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。

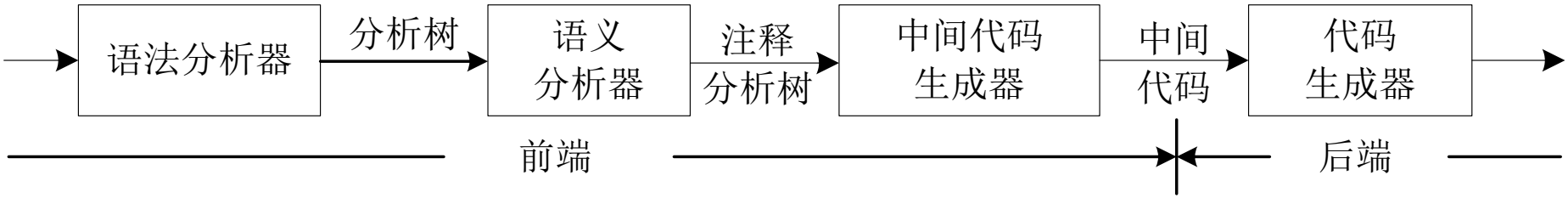


图6.1 编译器前端的逻辑结构

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，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：控制流转向的目标地址是否合法；
  - 惟一性检查：对象是否被重复定义；
  - 关联名检查：同一名字的多次特定出现是否一致。
- 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**。

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。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)。

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(如传送或处理语义信息、查填符号表、计算值、生成中间代码等)。
  - 语义信息的获取和加工是和语法分析同时进行的，而且这些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(对应分析树结点的数据结构)，主要用于描述语法结构的语义。
  - 一个变量的属性有类型、层次、存储地址等
  - 表达式的属性有类型、值等。

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。
  - $E \rightarrow E_1 + E_2$　　$E.val := E_1.val + E_2.val$
- 语法结构具有规定的语义
- 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？

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
    - $E \rightarrow E_1 + T \quad \{E.val := E_1.val + T.val\}$
    - $T \rightarrow T_1 * F \quad \{T.val := T_1.val * F.val\}$
    - $F \rightarrow \text{digit} \quad \{F.val := \text{digit}.lexval\}$
- 适宜在完成归约的时候进行

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
  - $D \rightarrow T \{ L.inh := T.type \} L$
  - $T \rightarrow \text{int} \{ T.type := \text{integer} \}$
  - $T \rightarrow \text{real} \{ T.type := \text{real} \}$
  - $L \rightarrow \{ L_1.inh := L.inh \} L_1, \text{id} \{ addtype(\text{id}.entry, L.inh) \}$
  - $L \rightarrow \text{id} \{ addtype(\text{id}.entry, L.inh) \}$

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下文无关文法
  - 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
- 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的。
- 属性通常被保存在分析树的相关节点中

概念术语

- 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

语法制导定义的形式

- 在一个**语法制导定义**中，$\forall A \to \alpha \in P$都有与之相关联的一套语义规则，规则形式为

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

f是一个函数，而且或者

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

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

这两种情况下，都说属性b依赖于属性$c_1, c_2, \ldots, c_k$

例6.1 台式计算器的语法制导定义

产生式	语义规则
$L \rightarrow En$	$print(E \cdot val)$(可看作是L的虚属性)
$E \rightarrow E_1 + T$	$E \cdot val := E_1 \cdot val + T \cdot val$
$E \rightarrow T$	$E \cdot val := T \cdot val$
$T \rightarrow T_1 * F$	$T \cdot val := T_1 \cdot val * F \cdot val$
$T \rightarrow F$	$T \cdot val := F \cdot val$
$F \rightarrow (E)$	$F \cdot val := E \cdot val$
$F \rightarrow \text{digit}$	$F \cdot val := \text{digit} \cdot lexval$

S-属性定义

- 只含综合属性的语法制导定义称为*S*-属性定义
- 对于*S*-属性定义，通常使用自底向上的分析方法，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，从叶结点到根结点进行计算。
- 没有副作用的语法制导定义有时又称为属性文法，属性文法的语义规则单纯根据常数和其它属性的值来定义某个属性的值

继承属性

- 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，产生这种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3*5和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：

$$T \to FT' \quad T' \to *FT_1' \quad T' \to \varepsilon \quad F \to \text{digit}$$

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
$T \rightarrow FT'$	$T'.inh := F.val$ $T.val := T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh := T'.inh \times F.val$ $T'.syn := T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn := T'.inh$
$F \rightarrow$ digit	$F.val :=$ digit.$lexval$

4*8*9的注释分析树

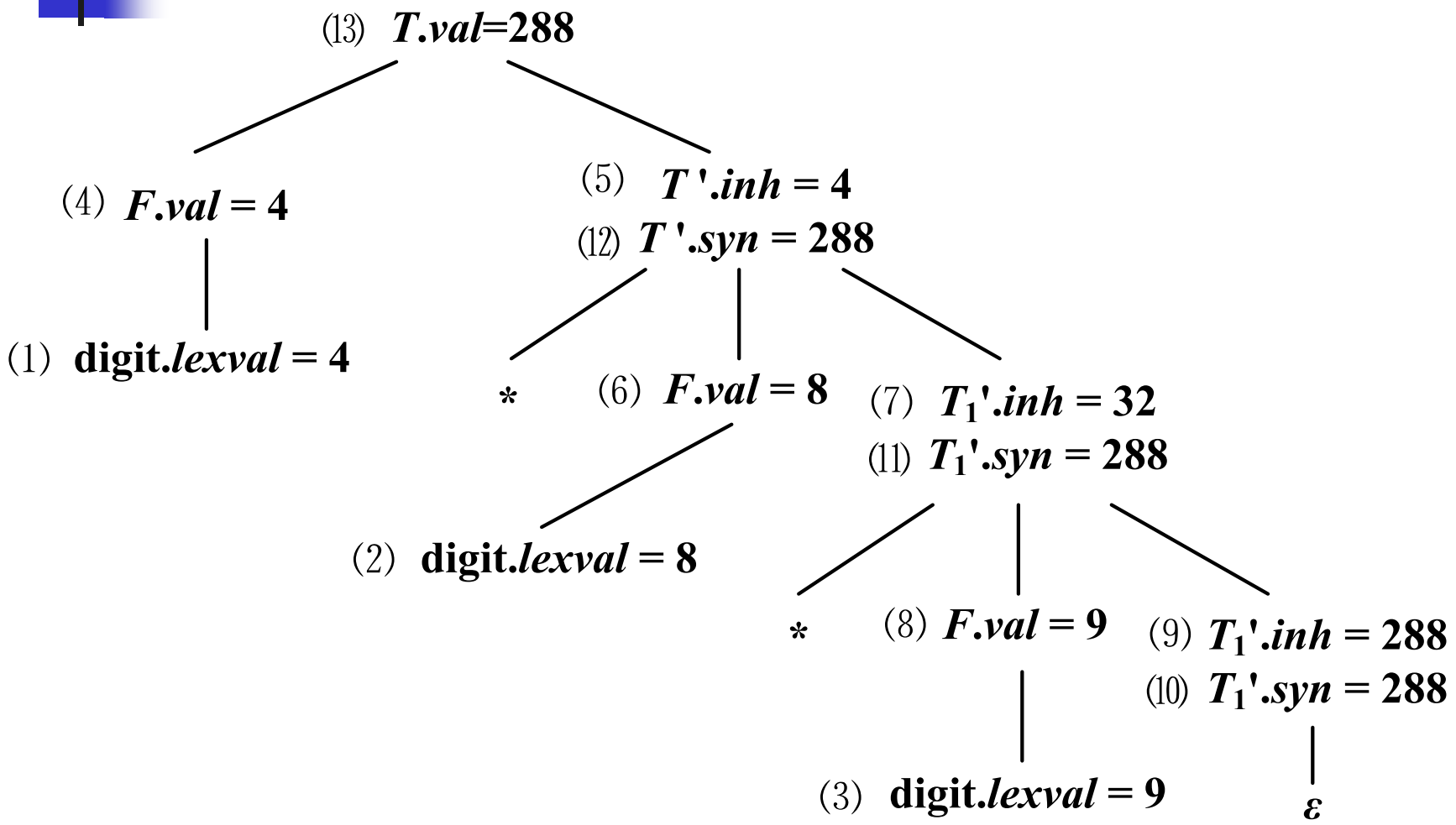

(13) T.val=288
(4) F.val = 4
(5) T '.inh = 4
(12) T '.syn = 288
(1) digit.lexval = 4
*
(6) F.val = 8
(7) T1'.inh = 32
(11) T1'.syn = 288
(2) digit.lexval = 8
*
(8) F.val = 9
(9) T1'.inh = 288
(10) T1'.syn = 288
(3) digit.lexval = 9
ε

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
- 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。
- 注释分析树只是给出了属性的值，而依赖图则可以帮助我们确定如何将这些属性值计算出来。

6.3.1 依赖图

- 所谓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，称为分析树的依赖图。
- 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。

