编译原理与技术

第5章：语法制导翻译技术

王吴凡

北京邮电大学计算机学院

主页：cswwf.github.io

邮箱：wufanwang@bupt.edu.cn

教学内容、目标与要求

- 教学内容
  - 语法制导定义和翻译方案；
  - S-属性定义的自底向上实现；
  - L-属性定义的自底向上实现。
- 教学目标与要求
  - 理解综合属性和继承属性；
  - 掌握语法制导定义/翻译方案的设计思路和方法；
  - 能够根据翻译目标和需求，设计所需属性、语义规则，基于上下文无关文法设计相应的语法制导定义或语法制导翻译方案。
  - 理解LR分析翻译程序的实现方法；
    掌握S属性定义和L属性定义的LR实现。

内容目录

5.1 语法制导翻译概述

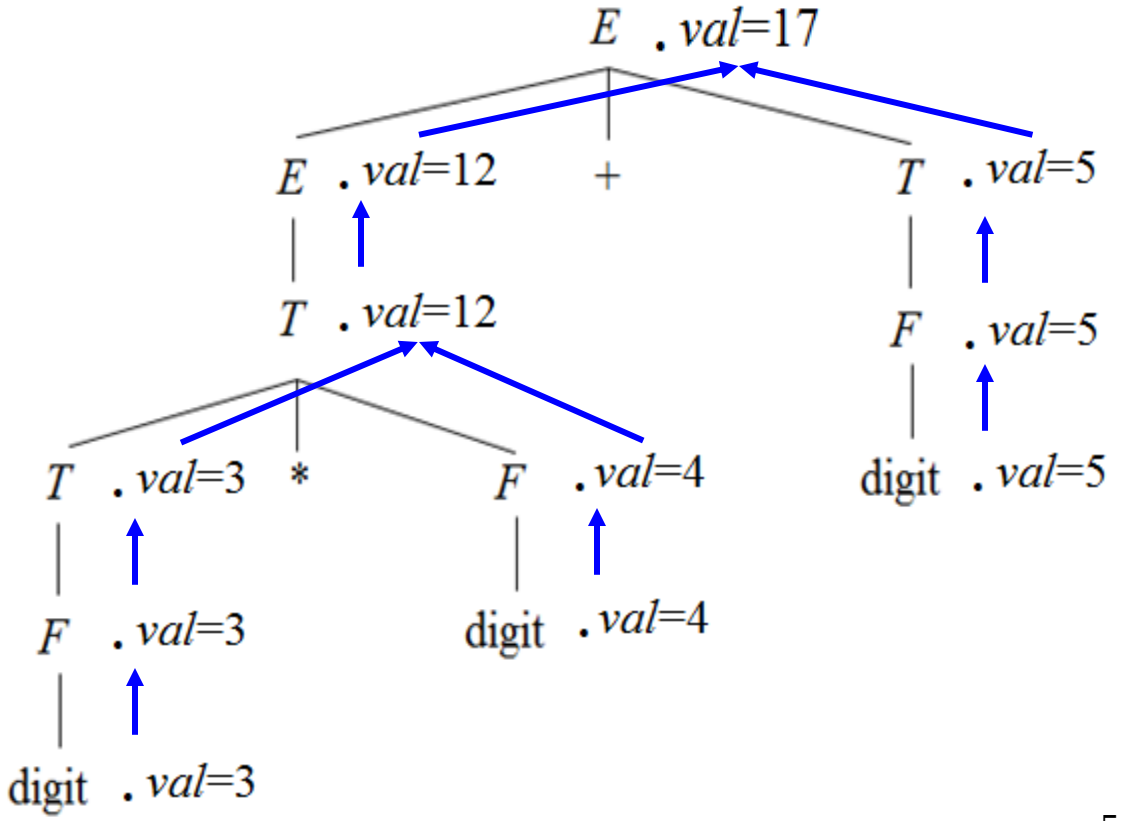
- 语法制导翻译技术
  - 编译程序普遍采用的一种技术
  - 比较接近形式化
- 语法制导定义：
  - 首先，根据翻译目标来确定每个产生式的语义；
  - 其次，根据产生式的含义，分析每个符号的语义；
  - 再次，把这些语义以属性的形式附加到相应的文法符号上（即把语义和语言结构联系起来）；
  - 最后，根据产生式的语义给出符号属性的求值规则（即语义规则），从而形成语法制导定义。
- 翻译：
  - 根据语法分析过程中所使用的产生式，执行与之相应的语义规则，完成符号属性值的计算，从而完成翻译。

语法制导翻译示例 1

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

例如：考虑算术表达式文法

- 翻译目标：计算表达式的值
- 分析确定每个产生式的语义；
  - $E \rightarrow E_1+T$：两个子表达式的值相加得到表达式的值
  - $F \rightarrow digit$：表达式的值即数字的值
- 分析确定每个符号的语义；
  - E、T、F、digit、+、*、（、）
- 确定文法符号的属性；
  - E.val、T.val、F.val、digit.val
- 设计语义规则，形成语法制导定义。
  - $E \rightarrow E_1+T$：$E.val=E_1.val+T.val$
  - 语法制导定义：产生式 语义规则
- 翻译表达式 3*4+5：

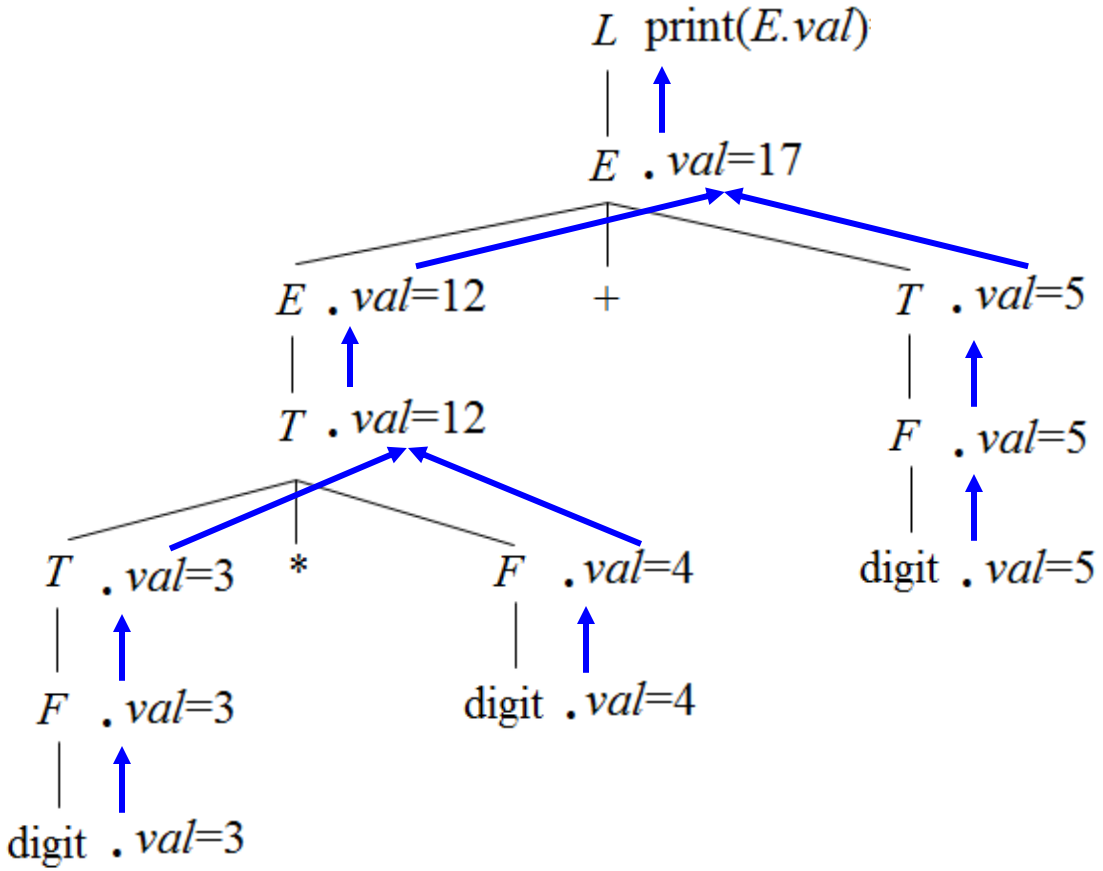


语法制导翻译示例 2

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
- 拓广文法：
  - 增加 L→E
  - Print(E.val)
- 例如：3*4+5
  - 分析树：

L→E	Print(E.val)
$E→E_1+T$	$E.val=E_1.val+T.val$
E→T	E.val=T.val
$T→T_1*F$	$T.val=T_1.val*F.val$
T→F	T.val=F.val
F→(E)	F.val=E.val
F→digit	F.val=digit.val



翻译目标决定语义规则　　翻译结果依赖于语义规则

- 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　　语法制导翻译一般步骤

- 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。

- 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。

输入符号串
① ⟶ 分析树
② ⟶ 依赖图
③ ⟶ 语义规则的计算顺序
④ ⟶ 计算结果

5.2 语法制导定义及翻译方案

1. 语法制导定义
2. 依赖图
3. 计算次序
4. S属性定义和L属性定义
5. 翻译方案

1. 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。

语法制导定义

- 对上下文无关文法的推广
- 每个文法符号都可以有一个属性集，可包括综合属性和继承属性。
  - 产生式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。
  - 产生式右部某文法符号的继承属性是从其所在产生式的左部符号和/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父结点和/或兄弟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为了完成一个特定的功能（而非计算一个具体的属性值），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。
  - 比如：print(E.val)
  - 看成是相应产生式左部符号的虚拟综合属性。

