提示语法分析程序, 当分析到达这些位置时应调用相应的语义子程序。
- 带有语义动作符号的文法又叫翻译文法。



3. L -属性定义的 $LL(1)$ 翻译法

■ 与递归子程序法的区别与联系

- 都是在扫描过程中进行产生式的推导，同时在适当的地方加入语义动作。
- 递归子程序法将语义动作融入分析程序； $LL(1)$ 分析法则由语义子程序完成相应的翻译。
- 递归子程序法隐式地使用语义栈； $LL(1)$ 分析法则用显式的语义栈（程序自身控制对栈的操作）。

例6.17

- 对于图6.11的翻译模式，设置两个栈，一个是分析栈，一个是语义栈。

$$(1) T \rightarrow F\{e1\}T'\{e2\}$$

$$(2) T' \rightarrow *F\{e3\}T_1'\{e4\}$$

$$(3) T' \rightarrow /F\{e5\}T_1'\{e4\}$$

$$(4) T' \rightarrow \varepsilon\{e6\}$$

$$(5) F \rightarrow (E)\{e7\}$$

$$(6) F \rightarrow \text{id}\{e8\}$$

$$(7) F \rightarrow \text{num}\{e9\}$$

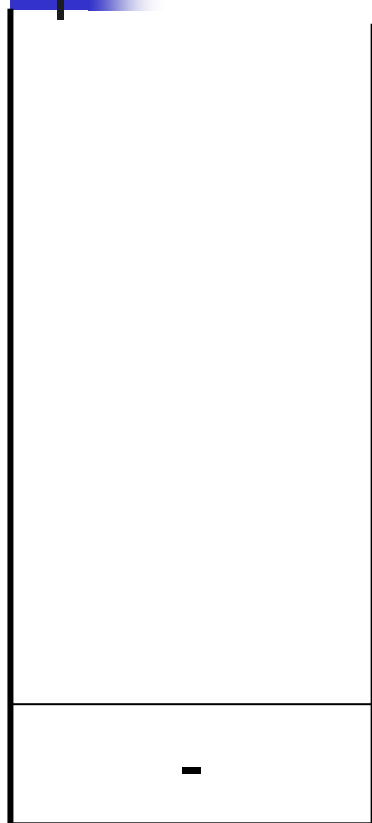
例6.17 对输入串 $3*x/y$ 的翻译

输入串 $3*x/y\#$

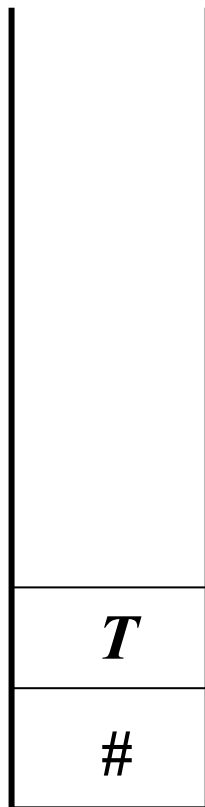


语法分析动作和语义操作

1. 初始化

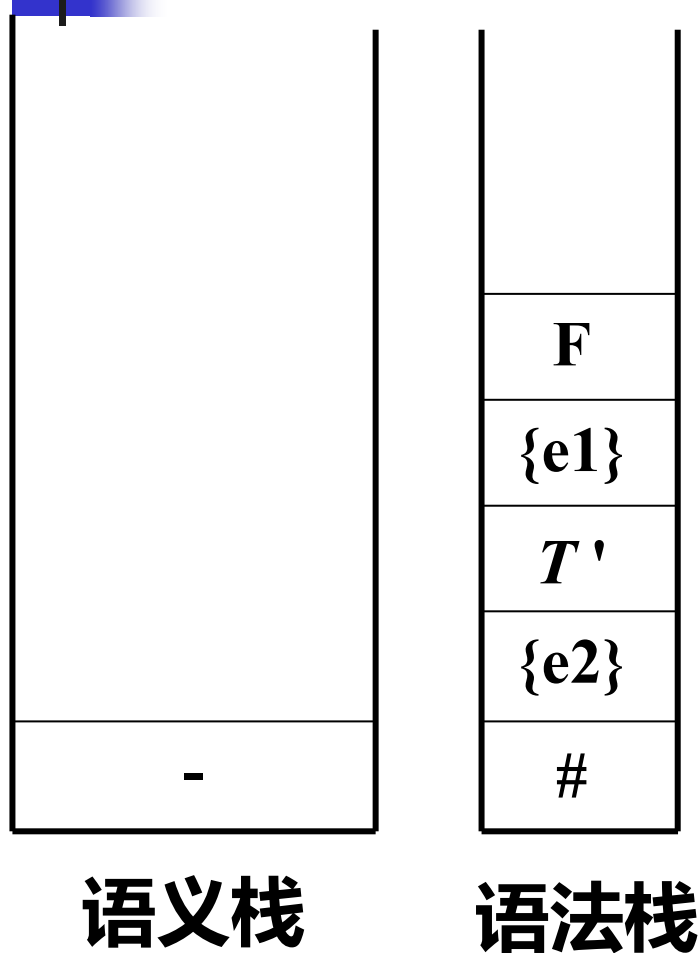


语义栈



语法栈

例6.17 对输入串 $3*x/y$ 的翻译



输入串 $3*x/y\#$



语法分析动作和语义操作

2. 选产生式①的右部进栈

$T \rightarrow F\{e1\}T'\{e2\}$

例6.17 对输入串 $3*x/y$ 的翻译

$F.node$
-

语义栈

{e9}
{e1}
T'
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

3.选择产生式(7), num_3 不进栈, 调用{e9}, 调用{e9}后, 叶结点 $F.node$ 被压入语义栈

$F \rightarrow num \{e9\}$

e9

$F.node = mkleaf(num, num.val)$

push $F.node$

例6.17 对输入串 $3*x/y\#$ 的翻译

$T'.inh$
-

语义栈

$\{e1\}$
T'
$\{e2\}$
$\#$

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

