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

郑馥丹
zhengfd5@mail.sysu.edu.cn

CONTENTS

目 录

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

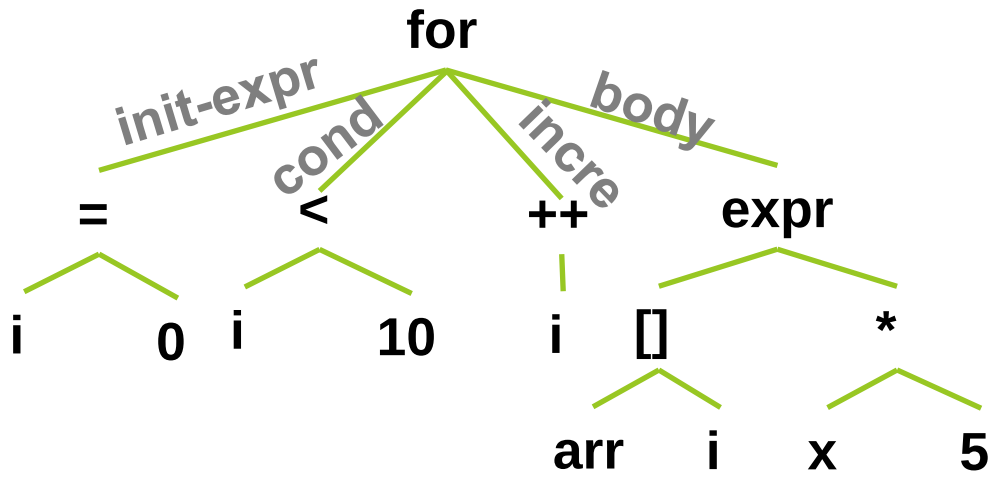
1. 语法制导翻译[Syntax-Directed Translation]

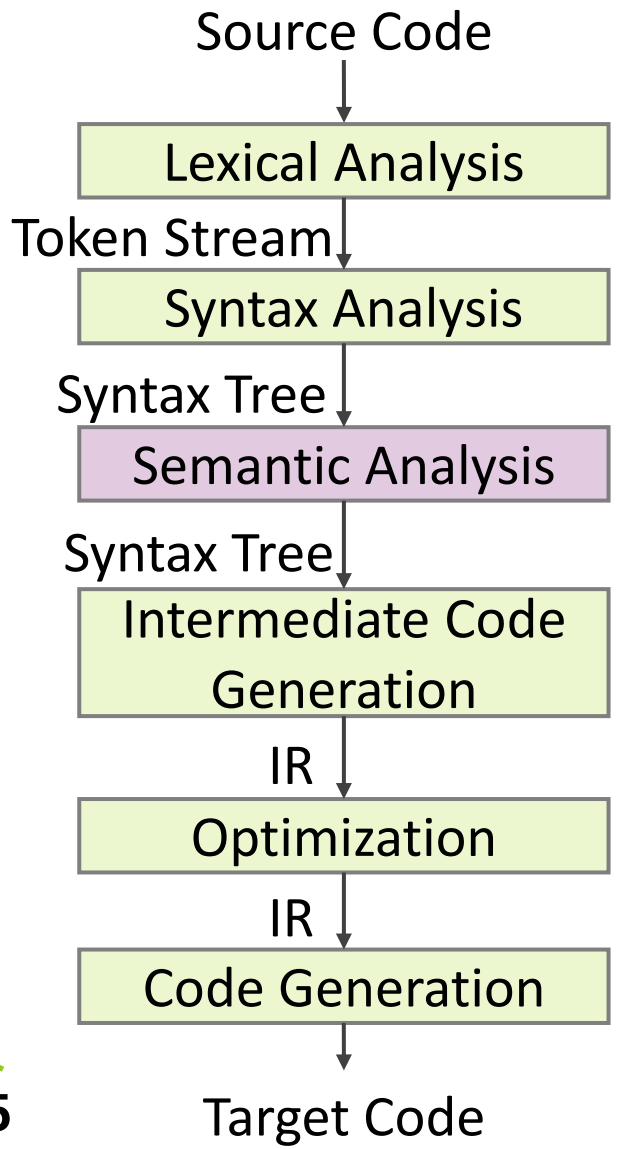
Semantic Analysis[语义分析]

- 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

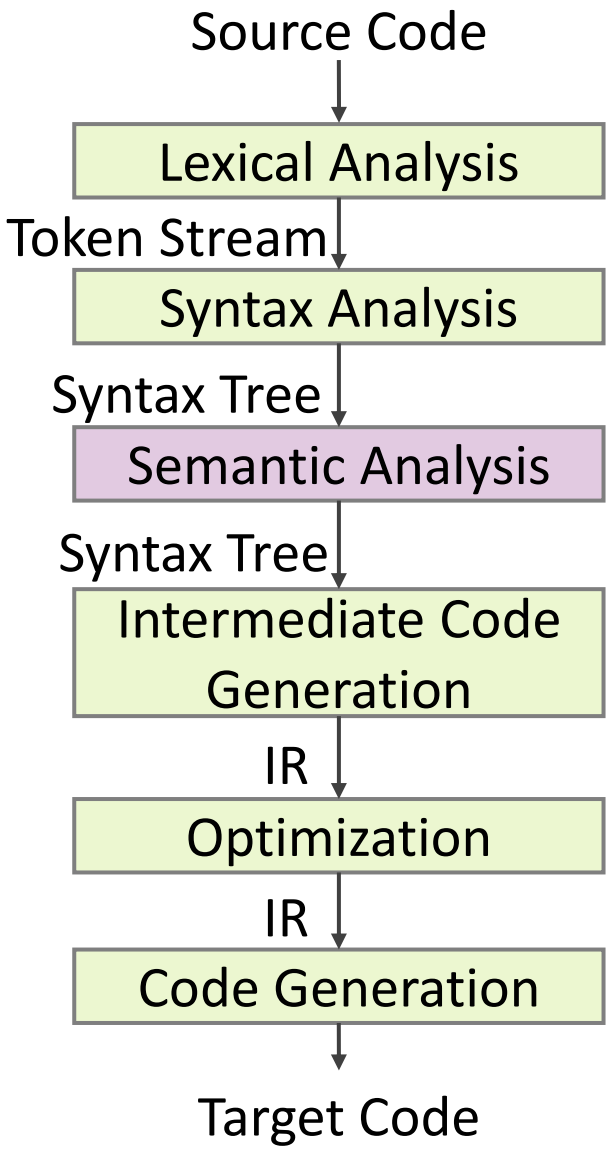
    声明和使用的函数没有定义

    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?

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

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(…)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

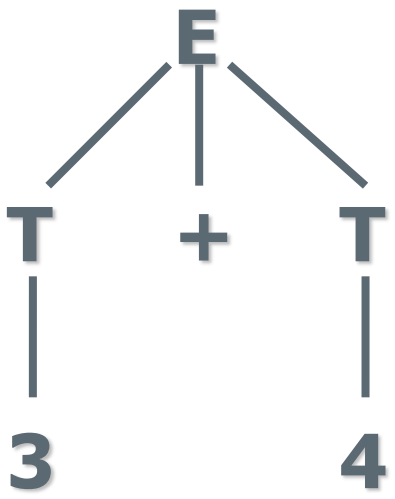
2. 语法制导定义

- 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

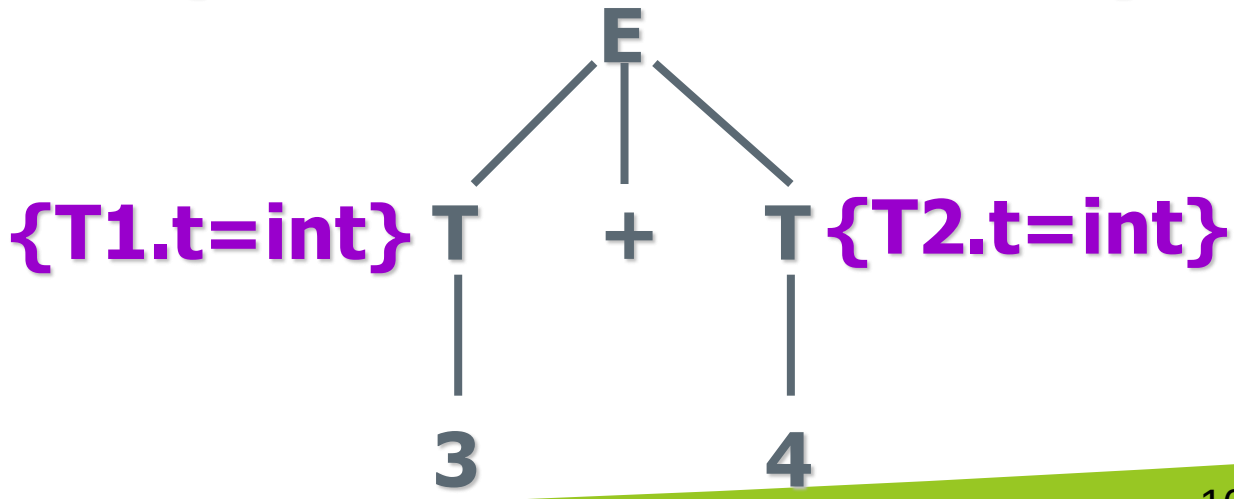
- 例1：类型审查的属性文法
  - E→T1+T2　　{T1.t = int AND T2.t = int}
  - E→T1 or T2　{T1.t = bool AND T2.t = bool}
  - T→num　　{T.t = int}
  - T→true　　{T.t = bool}
  - T→false　　{T.t = bool}

对输入串3+4的语法树：



类型审查

{T1.t=int AND T2.t=int}



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，如果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→E **n**　　{ L.val = E.val, Print(L.val) } (**n**表示表达式的结尾标记)
  - E→E1+T　　{ E.val = E1.val +T.val }
  - E→T　　{ E.val = T.val }
  - T→T1*F　　{ T.val = T1.val * F.val }
  - T→F　　{ T.val = F.val }
  - F→(E)　　{ F.val = E.val }
  - F→digit　　{ F.val = digit.lexval }

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

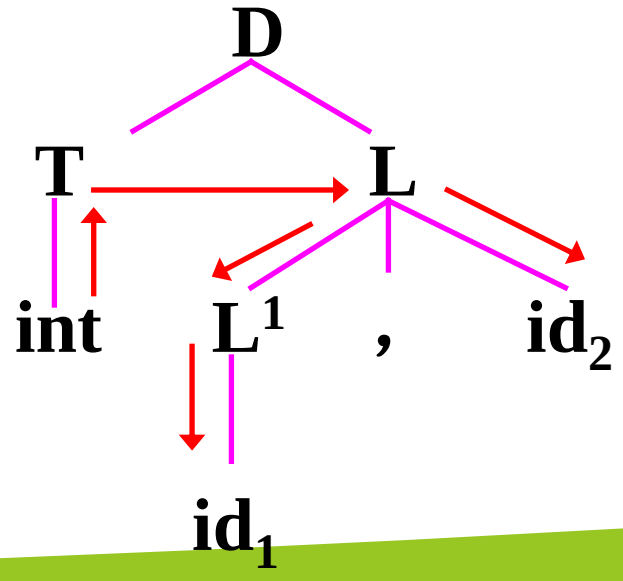
- 例3：描述变量类型说明的属性文法
  - D→TL　　{ L.in = T.type }
  - T→int　　{ T.type = int }
  - T→real　　{ T.type = real }
  - L→L1,id　　{ L1.in = L.in； addtype(id.entry， L.in)}
  - L→id　　{ addtype(id.entry， L.in)}