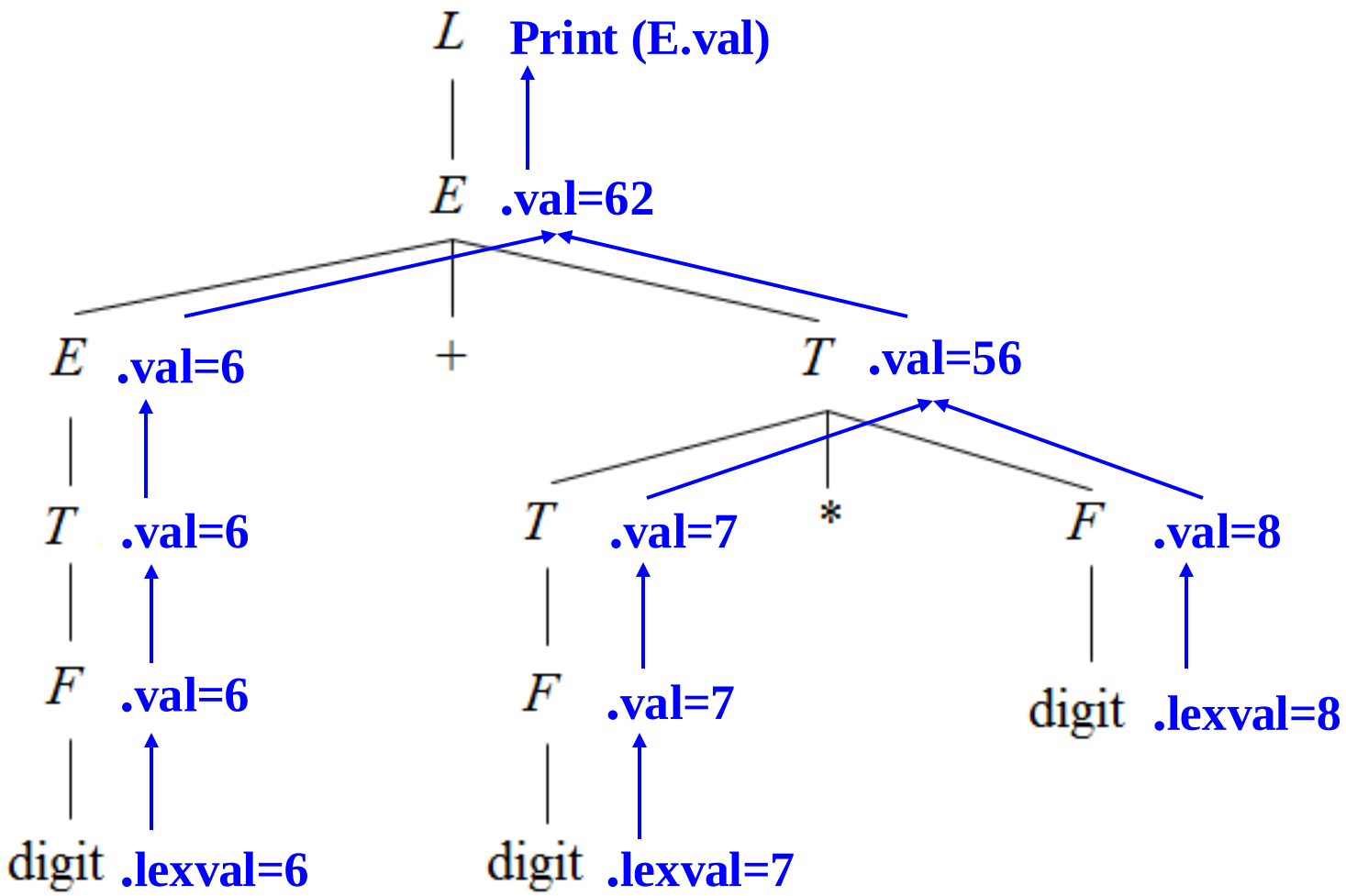
简单算术表达式求值的语法制导定义

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

- E、T和F：有综合属性 val
  - 表示相应子表达式的值
- L→E的语义规则：
  - 过程调用
  - 打印出表达式的值
  - L的虚拟综合属性

综合属性　　6+7*8的分析树加注释的过程

- 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。
- 简单计算器的语法制导定义是S-属性定义



- 属性值的计算可以在语法分析过程中进行。

继承属性　　用继承属性L.in传递类型信息

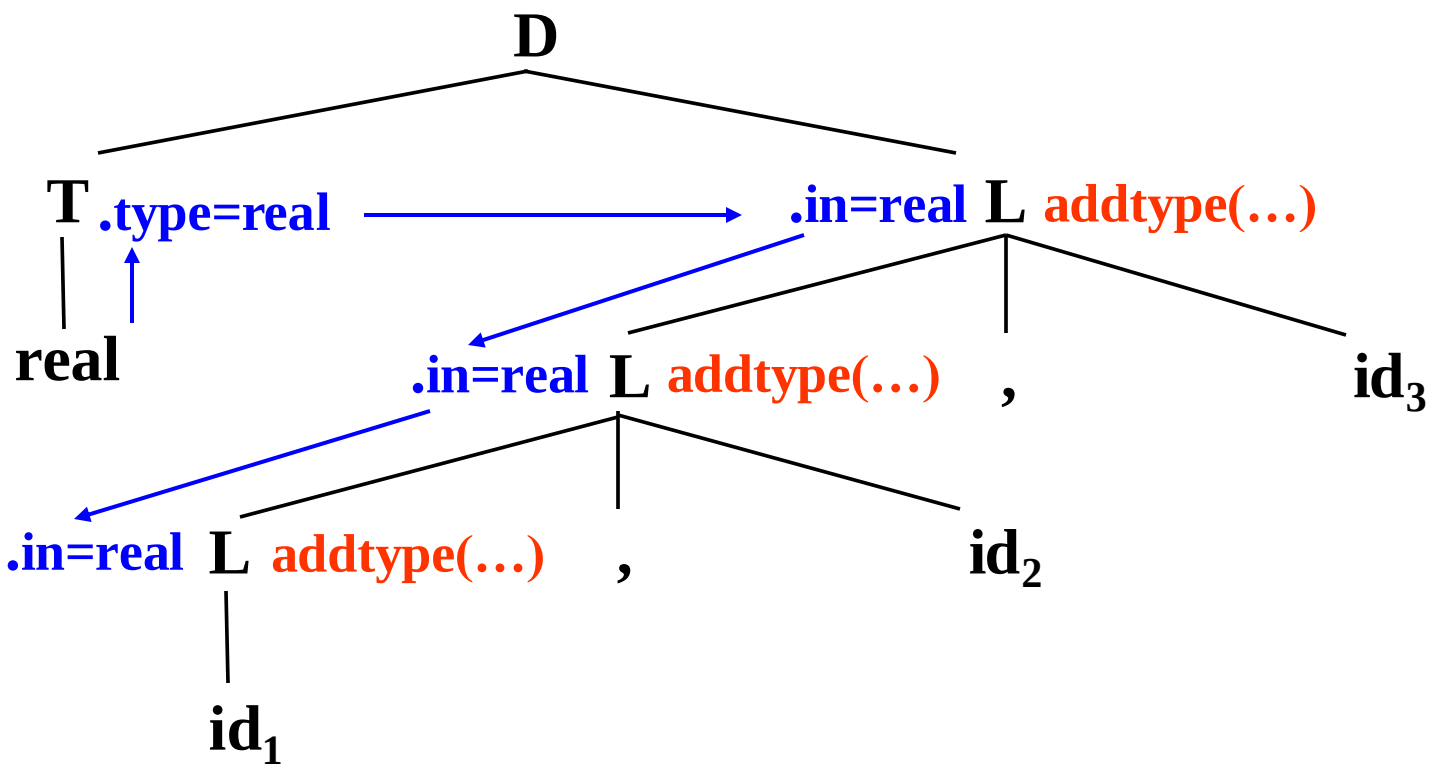
- 分析树中，一个结点的继承属性值由该结点的父结点和/或它的兄弟结点的属性值决定。
- 可用继承属性表示程序设计语言结构中上下文之间的依赖关系
  - 可以跟踪一个标识符的类型
  - 可以跟踪一个标识符，了解它是出现在赋值号的右边还是左边，以确定是需要该标识符的值还是地址。

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)

- 句子：int a, b　　real x, y
- T.type：由类型关键字确定
- L.in：
  - D→TL　　L.in继承T的类型信息
  - $L \rightarrow L_1$, id　L_1.in继承L的类型信息

语句 real id_1, id_2, id_3 的注释分析树

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)

把类型信息在分析树中向下传递

把类型信息填入符号表中标识符记录中

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
  - 把语义规则统一为如下形式 $b=f(c_1,c_2,\ldots,c_k)$
- 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。

