编译原理与技术

--属性文法和语法制导翻译I

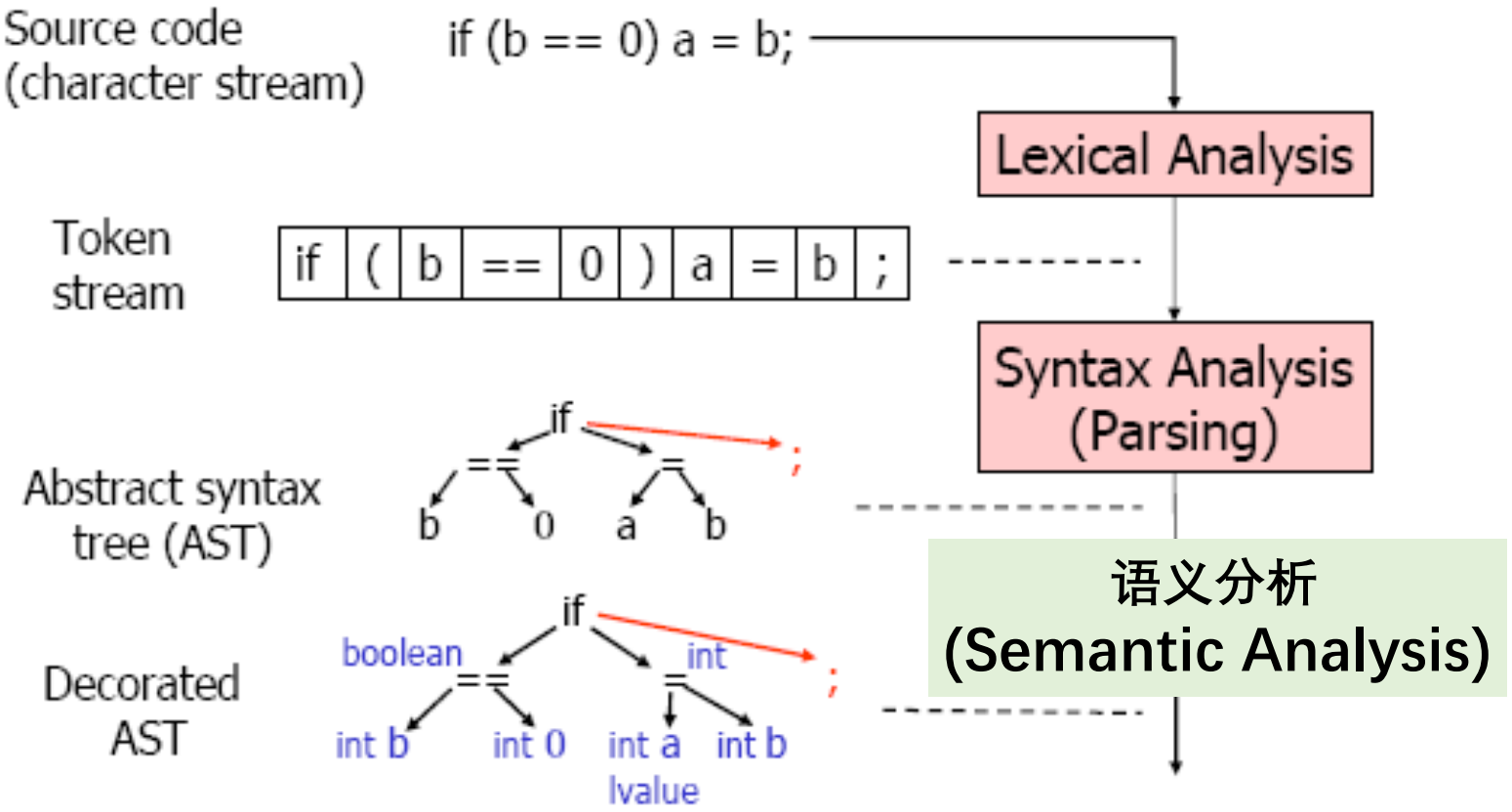
陈俊洁

天津大学智算学部

Outline

- 属性文法和语法制导翻译
- 基于属性文法的处理方法
- S – 属性文法的自下而上计算
- L – 属性文法的自顶向下翻译
- 自下而上计算继承属性

语义分析所处位置


Source code
(character stream)
if (b == 0) a = b;
Lexical Analysis
Token
stream
if (b == 0) a = b ;
Syntax Analysis
(Parsing)
Abstract syntax
tree (AST)
if
==
=
;
b
0
a
b
语义分析
(Semantic Analysis)
Decorated
AST
if
boolean
==
int
=
;
int b
int 0
int a
lvalue
int b

