三、语义分析
(2. 属性文法)

魏恒峰

hfwei@nju.edu.cn

2023 年 04 月 21 日

Regular Expression (词法分析)

Context-Free Grammar (语法分析)

用什么样的文法刻画语言的**语义**?

Donald Knuth (1938 $\sim$)

Semantics of Context-Free Languages

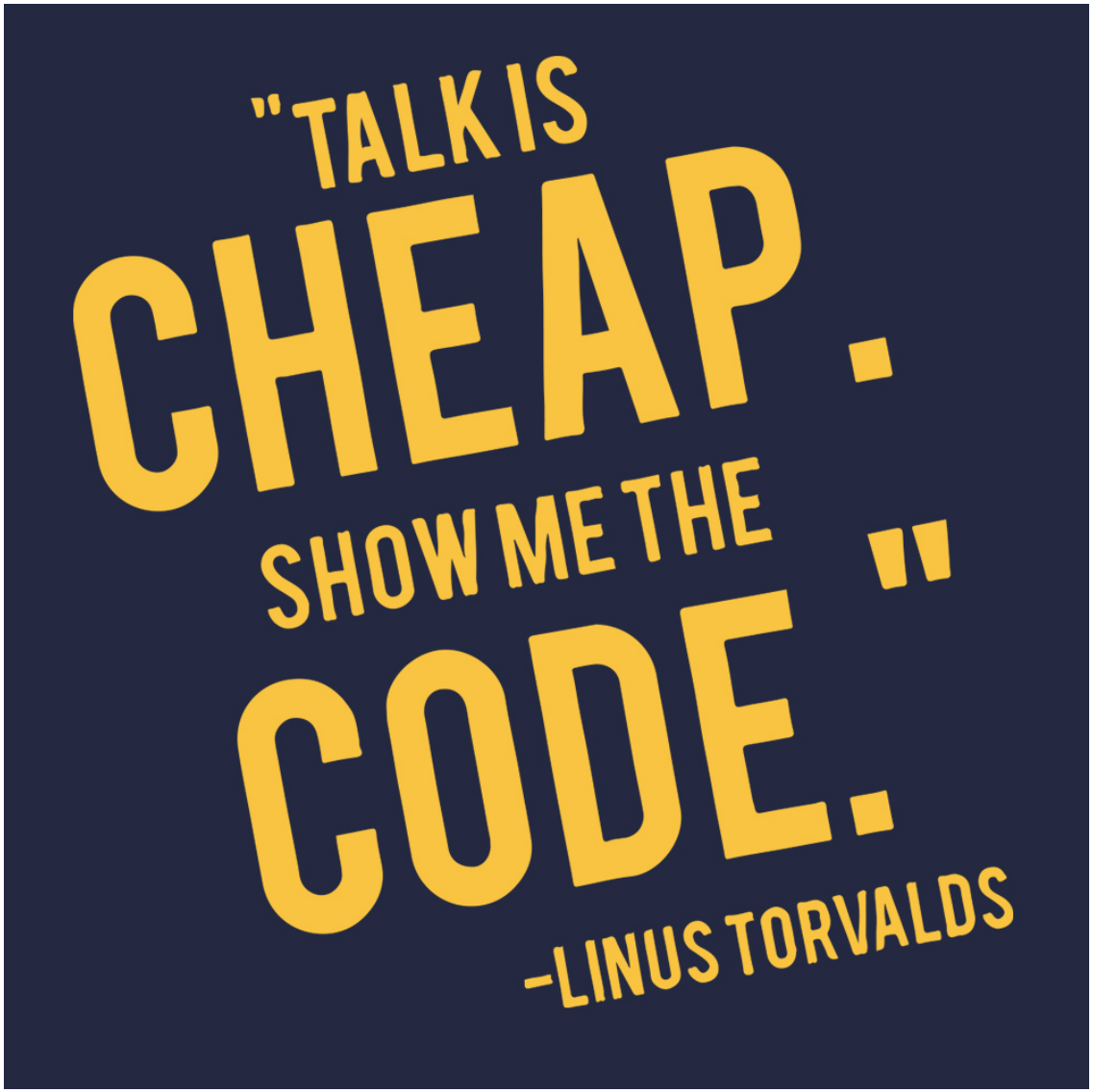
by

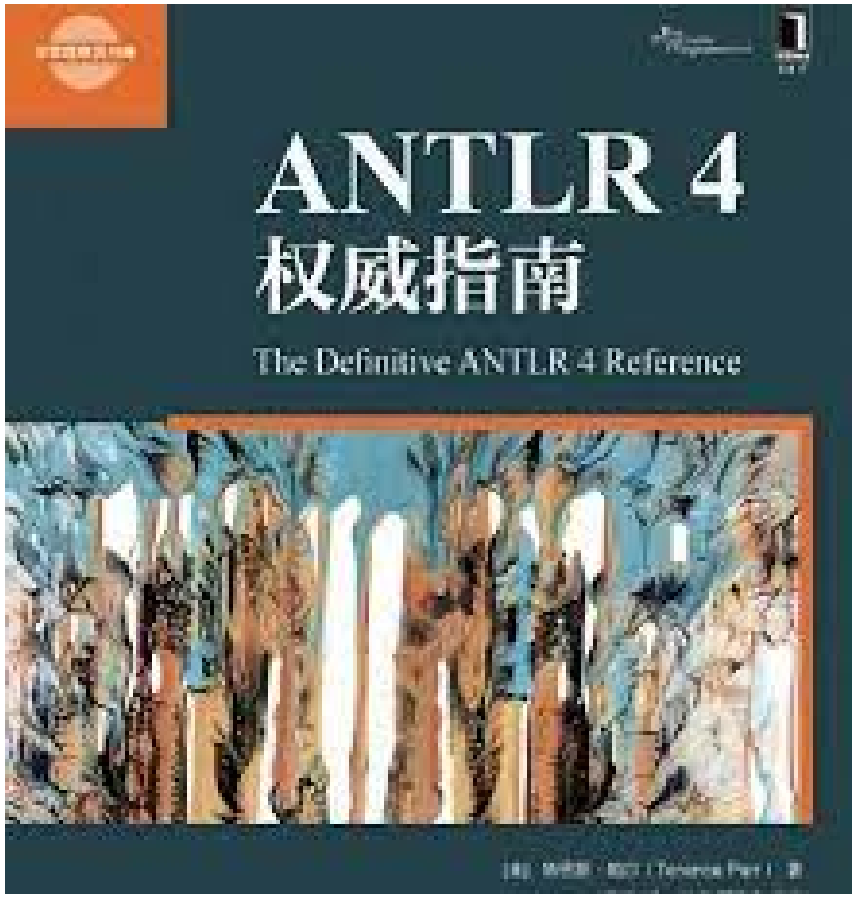
DONALD E. KNUTH
California Institute of Technology

ABSTRACT

"Meaning" may be assigned to a string in a context-free language by defining "attributes" of the symbols in a derivation tree for that string. The attributes can be defined by functions associated with each production in the grammar. This paper examines the implications of this process when some of the attributes are "synthesized", i.e., defined solely in terms of attributes of the *descendants* of the corresponding nonterminal symbol, while other attributes are "inherited", i.e., defined in terms of attributes of the *ancestors* of the nonterminal symbol. An algorithm is given which detects when such semantic rules could possibly lead to circular definition of some attributes. An example is given of a simple programming language defined with both inherited and synthesized attributes, and the method of definition is compared to other techniques for formal specification of semantics which have appeared in the literature.

