第7章　语法制导的语义计算

语义处理及形式语义概述

语言的语义和编译的语义处理

静态语义：语法规则的良形式条件
静态语义检查：**审查静态语义**

动态语义：程序单元执行的操作
动态语义处理：**生成中间(目标)代码**

编译的语义处理方式

- 由语法分析子程序直接调用相应的语义子程序进行语义处理
- 先生成语法树或此结构的某种表示，再进行语义处理

编译的语义处理方法

- **直接生成目标代码**；
- 采用**中间代码**：逻辑上简单明确，与机器相关的实现细节置于代码生成阶段处理，易于优化。

语义处理概述

高级程序设计语言的语义：

静态语义 是对程序约束的描述，这些约束无法通过抽象语法规则来妥善地描述，实质上就是语法规则的良形式条件，它可以分为类型规则和作用域/可见性规则两大类

动态语义 程序单位描述的计算

编译程序的语义处理工作

静态语义审查

动态语义翻译

静态语义审查

（1）**类型检查。**根据类型相容性要求，验证程序中执行的每个操作是否遵守语言的类型系统的过程，编译程序必须报告不符合类型系统的信息。

（2）**控制流检查。**控制流语句必须使控制转移到合法的地方。例如，在C语言中break语句使控制跳离包括该语句的最小while、for或switch语句。如果不存在包括它的这样的语句，则就报错。

（3）**一致性检查。**在很多场合要求对象只能被定义一次。例如Pascal语言规定同一标识符在一个分程序中只能被说明一次，同一case语句的标号不能相同，枚举类型的元素不能重复出现等等。

（4）**上下文相关性检查。**比如，变量名字必须先声明后引用；而有时，同一名字必须出现两次或多次，例如，Ada语言程序中，循环或程序块可以有一个名字，出现在这些结构的开头和结尾，编译程序必须检查这两个地方用的名字是相同的。

（5）**名字的作用域分析。**

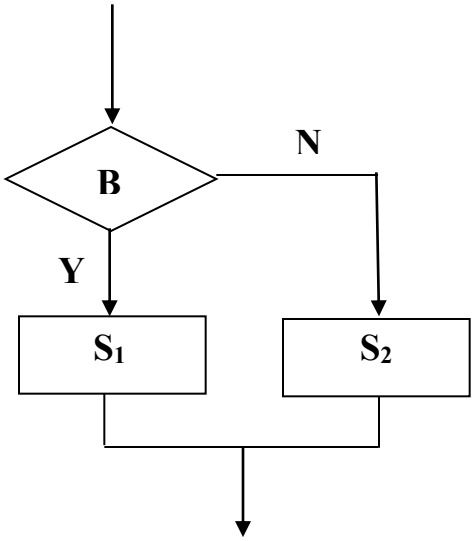
静态语义分析环境

符号表

为语义分析提供类型、作用域等信息。

为代码生成提供类型、作用域、存储类别、存储（相对）位置等信息。

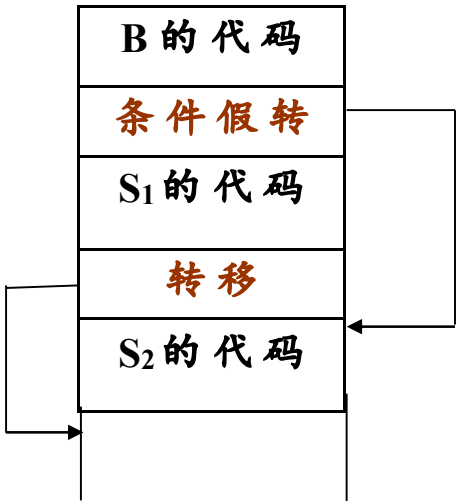
如 if B then S1 else S_2



解释执行动态语义

生成代码：

中间代码或目标代码



语义处理描述

属性文法：附带语义规则的文法。

语法制导的翻译：在语法分析的同时，执行语义子程序。

检查静态语义

翻译(生成)中间(目标)代码

例子

P→D;E

D→D;D | id:T

T→char | integer | array [num]of T | ↑T

E→literal | num | id | E mod E | E [E] | E↑

P代表程序；D代表说明；E代表表达式。

如程序语句：

key: integer;

String:char;

key mod 1999

语言本身提供两种基本类型：char和integer。

除此之外还有缺省的基本类型type_ error和void。假定所有数组下标都从1开始。

确定标识符类型的语义描述

（1）P→D;E

（2）D→D;D

（3）D→id:T　　{addtype (id. Entry, T. type)}

（4）T→char　　{T. Type:= char}

（5）T→integer　{T. Type:= integer}

（6）T→↑T_1　　{T. Type:= pointer (T_1. type)}

（7）T→array [num]of T_1

{T. Type:= array (num.Val, T_1.type)}

类型检查的语义描述

S→id:=E	{if id.Type = E.Type then S.Type := void else S.Type := type_ error}
S→if E then S_1	{if E.type = boolean then S.Type := S_1.type else S. type := type_ error}
S→while E do S_1	{if E. type = boolean then S. type := S_1. Type else S. type := type_ error}

语义处理的实现工具

Yacc(Bison)对语法制导翻译的支持;

Yacc(Bison)对语义值的支持

通过$伪变量访问文法符号的语义值。

Yacc(Bison)对语义动作的支持

在语法规则的动作中调用语义子程序：计算语义值，诊断语义错误，生成中间代码等。

```
Variable: Type T_Identifier
            { $$ = CreateVariableDeclaration(&@1, $1, $2); };
Type:       T_Int { $$ = Type::intType; } ;
```

语义形式化—语义模型

文法模型 --- **属性文法**

命令式或操作式模型 --- 操作语义学

应用式模型 --- 指称语义学

公理式模型 --- 公理语义学

规格说明模型 --- 代数数据类型

属性文法

表达式文法 E—>T+T | T or T
T—>n | b

$E \rightarrow T^1 + T^2$
{if (T^1.type = int)AND(T^2.type= T^1.type)
then E.type =int }

$E \rightarrow T^1$ or T^2
{if (T^1.type=bool)AND(T^2.type=T^1.type)
then E.type :=bool }

$T \rightarrow n$ {T.type := int }

$T \rightarrow b$ {T.type := bool}

操作语义

描述一段程序的含义是通过执行该段程序所改变的计算机（无论是真实计算机还是虚拟计算机）状态来反映的。计算机里所有的寄存器的值和存储单元的值视为计算机的状态。

```
For (expr1;expr2;expr3){
        ...
}
```

```
expr1;
Loop: if expr2=0 goto  out
   …
expr3;
goto loop
out:  ...
```

公理语义

公理语义的概念是随着程序正确性的证明而发展的。当正确性证明能构造时表明程序执行它的规格说明所描述的计算。在一个证明中，每一个语句之前之后都有一个逻辑表达式对程序的变量进行约束,以此说明这个语句的含义。

断言{P}S{Q}称为S关于(P, Q)的正确性断言。

$\{x>3\}\ x=x-3\ \{x>0\},\ (x>5)\Rightarrow(x>3),\ \{x>0\}\Rightarrow\{x>0\}$

$\{x>5\}\quad x=x-3\quad \{x>0\}$

While 循环

$(I\ \text{and}\ B)\ S\ \{I\}$

$\{I\}\ \text{while}\ B\ \text{do}\ S\ \text{end}\ \{I\ \text{and}\ (\text{not}\ B)\}$

I是循环不变式。

if--then--else

$\{B\ \underline{\text{and}}\ P\}\ S_1\ \{Q\},\ \{(\underline{\text{not}}\ B)\ \underline{\text{and}}\ P\}\ S_2\ \{Q\}$

$\{P\}\quad \underline{\text{if}}\ B\quad \underline{\text{then}}\quad S_1\quad \underline{\text{else}}\quad S2\{Q\}$