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

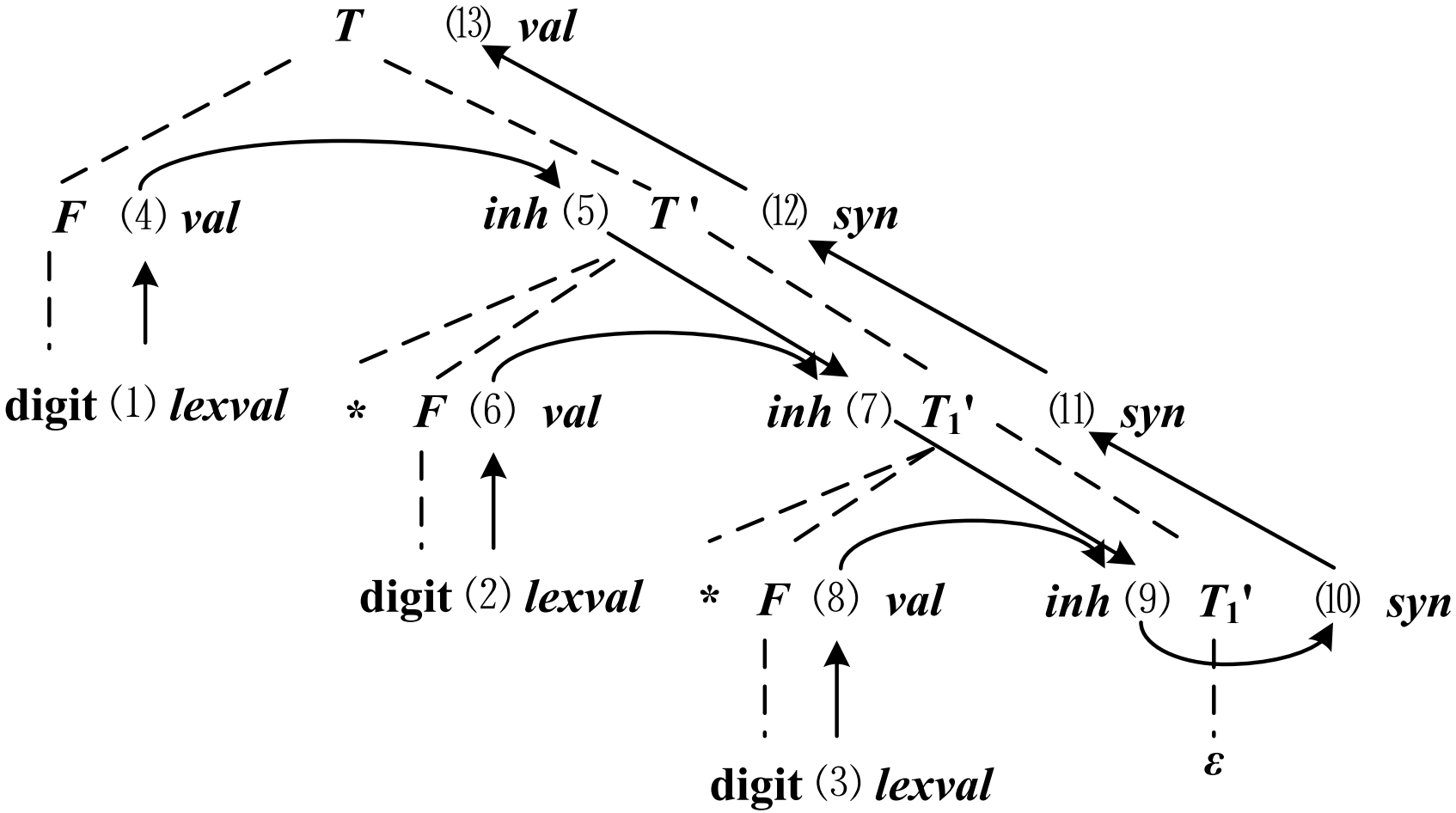
    for 节点n所用产生式对应的每条语义规则

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

      for $i := 1$ to k do

      构造一条从节点c_i到节点b的有向边;

例6.3 图6.3中注释分析树的依赖图



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，…，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。
  - 例6.4 图6.4的拓扑排序为：

    1，2，3，4，5，6，7，8，9，10，11，12，13

6.3.2 属性的计算顺序

- 根据拓扑排序得到的翻译程序
  - $a4:=4$　　$a5:=a4$　　$a6:=8$
  - $a7:=a5\times a6$　　$a8:=9$　　$a9:=a7\times a8$
  - $a10:=a9$　　$a11:=a10$　　$a12:=a11$
  - $a13:=a12$
- 上述属性计算方法又称为**分析树法**，这种方法在编译时需要显式地构造分析树和依赖图，所以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。
- 这种方法的优点是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。

计算语义规则的其他方法

- 基于规则的方法
  - 在构造编译器时，用手工或专门的工具来分析语义规则,确定属性值的计算顺序。
- 忽略语义规则的方法
  - 在分析过程中翻译，那么计算顺序由分析方法来确定而表面上与语义规则无关。这种方法限制了能被实现的语法制导定义的种类。
- 这两种方法都不必显式构造依赖图，因此时空效率更高。

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，又称为*S*-属性文法。
- 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，并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;
  - }

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$A \rightarrow X_1 X_2 \ldots X_n$，其右部符号$X_i(1 \le i \le n)$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、…、X_{i-1}的综合属性或继承属性；
  (3) X_i本身的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。
- L-属性定义又称为L-属性文法。

表6.3 L-属性定义示例

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

例6.7 不是L-属性定义的语法制导定义

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

语义规则$B.inh := f(C.c, A.syn)$定义了一个继承属性，所以整个语法制导定义就不是S-属性定义了。此外，虽然这条语义规则是合法的属性定义规则，但不满足L-属性定义的要求。这是因为：属性$B.inh$的定义中用到了属性$C.c$，而C在产生式的右部处在B的右边。虽然在L-属性定义中可以使用兄弟节点的属性来定义某个属性，但这些兄弟节点必须是左兄弟节点才行。因此，该语法制导定义也不是L-属性定义。

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—抽象语法树的构造

抽象语法树是表示程序层次结构的树，它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，也称作语法结构树，或结构树。

语法树是常用的一种**中间表示**形式。

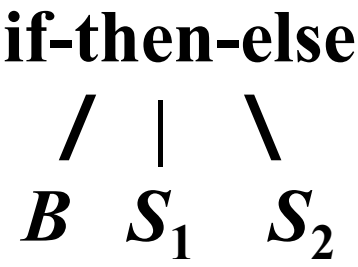
把语法分析和翻译分开。语法分析过程中完成翻译有许多优点，但也有一些不足：

1.适于语法分析的文法可能不完全反映语言成分的自然层次结构；

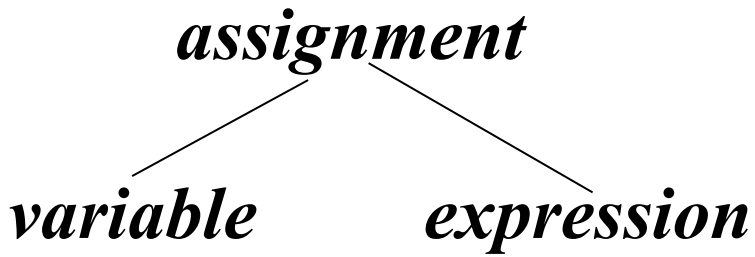
2. 由于语法分析方法的限制，对分析树结点的访问顺序和翻译需要的访问顺序不一致。

语法树

产生式$S \rightarrow$if B then S_1 else S_2的语法树



赋值语句的语法树



在语法树中，运算符号和关键字都不在叶结点，而是在内部结点中出现。

构造表达式的语法树

构造表达式的语法树使用的函数

1. *mknode*(*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。

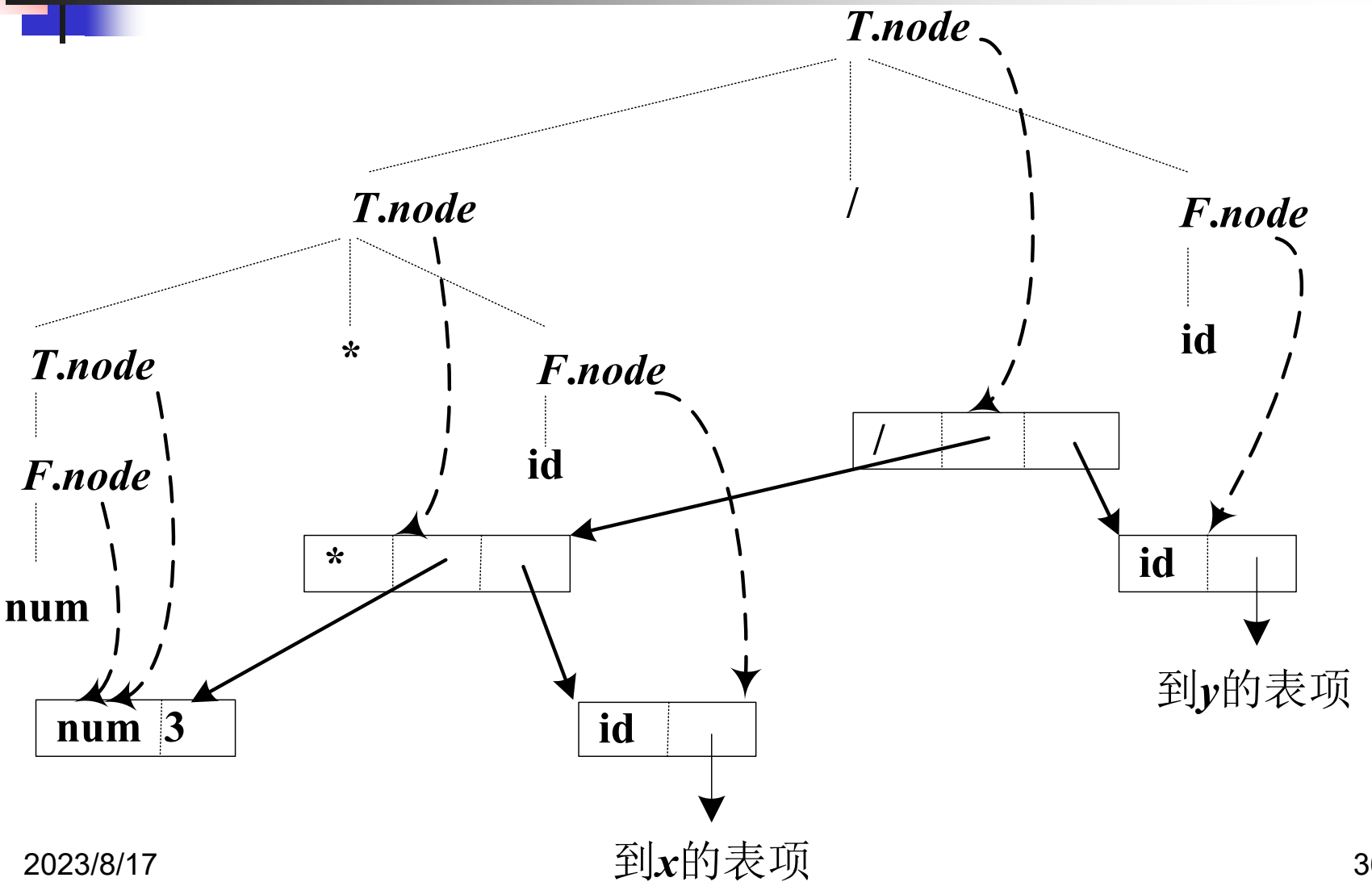
2． *mkleaf*(id, *entry*) 建立一个标记为id的标识符结点，其域*entry*是指向该标识符在符号表中相应表项的指针。