语义分析

- 语义分析和中间代码生成是编译程序构造的第三阶段,普遍采用语法制导翻译方法,为每个产生式配一个翻译子程序(称语义动作或语义子程序),并且在语法分析的同时执行这些子程序.
- 语义动作指出:

  (1)一个产生式所产生的符号的意义；

  (2) 按照这种意义规定了生成某种中间代码应作哪些基本动作。

属性文法

- 属性文法：上下文无关文法的基础上，为每个文法符号（终结符或非终结符）配备若干相关的“值”（称为“属性”）。
  - 属性代表与文法符号相关的信息，如类型、值、代码序列、符号表内容
  - 属性可以计算和传递，属性加工过程即是语义处理过程
  - 语法树结点中的属性要用语义规则来定义,语义规则和相应结点的产生式相关

属性(attribute)：编程语言结构的任意特性

- 静态(static)：执行之前绑定的属性
- 动态(dynamic)：执行期间绑定的属性
- 属性的典型例子有：变量的数据类型：静态； 表达式的值：动态；
  存储器中变量的位置：静态或动态； 程序的目标代码：静态

- 语义规则：文法的每个产生式都配备一组属性的计算规则
  - 包括属性计算，静态语义检查，符号表操作，代码生成等

属性的计算

- 属性：综合属性与继承属性
  - 综合属性用于“自下而上”传递信息
  - 继承属性用于“自上而下”传递信息

- 终结符只有综合属性，它们由词法分析器提供；
- 非终结符既可有综合属性也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值。

- 属性文法中，对应于每个产生式A→α都有一套与之相关联的语义规则，每条规则的形式为b:=f(c1,c2…ck)，f是一个函数，b和c1,c2…ck是该产生式文法符号的属性. 并且
  - b是A的一个综合属性，且c1,c2,...,ck是该产生式右边文法符号的属性
  - b是产生式右部某个文法符号的一个继承属性，且c1,c2,...,ck是A或者产生式右边任何文法符号的属性
  - 在这两种情况下，我们说属性b依赖于c1,c2,...,ck 。

例：A→X1 X2 …Xn

A的综合属性S(A)计算公式:

S(A):=f(I(X1),…,I(Xn))

Xj的继承属性T(Xj)计算公式:

T(Xj):=f(I(A),… ,I(Xn))

属性的计算

- 一般说来，对于出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个计算规则。
- 属性计算规则中只能使用相应产生式中的文法符号的属性，这有助于在产生式范围内“封装”属性的依赖性。
- 出现在产生式左边的继承属性和出现在产生式右边的综合属性不由所给定的产生式的属性计算规则进行计算，它们由其它产生式的属性规则计算或者由属性计算器的参数提供。

例如产生式A→BC：
- 这个产生式可以计算A的综合属性、B和C的继承属性。
- A的继承属性，可能需要根据某个类似于X→…A…的产生式求得
- B和C的综合属性可能需要根据某个类似于B→β，以及C →γ的产生式求得

属性文法举例

- 台式计算器程序的属性文法
  - 每个非终结符都有一个综合属性---整数值val
  - 每个产生式对应一个语义规则，产生式左边的非终结符的属性值val是从右边的非终结符的属性值val计算出来的