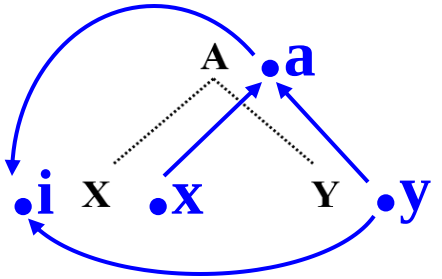
算法5.1 构造依赖图

输入：一棵分析树，语法制导定义　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;
```

产生式	语义规则
A→XY	A.a=*f*(X.x, Y.y)
	X.i=g(A.a, Y.y)

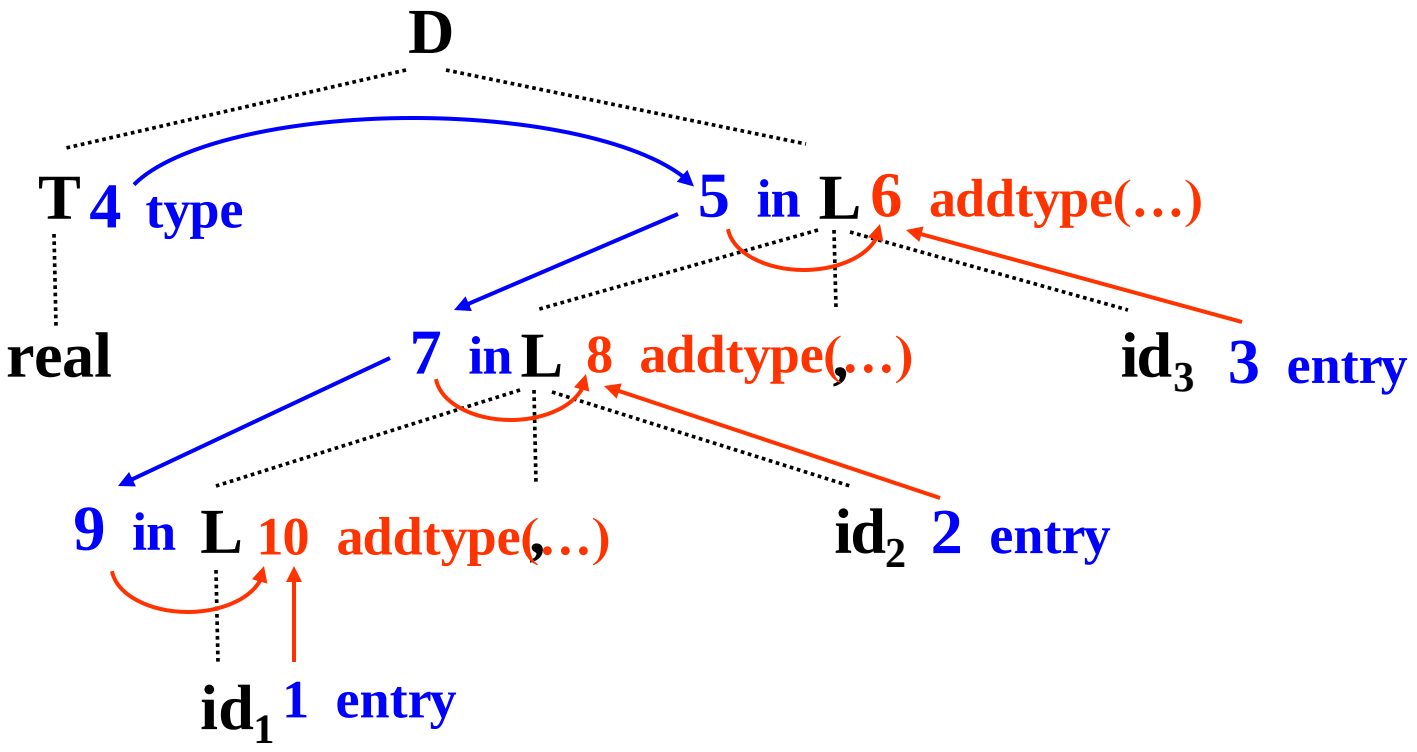


依赖图构造举例

real p, q, r

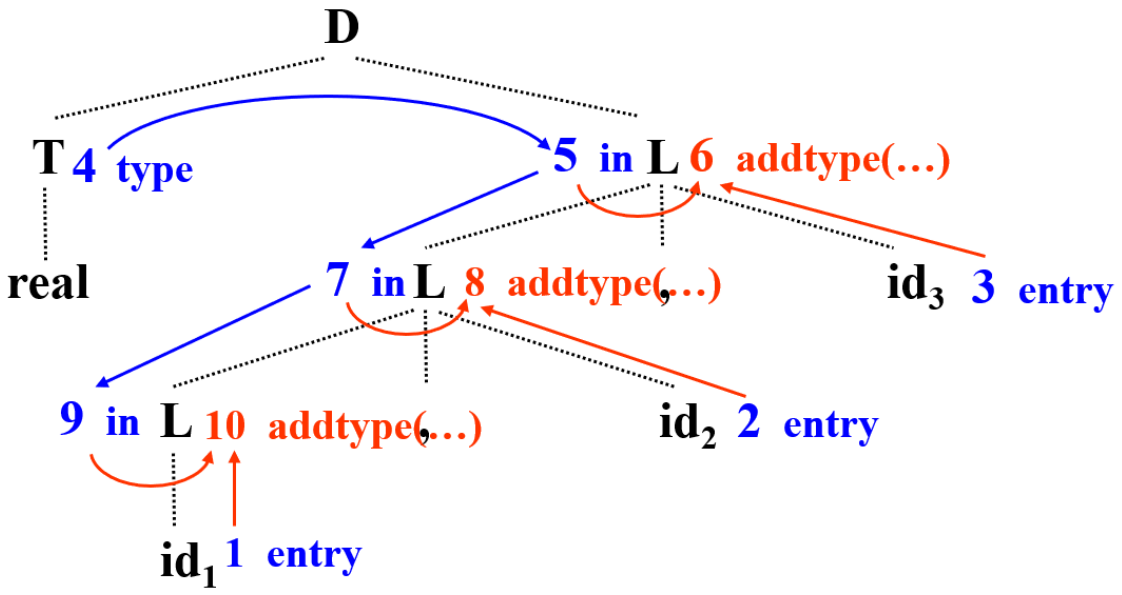
real id_1, id_2, id_3

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)



3. 计算次序

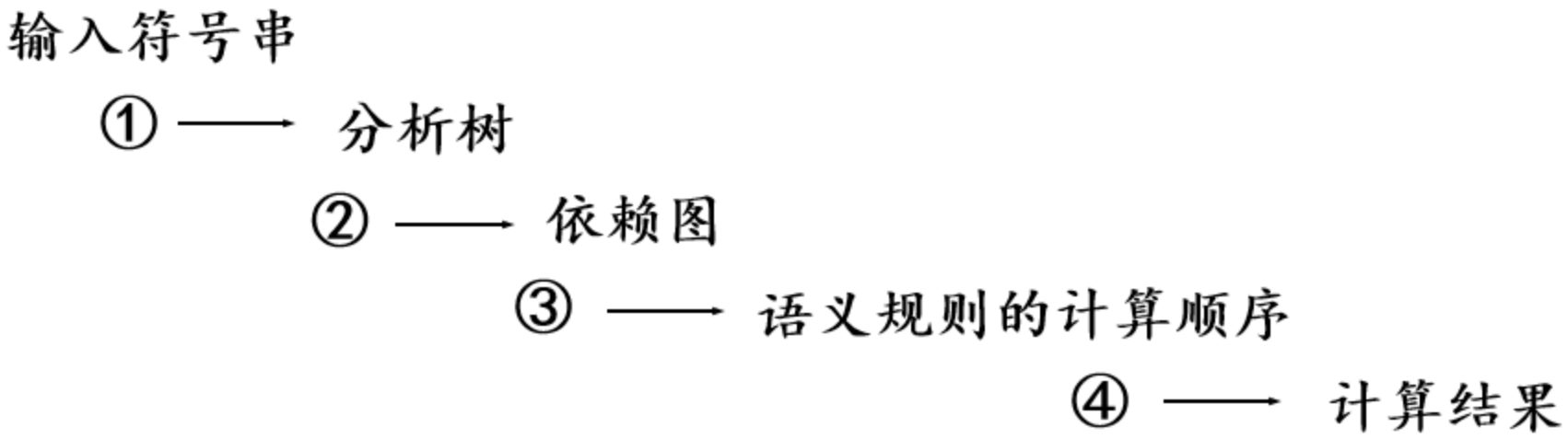
- 有向非循环图的拓扑排序
  - 图中结点的一种排序 $m_1, m_2, \ldots, m_k$
  - 有向边只能从这个序列中前边的结点指向后面的结点
  - 如果 $m_i \rightarrow m_j$ 是从 m_i 指向 m_j 的一条边，那么在序列中 m_i 必须出现在 m_j 之前。
- 依赖图的任何拓扑排序
  - 给出了分析树中结点的语义规则计算的有效顺序
  - 在拓扑排序中，一个结点上语义规则 $b=f(c_1, c_2, \ldots, c_k)$ 中的属性 $c_1, c_2, \ldots, c_k$ 在计算b时都是可用的。
- 拓扑排序：
  - 1、2、3、4、5、6、7、8、9、10
  - 4、5、3、6、7、2、8、9、1、10



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

语法制导翻译过程

- 最基本的文法用于建立输入符号串的分析树；
- 为分析树构造依赖图；
- 对依赖图进行拓扑排序；
- 从这个序列得到语义规则的计算顺序；
- 照此计算顺序进行求值，得到对输入符号串的翻译。



示例：翻译 3*4+5

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

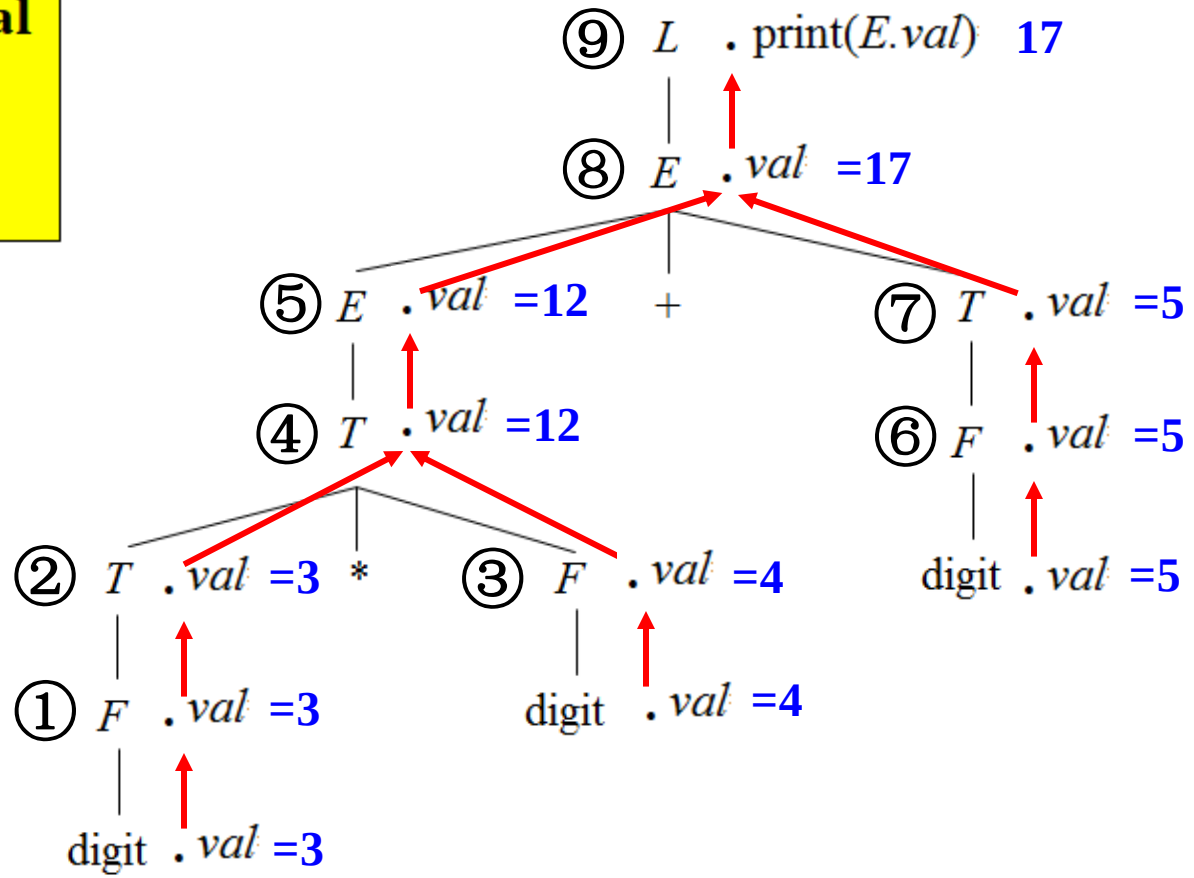
输入符号串

① ——→ 分析树

② ——→ 依赖图

③ ——→ 语义规则的计算顺序

④ ——→ 计算结果



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（$1 \le j \le n$）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

S属性定义

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

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

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)

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;
}.
```

- 实参：分析树的根结点
- 遍历顺序：深度优先
- 属性计算：
  - 进入结点前，计算其继承属性
  - 从结点返回时，计算其综合属性
- L属性定义的属性都可以按深度优先的顺序计算。

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

翻译方案示例

翻译方案：

$E \rightarrow TR$

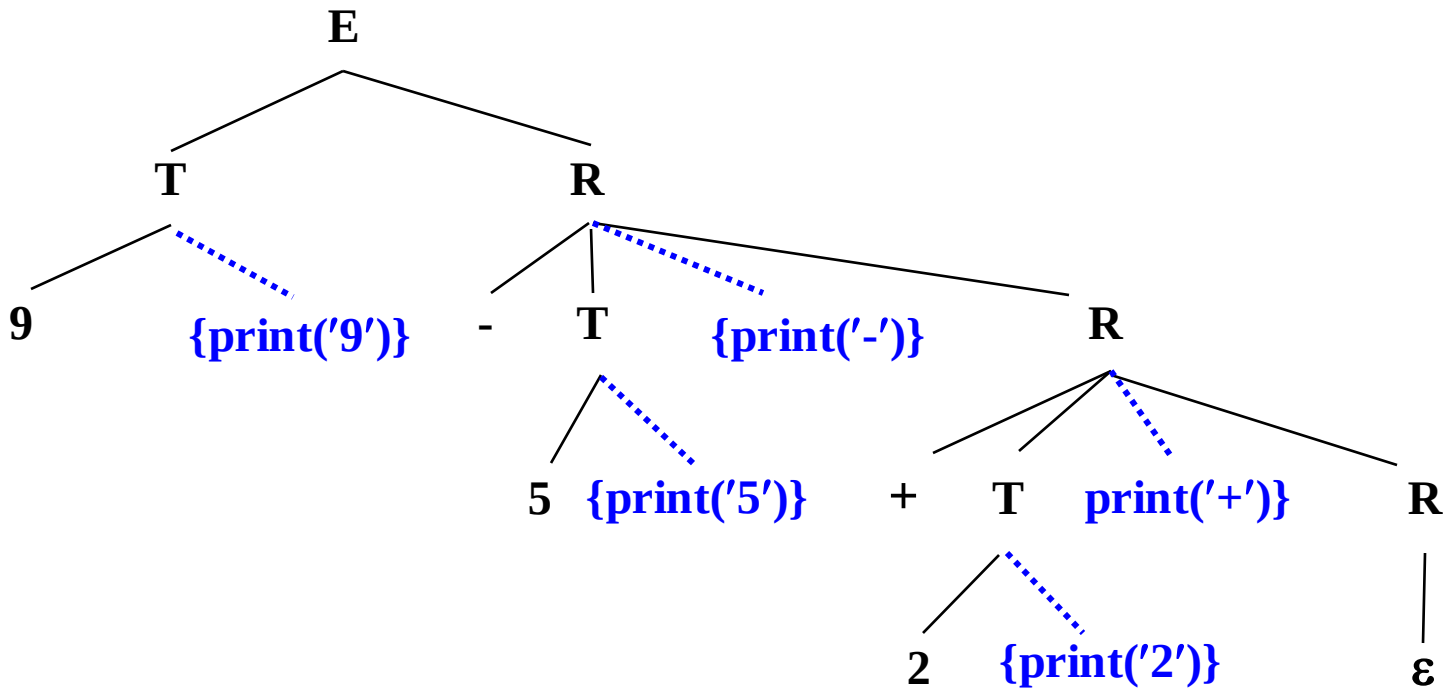
$R \rightarrow +T\ \{\ print('+')\ \}\ R_1$