属性文法 (Attribute Grammar): 为上下文无关文法赋予**语义**


"TALK IS
CHEAP.
SHOW ME THE
CODE."
-LINUS TORVALDS



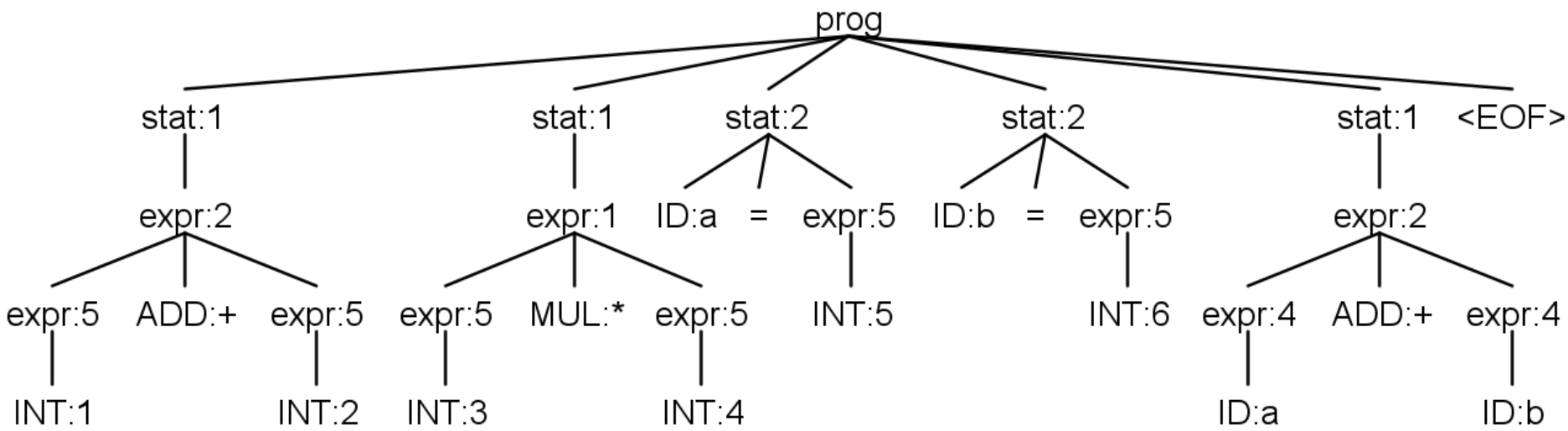
第 10 章：属性和动作

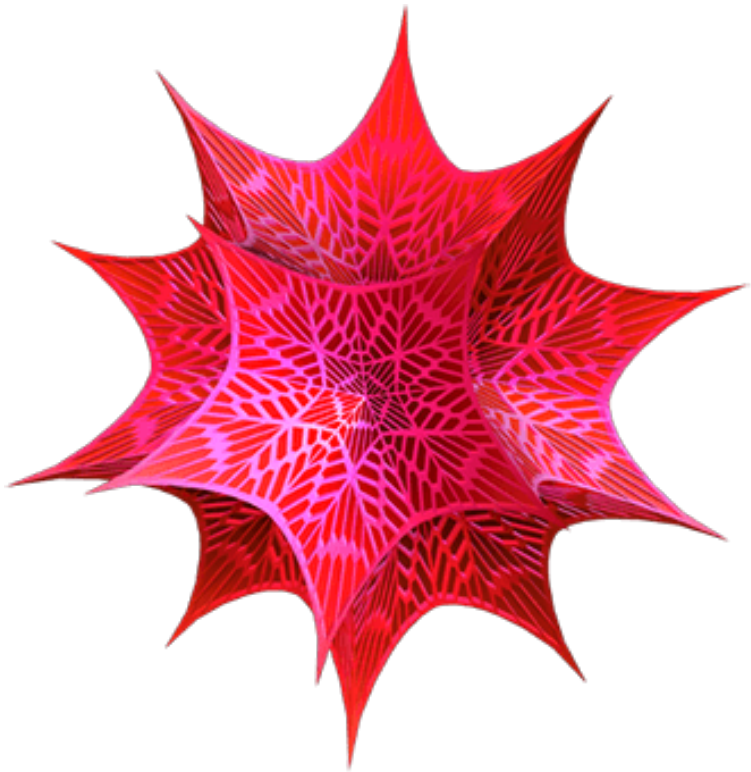
(交互式) 迷你计算器

Expr.g4

```
1 + 2
3 * 4
a = 5
b = 6
a + b
```

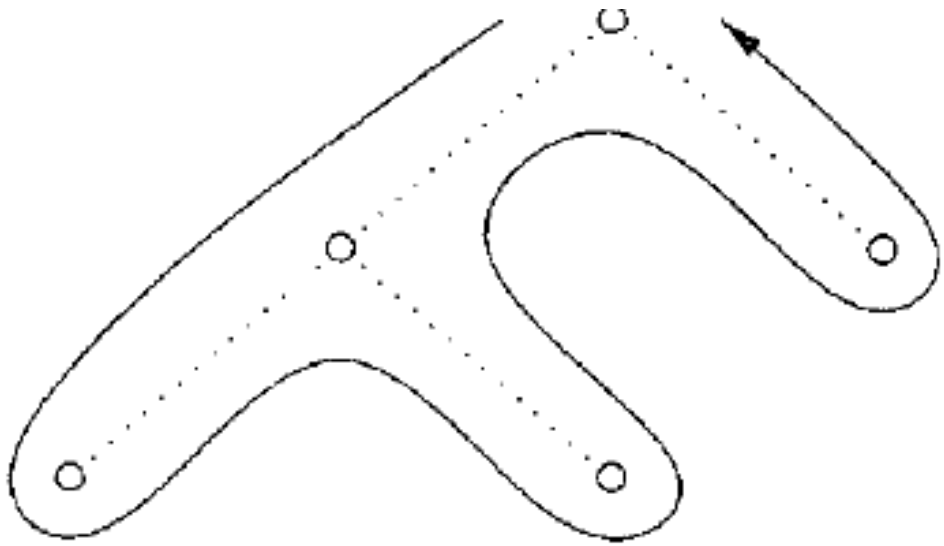
```
3
12
11
```





交互式

Offline 方式计算属性值: 已有语法分析树 (`calc`)