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



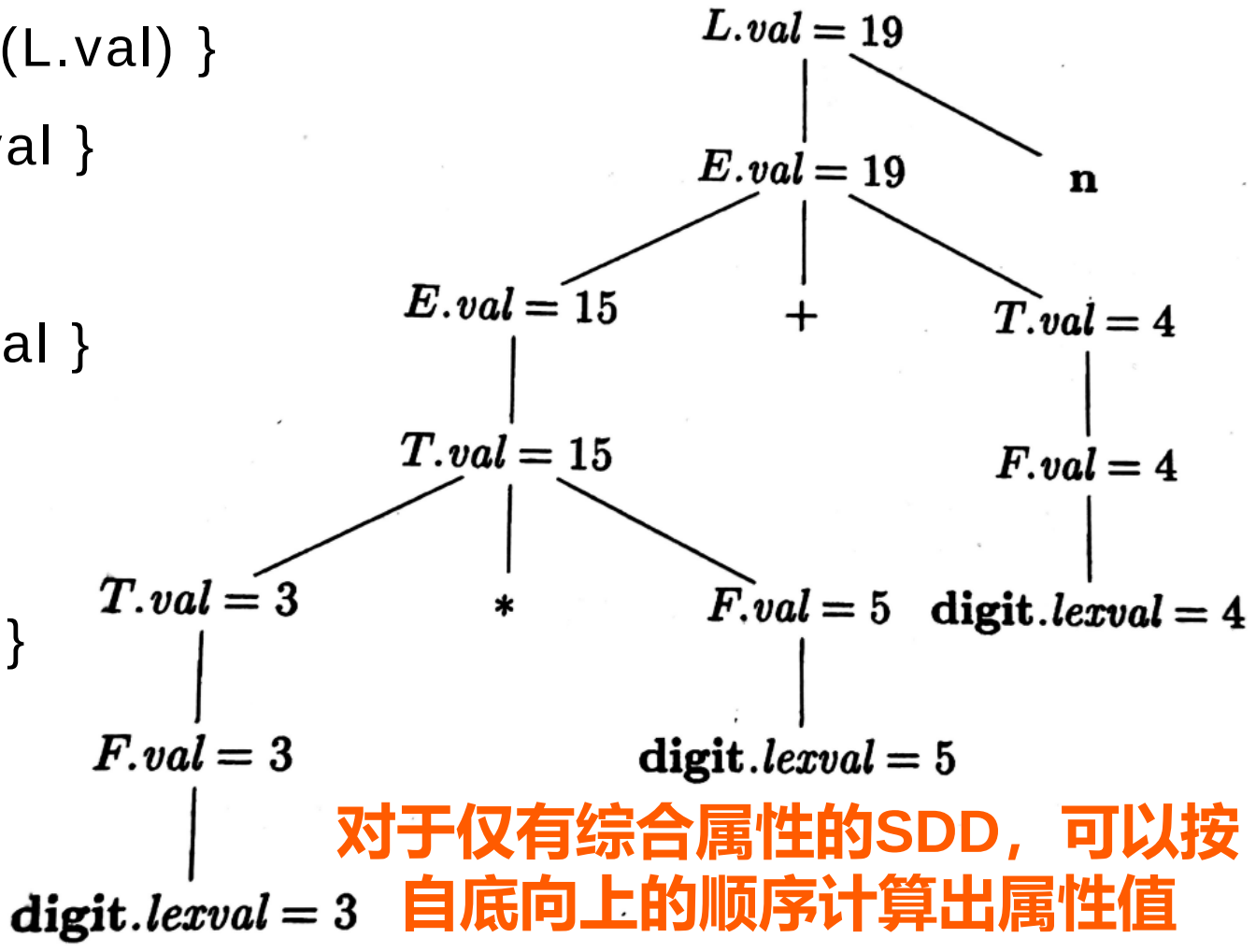
2. 语法制导定义

- 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→E **n**　　{ L.val = E.val, Print(L.val) }
  - E→E1+T　　{ E.val = E1.val +T.val }
  - E→T　　{ E.val = T.val }
  - T→T1*F　　{ T.val = T1.val * F.val }
  - T→F　　{ T.val = F.val }
  - F→(E)　　{ F.val = E.val }
  - F→digit　　{ F.val = digit.lexval }

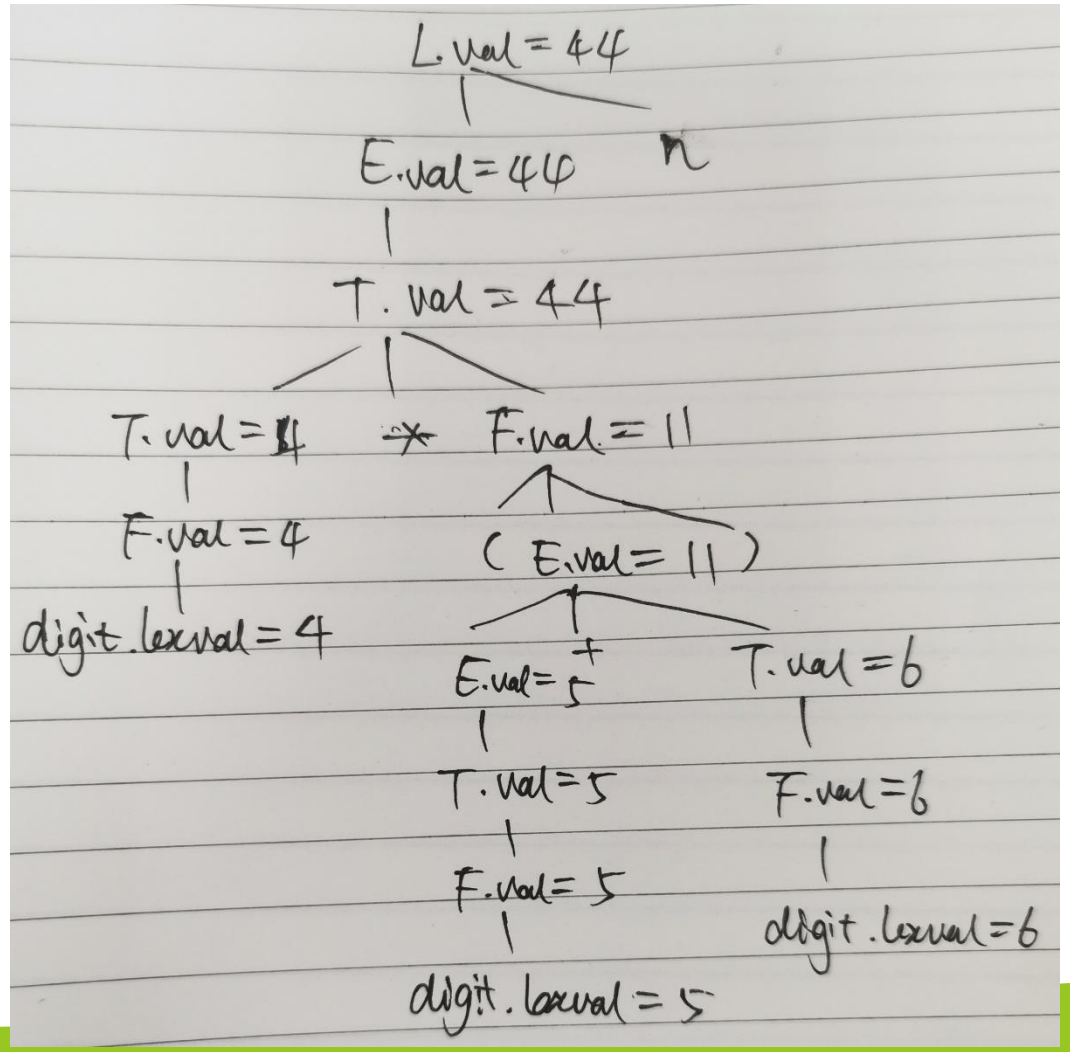
对应的3*5+4n的注释语法分析树：



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

随堂练习（1）

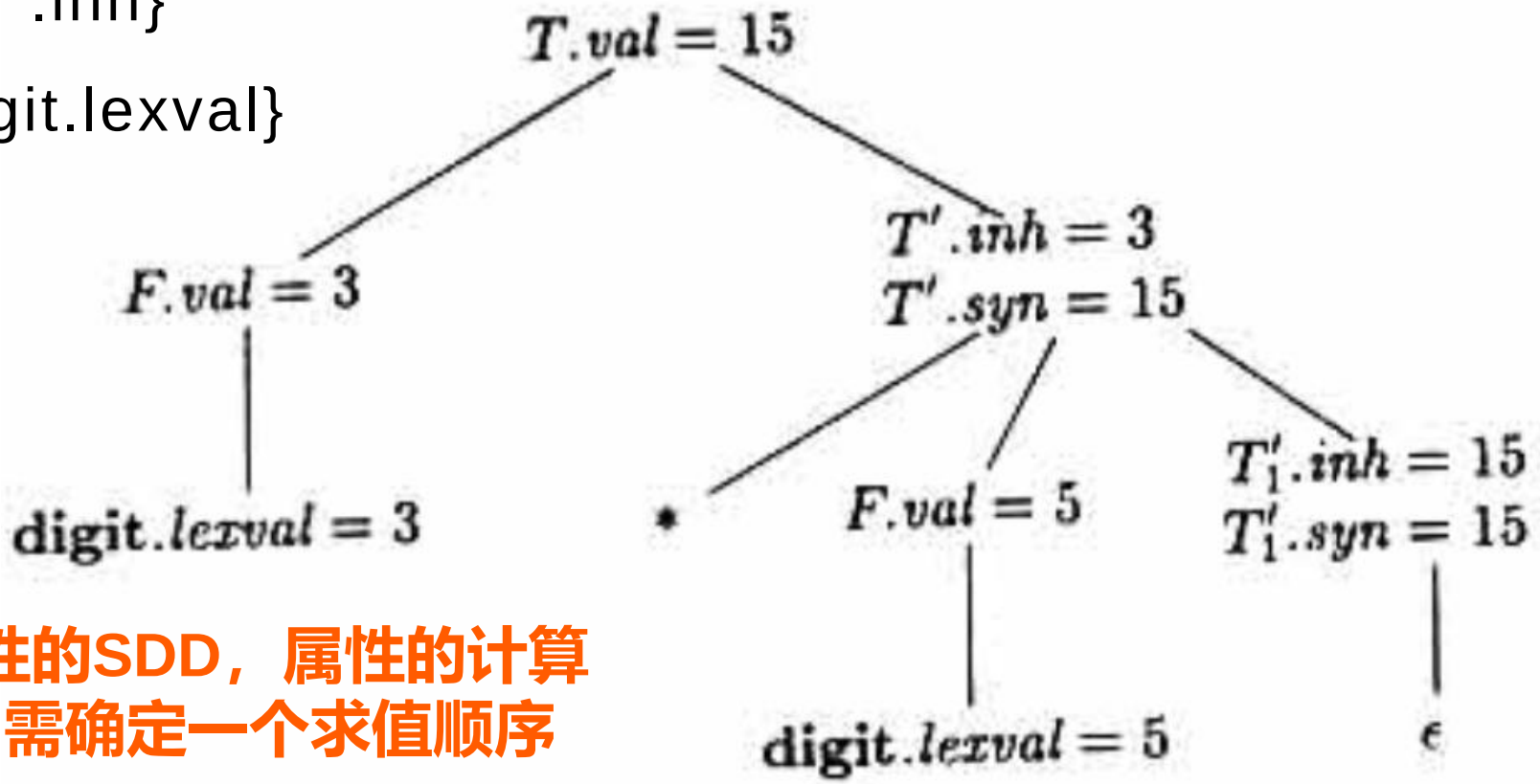
- 对以下简单算术表达式求值的属性文法，给出4*(5+6)n的注释语法分析树：
  - L→E **n**　　{ L.val = E.val, Print(L.val) } (**n**表示表达式的结尾标记)
  - E→E1+T　　{ E.val = E1.val +T.val }
  - E→T　　{ E.val = T.val }
  - T→T1*F　　{ T.val = T1.val * F.val }
  - T→F　　{ T.val = F.val }
  - F→(E)　　{ F.val = E.val }
  - F→digit　　{ F.val = digit.lexval }



3. 注释语法分析树[Annotated Parse Tree]