产生式	语义规则
L→En	**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.lexval**

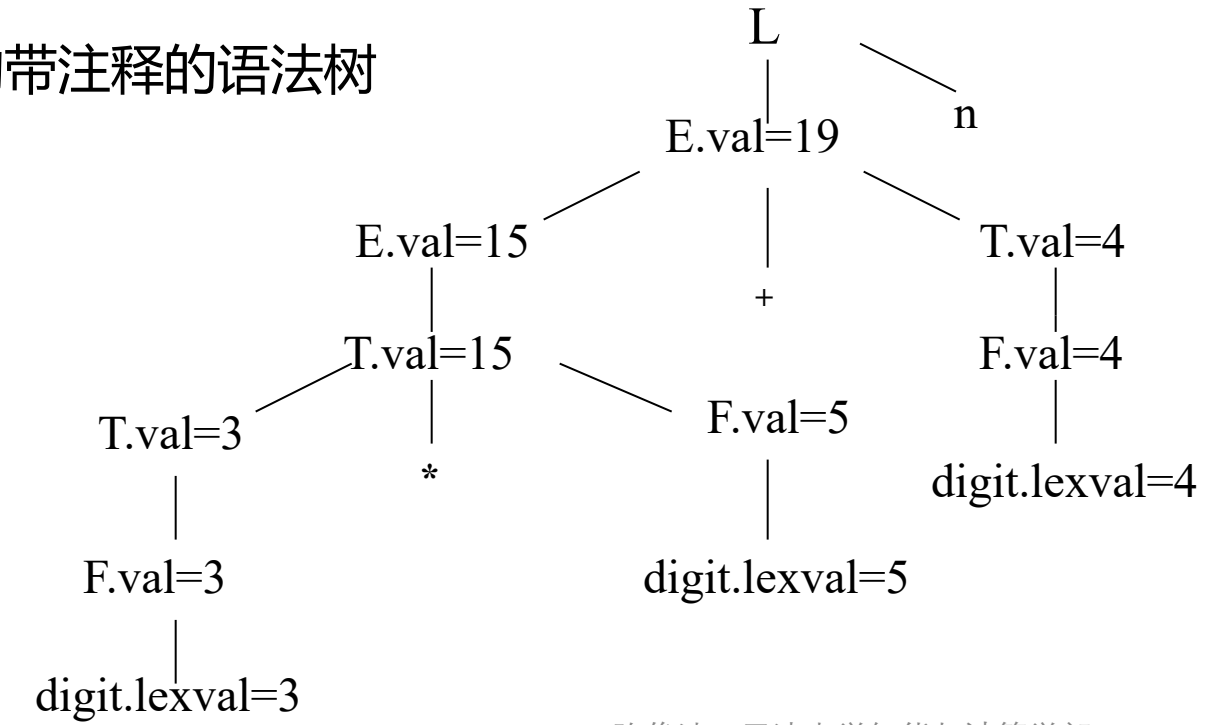
注：同一非终结符多次出现，用下标消除对这些非终结符的属性值引用的二义性

综合属性

- 结点的综合属性由其子结点的属性值确定
- 通常采用自底向上的方法计算综合属性

例：设表达式为3＊5+4n，则语义动作打印数值19

3*5+4n的带注释的语法树



产生式	语义规则
L→En	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.lexval

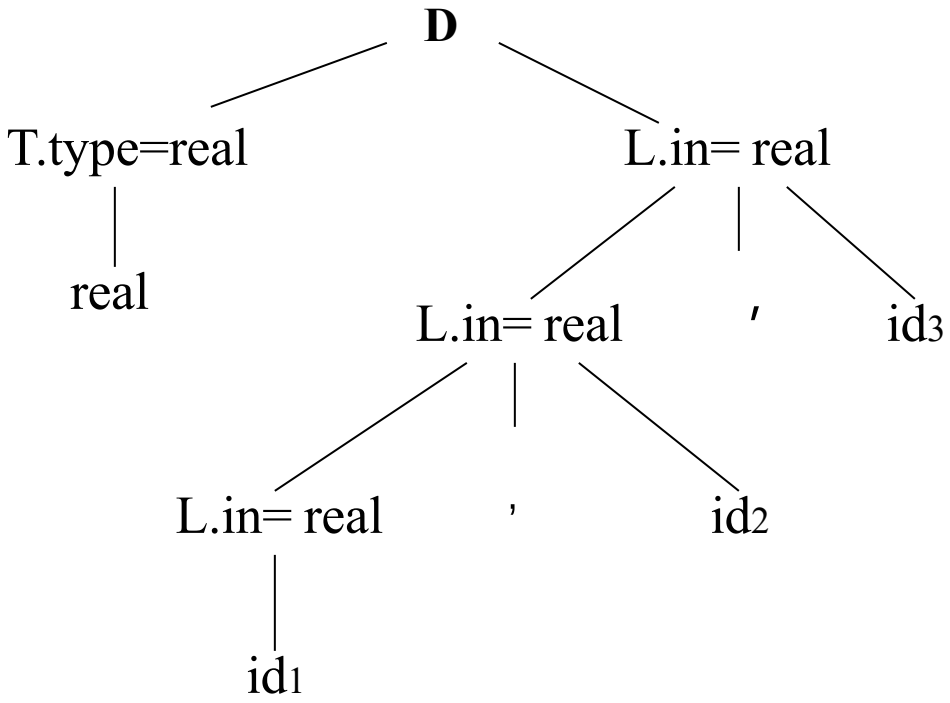
继承属性

- 结点的继承属性值是由此结点的父结点和/或兄弟结点的某些属性来决定的

例如：

产生式	语义规则
D →TL	L.in:=T.type
T→ int	T.type:=integer
T →real	T.type:=real
L → L1,id	L1.in:=L.in addtype(id.entry,L.in)
L→ id	addtype(id.entry,L.in)

句子 Real id1,id2,id3 的语法树