按照**从左到右**的**深度优先**顺序遍历语法分析树

关键: 在合适的时机执行合适的动作，计算相应的属性值

在**语法分析过程中**实现**属性文法**

$$B \to X\{a\}Y$$

语义动作嵌入的位置决定了**何时**执行该动作

基本思想: 一个动作在它**左边的**所有文法符号都**处理**过之后立刻执行

```
22  prog : stat+ ;
23
24  stat : expr          { System.out.println($expr.val); }
25       | ID '=' expr   { memory.put($ID.text, $expr.val); }
26       ;
27
28  expr returns [int val]
29      : l = expr op = ('*' | '/') r = expr { $val = eval($l.val, $r.val, $op.type); }
30      | l = expr op = ('+' | '-') r = expr { $val = eval($l.val, $r.val, $op.type); }
31      | '(' expr ')'                       { $val = $expr.val; }
32      | ID                                 { $val = memory.getOrDefault($ID.text, 0); }
33      | INT                                { $val = $INT.int; }
34      ;
```

ExprAG.g4

(交互式) 迷你计算器

ExprAGMain.java

“源码面前，了无秘密”

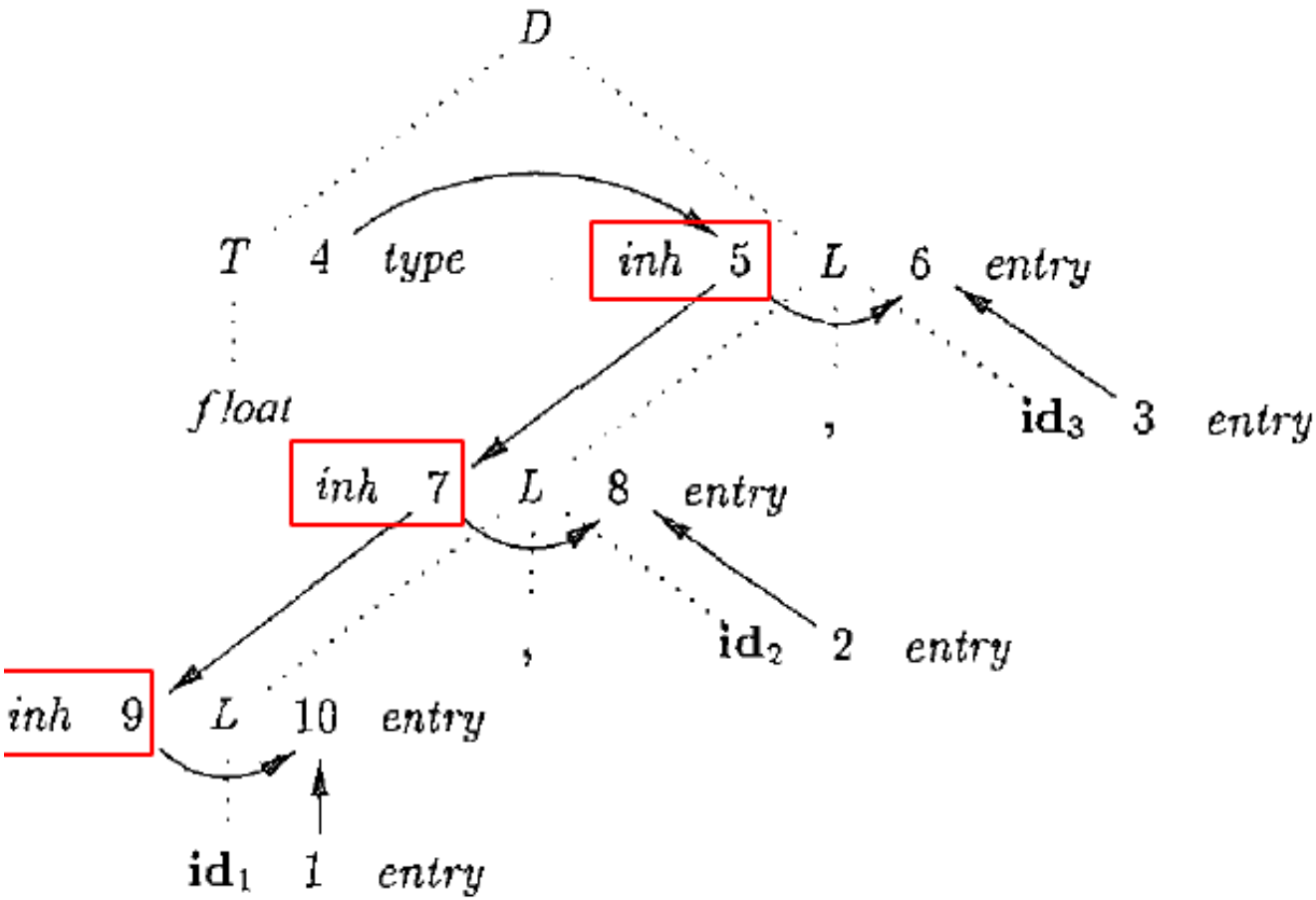
ExprAGParser.java

类型声明文法举例

	产生式	语义规则
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

$\mathbf{float}\ \mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$

$L.inh$ 将声明的类型沿着标识符列表向下传递



float $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$

```
7   decl : type vars[$type.text] ;
8   type : 'int'          # IntType
9        | 'float'        # FloatType
10       ;
11  // Unfortunately, ANTLR 4 does not support this:
12  // "rule vars is left recursive but
13  // doesn't conform to a pattern ANTLR can handle"
14  // See https://stackoverflow.com/q/76062088/1833118
15  vars[String type]
16       : left = vars[$type] ',' ID
17           { System.out.println($ID.text + " : " + $type); }
18       | ID
19           { System.out.println($ID.text + " : " + $type); }
20       ;
```