- 例：属性文法
  - T→FT’　　{**T’.inh=F.val**，T.val=T’.syn}
  - T’→*FT_1’　　{**T_1’.inh=T’.inh×F.val**，T’.syn=T_1’.syn}
  - T’→ε　　{T’.syn=T’.inh}
  - F→digit　　{F.val=digit.lexval}

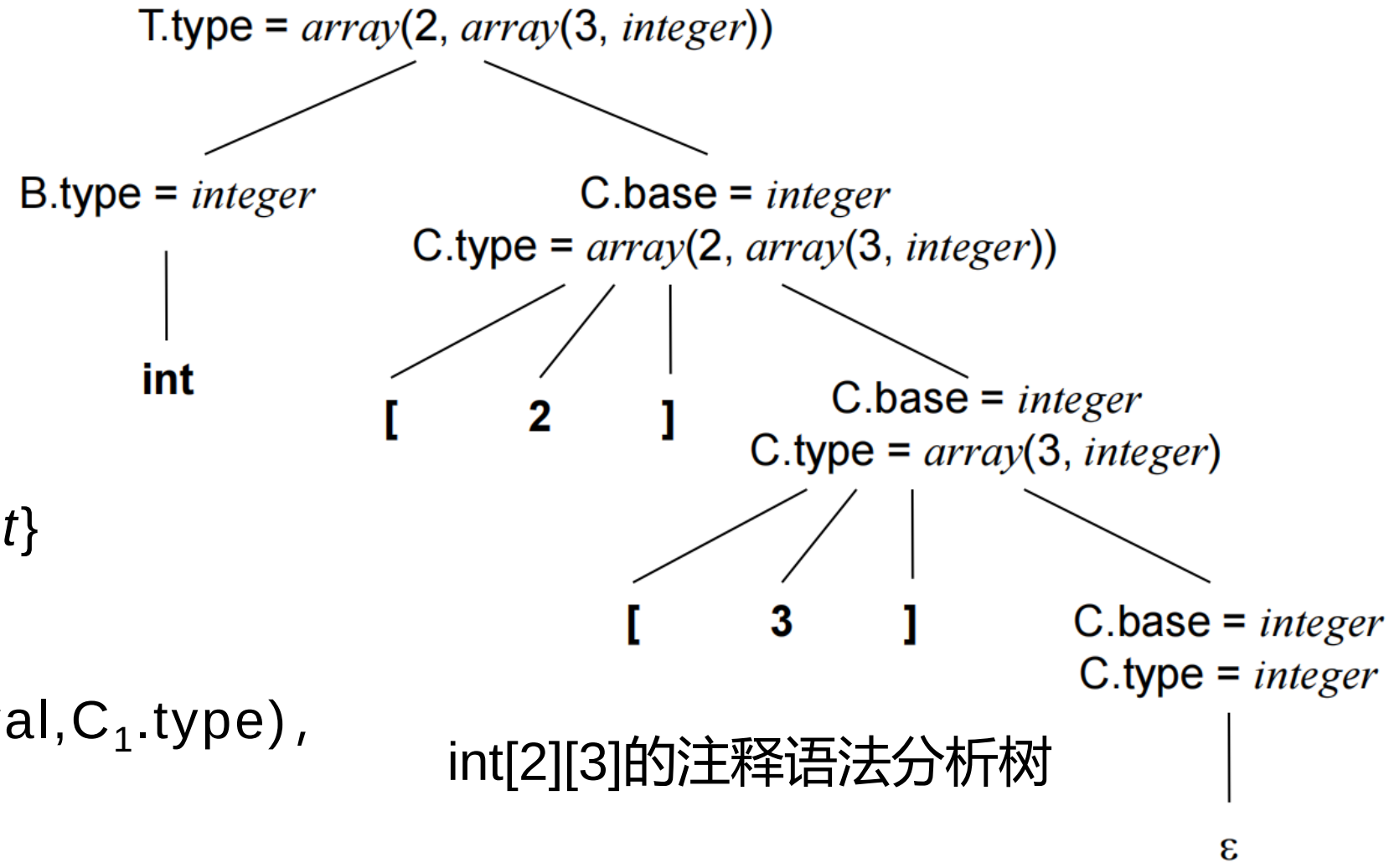
3*5的注释语法分析树：



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→BC
    { T.type=C.type，
      C.base=B.type }
  - B→int { B.type=*int* }
  - B→float { B.type=*float*}
  - C→[**num**]C_1
    { C.type=*array*(**num**.val,C_1.type)，
      C_1.base=C.base }
  - C→ε { C.type=C.base}



int[2][3]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]

- 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

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

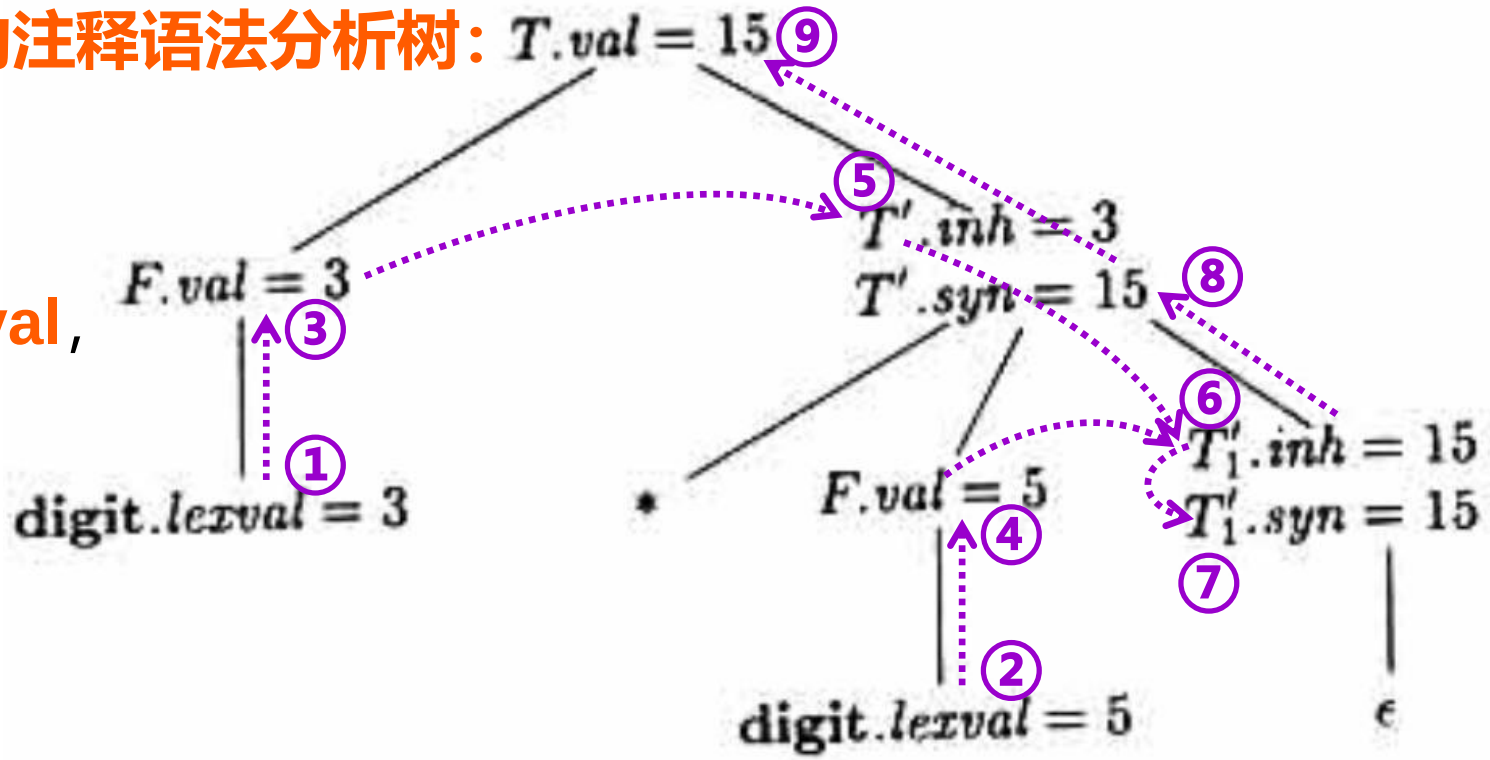
T→FT’ ⑤{ T’.inh=F.val,
⑨ T.val=T’.syn }

T’→*FT_1’ ⑥{ T_1’.inh=T’.inh×F.val,
⑧ T’.syn=T_1’.syn }

T’→ε ⑦{ T’.syn=T’.inh }

F→digit ③④{ F.val=digit.lexval }

3*5的注释语法分析树：

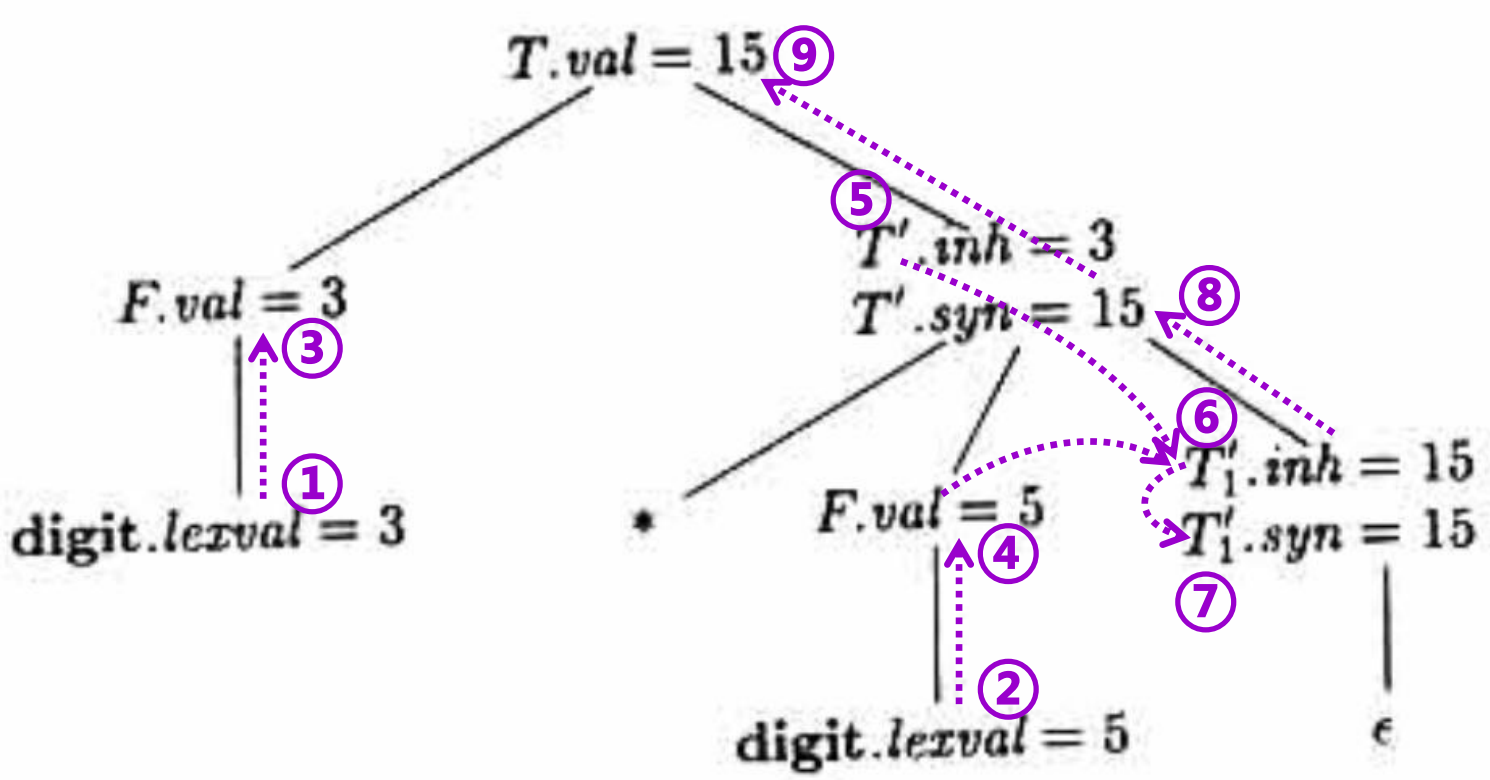


2. 属性求值的顺序

- 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$N_1,N_2,\ldots N_k$，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]