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

$\quad | \varepsilon$

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

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：

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

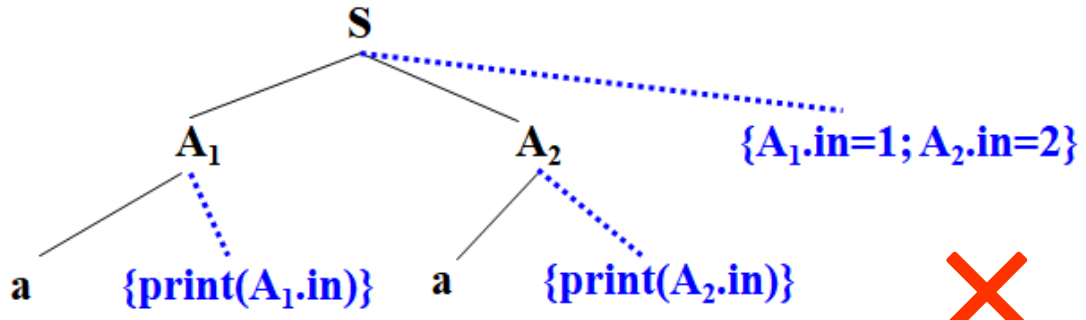
- 对于L属性定义：
  - 动作不能引用其右边文法符号的综合属性。
  - 左部符号的综合属性：
    - 只有在它所引用的所有属性都计算出来之后才能计算
    - 放在产生式右端末尾
  - 右部某文法符号的继承属性：
    - 必须在这个符号以前的动作中计算出来
    - 放在该文法符号前面

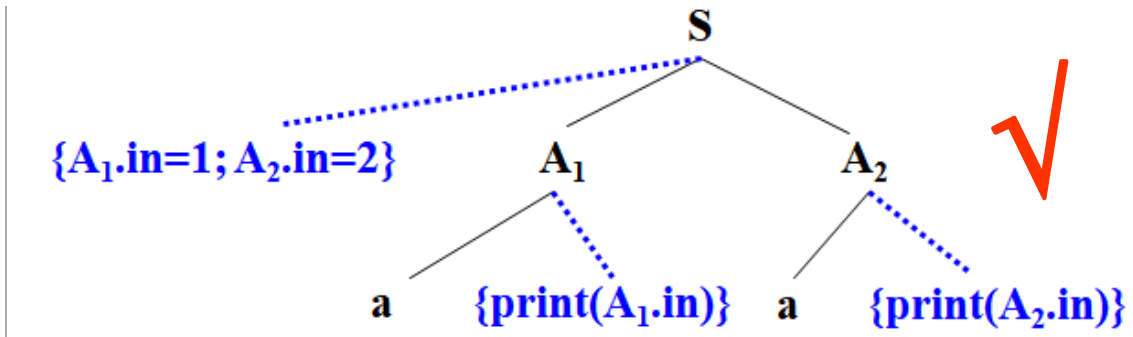
翻译方案设计示例

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) }





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 }

$L \rightarrow$ { L_1.in=L.in } L_1,id { addtype(id.entry, L.in) }

L→id { addtype(id.entry, L.in) }

例：为如下L属性定义设计翻译方案

产生式	语义规则
$S \rightarrow B$	B.ps=10;　S.ht=B.ht
$B \rightarrow B_1 B_2$	B_1.ps=B.ps B_2.ps=B.ps B.ht=max(B_1.ht, B_2.ht)
$B \rightarrow B_1 sub B_2$	B_1.ps=B.ps B_2.ps=shrink(B.ps) B.ht=disp(B_1.ht, B_2.ht)
$B \rightarrow text$	B.ht=text.h×B.ps

- 唯一的继承属性：B.ps
  - 常数
  - 依赖于左部B的B.ps
- 翻译方案如下：

作业：
5.16
5.17

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

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

$B \rightarrow$ {B_1.ps=B.ps} B_1 sub {B_2.ps=shrink(B.ps)} B_2 {B.ht=disp(B_1.ht, B_2.ht)}

$B \rightarrow$ text {B.ht=text.h×B.ps}

5.3 S-属性定义的自底向上翻译

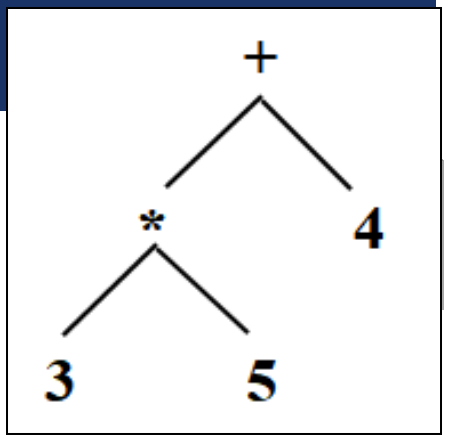
- S属性定义：
  - 只用综合属性的语法制导定义

1. 为表达式构造语法树的语法制导定义
2. S属性定义的自底向上实现

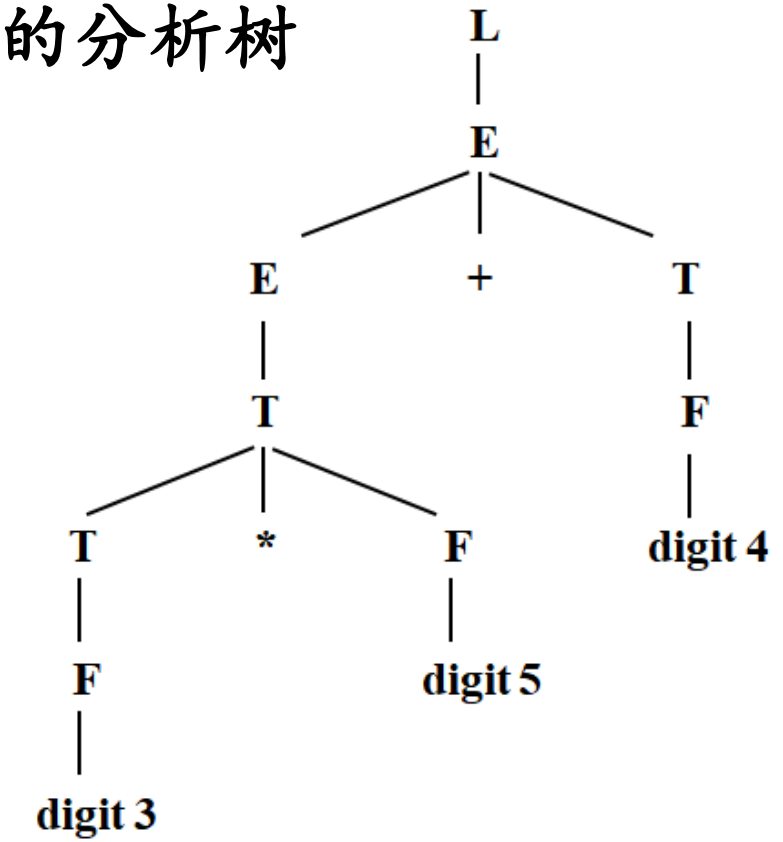
1. 为表达式构造语法树的语法制导定义

- 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)
- 语法树：
  - 分析树的抽象（或压缩）形式。
  - 内部结点表示运算符号，其子结点表示它的运算分量。
  - 也称为语法结构树或结构树。

- 3*5+4 的语法树



- 3*5+4 的分析树

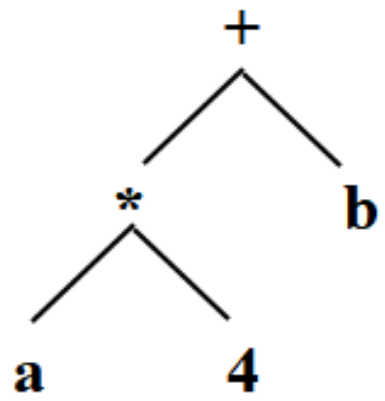


表达式的语法树　　构造函数

- 语法树的形式
  - 结点：运算符号、或运算分量
  - 运算符号结点的子结点：与该运算符的各个运算分量相应的子树的根。
  - 每个结点可包含若干个域：
    - 标识域、指针域、属性值域等
- 运算符结点中的域
  - 标识域：运算符号
  - 指针域：指向与各运算分量相应的结点
  - 称运算符号为该结点的标号

- **makenode (op, left, right)**
  - 建立一个运算符号结点，标号是 op；
  - 域left和right：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：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：保存该数的值。

建立表达式 a*4+b 的语法树



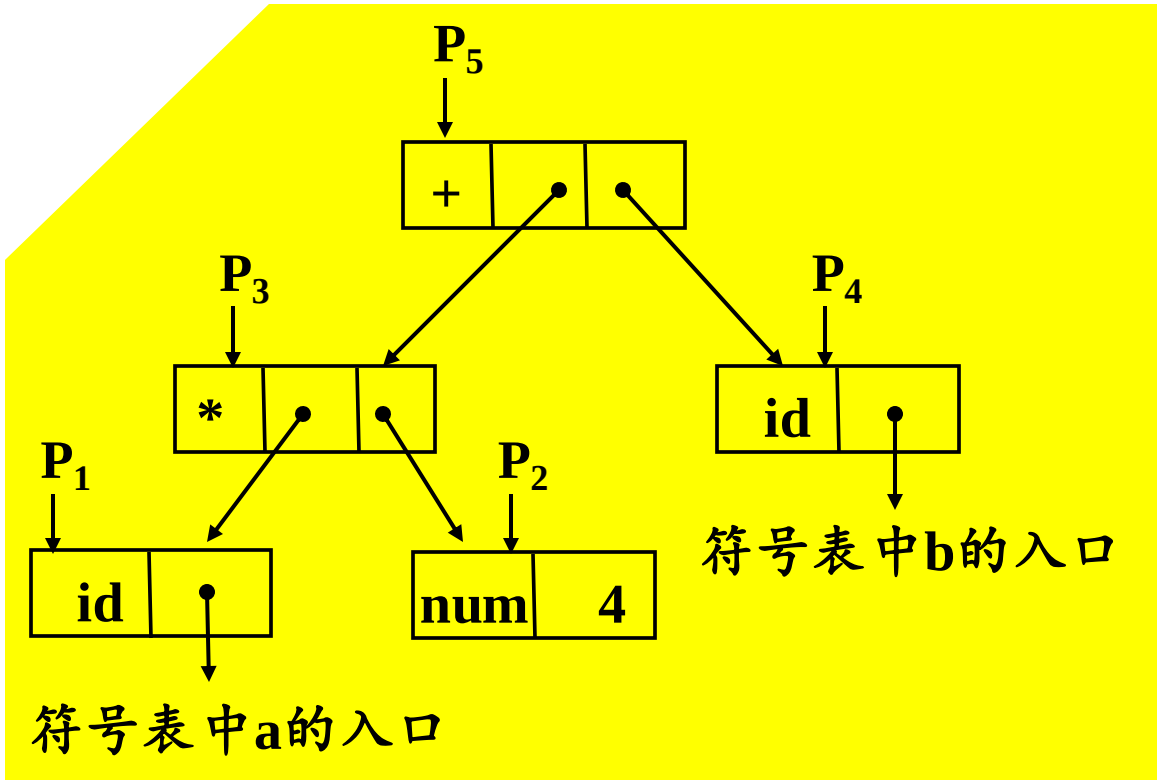
p_1=makeleaf(id, entrya);

p_2=makeleaf(num, 4);

p_3=makenode(‘*’, p_1, p_2);

p_4=makeleaf(id, entryb);

p_5=makenode(‘+’ ,p_3, p_4);



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)