指称语义

指称语义的基本概念是给每一段程序实体定义一个数学意义上的对象，和一个从实体向数学意义上的对象的映射的函数。

语义函数：将短语映射到它的指称。

特点：不但对整个程序赋予含义而且对程序设计语言的每一个语法成分短语（表达式，命令，声明…）都给予含义。

每一个语法成分（短语）的含义是以它的构成成分的含义的术语来定义的。

即语义结构平行于语法结构。

例如：

二进制数语言　　110或10101　　语法实体

指称（十进制数）　　6　或　21　　语义实体

二进制数文法　　Numeral::=0

::=1

::= Numeral 0

::=Numeral 1

自然数　　Natrual={0,1,2,3,…}

语义函数　　Valuation:Numeral→Natural

Valuation[101]　表示把Valution施用于101

Valuation[N]　表示把Valution施用于N

定义：Valuation(用四个方程)因为有四个形式numeral

Valuation[0] ≡0

Valuation[1] ≡1

Valuation[N0] ≡2×Valution [N]

Valuation[N1] ≡2×Valution [N]+1

例子：

Valuation[110]= 2 × Valuation[11]

= 2 × (2 × Valuation[1]+1)

= 2 × (2 × 1+1)

= 6

规格说明模型---代数数据类型

描述实现一个程序的各种函数间的关系。

如果证明一个实现服从任何两个函数间的这种关系，则可以声明这个实现是此规格说明的正确实现。

7.1 属性文法

属性文法A(attribute grammar)是一个三元组:

A=(G, V, F), 其中

G: 是一个上下文无关文法;

V: 有穷的属性集,每个属性与文法的一终结符或非终结符相连;

F: 关于属性的属性断言或谓词集. 每个断言与一个产生式相联, 而此断言只引用该产生式左端或右端的终结符或非终结符相联的属性.

属性文法: 允许为每个终结符和非终结符配备一些属性, 并将描述这些属性的语义规则附加在相应的文法规则中. 它既能描述程序设计语言的语法, 又为其**语义描述**提供了手段.

属性文法由D. E. Knuth于1968年引进, 后来才被用于编译程序的设计.

属性有不同的类型, 可以象变量一样地被赋值.

赋值规则附加于语法规则之上. 赋值与语法分析同时进行, 赋值过程就是语义处理过程.

在推导语法树的时候, 诸**属性的值被计算**并通过赋值规则层层传递. 有的从语法规则左边向右边传,有的从右边向左边传.

语法推导树最后完成时, 就得到开始符号的属性值, 也就是整个**程序的语义**.

属性文法中的**语义规则（动作）**和产生式相联系的方式有两种：**语法制导的定义**和**翻译模式/翻译方案**（Translation Schemes）。

语法制导的定义是较抽象的翻译，隐蔽了一些实现细节；

而**翻译方案**描述了一些实现细节，主要指名了语义规则的计算次序。

例如：定义表达式的文法如下：

E→E+E

E→(E)

E→n

给出定义表达式值的属性文法。

为文法符号**E**引进属性符号**val**，用**E.val**表示**E**的值，属性计算规则以赋值语句的形式给出，附在每个产生式后，并用大括号括起来。为了明确**E**的不同出现位置，用上角标区别。终结符**n**的值是词法分析程序提供的，这里用**n.lex**表示。**属性文法**如下：

$E \rightarrow E^1+E^2$ {**E.val := E^1.val +E^2.val**}

$E \rightarrow (E^1)$ {**E.val := E^1.val**}

$E \rightarrow n$ {**E.val := n.lex**}

属性文法的主要思想有两点：

首先，对于每个文法符号引进相关的属性符号；

其次，对于每个产生式写出属性值计算的规则。

例：$E \rightarrow T^1 + T^2 \mid T^1 \text{ or } T^2$

$T \rightarrow num \mid ture \mid false$

给出类型检查的属性文法。

$E \rightarrow T^1 + T^2$ $\{T^1.t = int \text{ AND } T^2.t = int\}$

$E \rightarrow T^1 \text{ or } T^2$ $\{T^1.t = bool \text{ AND } T^2.t = bool\}$

$T \rightarrow num$ $\{T.t := int\}$

$T \rightarrow ture$ $\{T.t := bool\}$

$T \rightarrow false$ $\{T.t := bool\}$

属性文法中，对应每个产生式A →α都有一套与之相关联的语义规则，每条规则的形式为 $b:=f(c_1, c_2, \ldots, c_k)$。其中，$f$ 是一个函数，b和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性。

(1) 如果b是A的一个属性，$c_1, c_2, \ldots, c_k$是产生式右部文法符号的属性或A的其它属性，则称b是A的**综合属性**(synthesized attribute)。

(2) 如果b是产生式右部某个文法符号X的一个属性，并且$c_1, c_2, \ldots, c_k$是A或产生式右边任何文法符号的属性，则称b是文法符号X的**继承属性**(Inherited attribute)。

