



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS290

Compilation Principle 编译原理

第五章 语法制导翻译 (1)

郑馥丹

zhengfd5@mail.sysu.edu.cn

CONTENTS

目录

01

语法制导翻译概述
Introduction

02

SDD的求值顺序
Evaluation Order

03

S-属性翻译方案
S-attribute Translation
Schemes

04

L-属性翻译方案
L-attribute Translation
Schemes

1. 语法制导翻译[Syntax-Directed Translation]

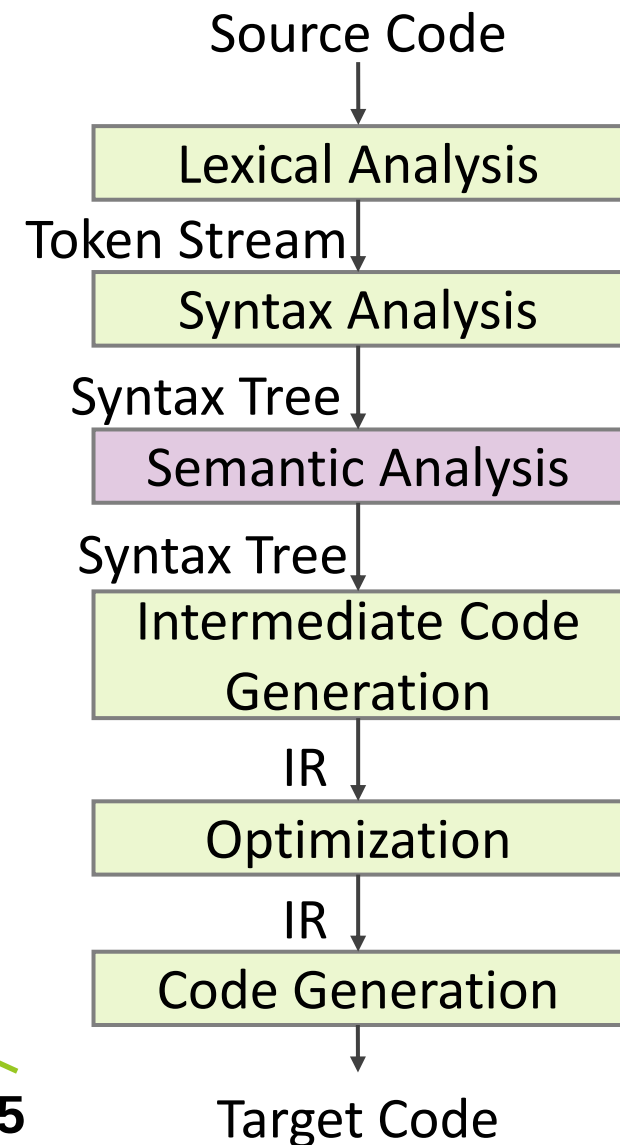
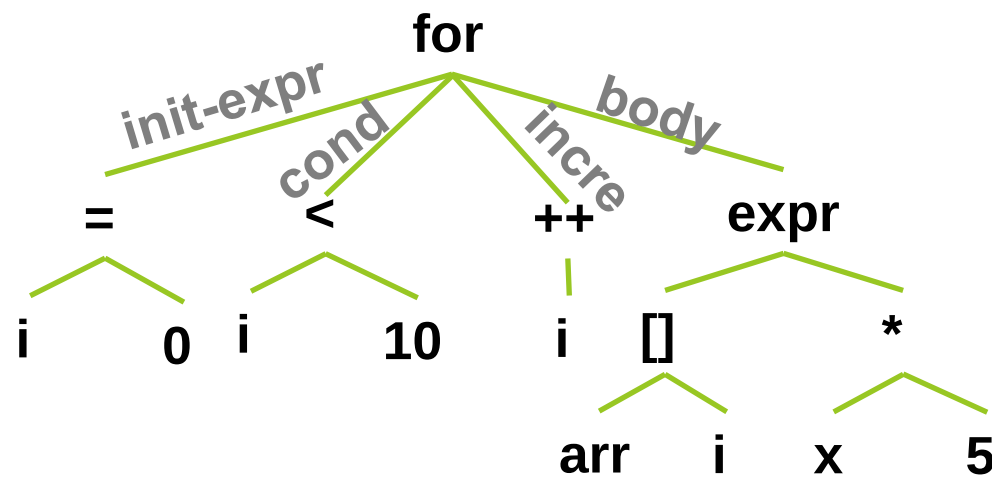
- 覆盖2个编译过程：
 - 语义分析
 - 中间代码生成
- 内容
 - 语法制导翻译的基本概念和框架
 - 这些概念和框架在语义分析和中间代码生成上的应用

1. 语法制导翻译[Syntax-Directed Translation]

Semantic Analysis[语义分析]

- 基于语法结果进一步分析语义
 - 输入：语法树，输出：语法树+符号表
 - 收集标识符的属性信息（type, scope等）
 - 输入程序是否符合**语义规则**?
 - ✓ 变量未声明即使用；重复声明
 - ✓ `int x; y = x(3);`

```
void main()  
{  
    int arr[10], i, x = 1;  
    for (i = 0; i < 10; i++)  
        arr[i] = x * 5;  
}
```



1. 语法制导翻译[Syntax-Directed Translation]

Semantic Analysis[语义分析]

- 基于语法结果进一步分析语义

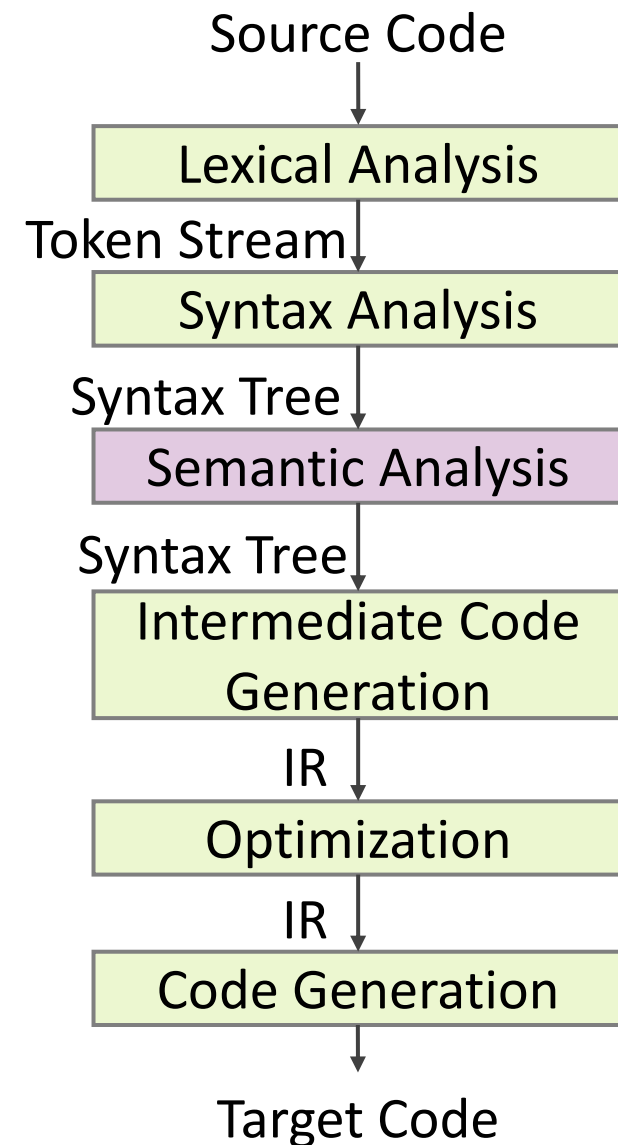
- 输入：语法树，输出：语法树+符号表
- 收集标识符的属性信息（type, scope等）
- 输入程序是否符合**语义规则**？ **数组下标越界**