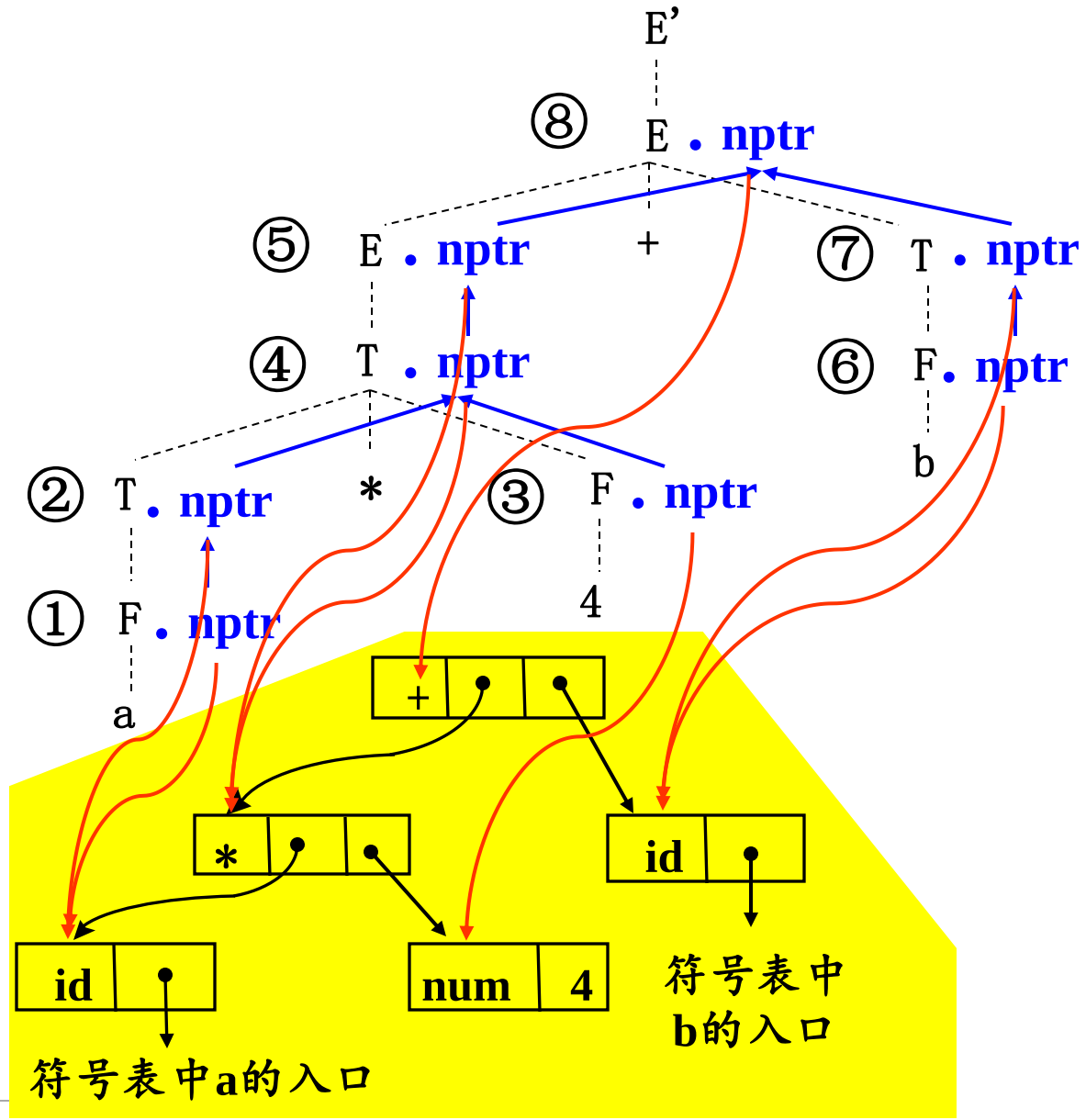
构造表达式语法树的语法制导定义

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)

- 综合属性 nptr
  - 记录在构造过程中建立的子树
  - 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。

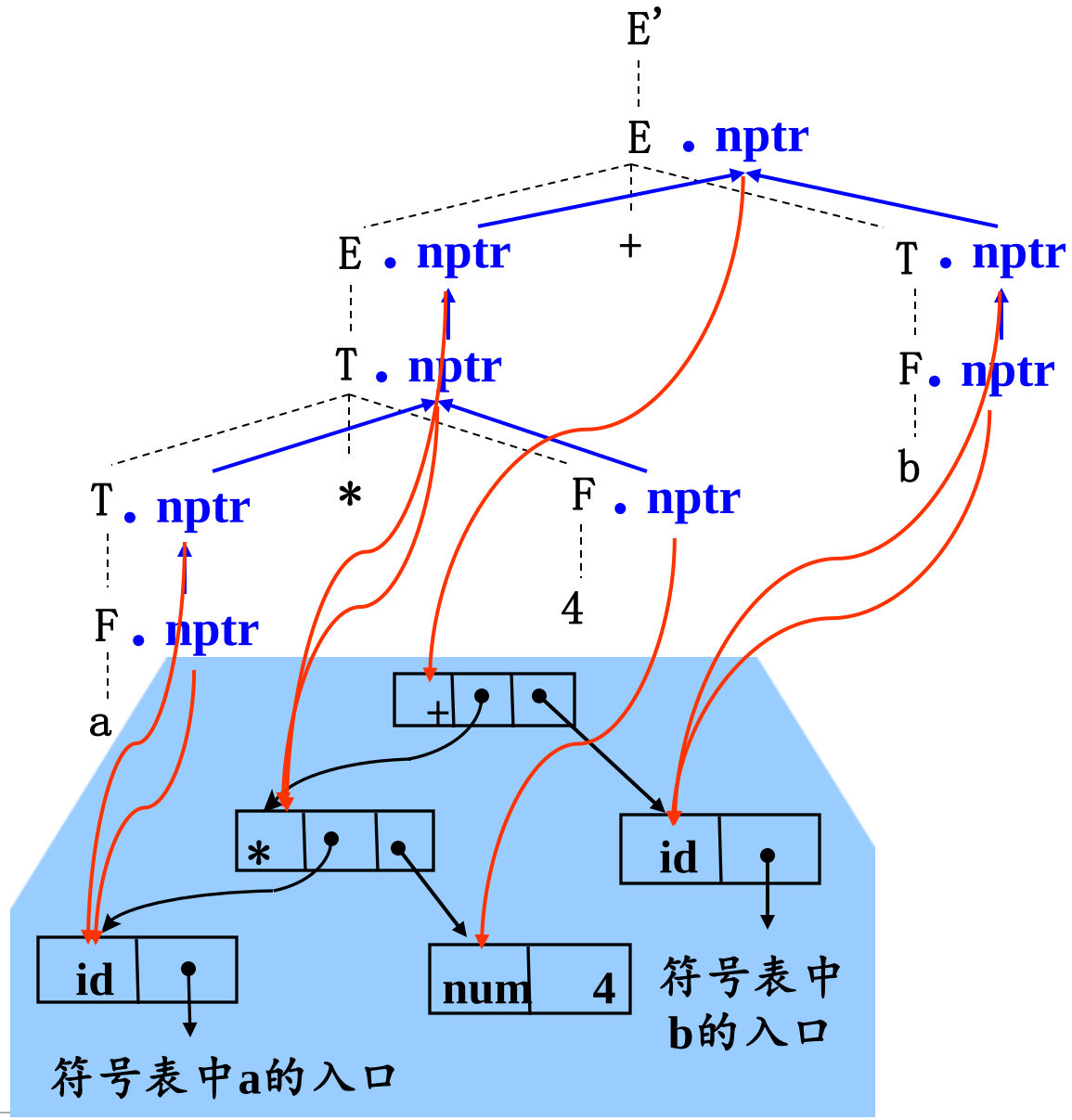
表达式a*4+b的语法树的构造（先分析再翻译）

E'→ E
E→E_1+T
E→T
T→T_1*F
T→F
F→(E)
F→id
F→num



① F.nptr=makeleaf(id, entrya)
② T.nptr=F.nptr
③ F.nptr=makeleaf(num, 4)
④ T.nptr=makenode('*', T_1.nptr, F.nptr)
⑤ E.nptr=T.nptr
⑥ F.nptr=makeleaf(id, entryb)
⑦ T.nptr=F.nptr
⑧ E.nptr=makenode('+', E_1.nptr, T.nptr)

表达式a*4+b的语法树的构造（边分析边翻译）

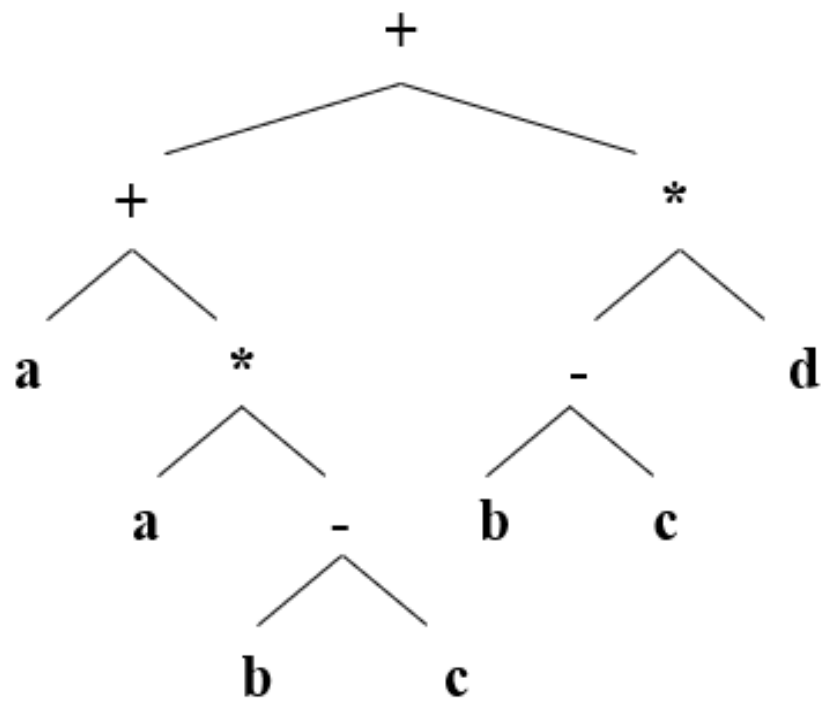


E'→ E
E→E_1+T
E→T
T→T_1*F
T→F
F→(E)
F→id
F→num

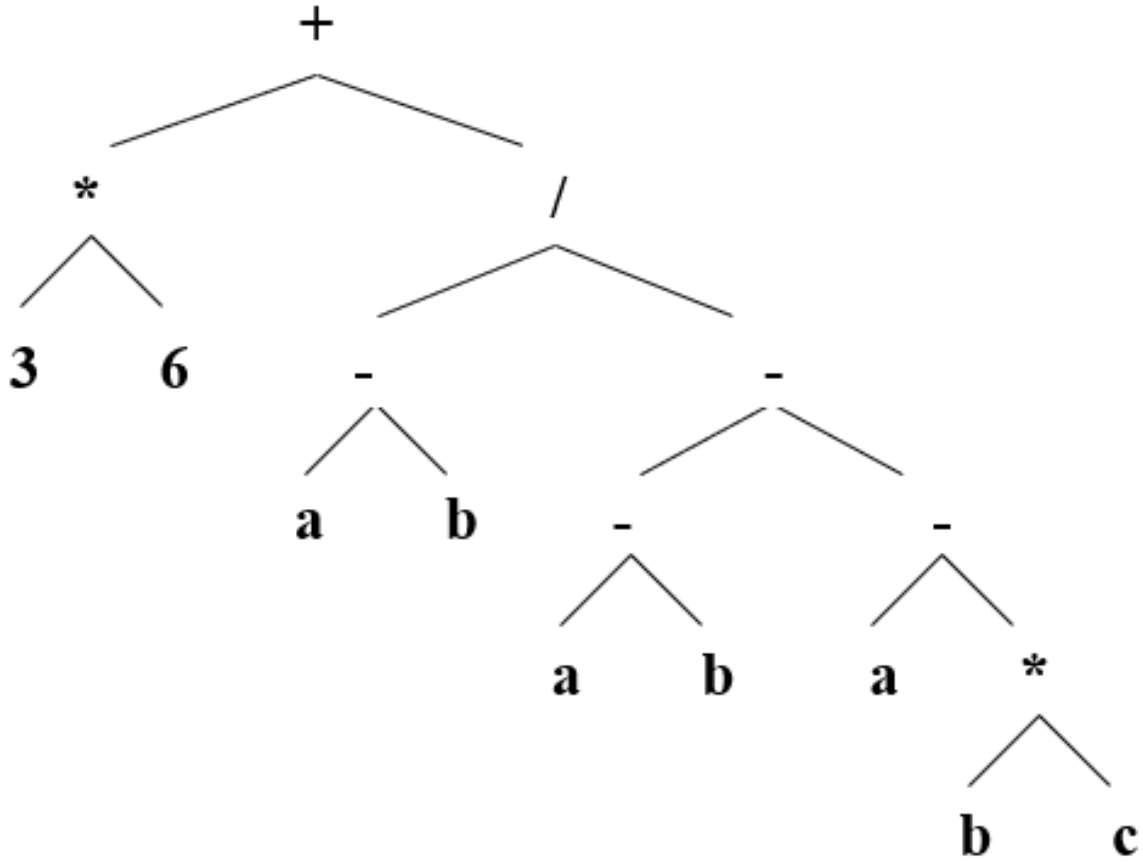
① F.nptr=makeleaf(id, entrya)
② T.nptr=F.nptr
③ F.nptr=makeleaf(num, 4)
④ T.nptr=makenode('*', T_1.nptr, F.nptr)
⑤ E.nptr=T.nptr
⑥ F.nptr=makeleaf(id, entryb)
⑦ T.nptr=F.nptr
⑧ E.nptr=makenode('+', E_1.nptr, T.nptr)