属性b依赖于属性$c_1, c_2, \ldots, c_k$。

注意：

(1) 非终结符既可以有综合属性也可以有继承属性，但文法开始符号没有继承属性。

(2) 终结符只有综合属性，属性值由词法分析程序提供。

(3) 继承属性的计算规则自顶向下，综合属性的计算规则自底向上。

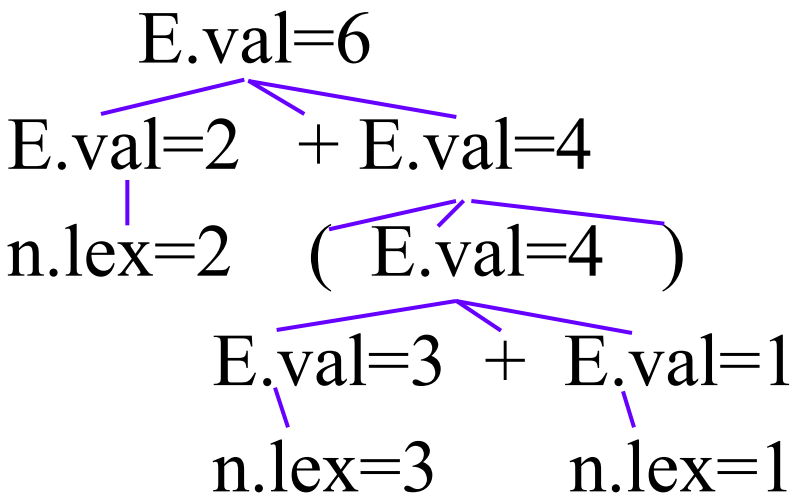
例如，定义表达式值的属性文法， E.val是一个综合属性的例子：

$E \rightarrow E^1+E^2$ {E.val := E^1.val +E^2.val}

$E \rightarrow (E^1)$ {E.val := E^1.val }

$E \rightarrow n$ {E.val := n.lex}

句子2 + （ 3 + 1 ）的求值顺序，将2 + （ 3 + 1 ）的分析树结点改为有附加属性的结点（带注释的分析树）：



例：简单算术表达式求值的语义描述（语法制导定义）

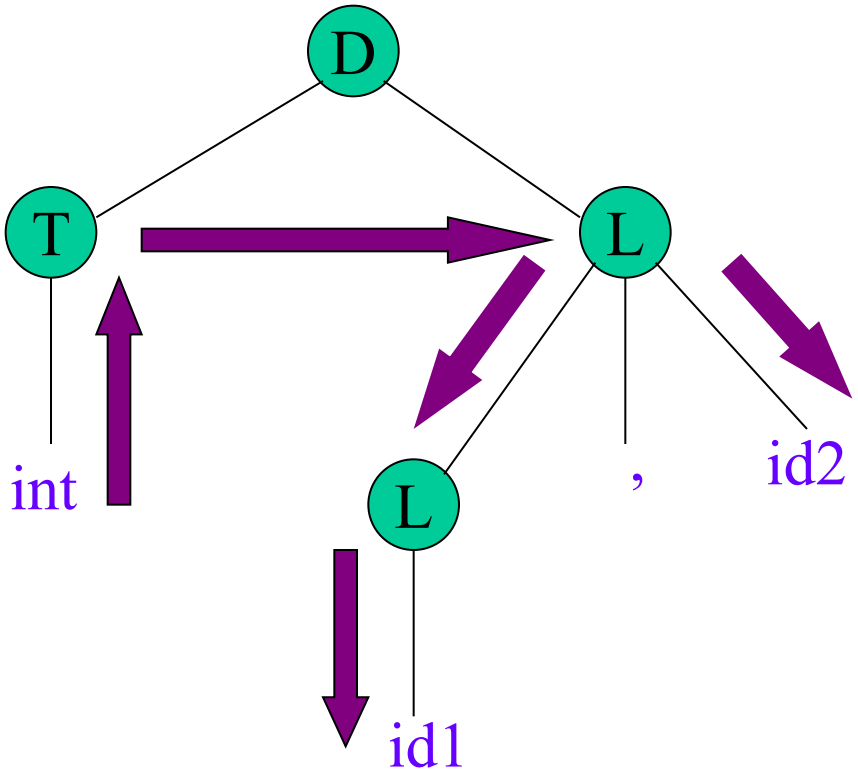
产生式	语 义 规 则
$L \longrightarrow E$	Print(E.val)
$E \longrightarrow E^1+T$	$E.val:=E^1.val+T.val$
$E \longrightarrow T$	E.val:=T.val
$T \longrightarrow T^1 * F$	$T.val:=T^1.val \times F.val$
$T \longrightarrow F$	T.val:=F.val
$F \longrightarrow (E)$	F.val:=E.val
$F \longrightarrow$ **digit**	F.val:=**digit**.lexval

例：构造表达式语法树的语法制导定义

产生式	语义规则
L→E	L.*nptr*:=E.*nptr*
$E \rightarrow E^1+T$	E.*nptr*:=*makenode*(‘+’, E^1.*nptr*, T.*nptr*)
E→T	E.*nptr*:=T.*nptr*
$T \rightarrow T^1*F$	T.*nptr*:=*makenode*(‘*’, T^1.*nptr*, F.*nptr*)
T→F	T.*nptr*:=F.*nptr*
F →(E)	F.*nptr*:=E.*nptr*
F→**num**	F.*nptr*:=*mkleaf*(**num**, **num**.*val*)

例：说明语句中各变量的类型信息的语法制导定义

产生式	语 义 规 则
D →TL	L.type:=T.type
T → **int**	T.Type:=integer
T → **real**	T.type:=real
L → L^1, **id**	L^1.type:=L.type addtype(**id**.entry, L.type)
L→ id	addtype(**id**.entry, L.type)