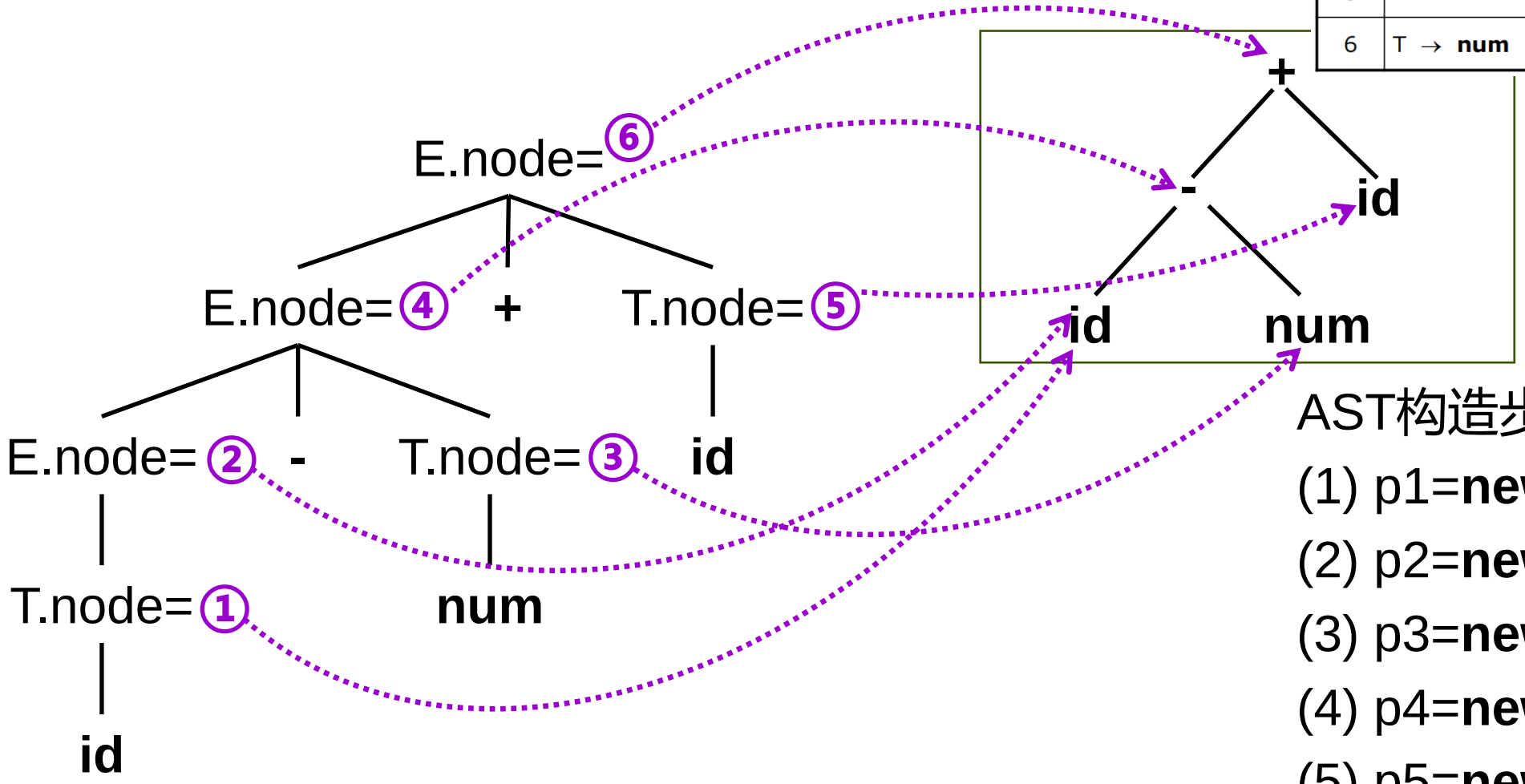
- 例：文法

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



AST构造步骤：

(1) p1=**new** Leaf(**id**, entry-a)

(2) p2=**new** Leaf(**num**, 4)

(3) p3=**new** Node('**-**', p1, p2)

(4) p4=**new** Leaf(**id**, entry-c)

(5) p5=**new** Node('**+**', p3, p4)

4. S-属性的定义

- 一个仅包含综合属性[Synthesized 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→E1+T{E.val=E1.val+T.val}
  - E→T{E.val=T.val}
  - T→num{T.val=num.val}

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→3	计算T.val = 3
归约E→T	计算E.val=T.val=3
扫描+，移进	
扫描5，归约T→5	计算T.val = 5
归约E→E1+T	计算E.val = 3 + 5 = 8

- 例：非S-属性SDD
  - 若SDD中包含继承属性，如： A→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

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$A \rightarrow X_1 X_2 \ldots X_n$，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→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→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→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

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**

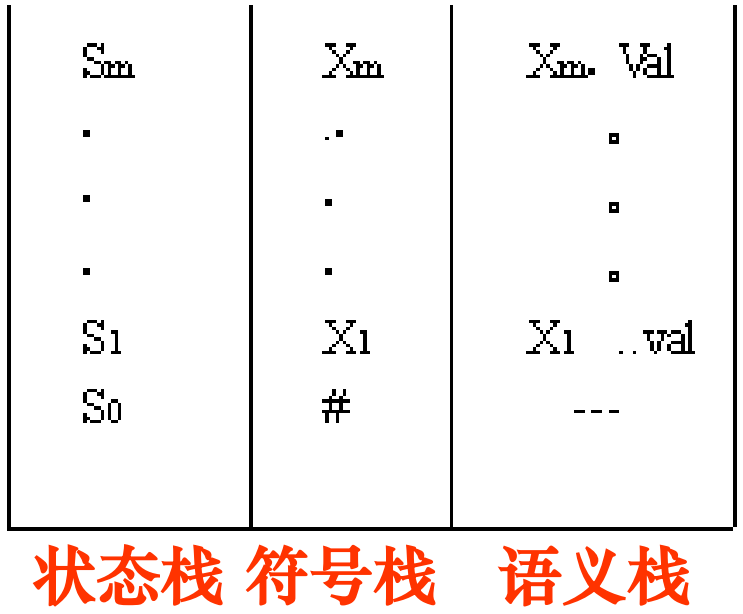
1. 语法制导的翻译方案

- 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：



2. S-属性翻译方案

- 例：简单算术表达式求值的属性文法

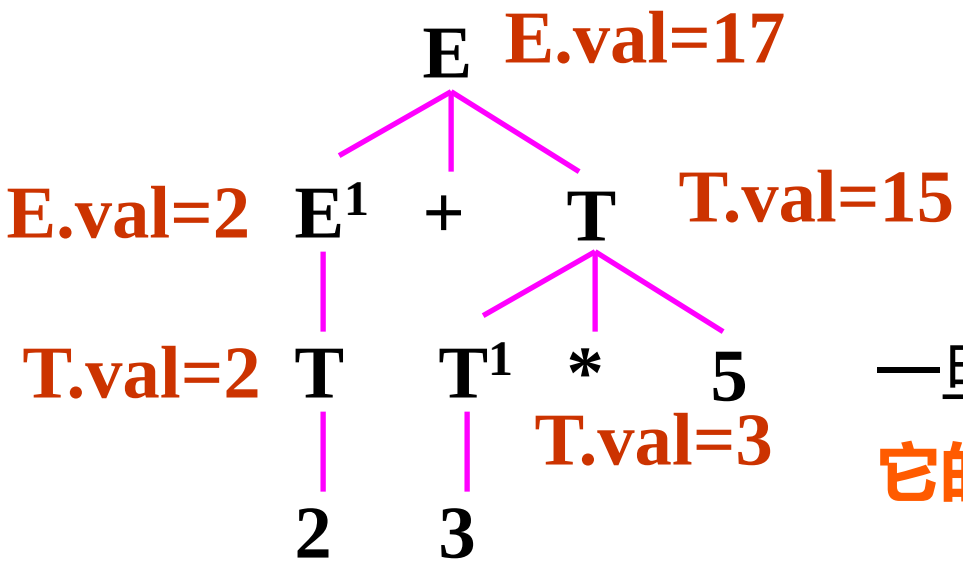
1）E→E1+T　　{ E.val = E1.val +T.val }

2）E→T　　{ E.val = T.val }

3）T→T1*digit　{ T.val = T1.val * digit.lexval }

4）T→digit　　{ 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→E1+T　　{ E.val = E1.val +T.val }

  2）E→T　　{ E.val = T.val }

  3）T→T1*digit　{ T.val = T1.val * digit.lexval }

  4）T→digit　　{ T.val = digit.lexval }

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		

程序片段：

```
E→E1+T {stack[top-2].val=stack[top-2].val+stack[top].val;
        top=top-2}
E→T
T→T1*digit  {stack[top-2].val=stack[top-2].val*digit.lexval;
             top=top-2}
T→digit
```

2．S-属性翻译方案

1）E→E1+T　{ E.val = E1.val +T.val }
2）E→T　　{ E.val = T.val }
3）T→T1*digit { T.val = T1.val * digit.lexval}
4）T→digit　　{ T.val = digit.lexval }

```
E→E1+T {stack[top-2].val=stack[top-2].val+stack[top].val;
        top=top-2}
E→T
T→T1*digit  {stack[top-2].val=stack[top-2].val*digit.lexval;
             top=top-2}
T→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	

CONTENTS

目 录

SDD vs. SDT

- **SDD[语法制导定义]**: 是CFG的推广，翻译的高层次规则说明
  - 上下文无关文法CFG+属性+语义规则
  - 其中无副作用[side effects]的部分称为**属性文法[attribute grammars]**
  - 语义规则中的**属性计算没有顺序**
- **SDT[语法制导翻译方案]**: SDD的补充，具体翻译实施方案
  - **程序片段附加在产生式的不同位置**
  - 语义规则中的**属性计算顺序很重要**

CFG	SDD	SDT
D -> T L	L.*inh* = T.*type*	D -> T { L.*inh* = T.*type* } L
T -> int	T.*type* = int	T -> int { T.*type* = int }
T -> float	T.*type* = float	T -> float { T.*type* = float }
L -> L_1, id	L_1.*inh* = L.*inh*	L -> { L_1.*inh* = L.*inh* }L_1, id
L -> id	id.*type* = L.*inh*	L -> { id.*type* = L.*inh*} id

1. L-属性定义

- 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$A \rightarrow X_1X_2...X_n$，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 **不依赖于父亲节点的综合属性**
    - ✓ 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） **不依赖于右兄弟的任何属性**
    - ✓ 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

不能有循环依赖

2. L-属性SDD的求值顺序

- L-属性SDD的求值顺序
  - 与**深度优先[Depth-First]**访问语法分析树相同
  - 与**自顶向下**解析的顺序相同

