编译原理

--属性文法和语法制导翻译

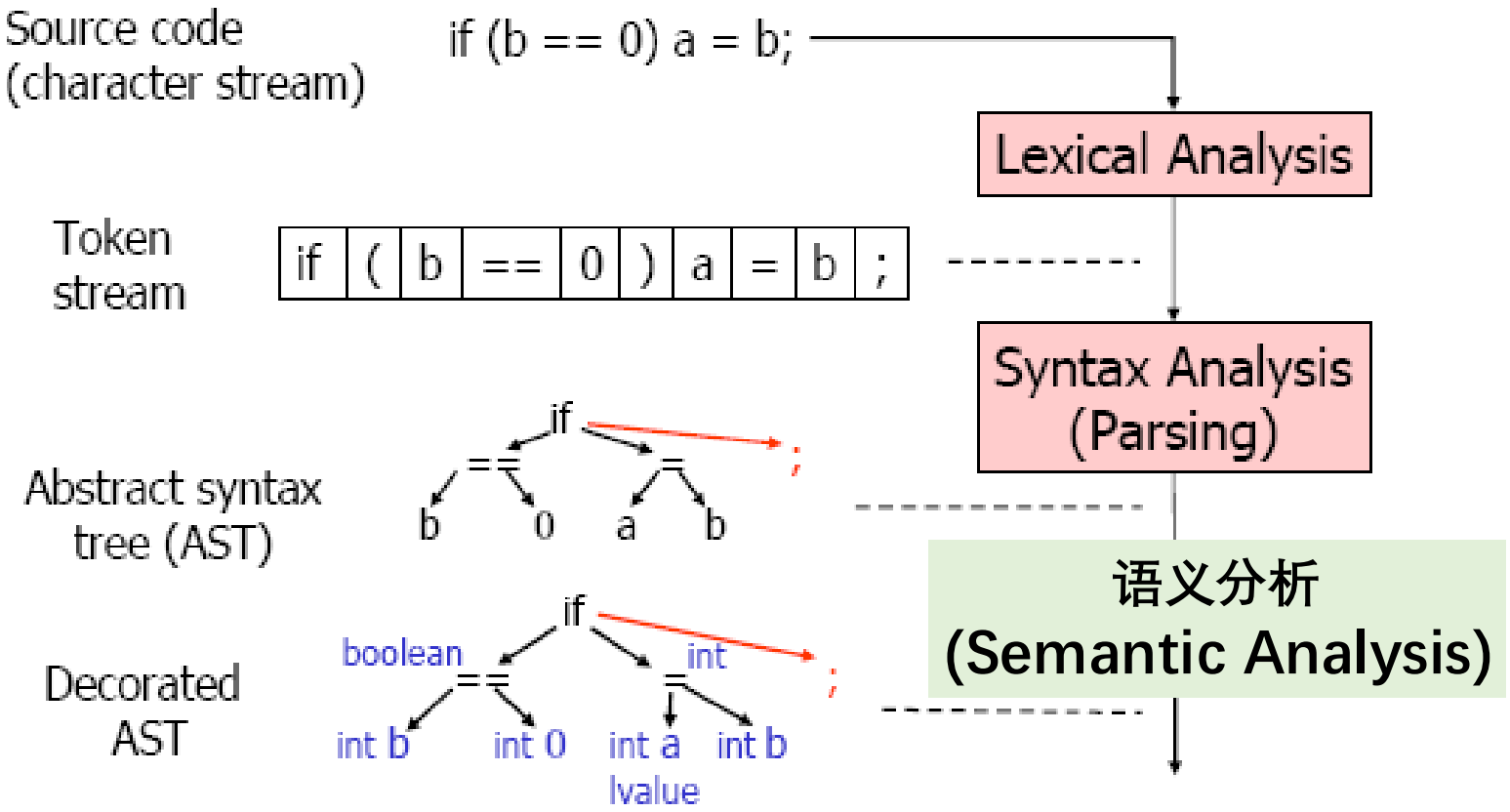
刘爽

中国人民大学信息学院

Outline

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

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。

翻译文法和语法制导翻译

有上下无关文法G[E]:

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

此文法是一个中缀算术表达式文法

- 翻译的任务是: 中缀表达式 → 逆波兰表示

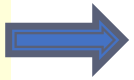
  即 a+b*c → abc*+

假如翻译任务是要将中缀表达式简单变换为波兰后缀表示，

只需在上述文法中插入相应的动作符号。

翻译文法和语法制导翻译

1. E →E+T	**4. T →F**
2. E →T	**5. F →(E)**
3. T →T*F	**6. F →i**



1. E →E+T@+	**4. T →F**
2. E →T	**5. F →(E)**
3. T →T*F @*	**6. F →i@i**

其中：

- @+,@*,@i 为动作符号。@为动作符号标记，后面为字符串。
- 在本例中，其对应语义子程序的功能是要输出打印动作符号标记后面的字符串。

  所以，产生式1：E→E+T@+ 的语义是分析E, + 和T，输出+

  产生式6：F→i @ i的语义是分析i，输出i

翻译文法和语法制导翻译

下面给出输入文法和翻译文法的概念:

输入文法: 未插入动作符号时的文法。

由输入文法可以通过推导产生输入序列。

翻译文法: 插入动作符号的文法。

由翻译文法可以通过推导产生活动序列。

例子

例：(i+i) * i

可以用输入文法推导：

$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E+T)*F \overset{*}{\Rightarrow} (i+i)*i$

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

用相应的翻译文法推导，可得：

$E \Rightarrow T$
$\Rightarrow T*F@*$
$\Rightarrow F*F@*$
$\Rightarrow (E)*F@*$
$\Rightarrow (E+T@+)*F@* \overset{*}{\Rightarrow} (i@i+i@i\ @+)*i@i@*$

1. E →E+T@+
2. E →T
3. T →T*F @*
4. T →F
5. F →(E)
6. F →i@i

翻译文法

(i@i+i@i @+)*i@i@*

活动序列：由翻译文法推导出的符号串，由终结符和动作符号组成。

- **从活动序列中，抽去动作符号，则得输入序列 (i+i)*i**
- **从活动序列中，抽去输入序列，则得动作序列，执行动作序列，则完成翻译任务：**

$$@i@i@+@i@* \Rightarrow ii+i*$$

定义7.1

翻译文法是上下文无关文法，其终结符号集由输入符号和动作符号组成。由翻译文法所产生的终结符号串称为活动序列。

语法制导翻译

上例题中的翻译文法为：

$G_T = (V_n, V_t, P, E)$

$V_n = \{E, T, F\}$

$V_t = \{i, +, *, (,), @+, @*, @i\}$

$P = \{E \to E+T@+,\ E \to T,\ T \to T*F@*,\ T \to F,\ F \to (E),\ F \to i@i\}$

语法导制翻译：按翻译文法进行的翻译。给定一输入符号串，根据翻译文法获得翻译该符号串的动作序列，并执行该序列所规定的动作的过程。

语法导制翻译的实现方法：在文法的适当位置插入语义动作符号，当按文法分析到动作符号时就调用相应的语义子程序，完成翻译任务。

属性文法

- 属性文法：翻译文法的基础上，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

- 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：综合属性与继承属性
  - 综合属性用于“自下而上”传递信息（用↑c表示，其中↑为综合属性标记，C为属性值）
  - 继承属性用于“自上而下”传递信息（用↓c表示，其中↓为继承属性标记，C为属性值）
- 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**

