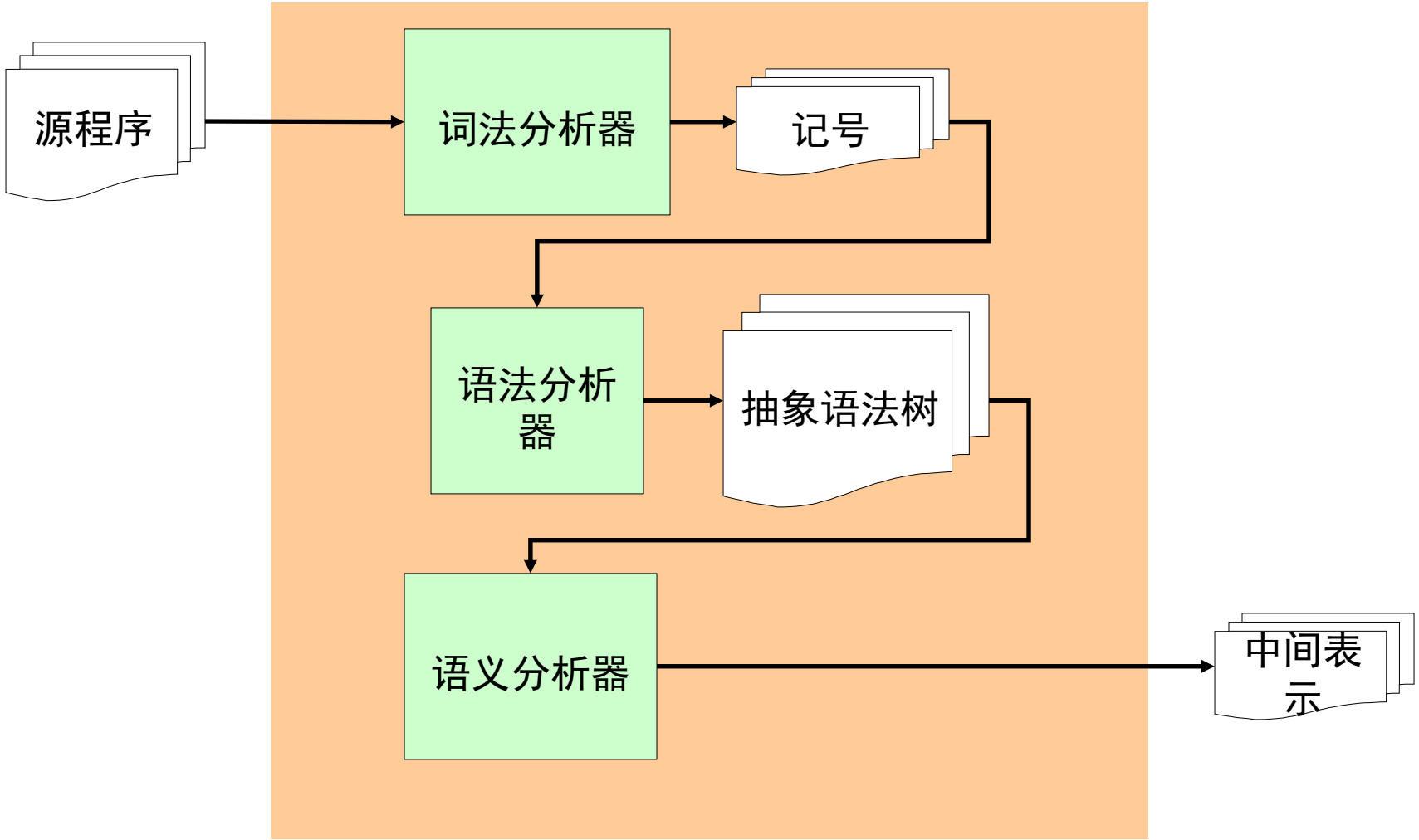
重庆大学
CHONGQING UNIVERSITY

第五讲 语法制导翻译

重庆大学 计算机学院 张敏

5.1 语法制导翻译概述



语法制导翻译概述

我们在解析表达式a+b的时候，得到一个语法树。但怎么知道它运算的时候是做加法呢?

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（Syntax directed definition，SDT）技术

- 多数编译程序普遍采用的一种技术
- 比较接近形式化

语法制导翻译概述

语法制导翻译的整体思路

- 根据翻译目标来确定每个产生式的语义；
- 根据产生式的含义，分析每个符号的语义；
- 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）；
- 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。

翻译时完成过程：

- 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。

语法制导翻译概述

语法制导翻译示例

例：考虑算术表达式文法

翻译目标：计算表达式的值

$E \rightarrow E_1+T$	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

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

语法制导翻译概述

语法制导翻译示例（续）

例如：考虑算术表达式文法

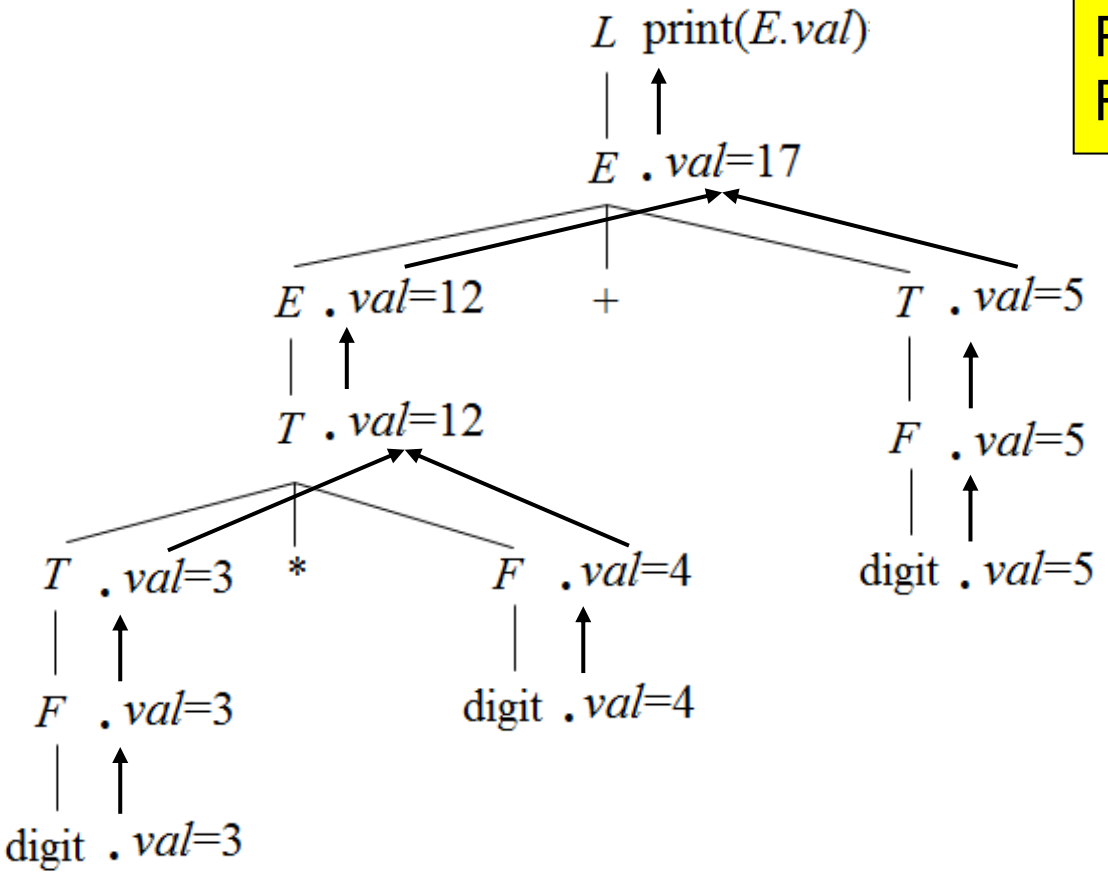
翻译目标：计算并打印表达式的值

例如：3*4+5

拓广文法：增加

分析树：

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



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→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。

5.1.1 语法制导定义

语义规则

一般情况：

语义规则函数可写成表达式的形式。

例： $E.val=E_1.val+T.val$

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。

例：print(E.val)

5.1.1 语法制导定义

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.

高德纳（Donald Knuth）在“The Genesis of Attribute Grammers”中提出，主要思想为在上下文无关文法的基础上做增强，使之能够计算属性值。