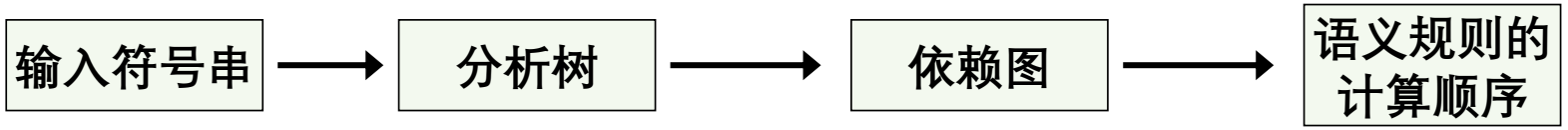


Outline

- 属性文法和语法制导翻译
- 基于属性文法的处理方法
- S – 属性文法的自下而上计算
- L – 属性文法的自顶向下翻译
- 自下而上计算继承属性

语法制导翻译

- 由源程序的语法结构所驱动的处理办法(在语法分析过程中,随着分析的步步进展,根据每个产生式对应的语义程序(语义动作)进行翻译(产生中间代码)的办法叫做语法制导翻译法.)
- 基于属性文法的处理过程通常是：
  - 对符号串进行语法分析,
  - 构造语法分析树
  - 根据需要遍历语法树并在语法树的各结点处按语义规则进行计算。



- 对输入串的翻译就是根据语义规则进行计算的结果
- 在某些情况下，在进行语法分析的同时完成语义规则的计算而无须明显地构造语法树或构造属性之间的依赖图。（一遍处理，L-属性文法）

语法制导定义

- 一个语法制导定义是一个上下文无关文法,其中每个文法符号都有一个相关的属性集合

例子： “算术表达式” E的 “值” 的语义(语法制导定义):

产生式	语义规则
(1)$E \rightarrow E^{(1)} + E^{(2)}$	{E.VAL:= $E^{(1)}$.VAL+$E^{(2)}$.VAL}
(2)$E \rightarrow 0$	{E.VAL:=0 }
(3)$E \rightarrow 1$	{E.VAL:=1 }