示例：构造表达式的语法树

- a+a*(b-c)+(b-c)*d



- 3*6+(a-b)/(a-b-(a-b*c))



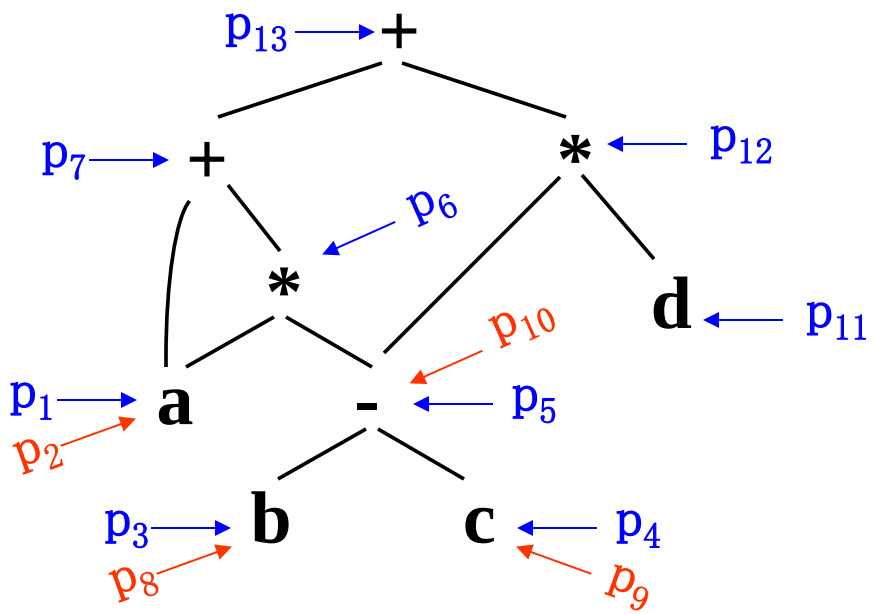
表达式的有向非循环图(dag)

- 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。

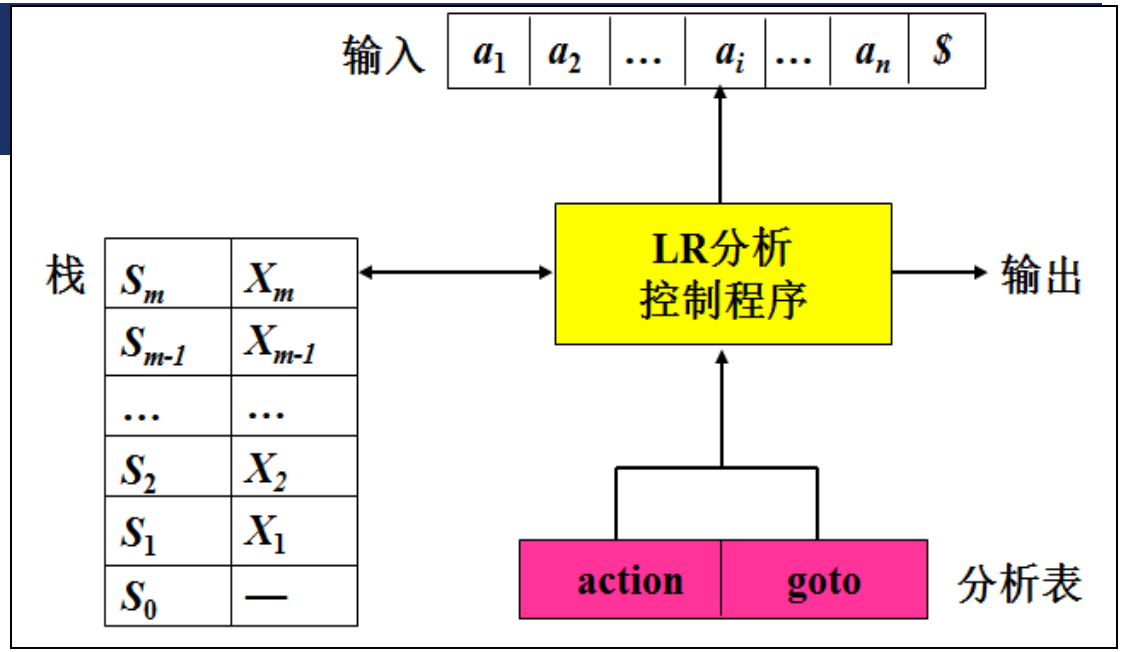
为表达式 a+a*(b-c)+(b-c)*d 构造dag

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});



2. S-属性定义的自底向上实现



- LR分析程序的模型
- 已知
  - LR分析方法中，分析程序使用栈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。

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。

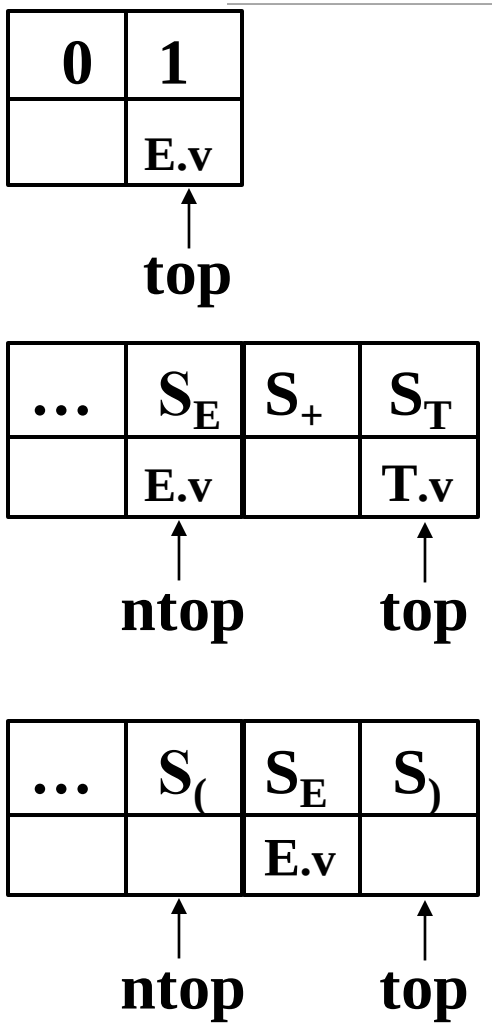
- 栈：一对数组state和val
  - state元素：指向LR(1)分析表中状态行的指针（或索引）
  - 如果state[i]保存对应符号A的状态，则 val[i]中就存放A的属性值。
- 综合属性刚好在每次归约前计算
  - A→XYZ 对应的语义规则是 A.a=*f*(X.x, Y.y, Z.z)

修改分析程序

- LR分析程序中增加计算属性值的代码段
- 对于终结符号
  - 移进时，属性值随状态一起入栈。
  - 综合属性值由词法分析程序产生
- 对产生式A→XYZ
  - 为每个语义规则编写一段代码。
  - 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属性出栈，左部符号的相应状态及属性入栈。

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→E_1+T	E.val=E_1.val+T.val	val[ntop]=val[top-2]+val[top]
E→T	E.val=T.val	
T→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：

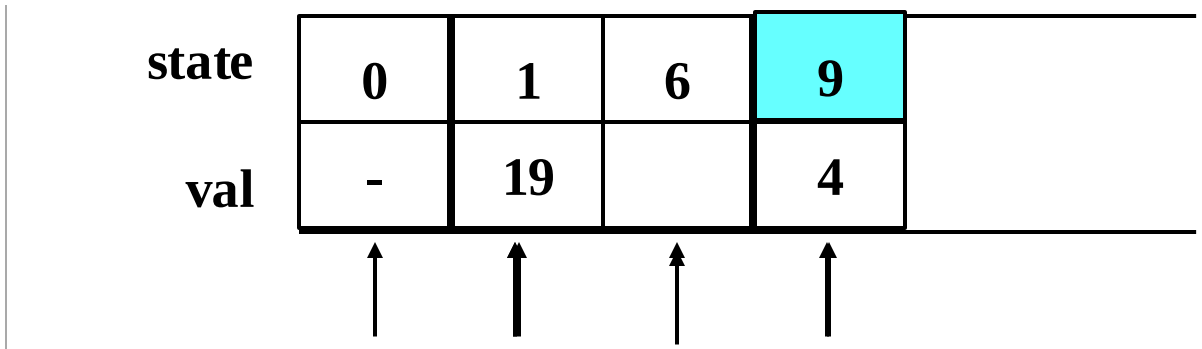
- 当用 A→β 归约时，若|β|=r，在执行相应的代码段之前， ntop=top-r+1。
- 在每一个代码段被执行之后，top=ntop

例：用LR分析程序实现表达式求值

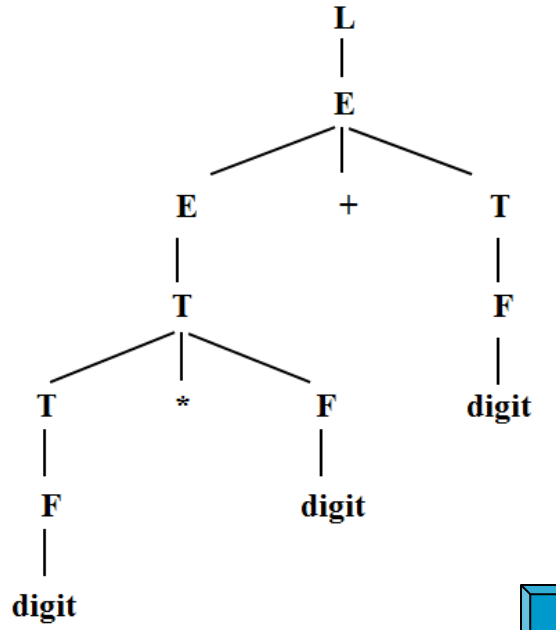
	产生式	语义规则	代码段
	L→E	print(E.val)	print(val[top])
①	$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$	val[ntop]=val[top-2]+val[top]
②	E→T	E.val=T.val	
③	$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$	val[ntop]=val[top-2]*val[top]
④	T→F	T.val=F.val	
⑤	F→(E)	F.val=E.val	val[ntop]=val[top-1]
⑥	F→digit	F.val=digit.lexval	

状态	Action						goto		
	num	+	*	(	)	$	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			

3 * 5 + 4 $



accept!