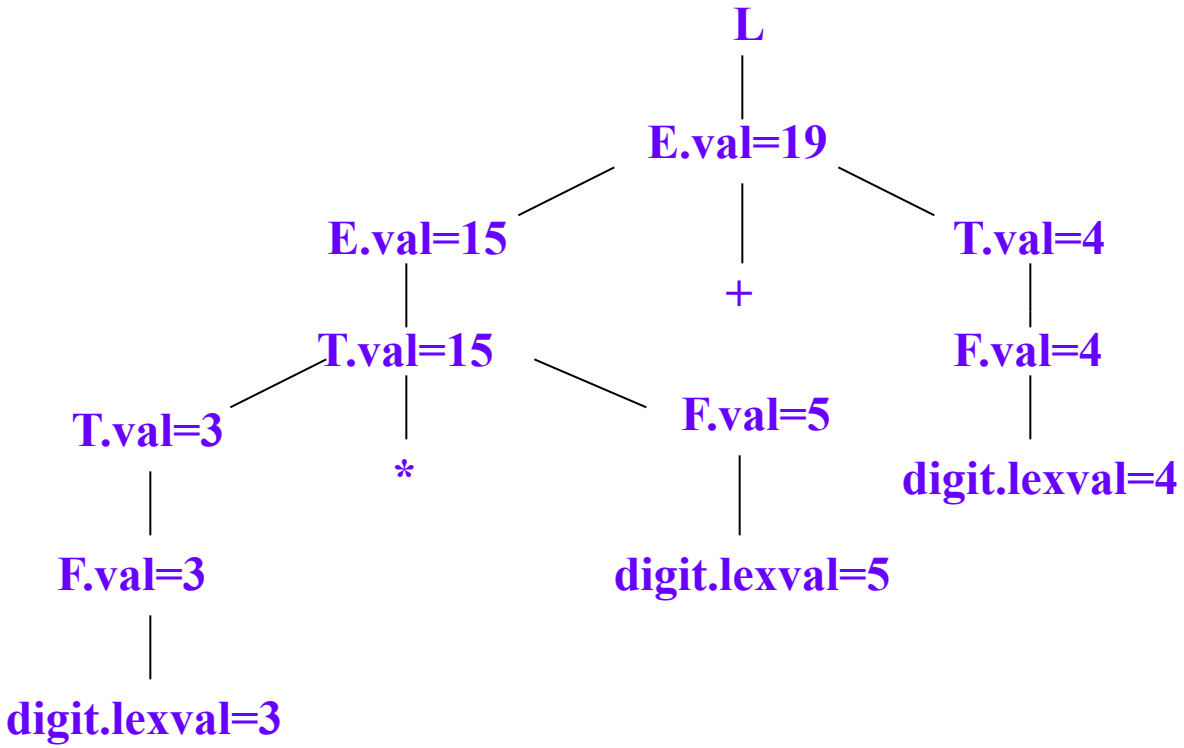


属性信息传递

7.2 语法制导的语义计算

注释分析树

每个结点的属性值都标注出来的分析树叫做**注释分析树。（分析树 vs 语法树）**



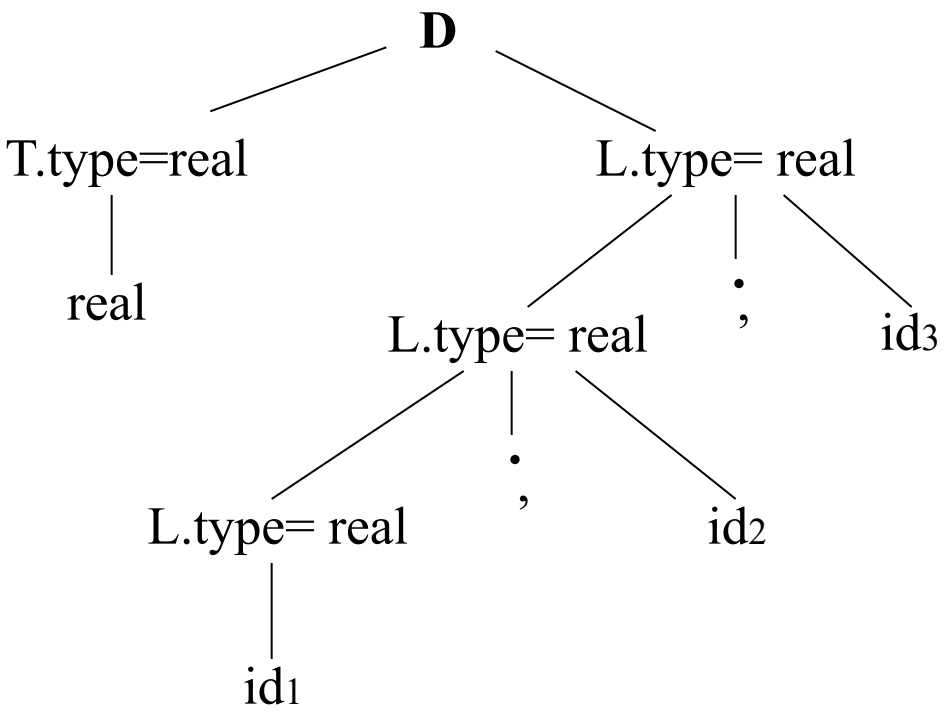
3*5+4的注释的分析树

计算语义规则

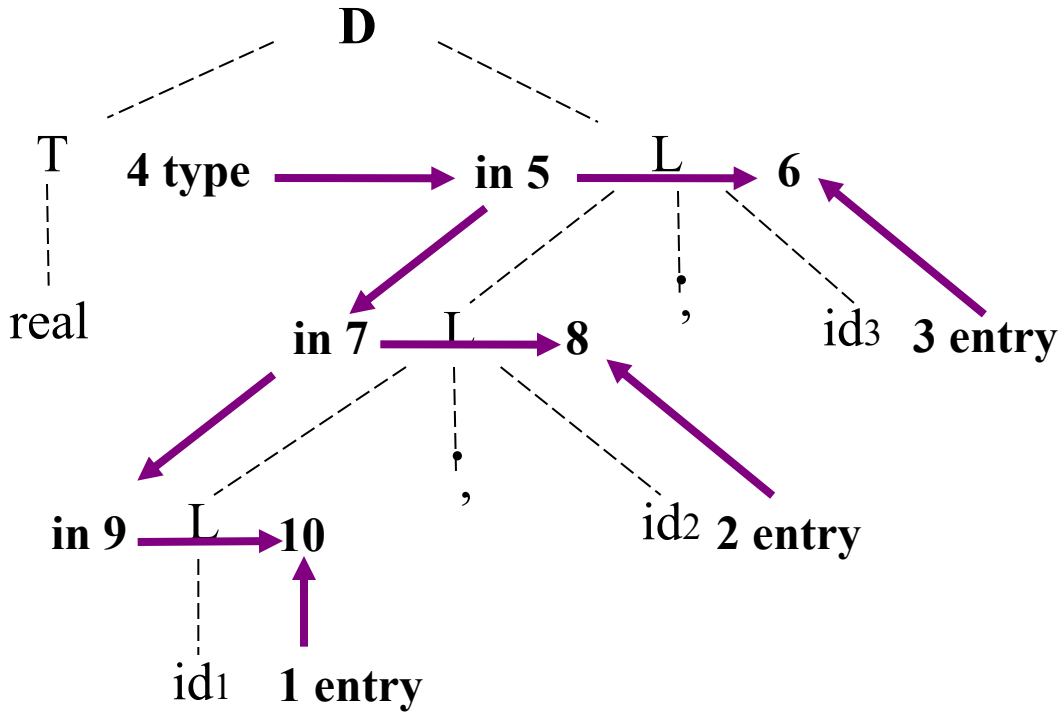
分析树结点的属性间的相互依赖关系图称为属性依赖图。构造算法如下：

for 分析树中的每一结点n *do*
  for 结点n的文法符号的每一属性a *do*
    为a在依赖图中建立一个结点；
for 分析树中的每一结点n *do*
  for 结点n所用产生式对应的每一语义规则
  $b:=f(c_1, c_2, \ldots, c_k)$ *do*
    for i :=1 to k *do* 从结点c_i到结点b构造一条有向边；

real id1,id2,id3的注释分析树



real id1,id2,id3分析树的属性依赖图



属性依赖图的拓扑序列，就是语义规则的计算顺序。

Forward

属性计算策略：

树遍历：常用的是深度优先、自左而右的遍历方法。

一遍扫描：在语法分析的同时计算属性值（无需构造语法树），而不是先构造语法树再计算属性。

S属性文法及其自下而上翻译

S属性文法：只含有综合属性的属性文法。S属性文法的翻译器通常可借助LR分析器实现。

实现方法：分析器保存与栈中文法符号有关的综合属性值，每当进行归约时，新的非终结符的属性值就由栈中待归约的产生式右边文法符号的属性值来计算。

例：简单算术表达式求值的语义描述(语法制导定义)

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

S_m	Y.val	Y
S_{m-1}	X.val	X
…	…	…
S_0	-	#
状态栈	语义值栈	符号栈

扩充的LR分析栈

步骤	状态栈	符号栈	语义栈	留余输入串	归约动作
1	0	#	–	2+3*5#	
2	05	#2	– –	+3*5#	
3	03	#F	– 2	+3*5#	r6
4	02	#T	– 2	+3*5#	r4
5	01	#E	– 2	+3*5#	r2
6	016	#E+	– 2 –	3*5#	
7	0165	#E+3	– 2 – –	*5#	
8	0163	#E+F	– 2 – 3	*5#	r6
9	0169	#E+T	– 2 – 3	*5#	r4
10	01697	#E+T*	– 2 – 3 –	5#	
11	016975	#E+T*5	– 2 – 3 – –	#	
12	01697510	#E+T*F	– 2 – 3 – 5	#	r6
13	0169	#E+T	– 2 – 15	#	r3
14	01	#E	– 17	#	r1
15					acc

2+3*5的分析计算过程

产生式	代码段
L→E	print(val[top])
E→E[1]+T	val[top-2]:=val[top-2]+val[top]
E→T	
T→T[1]*F	val[top-2]:=val[top-2]*val[top]
T→F	
F →(E)	val[top-2]:= val[top-1]
F→**digit**	

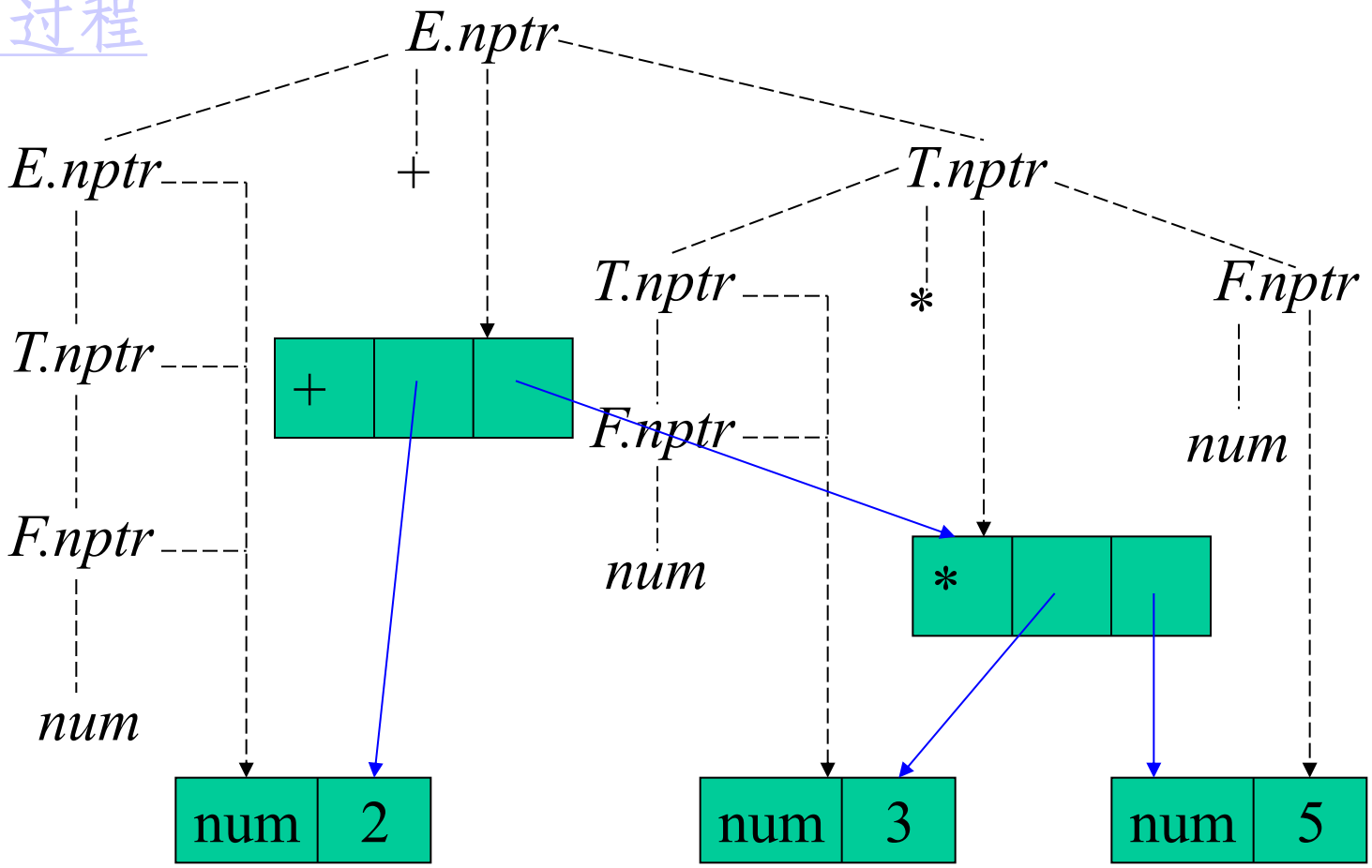
用LR分析器实现表达式求值

例：构造表达式文法语法树的语法制导定义

产生式	语义规则
L→E	L.*nptr*:=E.*nptr*
E→E1+T	E.*nptr*:=*makenode*(‘+’, E1.*nptr*, T.*nptr*)
E→T	E.*nptr*:=T.*nptr*
T→T1*F	T.*nptr*:=*makenode*(‘*’, T1.*nptr*, F.*nptr*)
T→F	T.*nptr*:=F.*nptr*
F →(E)	F.*nptr*:=E.*nptr*
F→**num**	F.*nptr*:=mkleaf(**num**, **num**.*val*)