5.1.1 语法制导定义

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。

5.1.1 语法制导定义

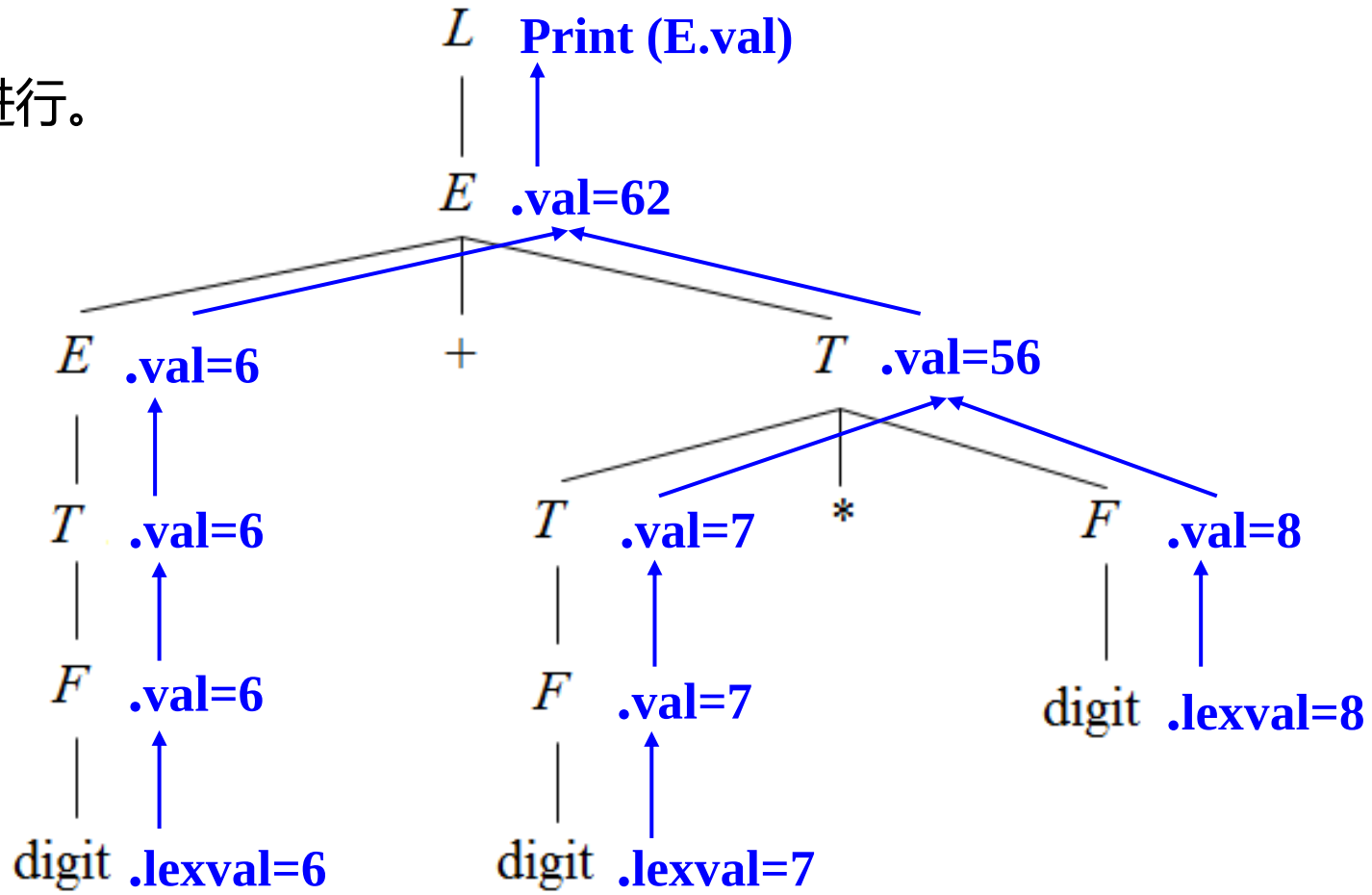
综合属性

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

5.1.1 语法制导定义

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。



5.1.1 语法制导定义

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。

5.1.1 语法制导定义

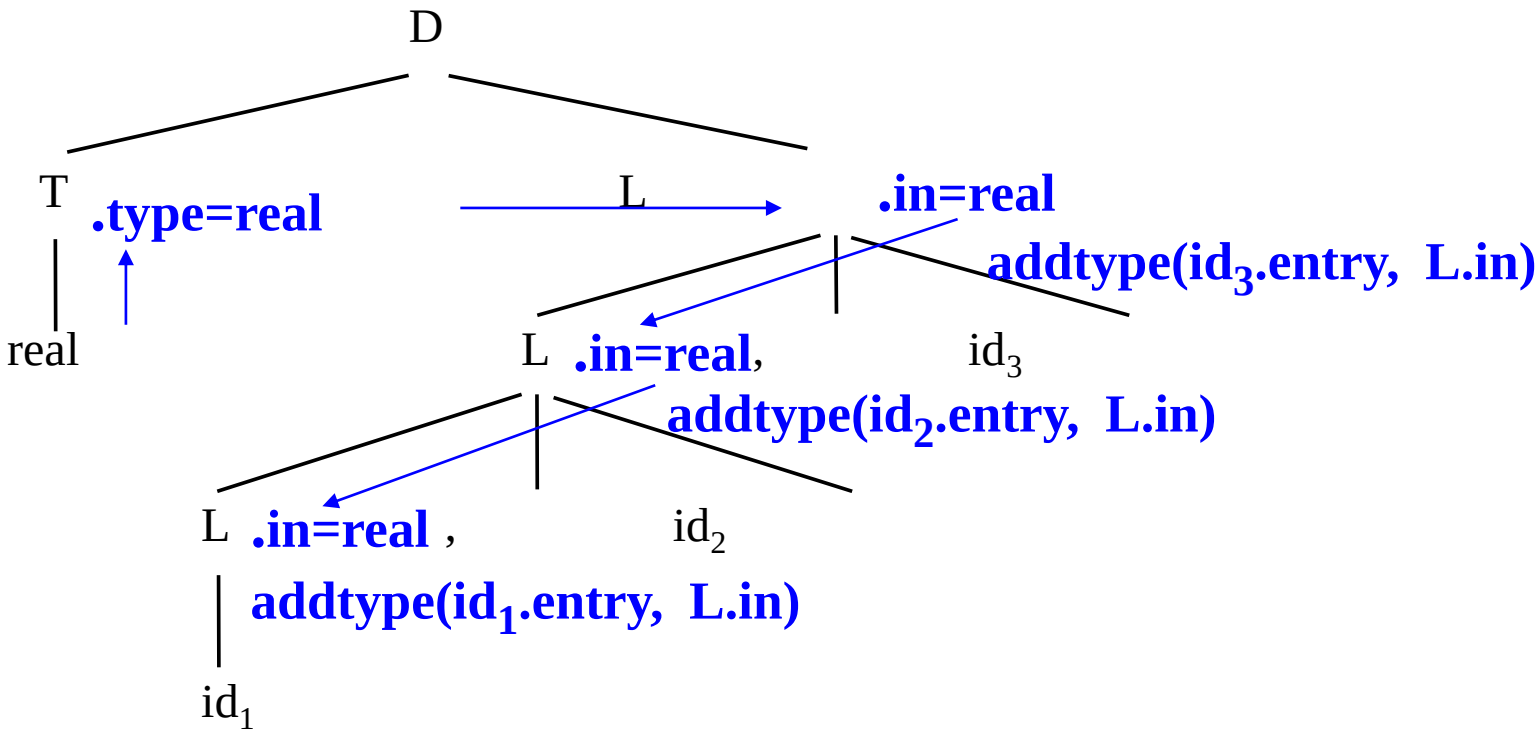
用继承属性L.in传递类型信息的语法制导定义

产生式	语义规则
D→TL	**L.in=T.type**
T→int	**T.type=integer**
T→real	**T.type=real**
L→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$L→L_1$, id　　L_1.in 表示从其父结点L继承下来的类型信息

5.1.1 语法制导定义

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。

5.1.2 依赖图

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：

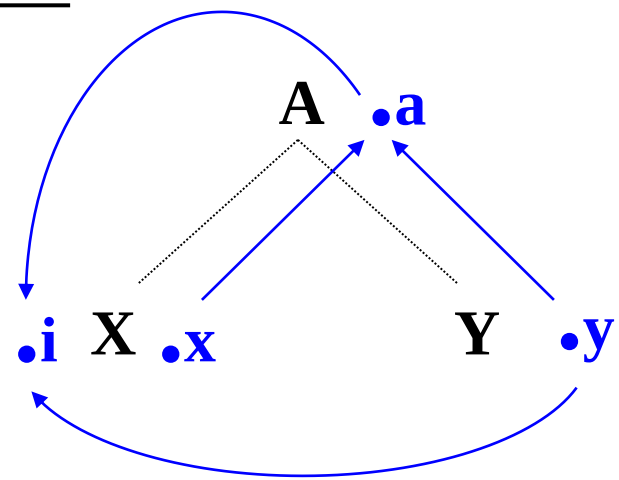
```
for (分析树中每一个结点n)
    for (结点n处的文法符号的每一个属性a)
      为a在依赖图中建立一个结点;
for (分析树中每一个结点n)
    for (结点n处所用产生式对应的每一个语义规则
        b=f(c1,c2,...,ck))
      for (i=1; i<=k; i++)
        从ci结点到b结点构造一条有向边;
```

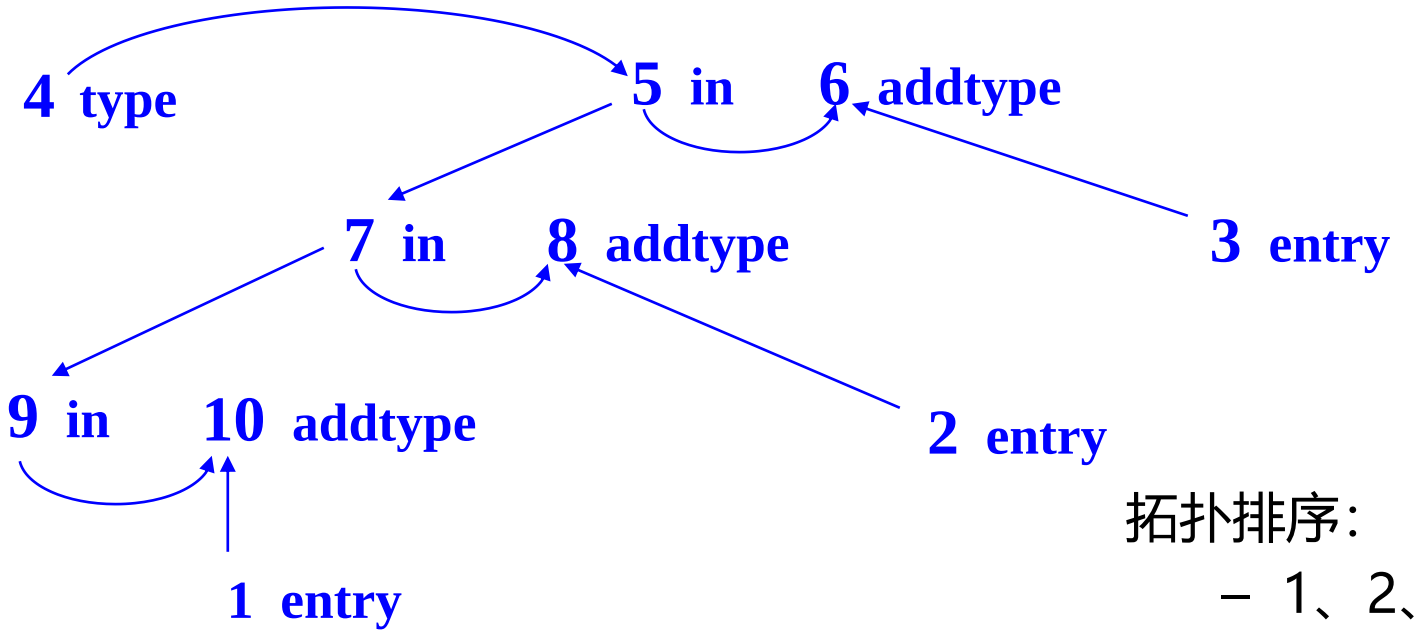
5.1.2 依赖图

依赖图构造举例

产生式　　语义规则

A→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

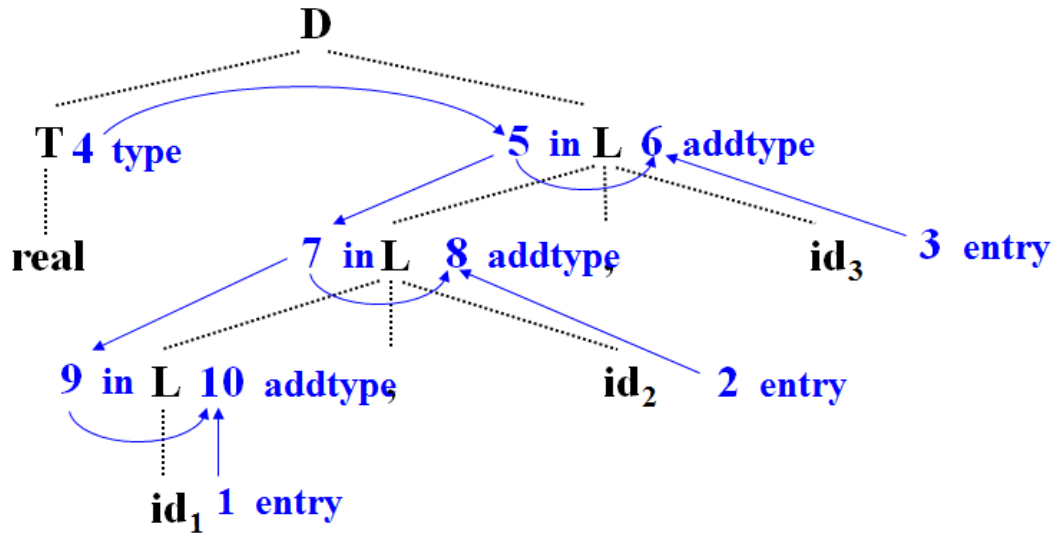
在拓扑排序中，一个结点上语义规则 $b=f(c_1, c_2, \ldots, c_k)$ 中的属性$c_1, c_2, \ldots, c_k$在计算b时都是可用的。

5.1.3 计算次序

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);

an代表依赖图中与序号n的结点有关的属性a



拓扑排序：

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

5.1.4 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 $A \rightarrow X_1X_2...X_n$ 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（$1 \le j \le n$）的继承属性，而该继承属性仅依赖于以下两种情况：

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

5.1.4 S属性定义和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)

5.1.4 S属性定义和L属性定义

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

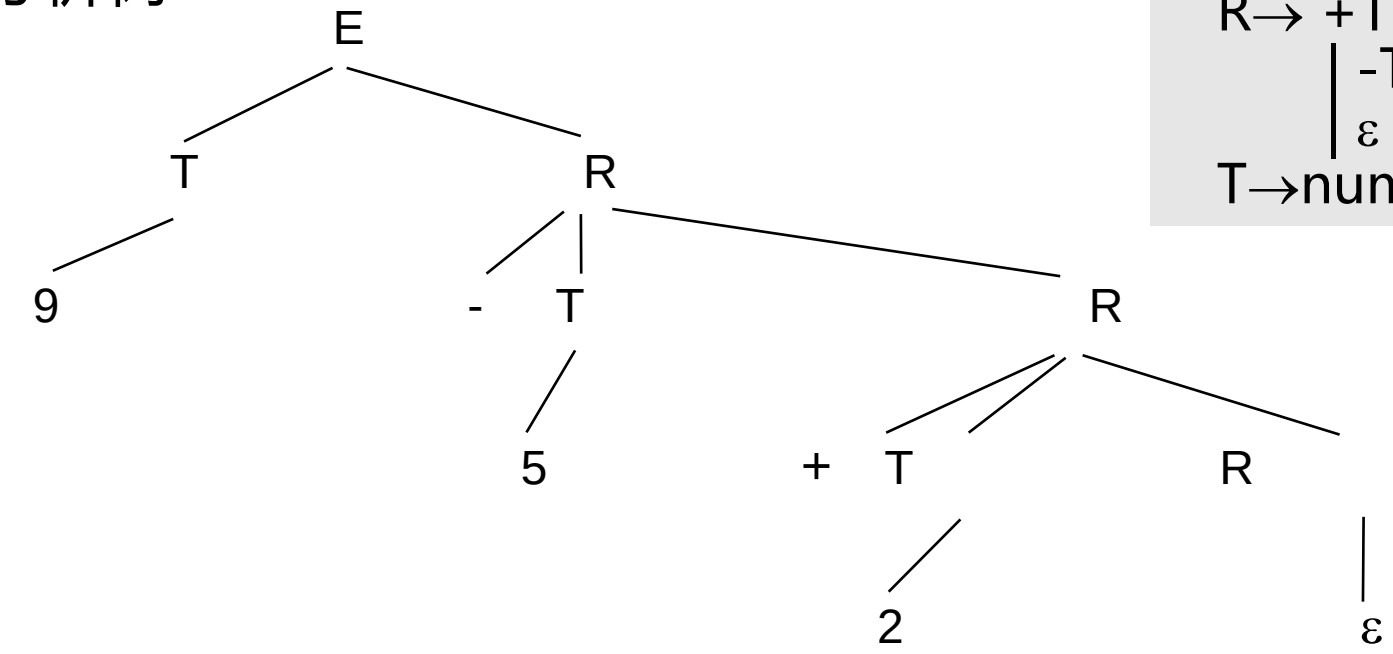
给出了使用语义规则进行属性计算的顺序

分析过程中翻译的注释

5.1.5 翻译方案

翻译方案示例

9-5+2的分析树：



一个简单的翻译方案：

E→TR

R→ +T { print('+') } R_1

| -T { print('-') } R_1

| ε

T→num { print(num.val) }

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+

5.1.5 翻译方案

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：

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

5.1.5 翻译方案

为L属性定义设计翻译方案的原则

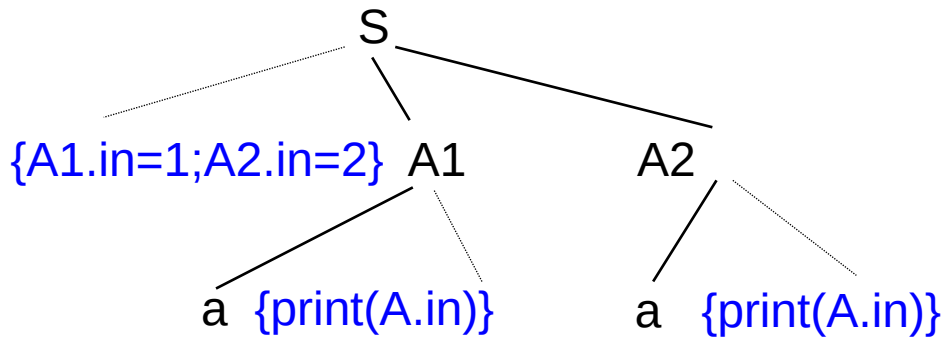
- 产生式右部文法符号的继承属性必须在这个符号以前的语义规则中计算出来
- 一个动作不能引用这个动作右边的文法符号的综合属性
- 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来之后才能计算，这种属性的计算动作放在产生式右端末尾

5.1.5 翻译方案

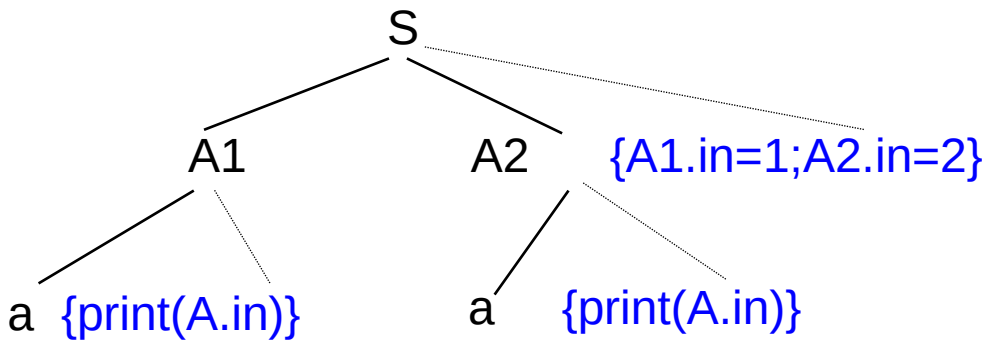
示例：　考虑如下翻译方案：

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

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



√



×

5.1.5 翻译方案

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
$L \rightarrow L_1$,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) }

5.2 S-属性定义的自底向上翻译

S属性定义：

只用综合属性的语法制导定义

5.2.1 为表达式构造语法树的语法制导定义

5.2.2 S属性定义的自底向上实现

5.2.1 为表达式构造语法树的语法制导定义

为什么要构建树?
最初为什么想要建立抽象语法树?
怎么组织抽象语法树的结构，为什么?
怎么实现抽象语法树?