声明和使用的函数没有定义
零作除数

```
#include<iostream.h>

void main() {
    cout<<"Hello world!"<<endl;
    int a=10/0;
}
```

error C2124: divide or mod by zero



1. 语法制导翻译[Syntax-Directed Translation]

Semantic Analysis[语义分析]

- 主要是类型相容检查，有以下几种：
 - 各种条件表达式的类型是不是boolean型？
 - 运算符的分量类型是否相容？
 - 赋值语句的左右部的类型是否相容？
 - 形参和实参的类型是否相容？
 - 下标表达式的类型是否为所允许的类型？
 - 函数说明中的函数类型和返回值的类型是否一致？

Semantic Analysis[语义分析]

- 其它语义检查：

- $V[E]$ 中的 V 是不是变量，而且是数组类型？
- $V.i$ 中的 V 是不是变量，而且是记录类型？ i 是不是该记录的域名？
- $x+f(\dots)$ 中的 f 是不是函数名？形参个数和实参个数是否一致？
- 每个使用性标识符是否都有声明？有无标识符的重复声明？

1. 语法制导翻译[Syntax-Directed Translation]

Semantic Analysis[语义分析]

- 在语义分析同时产生中间代码，在这种模式下，语义分析的主要功能如下：
 - 语义审查
 - 在扫描声明部分时构造标识符的符号表
 - 在扫描语句部分时产生中间代码
- 语义分析方法
 - **语法制导翻译方法**
 - 使用**属性文法**为工具来说明程序设计语言的语义

2. 语法制导定义

- 语法制导定义[Syntax-Directed Definition, **SDD**]是一个上下文无关文法和属性及语义规则的结合
- 语法制导定义也称**属性文法**[Attribute Grammar]

① 上下文无关文法

- ② **属性**：对文法的每一个符号，引进相关属性，来代表与文法符号相关的信息，如类型、值、存储位置等； **$X.a$** 表示文法符号X的属性a；
- ③ **语义规则**：为文法的每一个产生式配备的计算属性的规则，称为语义规则；它所描述的工作可以包括**属性计算、静态语义检查、符号表的操作、代码生成**等，有时写成函数或过程段。

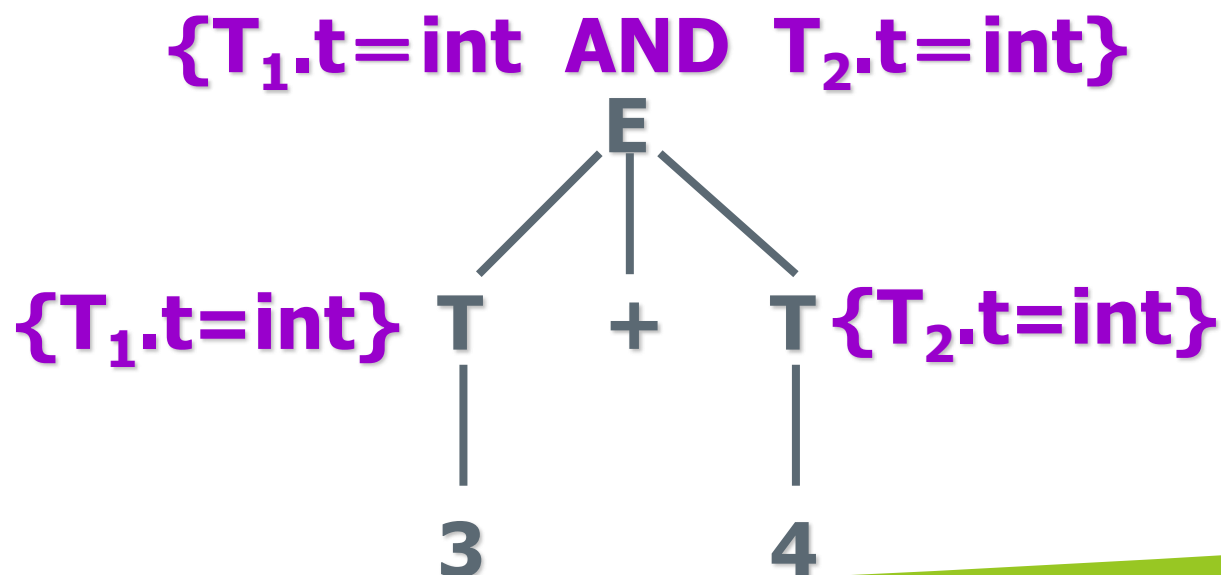
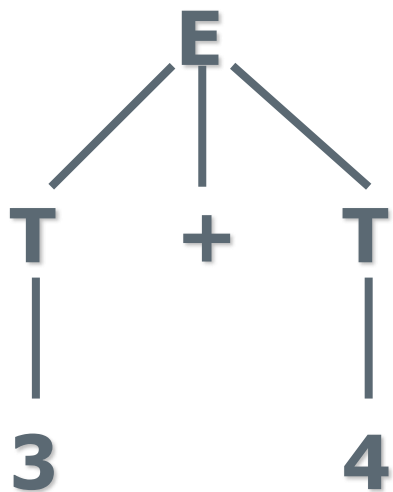
2. 语法制导定义

• 例1：类型审查的属性文法

- $E \rightarrow T_1 + T_2$ $\{T_1.t = \text{int} \text{ AND } T_2.t = \text{int}\}$
- $E \rightarrow T_1 \text{ or } T_2$ $\{T_1.t = \text{bool} \text{ AND } T_2.t = \text{bool}\}$
- $T \rightarrow \text{num}$ $\{T.t = \text{int}\}$
- $T \rightarrow \text{true}$ $\{T.t = \text{bool}\}$
- $T \rightarrow \text{false}$ $\{T.t = \text{bool}\}$

类型审查

对输入串**3+4**的语法树:



2. 语法制导定义

- 属性分类

- 综合属性[synthesized attribute]:

- ✓ 从语法树的角度来看，若一个结点的某一属性值是由该结点的**子结点**的属性值计算来的，则称该属性为综合属性
 - ✓ 用于“**自下而上**”传递信息

- 继承属性[**inherited attribute**]:

- ✓ 从语法树的角度来看，若一个结点的某一属性值是由该结点的**兄弟结点和（或）父结点**的属性值计算来的，则称该属性为继承属性
 - ✓ 用于“**自上而下**”传递信息

2. 语法制导定义

- 例2：简单算术表达式求值的属性文法

- $L \rightarrow E \ n$ $\{ L.val = E.val, \text{Print}(L.val) \}$ (n 表示表达式的结尾标记)
- $E \rightarrow E_1 + T$ $\{ E.val = E_1.val + T.val \}$
- $E \rightarrow T$ $\{ E.val = T.val \}$
- $T \rightarrow T_1 * F$ $\{ T.val = T_1.val * F.val \}$
- $T \rightarrow F$ $\{ T.val = F.val \}$
- $F \rightarrow (E)$ $\{ F.val = E.val \}$
- $F \rightarrow \text{digit}$ $\{ F.val = \text{digit.lexval} \}$