4. 执行动作 $\{e1\}$, $F.node$ 出栈,

$T'.inh$ 被压入栈

$e1$

pop $F.node$

$T'.inh := F.node$

push $T'.inh$

例6.17 对输入串 $3*x/y\#$ 的翻译

$T'.inh$
-

语义栈

F
$\{e3\}$
T'
$\{e4\}$
$\{e2\}$
$\#$

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

5.选择产生式(2),*不进栈

$T' \rightarrow *F\{e3\}T_1'\{e4\}$

例6.17 对输入串 $3*x/y$ 的翻译

$F.node$
$T'.inh$
-

语义栈

{e8}
{e3}
T'
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

6.选择产生式(6), id_x 不进栈, 调用 {e8}, 调用 {e8} 后, 叶结点 $F.node$ 被压入语义栈

$F \rightarrow id \{e8\}$

e8

$F.node := mkleaf(id, id.entry)$

push $F.node$

例6.17 对输入串 $3*x/y$ 的翻译

$T'.inh$
-

语义栈

{e3}
T'
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

7. 执行动作{e3}, $F.node$ 和 $T'.inh$ 均被弹出栈, 新的 $T'.inh$ 被压入栈

e3

pop $F.node$

pop $T'.inh$

$T'.inh := \text{mknode}('*', T'.inh, F.node)$

push $T'.inh$

例6.17 对输入串 $3*x/y\#$ 的翻译

$T'.inh$
-

语义栈

F
{e5}
T'
{e4}
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

8.选择产生式(3), /不进栈

$T' \rightarrow / F \{e5\} T_1' \{e4\}$

例6.17 对输入串 $3*x/y$ 的翻译

$F.node$
$T'.inh$
-

语义栈

{e8}
{e5}
T'
{e4}
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

9.选择产生式(6), id_y 不进栈, 调用 {e8}, 调用 {e8} 后, 叶结点 $F.node$ 被压入语义栈

$F \rightarrow id \{e8\}$

e8