5.2.1 为表达式构造语法树的语法制导定义

- 为什么要构建树?

如果要表示顺序和嵌套，树是最合适的数据结构；

解析树记录了解析器调用规则的次序及匹配的词法单元，构建方式规律，不便于遍历或修改。

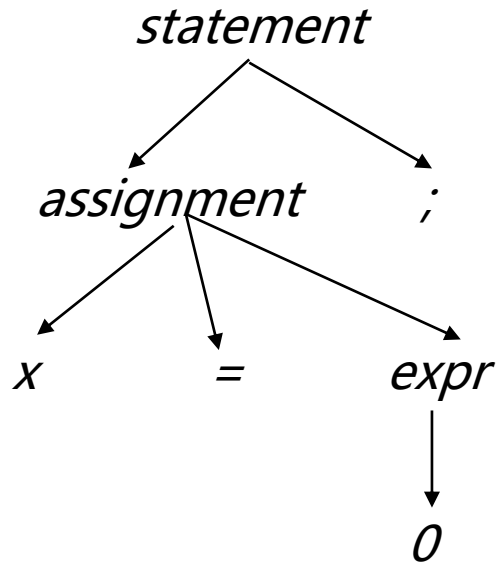
- 最初为什么想要建立抽象语法树?

理想的IR树应该具有以下性质：

紧凑：不包含无用节点

易用：很容易遍历

显意：突出操作符、操作对象，以及它们相互间的关系，不拘泥于文法中的东西。



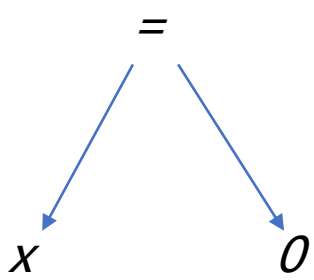
5.2.1 为表达式构造语法树的语法制导定义

抽象语法：

把语法规则中对语义无关紧要的具体规定去掉，剩下来的本质性的东西称为抽象语法。如：

赋值语句：x=y、x:=y、或y→x

抽象形式：assignment(variable,expression)



抽象语法树：

分析树的抽象（或压缩）形式，也称为语法结构树或结构树。

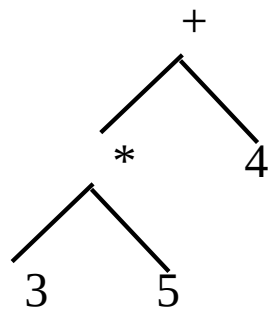
关键思想：子树根节点表示操作符/运算符，其子结点表示它的运算分量。

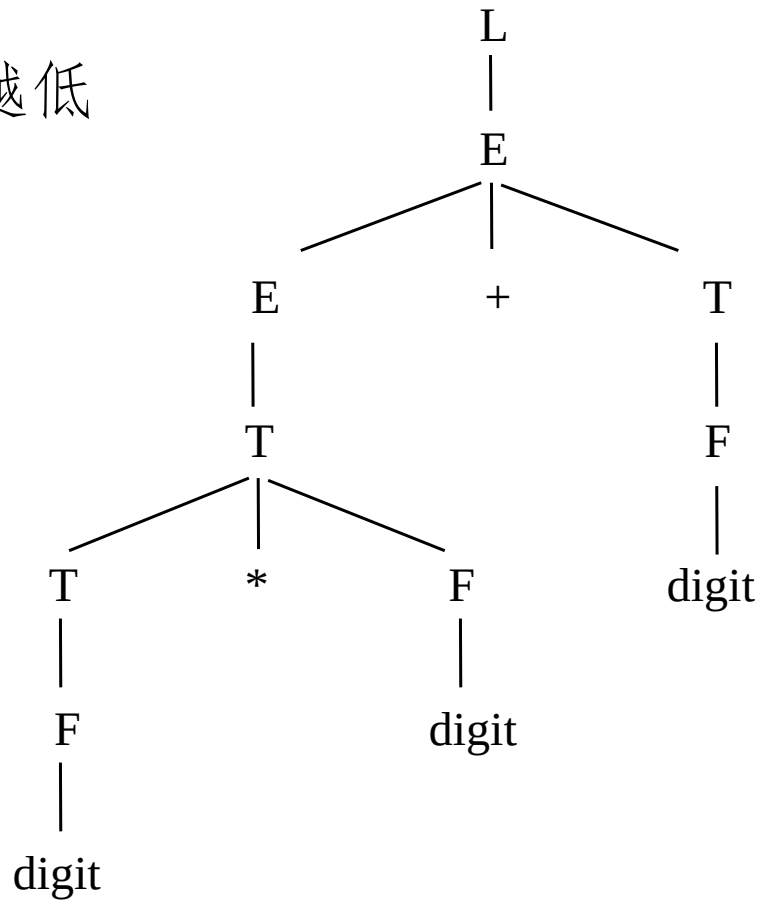
5.2.1 为表达式构造语法树的语法制导定义

抽象语法树如何记录操作的优先级?

AST中操作符的优先级越高，位置就越低

表达式 3*5+4 的语法树





思考：解析树和AST的优劣

AST一定优于解析树吗?

- 解析树有哪些用途?

开发环境中进行语法高亮和错误检查，文本改写系统，使用解析树能把文法规则名来指代语句中的一部分。

5.2.1 为表达式构造语法树的语法制导定义

构造表达式的抽象语法树

表达式的抽象语法树的形式

每一个运算符号或运算分量都对应树中的一个结点

运算符号结点的子结点是该运算符的各个运算分量

每一个结点可包含若干个域：标识域、指针域、属性值域等

在运算符结点中

一个域标识运算符号

其它各域包含指向与各运算分量相应的结点的指针

称运算符号为该结点的标号

5.2.1 为表达式构造语法树的语法制导定义

构造函数

makenode (op, left, right)

建立一个运算符号结点，标号是 op；
域left和right是指向其左右运算分量结点的指针。

makeleaf (id, entry)

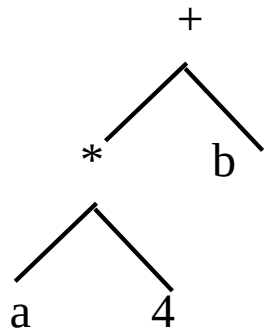
建立一个标识符结点，标号是 id；
域entry是指向该标识符在符号表中的相应条目的指针。

makeleaf (num, val)

建立一个数结点，标号为 num；
域val用于保存该数的值。

5.2.1 为表达式构造语法树的语法制导定义

建立表达式a*4+b的语法树



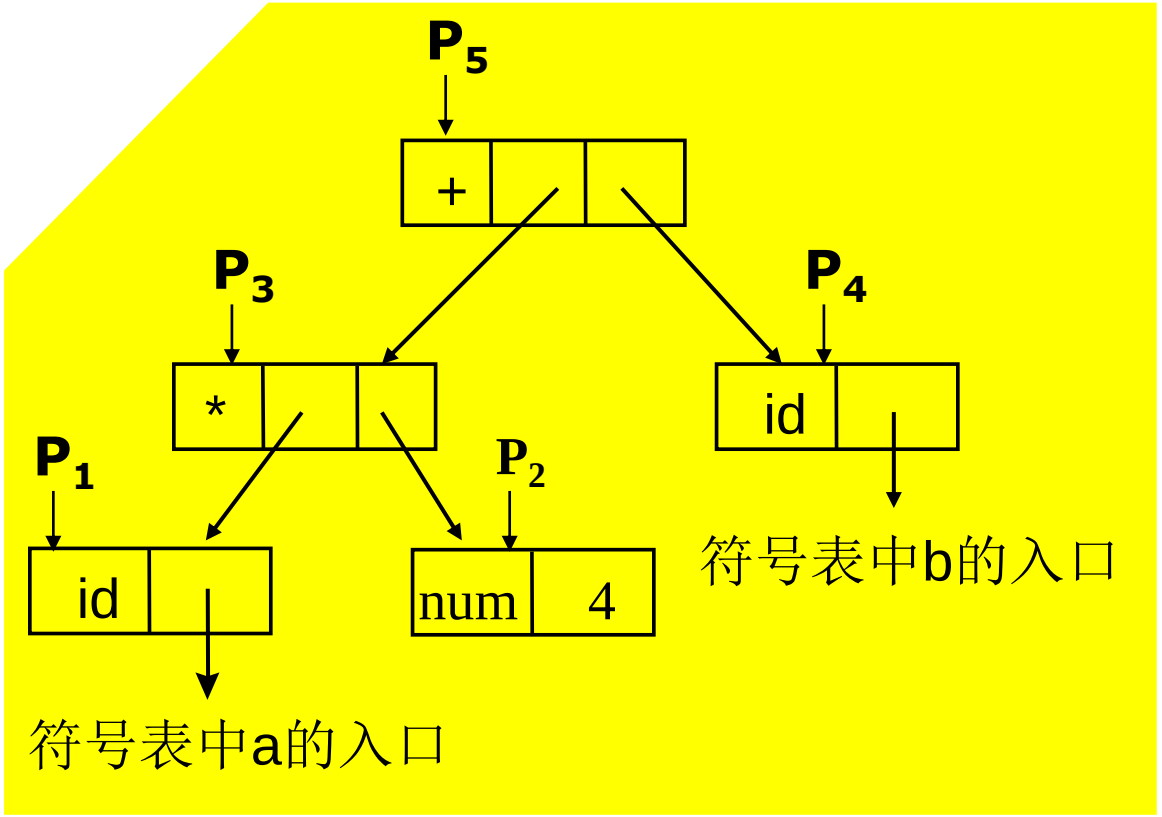
p_1=makeleaf(id, entrya);

p_2=makeleaf(num ,4);

p_3=makenode(‘*’ , p_1, p_2);

p_4=makeleaf(id, entryb);

p_5=makenode(‘+’ ,p_3, p_4);



5.2.1 为表达式构造语法树的语法制导定义

构造表达式语法树的语法制导定义

翻译目标：为表达式创建语法树

产生式语义：创建与产生式左部符号代表的子表达式对应的子树，即创建子树的根结点。

文法符号的属性：记录所建结点， E.nptr、 T.nptr、F.nptr 指向相应子树根结点的指针

产生式的语义动作举例：

$E \rightarrow E_1+T$	E.nptr=makenode('+',E_1.nptr,T.nptr)
$T \rightarrow F$	T.nptr=F.nptr
$F \rightarrow id$	F.nptr=makeleaf(id, id.entry)
$F \rightarrow num$	F.nptr=makeleaf(num, num.val)

5.2.1 为表达式构造语法树的语法制导定义

构造表达式语法树的语法制导定义（续）

产生式	语义规则
$E \rightarrow E_1+T$	E.nptr= makenode('+', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr=T.nptr
$T \rightarrow T_1* F$	T.nptr= makenode('*', T_1.nptr, F.nptr)
$T \rightarrow F$	T.nptr= F.nptr
$F \rightarrow (E)$	F.nptr= E.nptr
$F \rightarrow id$	F.nptr= makeleaf(id, id.entry)
$F \rightarrow num$	F.nptr= makeleaf(num, num.val)