$L.val$ 、 $E.val$ 、 $T.val$ 、 $F.val$ 的计算都来自它右部的非终结符(子节点)——综合属性

一个只包含综合属性的SDD称为S-属性[S-attribute]的SDD，
或称S-属性文法，该SDD与LR分析过程对应

2. 语法制导定义

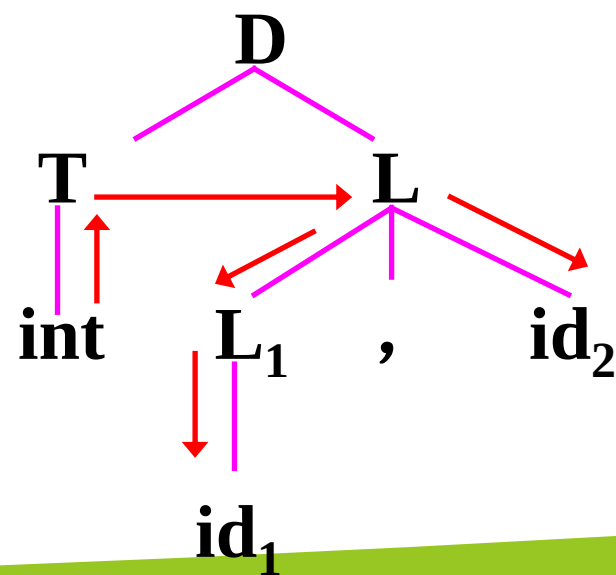
• 例3：描述变量类型说明的属性文法

- $D \rightarrow TL$ $\{ L.in = T.type \}$
- $T \rightarrow int$ $\{ T.type = int \}$
- $T \rightarrow real$ $\{ T.type = real \}$
- $L \rightarrow L_1, id$ $\{ L_1.in = L.in; addtype(id.entry, L.in) \}$
- $L \rightarrow id$ $\{ addtype(id.entry, L.in) \}$

T.type是**综合属性**（依赖于子节点）

L.in是**继承属性**（其属性值的计算依赖于其兄弟或父节点）

int id_1, id_2 的语法树：
用→表示属性的传递情况



2. 语法制导定义

- 属性分类

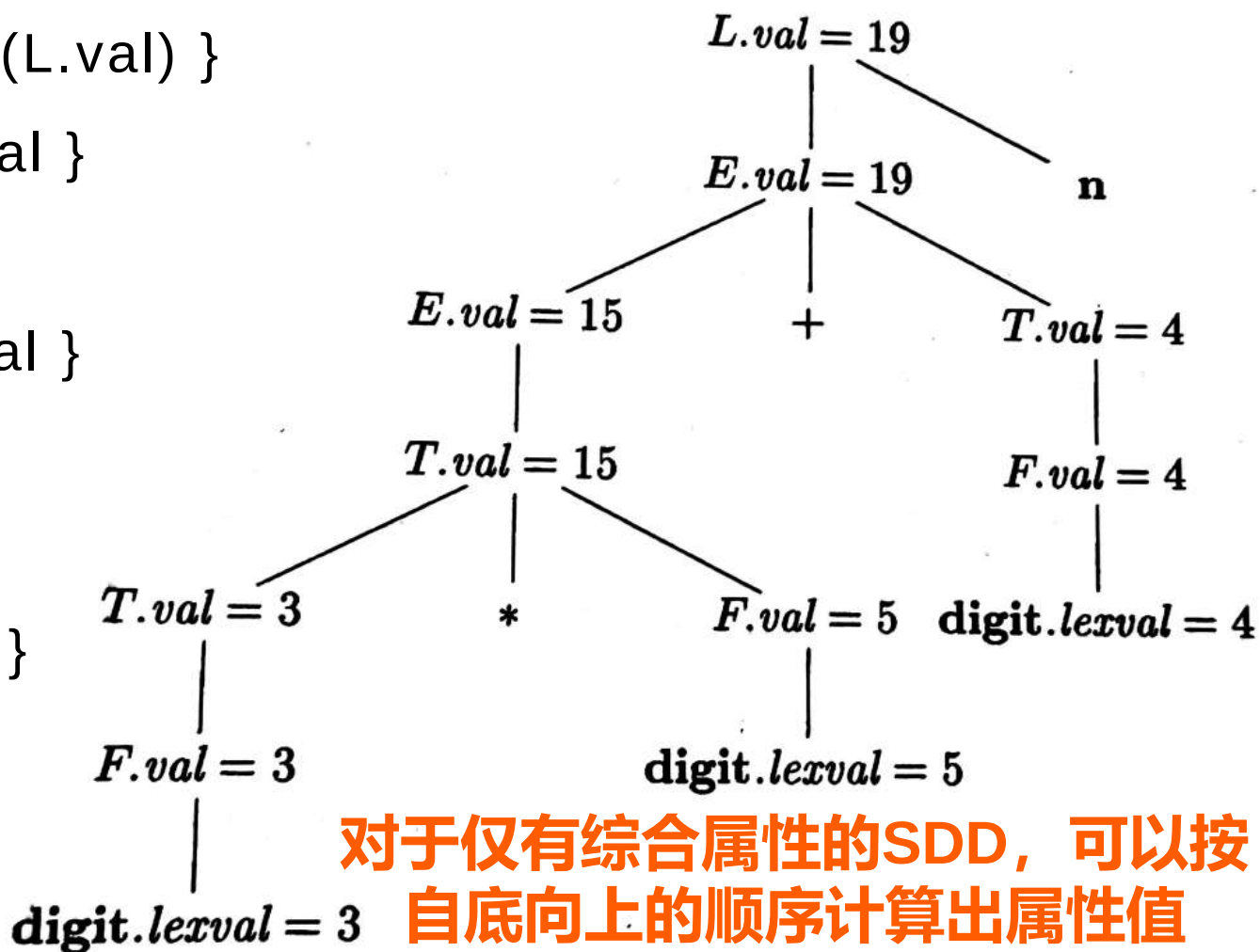
- **终结符只有综合属性**，它们由词法分析器提供
- 非终结符既有综合属性也有继承属性，但**文法开始符没有继承属性**

3. 注释语法分析树[Annotated Parse Tree]

- 注释语法分析树：显示各个属性的值的语法分析树
- 例2：简单算术表达式求值的属性文法

- $L \rightarrow E \ n$ $\{ L.val = E.val, \text{Print}(L.val) \}$
- $E \rightarrow E_1 + T$ $\{ E.val = E_1.val + T.val \}$
- $E \rightarrow T$ $\{ E.val = T.val \}$
- $T \rightarrow T_1 * F$ $\{ T.val = T_1.val * F.val \}$
- $T \rightarrow F$ $\{ T.val = F.val \}$
- $F \rightarrow (E)$ $\{ F.val = E.val \}$
- $F \rightarrow \text{digit}$ $\{ F.val = \text{digit.lexval} \}$

