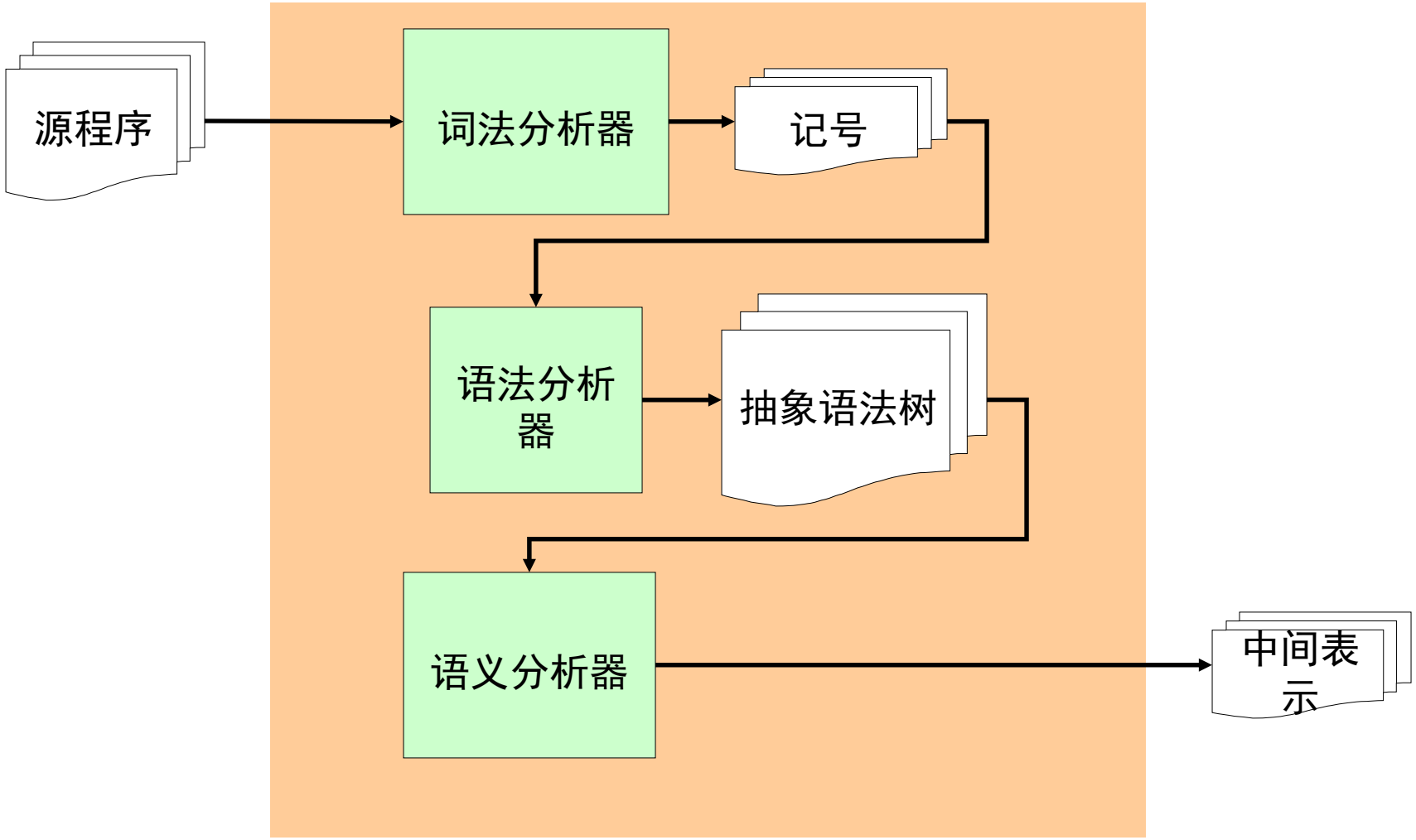
语法制导翻译技术（1）

语法制导定义

翻译方案

编译器前端



语法制导翻译概述

- 语义分析涉及到语言的语义
- 形式语义学可以分为四类
  - 操作语义学：通过语言的实现方式（即语言成分所对应的计算机的操作）定义语言成分的语义，着重模拟数据加工过程中计算机系统的操作。
  - 指称语义学：通过执行语言成分所得到的最终效果来定义该语言成分的语义，主要描述数据加工的结果，而不是加工过程的细节。
  - 代数语义学：用代数公理刻画语言成分的语义，主要研究抽象数据类型的代数规范，可看作是指称语义学的一个分支。
  - 公理语义学：采用公理化方法描述程序对数据的加工，用公理系统定义程序设计语言的语义，另外，公理语义学还研究和寻求适用于描述程序语义、便于语义推导的逻辑语言。
- 语法制导翻译技术
  - 多数编译程序普遍采用的一种技术
  - 比较接近形式化

语法制导翻译的整体思路

- 首先，根据翻译目标来确定每个产生式的语义；
- 其次，根据产生式的含义，分析每个符号的语义；
- 再次，把这些语义以属性的形式附加到相应的文法符号上（即把语义和语言结构联系起来）；
- 然后，根据产生式的语义给出符号属性的求值规则（即语义规则），从而形成语法制导定义。

- 翻译：
  根据语法分析过程中所使用的产生式，执行与之相应的语义规则，完成符号属性值的计算，从而完成翻译。

语法制导翻译示例

$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$
$E \rightarrow T$	$E.val=T.val$
$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$
$T \rightarrow F$	$T.val=F.val$
$F \rightarrow (E)$	$F.val=E.val$
$F \rightarrow digit$	$F.val=digit.val$