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。

构造表达式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	$F.node := mkleaf(\text{id}, \text{id}.entry)$
(6) $F \rightarrow$ num	$F.node := mkleaf(\text{num}, \text{num}.val)$

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

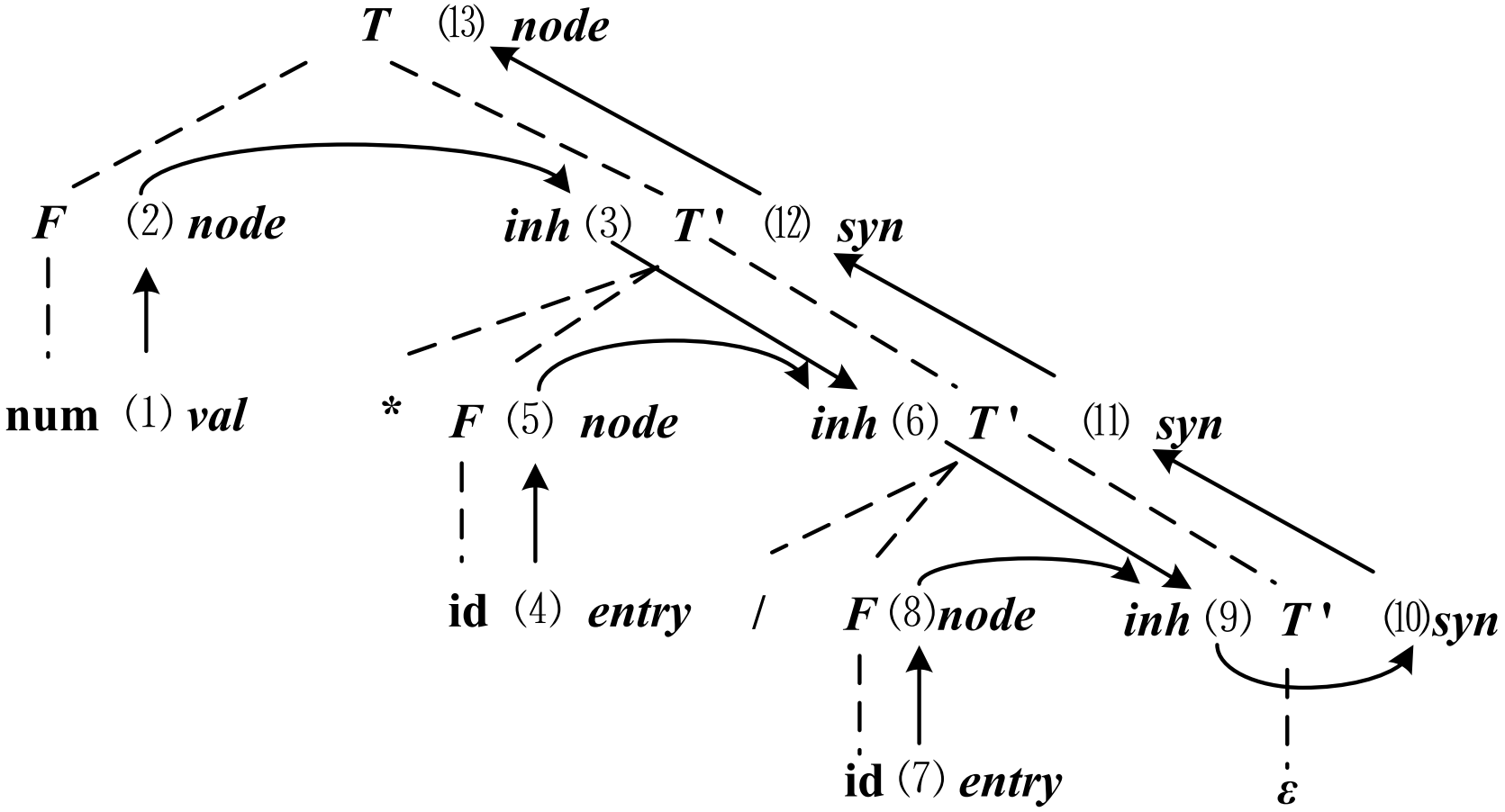
p_1:=*mkleaf*(num,3);
p_2:=*mkleaf*(id, *entry*-*x*);
p_3:=*mknode*('*', p_1, p_2);
p_4:=*mkleaf*(id, *entry*-*y*);
p_5:=*mknode*('/', p_3, p_4);

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	$F.node := mkleaf(\text{id}, \text{id}.entry)$
(7) $F \rightarrow$ num	$F.node := mkleaf(\text{num}, \text{num}.val)$

根据表6.6的语法制导定义构造的语法树



6.4 翻译模式

- 定义

翻译模式是语法制导定义的一种便于实现的书写形式。其中属性与文法符号相关联，语义规则或语义动作用花括号｛｝括起来，并可被插入到产生式右部的任何合适的位置上。

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顺序。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：

$E \rightarrow TR$

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

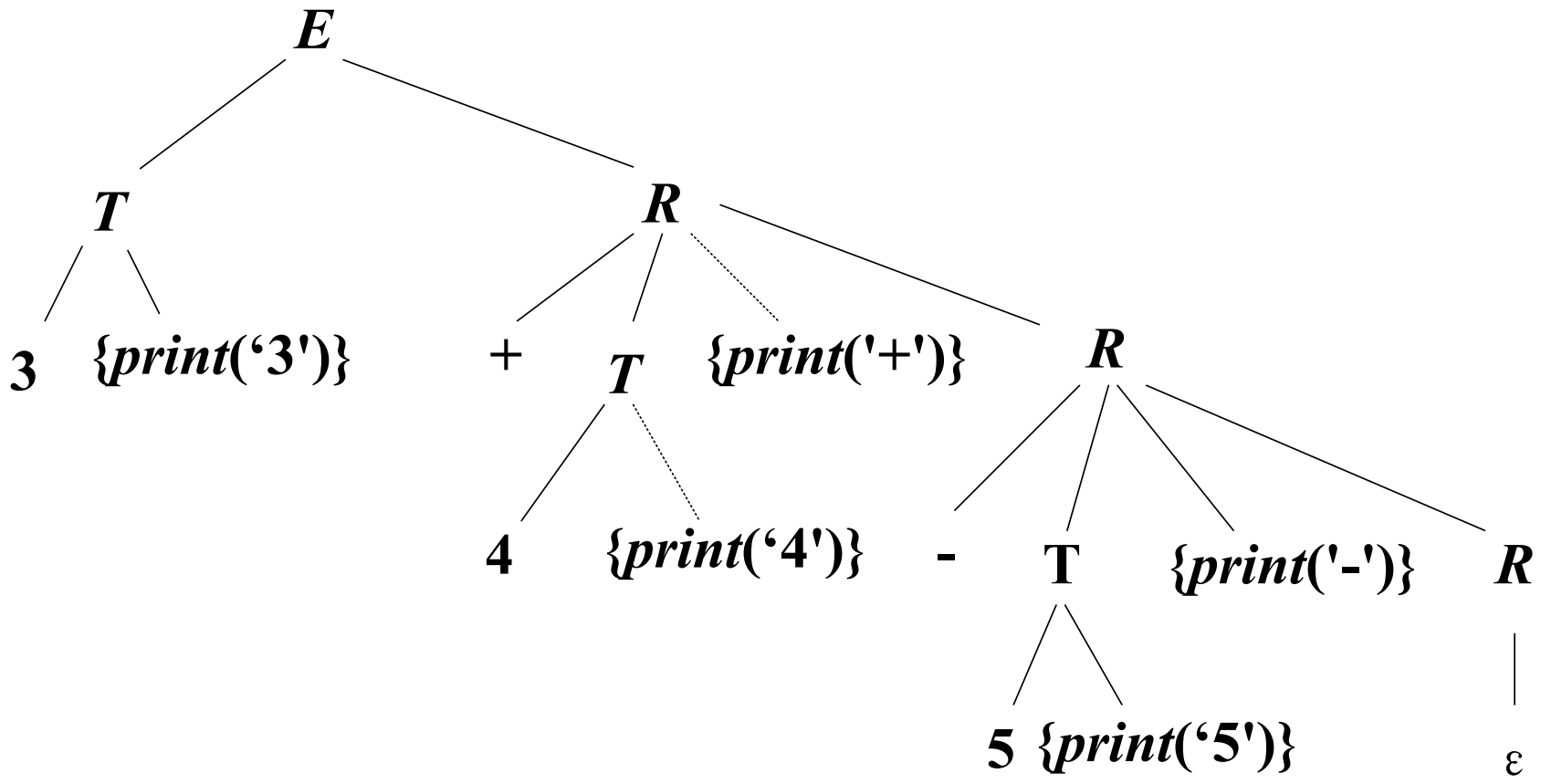
$T \rightarrow \text{num}\{print(\text{num}.val)\}$

把语义动作看成终结符号，输入3+4-5,其分析树如图6.8，当按深度优先遍历它，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