为了记录在构造过程中建立的子树，为每个非终结符号引入一个综合属性 nptr。

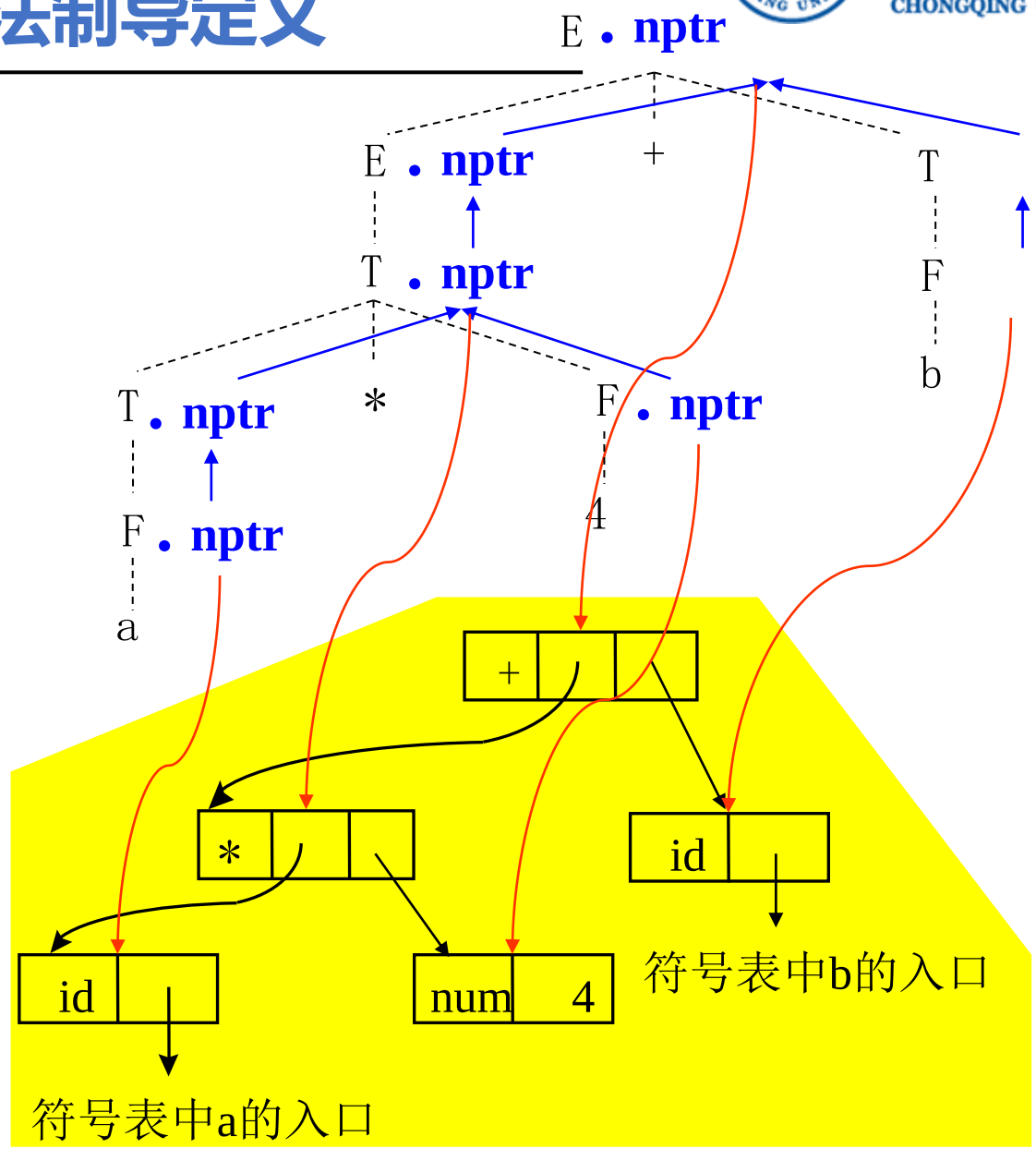
nptr 是一个指针，指向语法树中相应非终结符号产生的表达式子树的根结点。

5.2.1 为表达式构造语法树的语法制导定义

表达式a*4+b的语法树的构造

$E \rightarrow E_1+T$
$E \rightarrow T$
$T \rightarrow T_1*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$
$F \rightarrow num$

F.nptr=makeleaf(id, entrya)
T.nptr=F.nptr
F.nptr=makeleaf(num, 4)
T.nptr=makenode('*', T_1.nptr, F.nptr)
E.nptr=T.nptr
F.nptr=makeleaf(id, entryb)
T.nptr=F.nptr
E.nptr=makenode('+', E_1.nptr, T.nptr)



5.2.1 为表达式构造语法树的语法制导定义

E . nptr

表达式的有向非循环图(dag)

dag与语法树相同的地方：

表达式的每一个子表达式都有一个结点

一个内部结点表示一个运算符号，且它的子结点表示它的运算分量。

dag与语法树不同的地方：

dag中，对应一个公共子表达式的结点具有多个父结点

语法树中，公共子表达式被表示为重复的子树

为表达式创建dag的函数makenode和makeleaf

建立新结点之前先检查是否已经存在一个相同的结点

若已存在，返回一个指向先前已构造好的结点的指针；

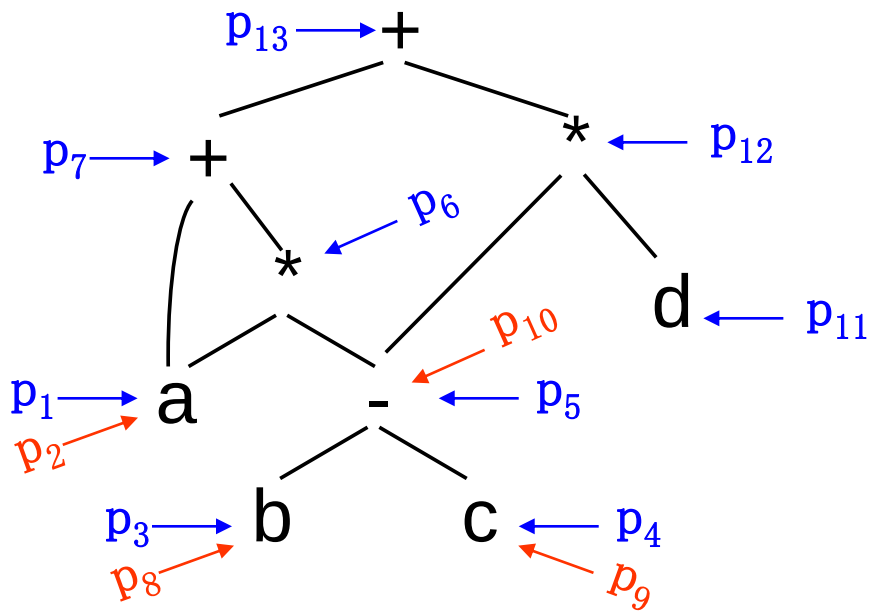
否则，创建一个新结点，返回指向新结点的指针。

5.2.1 为表达式构造语法树的语法制导定义

为表达式 a+a*(b-c)+(b-c)*d 构造dag

函数调用

p_1=makeleaf(id, a);
p_2=makeleaf(id, a);
p_3=makeleaf(id, b);
p_4=makeleaf(id, c);
p_5=makenode('-', p_3, p_4);
p_6=makenode('*', p_2, p_5);
p_7=makenode('+', p_1, p_6);
p_8=makeleaf(id, b);
p_9=makeleaf(id, c);
p_{10}=makenode('-', p_8, p_9);
p_{11}=makeleaf(id, d);
p_{12}=makenode('*', p_{10}, p_{11});
p_{13}=makenode('+', p_7, p_{12});



扩展：解析树模式

构建解析树时，每进入一个规则就要新建一个内节点，代码框架如下：

```
void 《规则名》(){
    RuleNode r = new RuleNode(“《规则名》");
    if (root==null ) root=r;                          //说明这是起始规则
    else currentNode.addChild(r);                     //将规则添加到当前的节点上
    Parse Tree_Save = currentNode;
    currentNode = r;                                  //将要递归下降分析这条规则
    《原始规则的代码》
     currentNode = _save;                             //将节点名恢复为之前的保存状态
}
```

扩展：解析树模式

通用解析树节点类

```
import java.util.*;
// 节点都是此类的实例，所以实际上并没有Node类
public abstract class ParseTree {
    public List<ParseTree> children; // normalized child list
    public RuleNode addChild(String value) {
        RuleNode r = new RuleNode(value);
        addChild(r);
        return r;
    }
    public TokenNode addChild(Token value) {
        TokenNode t = new TokenNode(value);
        addChild(t);
        return t;
    }
    public void addChild(ParseTree t) {
        if ( children==null ) children = new ArrayList<ParseTree>();
        children.add(t);
    }
}
```

扩展：解析树模式

重庆大学 CHONGQING UNIVERSITY

```
public class TokenNode extends ParseTree {
    public Token token;
    public TokenNode(Token token) { this.token = token; }
}

public class RuleNode extends ParseTree {
    public String name;
    public RuleNode(String name) { this.name = name; }
}
```

扩展：同型树AST模式

同型树（带有子节点列表的通用树节点）

```
public class AST {        // 同型树节点类型
    Token token;          // 节点源自哪个词法单元?
    List<AST> children; // 操作对象
    public AST()    {; } //创建作为根节点的空节点
    public AST(Token token)    { this.token = token; }
    /*根据词法单元创建节点，主要用于虚节点*/
    public AST(int tokenType) { this.token = new Token(tokenType); }
    /*对于同一类型的节点，外部访问者会执行同样的代码。*/
    public int getNodeType() { return token.type; }

    public void addChild(AST t) {
        if ( children==null ) children = new ArrayList<AST>();
        children.add(t);
    }
    public boolean isNil() { return token==null; }
```

扩展：同型树AST模式

打印同型树的文本形式： (root child1 child2 ...)

```
/** 生成单个节点的文本形式*/
    public String toString() { return token!=null?token.toString():"nil"; }
    /** 生成整个树而不是某个节点的文本形式*/
    public String toStringTree() {
        if ( children==null || children.size()==0 ) return this.toString();
        StringBuilder buf = new StringBuilder();
        if ( !isNil() ) {
            buf.append("(");
            buf.append(this.toString());
            buf.append(' ');
        }
        for (int i = 0; i < children.size(); i++) {
            AST t = (AST)children.get(i); // 规范化子节点列表
            if ( i>0 ) buf.append(' ');
            buf.append(t.toStringTree());
        }
        if ( !isNil() ) buf.append(")");
        return buf.toString();
    }
```

扩展：规范化异型AST模式

当需要为节点存储专有数据时，采用多种节点类型实现AST，用规范化列表来表示子节点

```
public abstract class ExprNode extends AST {
    public static final int tINVALID = 0; // 非法表达式
    public static final int tINTEGER = 1; // 整数表达式
    public static final int tVECTOR = 2;  // 向量表达式
    /** 记录每个expr节点的表达式类型（整数型或向量型）
     *  指的是表达式值的类型，不要与 getNodeType()混淆
     *  getNodeType()为外部访问者使用，能区分不同节点 */
    int evalType;

    public int getEvalType() { return evalType; }
    public ExprNode(Token payload) { super(payload); }
    /** 如果ExprNode已经知道表达式的类型，转换文本时就该加上 */
    public String toString() {
        if ( evalType != tINVALID ) {
            return super.toString()+"<type="+
                    (evalType == tINTEGER ? "tINTEGER" : "tVECTOR")+">";
        }
        return super.toString();
    }
}
```

扩展：规范化异型AST模式

当构建+子树时，不用在泛型节点后逐个添加子节点，用AddNode的构造方法

```
public class AddNode extends ExprNode {
    public AddNode(ExprNode left, Token addToken, ExprNode right) {
        super(addToken);
        addChild(left);
        addChild(right);
    }
    public int getEvalType() { // ...
```

整型和向量等操作对象的节点类型都是ExprNode的子节点

```
public class IntNode extends ExprNode {
    public IntNode(Token t) { super(t); evalType = tINTEGER; }
}
public class VectorNode extends ExprNode {
    public VectorNode(Token t, List<ExprNode> elements) {
        super(t); // 记录向量词法单元
        evalType = tVECTOR;
        for (ExprNode e : elements) { addChild(e); } // 加到子节点列表
    }
}
```

扩展：不规则异型AST模式

多种节点类型实现AST，用不规则列表来表示子节点

抽象基类HeteroAST没有规范列表字段

```
public abstract class HeteroAST {    // 异型AST节点类型
    Token token;                     // 原始词法单元
    public HeteroAST()               { ; }
    public HeteroAST(Token t)        { token = t; }
    /** 根据词法单元类型创建节点*/
    public HeteroAST(int tokenType) { this.token = new Token(tokenType); }

    public String toString() { return token.toString(); }
    /** 不同节点调用不同的toStringTree*/
    public String toStringTree()    { return toString(); }
}
```

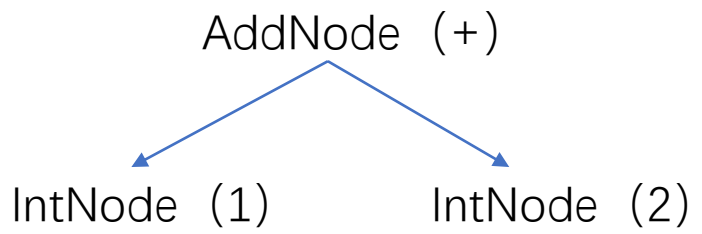
扩展：不规则异型AST模式

节点类AddNode，子节点字段有自己的名字，还有专用方法。

```
public class AddNode extends ExprNode {
    ExprNode left, right; // 不规则子节点，有节点自定义的名字
    public AddNode(ExprNode left, Token addToken, ExprNode right) {
        super(addToken);
        this.left = left;
        this.right = right;
    }
    public String toStringTree() {
        if ( left==null || right==null ) return this.toString();
        StringBuilder buf = new StringBuilder();
        buf.append("(");
        buf.append(this.toString());
        buf.append(' ');
        buf.append(left.toStringTree());
        buf.append(' ');
        buf.append(right.toStringTree());
        buf.append(")");
        return buf.toString();
    }
}
```

扩展：树的遍历模式

通常采用深度优先搜索算法，从根节点出发，递归的依次遍历子节点。
遍历过程到达节点t，称为发现节点t。访问，在发现和结束之间执行的一些动作。
节点发现的顺序是固定的，但确定发现顺序后，遍历顺序也可能有差异，这取决于操作相关的代码在walk（）中的位置。