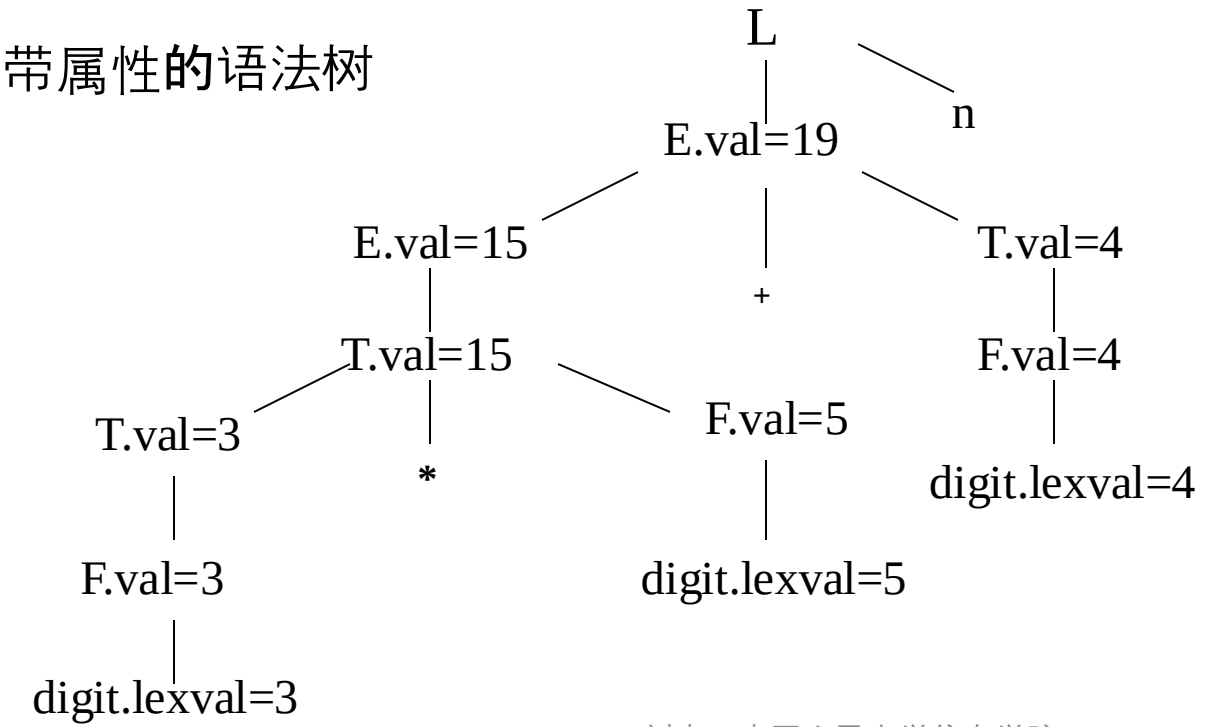
注：同一非终结符多次出现，用下标消除对这些非终结符的属性值引用的二义性

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：表达式为3＊5+4n，则语义动作打印数值19

3*5+4n的带属性的语法树



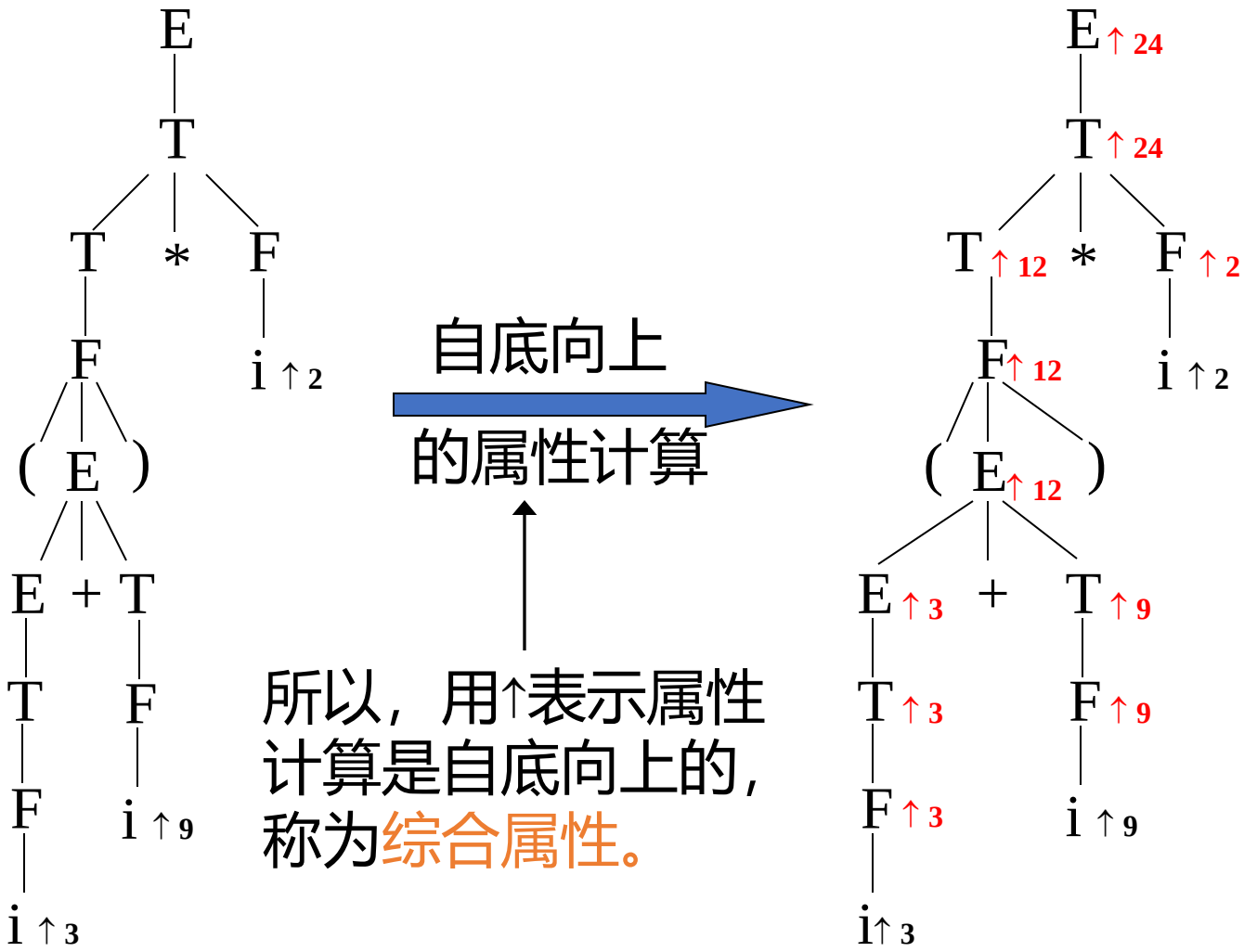
产生式	语义规则
L→En	**print(E.val)**
$E \rightarrow E_1 + T$	**$E.val := E_1.val + T.val$**
E→T	**E.val := T.val**
$T \rightarrow T_1 * F$	**$T.val := T_1.val * F.val$**
T→F	**T.val := F.val**
F→(E)	**F.val := E.val**
F→digit	**F.val := digit.lexval**

综合属性-例子

$(i\uparrow_3 + i\uparrow_9) * i\uparrow_2$

产生式	求值规则
1.$E\uparrow_{p4} \longrightarrow E\uparrow_{q5} + T\uparrow_{r2}$	$p_4 := q_5 + r_2$;
2.$E\uparrow_{p3} \longrightarrow T\uparrow_{q4}$	$p_3 := q_4$;
3.$T\uparrow_{p2} \longrightarrow T\uparrow_{q3} * F\uparrow_{r1}$	$p_2 := q_3 * r_1$;
4.$T\uparrow_{p2} \longrightarrow F\uparrow_{q2}$	$p_2 := q_2$;
5.$F\uparrow_{p1} \longrightarrow (E\uparrow_{q1})$	$p_1 := q_1$;
6.$F\uparrow_{p1} \longrightarrow i\uparrow_{q1}$	$p_1 := q_1$;