```
void dfvisit(n: Node) {
    for (each child m of n, from left to right) {
        evaluate inherited attributes of m;
        dfvisit(m);
    }
    evaluate synthesized attributes of n;
}
```

3. L-属性翻译方案

- 翻译方案与L-属性定义
  - 翻译方案中的**显式**求值顺序
  - 按照**从左到右深度优先**的顺序执行语义操作

3. L-属性翻译方案

- 语法制导翻译方案[Syntax-Directed Translation Schemes, **SDT**]
  - 后缀翻译方案[Postfix translation scheme]/**后缀SDT**：所有语义动作都在**产生式的最右端**

例：简单算术表达式求值的属性文法

1）E→E1+T　　{ E.val = E1.val +T.val }

2）E→T　　{ E.val = T.val }

3）T→T1*digit　{ T.val = T1.val * digit.lexval }

4）T→digit　　{ T.val = digit.lexval }

3. L-属性翻译方案

- 语法制导翻译方案[Syntax-Directed Translation Schemes, **SDT**]
  - 产生式内部带有语义动作的SDT：语义动作可以放置在**产生式右部的任何位置**上[Actions inside productions]
    - ✓ 若有产生式B→X**{a}**Y：
      - 如果X是终结符，则识别到X后，动作a就会执行
      - 如果X是非终结符，则识别到所有从X推导出的终结符后，动作a就会执行
      - 如果语法分析是自底向上的，则在X出现在分析栈的栈顶时，动作a就会执行
      - 如果语法分析是自顶向下的，则在推导展开Y（Y为非终结符）或者在输入中检测到Y（Y为终结符）之前，动作a就会执行

3. L-属性翻译方案

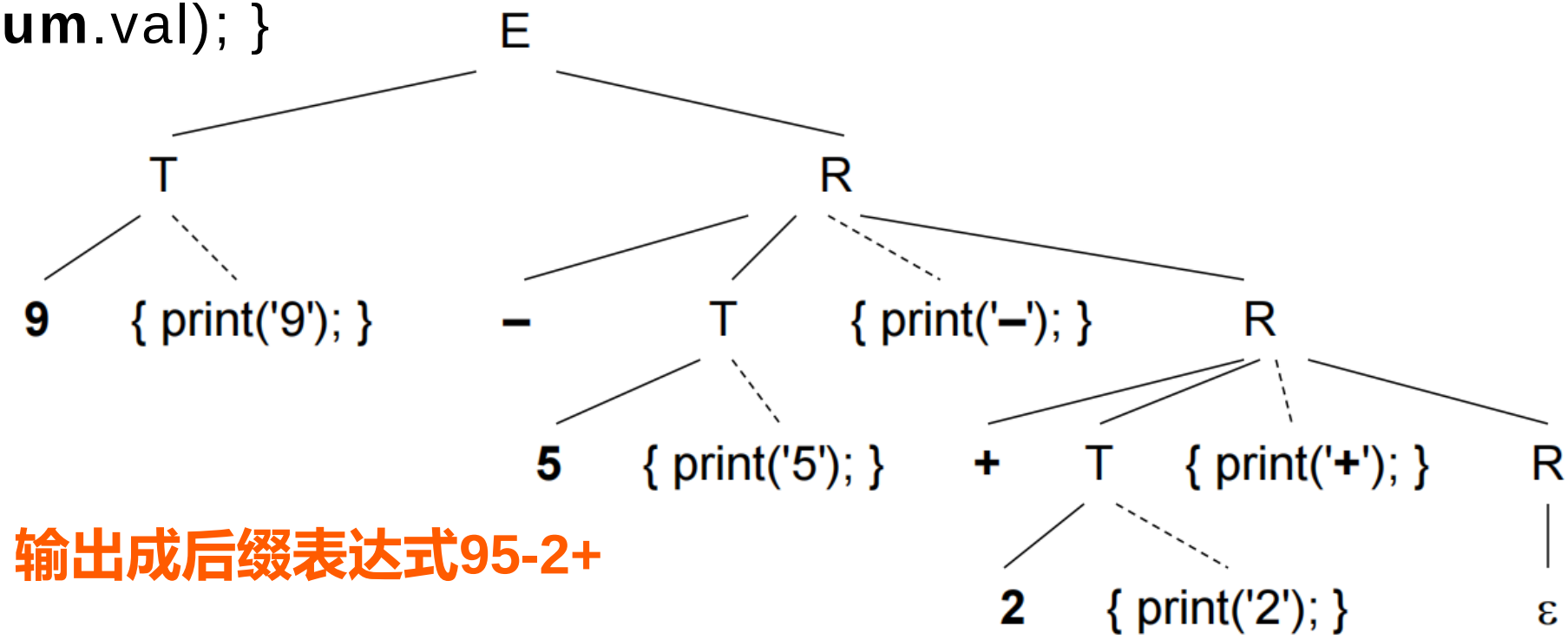
- 例：将中缀表达式转成后缀表达式的SDT

E → TR

R → **addop** T **{ print(addop.lexeme); }** R_1　　**产生式内部带有语义动作**

R → ε

T → **num** { print(**num**.val); }



将中缀表达式9-5+2，输出成后缀表达式95-2+

3. L-属性翻译方案

- 注意：**不是所有的SDT都可以在语法分析过程中实现**
- 例：将中缀表达式转成前缀表达式的SDT

L → E **n**

E → **{ print('+'); }** E1 + T

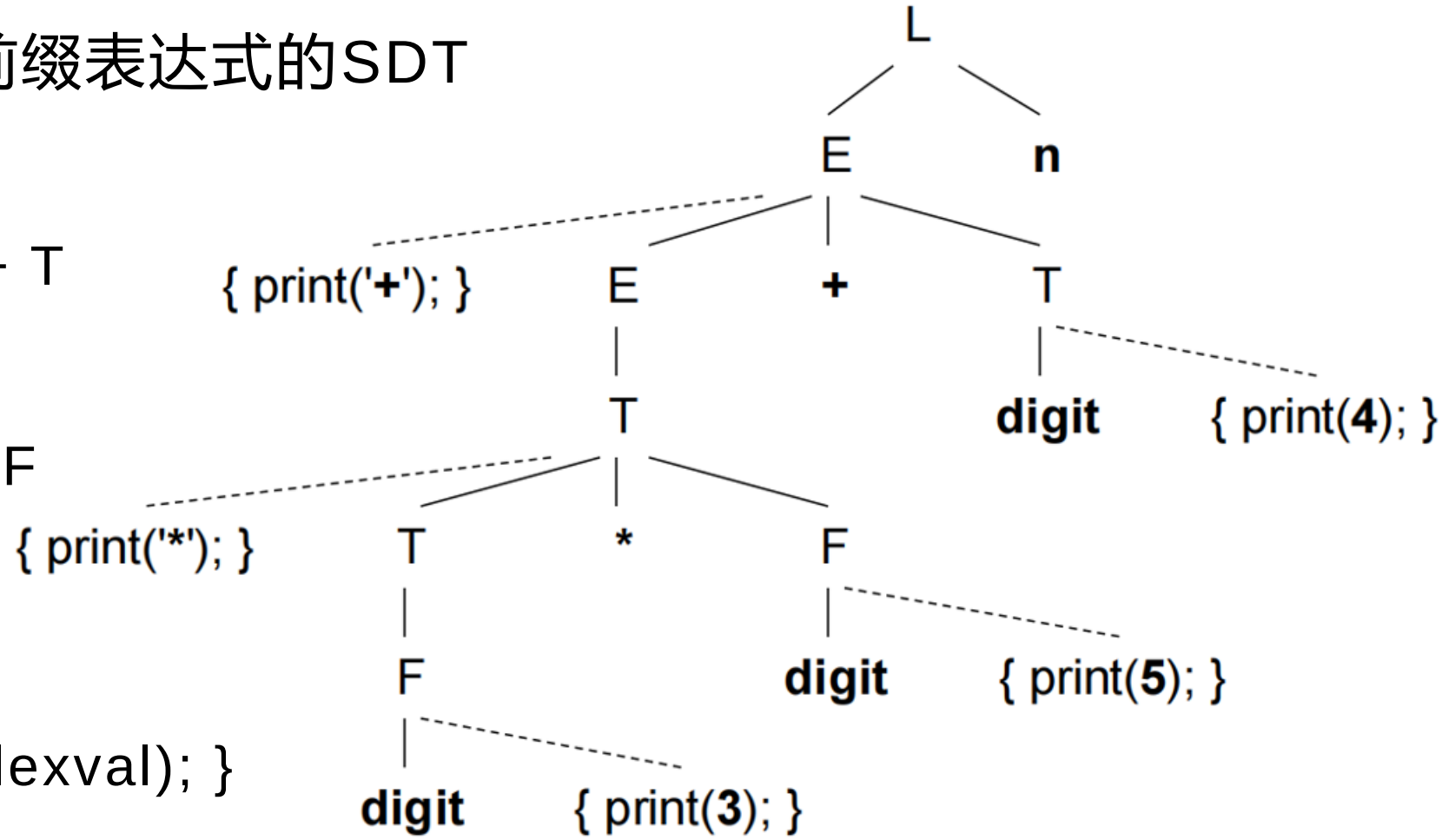
E → T

T → **{ print('*'); }** T1 * F

T → F

F → (E)

F → **digit** { print(**digit**.lexval); }

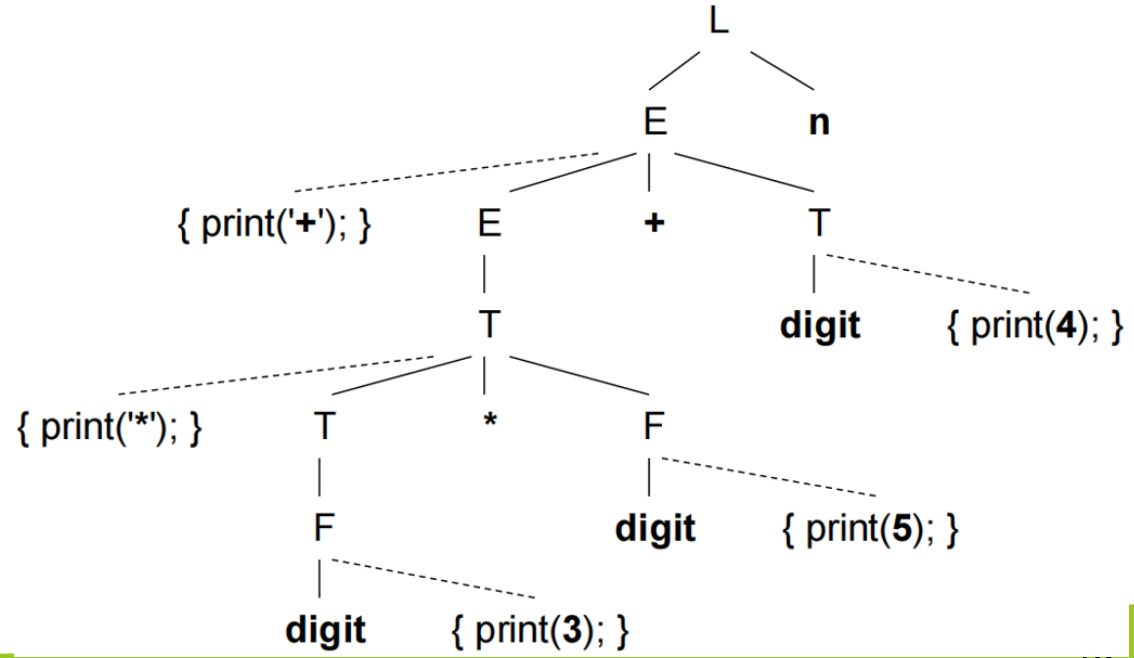


将中缀表达式3*5+4，输出成前缀表达式+*354

3. L-属性翻译方案

- 注意：**不是所有的SDT都可以在语法分析过程中实现**
- 例：将中缀表达式转成前缀表达式的SDT
  - 该文法无法在语法分析过程中实现
  - 存在问题：
    - ✓ **动作时机过早**：位于产生式最开始，语法分析器必须在看到整个表达式之前就决定打印哪个运算符
    - ✓ 自底向上：在预测使用哪个产生式之前就需要执行打印动作
    - ✓ 自顶向下：在知道运算符是什么之前就需要执行打印动作

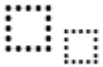
L → E **n**
E → { print('+'); } E1 + T
E → T
T → { print('*'); } T1 * F
T → F
F → (E)
F → **digit** { print(**digit**.lexval); }



4. L-属性的SDD转换为SDT

- 将一个L-属性的SDD转换为一个SDT的规则：
  - 把计算某个**非终结符A的继承属性的动作**插入到产生式体中紧靠在**A**的本次**出现之前**的位置上；如果A的多个继承属性以无环的方式相互依赖，就需要对这些属性的求值动作进行排序，以便先计算需要的属性
  - 将计算一个**产生式头的综合属性的动作**放置在这个**产生式体的最右端**

4. L-属性的SDD转换为SDT

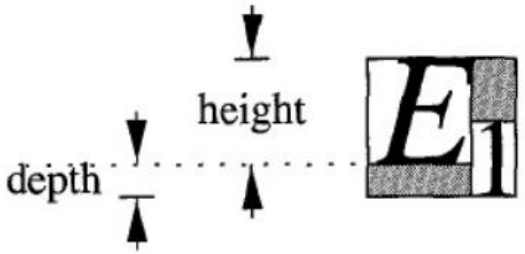
- 例：数学公式排版文法 $B \rightarrow B_1 B_2 | B_1 \textbf{sub} B_2 | (B_1) | \textbf{text}$　　B代表box
  - 两个并排的box： 
  - 两个box，一个是另一个的下标： 
  - 一个用括号括起来的box：()
  - 一串普通的文本串：**max**(...)
  - 约定：
    - **sub下标是右结合**
    - **并排关系是右结合**
    - **sub的优先级高于并排关系**

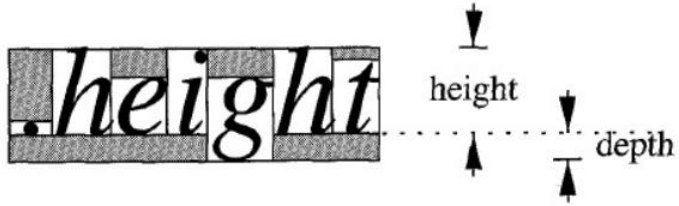
4. L-属性的SDD转换为SDT

- 例：数学公式排版文法 $B \rightarrow B_1 B_2 | B_1 \textbf{sub} B_2 | (B_1) | \textbf{text}$

Productions	Semantic Actions
1) $S \rightarrow B$	$B.ps = 10$
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) $B \rightarrow B_1 \textbf{sub} B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps \times 70\%$ $B.ht = \max(B_1.ht, B_2.ht - B.ps \times 25\%)$ $B.dp = \max(B_1.dp, B_2.dp + B.ps \times 25\%)$
4) $B \rightarrow (B_1)$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5) $B \rightarrow \textbf{text}$	$B.ht = getHight(B.ps, \textbf{text}.lexval)$ $B.dp = getDepth(B.ps, \textbf{text}.lexval)$