翻译模式的设计

——根据语法制导定义

- **前提——语法制导定义是L-属性定义**

保证语义动作不会引用还没计算出来的属性值

1. 只需要综合属性的情况

为每一个语义规则建立一个包含赋值的动作，并把该动作放在相应的产生式右部的末尾。

例如： $T \to T_1 * F \qquad T \cdot val := T_1 \cdot val * F \cdot val$

转换成：

$T \to T_1 * F \{T \cdot val := T_1 \cdot val * F \cdot val\}$

翻译模式的设计

——根据语法制导定义

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端的末尾。

下面的翻译模式不满足要求：

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

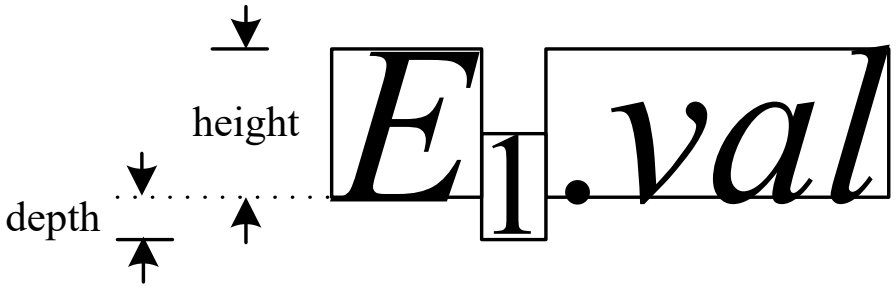
$A \rightarrow a$ $\{ print(A \cdot in) \}$ /*$A.in$尚无定义*/

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 sub 1$\cdot val$

编排结果



B表示盒子

(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) $B \rightarrow B_1$ 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。在EQN中，使用花括号进行分组，此处使用圆括号是为了避免跟语义动作外面的花括号产生冲突。

(4) $B \rightarrow$text代表文本字符串，即任意字符组成的串。

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(以点来计算，也就是字号)。如果标准盒子的尺寸为p，则下标盒子的尺寸为$0.7\times p$。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)，用来表示每个文本底部的垂直位置。

(3) height用来表示从盒子的顶部到基线的距离。属性$B.ht$表示盒子B的高度height，该属性是综合属性。

(4) depth用来表示从基线到盒子底部的距离。用属性$B.dp$表示盒子B的深度depth，该属性也是综合属性。

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)$B \rightarrow B_1$ 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	$B.ht := getheight(B.ps, \text{text}.lexval)$ $B.dp := getdepth(B.ps, \text{text}.lexval)$

从表6.7构造的翻译模式

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

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

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

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

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

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

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

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

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

从表6.6构造的翻译模式

$T \rightarrow F\ \{T'.inh := F.node\}\ T'\ \{T.node := T'.syn\}$

$T' \rightarrow *F\ \{T_1'.inh := mknode('*', T'.inh, F.node)\}$

$\quad T_1'\ \{T'.syn := T_1'.syn\}$

$T' \rightarrow /F\ \{T_1'.inh := mknode('/', T'.inh, F.node)\}$

$\quad T_1'\ \{T'.syn := T_1'.syn\}$

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

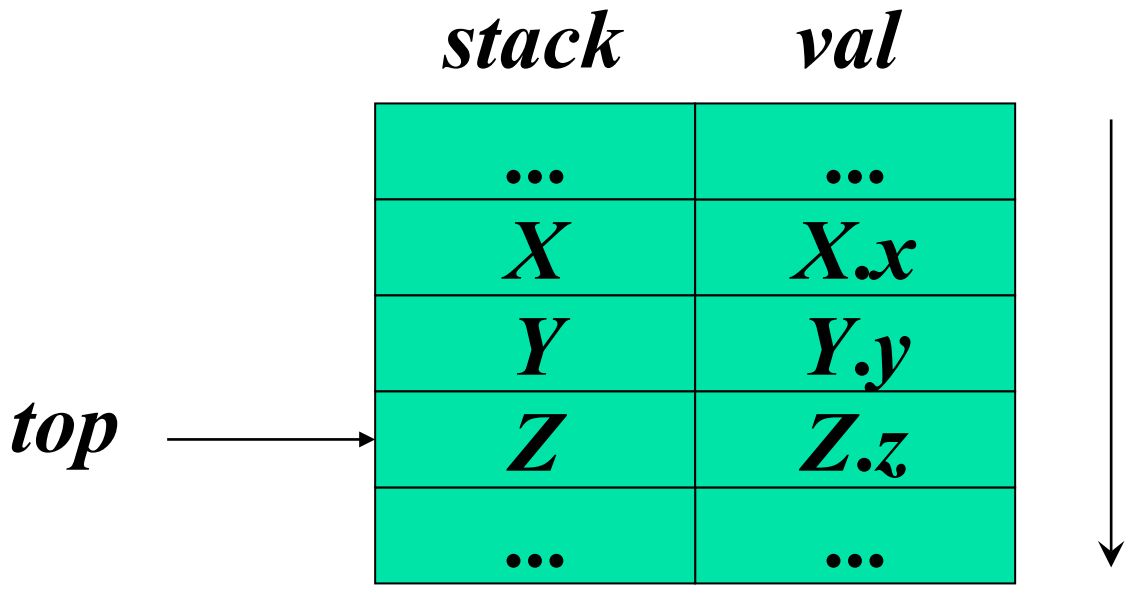
$F \rightarrow (E)\ \{F.node := E.node\}$

$F \rightarrow \mathbf{id}\ \{F.node := mkleaf(\mathbf{id}, \mathbf{id}.entry)\}$

$F \rightarrow \mathbf{num}\ \{F.node := mkleaf(\mathbf{num}, \mathbf{num}.val)\}$

6.4.2 S-属性定义的自底向上计算

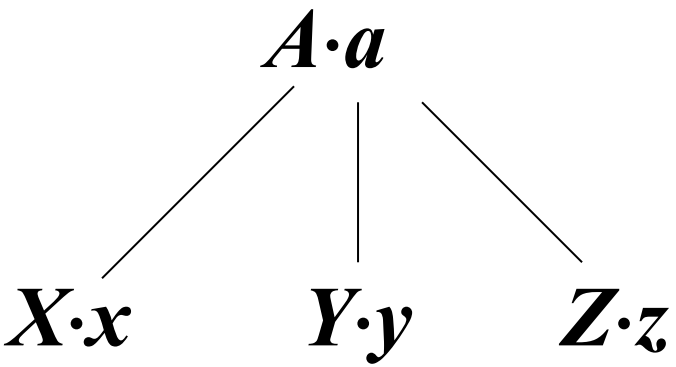
在分析栈中使用一个附加的域来存放综合属性值。下图为一个带有综合属性值域的分析栈：

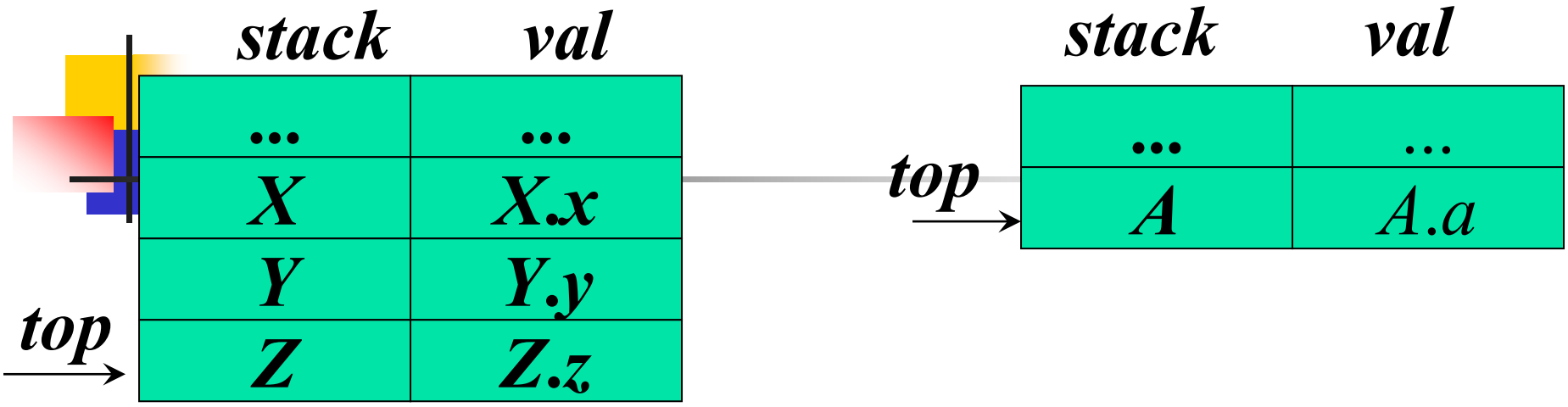


$\forall A \rightarrow \alpha \qquad b := f(c_1, c_2, \ldots, c_k)$

b是A的综合属性，$c_i (1 \le i \le k)$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[*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：

ntop:=*top*-*r*+1 ——*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

$L \rightarrow E$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

表6.8 翻译输入6+7*8n上的移动序列

输入	*state*	*val*	使用的产生式
6+7*8n	-	-	
+7*8*n*	6	6	
+7*8n	F	6	F→digit
+7*8n	T	6	T→F
+7*8n	E	6	E→T
7*8n	E+	6+	
*8n	E+7	6+7	

*8*n*	E+F	6+7	F →digit
*8n	E+T	6+7	T→ F
8n	E+T*	6+7*	
n	E+T*8	6+7*8	
n	E+T*F	6+7*8	F → digit
n	E+T	6+56	T→T*F
n	E	62	E→ E+T
	En	62	
	L	62	L→ En

S-属性定义小结

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。象一座建筑，语法分析是构架，归约处有一个“挂钩”，**语义分析和翻译的代码段（语义子程序）**就挂在这个钩子上。这样，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。