根据给定的文法，可写出该输入序列的语法树



所以，用↑表示属性计算是自底向上的，称为综合属性。

说明：

- p,q,r为属性变量名。
- 属性变量名局部于每个产生式，也可使用不同的名字。

继承属性

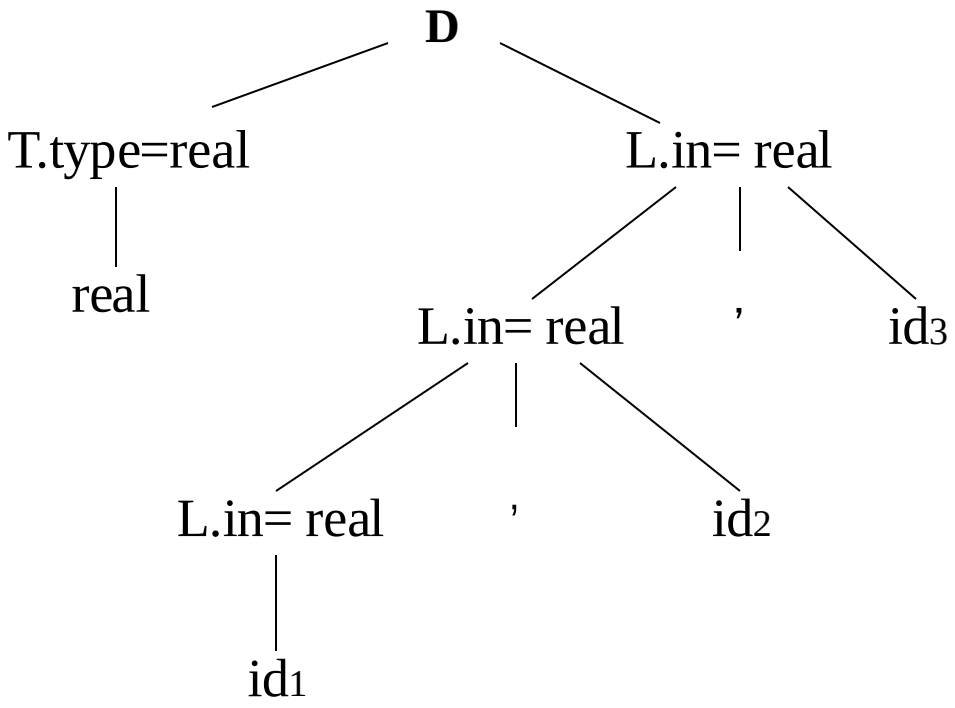
- 结点的继承属性值是由此结点的父结点和/或兄弟结点的某些属性来决定的

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)

type为综合属性，in为继承属性。

句子 Real id1,id2,id3 带属性的语法树

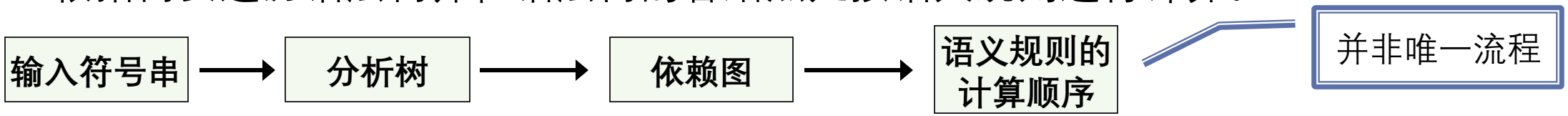


Outline

- 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）

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
  （1）最基本的文法用于构造输入串的分析树
  （2）构造某种拓扑排序，如依赖图
  （3）从拓扑排序可以得到语义规则的计算顺序
  （4）按该顺序计算语义规则即可得到输入串的翻译

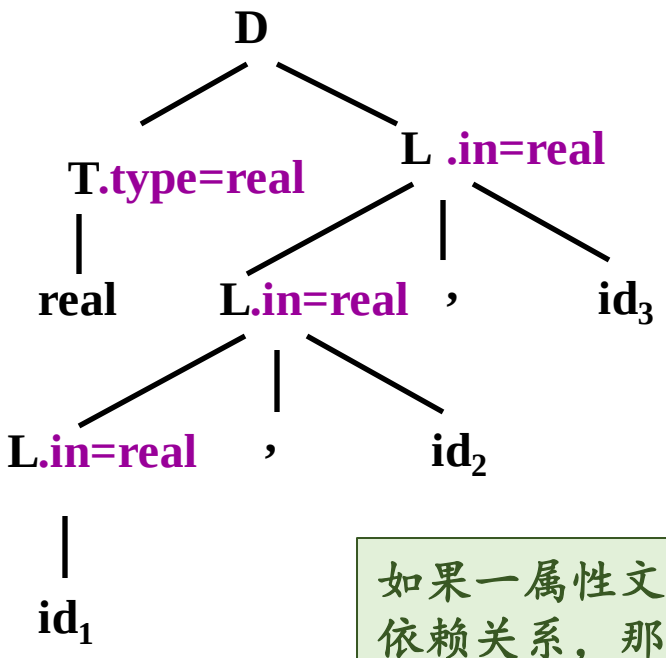
依赖图

- 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

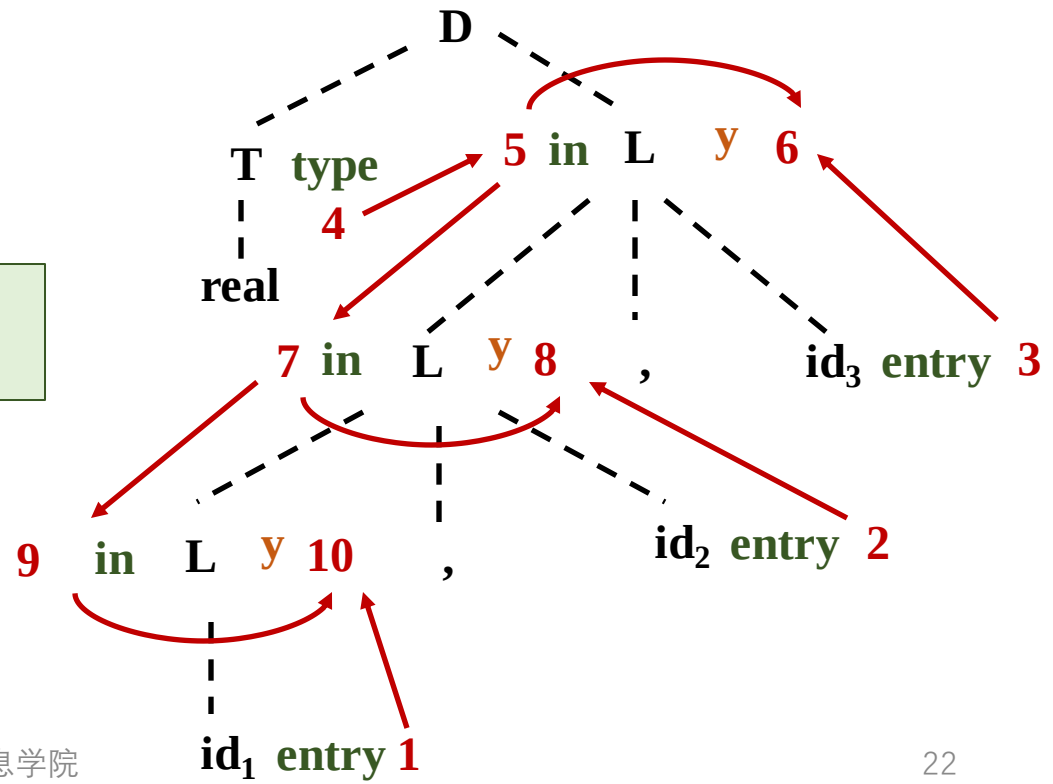
产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
$L \rightarrow L_1$, id	L_1.in := L.in
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