2+3*5的语法树的构造

归约过程



L属性文法及其自上而下分析

L属性文法：如果对于每个产生$A \rightarrow X_1X_2 \ldots X_n$的每个语义规则计算的的属性，或者是A的综合属性，或者是X_j的一个继承属性而且这个继承属性仅依赖于：

(1) 产生式X_j的左边符号X_1，X_2，…X_{j-1}的属性；

(2) 非终结符A的继承属性；

- **S属性文法一定是L属性文法。**

例：说明语句中各变量的类型信息的L属性文法

产生式	语 义 规 则
D →TL	L.type:=T.type
T → **int**	T.Type:=integer
T → **real**	T.type:=real
L → L^1, **id**	L^1.type:=L.type addtype(**id**.entry, L.type)
L→ id	addtype(**id**.entry, L.type)

分析过程

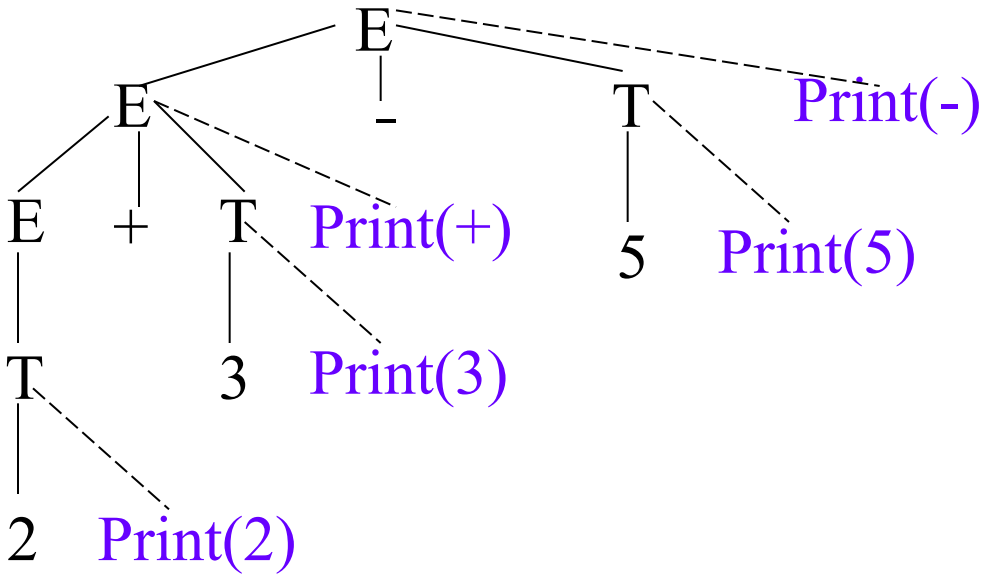
例：将中缀表达式翻译成相应的后缀表达式的属性文法。

E→E addop T print(addop.Lexeme)

E→T

T→num print(num.val)

若采用**LR分析方法**，串2+3-5时的**语义动作**图示如下：



例：将中缀表达式翻译成相应的后缀表达式的属性文法。

E→E addop T

E→T

T→num

仍以分析串2+3-5为例，若采用**LL(1)分析方法(自上而下的分析)**，其属性文法为？

消除左递归：

E→T R

R→addop T R1 |ε

T→num

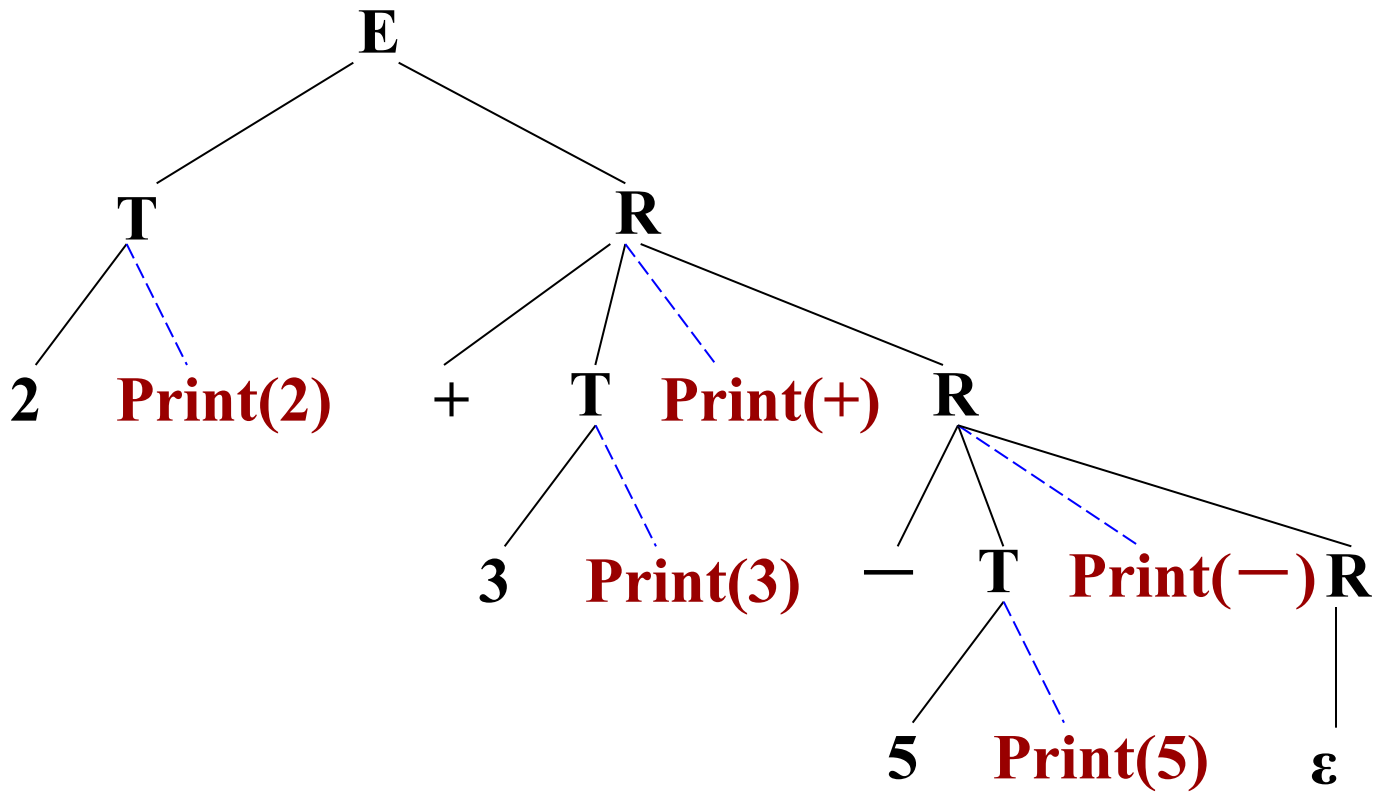
翻译模式 / 翻译方案 (translation schemes)：语法制导翻译的另一种描述形式。**语义动作**（区别于语义规则）放在{}括号内，可插入到产生式右部的**任何位置**。