ps = point size
ht = height
dp = depth





4. L-属性的SDD转换为SDT

- 例：数学公式排版文法 $B \rightarrow B_1 B_2 | B_1 \textbf{sub} B_2 | (B_1) | \textbf{text}$

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); }

- 把计算某个非终结符A的继承属性的动作插入到产生式体中紧靠在A的本次出现之前的位置
- 将计算一个产生式头的综合属性的动作放置在这个产生式体的最右端

5. 在预测分析中实现L-属性定义

- 进行递归下降预测翻译的步骤

  1）为语法规则编写一个LL(1)语法

  2）通过附加语义规则定义L-属性定义

  3）将L-属性定义转换为翻译方案

  4）消除翻译方案中的左递归（因为**带有左递归的文法无法进行确定的自顶向下语法分析**）

  5）编写一个递归下降预测解析器（翻译器）

5. 在预测分析中实现L-属性定义

4）消除翻译方案中的左递归

① 最简单的情况：SDD中的语义动作只涉及打印输出，而不涉及计算

– 此时，可将该动作当成终结符处理，然后使用消除左递归的通用方法

– 例：

$E \rightarrow E1 + T$ **{ print('+'); }**

$E \rightarrow T$

$E \rightarrow T\ R$

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

$R \rightarrow \varepsilon$

✓ **引入新的非终结符A’**，将产生式$A \rightarrow A\alpha|\beta$改写成：$A \rightarrow \beta A'$，$A' \rightarrow \alpha A'|\varepsilon$

5. 在预测分析中实现L-属性定义

4）消除翻译方案中的左递归

② 若SDD中的语义动作涉及计算，且SDD是S-属性的

– 此时，可以通过将计算属性值的动作放在新产生式中的适当位置上来构造出一个SDT

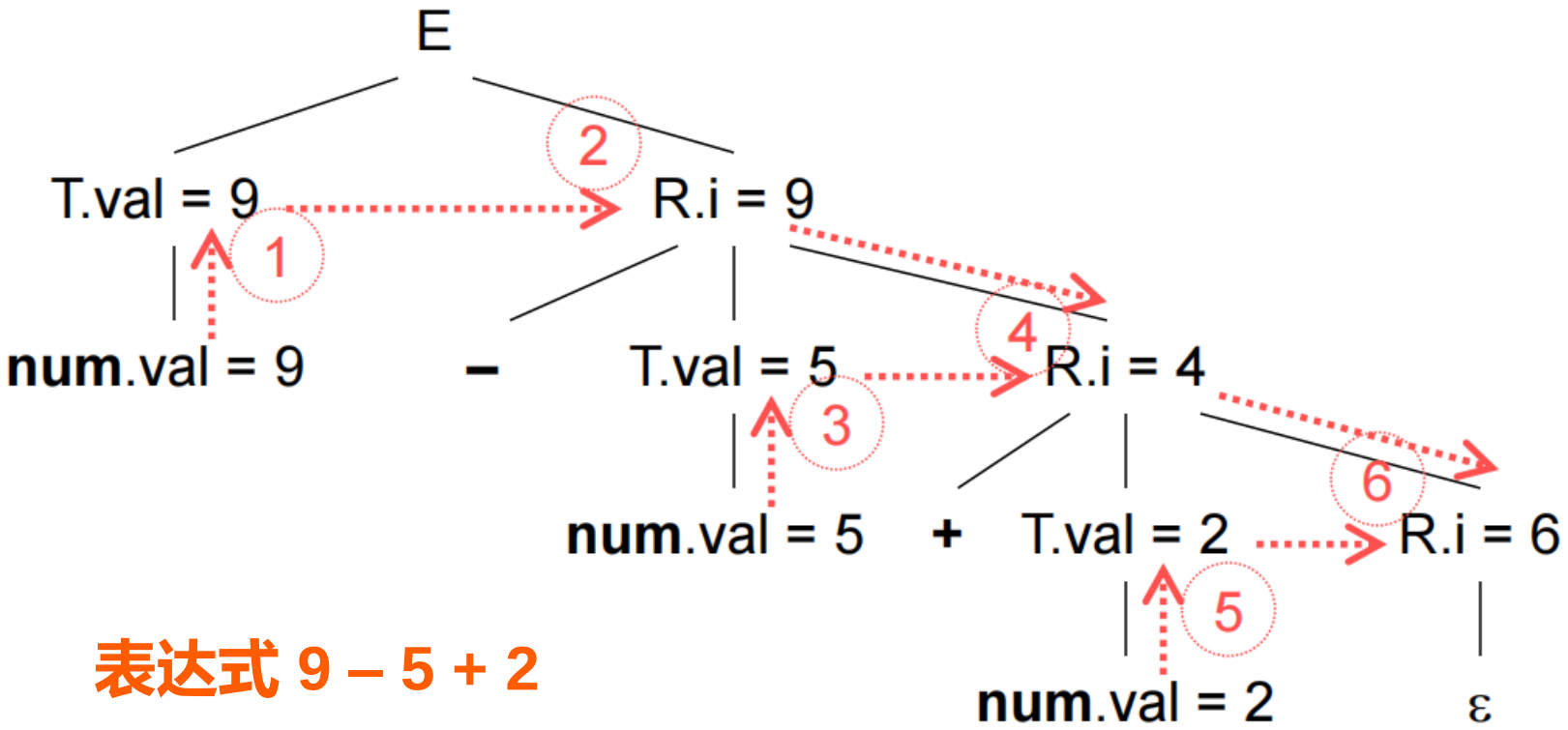
E → E1 **+** T **{ E.val = E1 .val + T.val; }**
E → E1 **–** T **{ E.val = E1 .val – T.val; }**
E → T **{ E.val = T.val; }**
T → **(** E **)** **{ T.val = E.val; }**
T → **num** **{ T.val = num.val; }**

E → T { R.i = T.val; }
R { E.val = R.s; }
R → **+** T { R1.i = R.i + T.val; }
R1 { R.s = R1.s; }
R → **–** T { R1.i = R.i – T.val; }
R1 { R.s = R1.s; }
R → ε { R.s = R.i; }
T → **(** E **)** { T.val = E.val; }
T → **num** { T.val = **num**.val; }

5. 在预测分析中实现L-属性定义

E → E1 + T { E.val = E1 .val + T.val; }
E → E1 – T { E.val = E1 .val – T.val; }
E → T { E.val = T.val; }
T → (E) { T.val = E.val; }
T → num { T.val = num.val; }

E → T { R.i = T.val; } R { E.val = R.s; }
R → + T { R1.i = R.i + T.val; } R1 { R.s = R1.s; }
R → – T { R1.i = R.i – T.val; } R1 { R.s = R1.s; }
R → ε { R.s = R.i; }
T → (E) { T.val = E.val; }
T → num { T.val = num.val; }



表达式 9 – 5 + 2

解析9-5+2：

1. T → **num**: T.val = **num**.val=9;
2. E → T R: R.i = T.val=9;
3. T → **num**: T.val = **num**.val=5;
4. R → – T R1: R1.i = R.i – T.val =9-5=4;
5. T → **num**: T.val = **num**.val=2;
6. R → + T R1: R1.i = R.i + T.val =4+2=6;
7. R → R.s = R.i=6;
8. E.val = R.s=6;

5. 在预测分析中实现L-属性定义

4）消除翻译方案中的左递归

② 若SDD中的语义动作涉及计算，且SDD是S-属性的

– 此时，可以通过将计算属性值的动作放在新产生式中的适当位置上来构造出一个SDT

– 通用的解决方案

A → A1 Y **{ A.a = g(A1.a, Y.y); }**

A → X **{ A.a = f(X.x); }**

A.a，X.x和Y.y均为综合属性

A → X **{ R.i = f(X.x); }** R **{ A.a = R.s; }**

R → Y **{ R1.i = g(R.i, Y.y); }** R1 **{ R.s = R1.s; }**

R → ε **{ R.s = R.i; }**

R.i为继承属性，R.s为综合属性

5. 在预测分析中实现L-属性定义

4）消除翻译方案中的左递归

A → A1 Y { A.a = g(A1.a, Y.y); }

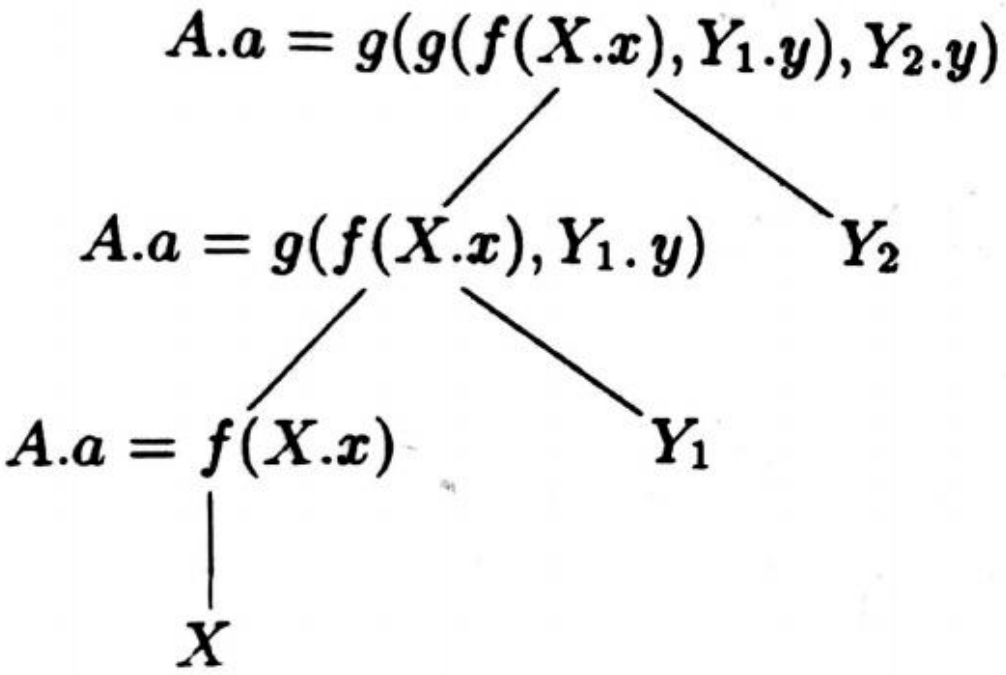
A → X { A.a = f(X.x); }

⇒

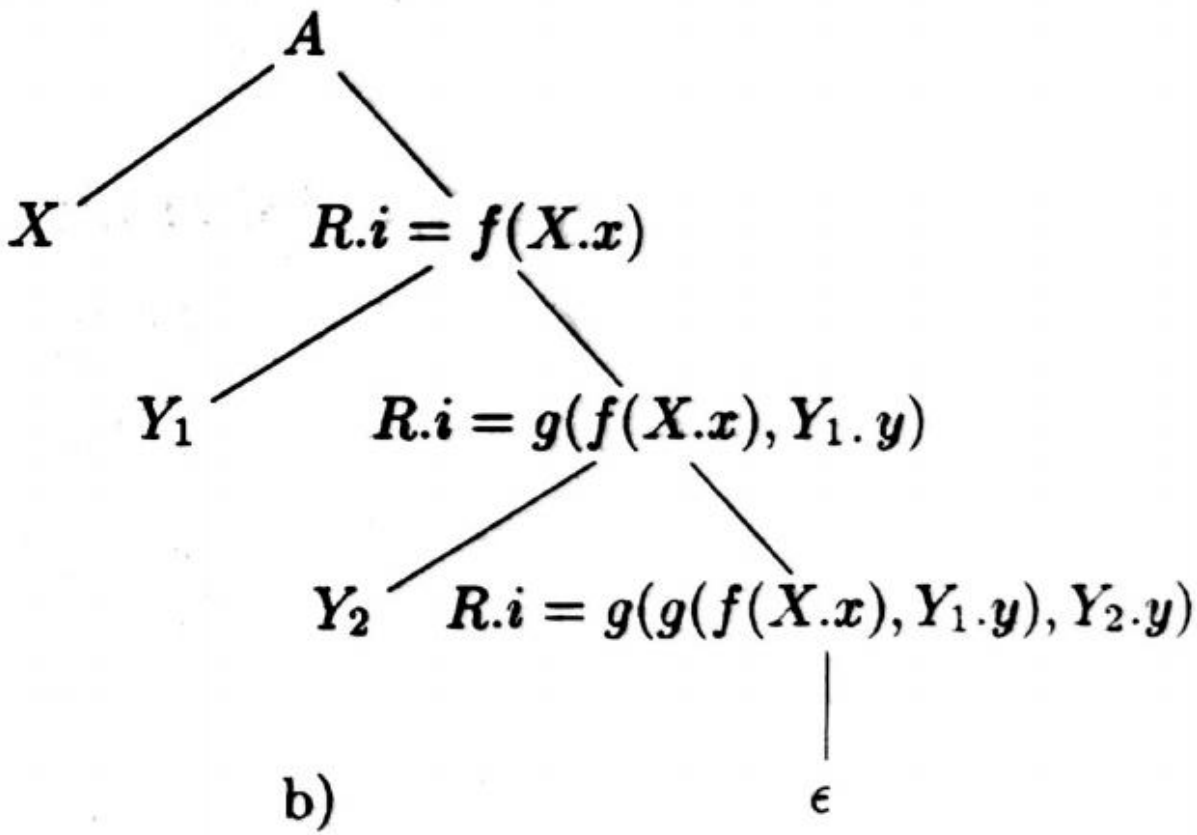
A → X { R.i = f(X.x); } R { A.a = R.s; }

R → Y { R1.i = g(R.i, Y.y); } R1 { R.s = R1.s; }

R → ε { R.s = R.i; }



a)



b)

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例： R → **addop** T { R1 .i = mknode(**addop**.lexeme, R.i, T.nptr); }

R1 { R.s = R1 .s; }

R → ε { R.s = R.i; }

– 只解析：

