第六章 属性文法

- 语义分析阶段计算编译所需的额外信息，这些信息与程序的最终含义密切相关，目的：
  - 保证程序正确及正常执行（语言定义）
  - 提高程序运行效率（优化）
- 语义举例：
  - 建立符号表保存声明中的名字的含义；
  - 类型检查和类型推理

6.1 属性

- **属性**：是任意一个跟程序设计语言构造相关的特性。属性所包含的信息、复杂程度、能够确定其值的时间区间等都有较大的范围。
  - **举例**：
    - 变量的类型；
    - 表达式的值；
    - 变量在存储器中的位置；
    - 过程(procedure)的目标代码；
    - 数中有效位数。

- 语义表示的困难性
  - 缺乏标准的方法描述语言的语义；
  - 语义分析中的量和种类在不同语言间变化很大；

- 为语言构造指定属性
  - 给定上下文无关文法，为它的每个文法符号关联相关的“值”（称为属性），以表示与这些文法符号相关的信息。

- 表示语义的较好的方法
  - 首先标识语言实体(language entity)的属性（特性）；
  - 然后写出属性方程（语义规则）：表明如何把这些属性的计算跟文法规则相关联；
  - 属性+属性方程=属性文法

属性的binding

- 计算属性的值及将其跟语言构造建立联系的过程，属性绑定发生的时刻称为绑定时间
  - 静态：在编译过程中（执行之前）
  - 动态：在执行时
  - 变量的类型；(C/Pascal/Lisp Type Checker)
  - 表达式的值；(Code/Constant Folding)
  - 变量在存储器中的位置；(Static/Dynamic Allocation)
  - 过程(procedure)的目标代码；(Static)
  - 数中有意义数字的个数。(Runtime Env.)

属性方程

- 将属性值a与文法符号X联系起来，记为X.a
- 对于产生式规则X_0->$X_1X_2…X_n$,有

  $X_i.a_j$ =

  $f_{ij}(X_0.a_1,…,X_0.a_k,X_1.a_1,…,X_1.a_k,…,X_n.a_1,…,X_n.a_k)$

其中，f_{ij}为数学函数，

则称为属性方程或语义规则

例6.1 无符号数

N→ND|D

D→0|1|2|3|4|5|6|7|8|9

文法规则	语义规则
$N_1→N_2D$	$N_1.val=N_2.val*10+D.val$
N→D	N.val=D.val
D→0	D.val=0
D→1	D.val=1
D→2	D.val=2
D→3	D.val=3
D→4	D.val=4
D→5	D.val=5
D→6	D.val=6
D→7	D.val=7
D→8	D.val=8
D→9	D.val=9

属性文法

- 针对文法所有规则构造的属性以及属性方程的集合

N→ND|D

D→0|1|2|3|4|5|6|7|8|9

用属性val表示语法单位数和数字的数值

N.val

D.val

N→D

N.val=D.val

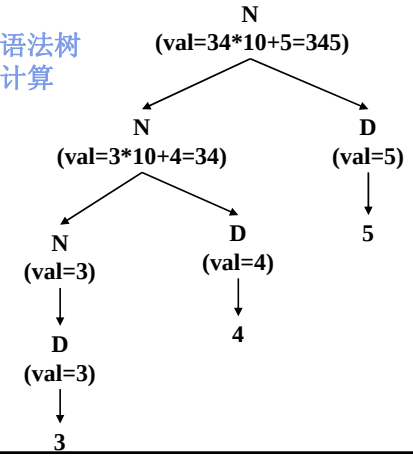
N→ND

?