例如：考虑算术表达式文法

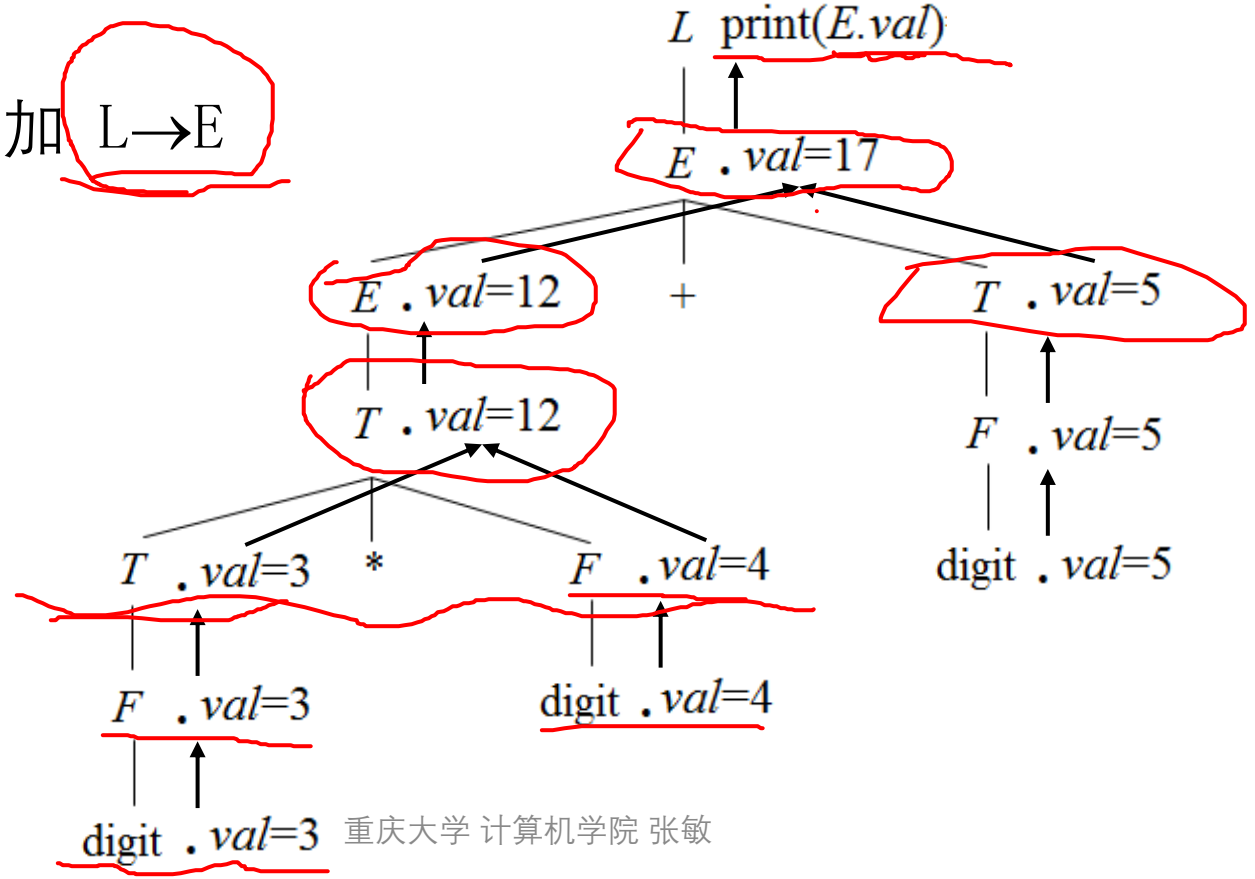
- 翻译目标：计算表达式的值
- 根据翻译目标确定每个产生式的语义；
  - $E \rightarrow E_1+T$：表达式的值由两个子表达式的值相加得到
  - $F \rightarrow digit$：表达式的值即数字的值
- 根据产生式的语义，分析每个符号的语义；
  - E、T、F、digit、+、*、（、）
- 把这些语义以属性的形式附加到相应的文法符号上；
  - E.val、T.val、F.val、digit.val
- 根据产生式的语义，给出符号属性的求值规则(即语义规则)，从而形成语法制导定义。
  - $E \rightarrow E_1+T$对应的求值规则：$E.val=E_1.val+T.val$
  - 语法制导定义：产生式 语义规则

语法制导翻译示例（续）

例如：考虑算术表达式文法

$E \rightarrow E_1+T$	E.val=E_1.val+T.val
$E \rightarrow T$	E.val=T.val
$T \rightarrow T_1*F$	T.val=T_1.val*F.val
$T \rightarrow F$	T.val=F.val
$F \rightarrow (E)$	F.val=E.val
$F \rightarrow digit$	F.val=digit.val

- 翻译目标：计算并打印表达式的值
- 根据语法分析过程中使用的产生式，执行相应的语义规则，完成相应的属性求值，从而完成翻译。
- 例如：3*4+5
  - 拓广文法：增加 L→E
  - 分析树：