对应的 $3*5+4n$ 的注释语法分析树：



- $L \rightarrow E \ n$ $\{ L.val = E.val, \text{Print}(L.val) \}$ (n 表示表达式的结尾标记)
 - $E \rightarrow E_1 + T$ $\{ E.val = E_1.val + T.val \}$
 - $E \rightarrow T$ $\{ E.val = T.val \}$
 - $T \rightarrow T_1 * F$ $\{ T.val = T_1.val * F.val \}$
 - $T \rightarrow F$ $\{ T.val = F.val \}$
 - $F \rightarrow (E)$ $\{ F.val = E.val \}$
 - $F \rightarrow \text{digit}$ $\{ F.val = \text{digit.lexval} \}$

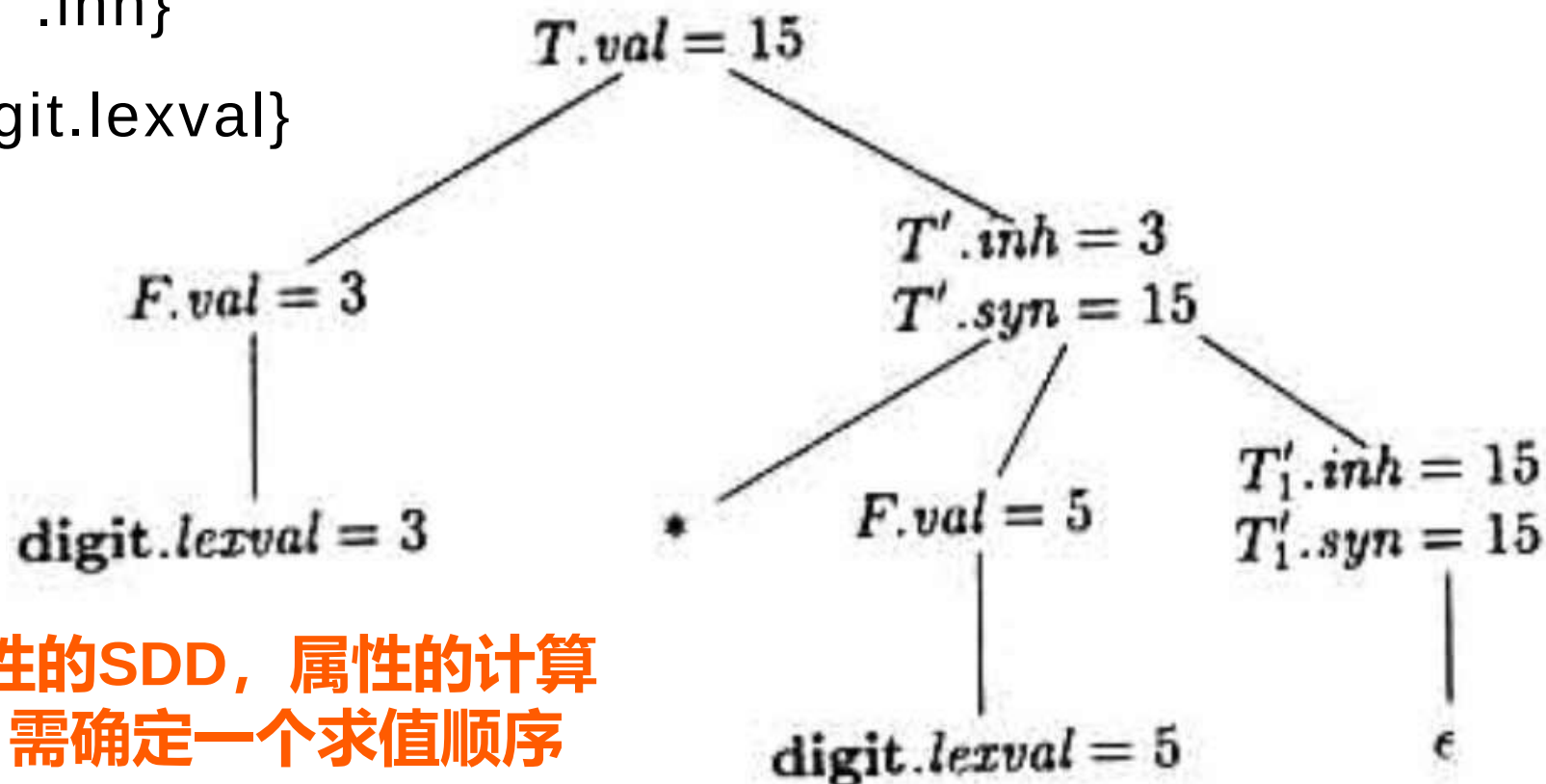


3. 注释语法分析树[Annotated Parse Tree]

• 例：属性文法

- $T \rightarrow FT'$ $\{T'.inh = F.val, T.val = T'.syn\}$
- $T' \rightarrow *FT_1'$ $\{T_1'.inh = T'.inh \times F.val, T'.syn = T_1'.syn\}$
- $T' \rightarrow \epsilon$ $\{T'.syn = T'.inh\}$
- $F \rightarrow \text{digit}$ $\{F.val = \text{digit.lexval}\}$

3*5的注释语法分析树：



对于有继承属性和综合属性的SDD，属性的计算
不按照自底向上的顺序，需确定一个求值顺序

3. 注释语法分析树[Annotated Parse Tree]

• 例：属性文法

– $T \rightarrow BC$

{ $T.type = C.type$,
 $C.base = B.type$ }

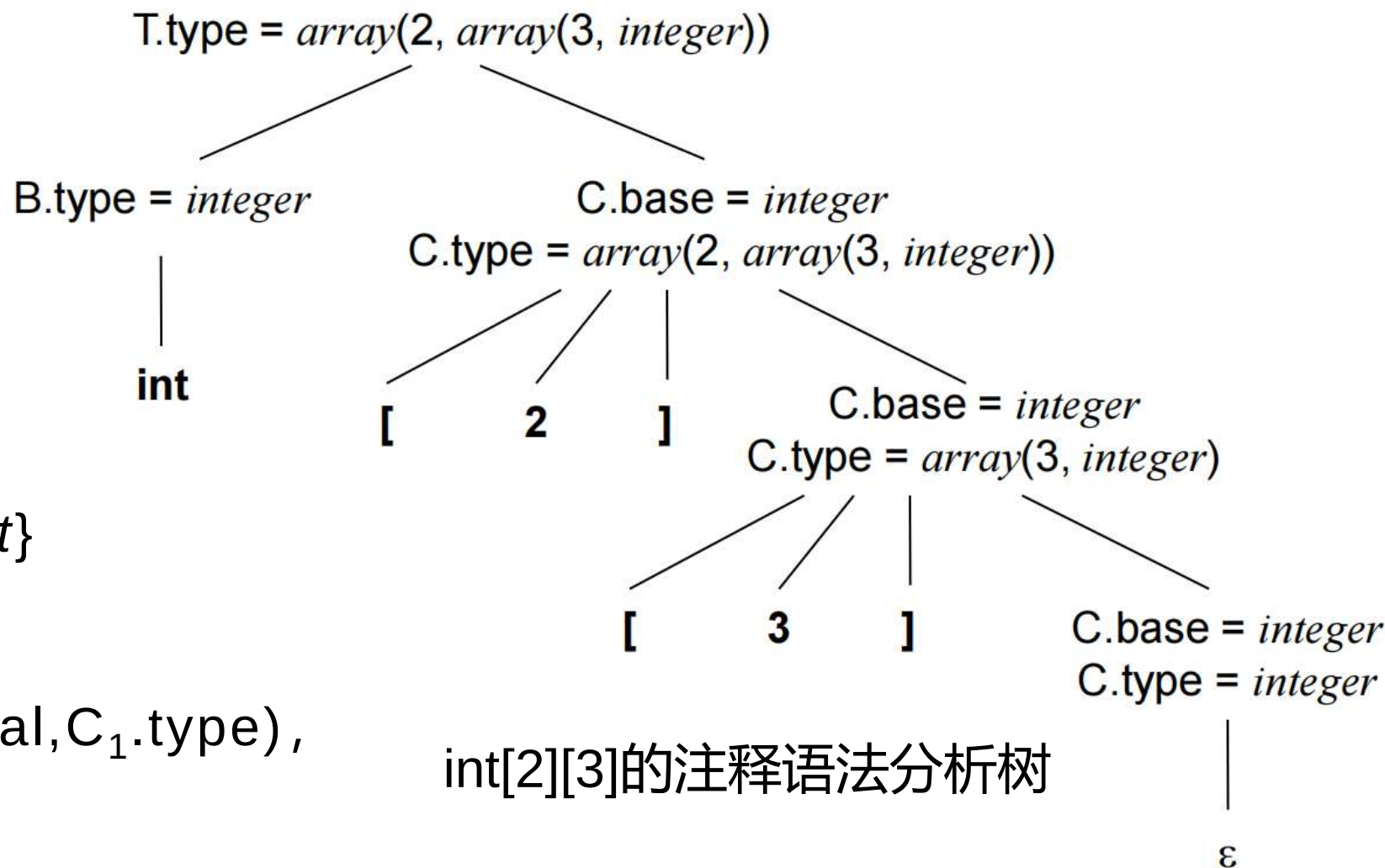
– $B \rightarrow int$ { $B.type = int$ }