$N_1→N_2D$

$N_1.val=N_2.val*10+D.val$

带注释的语法树表示属性计算



属性文法的表示形式

文法规则	语义规则
产生式规则 1	关联的属性方程（多个）
…	…
产生式规则 *n*	关联的属性方程（多个）

例6.2 简单整数算术表达式

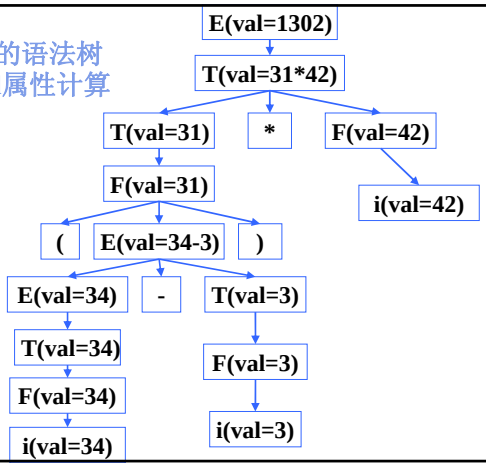
E→E+T|E-T|T

T →T*F|F

F →(E)|i

文法规则	语义规则
$E_1→E_2+T$	$E_1.val=E_2.val+T.val$
$E_1→E_2-T$	$E_1.val=E_2.val-T.val$
E→T	E.val=T.val
$T_1→T_2*F$	$T_1.val=T_2.val*F.val$
T→F	T.val=F.val
F→(E)	F.val=E.val
F→i	F.val=i.val

带注释的语法树
显示val属性计算



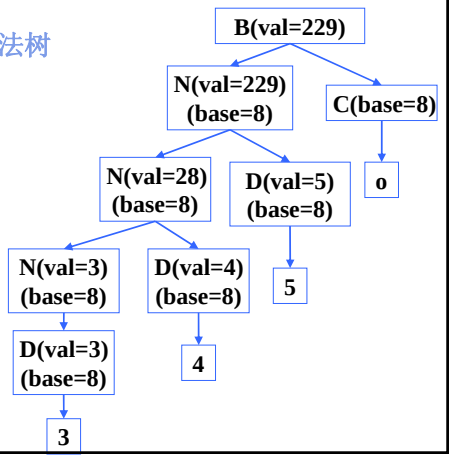
例6.4 进制数
B→NC
C→o | d
N→ND|D
D→0|1|2|3|4|
5|6|7|8|9

B→NC	B.val=N.val N.base=C.base
C → o	C.base=8
C → d	C.base=10
$N_1 \rightarrow N_2$ D	N_1.val=if D.val=error or N_2.val=error then error else N_2.val*N_1.base+D.val N_2.base=N_1.base D.base=N_1.base
N → D	N.val=D.val D.base=N.base
D → 0	D.val=0
…	…
D → 8	D.val= if D.base=8 then error else 8
…	…

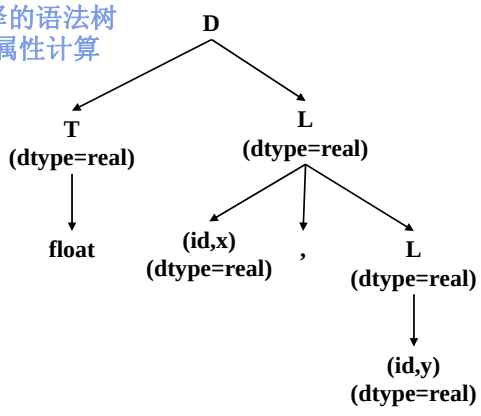
例6.3 变量声明
D → TL
T → int | float
L → id, L | id

文法规则	语义规则
D → TL	L.dtype=T.dtype
T → int	T.dtype=integer
T → float	T.dtype=real
$L_1 \rightarrow$ id, L_2	id.dtype=L_1.dtype L_2.dtype=L_1.dtype
L → id	id.dtype=L.dtype

带注释的语法树



带注释的语法树
dtype属性计算



对属性文法的简化和扩展

- 文法简化
  - 使用带二义性的文法
- 元语言扩展
  - 算术、逻辑等表达式，if-then-else, case, switch
  - 函数调用
  - 使用抽象语法树代替语法分析树