$F.node := mkleaf(id, id.entry)$

push $F.node$

例6.17 对输入串 $3*x/y$ 的翻译

$T'.inh$
-

语义栈

{e5}
T'
{e4}
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

10. 执行动作{e5}, $F.node$ 和 $T'.inh$ 均被弹出栈, 新的 $T'.inh$ 被压入栈
e3

pop $F.node$

pop $T'.inh$

$T'.inh := \text{mknode}('/', T'.inh, F.node)$

push $T'.inh$

例6.17 对输入串 $3*x/y$ 的翻译

$T'.syn$
-

语义栈

{e6}
{e4}
{e4}
{e2}
#

语法栈

输入串 $3*x/y\#$



语法分析动作和语义操作

11.选择产生式(4), $T'.inh$ 被弹出栈, $T'.syn$ 被压入栈

$T' \rightarrow \text{epsilon } \{e6\}$

$e6$

pop $T'.inh$

$T'.syn := T'.inh$

push $T'.syn$

例6.17 对输入串 $3*x/y$ 的翻译

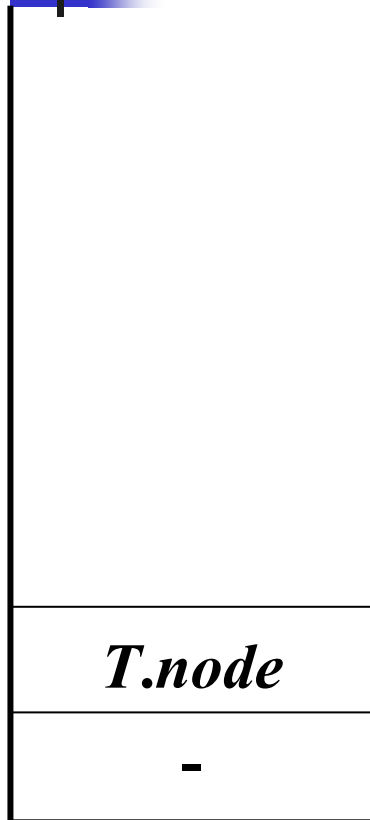
输入串 $3*x/y\#$



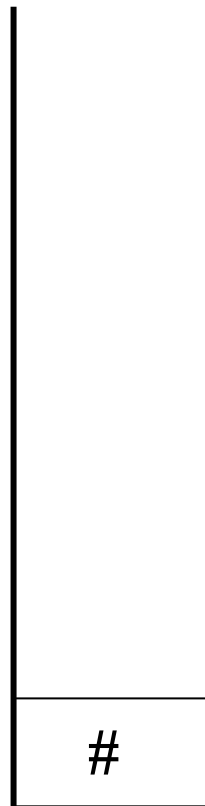
语法分析动作和语义操作

12.依次执行动作

$\{e6\}, \{e4\}, \{e4\}, \{e2\}$, 最终语义栈中只有 $T.node$, 代表 $3*x/y$ 的语法树的根结点



语义栈



语法栈



6.4.4 L-属性定义的自底向上翻译

- 自底向上的分析过程中实现S-属性定义和自顶向下的分析过程中实现L-属性定义都比较直观
- 问题：如何在自底向上的分析过程中实现L-属性文法
- 关键：**如何自底向上地计算继承属性**



6.4.4 L-属性定义的自底向上翻译

- 给定产生式 $A \rightarrow BC$, 继承属性 $B.inh$ 依赖于 A 的继承属性 $B.inh = f(A.inh, \dots)$
- 当栈顶规约出 B 时, 如何获得 $A.inh$ 的值

6.4.4 L-属性定义的自底向上翻译(续)

- (1) 将计算继承属性的动作嵌入在非终结符的前面，而将计算综合属性的动作放在产生式的末尾。
- (2) 在每个**嵌入动作**处引入一个**标记性非终结符**。不同位置所对应的标记是不同的，每个标记性非终结符 M 都有一个**形如 $M \rightarrow \varepsilon$ 的产生式**。