– $B \rightarrow float$ { $B.type = float$ }

– $C \rightarrow [num]C_1$

{ $C.type = array(num.val, C_1.type)$,
 $C_1.base = C.base$ }

– $C \rightarrow \varepsilon$ { $C.type = C.base$ }



对于有继承属性和综合属性的SDD，属性的计算
 不按照自底向上的顺序，需确定一个求值顺序

3. 注释语法分析树[Annotated Parse Tree]

- 在对一棵语法分析树的某个节点的一个属性进行求值之前，须首先求出这个属性值所依赖的所有属性值
- 对于仅有综合属性的SDD，可以按自底向上的顺序计算出属性值
- 对于有继承属性和综合属性的SDD，属性的计算不按照自底向上的顺序，需确定一个**求值顺序——依赖图**

CONTENTS

目 录

01

语法制导翻译概述

Introduction

02

SDD的求值顺序

Evaluation Order

03

S-属性翻译方案

S-attribute Translation
Schemes

04

L-属性翻译方案

L-attribute Translation
Schemes

1. 依赖图

• 依赖图[dependency graph]:

- 确定一棵语法分析树中各个属性的求值顺序
- 描述了某个语法分析树中的属性之间的信息流
- 从一个属性到另一个的边表示计算第二个属性时需要第一个属性的值

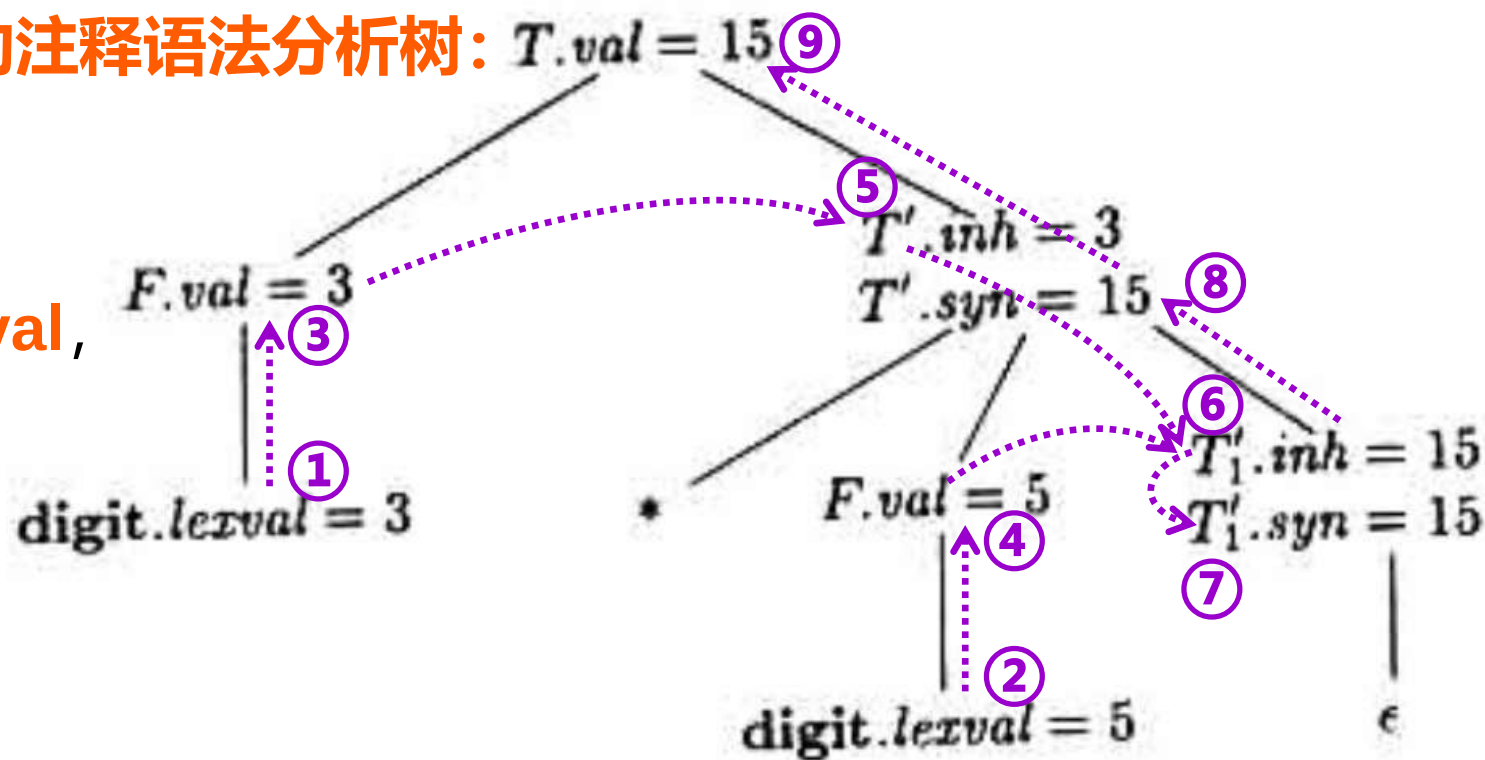
3*5的注释语法分析树: $T.val = 15$

$T \rightarrow FT'$ ⑤ { $T'.inh = F.val$,
⑨ $T.val = T'.syn$ }

$T' \rightarrow *FT_1'$ ⑥ { $T_1'.inh = T'.inh \times F.val$,
⑧ $T'.syn = T_1'.syn$ }