翻译目标决定语义规则

- 翻译目标决定产生式的含义、决定文法符号应该具有的属性，也决定了产生式的语义规则。
- 例如：考虑算术表达式文法
  - 翻译目标：检查表达式的类型
  - $E \rightarrow E_1+T$ 的语义：表达式的类型由两个子表达式的类型综合得到
  - 分析每个符号的语义，并以属性的形式记录：E.type、 E_1.type、T.type
  - 求值规则：

```
if (E1.type==integer)&&(T.type==integer)
    E.type=integer;
else …
```


翻译结果依赖于语义规则

- 翻译目标
  - 生成代码
    - 可以为源程序产生中间代码
    - 可以直接生成目标机指令
  - 对输入符号串进行解释执行
  - 向符号表中存放信息
  - 给出错误信息
- 翻译的结果依赖于语义规则
  - 使用语义规则进行计算所得到的结果就是对输入符号串进行翻译的结果。
  - 如：$E \rightarrow E+T$ 的翻译结果可以是：计算表达式的值、检查表达式的类型是否合法、为表达式创建语法树、生成代码等等。

语义规则的执行时机

- 可以用一个或多个子程序（称为语义动作）所要完成的功能描述产生式的语义。
- 在语法分析过程中使用某个产生式时，在适当的时机执行相应的语义动作，完成所需要的翻译。
- 把语义动作插入到产生式中适当的位置，从而形成翻译方案。

- 语法制导定义是对翻译的高层次的说明，它隐蔽了一些实现细节，无须指明翻译时语义规则的计算次序。
- 翻译方案指明了语义规则的计算次序，规定了语义动作的执行时机。

语法制导翻译的一般步骤

输入符号串

——→ 分析树

——→ 依赖图

——→ 语义规则的计算顺序

——→ 计算结果

5.1 语法制导定义及翻译方案

5.1.1 语法制导定义

5.1.2 依赖图

5.1.3 计算次序

5.1.4 S属性定义和L属性定义

5.1.5 翻译方案

5.1.1 语法制导定义

- 每个文法符号都可以有一个属性集，其中可以包括两类属性：综合属性和继承属性。
  - 左部符号的综合属性是从该产生式右部文法符号的属性值计算出来的；在分析树中，一个内部结点的综合属性是从其子结点的属性值计算出来的。
  - 出现在产生式右部的某文法符号的继承属性是从其所在产生式的左部非终结符号和/或右部文法符号的属性值计算出来的；
    在分析树中，一个结点的继承属性是从其兄弟结点和/或父结点的属性值计算出来的。
  - 分析树中某个结点的属性值是由与在这个结点上所用产生式相应的语义规则决定的。
- 和产生式相联系的语义规则建立了属性之间的关系，这些关系可用有向图（即：依赖图）来表示。

语义规则

- 一般情况：
  - 语义规则函数可写成表达式的形式。
  - 比如： $E.val=E_1.val+T.val$
- 某些情况下：
  - 一个语义规则的唯一目的就是产生某个副作用，如打印一个值、向符号表中插入一条记录等；
  - 这样的语义规则通常写成过程调用或程序段。
  - 看成是相应产生式左部非终结符号的虚拟综合属性。
  - 比如： print(E.val)

语法制导定义

在一个语法制导定义中，对应于每一个文法产生式$A\rightarrow\alpha$，都有与之相联系的一组语义规则，其形式为：$b=f(c_1,c_2,\ldots,c_k)$