E→TR

R→addop T print(addop.Lexeme) R1 |ε

T→num print(num.val)

包含语义动作的分析树



分析串2+3-5

转换左递归翻译模式的方法

若翻译模式为：

$A \rightarrow A^1 Y$ {A.a:=g(A1.a, Y.y)}

$A \rightarrow X$ {A.a:=f(X.x)}

消除左递归后的**翻译模式**如下(使用了R的继承属性i):

$A \rightarrow X$ {R.i:=f(X.x)}

R {A.a:=R.s}

$R \rightarrow Y$ {R1.i:=g(R.i,Y.y)}

R1 {R.s:=R1.s}

$R \rightarrow \varepsilon$ {R.s:=R.i}

例如：

E→E^1+T　　E.*nptr*:=*makenode*('+', E^1.*nptr*, T.*nptr*)

E→T　　E.*nptr*:=T.*nptr*

T→num　　F.*nptr*:=mkleaf(num, num.*val*)

转换成自上而下分析(消除左递归)：

E→T R

R→+TR^1

R→ε

转换成自上而下分析(消除左递归)：

E→T　　{R.*i*:=T.*nptr*}

　　R　　{E.*nptr*:=R.s}

R→+

　　T　　{R^1.*i*= *makenode*(‘+’, R.i, T.*nptr*)}

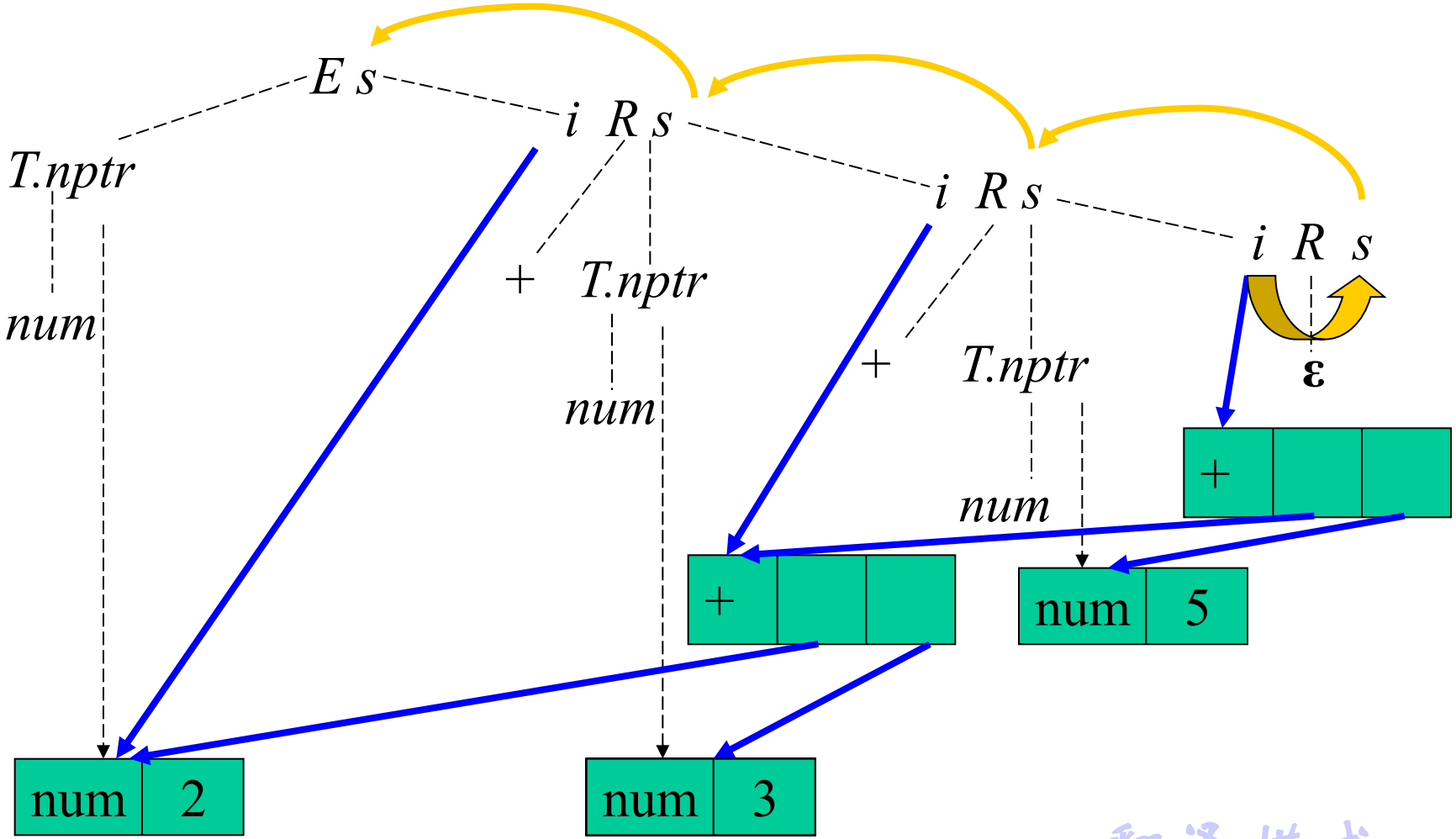
　　R^1　　{R.*s*:=R^1.*s*}

R→ε　　{R.*s*:=R.*i*}

此翻译模式的**属性信息**（语法树的结点指针）是**从左向右**流动的，而**归约方向**却是**从右向左**的，这种不一致性导致了继承属性的出现。

Forward

2+3+5的语法树的构造



翻译模式

L属性文法的自下而上分析

通过**自下而上**分析**计算继承属性**的方法：

(1) 删除翻译模式中嵌入的动作
(2)用综合属性代替继承属性
(3)用分析栈处理继承属性

删除翻译模式中嵌入的动作的方法：

引入新的非终结符N和产生式$N\rightarrow\varepsilon$，把嵌入在产生式中的**语义动作**用非终结符N代替，并把这个动作放在$N\rightarrow\varepsilon$的右端。

例如：

$E \rightarrow TR$

$R \rightarrow +T$ {print(+)} R^1

$R \rightarrow -T$ {print(-)} R^1

$R \rightarrow \varepsilon$

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

引入产生ε的标记非终结符后的翻译方案为：

$E \rightarrow TR$

$R \rightarrow +T\ MR^1$

$R \rightarrow -\ T\ N\ R^1$

$R \rightarrow \varepsilon$

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

$M \rightarrow \varepsilon$ {print(+)}

$N \rightarrow \varepsilon$ {print(-)}

用综合属性代替继承属性：

改变基础文法是一种可行的方法

例如，说明语句文法：

D→ L:T

T→ integer | char

L→ L, id | id

产生式	语 义 规 则
D →TL	L.type:=T.type
T → int	T.type=integer
T → real	T.type:=real
L → L^1, id	L^1.type:=L.type addtype(id.entry, L.type)
L→ id	addtype(id.entry, L.type)

分析：标识符由L产生，但类型不在L中，仅用综合类型不能把类型与标识符联系起来。

进一步，类型信息**从右向左**流动，而文法是**从左向右**归约的。

解决方法：重构文法！

原文法：

D→ L:T

T→ integer | char

L→ L, id | id

新文法如下：

D→ id L

L→, id L | :T

T→ integer | char

相应的，属性文法如下:

D→ id L　　{addtype(id.entry, L.type)}

L→, id L1　{L.type:= L1.type;
　　addtype(id.entry, L1.type)}

　　:T　　{L.type:= T.type}

T→integer {T.type:= int}

　　char　　{T.type:= char}

此时，类型信息流动方向和归约方向一致，都是从右向左的。

用分析栈处理继承属性

产生式	语 义 规 则
D →TL	L.type:=T.type
T → int	T.type=integer
T → real	T.type:=real
L → L^1, id	L^1.type:=L.type addtype(id.entry, L.type)
L→ id	addtype(id.entry, L.type)

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

其中，val[i]保存对应文法符号的综合属性

习 题

1

3

13(1)