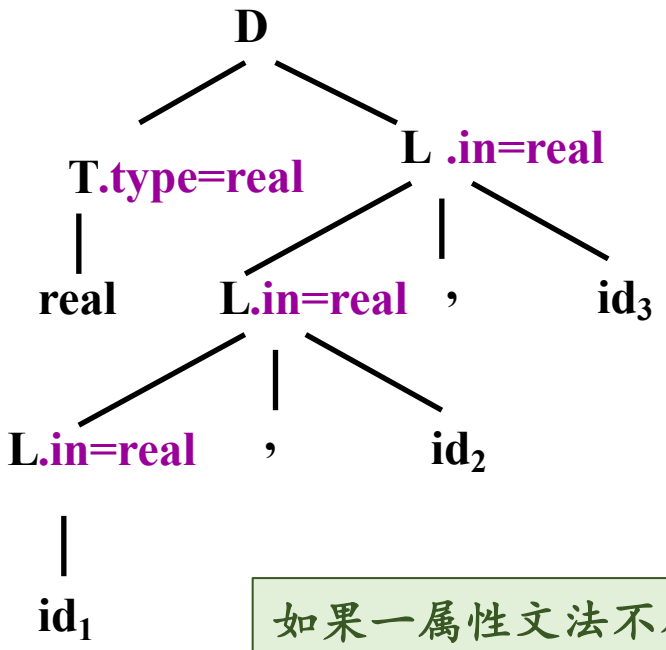
依赖图

- 描述语法分析树中的继承属性和综合属性之间的相互依赖关系的有向图。
  - 表示了节点属性的计算先后顺序。如果分析树中某个节点的属性b依赖于属性c，那么在该节点处b的语义规则必须在c的语义规则之后计算。
- 依赖图的构造方法
  - 为每个包括过程调用的语义规则引入一个虚综合属性b，把每条语义规则都变成b=f(c1,c2,...,ck)的形式
  - 依赖图的每个结点表示一个属性
  - 边表示属性间的依赖关系。如果属性b依赖于属性c，那么从c到b就有一条有向边

```
FOR 语法树中每一个结点n DO
  FOR 结点n的文法符号的每一个属性 a DO
    为a在依赖图中建立一个结点；
FOR 语法树中每一个结点n DO
  FOR结点n所用产生式对应的每一个语义规则
      b:=f(c1,c2,…,ck)DO
      FOR  i :=1 to k DO
          从ci结点到b结点构造一条有向边
```

依赖图举例

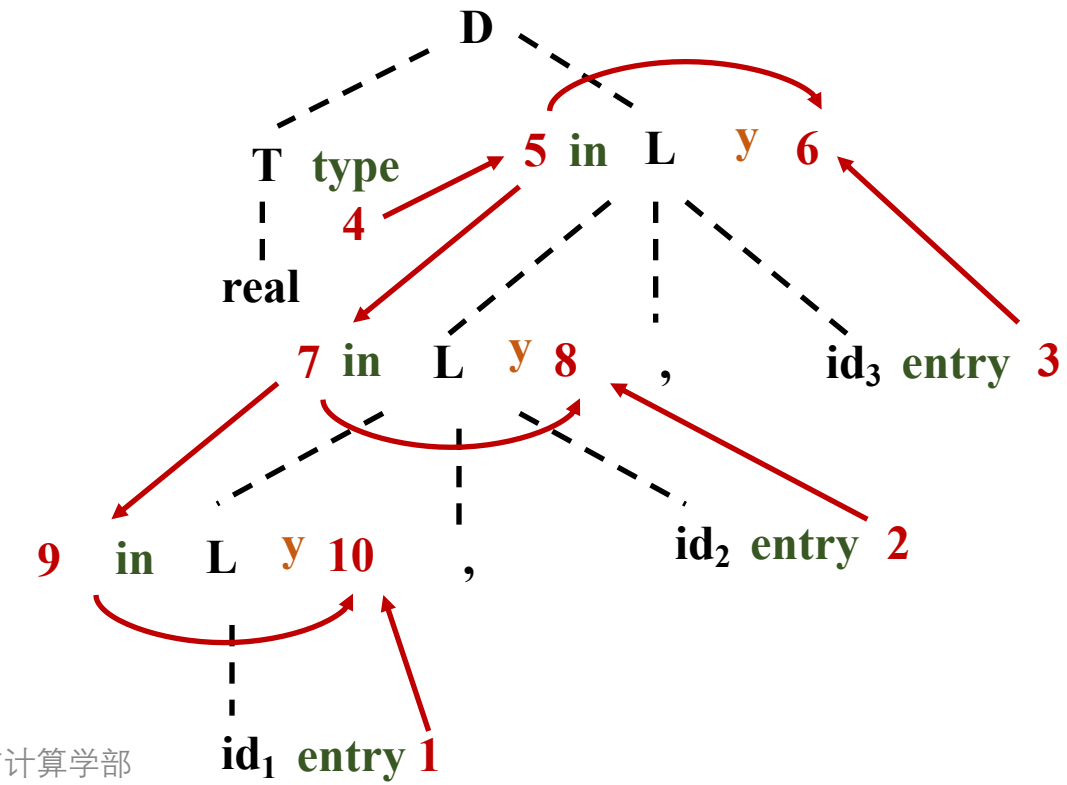
产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L \rightarrow L_1$, id	L_1.in := L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



6，8，10是为虚属性构造的结点，对应操作addtype(id.entry, L.in)