这里，f是一个函数，而且

(1) 如果b是A的一个综合属性，则 c_1、c_2、…、c_k是产生式右部文法符号的属性或者A的继承属性；

(2) 如果b是产生式右部某个文法符号的一个继承属性，则c_1、c_2、…、c_k是A或产生式右部任何文法符号的属性。

- 属性b依赖于属性c_1、c_2、…、c_k。
- 语义规则函数都不具有副作用的语法制导定义称为属性文法

简单算术表达式求值的语法制导定义

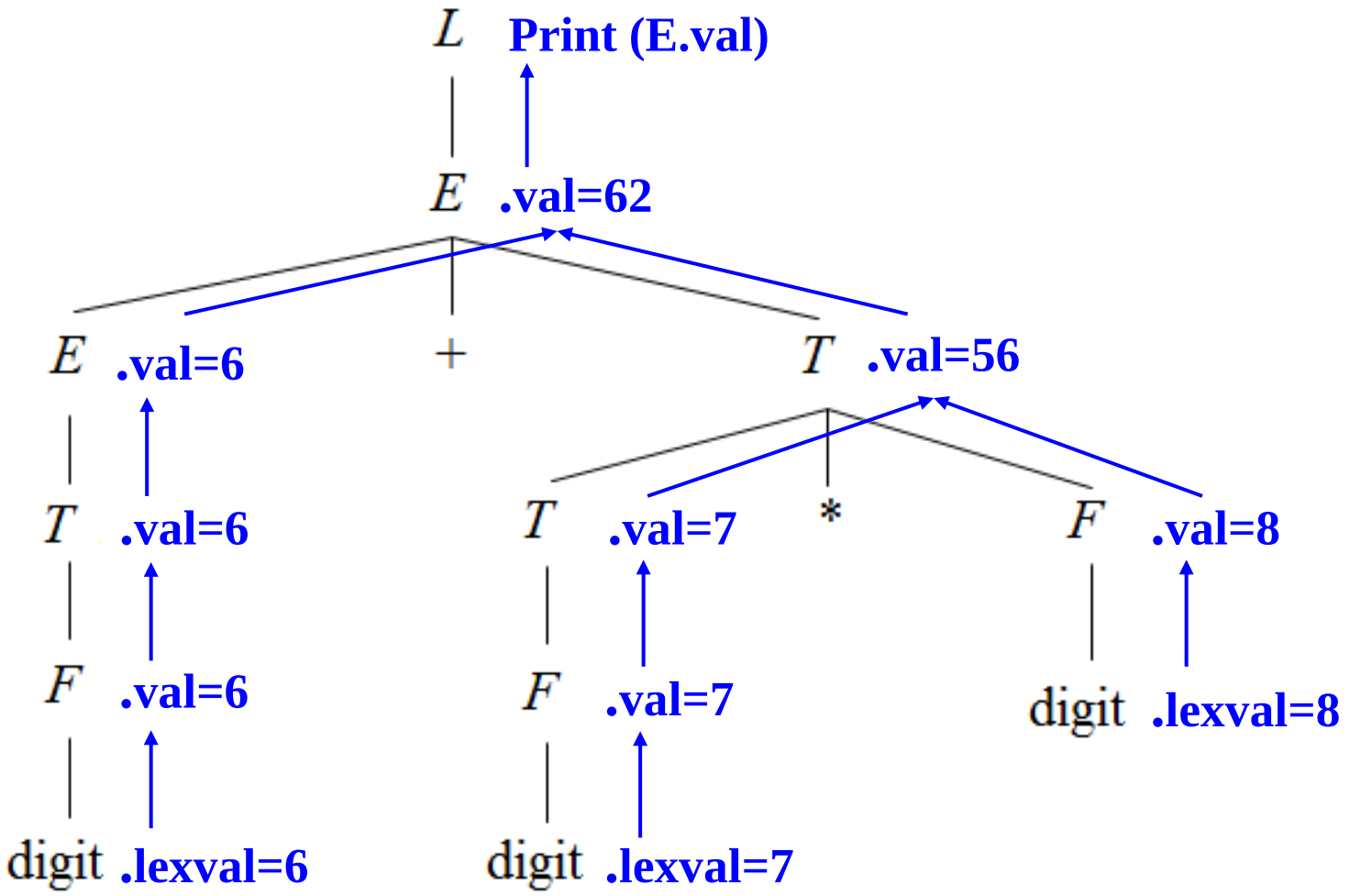
产生式	语义规则
$L \rightarrow E$	print(E.val)
$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$
$E \rightarrow T$	E.val=T.val
$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$
$T \rightarrow F$	T.val=F.val
$F \rightarrow (E)$	F.val=E.val
$F \rightarrow digit$	F.val=digit.lexval

- 综合属性val与每一个非终结符号E、T、F相联系
- 表示相应非终结符号所代表的子表达式的整数值
- L→E的语义规则是一个过程，打印出由E产生的算术表达式的值，可以认为是非终结符号L的一个虚拟综合属性。

综合属性

- 分析树中，如果一个结点的某一属性由其子结点的属性确定，则这种属性为该结点的综合属性。
- 如果一个语法制导定义仅仅使用综合属性，则称这种语法制导定义为S-属性定义。
- 对于S-属性定义，通常采用自底向上的方法对其分析树加注释，即从树叶到树根，按照语义规则计算每个结点的属性值。
- 简单台式计算机的语法制导定义是S-属性定义