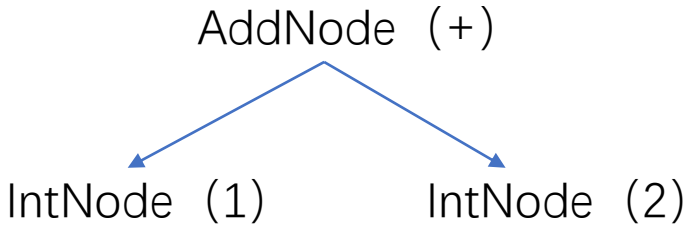



```
public abstract class ExprNode {
    Token token; //原始词法单元
    public void walk();  //基本遍历操作
}
```

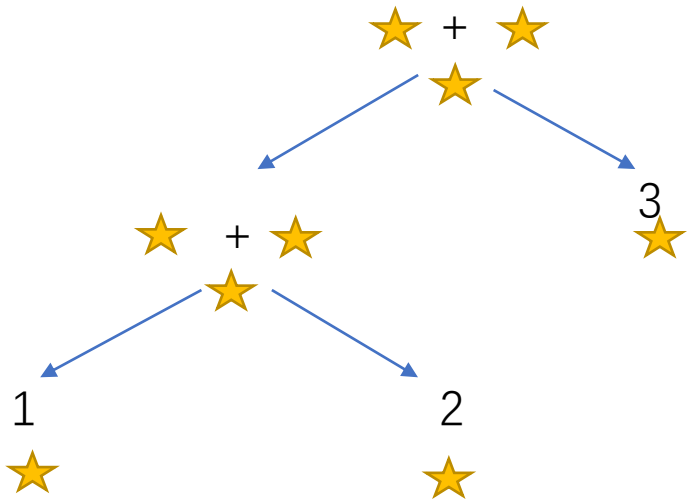
```
public class IntNode extends ExprNode {
    public void walk() {; } //无子节点，什么都不做
}

public class AddNode extends ExprNode {
   ExprNode left,right; //不规则子节点，有自定义名字
   public walk() {
      left.walk(); //遍历左子树
      right.walk(); //遍历右子树
 }
}
```

扩展：树的遍历模式



调用根节点+的walk（）方法，节点发现顺序为+12



节点发现顺序为： ++123
结束顺序为：12+3+
遍历节点间的操作顺序： 1+2+3

```
public void walk() {
        《前序操作》
       left.walk(); //遍历左子树
        《中序操作》
       right.walk(); //遍历右子树
        《后序操作》
}
```

扩展：封装访问节点模式

主要目的：隔离遍历代码和树的定义代码，访问者在遍历对象之外。

```
public class IndependentPostOrderPrintVisitor {
    // 调度方法，根据参数类型调用其他方法
    public void print(ExprNodee n) {
        if (n.getClass()== AddNode.class) print ((AddNode) n );
        else if (n.getClass()== IntNode.class) print ((IntNode) n );
        else 《错误，不支持这种节点类型》
    }

    public void print (AddNode n) {
       print(n.left);
       print(n.right);
       System.out.print(n.token); // 后续遍历，操作放在这里
    }

    public void print (IntNode n) {
       System.out.print(n.token);
    }
}
```

5.2.2 S-属性定义的自底向上实现

已知

LR分析方法中，分析程序使用一个栈来存放已经分析过的子树的信息。
分析树中某结点的综合属性由其子结点的属性值计算得到
LR分析程序在分析输入符号串的同时可以计算综合属性

考虑

如何保存文法符号的综合属性值?
保存属性值的数据结构怎样与分析栈相联系?
怎样保证：每当进行归约时，由栈中正在归约的产生式右部符号的属性值计算其左部符号的综合属性值。

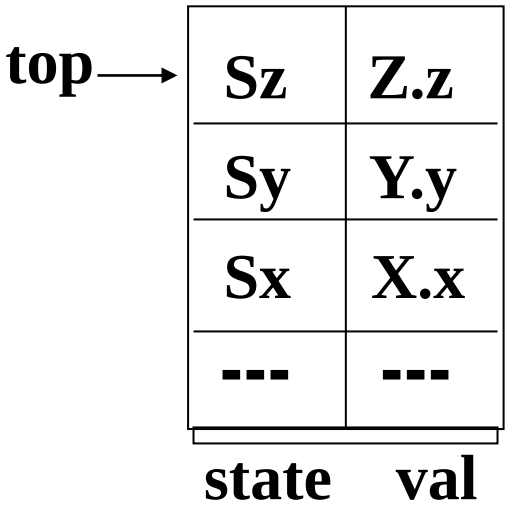
5.2.2 S-属性定义的自底向上实现

修改分析栈

目的：使之能够保存综合属性

做法：在分析栈中增加一个域，存放综合属性值

例：带有综合属性域的分析栈



栈由一对数组state和val实现

state元素是指向LR(1)分析表中状态的指针（或索引）

如果state[i]保存对应符号A的状态，val[i]中就存放分析树中与结点A对应的属性值。

假设综合属性刚好在每次归约前计算

A→XYZ 对应的语义规则是 A.a=f(X.x, Y.y, Z.z)

5.2.2 S-属性定义的自底向上实现

修改分析程序

对于终结符号

其综合属性值由词法分析程序产生

当分析程序执行移进操作时，其属性值随状态符号一起入栈。

为每个语义规则编写一段代码，以计算属性值

对每一个产生式A→XYZ

把属性值的计算与归约动作联系起来

归约前，执行与产生式相关的代码段

归约：右部符号的相应状态及其属性出栈

左部符号的相应状态及其属性入栈

LR分析程序中应增加计算属性值的代码段

5.2.2 S-属性定义的自底向上实现

例：用LR分析程序实现表达式求值

产生式	语义规则	代码段
L→E	print(E.val)	print(val[top])
$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$	val[ntop]=val[top-2]+val[top]
E→T	E.val=T.val	
$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$	val[ntop]=val[top-2]*val[top]
T→F	T.val=F.val	
F→(E)	F.val=E.val	val[ntop]=val[top-1]
F→digit	F.val=digit.lexval	

栈指针变量 top 和 ntop 的控制：

当用 A→β 归约时，若|β|=r，在执行相应的代码段之前， ntop=top-r+1。

在每一个代码段被执行之后，top=ntop

5.2.2 S-属性定义的自底向上实现

表达式构造改进的SLR(1)分析表

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

对 3*5+4 进行分析的动作序列 (分析表见表4-8)

步骤	输入	分析栈	分析动作
(1)	3*5+4$	state: 0	
		val: -	移进 5
(2)	*5+4$	state: 0 5	
		val: - 3	归约，用F→digit goto[0,F]=3
(3)	*5+4$	state: 0 3	
		val: - 3	归约，用T→F goto[0,T]=2
(4)	*5+4$	state: 0 2	
		val: - 3	移进 7
(5)	5+4$	state: 0 2 7	
		val: - 3 -	移进 5
(6)	+4$	state: 0 2 7 5	
		val: - 3 - 5	归约，用F→digit goto[7,F]=10
(7)	+4$	state: 0 2 7 10	
		val: - 3 - 5	归约，用T→T*F goto[0,T]=2

对 3*5+4 进行分析的动作序列 (分析表见表4-8)

步骤	输入	分析栈	分析动作
(8)	+4$	state: 0 2	
		val: - 15	归约，用E→T goto[0,E]=1
(9)	+4$	state: 0 1	
		val: - 15	移进 6
(10)	4$	state: 0 1 6	
		val: - 15 -	移进 5
(11)	$	state: 0 1 6 5	
		val: - 15 - 4	归约，用F→digit goto[6,F]=3
(12)	$	state: 0 1 6 3	
		val: - 15 - 4	归约，用T→F goto[6,T]=9
(13)	$	state: 0 1 6 9	
		val: - 15 - 4	归约，用E→E+T goto[0,E]=1
(14)	$	state: 0 1	
		val: - 19	接受

5.3 L-属性定义的自顶向下翻译

在自顶向下的分析过程中实现L属性定义的翻译

预测分析方法对文法的要求：

- 不含左递归
- $A\rightarrow\alpha|\beta$，则$FIRST(\alpha)\cap FIRST(\beta)=\phi$

5.3 L-属性定义的自顶向下翻译

5.3.1 消除翻译方案中的左递归

例：考虑对简单表达式求值的语法制导定义

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

翻译方案：

(1) $L \to E$ { print(E.val) }
(2) $E \to E_1+T$ { $E.val=E_1.val+T.val$ }
(3) $E \to T$ { E.val=T.val }
(4) $T \to T_1*F$ {$T.val=T_1.val*F.val$ }
(5) $T \to F$ { T.val=F.val }
(6) $F \to (E)$ { F.val=E.val }
(7) $F \to digit$ { F.val=digit.val }