6.4.4 L-属性定义的自底向上翻译(续)

- (3) 如果标记性非终结符 M 取代了某个产生式 $A \rightarrow \alpha \{a\} \beta$ 中的动作 a , 则按如下方式将 a 修改为 a' , 并将动作 $\{a'\}$ 放在产生式 $M \rightarrow \varepsilon$ 的末尾。
- ① 为 M 设置继承属性来复制动作 a 所需要的 A 或 a 中符号的继承属性;
 - ② 以与动作 a 相同的方式计算属性, 只不过要将这些属性置为 M 的综合属性。

6.4.4 L-属性定义的自底向上翻译(续)

- 与 $M \rightarrow \varepsilon$ 相关联的语义动作可能需要用到没出现在该产生式中的文法符号的属性。
- 由于要在 LR 分析栈中实现所有的语义动作，所以在分析栈中 M 下面的某个已知位置总能找到所需的属性。

6.4.4 L-属性定义的自底向上翻译(续)

- 例如, 给定形如 $A \rightarrow BC$ 的产生式, B 的继承属性 $B.inh$ 是从 A 的继承属性 $A.inh$ 按照公式 $B.inh := f(A.inh)$ 来计算的, 亦即翻译模式可能包含如下片断:

$$A \rightarrow \{B.inh := f(A.inh);\}BC$$

6.4.4 L-属性定义的自底向上翻译(续)

- 根据上面的论述，为了在自底向上的分析过程中计算 $B.inh := f(A.inh)$ ，需要引入一个标记性非终结符 M ，并为其设置一个继承属性 $M.inh$ 用来复制 A 的继承属性，而且还要用 M 的综合属性 $M.syn$ 代替 $B.inh$

6.4.4 L-属性定义的自底向上翻译(续)

- 该翻译模式片段将被修改为如下形式:

$$A \rightarrow MBC$$

$$M \rightarrow \varepsilon \{M.inh := A.inh; M.syn := f(M.inh)\}$$

6.4.4 L-属性定义的自底向上翻译(续)

- 注意, 执行 M 的语义规则时, $A.inh$ 是不可用的
- 实际实现时会把每个非终结符 X 的继承属性都放在堆栈中紧靠在 X 将被归约出来的位置之下
- 当将 ε 归约为 M 时, $A.inh$ 恰好就在它的下面。
- 随 M 保存在栈中的 $M.syn$ 的值, 也就是 $B.inh$ 的值, 亦将被放在紧靠在 B 将被归约出来的位置之下, 需要时同样可用。

6.4.4 L-属性定义的自底向上翻译(续)

- 给定 $A \rightarrow M_1 X_1 \dots M_n X_n$, 下面的化简可以减少标记性非终结符的个数:

(1) 如果 X_j 没有继承属性, 则不需要使用标记 M_j 。当然, 如果省略了 M_j , 属性在栈中的预期位置就会改变, 但是分析器可以很容易地适应这种变化。

(2) 如果 $X_1.inh$ 存在, 但它是由复制规则 $X_1.inh := A.inh$ 计算的, 此时可以省略 M_1 。因为 $A.inh$ 存放在栈中紧挨在 X_1 下面的地方, 所以该值也可同时作为 $X_1.inh$ 的值。

6.4.4 L-属性定义的自底向上翻译(续)

例6.18 数学排版语言EQN

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

$B \quad \{S.ht := B.ht; S.dp := B.dp\}$

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

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

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

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

B_1

sub $\{B_2.ps := 0.7 \times B.ps\}$

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

$B.dp := \max(B_1.dp, B_2.dp)$

$B \rightarrow \text{text}\{B.ht := \text{getheight}(B.ps, \text{text.lexval});$

$B.dp := \text{getdepth}(B.ps, \text{text.lexval})\}$

保证在 B 的子树被归约时, $B.ps$ 的值出现在分析栈中的已知位置

归约 B_1 之前, $B.ps$ 可以在栈中找到, 所以 $B_1.ps := B.ps$ 可以省略。但归约 B_2 之前, 无法确定其前有几个 B_1 , 因此, 无法预测 $B.ps$ 在栈中的位置。