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

- 方法
  1. 移走翻译方案中嵌入的语义规则
  2. 直接使用分析栈中的继承属性
  3. 变换继承属性的计算规则
  4. 改写语法制导定义为*S*属性定义

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→ε
  - 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→ε的末尾

移除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) }

- 引入：标记非终结符号M和N，及产生式 M→ε 和 N→ε
- 新的翻译方案

E→TR

R→+TMR | -TNR | ε

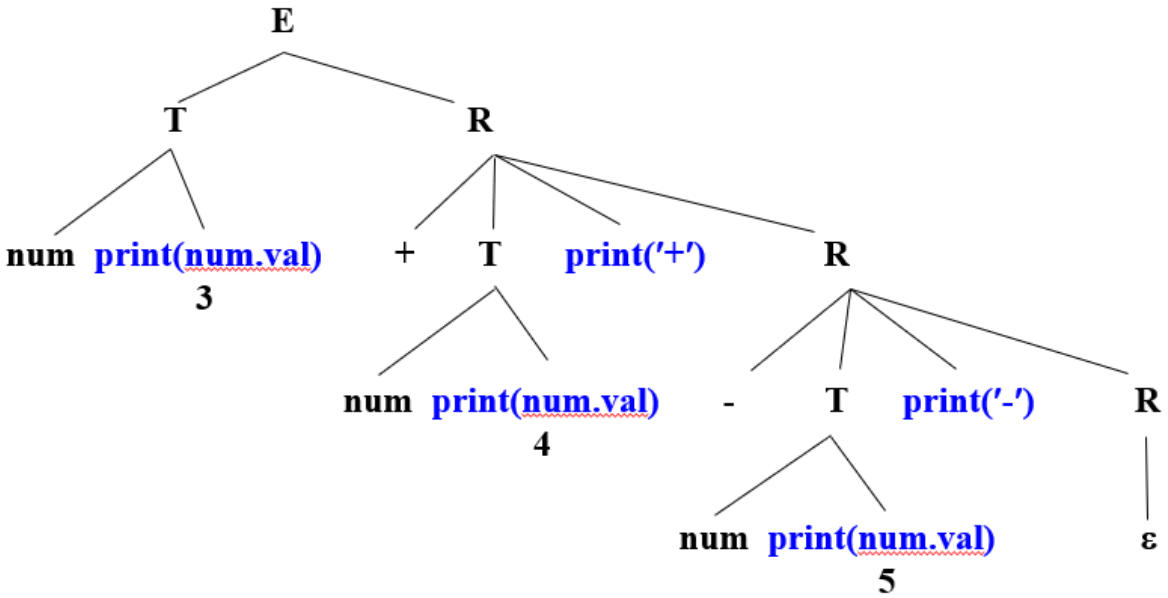
T→num { print(num.val) }

M→ε { print('+') }

N→ε { print('-') }

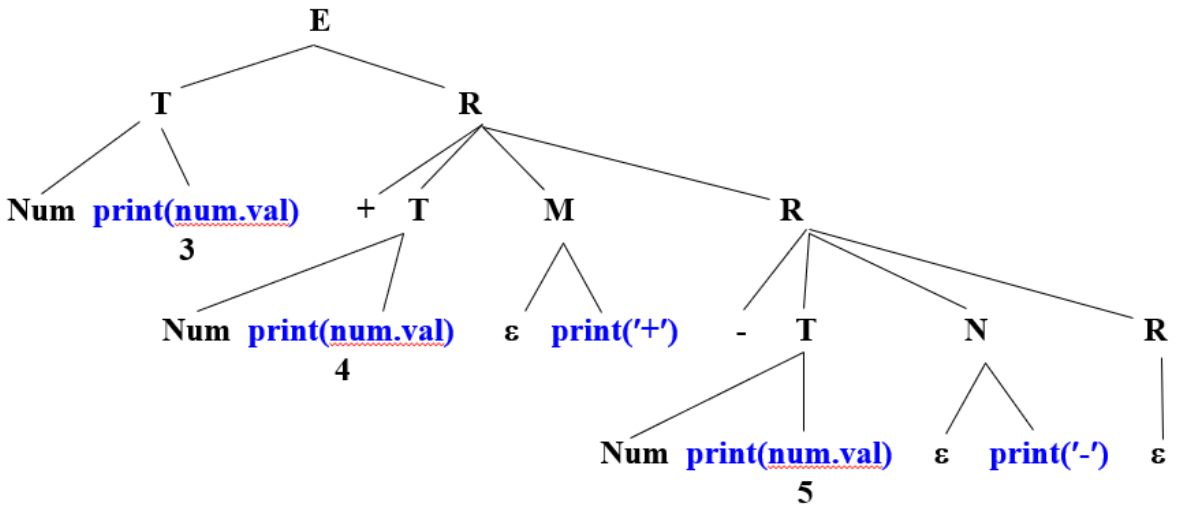
方案等价

- 变换前，表达式 3+4-5 的分析树：



- 深度优先遍历
- print(num1.val) print(num2.val) print('+') print(num3.val) print('-')
- 结果：34+5-

- 变换后，表达式 3+4-5 的分析树：



- 深度优先遍历
- print(num1.val) print(num2.val) print('+') print(num3.val) print('-')
- 结果：34+5-

2. 直接使用分析栈中的继承属性

- 前提：
  - 继承属性的值在栈中；
  - 在栈中的位置已知。
- 翻译方案：

  D→T { L.in=T.type } L

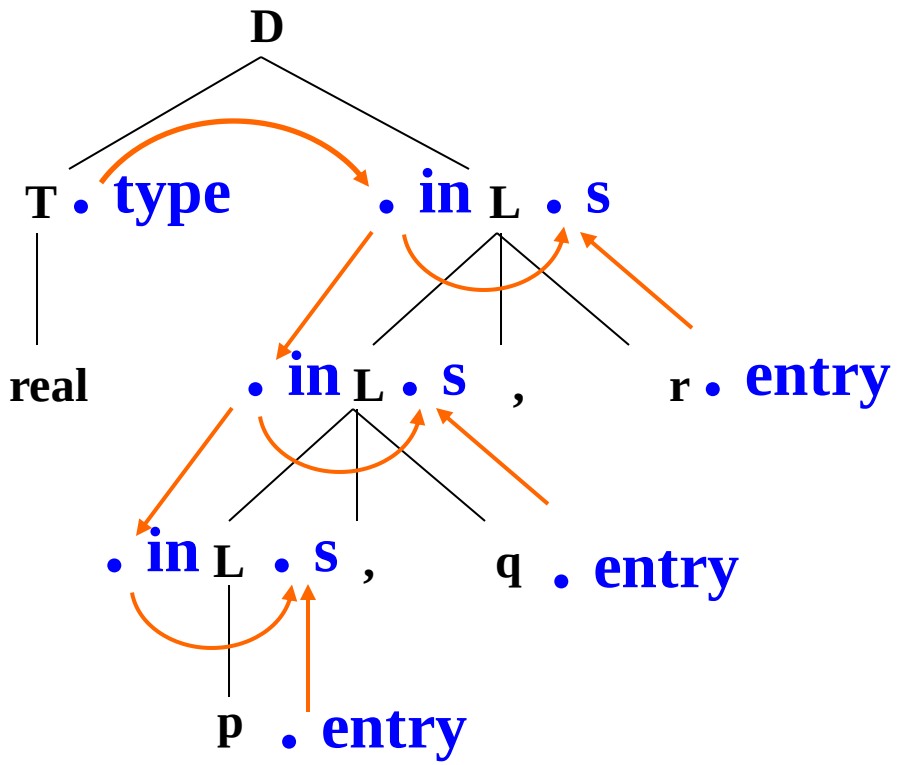
  T→int { T.type=integer }

  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) }
- 翻译输入符号串：

  real p, q, r



real p, q, r 的分析和翻译

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

addtype(id.entry, L.in)

- 用 L→id 归约时，L.in?
- 用 L→L, id 归约时，L.in?
- 和 L.in 有关的动作？
- 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])

3. 变换继承属性的计算规则

- 目的：使继承属性的值在栈中的位置可以预测。
- 例：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)

- 继承属性的值在栈中，但位置不可预测

val[top-1]？ val[top-2]？

- 引入标记非终结符号M及M→ε
- 设置属性M.i和M.s
- 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

- 继承属性的值在栈中，且位置可预测。

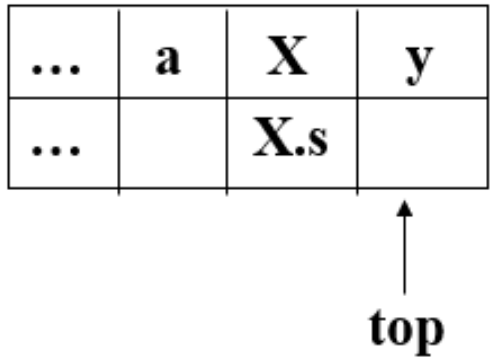
val[top-1]

模拟非复制规则的计算

- 例：考虑如下的产生式及语义规则：

  A→aXY　Y.i=f(X.s)

  Y→y　Y.s=g(Y.i)

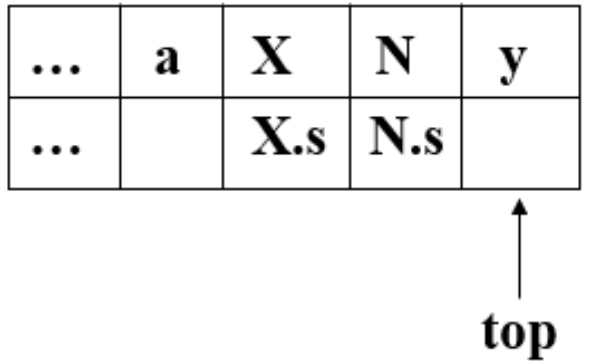


- 引入标记非终结符号N及N→ε
  （设置属性N.i 和 N.S）

  A→aXNY　N.i=X.s; Y.i=N.s

  N→ε　N.s=f(N.i)

  Y→y　Y.s=g(Y.i)



N.s：val[top+1]=f(val[top])

Y.s：val[top]=g(val[top-1])

- 结果：
  - 所有继承属性均使用复制规则
  - 继承属性的值在栈中，且位置可确定

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 $A \rightarrow X_1X_2 \ldots X_n$

引入n个新的标记非终结符号M_1、M_2、…、M_n

用产生式 $A \rightarrow M_1X_1M_2X_2 \ldots M_nX_n$ 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 的综合属性相联系

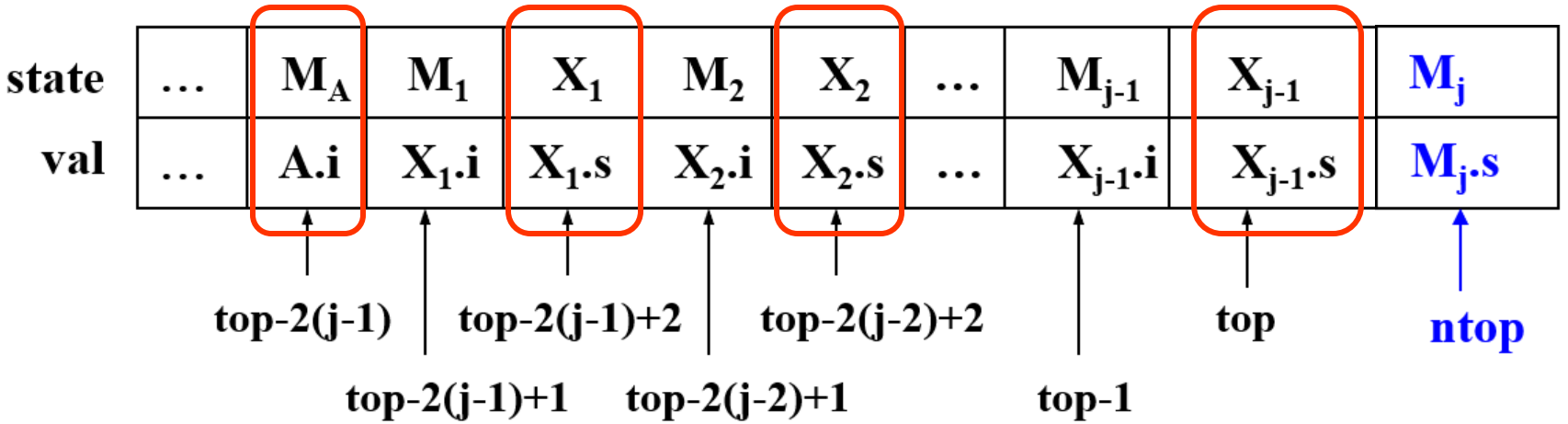
属性 $X_j.i$ $(=M_j.s)$ 总是在 M_j 处计算，且发生在完成X_{j-1}入栈之后，开始分析 X_j 的动作之前。