无符号数
N → ND|D
D → 0|1|2|3|4|5|6|7|8|9

$N_1 \rightarrow N_2D$	$N_1.val=N_2.val*10+D.val$
N → D	N.val=D.val
D → 0	D.val=0
D → 1	D.val=1
…	…
D → 9	D.val=9

```
D.val=numval(d)
int numval(char d){
  return (int)d - (int)'0'}
```

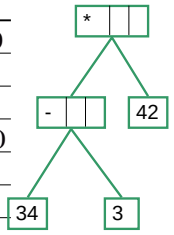
$N_1 \rightarrow N_2D$	$N_1.val=N_2.val*10+D.val$
N → D	N.val=D.val
D → d	D.val=numval(d)

文法：
E → E+T|E-T|T
T → T*F|F
F →(E)|i

mkOpNode()返回树结点，其操作符为第一个参数，孩子是第二、第三个参数

mkNumNode()构造树叶子结点，含有参数所指的数值。

$E_1 \rightarrow E_2+T$	$E_1tree=mkOpNode(+, E_2.tree, T.tree)$
$E_1 \rightarrow E_2-T$	$E_1tree=mkOpNode(-, E_2.tree, T.tree)$
E→T	E.tree=T.tree
$T_1 \rightarrow T_2*F$	$T_1.tree=mkOpNode(*, T_2.tree, F.tree)$
T→F	T.tree=F.tree
F→(E)	F.tree=E.tree
F→i	F.tree=*mkNumNode*(*numval*(i))



例 简单整数算术表达式
E→E+T|E-T|T
T→T*F|F
F→(E)|i

$E_1 \rightarrow E_2+T$	$E_1.val=E_2.val+T.val$
$E_1 \rightarrow E_2-T$	$E_1.val=E_2.val-T.val$
E→T	E.val=T.val
$T_1 \rightarrow T_2*F$	$T_1.val=T_2.val*F.val$
T→F	
F→(E)	F.val=E.val
F→ i	F.val=i.val

带二义性的简单整数算术表达式文法:
E → E+E|E-E|
E*E|(E)|i

$E_1 \rightarrow E_2+E_3$	$E_1.val=E_2.val+E_3.val$
$E_1 \rightarrow E_2-E_3$	$E_1.val=E_2.val-E_3.val$
$E_1 \rightarrow E_2*E_3$	$E_1.val=E_2.val*E_3.val$
$E_1 \rightarrow (E_2)$	$E_1.val=E_2.val$
E→i	E.val=numval(i)

6.2属性计算算法

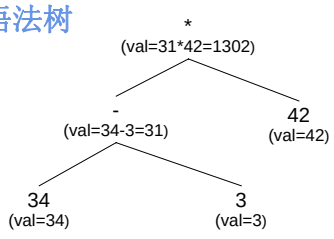
产生式规则$X_0 \rightarrow X_1X_2…X_n$的属性方程：

$X_i.a_j=f_{ij}(X_0.a_1,…,X_0.a_k,X_1.a_1,…,X_1.a_k,…,X_n.a_1,…,X_n.a_k)$

- 根据属性方程决定属性的计算与使用
  - 右边已知计算出左边；
  - 反之；
  - 一般，一些已知计算出另一些
- 属性文法中的属性的计算次序问题

抽象语法树

带二义性的简单整数算术表达式文法:
E → E+E|
E-E|
E*E|
(E)|
i



在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称之为抽象语法树。

在抽象语法树中，操作符和关键字都不作为叶结点出现，而是把它们作为内部结点，即这些叶结点的父结点。

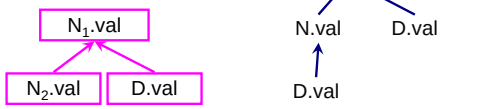
6.2.1 依赖图与求值次序

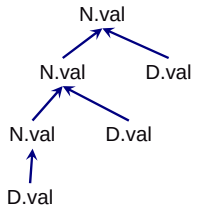
- 给定属性文法，则文法每个产生式规则对应一个依赖图
  - 为一个有向图，结点为属性如$X_i.a_j$，弧的定义如下：若有属性方程 $X_i.a_j=f_{ij}(…, X_m.a_k, …)$，则结点$X_m.a_k$到结点$X_i.a_j$有一条弧存在。
- 对于上下文无关文法的语言中的任意串，它的依赖图就是把表示该串的语法树的每个内结点的产生式规则的依赖图进行合并而成