6+7*8的分析树加注释的过程



- 属性值的计算可以在语法分析过程中进行。

继承属性

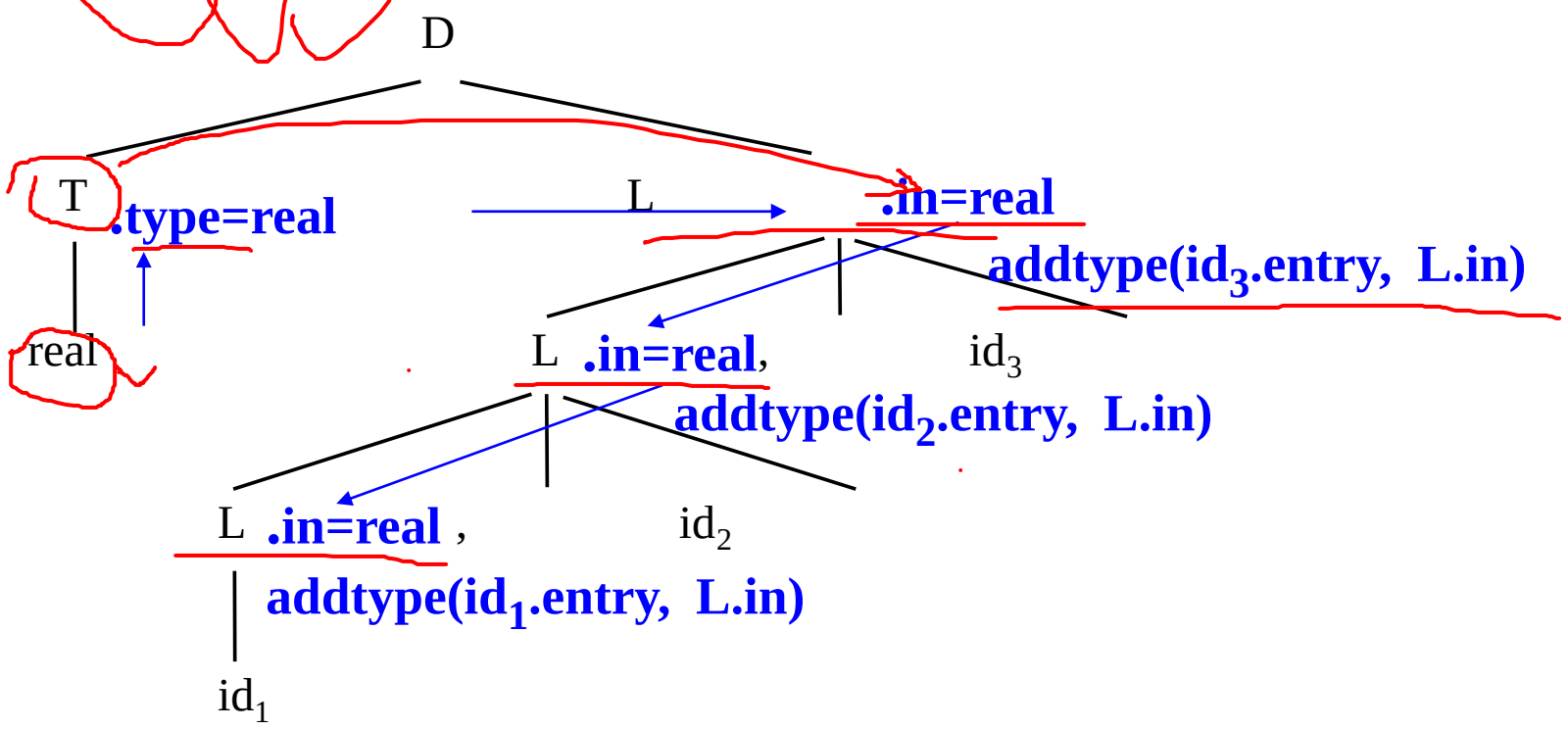
- 分析树中，一个结点的继承属性值由该结点的父结点和/或它的兄弟结点的属性值决定。
- 可用继承属性表示程序设计语言结构中上下文之间的依赖关系
  - 可以跟踪一个标识符的类型
  - 可以跟踪一个标识符，了解它是出现在赋值号的右边还是左边，以确定是需要该标识符的值还是地址。

用继承属性L.in传递类型信息的语法制导定义

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

- D产生的声明语句包含了类型关键字int或real，后跟一个标识符表。
- T有综合属性type，其值由声明中的关键字确定。
- L的继承属性L.in，
  - 产生式D→TL　　L.in 表示从其兄弟结点T继承下来的类型信息。
  - 产生式$L \rightarrow L_1$, id　　L_1.in 表示从其父结点L继承下来的类型信息

语句real id_1,id_2,id_3的注释分析树



L产生式的语义规则使用继承属性L.in把类型信息在分析树中向下传递；并通过调用过程addtype，把类型信息填入标识符在符号表中相应的表项中。

5.1.2 依赖图

- 分析树中，结点的继承属性和综合属性之间的相互依赖关系可以由依赖图表示。
- 为每个包含过程调用的语义规则引入一个虚拟综合属性b，以便把语义规则统一为$b=f(c_1,c_2,\ldots,c_k)$的形式。
- 依赖图中：
  - 为每个属性设置一个结点
  - 如果属性b依赖于c，那么从属性c的结点有一条有向边连到属性b的结点。

算法5.1 构造依赖图

输入：一棵分析树

输出：一张依赖图

方法：

for (分析树中每一个结点n)
    for (结点n处的文法符号的每一个属性a)
        为a在依赖图中建立一个结点;

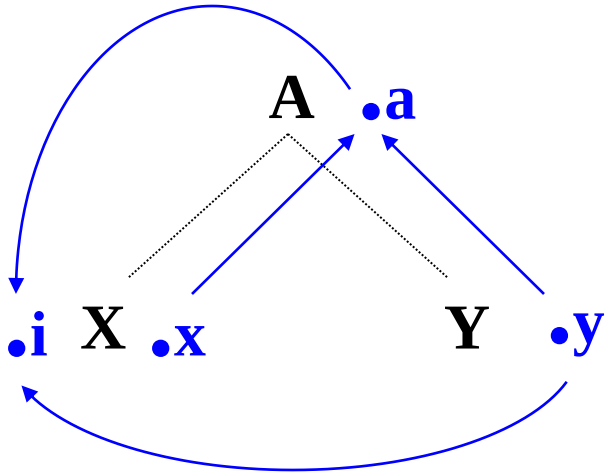
for (分析树中每一个结点n)
    for (结点n处所用产生式对应的每一个语义规则
        $b=f(c_1,c_2,\ldots,c_k)$)
        for (i=1; i<=k; i++)
            从c_i结点到b结点构造一条有向边;