$T' \rightarrow \epsilon$ ⑦ { $T'.syn = T'.inh$ }

$F \rightarrow \text{digit}$ ③④ { $F.val = \text{digit.lexval}$ }



2. 属性求值的顺序

- 依赖图刻画了对一棵语法分析树中不同结点上的属性求值时可能采取的顺序
- 如果依赖图中有一条从结点M到结点N的边，那么要先对M对应的属性求值，再对N对应的属性求值
- 可行的求值顺序就是满足下列条件的结点顺序 $N_1, N_2, \dots, N_k$ ，如果有一条从结点 N_i 到 N_j 的依赖图的边，则 $i < j$ ，这个排序称为这个图的**拓扑排序 (topological sort)**
- **若图中有环，则不存在拓扑排序**
- **有向无环图(Directed Acyclic Graph, DAG)则至少存在一个拓扑排序**

2. 属性求值的顺序

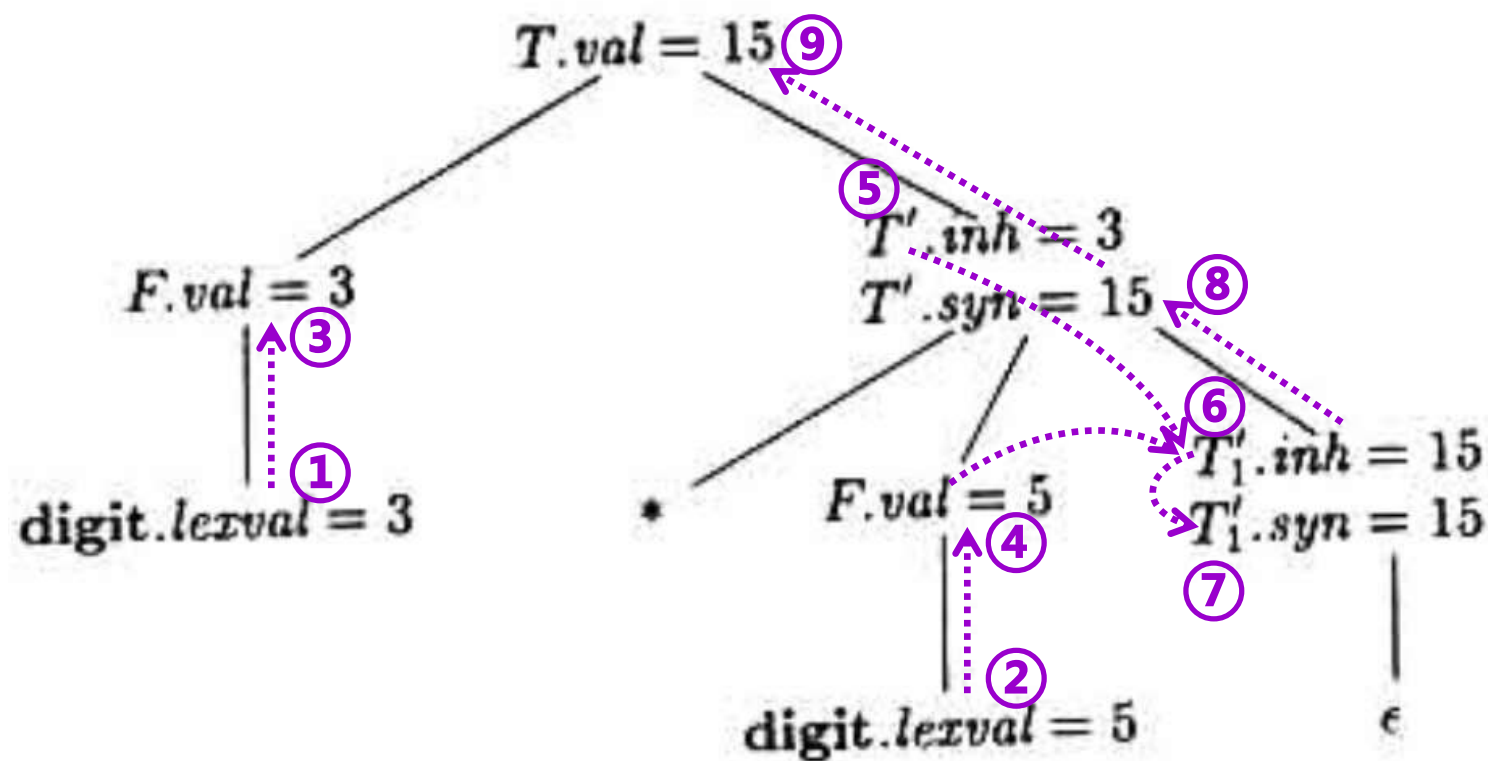
- 同一个依赖图可能存在多个拓扑排序

- 例：右图的拓扑排序：

– ①②③④⑤⑥⑦⑧⑨

– ①③⑤②④⑥⑦⑧⑨

–



3. 抽象语法树[Abstract Syntax Tree, AST]

- 抽象语法树比语法分析树 (Parse Tree) 更简洁, 直接反映了源代码的语法结构, 同时剔除了无关的语法细节 (如分号、括号等)
- 例: 对于代码 $a=3+5*2$;
 - 语法分析树: 会包含全部结点 ($=$, $+$, $*$, $;$)
 - 抽象语法树: 仅保留关键结构:



3. 抽象语法树[Abstract Syntax Tree, AST]