Fortunately, you can rewrite it as `vars : ID (',' ID)*`

Fortunately, you can rewrite it as `vars : ID (',' ID)*`

```
7  decl : type vars[$type.text] ;
8  type : 'int'          # IntType
9       | 'float'        # FloatType
10      ;
11 vars[String typeStr]
12      : ID { System.out.println($ID.text + " : " + $typeStr); }
13        (',' ID { System.out.println($ID.text + " : " + $typeStr); })*
14      ;
```

VarsDeclStarAG.g4

Definition (语法制导定义 (Syntax-Directed Definition; SDD))

SDD 是一个上下文无关文法和**属性**及**规则**的结合。

每个文法符号都可以关联多个属性

产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

每个产生式都可以关联一组规则

Definition (语法制导定义 (Syntax-Directed Definition; SDD))

SDD 是一个上下文无关文法和**属性**及**规则**的结合。

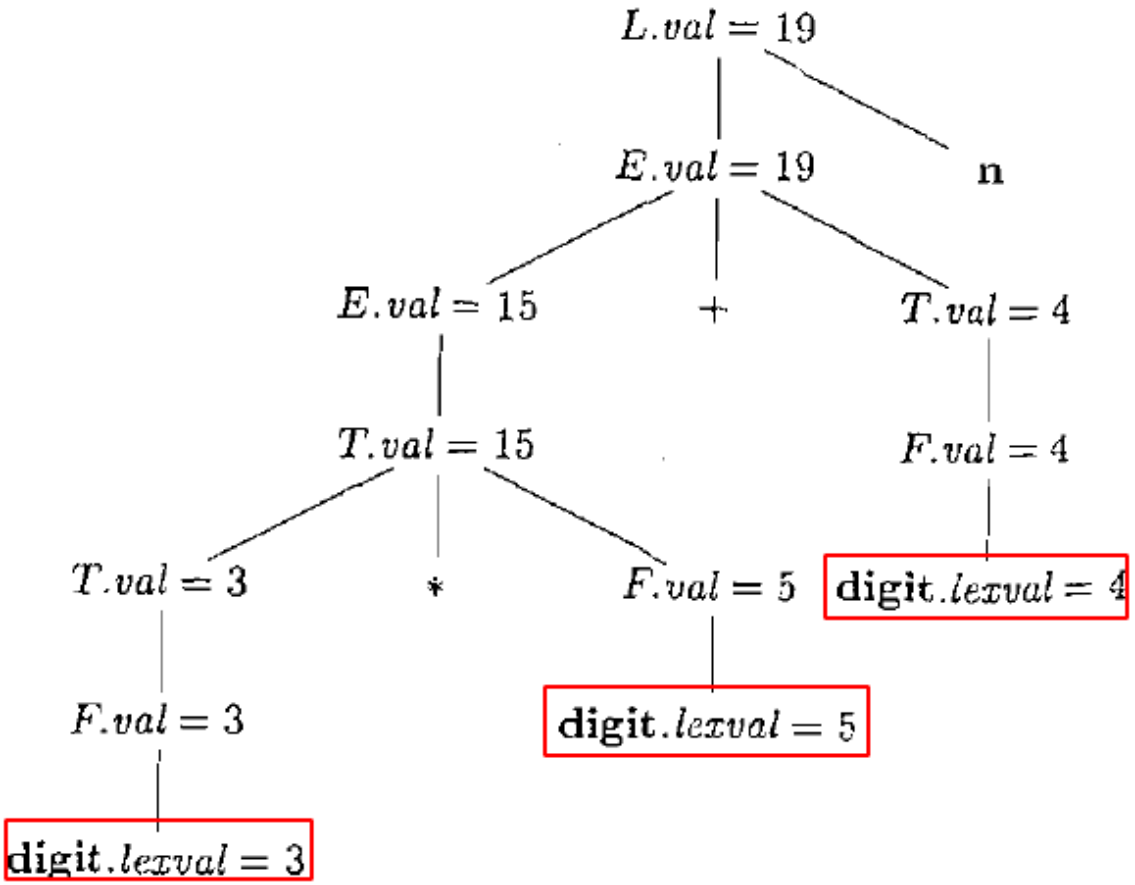
SDD **唯一确定**了语法分析树上每个非终结符节点的属性值

产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

SDD **没有**规定以什么方式、什么顺序计算这些属性值

注释 (annotated) 语法分析树: 显示了各个属性值的语法分析树

`ParseTreeProperty<Integer>` `put(ctx, ...), get(ctx, ...)`



$3 * 5 + 4$

Definition (综合属性 (Synthesized Attribute))