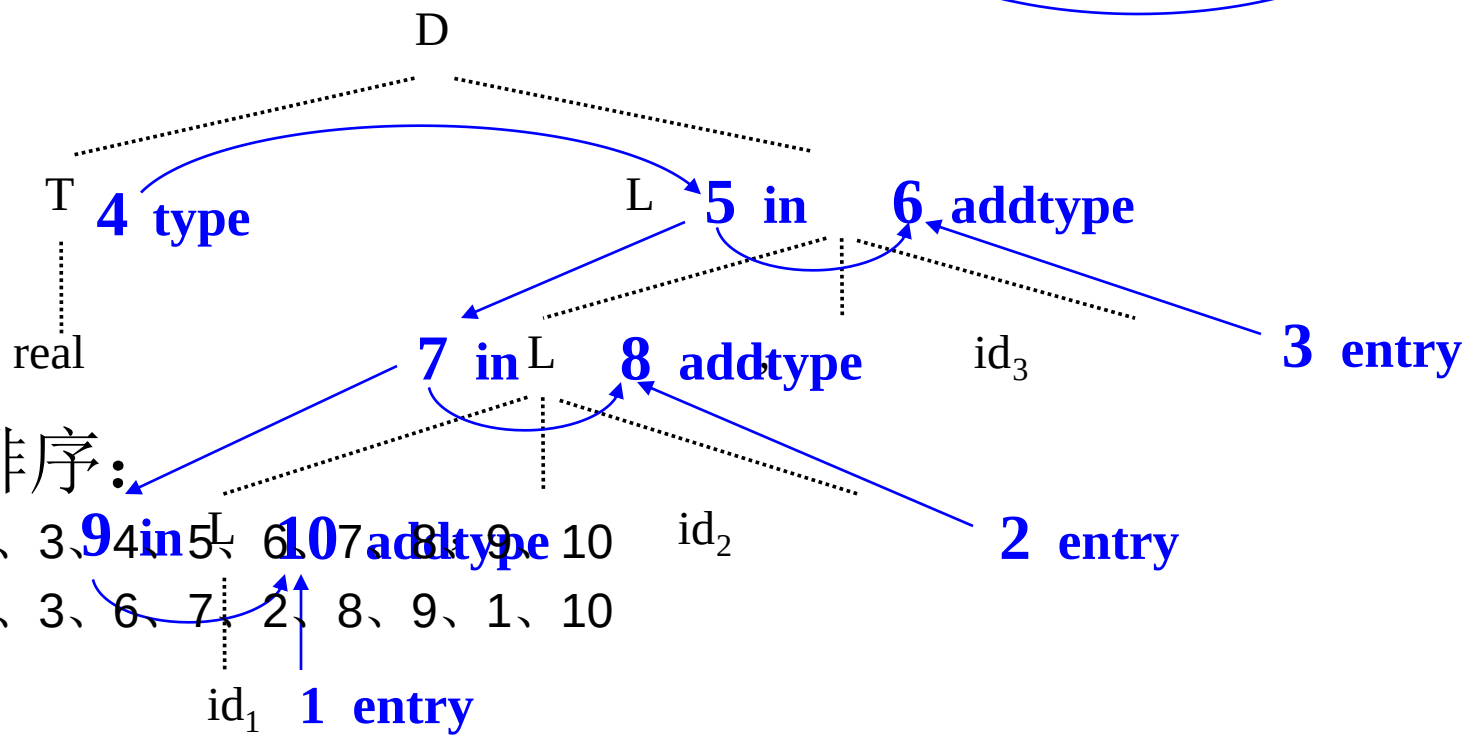
依赖图构造举例

产生式　　语义规则

$A \rightarrow XY$　　$A.a = f(X.x, Y.y)$

$X.i = g(A.a, Y.y)$





拓扑排序：

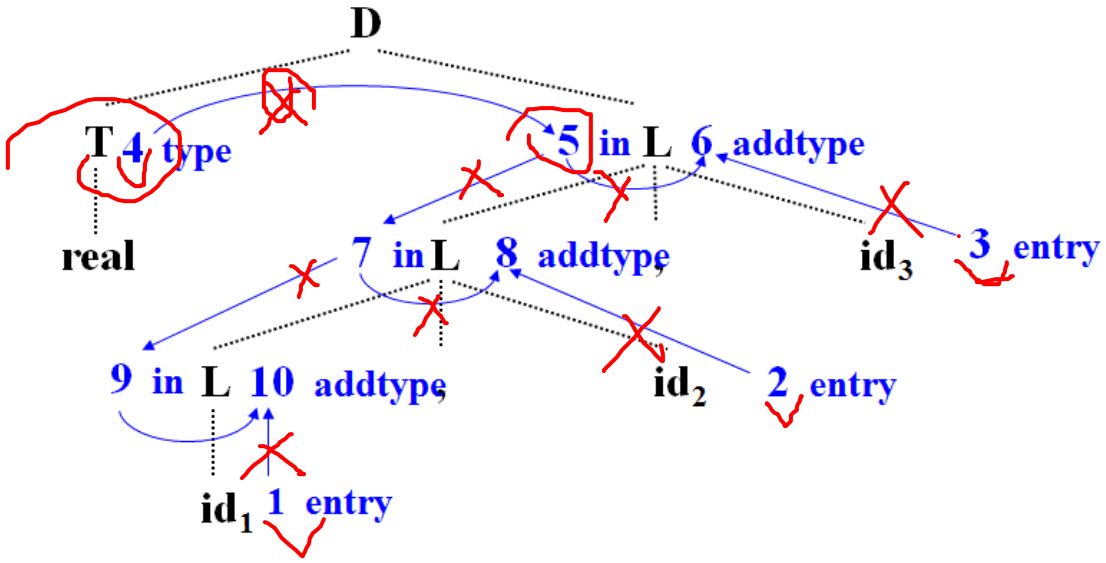
- 1、2、3、4、5、6、7、8、9、10
- 4、5、3、6、7、2、8、9、1、10