定点二进制数的CFG

N→S•S

S→SB

S→B

B→0

B→1

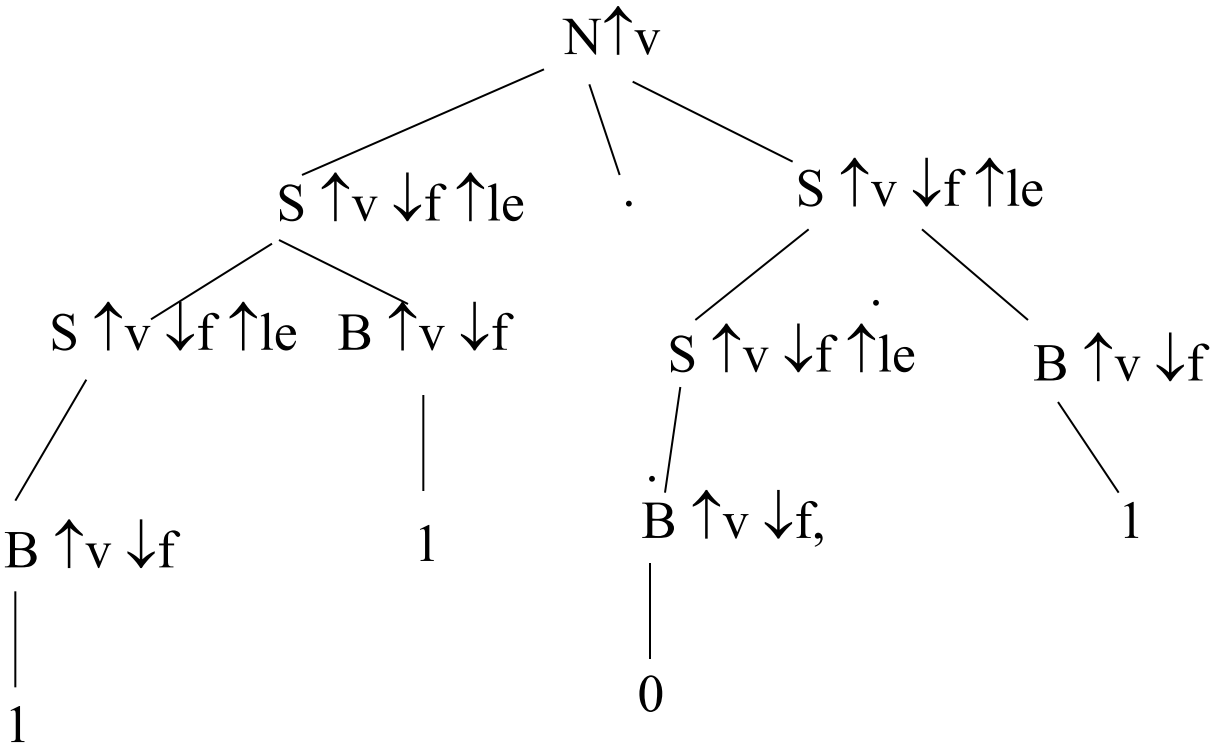
显然，可以使用一个正规文法表示同样的语言。D.E.Knuth使用这个CFG是为了说明综合属性和继承属性如何附加在非终结符上。

非终结符N表示整个二进制数。

非终结符N表示整个二进制数，该例中共使用了三个属性：综合属性v和le，继承属性f，分别使用↑和↓显示综合属性和继承属性.综合属性v附加在N上：N↑v 表示整个二进制数的数值。非终结符B表示一个二进制数字，它有自己的值v ，但该值与它的位置有关，与2成比例，但这个比例因子是无法用综合属性的，因此设计一个从S继承的属性f，那么附加在B的属性表示为：B↑v↓ f

非终结符S 表示一个二进制数字串，它的值v与它在串中的位置有关，与串的长度le有关， f的计算基于串的长度和相对于小数点的位置，所以有:
S↓f↑v↑le

继承属性和综合属性---带注释的分析树



构造数值的属性和断言

$N \uparrow v \rightarrow S \downarrow f_1 \uparrow v_1 \uparrow le_1 \bullet S \downarrow f_2 \uparrow v_2 \uparrow le_2$

$\{v=v_1+v_2;\ f_1=1;\ f_2=2^{-le2}\}$

/* 一个数的值是整数部分v_1和小数部分v_2的和；整数部分和小数部分已分别选定了比例因子f，并沿语法树传下去了,该数的整数部分的数字串继承的比例因子是1，而该数的小数部分的数字串继承的比例因子基于小数点至它的右端的数字的数目，即小数部分的长度*/

$S \downarrow f \uparrow v \uparrow le \rightarrow S \downarrow f_1 \uparrow v_1 \uparrow le_1\ \ B \downarrow f_2 \uparrow v_2$

$\{f_1=2f;\ \ f_2=f;\ \ v=v_1+v_2; le=le_1+1\}$

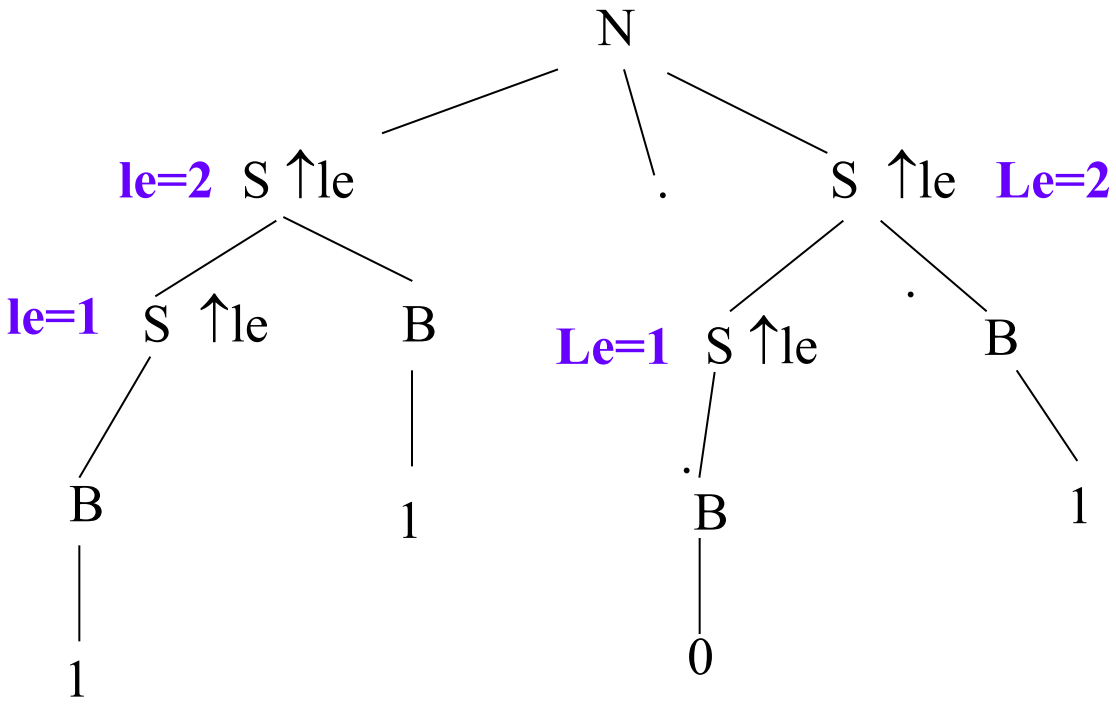
/*任何二进制数串都是由子串后跟一位二进制数字构成，二进制数字B的比例因子与数串的一样，子串的要乘以2，数串的长度增1*/