例：依赖图

$N_1 \rightarrow N_2 D$

$N_1.val = N_2.val * 10 + D.val$





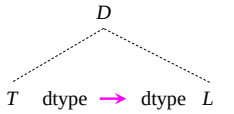
例6.4 进制数
B → NC
C → o | d
N → ND|D
D → 0|1|2|3|4|
5|6|7|8|9

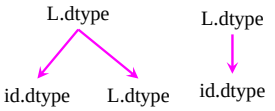
文法规则	语义规则
B → NC	B.val=N.val N.base=C.base
C → o	C.base=8
C → d	C.base=10
$N_1 \rightarrow N_2$ D	N_1.val=if D.val=error or N_2.val=error then error else N_2.val*N_1.base+D.val N_2.base=N_1.base D.base=N_1.base
N → D	N.val=D.val D.base=N.base
D → 0	D.val=0
D → 8	D.val=if D.base=8 then error else 8

例：变量声明依赖图

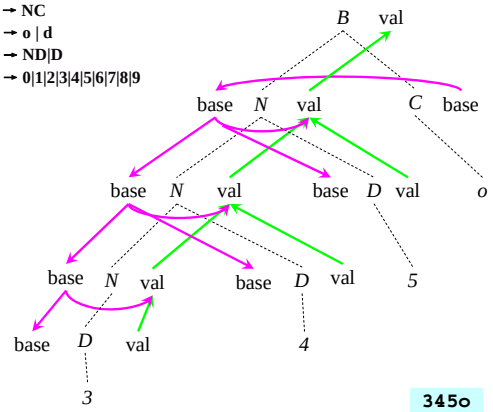
D → TL
T → int | float
L → id, L | id

文法规则	语义规则
D → TL	L.dtype=T.dtype
T → int	T.dtype=integer
T → float	T.dtype=real
$L_1 \rightarrow$ id, L_2	id.dtype=L_1.dtype
	L_2.dtype=L_1.dtype
L → id	id.dtype=L.dtype

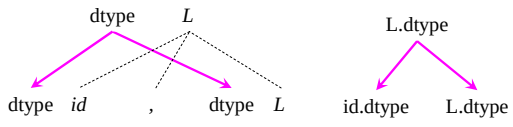


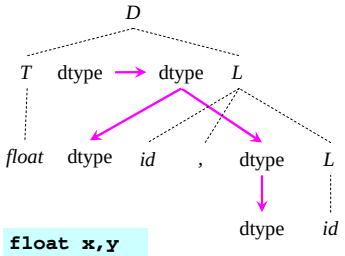


B → NC
C → o | d
N → ND|D
D → 0|1|2|3|4|5|6|7|8|9



345o






```
float x,y
```

为语法树添加依赖图的算法

for (语法树中每个结点n)
  for (结点n的每个属性a)
    为a在依赖图中建立一个结点;
for (语法树中每个内结点n)
  for (结点n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;

计算次序

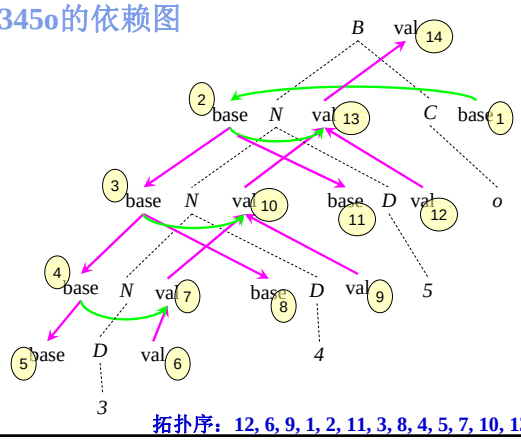
- 计算语法树中每个属性的值（计算属性文法的属性的值）：利用属性方程
- 计算的次序：对该属性的依赖图中的结点进行拓扑排序
- 拓扑排序：