算法5.3：L属性定义的自底向上分析和翻译

(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

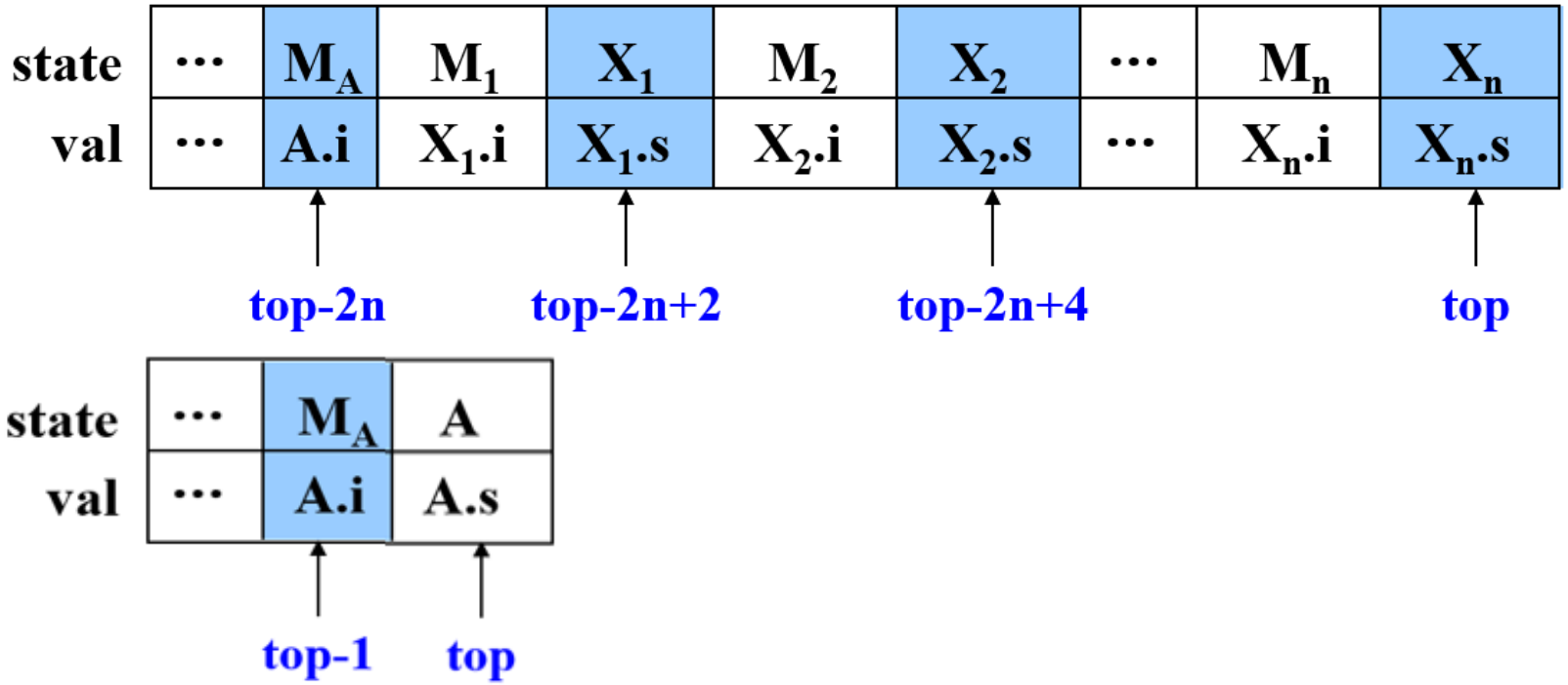
- 已知：$X_j.i = M_j.s = f(A.i, X_1.s, X_2.s, \ldots, X_{j-1}.s)$
  - 每个标记非终结符号在文法中是唯一的
  - M_j 所在产生式：$A \to M_1X_1M_2X_2\ldots M_{j-1}X_{j-1}M_jX_j\ldots M_nX_n$
  - 计算属性 $X_j.i$ 需要哪些属性、以及它们的位置



算法5.3：L属性定义的自底向上分析和翻译

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

- 已知：$A.s= f(A.i, X_1.s, X_2.s, \ldots, X_n.s)$
  - 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



4. 改写语法制导定义为S属性定义

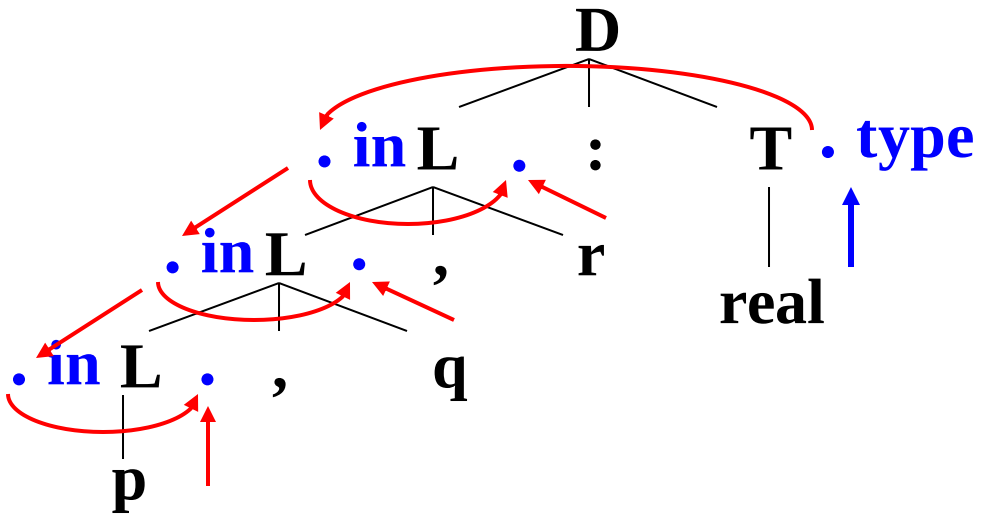
- 例：PASCAL的变量声明语句可由如下文法产生：

  D→L:T

  T→integer | real

  L→L, id | id

- p, q, r: real



- 问题：
  - 标识符由L产生，类型由T产生，且不在L的子树中；
  - LR分析，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。

解决方法

- 改写文法，使类型作为标识符表的最后一个元素

  D → id L

  L → , id L | : T

  T → integer | real

- 如：p, q：real
- 改写后，归约方向与类型信息传递方向一致
  - 翻译方案

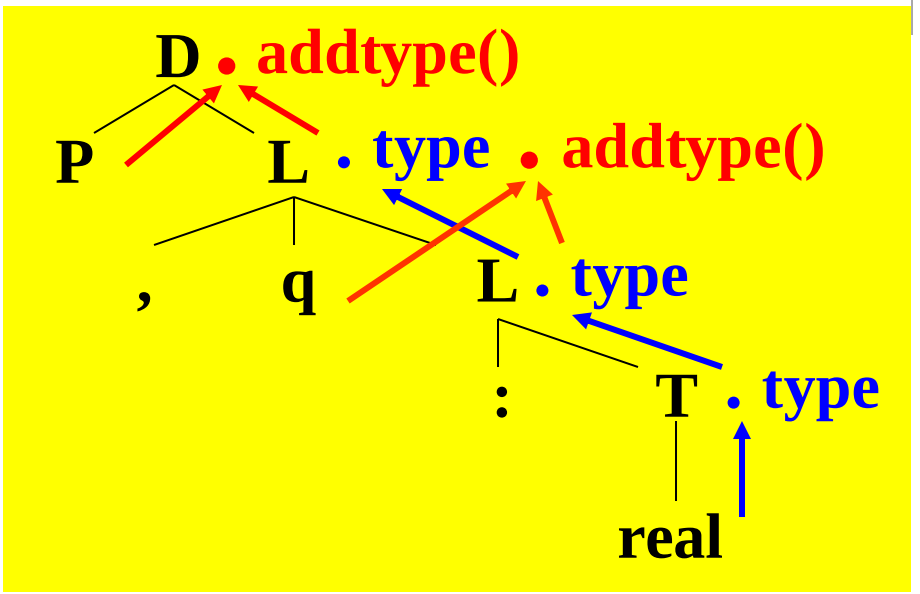
  D → id L { addtype(id.entry, L.type) }

  L → , id L_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 }



本章小结

- 综合属性、继承属性
- 语法制导定义
  - S-属性定义
  - L-属性定义（继承属性应满足的限制条件）
- 翻译方案
  - 构造S-属性定义的翻译方案（语义动作放在产生式右尾）
  - 构造L-属性定义的翻译方案（语义动作插入产生式之中）
- S-属性定义的自底向上翻译（改造LR分析程序）
  - 分析栈的扩充
  - 分析控制程序的修改
- L-属性定义的自底向上翻译
  - 通过复制规则引用继承属性
  - 引入标记非终结符号及相应的产生式
  - 修改语义动作、使继承属性由复制规则实现

学习任务

- 作业要求
  - 利用语法制导定义/翻译方案，翻译输入符号串；
  - 对于给定文法，根据翻译目标的要求，设计语法制导定义/翻译方案。
- 研究性学习
  - L属性定义的自顶向下实现。

课堂练习

☆作业 5.16

有如下文法：

S→(L) | a

L→L, S | S

(1) 设计一个语法制导定义，统计并输出配对的括号个数。

(2) 构造一个翻译方案，计算并输出每个a的嵌套深度。

如对句子(a, (a , a))

(1) 输出2

(2) 输出结果是1，2，2

☆作业 5.17

令综合属性 val 给出在下面的文法中 S 产生的二进制数的十进制数值。

如对于输入101.101，S.val=5.625

S→L.L | L

L→LB | B

B→0 | 1

请写出确定 S.val 值的语法制导定义。

★练习

针对如下文法，设计一个翻译方案，
打印出输入二进制数中每个1的权值。

S→L.L | L

L→LB | B

B→0 | 1

如对于输入**101.101**，
打印输出：**4**，**1**，**0.5**，**0.125**。

★课堂练习1

- 有文法G[S]:

  S→SaA | A

  A→AbB | B

  B→cSd |e

(1)说明 ebeaebced 是该文法的一个句子;

(2)为该文法设计一个翻译方案，利用该翻译方案，可以在自底向上的分析中把上述句子翻译为1314513135246。

ebcedae==>1313524136

★课堂练习2

有如下文法：

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

设计一个翻译方案，使其打印出每个a在输入符号串中的位置。

比如，对于输入符号串(a, (a, a))，打印输出：2 5 7

★课堂练习3

有如下文法：

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

设计一个翻译方案，打印输出每个a在输入符号串中的位置，并统计输出a的个数。

比如，对于输入符号串(a, (a, a))，

打印输出：

第1个a的位置是2

第2个a的位置是5

第3个a的位置是7

一共有3个a。

★课堂练习4

有如下文法：

S→(L) | a

L→L, S | S

设计一个翻译方案，使其打印出每个a在输入符号串中的位置,并统计输出a的个数。

比如，对于输入符号串(a, (a, a))，打印输出：

第1个a的位置是2，嵌套深度是1

第2个a的位置是5，嵌套深度是2

第3个a的位置是7，嵌套深度是2

字符串长度为9，一共有3个a。