对于自顶向下语法分析，假设语义动作是在与之处于同一位置的文法非终结符被展开时执行的，而且**不考虑继承属性的处理（很简单）**。

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：

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

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

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

$$A \rightarrow X\ R$$
$$R \rightarrow Y\ R \mid \varepsilon$$

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

S-属性定义的左递归消除

- 再考虑语义动作，翻译模式变为：

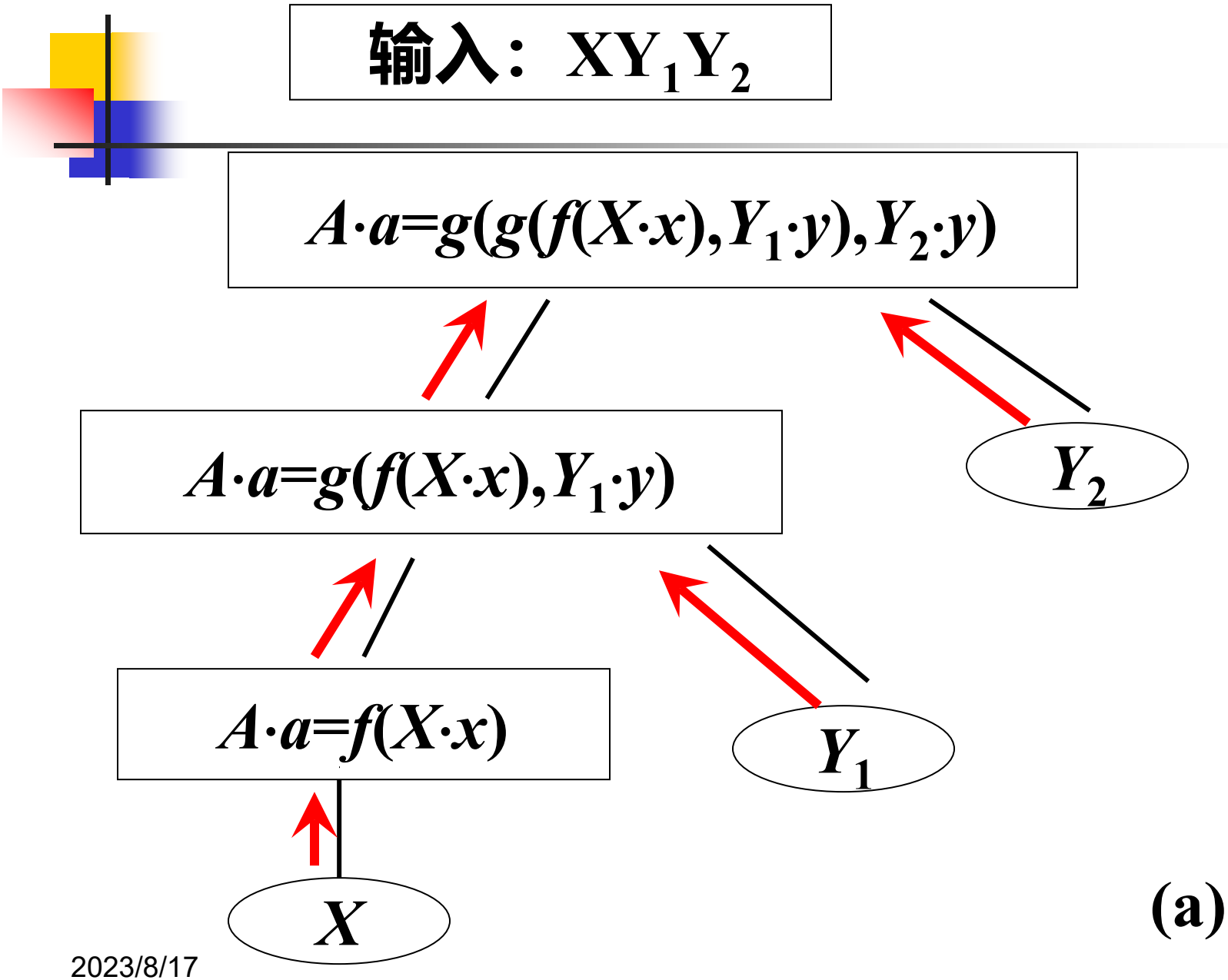
$A \rightarrow X \quad \{R \cdot i := f(X \cdot x)\}$
$R \quad \{A \cdot a := R \cdot s\}$
$R \rightarrow Y \quad \{R_1 \cdot i := g(R \cdot i, Y \cdot y)\}$
$R_1 \quad \{R \cdot s := R_1 \cdot s\}$
$R \rightarrow \varepsilon \quad \{R \cdot s := R \cdot i\}$

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$A.a := f(X.x)$

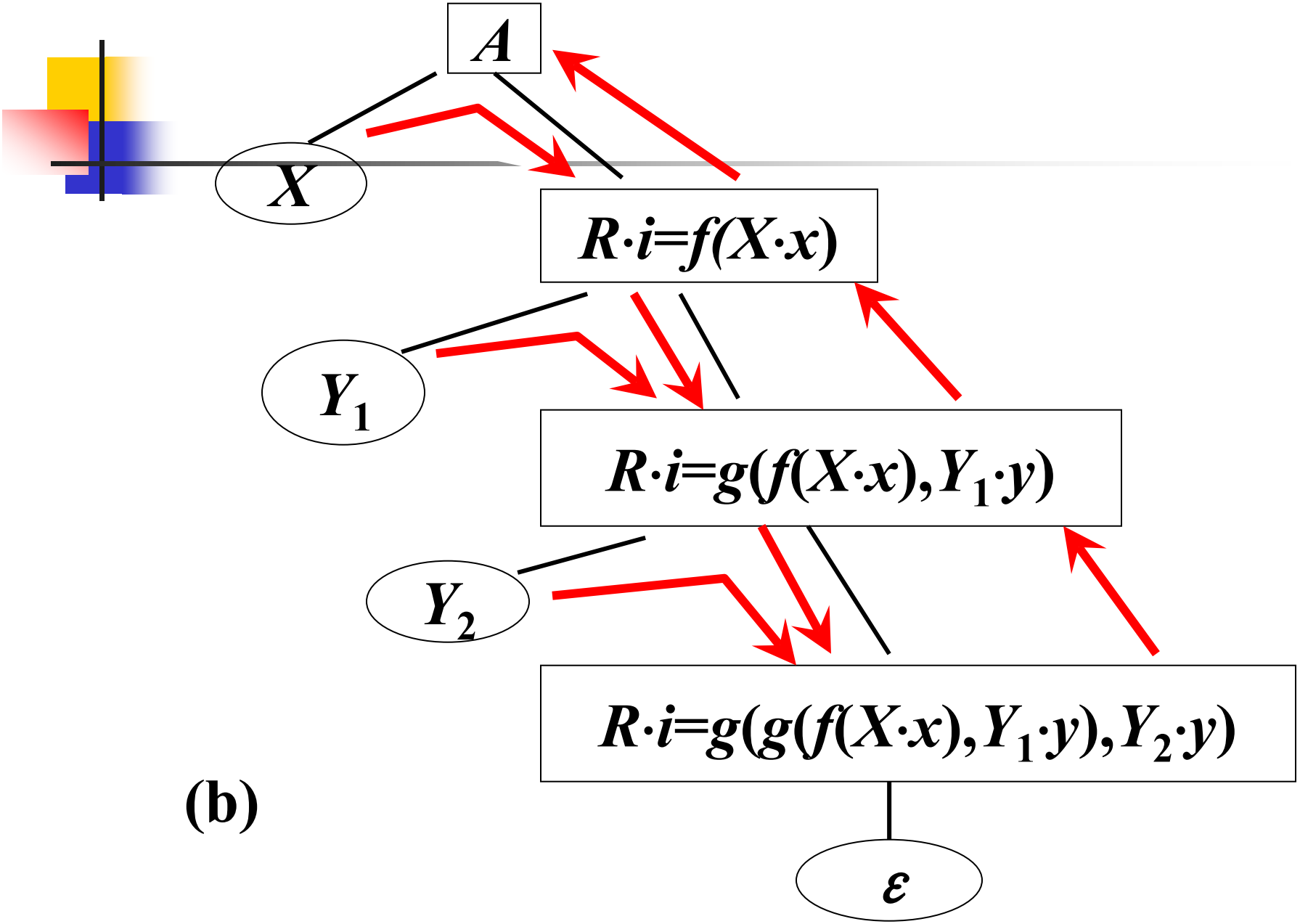
经过转换的翻译模式使用R的继承属性i和综合属性s。转换前后的结果是一样的，

为什么?