节点 N 上的**综合属性**只能通过 N 的子节点或 N 本身的属性来定义。

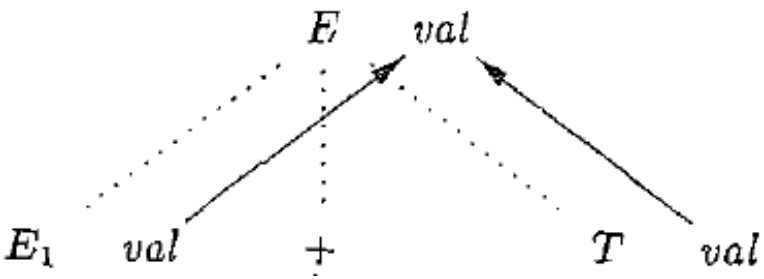
产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

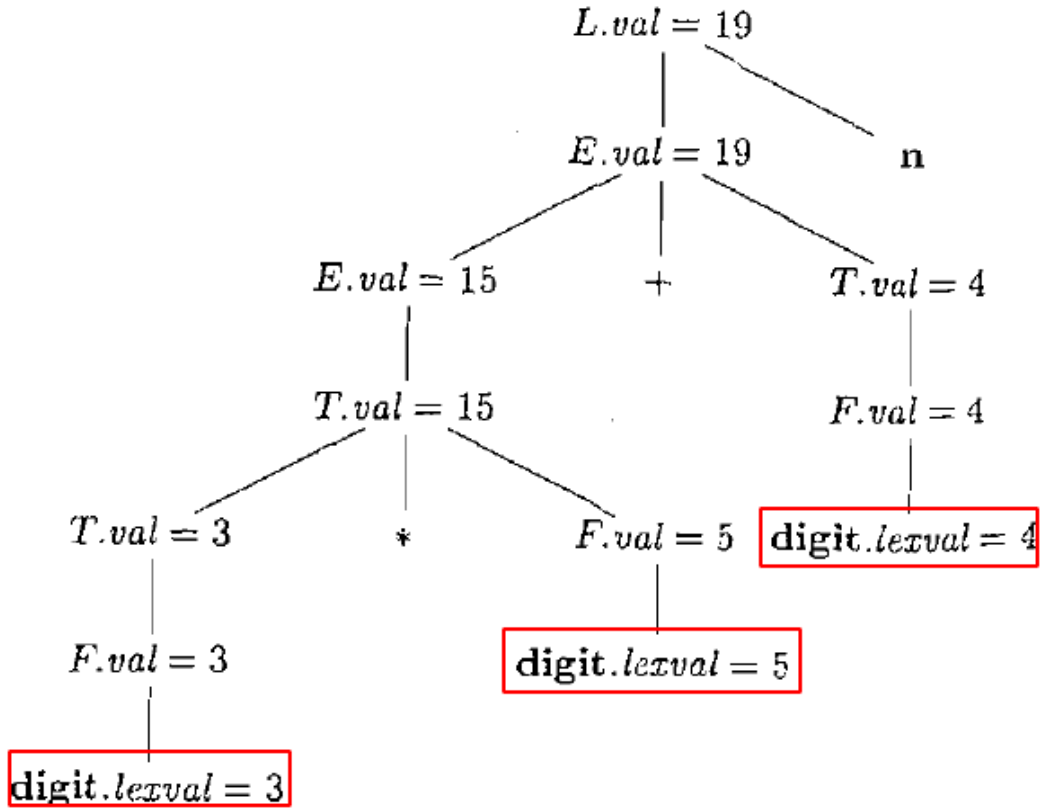
Definition (S 属性定义 (S-Attributed Definition))