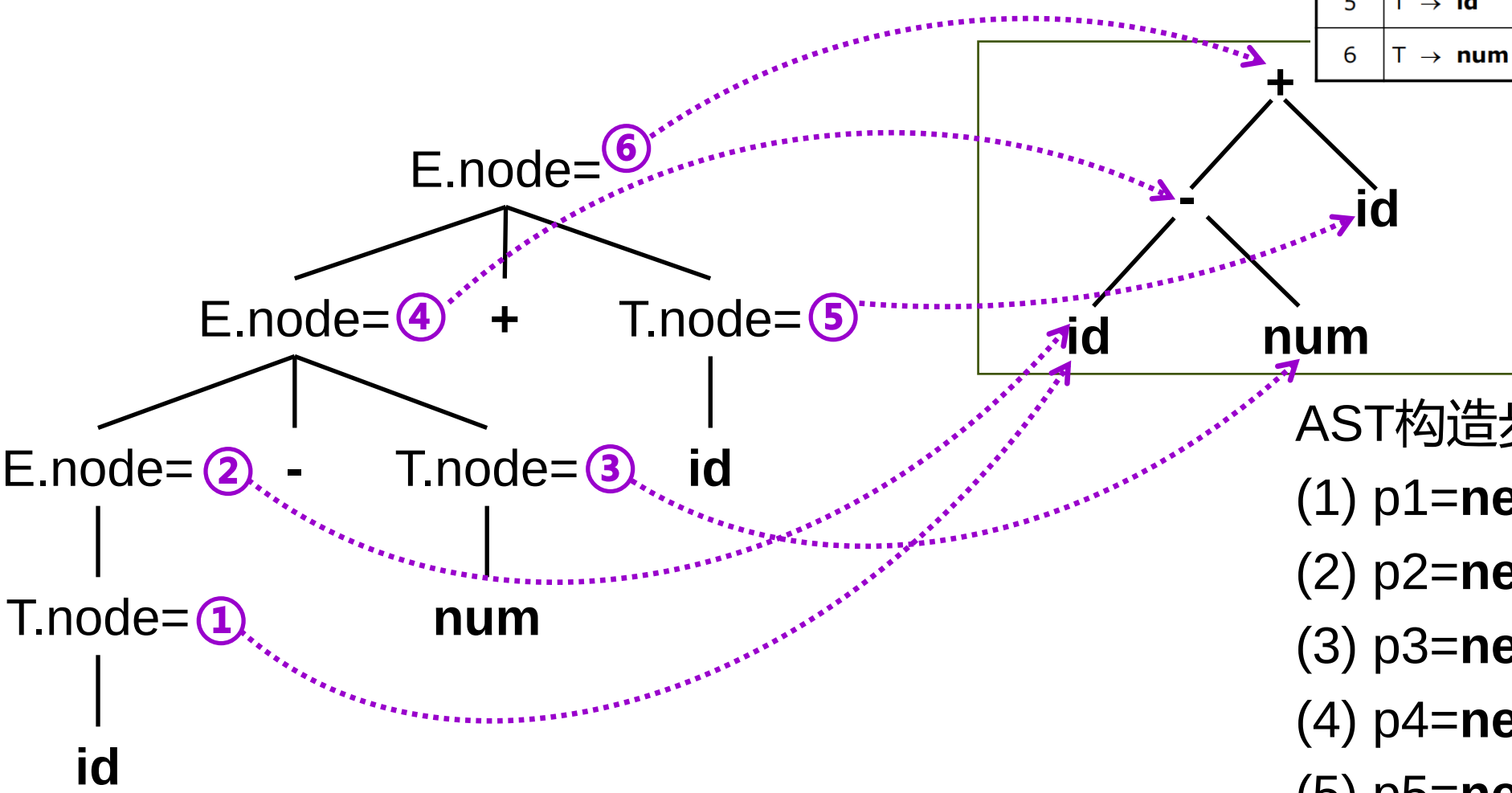
- 例：文法

No.	Productions	Semantic Rules
1	$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+', E_1.\text{node}, T.\text{node})$
2	$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-', E_1.\text{node}, T.\text{node})$
3	$E \rightarrow T$	$E.\text{node} = T.\text{node}$
4	$T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
5	$T \rightarrow \text{id}$	$T.\text{node} = \text{new Leaf}(\text{id}, \text{id}.\text{entry})$
6	$T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.\text{val})$

3. 抽象语法树[Abstract Syntax Tree, AST]

• **a-4+c**的抽象语法树

No.	Productions	Semantic Rules
1	$E \rightarrow E_1 + T$	$E.node = \mathbf{new} \text{ Node}('+', E_1.node, T.node)$
2	$E \rightarrow E_1 - T$	$E.node = \mathbf{new} \text{ Node}('-', E_1.node, T.node)$
3	$E \rightarrow T$	$E.node = T.node$
4	$T \rightarrow (E)$	$T.node = E.node$
5	$T \rightarrow \mathbf{id}$	$T.node = \mathbf{new} \text{ Leaf}(\mathbf{id}, \mathbf{id.entry})$
6	$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new} \text{ Leaf}(\mathbf{num}, \mathbf{num.val})$



- AST构造步骤:
- (1) $p1 = \mathbf{new} \text{ Leaf}(\mathbf{id}, \text{entry-a})$
 - (2) $p2 = \mathbf{new} \text{ Leaf}(\mathbf{num}, 4)$
 - (3) $p3 = \mathbf{new} \text{ Node}('-', p1, p2)$
 - (4) $p4 = \mathbf{new} \text{ Leaf}(\mathbf{id}, \text{entry-c})$
 - (5) $p5 = \mathbf{new} \text{ Node}('+', p3, p4)$

4. S-属性的定义

- 一个仅包含综合属性 [**S**ynthesized attribute] 的 SDD 称为 **S-属性** [**S-attribute**] 的 SDD，或称 S-属性文法
 - 每个属性都必须是综合属性
 - 每个节点的属性值 **仅由其子节点的属性值计算** 而来（自底向上传递信息）
 - 可以保证 **求值顺序与 LR 分析的输出顺序相同**
 - 可按照语法分析树结点的任何自底向上顺序来计算属性值：
 - ✓ **后序遍历**

```
postorder(N){  
    for(从左边开始, 对N的每个子结点C) postorder(C);  
    对N关联的各个属性求值;  
}
```
 - ✓ 当遍历最后一次离开某个结点 N 时计算出 N 的各个属性值

4. S-属性的定义

• 例：S-属性SDD

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

• 例：非S-属性SDD

- 若SDD中包含继承属性，如： $A \rightarrow BC \{ B.in = A.in, C.in = B.s, A.s = C.s \}$
- 则B.in依赖A.in（父节点），C.in依赖B.s（左边的兄弟节点）
- 这种计算顺序不能直接用LR分析，因为LR分析是严格自底向上的，无法在归约时访问父节点或右边兄弟节点的属性

算术表达式3+5:

LR分析过程	S属性求值顺序
扫描3, 归约 $T \rightarrow 3$	计算 $T.val = 3$
归约 $E \rightarrow T$	计算 $E.val = T.val = 3$
扫描+, 移进	
扫描5, 归约 $T \rightarrow 5$	计算 $T.val = 5$
归约 $E \rightarrow E_1 + T$	计算 $E.val = 3 + 5 = 8$

5. L-属性的定义

- 一个SDD的产生式右部所关联的各个属性之间，依赖图的边总是从左到右 [Left-to-right]，则称该SDD为**L-属性[L-attribute]**的SDD
- L-属性[L-attribute]的SDD中，每个属性：
 - 要么是一个综合属性
 - 要么是一个继承属性，但有以下约束：假设存在产生式 $A \rightarrow X_1X_2 \dots X_n$ ，其 X_i 的继承属性 $X_i.a$ 的计算规则只能依赖：
 - ✓ 产生式左部A的继承属性
 - ✓ X_i 左边的文法符号 X_1 、 X_2 、...、 X_{i-1} 的继承属性或综合属性（即已经计算过的兄弟节点）
 - ✓ X_i 本身的属性，且由 X_i 的属性组成的依赖图中不存在环

5. L-属性的定义

- 即：L-属性定义的目的是**确保属性计算可以按从左到右的顺序进行**：
 - **继承属性不能依赖右边的符号**，否则计算顺序会混乱（因为右边的符号可能还未被处理）
 - 不能有循环依赖，否则无法确定计算顺序
- 适用于：
 - **递归下降、LL(1)等自顶向下分析方法**
 - **LR(1)等自底向上的分析方法**

5. L-属性的定义

- 例：L-属性SDD

- $D \rightarrow Tid\{id.in_type = T.type, D.type = T.type\}$
- id 的属性 in_type 是继承属性，只依赖左边的 T 的属性
- D 的属性 $type$ 是综合属性
- 没有循环依赖

- 例：非L-属性SDD

- $A \rightarrow BC\{B.in = C.s\}$
- B 的属性 in 是继承属性，且依赖右边的 C

随堂练习 (2)

- 假设我们有一个产生式 $A \rightarrow BCD$ ，其中， A 、 B 、 C 、 D 这四个非终结符都有两个属性， s 是一个综合属性， i 是一个继承属性，对于下面每组规则，指出这些规则是否满足S-属性定义的要求，是否满足L-属性定义的要求？