引入综合属性$R.s$在R结束生成Y时复制$R.i$的值。


输入：XY_1Y_2
$A\cdot a=g(g(f(X\cdot x),Y_1\cdot y),Y_2\cdot y)$
$A\cdot a=g(f(X\cdot x),Y_1\cdot y)$
$A\cdot a=f(X\cdot x)$
Y_2
Y_1
X

(a)



(b)

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$c:=B(b1,b2,\ldots,bk)$，其中，$b1,b2,\ldots,bk$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:↑syntax_tree_node;
function T '(inh: ↑syntax_tree_node):↑syntax_tree_node;
function F:↑syntax_tree_node;
function T:↑syntax_tree_node;
    var node, syn: ↑syntax_tree_node;
    begin
    node := F;
    syn := T '(node);
    return syn
end;
```

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

```
function F:↑syntax_tree_node;
var node: ↑syntax_tree_node;
begin    if lookahead = '(' then begin
            /* 匹配产生式F → (E) */
match('('); node := E;
match(')')
end
else if lookahead = id then begin
            /* 匹配产生式F → id */
match(id); node := mkleaf(id, id.entry)    end
else if lookahead = num then begin
            /* 匹配产生式F → 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溶入分析程序；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）。
    注：语义与语法栈不同步。

例6.17

- 对于图6.14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) $T' \rightarrow /F\{e5\}T_1'\{e4\}$