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

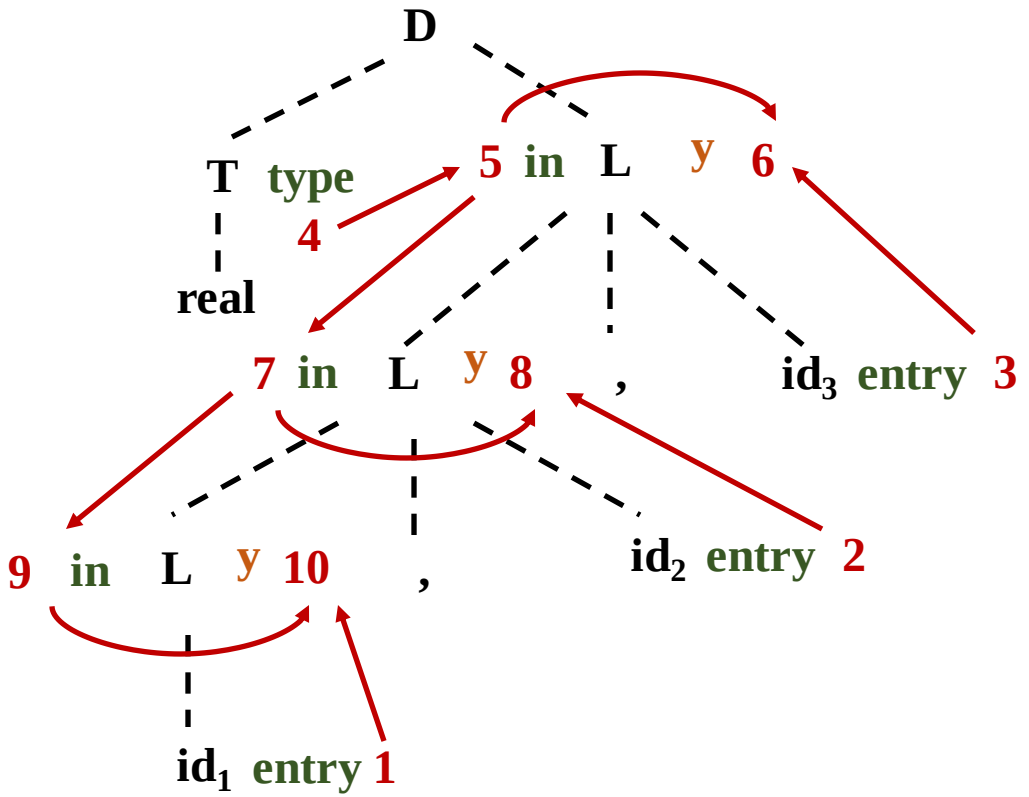
一个属性对另一个属性的循环依赖关系。
如，p、c1、c2都是属性，若p: = f1(c1)、c1:= f2(c2)、c2: = f3(p)时，就无法对p求值。



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*/
```

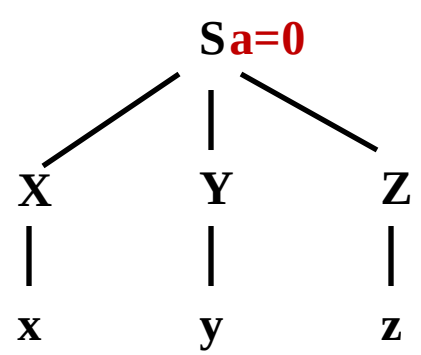
最坏的情况：计算复杂度$O(n^2)$
n为结点个数