消除左递归的方法：

$A \to A\alpha|\beta$

替换为： $A \to \beta R$

$R \to \alpha R|\varepsilon$

由(2)和(3)有：

(2′) $E \to T$ {E.val=T.val} M

(3′) $M \to +T$ {$E.val=E_1.val+T.val$} M_1

(3") $M \to \varepsilon$

继承属性M.i：表示在M之前已经推导出的子表达式的值
综合属性M.s：表示在M完全展开之后得到的表达式的值

5.3 L-属性定义的自顶向下翻译

为(3")设置把M.i传递给M.s的语义动作，得到：

(3") M→ε {M.s=M.i}

对于(2′)， E→T {E.val=T.val} M 通过M的属性 M.s 和 M.i 完成E和T的综合属性的传递E.val=T.val，得到：

(2′) E→T {M.i=T.val}
M {E.val=M.s}

对于(3′) M→+T {E.val=E_1.val+T.val} M_1

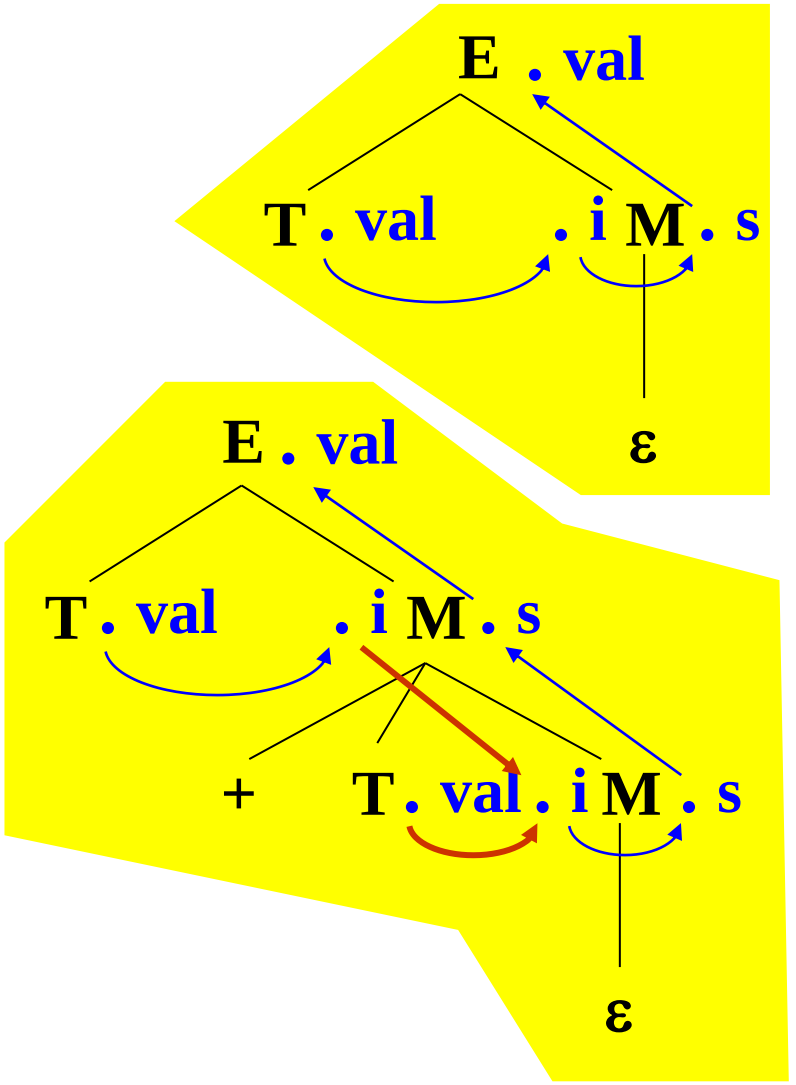
M_1 .i的语义规则为： M_1.i=M.i+T.val

M.s的语义规则为： M.s=M_1.s

于是得到：

(3′) M→+T {M_1.i=M.i+T.val}
M_1 {M.s=M_1.s}

同样，通过引入非终结符号N，
可以得到(4)和(5)的变换结果

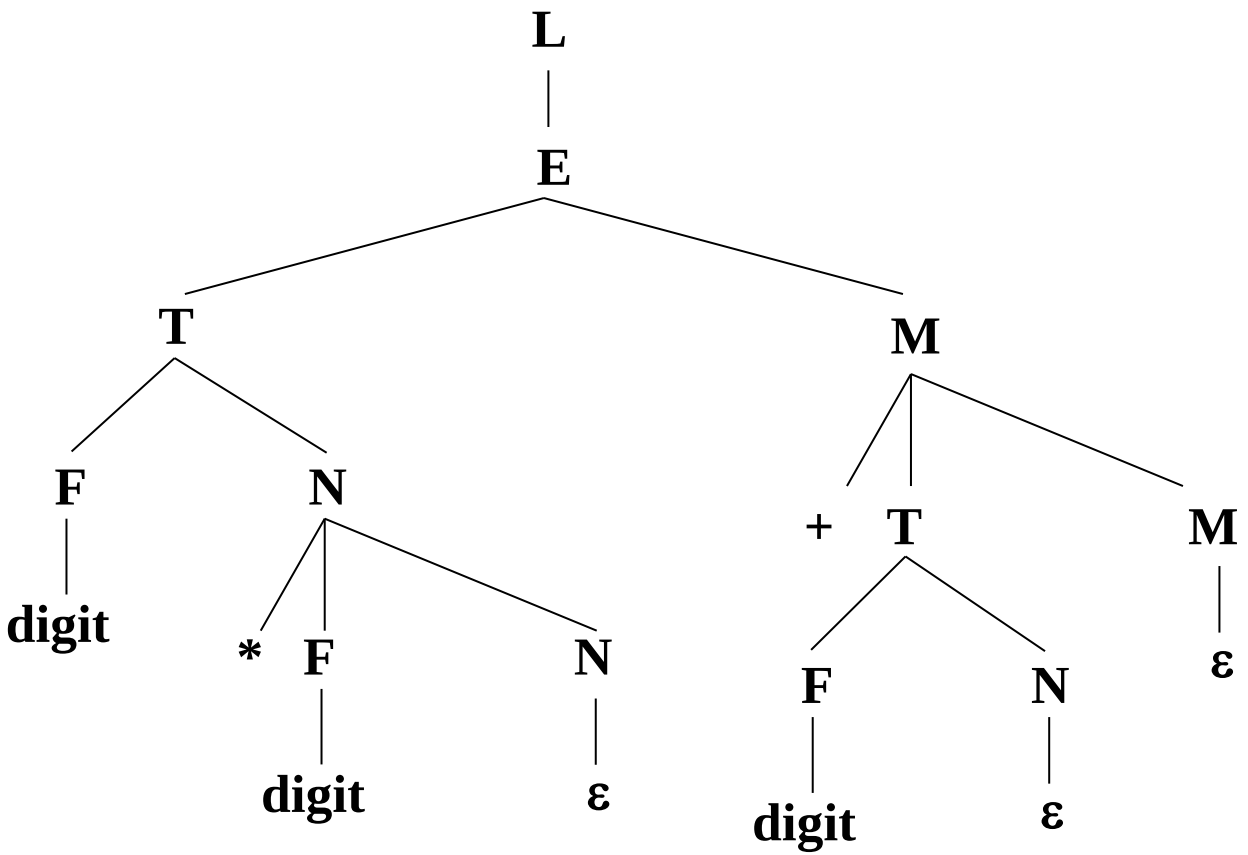


5.3 L-属性定义的自顶向下翻译

翻译方案

L→ E {print(E.val)}
E→ T {M.i=T.val}
　　M {E.val=M.s}
M→ +T {M_1.i=M.i+T.val}
　　M_1 {M.s=M_1.s}
M→ ε {M.s=M.i}
T→F {N.i=F.val}
　　N {T.val=N.s}
N→ *F {N_1.i=N.i*F.val}
　　N_1 {N.s=N1.s}
N→ ε {N.s=N.i}
F→(E) {F.val=E.val}
F→digit {F.val=digit.lexval}

表达式3*5+4的翻译过程



5.3.2 预测翻译程序的设计

从翻译方案出发构造自顶向下的语法制导翻译程序

算法5.2：构造语法制导的预测翻译程序

输入：基础文法适合于预测分析的语法制导翻译方案

输出：语法制导翻译程序

方法：（修改预测分析程序的构造技术）

(1) 为每个非终结符号A建立一个函数(可以是递归函数)

A的每一个继承属性对应函数的一个形参

A的综合属性作为函数的返回值

A产生式中的每个文法符号的每个属性都对应一个局部变量

(2) A的函数的代码由多个分支组成

算法5.2：构造语法制导的预测翻译程序（续）

(3) 与每个产生式相关的程序代码

- 按照从左到右的顺序考虑产生式右部的记号、非终结符号和语义动作
- 对带有综合属性x的记号X
  - 把属性x的值保存于为X.x声明的变量中
  - 产生一个匹配记号X的调用
  - 推进扫描指针
- 对非终结符号B
  - 产生一个函数调用语句c=B(b_1,b_2,...,b_k)
  - b_i(i=1,2,...,k)是对应于B的继承属性的变量
  - c是对应于B的综合属性的变量
- 对每一个语义动作
  - 把动作代码复制到分析程序中
  - 用代表属性的变量代替翻译方案中引用的属性

算法5.2：构造语法制导的预测翻译程序（续）

示例：为简单表达式求值的翻译方案构造翻译程序

为每个非终结符号构造一个函数：

```
void fxL(void)
int  fxE(void)
int  fxM(int in)
int  fxT(void)
int  fxN(int in)
int  fxF(void)
```

L→ E {print(E.val)}

E→ T {M.i=T.val}

M {E.val=M.s}

M→ +T {M_1.i=M.i+T.val}

M_1 {M.s=M_1.s}

M→ ε {M.s=M.i}

T→F {N.i=F.val}

N {T.val=N.s}

N→ *F {N_1.i=N.i*F.val}

N_1 {N.s=N1.s}

N→ ε {N.s=N.i}

F→(E) {F.val=E.val}

F→digit {F.val=digit.lexval}

算法5.2：构造语法制导的预测翻译程序（续）

与E→TM、M→+TM|ε 相应的分析过程

```
//E→TM

void proc_E(void)  {
      proc_T();
      proc_M();
};
```

```
//M→+TM|ε

void proc_M(void)  {
   if (lookahead== '+') {
      match( '+' );
      proc_T();
      proc_M();
   }
};
```

算法5.2：构造语法制导的预测翻译程序（续）

实现翻译方案的函数

```
int  fxE(void)  {
   int eval, tval, mi, ms;
   tval= fxT();
   mi=tval;
   ms= fxM(mi);
   eval=ms;
   return eval;
}
```

E→ T {M.i=T.val}
M {E.val=M.s}

算法5.2：构造语法制导的预测翻译程序（续）

实现翻译方案的函数

```
int  fxM(int in)  {
   int tval, i1, s1, s;
   char addoplexeme;
   if (lookahead== '+') {   // 产生式M→+TM
      addoplexeme=lexval;
      match( '+' );
      tval=fxT();
      i1=in+tval;
      s1=fxM(i1);
      s=s1;
   };
   else  s=in; // 产生式M→ε
   return s
}
```

$M \rightarrow +$
$\quad T \ \{ M_1.i=M.i+T.val \}$
$\quad M_1 \ \{ M.s=M_1.s \}$
$M \rightarrow \varepsilon \ \{ M.s=M.i \}$

5.4 L属性定义的自底向上翻译

在自底向上的分析过程中实现L属性定义的翻译
可以实现任何基于LL(1)文法的L属性定义
可以实现许多（不是全部）基于LR(1)文法的L属性定义

5.4.1 移走翻译方案中嵌入的语义规则

自底向上地处理继承属性

等价变换：

使所有嵌入的动作都出现在产生式的右端末尾

方法：

在基础文法中引入新的产生式，形如：$M \rightarrow \varepsilon$

M：标记非终结符号，用来代替嵌入在产生式中的动作

把被M替代的动作放在产生式$M \rightarrow \varepsilon$的末尾

5.4.1 移走翻译方案中嵌入的语义规则

示例：去掉如下翻译方案中嵌入的动作：

E→TR

R→+T {print('+')} R |-T {print('-')} R | ε

T→num {print(num.val)}

标记非终结符号M和N，及产生式 M→ε 和 N→ε

用M和N替换出现在R产生式中的动作

新的翻译方案

E→TR

R→+TMR | -TNR | ε

T→num {print(num.val)}

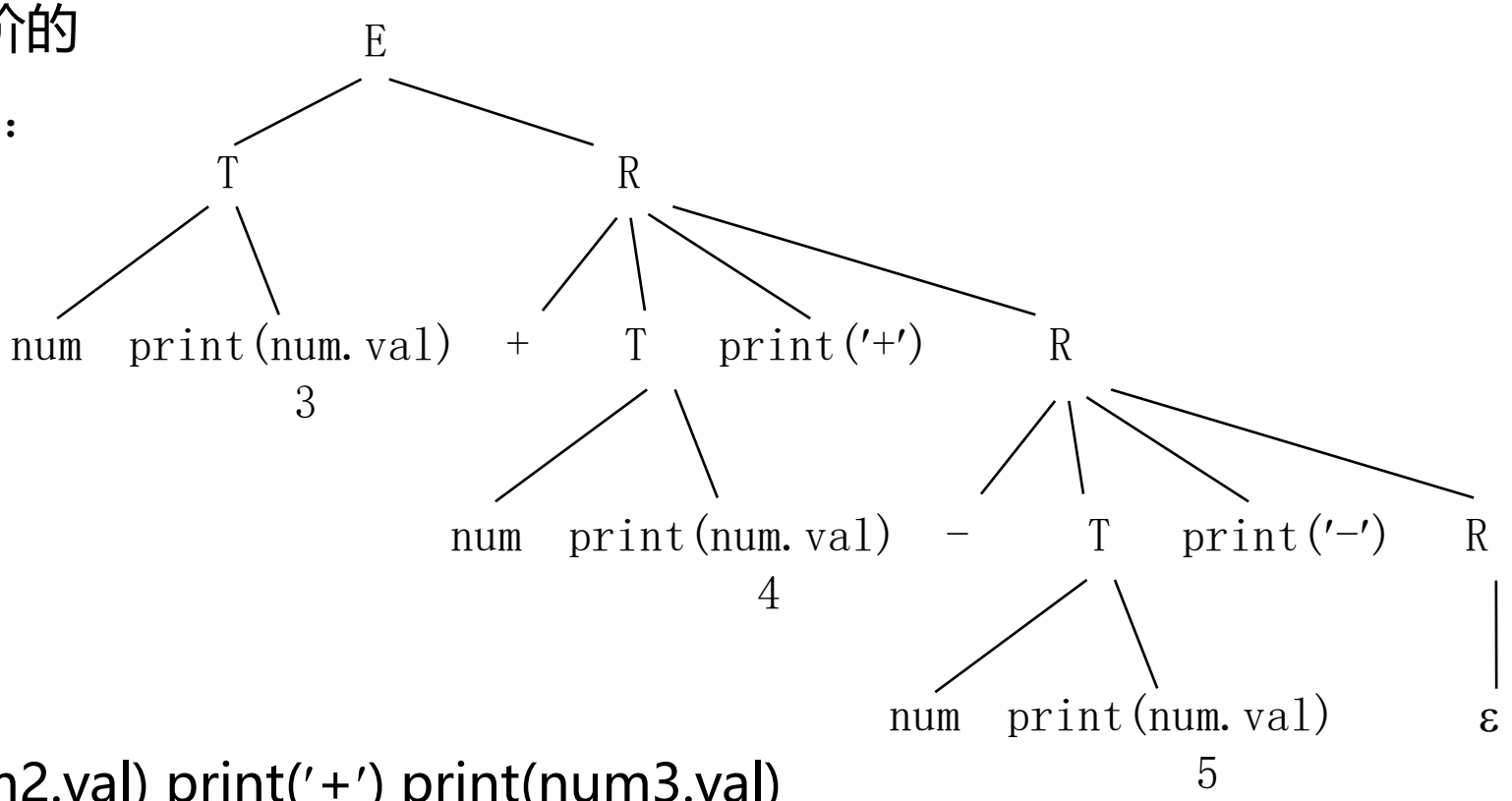
M→ε {print('+')}

N→ε {print('-')}

5.4.1 移走翻译方案中嵌入的语义规则

变换前、后的翻译方案是等价的

变换前，表达式3+4-5的分析树：



深度优先的顺序进行遍历

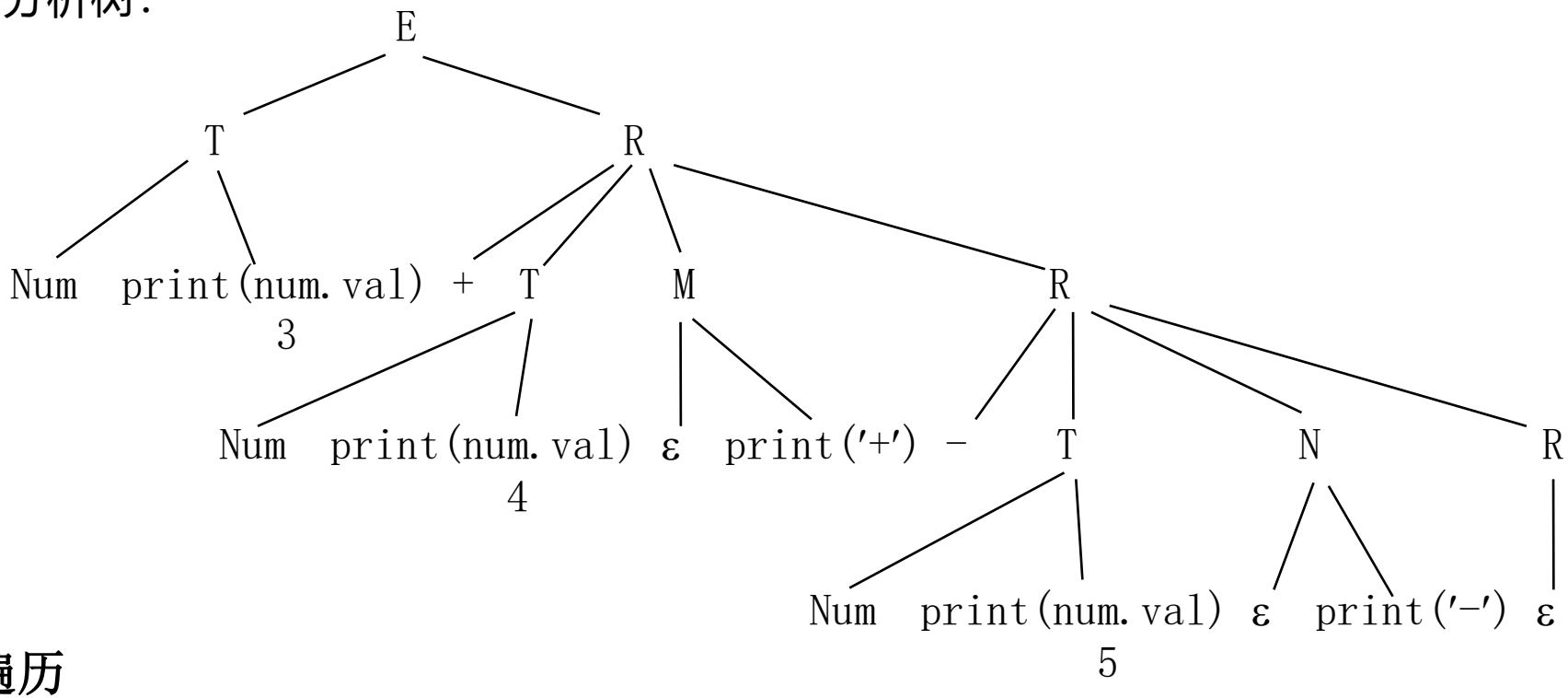
print(num1.val) print(num2.val) print('+') print(num3.val) print('-')