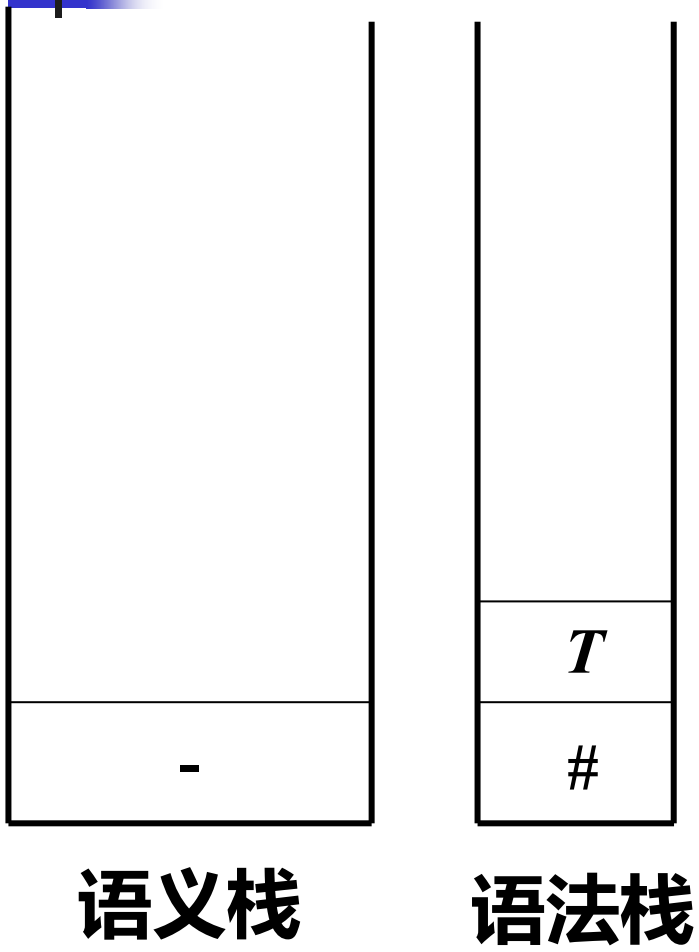
(4) $T' \rightarrow \varepsilon\{e6\}$

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

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

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

例6.17 对输入串3*x/y的翻译



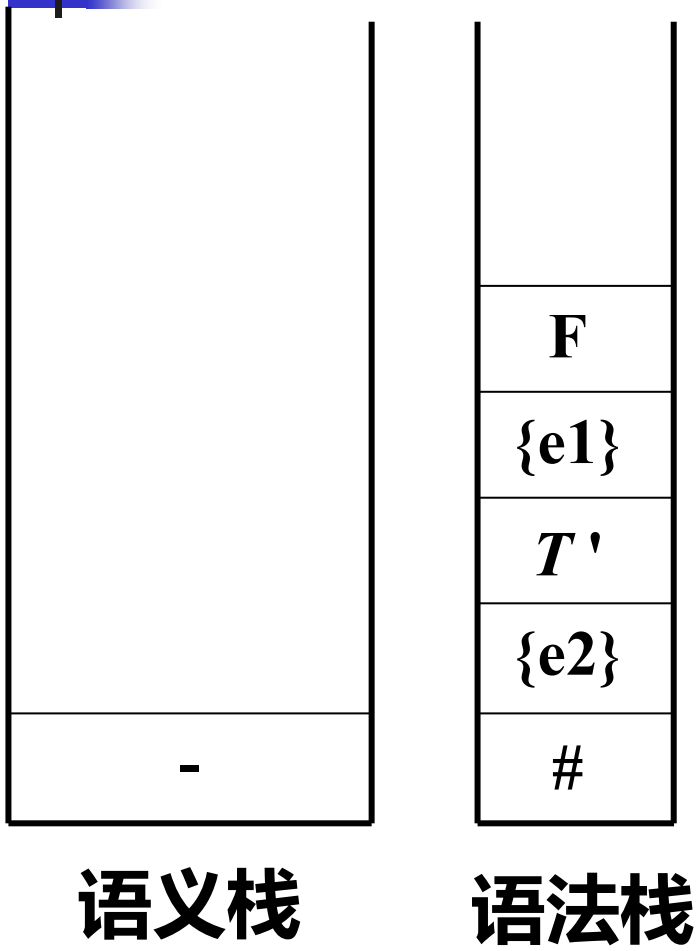
输入串　3*x/y#

↑

语法分析动作和语义操作

1. 初始化

例6.17 对输入串3*x/y的翻译

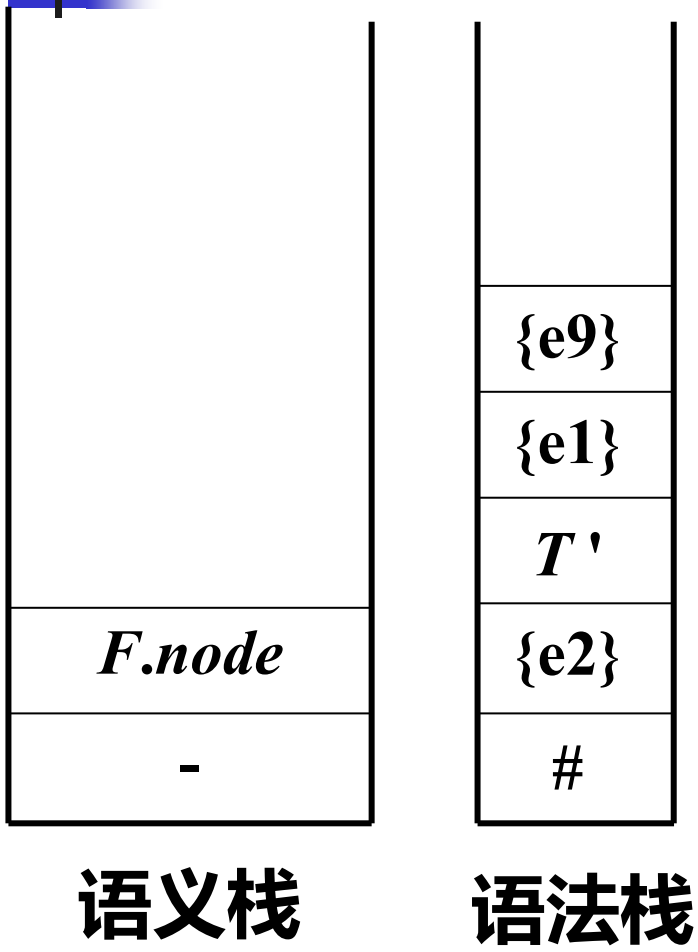


输入串　3*x/y#

语法分析动作和语义操作

2. 选产生式①的右部进栈

例6.17 对输入串3*x/y的翻译



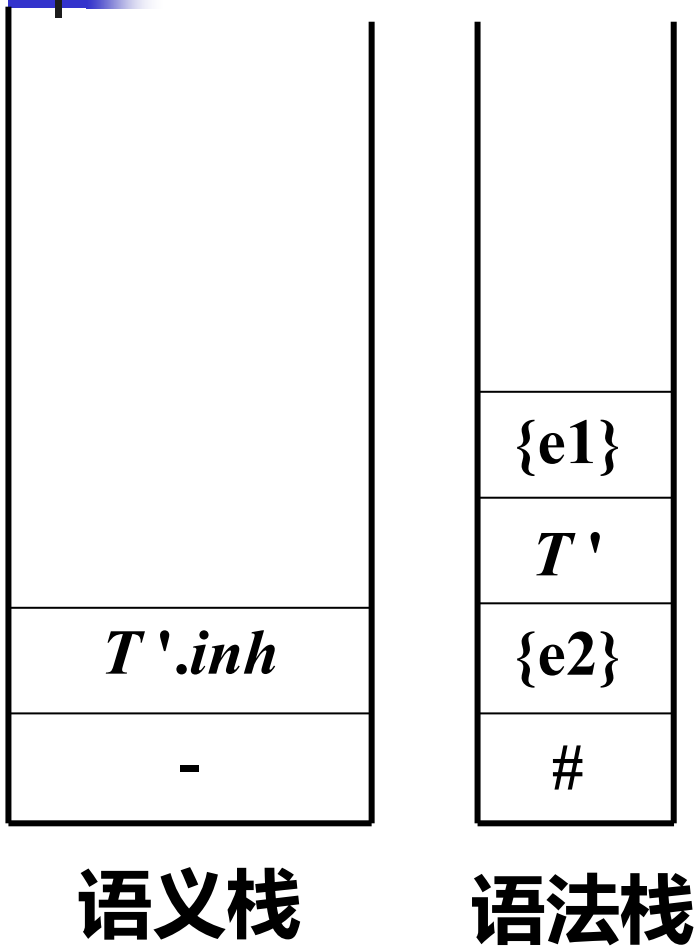
输入串　3*x/y#

↑

语法分析动作和语义操作

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

例6.17 对输入串3*x/y的翻译



输入串　3*x/y#

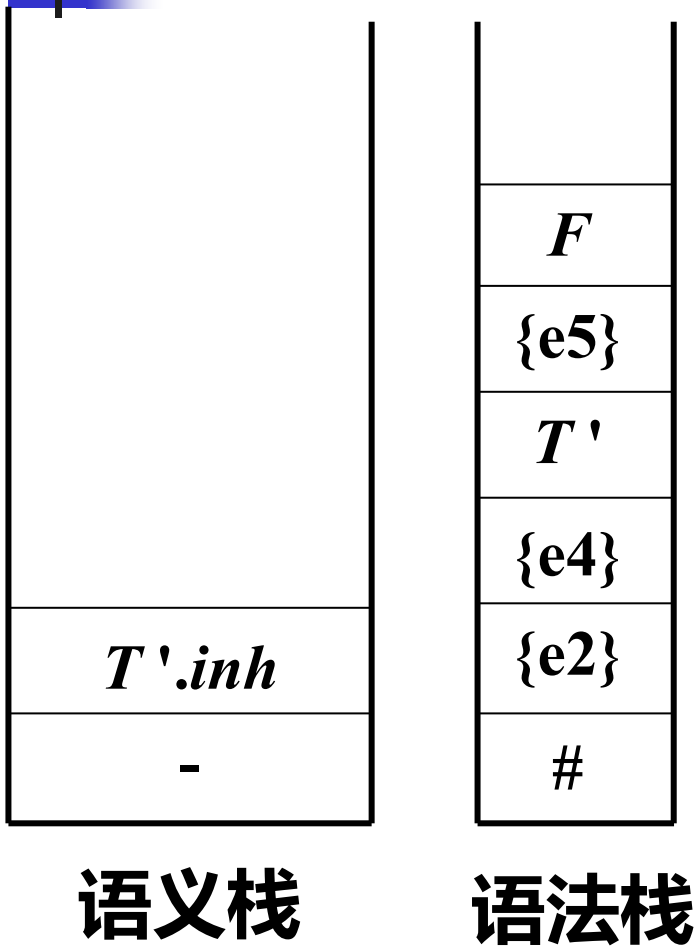
↑

语法分析动作和语义操作

4.执行动作{e1}，*F.node*出栈，

*T '.inh*被压入栈

例6.17 对输入串3*x/y的翻译

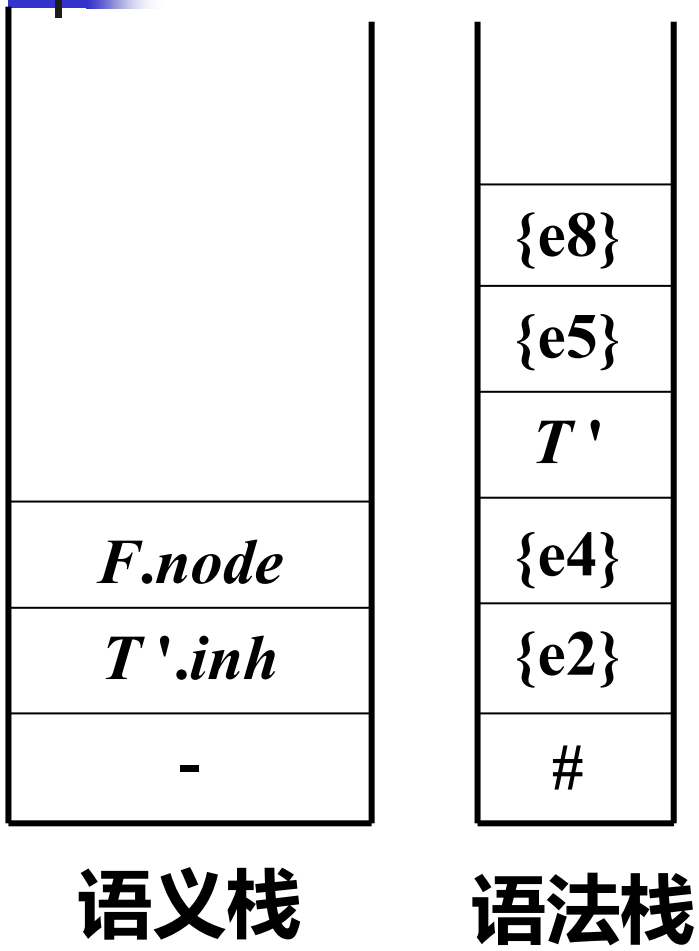


输入串　3*x/y#

语法分析动作和语义操作

5.选择产生式(3),*不进栈

例6.17 对输入串3*x/y的翻译



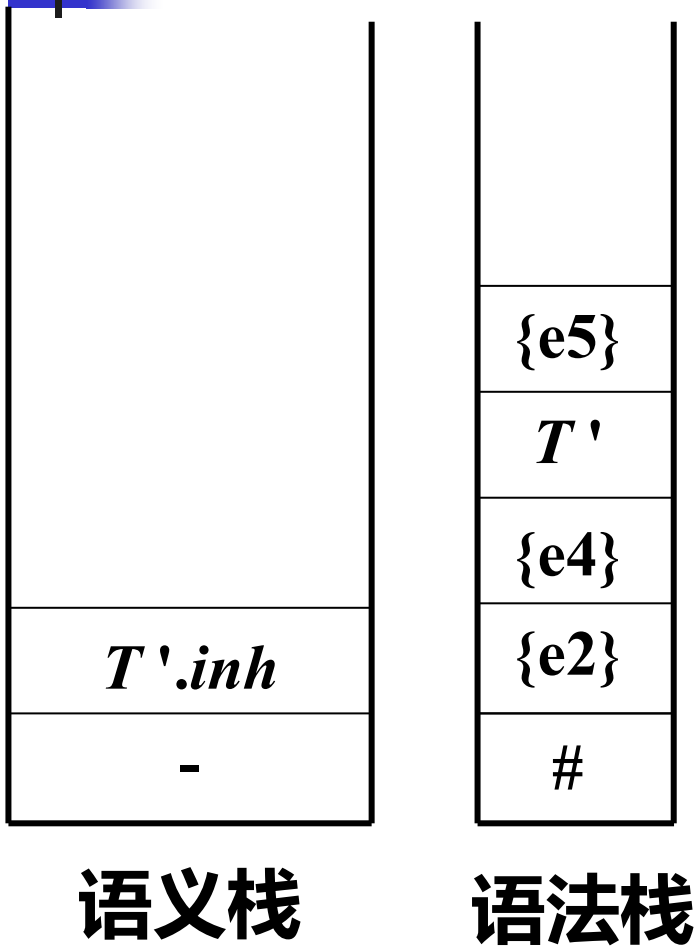
输入串　3*x/y#

↑

语法分析动作和语义操作

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

例6.17 对输入串3*x/y的翻译

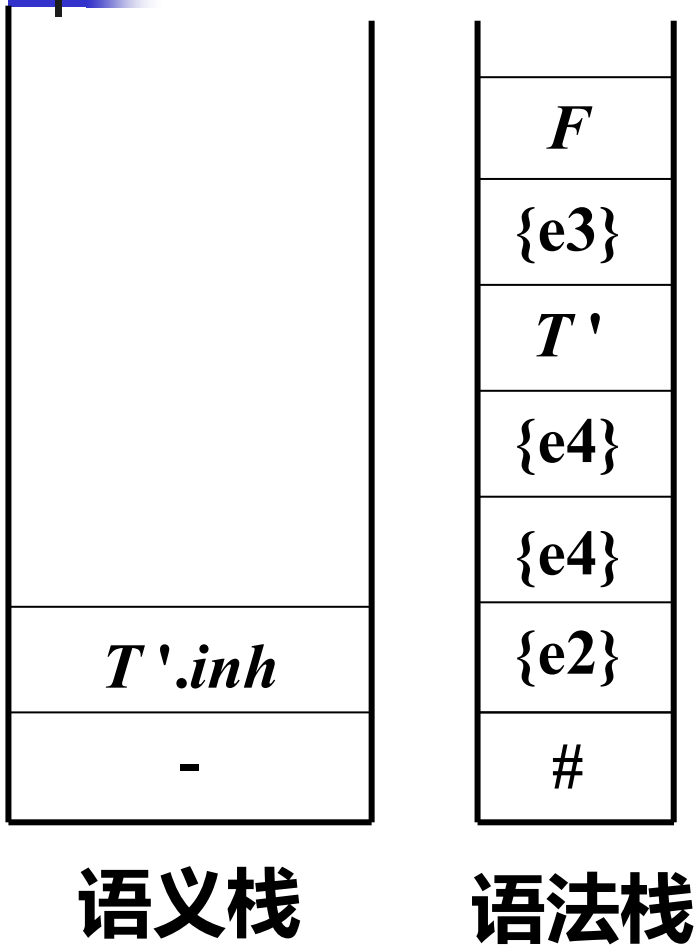


输入串　3*x/y#

语法分析动作和语义操作

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

例6.17 对输入串3*x/y的翻译

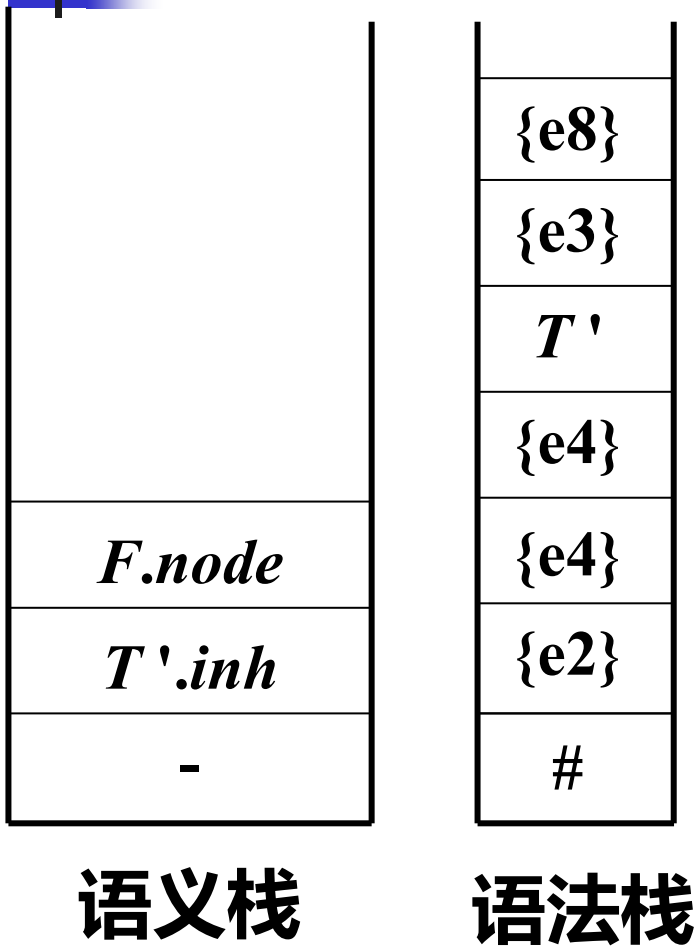


输入串　3*x/y#

语法分析动作和语义操作

8.选择产生式(2), /不进栈

例6.17 对输入串3*x/y的翻译

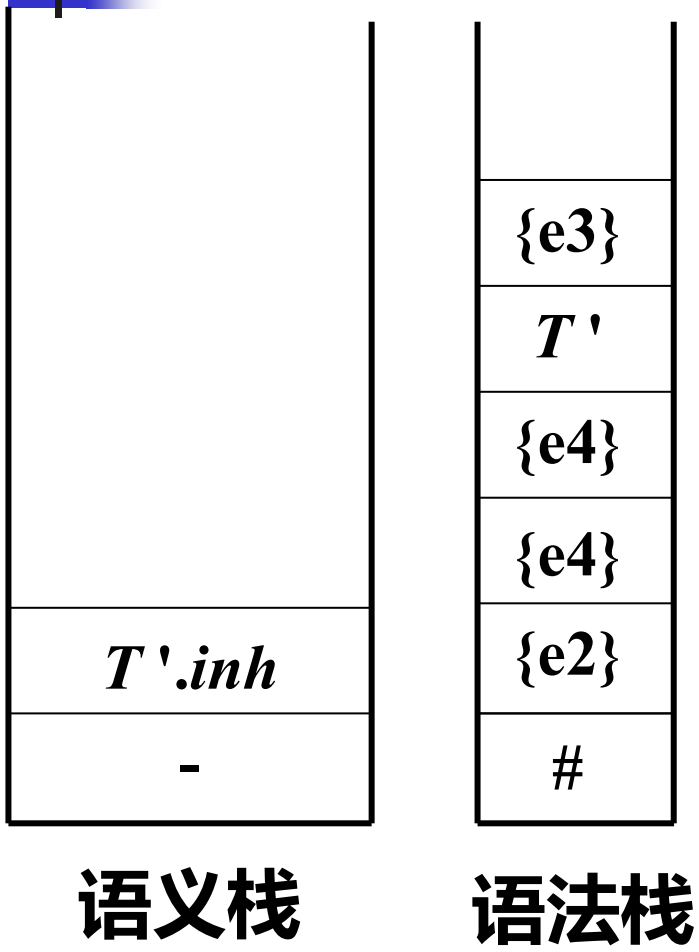


输入串　3*x/y#

语法分析动作和语义操作

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

例6.17 对输入串3*x/y的翻译

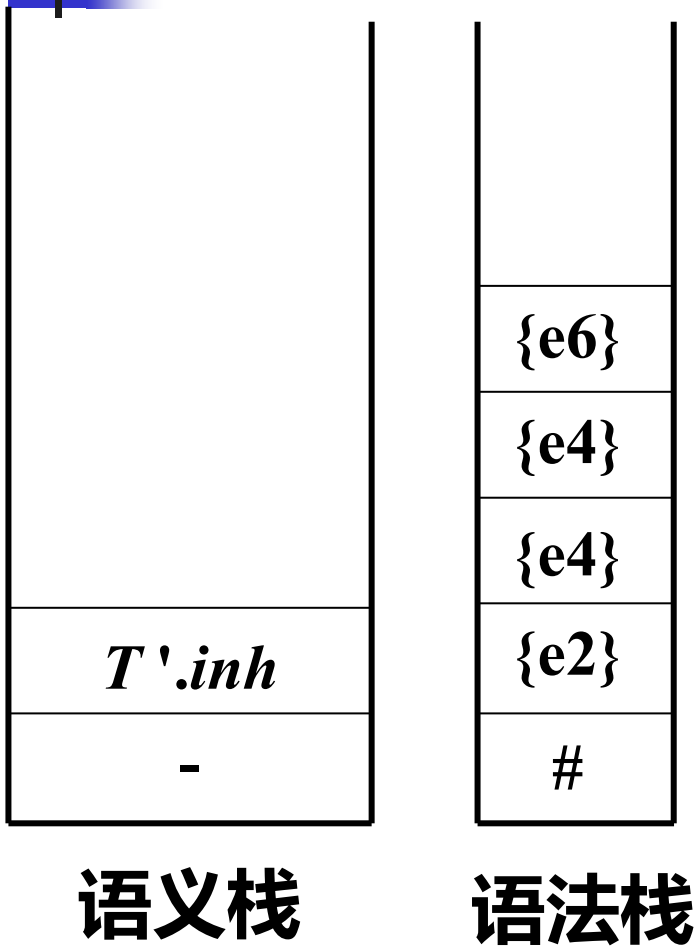


输入串　3*x/y#

语法分析动作和语义操作

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

例6.17 对输入串3*x/y的翻译

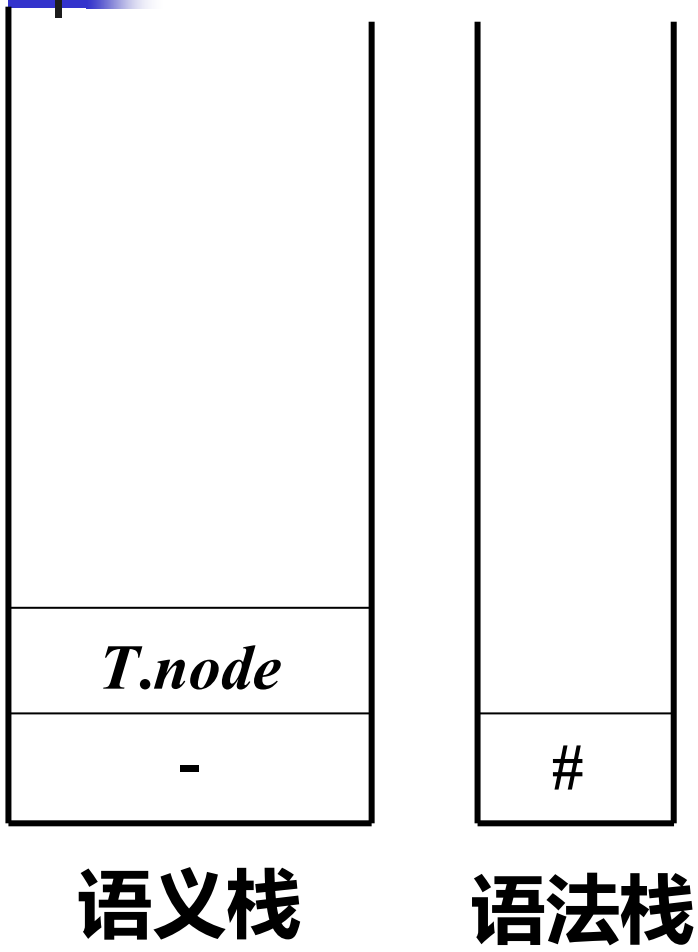


输入串　3*x/y#

语法分析动作和语义操作

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

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

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

6.4.4 L-属性定义的自底向上翻译

在自底向上分析的框架中实现L属性定义的方法

- 它能实现任何基于$LL(1)$文法的L属性定义。
- 也能实现许多（但不是所有的）基于$LR(1)$ 文法的L属性定义。

6.4.4 L-属性定义的自底向上翻译(续)

(1) 首先像6.4.1所介绍的那样构造翻译模式，它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(marker nonterminals)。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α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$M \to \varepsilon$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。不过，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。
- 例如，假设在某个$LL(1)$文法中有一个形如$A \to BC$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：