```
void R() {
    if (lookahead == addop) {
        match(addop);
        T();
        R()
    } else {
        // do nothing
    }
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 解析并翻译：

```
SyntaxTreeNode R(SyntaxTreeNode i) {
    SyntaxTreeNode s;  // synthesized attributes
    SyntaxTreeNode t_nptr, r1_i, r1_s;  // for children
    char addopLexeme;  // temporary

    if (lookahead == addop) {
        addopLexeme = lookahead.lexval;
        match(addop);
        t_nptr = T();
        r1_i = mknode(addopLexeme, i, t_nptr);
        r1_s = R(r1_i);
        s = r1_s;
    } else {
        s = i;
    }
    return s;
}
```

R → **addop** T { R1.i = mknode(**addop**.lexeme, R.i, T.nptr); }

R1 { R.s = R1.s; }

R → ε { R.s = R.i; }

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 解析并翻译：

✓ 每个继承属性对应于一个形式参数（函数**A的参数**是非终结符**A的继承属性**）

✓ 所有综合属性对应返回值（函数**A的返回值**是非终结符**A的综合属性集合**）

- 多条综合属性可以合并到一条记录返回中

✓ 在函数A的函数体中，进行语法分析并处理属性：

- 决定用哪一个产生式来展开A
- 需读入某个**终结符**时，在输入中检查是否出现，若是，**match()**，否则报错
- 在局部变量中**保存所有必要的属性值**，这些值将用于计算产生式体中非终结符的继承属性，或产生式头的非终结符的综合属性
- **调用**产生式体**中非终结符的函数**，并提供参数

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

$N \rightarrow .\ \{ S.f = 1 \}\ S\ \{ print(S.v) \}$

$S \rightarrow \{ B.f = S.f \}\ B\ \{ S_1.f = S.f + 1 \}\ S_1\ \{ S.v = S_1.v + B.v \}$

$S \rightarrow \varepsilon\ \{ S.v = 0 \}$

$B \rightarrow 0\ \{ B.v = 0 \}$

$B \rightarrow 1\ \{ B.v = 2^{-B.f} \}$

例：$.1011=2^{-1}+0+2^{-3}+2^{-4}$

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓ 根据产生式：**N→ . { S.f = 1} S { print(S.v) }**

对非终结符N，构造如下函数：

```
void N()
{
    MatchToken('.');          //匹配'.'
    Sf =1;                    //变量 Sf 对应属性S.f
    Sv = S(Sf);               //变量 Sv 对应属性S.v
    print(Sv);
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓ 根据产生式 $S\rightarrow \{ B.f = S.f \}\ B\ \{ S_1.f = S.f + 1 \}\ S1\ \{S.v = S_1.v + B.v \}$

$S\rightarrow \varepsilon\ \{ S.v = 0 \}$

对非终结符S，构造如下函数：

```
float S(int f){
    if (lookahead=='0' or lookahead=='1'){
        Bf = f;   Bv = B(Bf); S1f = f+1;
        S1v = S(S1f); Sv = S1v + Bv;
    }
    else if (lookahead=='$')  Sv = 0;
    else { printf("syntax error \n"); exit(0); }
    return Sv;
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓ 根据产生式 $B \rightarrow 0 \quad \{ B.v = 0 \}$

$B \rightarrow 1 \quad \{ B.v = 2^{-B.f} \}$

对非终结符B，构造如下函数：

```
float B(int f){
    if (lookahead=='0'){ MatchToken('0'); Bv = 0 }
    else if(lookahead=='1'){
        MatchToken('1');    Bv = 2^(-f);
    }
    else{ printf("syntax error \n"); exit(0); }
    return Bv;
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：while循环语句文法 $S \rightarrow \textbf{while}(C)S_1$

SDD

Productions	Semantic Actions

$S \rightarrow \textbf{while}(C)S_1$	L1 = new(); //while语句开始位置
	L2 = new(); //S_1语句开始位置
	S_1.next = L1;
	C.false = S.next;
	C.true = L2;
	S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;

SDT

$S \rightarrow \textbf{while}(\ \{$ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; $\}$
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; }
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

存储并返回结果：

```
string S(label next){
    string Scode, Ccode; //存放代码片段的局部变量
    label L1,L2; //局部标号
    if(当前输入 == 词法单元while){
        读取输入;
        检查'('是下一个输入符号，并读取输入;
        L1 = new();
        L2 = new();
        Ccode = C(next,L2);
        检查')'是下一个输入符号，并读取输入;
        Scode = S(L1);
        return("label"||L1||Ccode||"label"||L2||Scode);
    }
    else … //其他语句类型
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; print(“label”,L1); }
C) { S_1.next = L1; print(“label”,L2); }
S_1

边扫描边生成代码：

```
void S(label next){
    label L1,L2; //局部标号
    if(当前输入 == 词法单元while){
        读取输入;
        检查‘(’是下一个输入符号，并读取输入;
        L1 = new();
        L2 = new();
        print(“label”,L1);
        C(next,L2);
        检查‘)’是下一个输入符号，并读取输入;
        print(“label”,L2);
        S(L1);
    }
    else … //其他语句类型
}
```

随堂练习（3）

- 给定LL(1)文法G[E]及其翻译模式:

  E → T { R.in = T.val; } R { E.val = R.val; }

  R → + T { R1.in = R.in + T.val; } R1 { R.val = R1.val; }

  R → – T { R1.in = R.in – T.val; } R1 { R.val = R1.val; }

  R → ε { R.val = R.in; }

  T → num { T.val = num.val; }

  试对该翻译模式构造相应的递归下降预测翻译程序

随堂练习（3）

- 参考答案

E → T { R.in = T.val; } R { E.val = R.val; }
R → + T { R1.in = R.in + T.val; } R1 { R.val = R1.val; }
R → – T { R1.in = R.in – T.val; } R1 { R.val = R1.val; }
R → ε { R.val = R.in; }
T → num { T.val = num.val; }

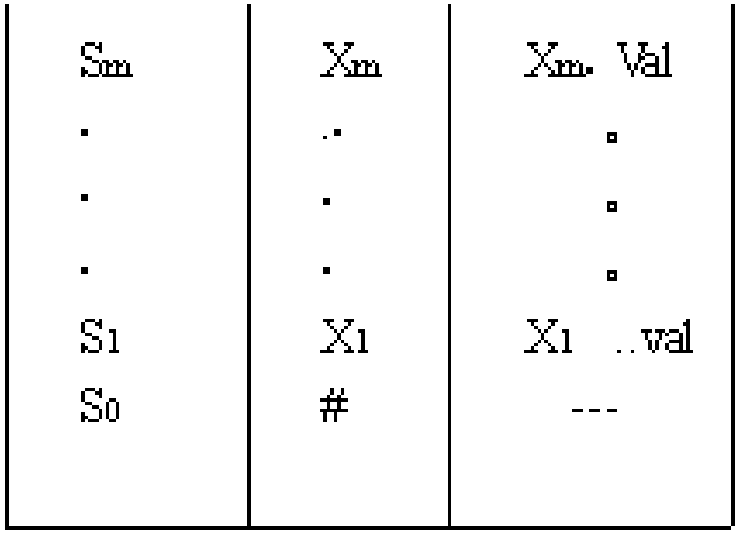
```
int E() {
    int t_val = T();
    int r_in = t_val;
    int r_val = R(r_in);
    return r_val;
}
```

```
int T() {
    if (isdigit(current_token.val) || current_token.val > 0) {
        int num_val = current_token.val;
        match(num_val);
        return num_val;
    } else {
        fprintf(stderr, "Syntax error: expected number\n");
        exit(1);
    }
}
```

```
int R(int in) {
    if (current_token.val == '+') {
        match('+');
        int t_val = T();
        int r1_in = in + t_val;
        int r1_val = R(r1_in);
        return r1_val;
    } else if (current_token.val == '-') {
        match('-');
        int t_val = T();
        int r1_in = in - t_val;
        int r1_val = R(r1_in);
        return r1_val;
    } else {
        // R → ε
        return in;
    }
}
```