5.1.3 计算次序

- 有向非循环图的拓扑排序
  - 图中结点的一种排序 $m_1, m_2, \ldots, m_k$
  - 有向边只能从这个序列中前边的结点指向后面的结点
  - 如果 $m_i \rightarrow m_j$ 是从 m_i 指向 m_j 的一条边，那么在序列中m_i 必须出现在 m_j 之前。
- 依赖图的任何拓扑排序
  - 给出了分析树中结点的语义规则计算的有效顺序
  - 在拓扑排序中，一个结点上语义规则 $b=f(c_1, c_2, \ldots, c_k)$ 中的属性$c_1, c_2, \ldots, c_k$在计算b时都是可用的。

计算顺序

type=real;
in5=type;
addtype(id_3.entry, in5);
in7=in5;
addtype(id_2.entry, in7);
in9=in7;
addtype(id_1.entry, in9);



拓扑排序：

- 1、2、3、4、5、6、7、8、9、10
- 4、5、3、6、7、2、8、9、1、10

- an代表依赖图中与序号n的结点有关的属性a

5.1.4 S属性定义和L属性定义

- S属性定义：仅涉及综合属性的语法制导定义
- L属性定义：一个语法制导定义是L属性定义，如果
  - 与每个产生式 $A \rightarrow X_1X_2 \ldots X_n$ 相应的每条语义规则计算的属性都是A的综合属性，或是 X_j（$1 \leq j \leq n$）的继承属性，而该继承属性仅依赖于：
    - A的继承属性；
    - 产生式中X_j左边的符号X_1、X_2、…、X_{j-1} 的属性；

- 每一个S属性定义都是L属性定义

语法制导定义示例：

例：非L属性定义 ➡

产生式	语义规则
A→LM	L.i=l(A.i) M.i=m(L.s) A.s=f(M.s)
A→QR	R.i=r(A.i) Q.i=q(R.s) A.s=f(Q.s)

例：L属性定义 ➡

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

属性计算顺序——深度优先遍历分析树

```
void  deepfirst (n: node)
 {
   for  (n的每一个子结点m，从左到右) {
    计算m的继承属性;
    deepfirst(m);
   };
  计算n的综合属性;
 }.
```

- 以分析树的根结点作为实参
- L属性定义的属性都可以用深度优先的顺序计算。
  - 进入结点前，计算它的继承属性
  - 从结点返回时，计算它的综合属性

5.1.5 翻译方案

- 上下文无关文法的一种便于翻译的书写形式
- 属性与文法符号相对应
- 语义动作括在花括号中，并插入到产生式右部某个合适的位置上
- 给出了使用语义规则进行属性计算的顺序
- 分析过程中翻译的注释

翻译方案示例

一个简单的翻译方案：

$E \rightarrow TR$

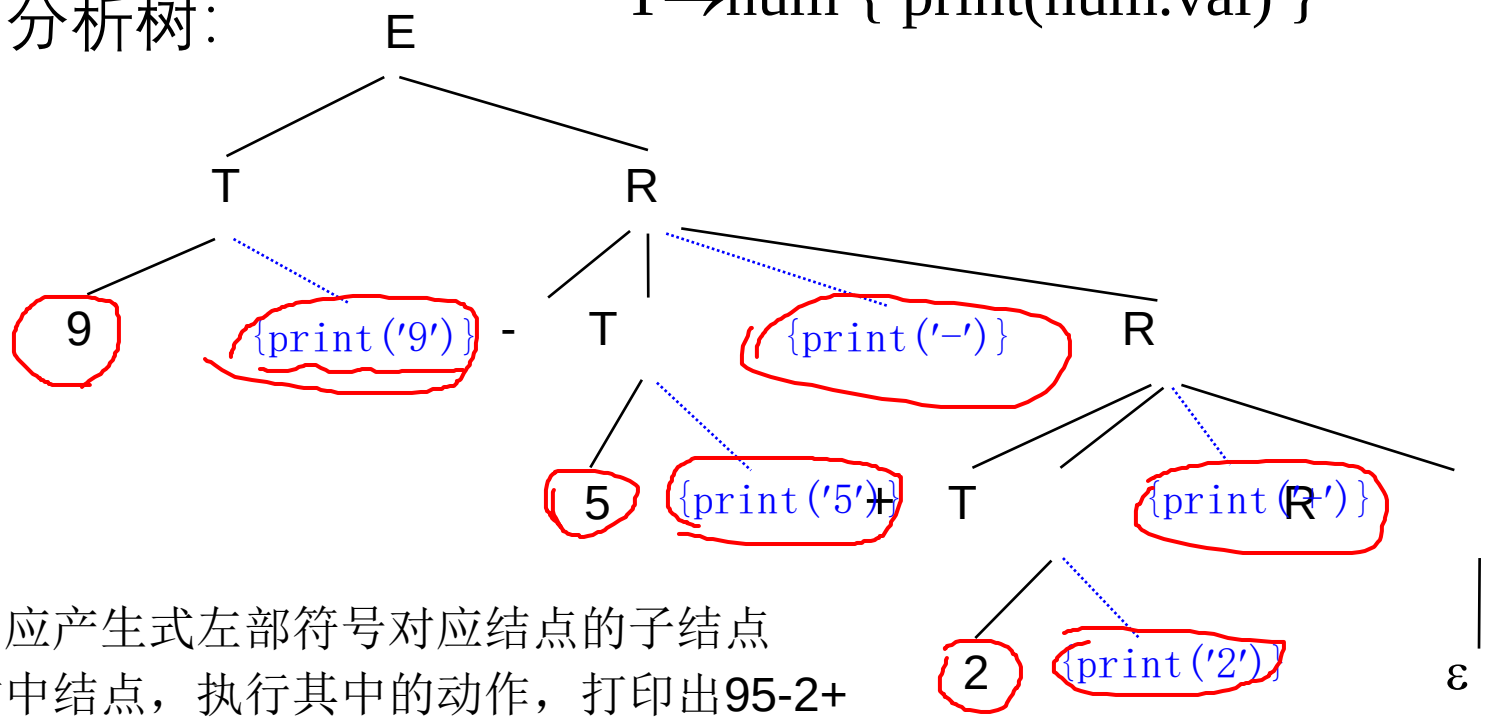
$R \rightarrow +T\ \{\ print('+')\ \}\ R_1$