如果一个 SDD 的每个属性都是综合属性, 则它是 S 属性定义。

依赖图用于确定一棵给定的语法分析树中各个属性实例之间的依赖关系

产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval





S **属性定义**的依赖图刻画了属性实例之间**自底向上**的信息流动

S **属性定义**的依赖图描述了属性实例之间**自底向上**的信息流

此类属性值的计算可以在自顶向下的 LL 语法分析**过程中**实现

在 LL 语法分析器中, 递归下降函数 A **返回** 时,
计算相应节点 A 的综合属性值

T' 有一个综合属性 syn 与一个继承属性 inh

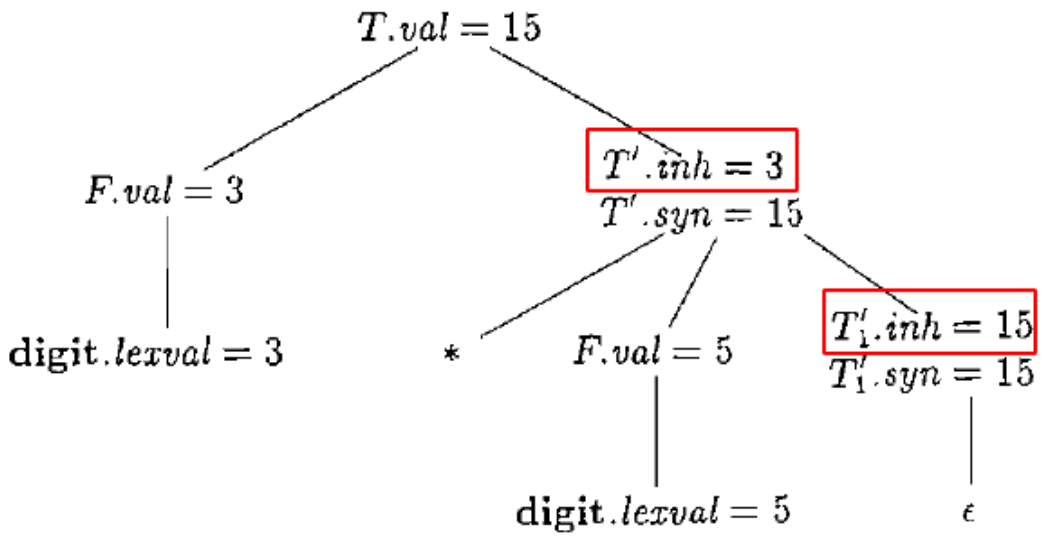
产生式	语义规则
1) $T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Definition (继承属性 (Inherited Attribute))

节点 N 上的**继承属性**只能通过N 的父节点、N 本身和 N 的兄弟节点上的属性来定义。

继承属性 $T'.inh$ 用于在表达式中从左向右传递中间计算结果

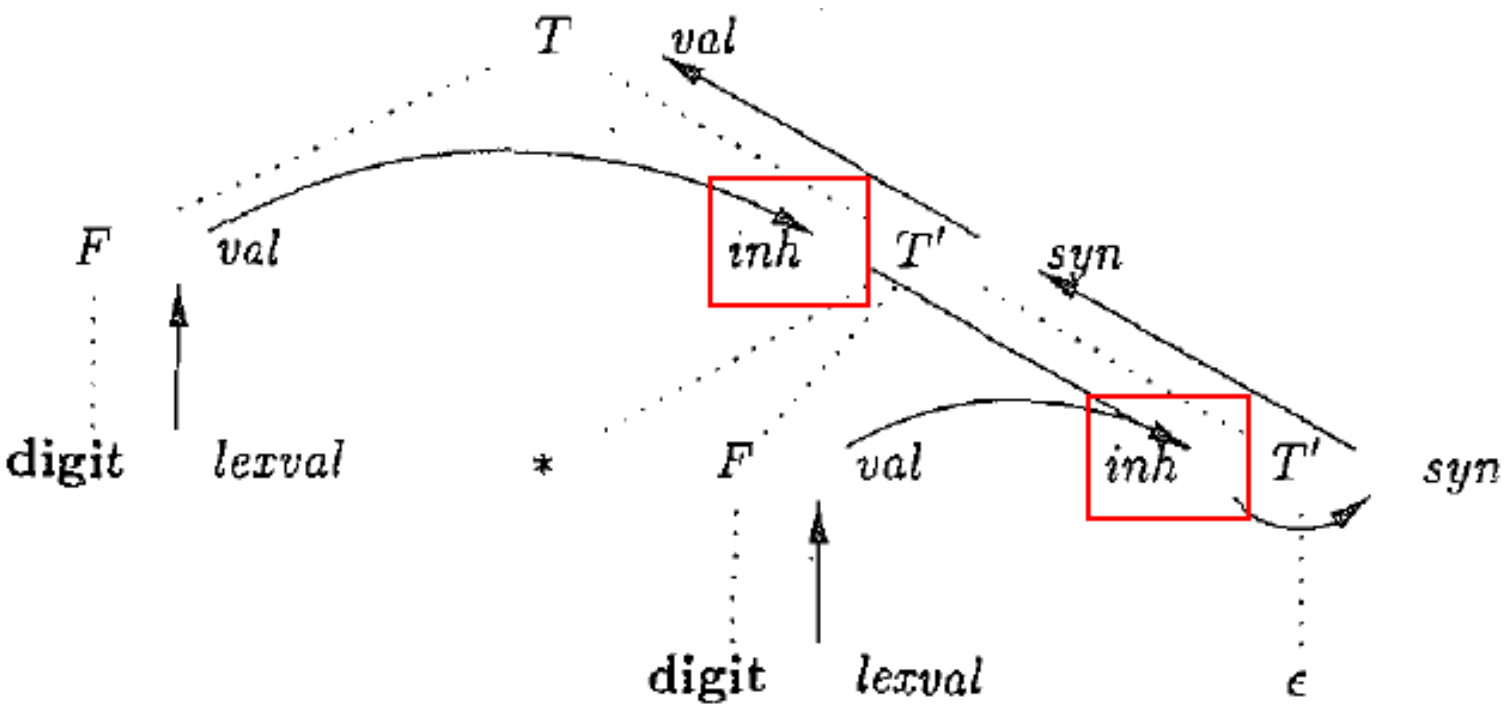
产生式	语义规则
1) $T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