6. 在LR分析中实现L-属性定义

- S-属性定义
  - 很容易进行LR分析——自底向上
  - 引入语义栈来存储属性值



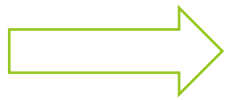
- L-属性定义进行LR分析
  - 存在的挑战：
    - ✓ 若仅含综合属性：并不是所有语义动作都位于生产式体的最右边
    - ✓ 若含有继承属性：继承属性不存储在语义栈中
  - 应对策略：
    - ✓ 若仅含综合属性：**使用markers标记将所有嵌入的动作移到产生式体的最右边**
    - ✓ 若含有继承属性：**在语义栈中跟踪继承属性**

6. 在LR分析中实现L-属性定义

(1) 若仅含综合属性：使用markers标记将所有嵌入的动作移到产生式体的最右边

- marker可看成一个占位符
- marker替换掉L-属性SDT中的某个语义规则
- 对marker引入一条ε产生式，在产生式最右边附加原语义规则

```
E → T R
R → + T { print('+'); } R
    | – T { print('–'); } R
    | ε
T → num { print(num.val); }
```



```
E → T R
R → + T M R
    | – T N R
    | ε
T → num { print(num.val); }
M → ε { print('+'); }
N → ε { print('–'); }
```

6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 最简单的情况：继承属性是通过复写规则(copy)从某个综合属性传播而来的

✓ 如：A→XYZ，将XYZ归约成A时，X的属性会先于Y的属性存在于语义栈中，若Y.i=X.s，则Y.i可直接从语义栈的X.s得来

D → T { L.inh = T.type; } L

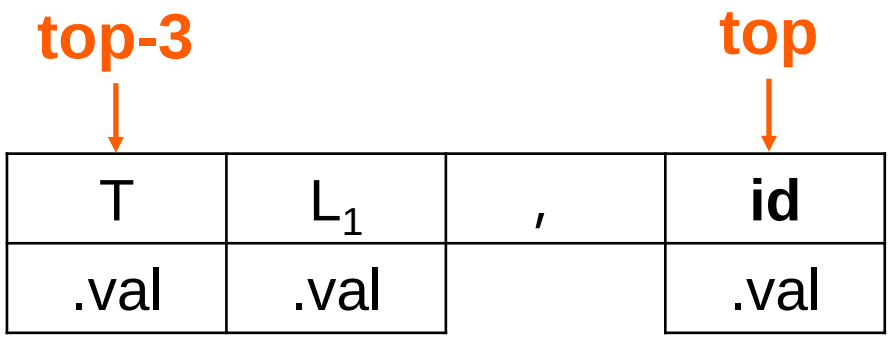
T → **int** { T.type = INTEGER; }

T → **real** { T.type = REAL; }

L → { L_1.inh = L.inh; }

L_1 , **id** { addType(**id**.entry, L.inh); }

L → **id** { addType(**id**.entry, L.inh); }



Productions	Code
D → T L	
T → **int**	stack[ntop].val = INTEGER;
T → **real**	stack[ntop].val = REAL;
L → L_1 , **id**	addType(stack[top].val, stack[top – 3].val);
L → **id**	addType(stack[top].val, stack[top – 1].val);

6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 更复杂的情况1：若继承属性是通过普通函数而不是通过复写规则来定义的

✓ 如：S→aA {C.i=**f**(A.s)} C，在计算C.i时，A.s在语义栈上，但f(A.s)并未存在于语义栈，则：

- 引入新的marker **M**，将上述产生式规则改写为：

  S→aA **{M.i=A.s}** **M** **{C.i=M.s}** C

  M→ε {M.s=f(M.i)}

- 先执行**M→ε**的规则**{M.s=f(M.i)}**，算好f函数值M.s并保存在语义栈中后，才执行第一条产生式M后的规则**{C.i=M.s}**

6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 更复杂的情况2：若继承属性的访问存在不确定性

✓ 如：S→aA {C.i=A.s} C | bAB {C.i=A.s} C　　C→c {C.s=g(C.i)}

在使用C→c进行归约时，不确定C.i应该使用语义栈top-1位置的，还是语义栈top-2位置的，则：

- 引入新的marker **M**，将上述产生式规则改写为：

  S→aA {C.i=A.s} C | bAB **{M.i=A.s} M {C.i=M.s}** C

  M→ε {M.s=M.i}

  C→c {C.s=g(C.i)}

- 此时在使用C→c进行归约时，C.i的值就确定地可以通过访问语义栈top-1而得

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

SDT

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; }
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

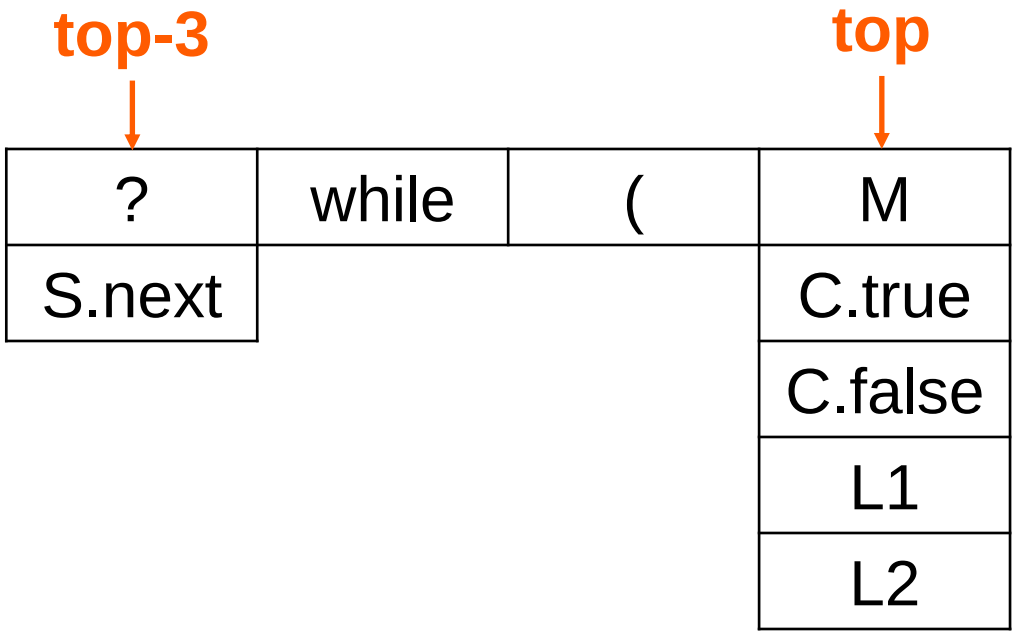
语义代码1： L1 = new(); L2 = new(); C.false = S.next; C.true = L2;
语义代码2： S_1.next = L1;
语义代码3： S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码1：
L1 = new(); L2 = new();
C.false = S.next; C.true = L2;



语义代码1：
L1 = new();
L2 = new();
C.true = L2;
C.false = stack[top-3].next;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码2： S_1.next = L1;

top-3 **top**

?	while	(	M	C	)	N
S.next			C.true	C.code		S_1.next
			C.false			
			L1			
			L2			

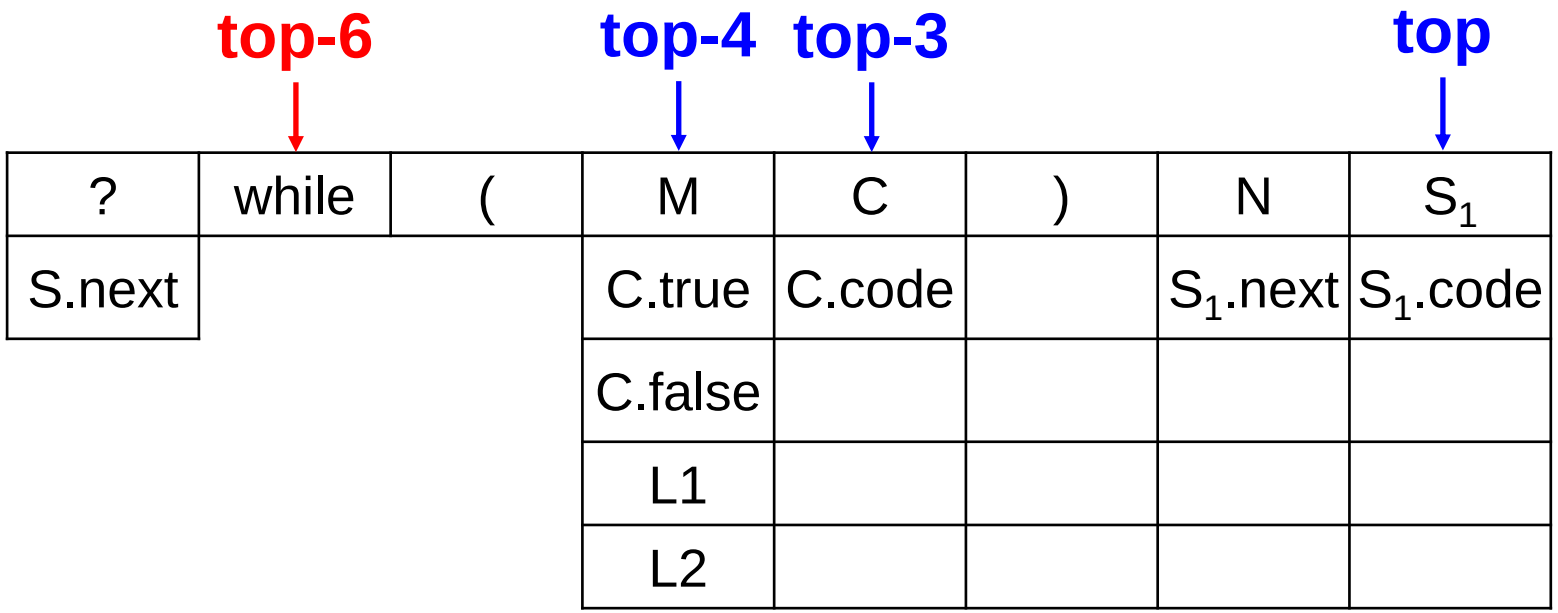
语义代码2：
S_1.next = stack[top-3].L1;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码3：
S.code = **label** || L1 || C.code ||
label || L2 || S_1.code;



?	while	(	M	C	)	N	S_1
S.next			C.true	C.code		S_1.next	S_1.code
			C.false				
			L1				
			L2				

语义代码3：
tempCode = **label** ||stack[top-4].L1 || stack[top-3].code || **label** || stack[top-4].L2 || stack[top].code;
top=top-6;
stack[top].code=tempCode;

第五章作业

- 给定LL(1)文法G[S]及其翻译模式：

  S → Ab {B.in_num=A.num} B {if B.num=0 then print(“Accepted!”) else print(“Refused!”)}

  A → a A_1 {A.num=A_1.num+1}

  A → ε {A.num=0}

  B → a {B_1.in_num=B.in_num} B_1 {B.num=B_1.num-1}

  B → b {B_1.in_num=B.in_num} B_1 {B.num=B_1.num}

  B → ε {B.num=B.in_num}

  试对该翻译模式构造相应的递归下降预测翻译程序

第五章作业

- 提交要求：
  - 文件命名：学号-姓名-第五章作业；
  - 文件格式：.pdf文件；
  - 手写版、电子版均可；若为手写版，则拍照后转成pdf提交，但**须注意将照片旋转为正常角度，且去除照片中的多余信息**；电子版如word等转成pdf提交；
  - 提交到超算习堂（第五章作业）处；
  - 提交ddl：**4月29日晚上12:00**；
  - **重要提示：不得抄袭！**