```
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
```

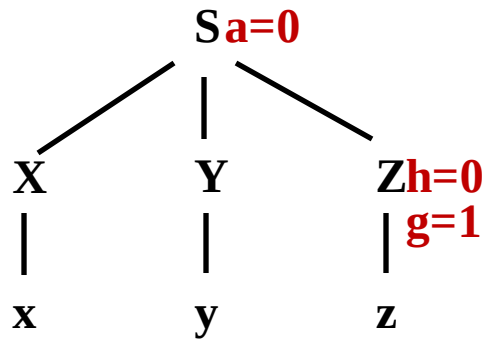
树遍历的属性计算方法

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

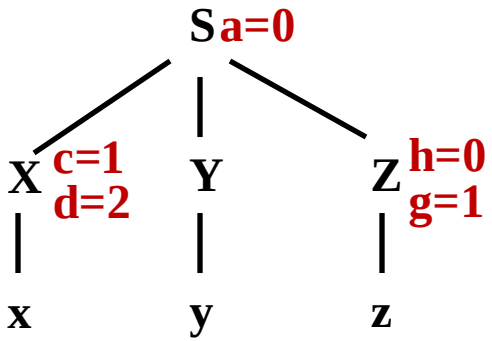
S有继承属性a，综合属性b
X有继承属性c，综合属性d
Y有继承属性e，综合属性f
Z有继承属性h，综合属性g
假设S.a的初始值为0



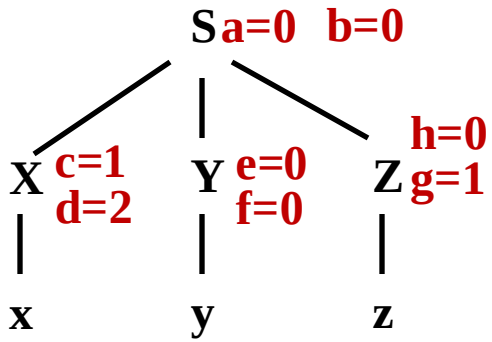
初始状态



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，
  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
- L – 属性文法的自顶向下翻译
- S – 属性文法的自下而上计算
- 自下而上计算继承属性

L属性文法定义

- 在语法分析过程中进行翻译时，属性的计算顺序将与分析方法建立分析树节点的顺序相关。有一种能够描述许多自顶向下和自底向上翻译方法的自然顺序——深度优先顺序。
- L属性定义
  - 一个属性文法是L-属性文法，如果对于每个产生式 A→X1X2…Xn，其每个语义规则中的每个属性，或者是综合属性，或者是X_j（1≤j≤n）的一个继承属性且这个继承属性仅依赖于：
  - （1）产生式X_j在左边符号X1，X2，…，X_{j-1}的属性；
  - （2）A的继承属性。
- L属性的计算
  - 可以使用深度优先顺序来计算
  - LL(1)的分析过程，可以看成是深度优先建立语法树的过程，可以在自上而下语法分析的同时实现L属性文法的计算

S-属性文法一定是L-属性文法

L属性文法的正/反例

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

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

Q.i 依赖于右边的R.s

语法制导翻译(基于翻译模式)

- 语法制导翻译给出了使用语义规则进行计算的次序，这样就可以把某些细节表示出来。
- 翻译模式：用于语法制导翻译的一种描述形式。
  - 语义动作用花括号{ }括起来，插入到产生式右部的合适位置上（指示使用语义规则的计算次序）

例如：中缀表达式翻译为后缀表达式

LL分析:

E→TR

R→addop T {print(addop. Lexeme)} R1|ε

T→num {print(num.val)}

语法制导翻译举例(基于翻译模式)

- E→TR
- R→addop T {print(addop.lexeme)} R_1 | ε
- T→num {print(num.val)}

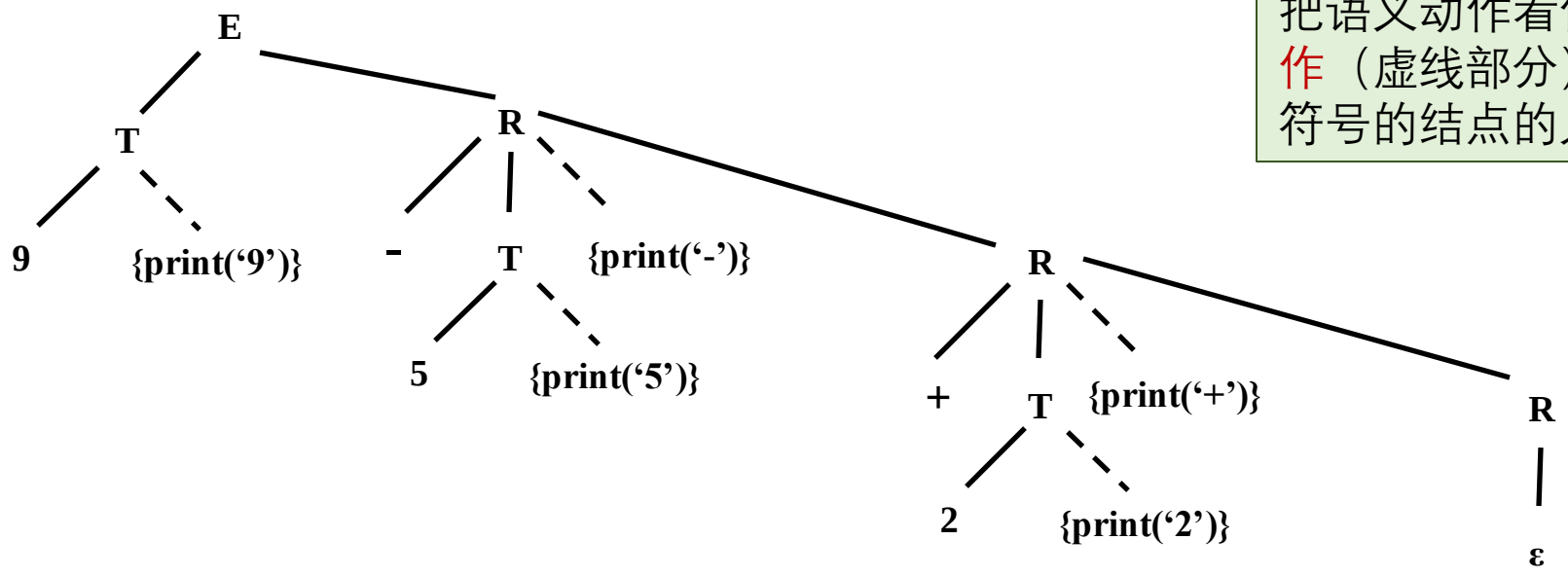
9−5+2

按深度优先遍历之后

95−2+

实线部分是输入串9-5+2的语法分析树

把语义动作看作是终结符,每个语义动作（虚线部分）作为相应产生式左部符号的结点的儿子



L-属性定义的语法制导翻译

- 设计L-属性定义的语法制导翻译需要注意以下几点：
  - 基本设计原则：当某个动作引用一个属性时，这个属性是可用的。也就是说，一个动作不会引用一个没有计算出来的属性。
  - 只有综合属性时
    - 为每一个语义规则建立一个包含赋值的动作，并把该动作放在产生式右部的末尾
    - $T \rightarrow T_1 * F$ $\{T.val := T_1.val \times F.val\}$
  - 同时存在综合属性和继承属性时：

    （1）产生式右部符号的继承属性必须在这个符号以前的动作中计算出来

    （2）一个动作不能引用该动作右部符号的综合属性

    （3）产生式左部非终结符的综合属性只有在其引用的所有属性值都计算出来以后才能计算。计算该属性的动作通常放在产生式右部的末尾。

例子：下面的翻译模式不符合该定义(1)：

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

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

按深度优先遍历时，要打印第二个产生式里的继承属性A.in时，该属性还没有被定义

L-属性文法及翻译模式举例

产生式	语义规则
S→B	B.ps := 10
	S.ht := B.ht
B→B_1B_2	B_1.ps := B.ps
	B_2.ps := B.ps
	B.ht := max(B_1.ht, B_2.ht)
B→ 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→text	B.ht := text.h ×B.ps

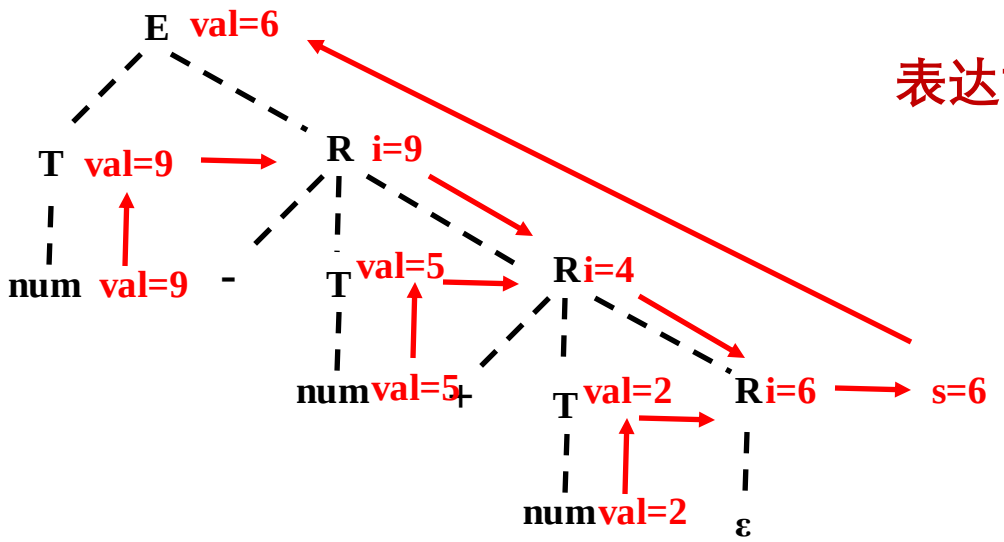
S→{B.ps := 10}
B
{S.ht := B.ht}
B→{B_1.ps := B.ps}
B_1
{B_2.ps := B.ps}
B_2
{B.ht := max(B_1.ht, B_2.ht)}
B→ {B_1.ps := B.ps}
B_1
sub
{B_2.ps := shrink(B.ps)}
B_2
{B.ht := disp(B_1.ht, B_2.ht)}
B→text
{B.ht := text.h ×B.ps}

其中:
.ps为继承属性
.ht为综合属性

例子

$E \rightarrow E_1+T$ {E.val := E_1.val + T.val}
$E \rightarrow E_1-T$ {E.val := E_1.val - T.val}
$E \rightarrow T$ {E.val := T.val}
$T \rightarrow (E)$ {T.val := E.val}
$T \rightarrow num$ {T.val := num.val}

$E \rightarrow T$ {R.i := T.val}
R {E.val := R.s}
$R \rightarrow +$
T {R_1.i := R.i + T.val}
R_1 {R.s := R_1.s}
$R \rightarrow -$
T {R_1.i := R.i - T.val}
R_1 {R.s := R_1.s}
$R \rightarrow \varepsilon$ {R.s := R.i}
$T \rightarrow ($
E
) {T.val := E.val}
$T \rightarrow num$ {T.val := num.val}



表达式9-5+2

一个符号继承属性必须由出现这个符号之前的动作来计算，产生式左边非终结符的综合属性必须在它所依赖的所有属性都计算出来之后才能计算

Review: 递归下降分析程序的构造

- 当文法满足**LL(1)** 条件时，我们就可以为它构造一个不带回溯的自顶向下分析程序，这个分析程序是由一组递归过程组成的。每个过程对应文法的一个非终结符。这样的一个分析程序称为递归下降分析器。

LL(1) 条件

- 文法不含左递归
- 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。即，若$A\rightarrow\alpha_1|\alpha_2|\cdots|\alpha_n$，则
  $FIRST(\alpha_i)\cap FIRST(\alpha_j)=\Phi \quad (i\neq j)$
- 对文法中的每个非终结符A，若它存在某个候选首符集包含ε，则，$FIRST(A)\cap FOLLOW(A)=\Phi$

Review:递归下降分析程序构造

E→T{+T}