(1) $A.s = B.i + C.s$

(2) $A.s = B.i + C.s$, $D.i = A.i + B.s$

(3) $A.s = B.s + D.s$

规则	S-属性定义	L-属性定义	原因
(1) $A.s = B.i + C.s$	✗ 否	☑ 是	依赖继承属性 $B.i$ ，但L-属性允许，因 $B.i$ 来源合法（非由右边 C 而来）
(2) $A.s = B.i + C.s$, $D.i = A.i + B.s$	✗ 否	☑ 是	继承属性 $D.i$ 依赖父节点和左边兄弟节点，符合L-属性
(3) $A.s = B.s + D.s$	☑ 是	☑ 是	仅依赖综合属性，符合S-属性和L-属性

CONTENTS

目 录

01

语法制导翻译概述

Introduction

02

SDD的求值顺序

Evaluation Order

03

S-属性翻译方案

S-attribute Translation
Schemes

04

L-属性翻译方案

L-attribute Translation
Schemes

1. 语法制导的翻译方案

- 语法制导翻译的工作步骤

- ① 向语法符号引入属性
- ② 为每个产生式定义**语义规则**
- ③ 根据注释语法分析树绘制依赖图
- ④ 依赖图的拓扑排序确定评估顺序
- ⑤ 按照求值顺序执行**语义规则**

1. 语法制导的翻译方案

- 语法制导的翻译方案[Syntax-Directed Translation Schemes, SDT]
 - 在产生式右部嵌入了程序片段的上下文无关文法
 - 这些程序片段称为**语义规则**（语义动作）
- **自底向上分析**对应**S-属性翻译方案**

2. S-属性翻译方案

- 以LR分析为例：

- 将LR分析器能力扩大，增加在归约后**调用语义规则**的功能
- **语义动作是在每一步归约之后执行的**
- **增加语义栈**，语义值放到与符号栈同步操作的语义栈中，多项语义值可设多个语义栈，栈结构为：

S_m	X_m	$X_m.val$
.	.	.
.	.	.
.	.	.
S_1	X_1	$X_1.val$
S_0	#	---

状态栈 符号栈 语义栈

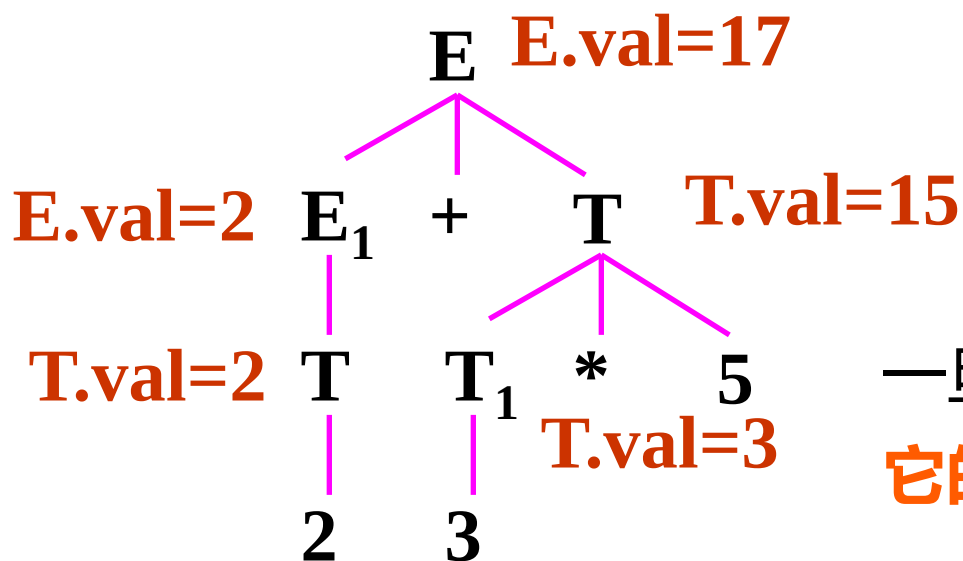
2. S-属性翻译方案

• 例：简单算术表达式求值的属性文法

- 1) $E \rightarrow E_1 + T \quad \{ E.val = E_1.val + T.val \}$
- 2) $E \rightarrow T \quad \{ E.val = T.val \}$
- 3) $T \rightarrow T_1 * digit \quad \{ T.val = T_1.val * digit.lexval \}$
- 4) $T \rightarrow digit \quad \{ T.val = digit.lexval \}$

2+3*5的语法树：

自下而上语法制导翻译过程：



一旦语法分析确认输入符号串是一个句子，
它的值也同时由语义规则计算出来

2. S-属性翻译方案

• 例：简单算术表达式求值的属性文法

- 1) $E \rightarrow E_1 + T$ $\{ E.val = E_1.val + T.val \}$
- 2) $E \rightarrow T$ $\{ E.val = T.val \}$
- 3) $T \rightarrow T_1 * digit$ $\{ T.val = T_1.val * digit.lexval \}$
- 4) $T \rightarrow digit$ $\{ T.val = digit.lexval \}$

程序片段：

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

$E \rightarrow T$

$T \rightarrow T_1 * digit$ {stack[top-2].val=stack[top-2].val*digit.lexval;
top=top-2}

$T \rightarrow digit$

状态	ACTION				GOTO	
	d	+	*	#	E	T
0	S ₃				1	2
1		S ₄		acc		
2		r ₂	S ₅	r ₂		3
3		r ₄	r ₄	r ₄		
4	S ₃					7
5	S ₆					
6		r ₃	r ₃	r ₃		
7		r ₁	S ₅	r ₁		

2. S-属性翻译方案

- 1) $E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }
- 2) $E \rightarrow T$ { $E.val = T.val$ }
- 3) $T \rightarrow T_1 * digit$ { $T.val = T_1.val * digit.lexval$ }
- 4) $T \rightarrow digit$ { $T.val = digit.lexval$ }

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

$E \rightarrow T$

$T \rightarrow T_1 * digit$ { $stack[top-2].val = stack[top-2].val * digit.lexval;$
 $top = top - 2$ }

$T \rightarrow digit$

状态	ACTION				GOTO	
	d	+	*	#	E	T
0	S ₃				1	2
1		S ₄		acc		
2		r ₂	S ₅	r ₂		3
3		r ₄	r ₄	r ₄		
4	S ₃					7
5	S ₆					
6		r ₃	r ₃	r ₃		
7		r ₁	S ₅	r ₁		

分析并计算2+3*5的过程:

步骤	状态栈	语义栈	符号栈	剩余输入串	Action	GOTO
0	0	-	#	2 + 3*5 #	S ₃	
1	03	- -	#2	+ 3*5 #	r ₄	2
2	02	-2	#T	+ 3*5 #	r ₂	1
3	01	-2	#E	+ 3*5 #	S ₄	
4	014	-2-	#E+	3*5 #	S ₃	
5	0143	-2- -	#E+3	*5 #	r ₄	7
6	0147	-2-3	#E+T	*5 #	S ₅	
7	01475	-2-3-	#E+T*	5 #	S ₆	
8	014756	-2-3--	#E+T*5	#	r ₃	7
9	0147	-2-15	#E+T	#	r ₁	1
10	01	-17	#E	#	acc	39