6.4.4 L-属性定义的自底向上翻译(续)

产生式	语义规则
$S \rightarrow LB$	$B.ps := L.syn; S.ht := B.ht; S.dp := B.dp$
$L \rightarrow \varepsilon$	$L.syn := 10$
$B \rightarrow B_1 MB_2$	$B_1.ps := B.ps; M.inh := B.ps;$ $B_2.ps := M.syn; B.ht := \max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.syn := M.inh$
$B \rightarrow B_1 \text{ sub } NB_2$	$B_1.ps := B.ps; N.inh := B.ps;$ $B_2.ps := N.syn; B.ht := \text{disp}(B_1.ht, B_2.ht);$ $B.dp := \max(B_1.dp, B_2.dp)$
$N \rightarrow \varepsilon$	$N.syn := 0.7 \times N.inh$
$B \rightarrow \text{text}$	$B.ht := \text{getheight}(B.ps, \text{text.lexval});$ $B.dp := \text{getdepth}(B.ps, \text{text.lexval})$

6.4.4 L-属性定义的自底向上翻译(续)

- 由于存在一个继承属性(ps)和两个综合属性(ht, dp), 所以语义栈 val 需要被扩展为 val_1 、 val_2 和 val_3
 - val_1 用于保存继承属性 ps 的值
 - val_2 和 val_3 分别用于保存综合属性 ht 和 dp 的值
 - 假设分析栈仍为 $stack$
 - top 和 $ntop$ 分别是归约前和归约后栈顶的下标。

6.4.4 L-属性定义的自底向上翻译(续)

产生式	代码段
$S \rightarrow LB$	$stack[ntop].val_2 := stack[top].val_2$ $stack[ntop].val_3 := stack[top].val_3$
$L \rightarrow \varepsilon$	$stack[ntop].val_1 := 10$
$B \rightarrow B_1 MB_2$	$stack[ntop].val_2 := \max(stack[top-2].val_2, stack[top].val_2)$ $stack[ntop].val_3 := \max(stack[top-2].val_3, stack[top].val_3)$
$M \rightarrow \varepsilon$	$stack[ntop].val_1 := stack[top-1].val_1$
$B \rightarrow B_1 \text{ sub } NB_2$	$stack[ntop].val_2 :=$ $\max(stack[top-3].val_2, stack[top].val_2 - 0.25 \times stack[top-4].val_1)$ $stack[ntop].val_3 :=$ $\max(stack[top-3].val_3, stack[top].val_3 + 0.25 \times stack[top-4].val_1)$
$N \rightarrow \varepsilon$	$stack[ntop].val_1 := 0.7 \times stack[top-2].val_1$
$B \rightarrow \text{text}$	$stack[ntop].val_2 := \text{getheight}(stack[top-1].val_1, \text{text.lexval})$ $stack[ntop].val_3 := \text{getdepth}(stack[top-1].val_1, \text{text.lexval})$



本章小结

- **语法分析中进行静态语义检查和中间代码生成的技术称为语法制导翻译技术。**
- **为了通过将语义属性关联到文法符号、将语义规则关联到产生式，有效地将语法和语义关联起来，人们引入了语法制导定义。没有副作用的语法制导定义又称为属性文法。**



本章小结

- 为相应的语法成分设置表示语义的属性，属性的值是可以计算的，根据属性值计算的关联关系，将其分成综合属性和继承属性，根据属性文法中所含的属性将属性文法分成S-属性文法和L-属性文法。
- 如果不仅将语义属性关联到文法符号、将语义规则关联到产生式，而且还通过将语义动作嵌入到产生式的适当位置来表达该语义动作的执行时机，这就是翻译模式。翻译模式给语义分析的实现提供了更好的支持。



本章小结

- **注释分析树和相应的依赖图是属性值的关联关系和计算顺序的表达形式，语义关系可以使用抽象语法树表示。**
- **依据语法分析方法有自底向上的和自顶向下的，语法制导翻译既可以按照自底向上的策略进行，也可以按照自顶向下的策略进行。**