```
PROCEDURE E;
BEGIN
   T;
   WHILE SYM = '+' DO
   BEGIN
      ADVANCE; T
   END
END
```

T→F{*F}

```
PROCEDURE T;
BEGIN
   F;
   WHILE SYM = '*' DO
   BEGIN
      ADVANCE; F
   END
END
```

F→(E) | i

```
PROCEDURE F;
   IF SYM ='i' THEN ADVANCE;
   ELSE
      IF SYM='(' THEN
         BEGIN
               ADVANCE
               E;
               IF SYM=')' THEN ADVANCE;
               ELSE ERROR;
         END
      ELSE ERROR;
```

递归下降翻译器的设计

- 为每个非终结符A构造一个函数
  - A的每个继承属性均对应该函数的一个形式参数，其返回值为A的综合属性的值(可能是一个记录、一个指针或者使用传地址参数的传递机制)，为出现在A的产生式中的每个文法符号的每个属性设置一个局部变量
  - 非终结符A的代码会根据当前的输入决定使用哪个产生式
  - 与每个产生式有关的代码执行如下动作：从左到右考虑产生式右部的记号、非终结符及语义动作
    - 对于带有综合属性x的终结符X，把x的值保存在X.x中，然后产生一个匹配X的调用，并继续输入
    - 对于非终结符B，产生一个右部带有函数调用的赋值语句$c=B(b_1,b_2,...b_k)$，其中$b_1,b_2,...b_k$是代表B的继承属性变量，c是代表B的综合属性的变量
    - 对于每个动作，将其代码复制到语法分析器，并把对属性的引用改为对相应变量的引用

例子

E→T　{R.i := T.val}
**　R　{E.val := R.s}**
R→+
**　T　{R_1.i := R.i + T.val}**
**　R_1　{R.s := R_1.s}**
R→-
**　T　{R_1.i := R.i - T.val}**
**　R_1　{R.s := R_1.s}**
R→ε　{R.s := R.i}
T→(
**　E**
**　)　{T.val := E.val}**
T→num　{T.val := num.val}

构造计算器的翻译模式

E→T　{R.i := T.nptr}
**　R　{E. nptr := R.s}**
R→+
**　T　{R_1.i := mknode('+', R.i , T. nptr)}**
**　R_1　{R.s := R_1.s}**
R→-
**　T　{R_1.i := mknode('-', R.i , T. nptr)}**
**　R_1　{R.s := R_1.s}**
R→ε　{R.s := R.i}
T→(
**　E**
**　)　{T.nptr := E.nptr}**
T→num　{T.nptr := mkleaf(num, num.val)}
T→id　{T.nptr := mkleaf(id, id.entry)}

构造抽象语法树的翻译模式

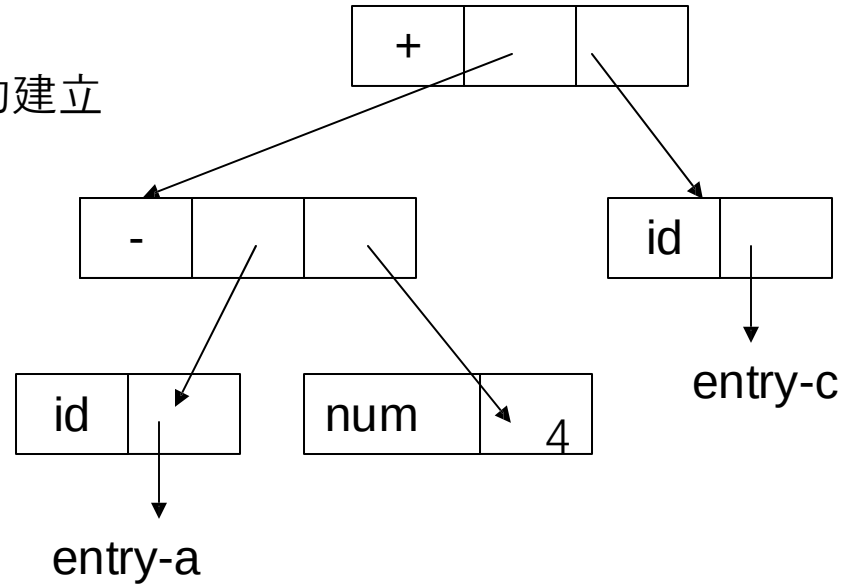
抽象语法树的构造

- 用抽象语法树作为中间表示，可以把翻译从语法分析中分离出来
  - 抽象语法树是语法分析树的压缩形式，去掉了那些对翻译不必要的信息，对表示语言的结构很有用。
- 表达式的抽象语法树的构造
  - 为每个运算符和运算对象建立结点来为子表达式构造子树。
  - 运算符结点的子结点分别是表示该运算符各运算分量的子表达式组成的子树的根
- 构造的过程（方法）
  - mknode(op, left, right)：建立一个标记为op的运算符节点，其中两个域left和right是指向其左右运算分量的指针
  - mkleaf(id,entry)：建立标记为id的标识符节点，entry是指向该标识符在标识符表中的相应表项的指针
  - mkleaf(num,value)：建立标记为num的数节点，域value保存该数的值。

例：a-4+c抽象语语法树的建立

```
p1:=mkleaf(id,entry-a);
P2:=mkleaf(num,4);

P3:=mknode('-',p1,p2);
P4:=mkleaf(id,entry-c);
P5:=mknode('+',p3,p4);
```



例子1

E→T　{R.i := T.nptr}
**　　R　{E. nptr := R.s}**
R→+
**　　T　{R_1.i := mknode('+', R.i , T. nptr)}**
**　　R_1　{R.s := R_1.s}**
R→-
**　　T　{R_1.i := mknode('-', R.i , T. nptr)}**
**　　R_1　{R.s := R_1.s}**
R→ε　{R.s := R.i}
T→(
**　　E**
**　　)　{T.nptr := E.nptr}**
T→num　{T.nptr := mkleaf(num, num.val)}
T→id　{T.nptr := mkleaf(id, id.entry)}

```
Function T: *AST-node;
```

```
Function R (in: *AST-node): *AST-node;
  var nptr, i1, s1, s: *AST-node;
  oplexeme: char;
  begin
      if sym='op' then begin
          oplexeme:=lexval;
          advance;
          nptr:=T;
          i1:=mknode(oplexeme, in, nptr);
          s1:=R (i1);
          s:=s1;
      end
      else s:=in;
      return s;
  end
```

例子2

1. $<S>_{\downarrow R}$ —> $a\uparrow_T <A>\uparrow_Q @ X_{\downarrow T,R} <S>_{\downarrow Q}$
2. $<S>\downarrow_R$ —> $b @ z \downarrow_R$

全局变量和过程声明：

CLASS; / * 存放单词类别码 * /

TOKEN; / * 存放单词值 */

NEXTSYM; / *词法分析程序，每调用一次单词类别码 ⇒ CLASS,

单词值 ⇒ TOKEN,该符号指针指向下一个单词 */

主程序：