一个有向非循环图的拓扑序是图中结点的任何顺序m1, m2, …, mk，使得边必须是从序列中前面的结点指向后面的结点。

DAGs与属性计算

- 一个有向图存在拓扑序的充要条件是该图是有向无环图（DAG, directed acyclic graph）；
  - 如果一个属性文法不存在属性之间的循环依赖关系，则该属性文法是良定义文法
- 依赖图的根是指那些入度为0的结点；根结点的属性的值由Token直接得到，通常由scanner或parser得出；
- 无循环的属性文法：属性文法的每个依赖图均没有循环；

345o的依赖图

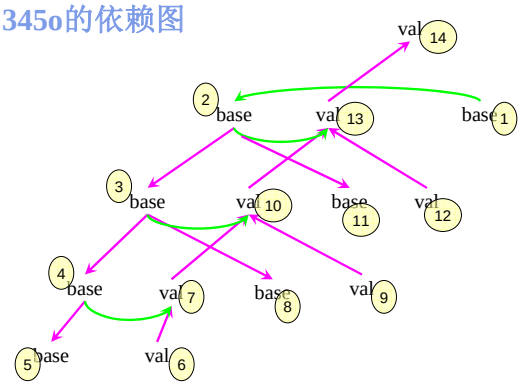


拓扑序：12, 6, 9, 1, 2, 11, 3, 8, 4, 5, 7, 10, 13, 14

DAGs与属性计算（续）

- 属性计算的语法分析树方法
  - 根据依赖图和拓扑次序计算某属性的值；
  - 在编译阶段进行
  - 对属性文法的循环性的判定是指数复杂度；
- 属性计算的基于规则的方法
  - 在写编译器的时候就根据属性文法确定好属性的计算次序；
  - 可不判断属性文法的循环性

345o的依赖图



拓扑序：12, 6, 9, 1, 2, 11, 3, 8, 4, 5, 7, 10, 13, 14

6.2.2 综合属性与继承属性

- 一个属性是综合属性是指在语法树中，该属性的值由其子结点的属性值计算而得。
- a是综合属性，若给定$A \rightarrow X_1X_2\ldots X_n$,其唯一关联的属性方程中a在等号左边

$$A.a_j = f(X_1.a_1,\ldots,X_1.a_k, \ldots,X_n.a_1,\ldots,X_n.a_k)$$

- S-属性文法
  - 所有属性都是综合属性的属性文法

继承属性

- 不是综合属性的属性是继承属性
  - 语法树中，一个结点的继承属性由此结点的父结点和或兄弟结点的某些属性确定。
  - 由双亲、兄弟 、子结点共同计算的继承属性？

例，文法: E→E+E|E-E|E*E|(E)|i

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpKind,ConstKind} ExpKind;
typedef struct streenode {
    ExpKind kind;
    OpKind op;
    struct streenode *lchild,*rchild;
    int val;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

6.2.3 基于树遍历的属性计算方法

- 综合属性的计算是基于后根遍历语法树
- 继承属性的计算是基于先根遍历语法树

```
procedure PostEval(T:treenode);
begin
    for each child C of T do
        PostEval(C);
    compute all synthesized attributes of T;
end;
```

计算综合属性

```
void postEval(SystaxTree t){
    int temp;
    if (t->kind == opKind){
        postEval(t->lchild); postEval(t->rchild);
        switch (t->op) {
            case Plus:
                t->val=t->lchild->val + t->rchild->val;
                break;
            case Minus:
                t->val=t->lchild->val - t->rchild->val;
                break;
            case Times:
                t->val=t->lchild->val * t->rchild->val;
                break;
        }}}
```