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

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$B.inh$，于是，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，但实际上，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。于是，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，需要的时候同样是可用的。

6.4.4 L-属性定义的自底向上翻译(续)

- 下面的化简可以减少标记性非终结符的个数，其中第2条还可以避免左递归文法中的分析冲突：

  (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)

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

例6.18 数学排版语言EQN

$S \rightarrow$ {$B.ps := 10$ }
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 {$B.ht := max(B_1.ht, B_2.ht)$ }

$B \rightarrow$ { $B_1.ps := B.ps$ }
B_1
sub { $B_2.ps := 0.7 \times B.ps$ }
B_2 {$B.ht := disp\ (B_1.ht, B_2.ht)$
$B.dp := \max\ (B_1.dp, B_2.dp)$}

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

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)

- 由于存在一个继承属性和两个综合属性，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	语 义 规 则
$S \rightarrow LB$	$B.ps := L.syn$; $S.ht := B.ht$; $S.dp := B.dp$
$L \rightarrow \varepsilon$	$L.syn := 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps := B.ps$; $M.inh := B.ps$; $B_2.ps := M.syn$;$B.ht := max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.syn := M.inh$
$B \rightarrow B_1$ sub NB_2	$B_1.ps := B.ps$; $N.inh := B.ps$; $B_2.ps := N.syn$; $B.ht := disp\ (B_1.ht, B_2.ht)$; $B.dp$:=max($B_1.dp$, $B_2.dp$)
$N \rightarrow \varepsilon$	$N.syn := 0.7 \times N.inh$
$B \rightarrow$ text	$B.ht$:= getheight($B.ps$,text.*lexval*); $B.dp$:= getdepth($B.ps$,text.*lexval*)

6.4.4 L-属性定义的自底向上翻译(续)

产 生 式	代 码 段
$S \rightarrow LB$	$stack[ntop].val_2 := stack[top].val_2$ $stack[ntop].val_3 := stack[top].val_3$
$L \rightarrow \varepsilon$	$stack[ntop].val_1 := 10$
$B \rightarrow B_1\ MB_2$	$stack[ntop].val_2 := max(stack[top-2].val_2, stack[top].val_2)$ $stack[ntop].val_3 := max(stack[top-2].val_3, stack[top].val_3)$
$M \rightarrow \varepsilon$	$stack[ntop].val_1 := stack[top-1].val_1$
$B \rightarrow B_1$ 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	$stack[ntop].val_2 := getheight(stack[top-1].val_1, \text{text}.lexval)$ $stack[ntop].val_3 := getdepth(stack[top-1].val_1, \text{text}.lexval)$

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。