$\rightarrow B \downarrow f\ \uparrow v\ \ \{le=1\}$ /*当数串只有一位数字时，f和v都不变*/

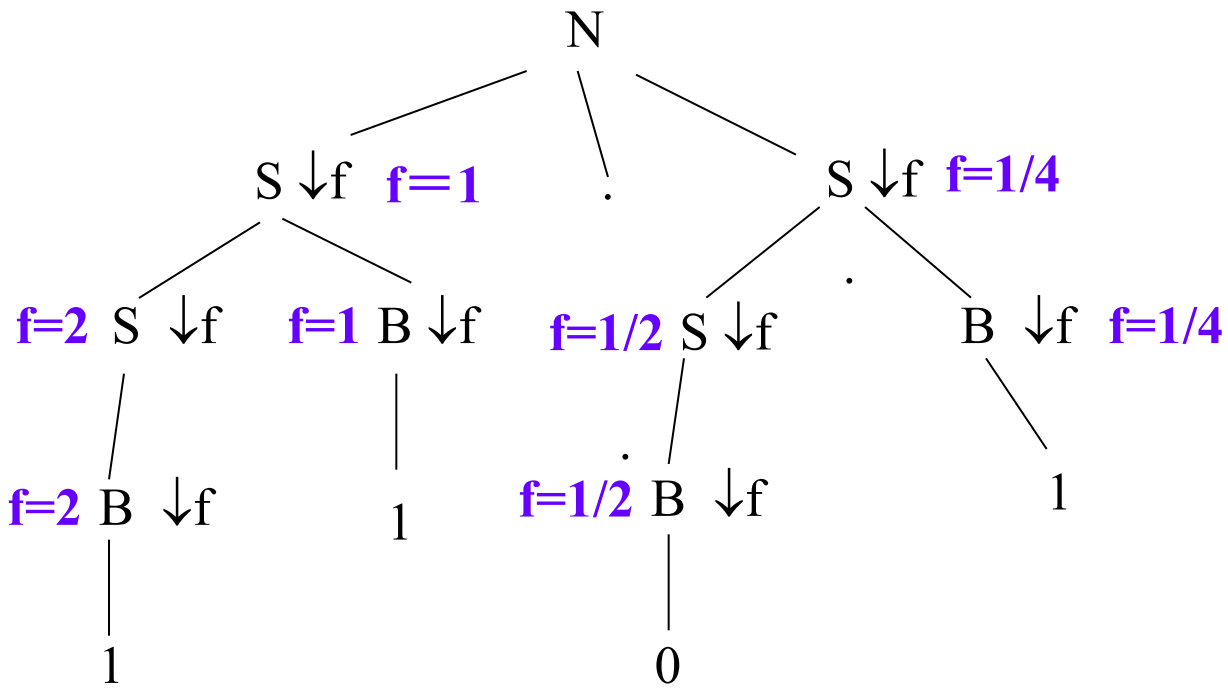
$B \downarrow f \uparrow v \rightarrow 0\ \ \{v=0\}$

$\rightarrow 1\ \ \{v=f\}$ /*属性v和f有关,f是从父结点继承的*/

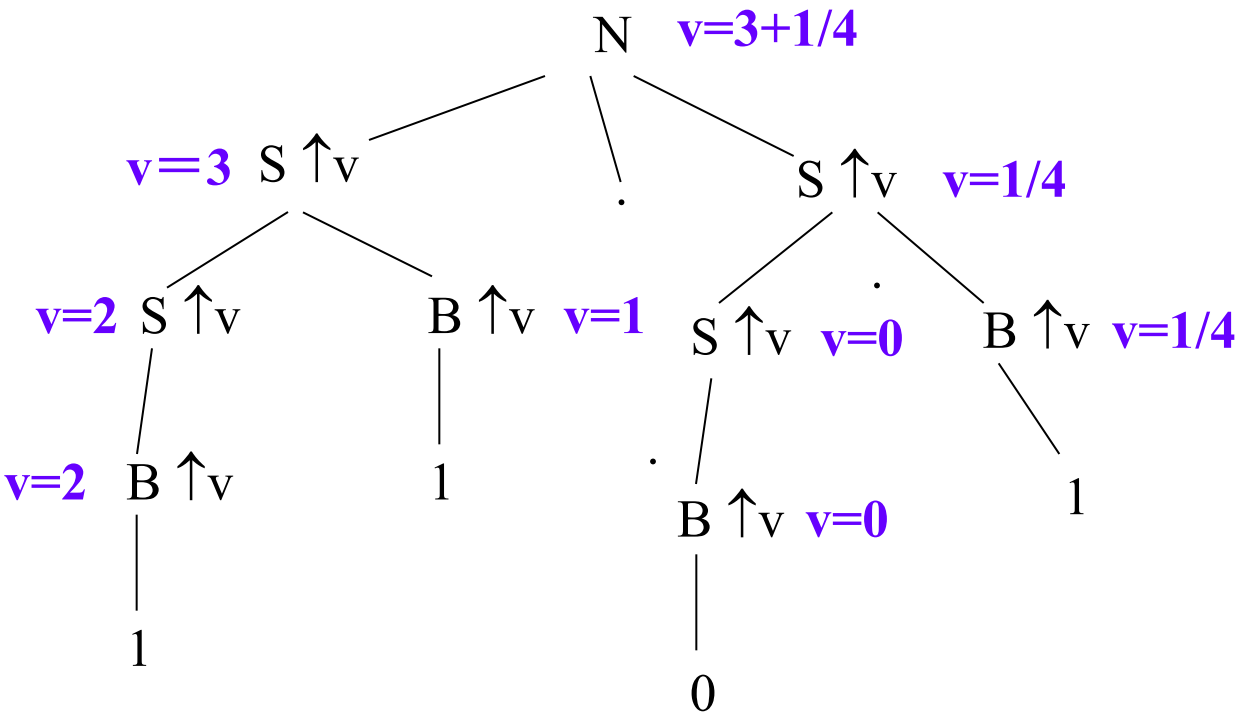
语法树中的属性流---属性le自底向上



语法树中的属性流---属性f自上而下



语法树中的属性流---属性v自底向上



当然，只使用综合属性i和l也可以描述二进制数值的计算，这样属性计算可以使用一遍扫描的处理方法：

$N \uparrow v \quad \rightarrow S \uparrow i_1 \uparrow l_1$ “•” $S \uparrow i_2 \uparrow l_2$

$\{v = i_1 + 2^{-l2} \bullet i_2\}$

$S \uparrow i \uparrow l \quad \rightarrow S \uparrow i_1 \uparrow l_1 \quad B \uparrow i_2$

$\{i = 2\, i_1 + i_2;\ l = l_1 + 1\}$

$\rightarrow B \uparrow i \qquad \{l = 1\}$

$B \uparrow i \quad \rightarrow$ “0” $\qquad \{i = 0\}$

$\rightarrow$ “1” $\qquad \{i = 1\}$

Yacc或bison作为编译程序的生成工具，利用的就是语法制导翻译方法

下面的文法定义了类似Pascal语法的声明和简单赋值语句:

P → DS

D → var V; D | ε

S → V := E; S | ε

V → x | y | z

考虑为该文法增加两个属性，name和 dl

每当一个新变量被声明，就将它的name属性附加给变量，然后将这个名字加列到名字清单中， 属性dl表示目前声明块已声明了的所有名字

要进行的语义检查是：使用的名字必须已声明。

属性文法如下：

P → **DS** {S.dl = D.dl}

D^1 → **var V; D**^{2} {D^1.dl = addlist (V.name, D^2.dl)}

| **ε** {D^1.dl = NULL}

S^1 → **V := E; S**^{2} {check(V.name, S^1.dl); S^2.dl = S^1.dl}

| **ε**

V → **x** {V.name = 'x'}

| **y** {V.name = 'y'}

| **z** {V.name = 'z'}

Yacc或bison作为编译程序的生成工具，利用的就是语法制导翻译方法。它使用符号$$表示产生式左端的属性，$n表示存取产生式右端第n个文法符号相联的属性。

如上例作为Yacc的输入，可写成：

P → DS {$2.dl=$1.dl}

D^1 → var V; D^2 {$$.dl=addlist($2.name，$4.dl)}

| ε {$$.dl=null}

S^1 → V := E; S^2 {check($1.name，$$.dl);

$5.dl=$$.dl}

| ε

V → x {$$.name='x'}

|y {$$.name='y'}

|z {$$.name='z'}