如果一属性文法不存在属性之间的循环依赖关系，那么称该文法为良定义的

一个属性对另一个属性的循环依赖关系。
如，p、c1、c2都是属性，若p: = f1(c1)、c1:= f2(c2)、c2: = f3(p)时，就无法对p求值。



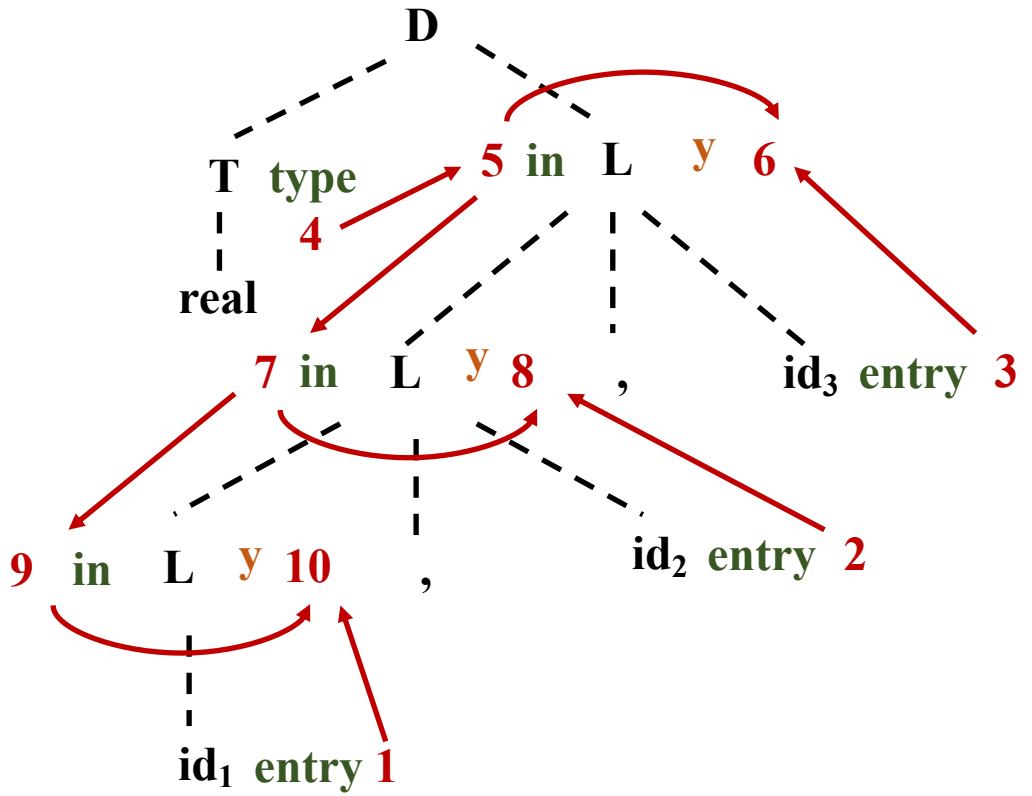
属性的计算顺序

- 无环有向图的拓扑排序
  - 无环有向图中节点$m_1,m_2,\cdots,m_k$的拓扑排序是：若$m_i \rightarrow m_j$是从m_i到m_j的边，那么在此排序中m_i先于m_j
  - 依赖图的任何拓扑排序都给出了一个分析树中各节点语义规则计算的正确顺序，即在计算f之前，语义规则$b=f(c_1,c_2,...,c_k)$中的依赖属性$c_1,c_2,...,c_k$都是已知的
- 属性文法所说明的翻译可以按照下面的步骤进行
  - 最基本的文法用于构造输入串的分析树
  - 用前面的方法构造依赖图
  - 从依赖图的拓扑排序可以得到语义规则的计算顺序
  - 按该顺序计算语义规则即可得到输入串的翻译

属性文法计算顺序举例

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)

- a_4 := real
- a_5 := a_4
- addtype(id_3.entry, a_5)
- a_7 := a_5
- addtype(id_2.entry, a_7)
- a_9 := a_7
- addtype(id_1.entry, a_9)



树遍历的属性计算方法

- 通过树遍历计算属性值的方法都假设语法树已经建立，并且树中已带有开始符号的继承属性和终结符的综合属性。
- 最常用的遍历方法：深度优先，从左到右的遍历方法。如果需要可使用多次遍历。
- 只要文法的属性是非循环定义的，则每次扫描至少有一个属性值被计算出来。