动作执行的结果是：34+5-

5.4.1 移走翻译方案中嵌入的语义规则

变换前、后的翻译方案是等价的（续）

变换后，表达式3+4-5的分析树：



深度优先的顺序进行遍历

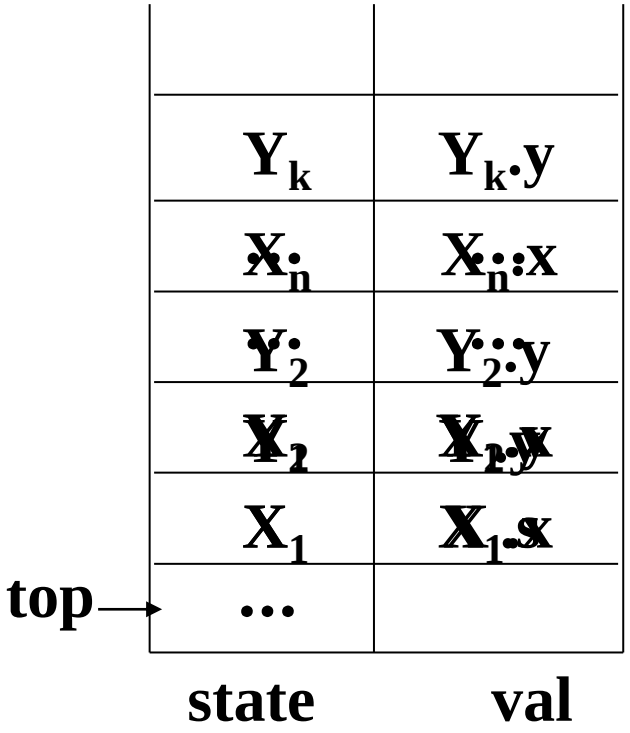
print(num1.val) print(num2.val) print('+') print(num3.val) print('-')

动作执行的结果是：34+5-

5.4.2 直接使用分析栈中的继承属性

LR分析程序对产生式A→XY的归约

考虑分析过程中属性的计算



$X \rightarrow X_1 X_2 \ldots X_n$

$Y \rightarrow Y_1 Y_2 \ldots Y_k$

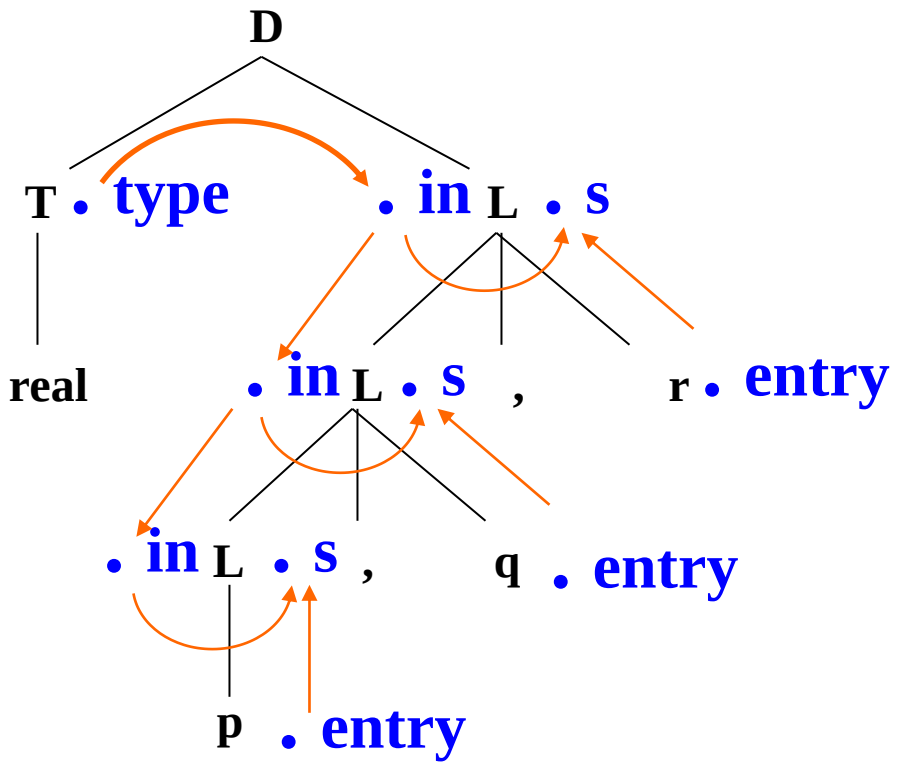
$Y.i = X.s$

复制规则的重要作用

输入符号串：real p, q, r

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)}



例：应用继承属性，用复制规则传递标识符的类型

输入	栈		分析动作
real p,q,r$	state:		
	val:		移进
p,q,r$	state: real		
	val: real		归约，用T→real
p,q,r$	state: T		
	val: real		移进
,q,r$	state: T	p	
	val: real	pentry	归约，用L→id
,q,r$	state: T	L	
	val: real	-	移进
q,r$	state: T	L ,	
	val: real	- ,	移进
,r$	state: T	L , q	
	val: real	- , qentry	归约，用L→L,id
,r$	state: T	L	
	val: real	-	移进
r$	state: T	L ,	
	val: real	- ,	移进
$	state: T	L , r	
	val: real	- , rentry	归约，用L→L,id
$	state: T	L	
	val: real	-	归约，用D→TL
$	state: D		
	val: -		接受

计算属性值的代码段

产生式	代码段
D→TL	
T→int	val[ntop]=integer
T→real	val[ntop]=real
L→L,id	addtype(val[top],val[top-3])
L→id	addtype(val[top],val[top-1])

top和ntop分别是归约前和归约后的栈顶指针

当用产生式 L→id 归约时，L.in 的位置?

当用产生式 L→L, id 进行归约时，L.in 的位置?

和 L.in 有关的动作？

5.4.3 变换继承属性的计算规则

要想从栈中取得继承属性，当且仅当文法允许属性值在栈中存放的位置可以预测。

例：属性值在栈中的位置不可预测的语法制导定义

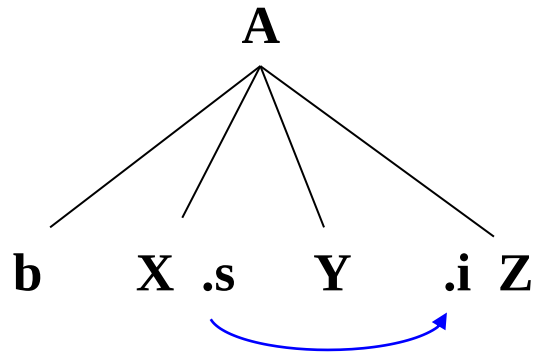
	产生式	语义规则
(1)	A→aXZ	Z.i=X.s
(2)	A→bXYZ	Z.i=X.s
(3)	X→x	X.s=5
(4)	Y→y	Y.s=7
(5)	Z→z	Z.s=g(Z.i)

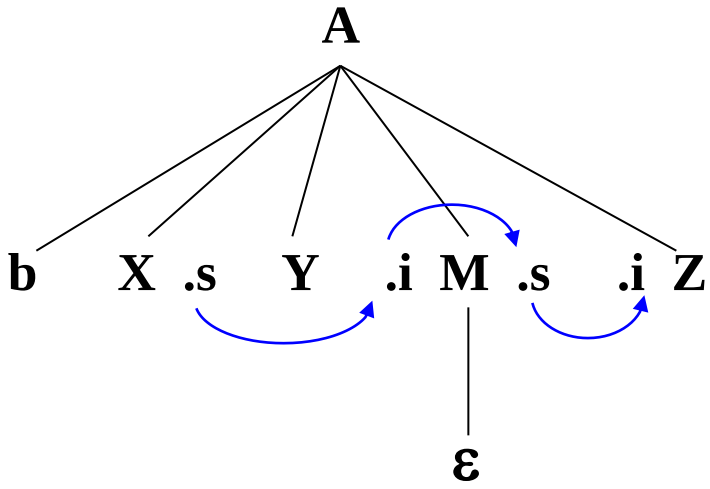
当用Z→z进行归约时，
Z.i 可能在 val[top-1] 处
也可能在 val[top-2] 处

模拟继承属性的计算

引入标记非终结符号，对原语法制导定义进行等价变换

	产生式	语义规则
(1)	A→aXZ	Z.i=X.s
(2')	A→bXYMZ	M.i=X.s; Z.i=M.s
(3)	X→x	X.s=5
(4)	Y→y	Y.s=7
(5)	Z→z	Z.s=g(Z.i)
(6)	M→ε	M.s=M.i



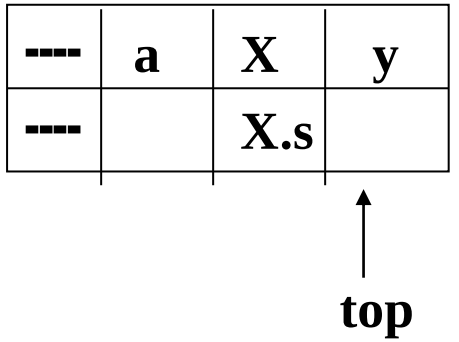


用标记非终结符号模拟非复制规则的语义规则

例：考虑如下的产生式及语义规则：

$A \rightarrow aXY$　　$Y.i=f(X.s)$

$Y \rightarrow y$　　$Y.s=g(Y.i)$



引入标记非终结符号N

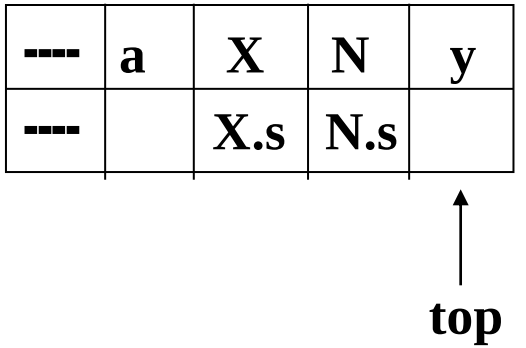
$A \rightarrow aXNY$　　$N.i=X.s$；$Y.i=N.s$

$N \rightarrow \varepsilon$　　$N.s=f(N.i)$

$Y \rightarrow y$　　$Y.s=g(Y.i)$

所有继承属性均由复制规则实现

继承属性在栈中的位置可以预测



算法5.3：L属性定义的自底向上分析和翻译

输入：基础文法是LL(1)文法的L属性定义

输出：在分析过程中计算所有属性值的分析程序

方法：

假设：

每个非终结符号A 都有一个继承属性 A.i

每一个文法符号X 都有一个综合属性 X.s

(1) 对每个产生式 $A\rightarrow X_1X_2...X_n$，引入n个新的标记非终结符号$M_1$、$M_2$、…、$M_n$;