```
int main(){
        nextsym();
        if (class == 'a' || class == 'b')
                proc_S(7);
        if (class != '#')
                error();
        return 0;
}
```

例子2

1. $<S>_{\downarrow R}$ —> $a\uparrow_T <A>\uparrow_Q @ X\downarrow_{T,R} <S>_{\downarrow Q}$
2. $<S>\downarrow_R$ —> $b @ z \downarrow_R$

过程 proc_S(int R)

```
void proc_S(int R)
{
        if (class == 'a'){
                int T, Q;       /*  属性变量作为局部变量申明  */
                T = token;/ * 将单词值赋给终结符的综合属性 * /
                nextsym();      /*  读输入符号    */
                proc_A(&Q);     /*  调用A的分析程序，将返回一个值，存于Q（综合属性）中    */
                out('x', T);;   /*  调用输出函数   */
                proc_S(Q);      /*  调用S的分析程序，实参为Q   */
        }
        else if (class = 'b'){
                nextsym();      /*  读输入符号    */
                out('z', R);;   /*  调用输出函数   */
                }
                else{
                        error ();  }}
```

例子2

3. $<A>\uparrow_P$ —> $C\uparrow_u$ @ $y\downarrow_u <A>\uparrow_Q <S>\downarrow_z$ @ $v\downarrow_P$ b　　P := Q + U, z := U – 3

4. $<A>\uparrow_P$ —> @ w　　P:=8

过程 proc_A(int *P)