树遍历算法：

```
While 有未被计算的属性do
    VisitNode (S)    /*S是开始符号*/
```

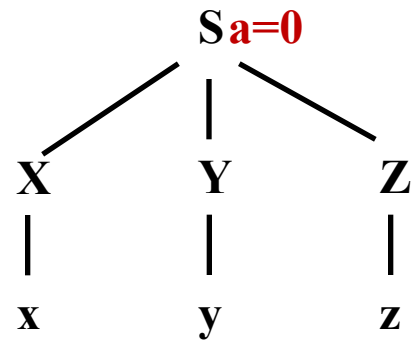
最坏的情况：计算复杂度$O(n^2)$
n为结点个数

```
Procedure VisitNode ( N:Node);
begin
  if N 为非终结符then          /*假设它的产生式为N→X1…Xm*/
    for i:=1 to m do
      if  Xi∈VN then
        begin
          计算Xi的所有能够计算的继承属性；
           VisitNode (Xi)
        end;
    计算N的所有能够计算的综合属性
end
```

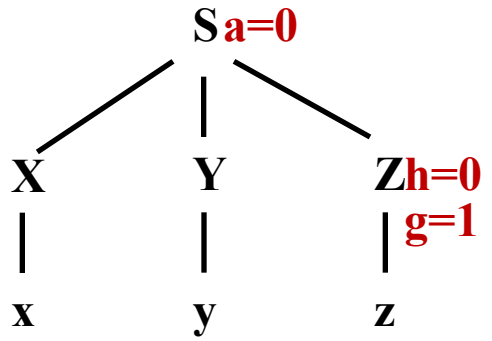
树遍历的属性计算方法

产生式	语义规则
S→XYZ	Z.h := S.a
	X.c := Z.g
	S.b := X.d – 2
	Y.e := S.b
X→x	X.d := 2*X.c
Y→y	Y.f := Y.e * 3
Z→z	Z.g := Z.h + 1

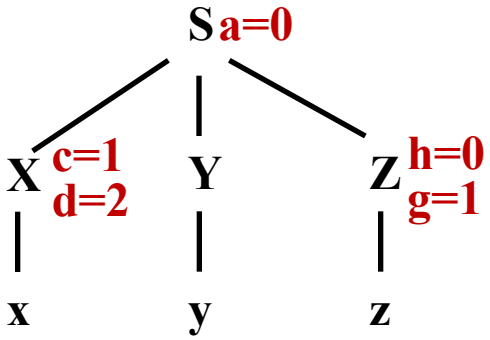
S有继承属性a，综合属性b
X有继承属性c，综合属性d
Y有继承属性e，综合属性f
Z有继承属性h，综合属性g
假设S.a的初始值为0



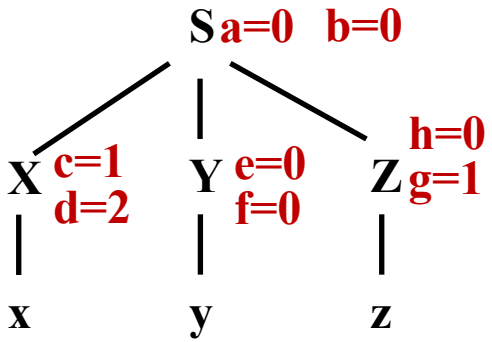
初始状态



第一遍扫描



第二遍扫描



第三遍扫描

一遍扫描的计算方法

- 在语法分析的同时计算属性值，而不是语法分析构造语法树之后进行属性的计算，而且无需构造实际的语法树。
- 与两个因素密切相关：所采用的语法分析方法、 属性的计算次序。
- 语法制导翻译：为文法中每个产生式配上一组语义规则，在语法分析的同时执行语义规则
  - 在自顶向下语法分析中，若一个产生式匹配输入串成功时
  - 在自底向上语法分析中，当一个产生式被用于进行归约时
- L-属性文法可用于一遍扫描的自顶向下分析，而S-属性文法适合于一遍扫描的自底向上分析。