用产生式 $A\rightarrow M_1X_1M_2X_2...M_nX_n$ 代替原来的产生式

X_j 的继承属性与标记非终结符号 M_j 相联系；属性 X_j.i (也就是 M_j.s)总是在 M_j 处计算，且发生在开始做归约到 X_j 的动作之前。

算法5.3：L属性定义的自底向上分析和翻译（续）

(2) 在自底向上分析过程中，各个属性的值都可以被计算出来

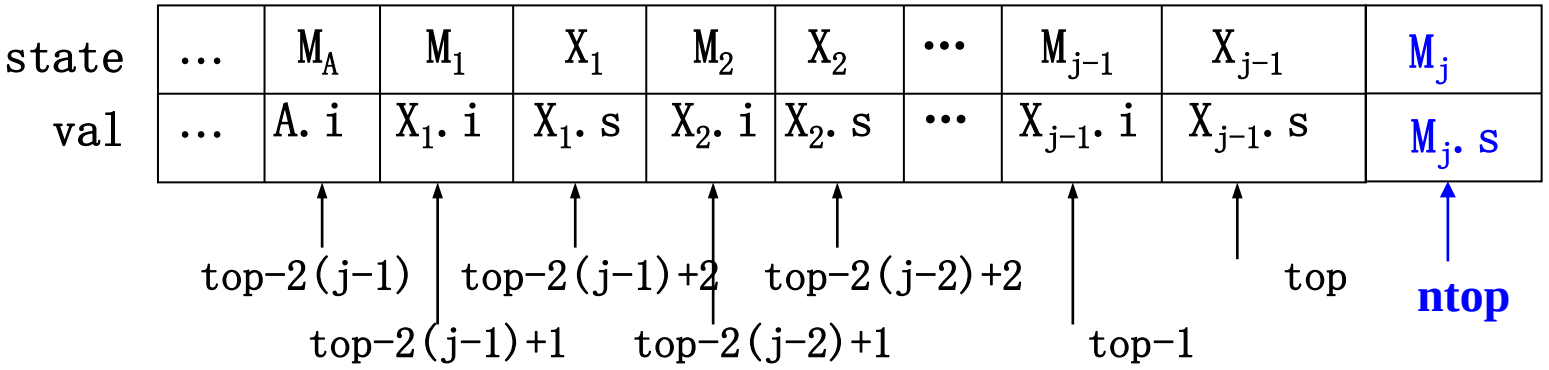
第一种情况：用 $M_j \to \varepsilon$ 进行归约

已知：

每个标记非终结符号在文法中是唯一的

M_j 属于哪个形式为 $A \to M_1X_1M_2X_2 \ldots M_nX_n$ 的产生式

计算属性 $X_j.i$ 需要哪些属性、以及它们的位置



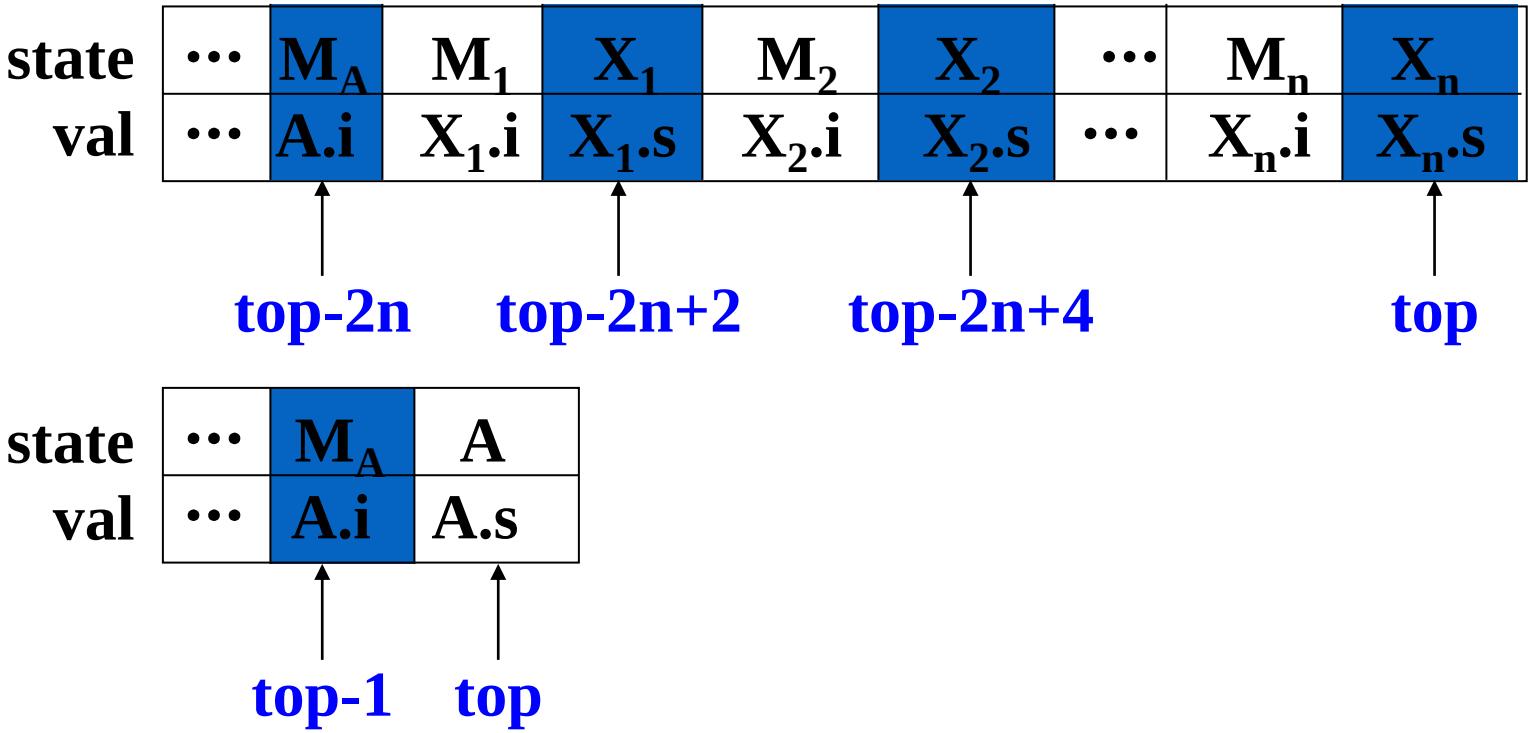
算法5.3：L属性定义的自底向上分析和翻译（续）

第二种情况：用 $A \rightarrow M_1X_1M_2X_2...M_nX_n$ 进行归约

已知：

A.i的值、及其位置

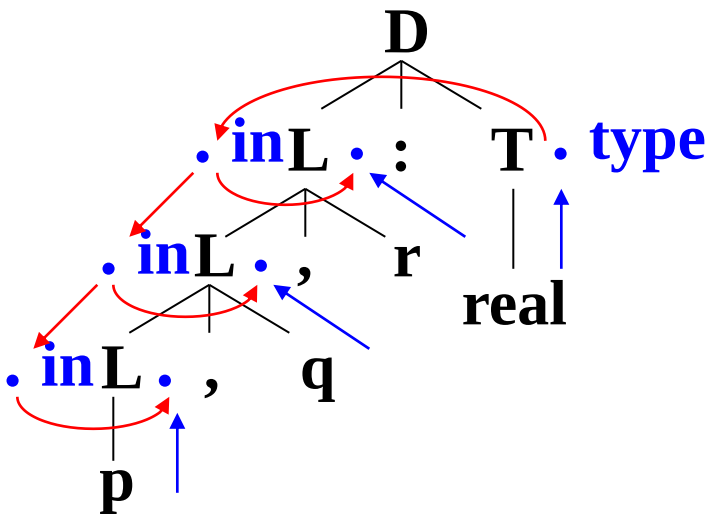
计算 A.s 所需要的属性值均已在栈中已知的位置即各有关 X_j 的位置上



5.4.4 改写语法制导定义为S属性定义

例：PASCAL的变量声明语句可由如下文法产生：

D→L:T
T→integer | real
L→L, id | id



问题：

标识符由L产生，而类型不在L的子树中
归约从左向右进行，类型信息从右向左传递
只用综合属性不能使类型和标识符联系在一起

5.4.4 改写语法制导定义为S属性定义

改写文法，使类型作为标识符表的最后一个元素

D→idL

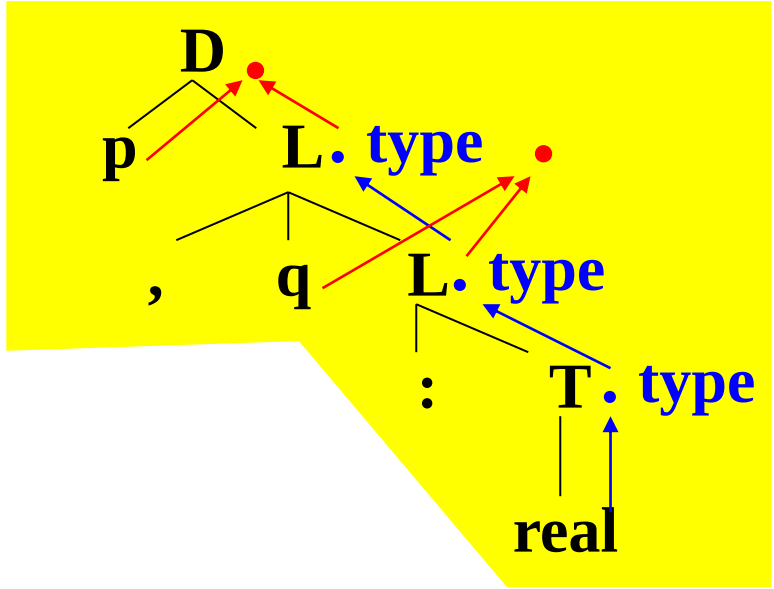
L→,idL | :T

T→integer | real

改写后：

归约从右向左进行，类型信息从右向左传递

仅用综合属性即可把类型信息和标识符联系起来



D→idL	addtype(id.entry, L.type)
L→,idL_1	L.type=L_1.type; addtype(id.entry, L_1.type)
L→:T	L.type=T.type
T→integer	T.type=integer
T→real	T.type=real

5.5 通用的语法制导翻译方法

按照一般的语法制导翻译步骤完成翻译。

输入符号串 ——→ 分析树 ——→ 依赖图 ——→ 语义规则的计算顺序 ——→ 计算结果

根据语法制导定义，为每个非终结符号构造一个翻译函数。
分析树的结点编号作为翻译函数的一个形参，函数根据在该结点处所用的产生式及其语义规则定义的属性之间的依赖关系以适当的顺序访问其诸子结点。
在遍历分析树的过程中，调用相应的函数来计算属性值，从而实现对非L属性定义的翻译。

算法5.4 根据语法制导定义构造语法制导翻译程序

输入：语法制导定义

输出：语法制导翻译程序

方法：

为每一个非终结符号A建立一个函数，该函数可以是递归的。

(1) 设计函数头：

分析树结点作为函数的形参，A的每一个继承属性对应函数的一个形参，A的综合属性作为函数的返回值。

为A产生式中的每个文法符号的每一个属性都声明一个相应的局部变量。

(2) 函数体结构：

如果非终结符号A有多个候选式，则A的函数体首先要根据当前结点处使用的产生式来确定应执行的分支代码，即A的函数代码可由多个分支组成。

算法5.4 根据语法制导定义构造语法制导翻译程序(续)

(3) 设计分支代码：

依据语法制导定义中与A的每个候选产生式相关的语义规则来设计相应的分支程序代码

根据属性之间的依赖关系确定访问子结点的顺序

子结点可以是内部结点、或者叶子结点。

① 若子结点是叶子结点，并且对应的记号X有综合属性x，则把它的值保存于为属性X.x声明的变量中。

② 若子结点是内部结点，且对应于非终结符号B

如果B有继承属性B.i，则先根据语义规则生成计算B.i值的代码，即将语义规则中出现的属性替换为相应的变量；

然后，产生一个函数调用语句 $c=B(n, b_1, b_2, \ldots, b_k)$，其中n是B对应的分析树结点，$b_i(i=1, 2, \ldots, k)$是对应于B的继承属性的变量，c是对应于B的综合属性的变量。

算法5.4 根据语法制导定义构造语法制导翻译程序(续)

示例：为表5-2中的语法制导定义构造语法制导翻译程序

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)

函数：

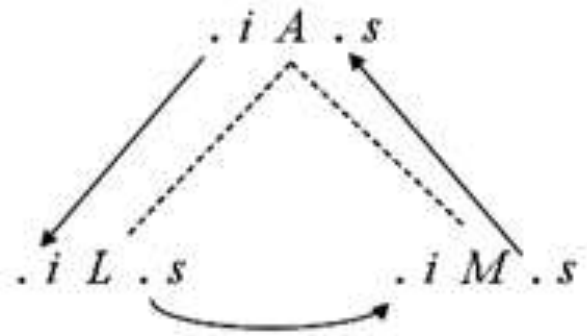
*fx*A(n, ai)

*fx*L(n, li)

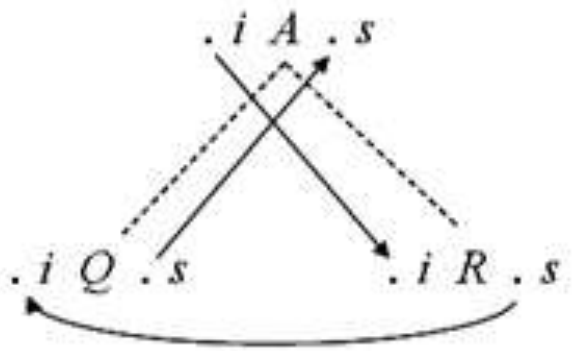
*fx*M(n, mi)

*fx*Q(n, qi)

*fx*R(n, ri)



(a) 产生式$A \rightarrow LM$的依赖图



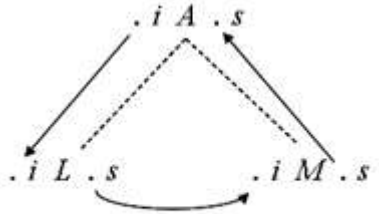
(b) 产生式$A \rightarrow QR$的依赖图

翻译函数 *fx*A(n, ai)示意

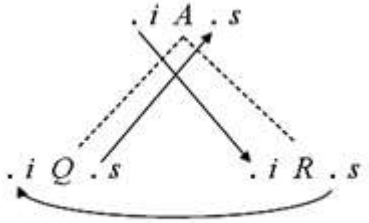
```
float fxA(n, ai) {
    float as, li, ls, mi, ms, qi, qs, ri, rs;
    switch ( 在结点n处使用的产生式 )
    {
    case  'A→LM':
        li=l(ai);
        ls=fxL(child(n, 1),  li);
        mi=m(ls);
        ms=fxM(child(n, 2), mi);
        as=f(ms);
        return  as;
```

```
case  'A→QR':
        ri=r(ai);
        rs=fxR(child(n, 2), ri);
        qi=q(rs);
        qs=fxQ(child(n, 1), qi);
        as=f(qs);
        return  as;
    default:
        error();
    }
}
```

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)



(a) 产生式$A \to LM$的依赖图



(b) 产生式$A \to QR$的依赖图

课后作业 1

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。

课后作业 2

2、假定声明由下面的文法产生：

D→idL

L→,idL|:T

T→integer|real

（1）试设计一个翻译方案，它把每一个标识符的类型信息加入到符号表中；

（2）根据（1）的翻译方案构造一个预测翻译程序。

课后作业 3

有如下文法：

S→(L)|a

L→L,S|S

（1）设计一个语法制导定义，它输出配对的括号数；

（2）构造一个语法翻译方案，它输出每个a的嵌套深度。例如对句子（a,(a,a))的输出结果是1，2，2，