```
void proc_A(int *P){
if (class == 'c'){
                    int U,Q,Z;     /* 属性变量申明          */
                    U = token;/ * 单词值赋给终结符的综合属性 * /
                    nextsym();     /*  读输入符号           */
                    Z = U - 3;     /*  利用求值规则求出Z的值  */
                    out('y', U);   /*  调用输出函数         */
                    proc_A(&Q);          /*  调用A的分析程序，将返回一个值，存于Q（综合属性）中    */
                    *P = Q + U;    /*  利用求值规则求出*P的值  */
                    proc_S(Z);     /*  调用S的分析程序，实参为Z  */
                    out('v', *P);;                  /*  调用输出函数  */
                    if (class != 'b') {   /*  匹配符号’ b’ */
                       error();}
                    nextsym(); }     /*  预读一个输入符号   */
else{
                    *P = 8;        /*  根据求值规则求值  */
                    out('w', *P);;         /*  调用输出函数  */
                    return;}}
```

练习

- 下列文法由开始符号S产生一个二进制数，令综合属性val给出该数的值：
  - S→L.L | L
  - L→LB | B
  - B→0 | 1
- 试设计求S.val的属性文法，其中，已知B的综合属性c，给出由B产生的二进位的结果值。
- 写出语法制导翻译过程的伪代码

Outline

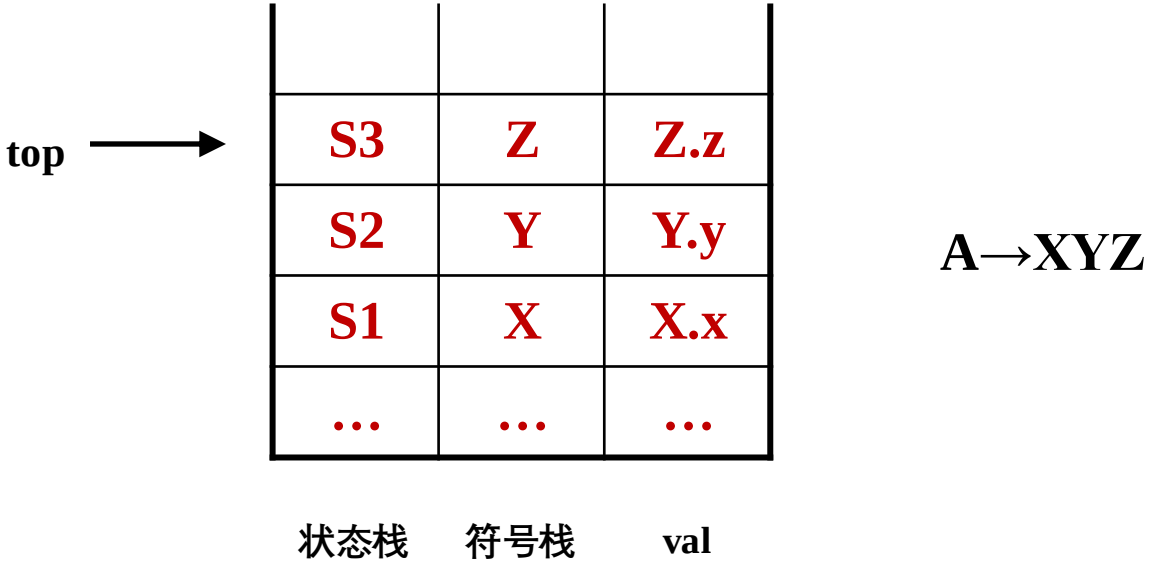
- 属性文法和语法制导翻译
- 基于属性文法的处理方法
- L – 属性文法的自顶向下翻译
- S – 属性文法的自下而上计算
- 自下而上计算继承属性

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属性文法的翻译例子

例：对二目运算符的运算对象进行类型匹配审查

G[E]:
(1) E →T+T
(2) E →T or T
(3) T →n
(4) T →b

$E \rightarrow T^1 + T^2$
{ if T^1.type = int and T^2.type= int
then E.type :=int
else error}

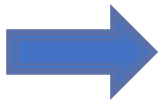
$E \rightarrow T^1$ or T^2
{ if T^1.type = bool and T^2.type= bool
then E.type :=bool
Else error}

T → n { T.type :=int}

T → b { T.type := bool}

S-属性文法计算举例

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



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	

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

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。

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
$E \rightarrow E_1 + T$	val[ntop] := val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntop] := val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] := val[top-1]
$F \rightarrow digit$	

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

S-属性文法练习

2 + 3 * 5 的分析和计值过程

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

动作	状态栈	语义栈(值栈)	符号栈	余留输入串
	0	-	#	2+3*5#
移进	05	-2	#2	+3*5#
规约	03	-2	#F	+3*5#
规约	02	-2	#T	+3*5#
规约	01	-2	#E	+3*5#
移进	016	-2-	#E+	3*5#
移进	0165	-2-3	#E+3	*5#
规约	0163	-2-3	#E+F	*5#
规约	0169	-2-3	#E+T	*5#
移进	01697	-2-3-	#E+T*	5#
移进	016975	-2-3-5	#E+T*5	#
规约	0169710	-2-3-5	#E+T*F	#
规约	0169	-2-15	#E+T	#
规约	01	-17	#E	#
接受	01	-17	#E	#

S-属性文法练习

产生式	语义规则
0） L→E	print（E.val）
1） E→E¹＋T	E.val:＝E¹.val＋T.val
2） E→T	E.val:＝T.val
3） T→T¹*F	T.val:＝T¹.val×F.val
4） T→F	T.val:＝F.val
5） F→（E）	F.val:＝E.val
6） F→digit	F.val:＝digit.lexval

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			

Outline

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

SL-ATG

简单赋值形式的L_属性翻译文法(SL-ATG)

一般属性值计算：　　x:=f (y , z)

SL-ATG属性值计算:　x:=某符号的属性值或常量。

例　x:=y,　　x,y,z:=17 —— 称为复写规则

为了实现上的方便，常希望文法符号的属性求值规则为上述简单形式的。为此，对现有的L-ATG的定义做一点改变，从而形成一个称为简单赋值形式的L-ATG。

SL-ATG

•定义7.4 一个L-ATG被定义为简单赋值形式的(SL-ATG)，当且仅当满足如下条件:

1. 产生式右部符号的继承属性是一个常量，它等于左部符号的继承属性值或等于出现在所给符号左边符号的一个综合属性值。

2.产生式左部非终结符号的综合属性是一个常量，它等于左部符号的继承属性值或等于右部符号的综合属性值。

因此，一个简单赋值形式的L-ATG除动作符号外，其余符号的属性求值规则其右部是属性或是常量。

- L-ATG ⇒ SL-ATG

给定一个L-ATG，如何找一个等价的赋值形式的L-ATG?

考虑产生式: $<A> \longrightarrow a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I}$,　　$I := f(R, S)$ 显然，该属性求值规则不是简单赋值形式的，因为它需要对f求值。

第一步：设动作符号“@ f”表示函数f求值，该动作符号有两个继承属性和一个综合属性。

$@f_{\downarrow I_1, I_2 \uparrow S_1}$　　且 $S_1 := f(I_1, I_2)$

第二步：修改产生式

1. 插入“@ f”（在适当位置）
2. 引进新的复写规则（将R , S 赋给I_1和I_2, f值赋给S_1 ）
3. 删去原有包含f 的规则

$<A>\rightarrow @f_{\downarrow I_1, I_2 \uparrow S_1}\ a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I},$

$I_1 := R,\quad I_2 := S,\quad S_1 := f(I_1, I_2),\quad I := S_1$

$<A>\rightarrow a_{\uparrow R} @f_{\downarrow I_1, I_2 \uparrow S_1} <B>_{\uparrow S} <C>_{\downarrow I},$

$I_1 := R,\quad I_2 := S,\quad S_1 := f(I_1, I_2),\quad I := S_1$

$<A>\rightarrow a_{\uparrow R} <B>_{\uparrow S} @f_{\downarrow I_1, I_2 \uparrow S_1} <C>_{\downarrow I},$

$I_1 := R,\quad I_2 := S,\quad S_1 := f(I_1, I_2),\quad I := S_1.$

$<A>\rightarrow a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I} @f_{\downarrow I_1, I_2 \uparrow S_1},$

$I_1 := R,\quad I_2 := S,\quad S_1 := f(I_1, I_2),\quad I := S_1..$

（2）右部符号的继承属性值，用该产生式左部符号的继承属性 或出现在该符号左部的符号的属性值进行计算。

（3）动作符号的综合属性用该符号的继承属性或某个右部符号的属性进行计算。

$$<A> \longrightarrow a_{\uparrow R} <B>_{\uparrow S} <C>_{\downarrow I}, \qquad I := f(R, S)$$

$$<A> \longrightarrow a_{\uparrow R} <B>_{\uparrow S} @ f_{\downarrow I_1, I_2 \uparrow S_1} <C>_{\downarrow I},$$

$$I_1 := R, \quad I_2 := S, \quad S_1 := f(I_1, I_2), \quad I := S_1.$$

该文法是简单赋值形式的L-ATG.

注意： 无参函数过程作为常数处理，如

$$<A> \longrightarrow <B>_{\uparrow x} <C>_{\uparrow y} \qquad x, y := NEWT$$

自底向上计算继承属性

- 删除嵌入在翻译模式中的动作
  - 在自顶向下分析中我们可以在产生式右部的任何地方嵌入动作
  - 在自底向上翻译方法中，需要把所有的翻译动作都放在产生式右部的末尾
  - 在基础文法中加入新的形如M→ ε 的产生式，其中M为标记非终结符。将每个嵌入动作用不同的标记非终结符M来代替，并把该动作放在此空产生式的末端

例如 E→TR
R→+T {print('+')} R | -T {print('-')} R | ε
T →num {print(num.val)}

转化为

E→TR
R→+TMR | -TNR | ε
M→ ε {print('+')}
N→ ε {print('-')}
T →num {print(num.val)}

转换后的语法制导翻译和原语法制导翻译比较：

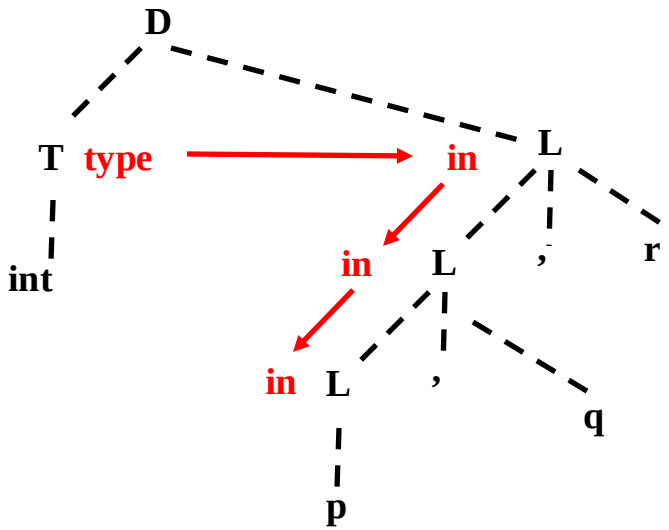
- 用额外的节点表示动作，但动作的执行顺序是一样的
- 转换后的翻译模式中，动作都在产生式的末尾，可以在自底向上的分析过程中刚好在产生式右部被归约之前执行

分析栈中的继承属性

- 对于继承属性是由复写规则定义的产生式
  - 自底向上语法分析器对产生式A→XY的归约就是从分析栈顶移走X和Y并用A来代替它们。假设X有一个综合属性X.s。
  - X的综合属性在分析中放入属性栈，和状态栈符号栈是一一对应的。
  - X.s在Y以下的子树的任何归约之前已经放在栈中，这个值可以被Y继承，也就是说，如果继承属性Y.i是由复写规则Y.i := X.s定义，那么在需要Y.i的地方可以使用X.s的值。

例子

int p,q,r



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

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)}

输入串	栈	所用产生式
int p,q,r	-	
p,q,r	int	
p,q,r	T	T→int
,q,r	Tp	
,q,r	TL	L→id
q,r	TL,	
,r	TL,q	
,r	TL	L→L,id
r	TL,	
	TL,r	
	TL	L→L,id
	D	D→TL

模拟继承属性的计算

- 使用自底向上的方法计算继承属性，必须要可以预测属性值在栈中的位置。
- 但并非总是如此：

产生式	语义规则
S→aAC	C.i := A.s
S→bABC	C.i := A.s
C→c	C.s :=g(C.i)

在通过C→c进行归约时，C.i（即A.s）的值可能在val[top-1]也可能在val[top-2]处，无法确定具体是哪个位置

产生式	语义规则
S→aAC	C.i := A.s
S→bABMC	M.i := A.s; C.i := M.s
C→c	C.s := g(C.i)
M→ε	M.s := M.i

修改后当应用C→c归约时，C.i（或者是A.s，或者是M.s）的值可以在val[top-1]处找到

模拟继承属性的计算

- 模拟由复写规则计算的继承属性
  - 引入一个新的标记非终结符M，用M的继承属性和综合属性来传递后面非终结符需要复写的继承属性。
- 模拟不是复写规则的语义规则（计算函数）
  - 也可以加入新的标记非终结符，用复写规则继承前面非终结符的属性值，其综合属性被置为用计算函数进行计算

产生式	语义规则
S→aAC	C.i:=f(A.s)

产生式	语义规则
S→aANC	N.i:=A.s; C.i:=f(N.s);
N→ε	N.s:=N.i