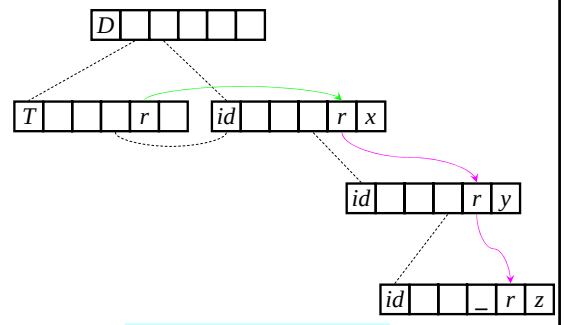
继承属性的计算

```
procedure PreEval(T:treenode);
begin
   for each child C of T do
      compute all inherited attributes of C;
   PreEval(C);
end;
```

D->TL　　T->int | float　　L->id, L | id

```
typedef enum {D,T,L,id} nodkind;
typedef enum {integer,real} typekind;
typedef struct treeNode {
    nodekind kind;
    struct treeNode *lchild,*rchild,*sibling;
    typekind dtype; /*for type and id nodes*/
    char *name;  /*for id node only*/
} *SyntaxTree;
```

例. 遍历语法树1计算属性



float x,y,z抽象语法树1

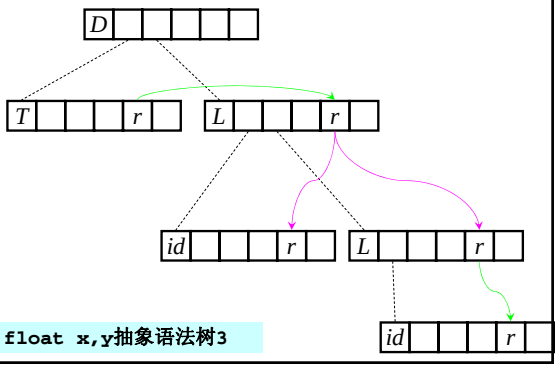
```
void evalType (SyntaxTree t){
 switch(t->kind){
 case D:  t->rchild->dtype=t->lchild->dtype;
     evalType(t->rchild);  break;
 case id:  if (t->sibling!=NULL){
     t->sibling->dtype=t->dtype;
     evalType(t->sibling);} break;
 case L:   t->lchild->dtype=t->dtype;
     break;
   }
}
```

遍历抽象语法树2的代码片段

```
void evalType (SyntaxTree t){
 switch(t->kind){
 case D:
     t->rchild->dtype=t->lchild->dtype;
     evalType(t->rchild);
     break;
 case id:
     if (t->sibling!=NULL){
          t->sibling->dtype=t->dtype;
          evalType(t->sibling);}
     break;
   }
}
```