Outline

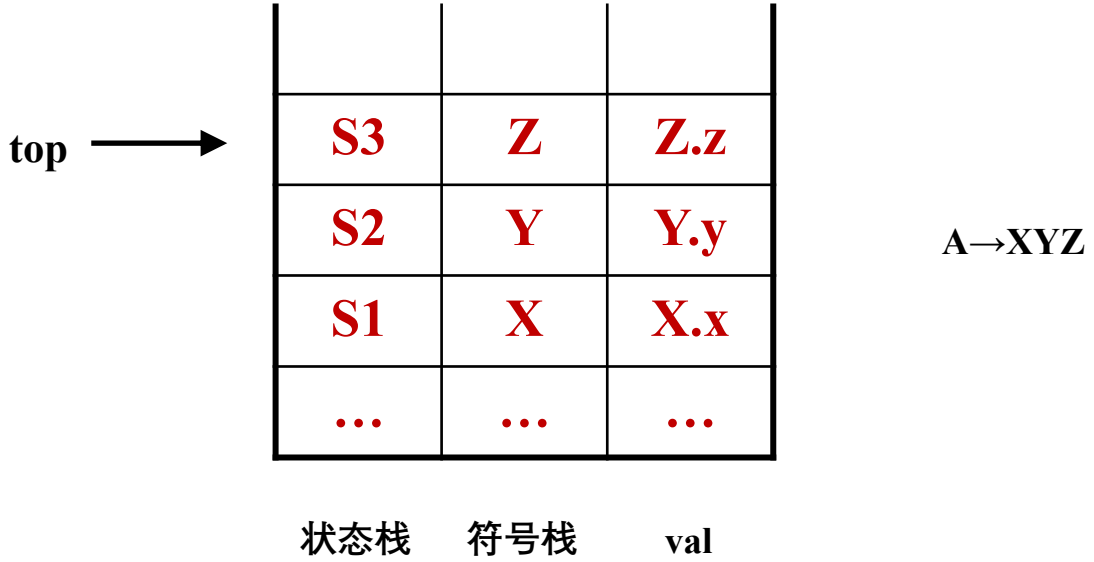
- 属性文法和语法制导翻译
- 基于属性文法的处理方法
- S – 属性文法的自下而上计算
- L – 属性文法的自顶向下翻译
- 自下而上计算继承属性

S-属性文法的特点

- S-属性文法，只有综合属性
- 产生式左边的文法的综合属性要根据产生式右边符号的综合属性来进行计算。
- 适用于那些需要类似于表达式，需要计算结果的文法。
- 综合属性可以在分析输入符号串的同时由自底向上的分析器来计算。
  - 分析器保存与栈中文法符号有关的综合属性
  - 每当归约时，新的属性值就由栈中正在归约的产生式右边符号的属性值来计算。

S-属性文法翻译器的实现

- S-属性文法的翻译器通常可借助于LR分析器实现。
- 在自底向上的分析方法中，我们使用栈来存放已经分析过了的子树，现在我们可以在分析栈中使用一个附加域来存放综合属性值。
- 假设综合属性是刚好在每次归约前计算的



S-属性文法计算举例

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

代码段刚刚好在**归约前执行**。

这是利用归约提供一个“挂钩”，使得用户把一个语义动作与一个产生式联系起来。

翻译模式可以提供一种与分析器相互穿插动作的描述方法。

我们要控制两个变量top和ntop。

当右边带有r个符号的产生式被归约时，执行相应的代码段之前，先将top-r+1赋给ntop，在代码段被执行之后将ntop的值赋给top

S-属性文法计算举例

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

	输入	Symbol	Val	用到的产生式
0	3*5+4n	-	-	
1	*5+4n	3	3	
2	*5+4n	F	3	F→digit
3	*5+4n	T	3	T→F
4	5+4n	T*	3-	
5	+4n	T*5	3-5	
6	+4n	T*F	3-5	F→digit
7	+4n	T	15	T→T*F
8	+4n	E	15	E→T
9	4n	E+	15-	
10	n	E+4	15-4	
11	n	E+F	15-4	F→digit
12	n	E+T	15-4	T→F
13	n	E	19	E→E+T
14		En	19-	
15		L	19	L→En

S-属性文法练习

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

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			

S-属性文法练习

	动作	状态栈	语义栈(值栈)	符号栈	余留输入串

2 + 3 * 5 的分析和计值过程