$\quad | -T\ \{\ print('-')\ \}\ R_1$

$\quad | \varepsilon$

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

9-5+2的分析树：



语义动作作为相应产生式左部符号对应结点的子结点

深度优先遍历树中结点，执行其中的动作，打印出95-2+

翻译方案的设计

- 对于S属性定义：
  - 为每一个语义规则建立一个包含赋值的动作
  - 把这个动作放在相应的产生式右边末尾

例：产生式　　语义规则

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

如下安排产生式和语义动作：

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

为L属性定义设计翻译方案的原则

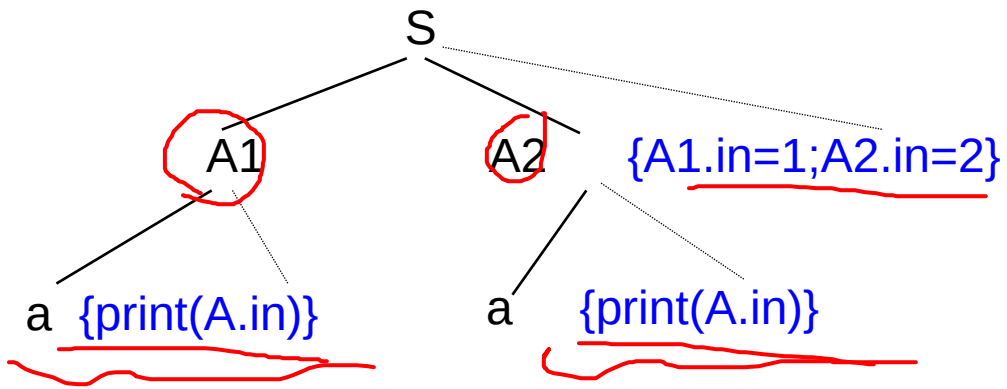
- 产生式右部文法符号的继承属性必须在这个符号以前的语义规则中计算出来
- 一个动作不能引用这个动作右边的文法符号的综合属性
- 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来之后才能计算，这种属性的计算动作放在产生式右端末尾

示例：

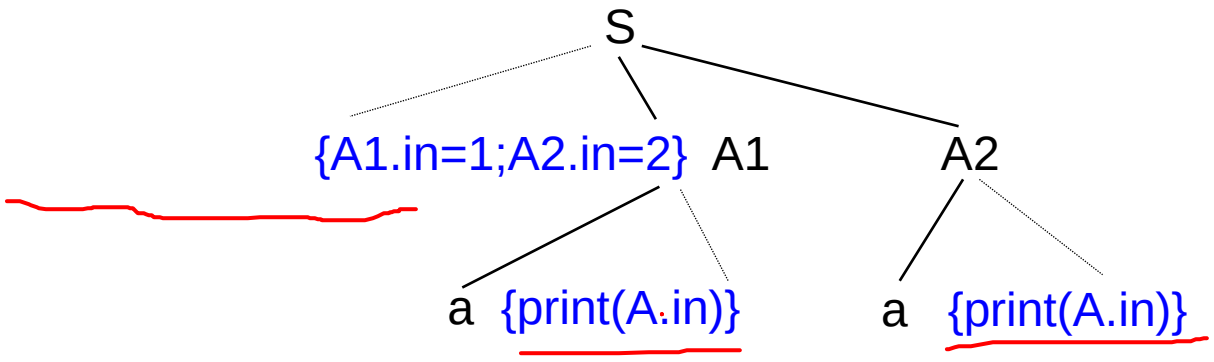
考虑如下翻译方案：

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

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



×



√

L属性定义翻译方案设计举例

- 语法制导定义

产生式	语义规则
D→TL	L.in=T.type
T→int	T.type=integer
T→real	T.type=real
L→L_1, id	L_1.in=L.in addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

- 翻译方案

D→T { L.in=T.type } L

T→int { T.type=integer }

T→real { T.type=real }

L→{ L_1.in=L.in } L_1,id { addtype(id.entry, L.in) }

L→id { addtype(id.entry, L.in) }

作业

- 1、下列文法产生对整型数和实型数进应用“+”算符形成的表达式。两个整型数相加，结果仍为整型数；否则为实型数。

  E→E+T|T

  T→ num,num|num

（1）给出一个确定每个子表达式类型的语法制导定义

（2）扩充（1）中的语法制导定义，使之既确定类型，又把表达式翻译为前缀形式。使用一元算符inttoreal把整型数转换为等价的实型数，使得前缀形式中的“+”作用于两个同类型的运算对象。