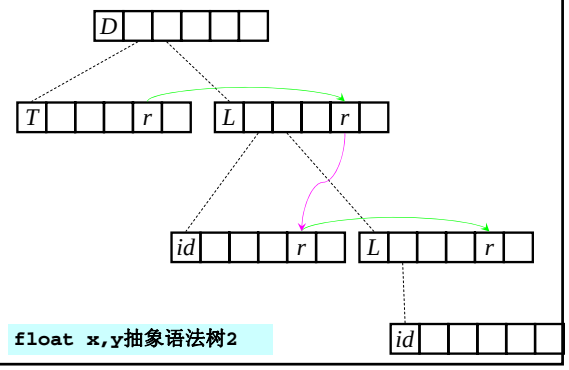
遍历抽象语法树1的代码片段

例. 遍历语法树3计算属性



float x,y抽象语法树3

例. 遍历语法树2计算属性

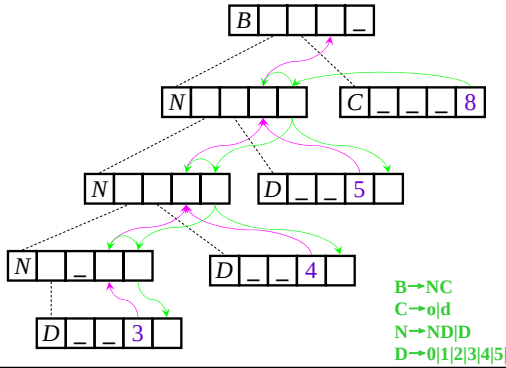


float x,y抽象语法树2

```
void evalType (SyntaxTree t){
 switch(t->kind){
 case D:  t->rchild->dtype=t->lchild->dtype;
     evalType(t->rchild);  break;
 case L:   t->lchild->dtype=t->dtype;
     if (t->rchild!=NULL){
     t->rchild->dtype=t->dtype;
     evalType(t->rchild);
     } break;
   }
}
```

遍历抽象语法树3的代码片段

基于345o的抽象语法树的属性计算



B→NC
C→o|d
N→ND|D
D→0|1|2|3|4|5|6|7|8|9

在语法树上一遍计算

- 如果：继承属性不依赖于任何综合属性；综合属性依赖于继承属性和其他综合属性

```
procedure CombinedEval(T:treenode);
begin
   for each child C of T do
      compute all inherited attributes of C;
      CombinedEval(C);
   compute all synthesized attributes of T;
end;
```

B→NC C→o|d N→ND|D D→0|1|2|3|4|5|6|7|8|9

```
Procedure EvalWithBase (T:treenode);
begin
   case nodekind of T of
   B:
      EvalWithBase(right child of T);
      assign base of right child of T to base of left child;
      EvalWithBase(left child of T);
      assign val of left child of T to T.val;
   N:
      assign T.base to base of left child of T;
      EvalWithBase(left child of T);
      if right child of T is not nil then
         assign T.base to base of right child of T;
         EvalWithBase(right child of T);
         if vals of left and right children != error then
            T.val=T.base*(val of left child) + val of right child;
         else T.val:=error;
      else T.val:=val of left child;
```

在语法树上多遍计算

- 继承属性依赖于综合属性

例. 文法:
E → E/E|i|i.i

- 属性
  - isFloat（综合属性）：E的任何部分是否为浮点数
  - etype（继承属性）： 表示任何子表达式的类型
  - val（综合属性）：表达式的值

```
   C:
      if child of T = o then T.base:=8 else T.base:=10;
   D:
      if T.base = 8 and child of T = 8 or 9 then T.val :=error
      else T.val = numval(child of T);
   end case;
end EvalWithBase;
```

B→NC C→o|d N→ND|D D→0|1|2|3|4|5|6|7|8|9

例. 文法: E → E/E|i|i.i

S→E	E.etype=E.isFloat?float:int S.val=E.val
E_1→E_2/E_3	E_1.isFloat=E_2.isFloat or E_3.isFloat E_2.etype= E_1.etype ; E_3.etype= E_1.etype E_1.val= (E_1.etype==int)? E_2.val div E_3.val : E_2.val / E_3.val
E→i	E.isFloat=flase E.val= E.etype==int? i.val : Float(i.val)
E→i.i	E.isFloat=true ; E.val=i.i.val

结合到语法分析遍中的计算

- S-属性文法适合于一遍扫描的自下而上分析
- L-属性文法可用于一遍扫描的自上而下分析，

语义动作

# E*(E +E	)#	reduce E→E+E	#3*(4+5	$E_1.val=E_2.val+E_3.val$
#i	*(i+i)#	reduce E→i	#i	E.val=i.val

pop t3
pop
pop t2
t1=t2+t3
push t1

pop t2
t1=*numval*(t2)
push t1

扩充LR分析过程计算综合属性

- 例：表达式求值
  - 将属性值保存在栈上；
  - 归约时根据产生式规则的语义方程进行；

在LR分析中继承先前计算过的综合属性

例1. i为继承属性；
s为综合属性

文法规则	语义规则
A → BC	C.i=*f*(B.s)

文法规则	语义规则
A → BDC	
B →…	计算B.s
D → ε	*saved_i*=*f*(valstack[top])
C →…	*saved_i* 可用

Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
#	i*(i+i)#	shift	#	
#i	*(i+i)#	reduce E→i	#i	E.val=i.val
#E	*(i+i)#	shift	#3	
#E*	(i+i)#	shift	#3*	
#E*(	i+i)#	shift	#3*(	
#E*(i	+i)#	reduce E→i	#3*(i	E.val=i.val
#E*(E	+i)#	shift	#3*(4	
# E*(E +	i)#	shift	#3*(4+	
# E*(E + i	)#	reduce E→i	#3*(4+i	E.val=i.val
# E*(E +E	)#	reduce E→E+E	#3*(4+5	$E_1.val=E_2.val+E_3.val$
# E*(E	)#	shift	#3*(9	
# E*(E)	#	reduce E→(E)	#3*(9)	$E_1.val=E_2.val$
# E*E	#	reduce E→E*E	#3*9	$E_1.val=E_2.val*E_3.val$
#E	#		#27	

在LR分析中继承先前计算过的综合属性

D → TL	L.dtype=T.dtype
T → int	T.dtype=integer
T → float	T.dtype=real
$L_1 \rightarrow L_2$, id	insert(id.name, L_1.dtype) L_2.dtype=L_1.dtype
L → id	insert(id.name, L.dtype)

- 综合属性dtype的值存放的位置是可以确定的；
  - 在归约L → id时；
  - 在归约$L_1 \rightarrow L_2$, id时；

D→TL	L.dtype=T.dtype
T → int	T.dtype=integer
T → float	T.dtype=real
$L_1 \rightarrow L_2$, id	insert(id.name, L_1.dtype) L_2.dtype=L_1.dtype
L → id	insert(id.name, L.dtype)

D → TL	
T → int	T.dtype=integer
T → float	T.dtype=real
$L_1 \rightarrow L_2$, id	insert(id.name, valstack[top-3])
L → id	insert(id.name,valstack[top-1])

只有综合属性的翻译模式

- 语义动作跟在产生式规则的后边

$E_1 \rightarrow E_2+T$	E_1.val=E_2.val+T.val

- $E_1 \rightarrow E_2+T\{E_1.val=E_2.val+T.val\}$

L-属性文法的定义

有属性$a_1,\ldots,a_k$的属性文法是L-属性文法，如果：

对于任何继承属性a_j和每个文法规则

$$X_0 \rightarrow X_1 \ldots X_n$$

与a_j关联的语义方程都是如下形式：

$$X_i.a_j = f_{ij}(X_0.a_1,\ldots,X_0.a_k,\ \ldots,\ X_{i-1}.a_1,\ldots,X_{i-1}.a_k)$$

只依赖于候选式中左边的那些文法符号的属性

L-属性文法的翻译模式

- 条件
  - 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来；
  - 一个动作不能引用这个动作右边的符号的综合属性；
  - 产生式左边非终结符的综合属性只有在它所引用的所有属性都计算出来以后才能计算；

翻译模式

- 把语义规则用花括号括起来插入到产生式右部合适的位置上，以明确使用语义规则进行计算的次序。

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

在移进时执行动作： 9-5+2 → 95-2+

在归约时执行动作： 9-5+2 → 952+-

例. 语义动作的位置

- S → A_1A_2{A_1.in:=1;A_2.in:=2}
- A →a {print(A.in)}

- S → {A_1.in:=1}A_1{A_2.in:=2}A_2
- A →a {print(A.in)}

本章小结

- 属性文法是描述语义的一种手段；由属性和属性方程构成；属性刻画语义分析中用到的信息；属性方程是描述属性值的计算公式
- 属性的计算可是单独的遍或者放在语法分析遍中；
- 综合属性与继承属性；属性依赖图
- 单独的遍中计算属性值：后根遍历语法树与先跟遍历语法树；一般按照依赖图构造计算过程
- S-属性文法与L-属性文法；
- 在语法分析中计算属性值
- 描述语义的另外一种手段：翻译模式

习题

- P164
  - 1，2，5(2), 7