$3 * 5$

在右递归文法下实现了左结合

依赖图用于确定一棵给定的语法分析树中各个属性实例之间的依赖关系



信息流向: 先从左向右、从上到下传递信息, 再从下到上传递信息

Definition (L 属性定义 (L-Attributed Definition))

如果一个 SDD 的每个属性

(1) 要么是综合属性,

(2) 要么是继承属性, 但是它的规则满足如下限制:
对于产生式 $A \to X_1 X_2 \ldots X_n$ 及其对应规则定义的继承属性 $X_i.a$,
则这个规则只能使用

(a) 和**产生式头** A关联的**继承**属性;

(b) 位于X_i **左边**的文法符号实例 X_1、X_2、...、X_{i-1} 相关的**继承**属性或**综合**属性;

(c) 和**这个** X_i **的实例本身**相关的继承属性或综合属性, 但是在由这个 X_i 的全部属性组成的依赖图中**不存在环**。

则它是 L 属性定义。

语法分析树上的**有序**信息流动

Learn by Examples

Definition (后缀表示 (Postfix Notation))

(1) 如果 E 是一个 **变量或常量**, 则 E 的后缀表示是 E 本身;

(2) 如果 E 是形如 E_1 **op** E_2 的表达式, 则 E 的后缀表示是 $E_1' E_2'$**op**, 这里 E_1' 和 E_2' 分别是 E_1 与 E_2 的后缀表达式;

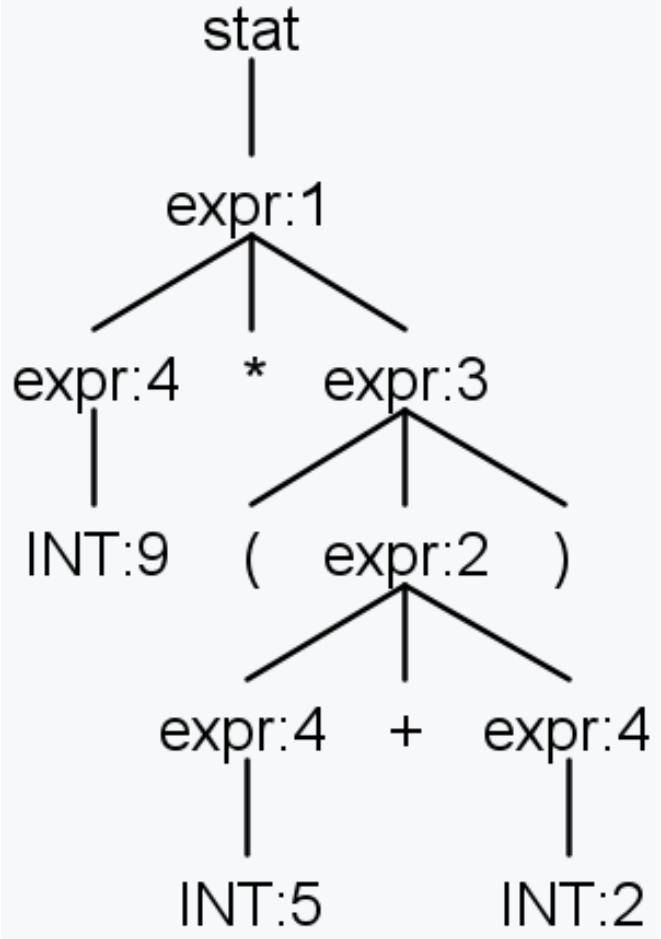
(3) 如果 E 是形如 (E_1) 的表达式, 则 E 的后缀表示是 E_1 的后缀表示。

$$(9 * 5) + 2 \implies 95 * 2+$$

$$9 * (5 + 2) \implies 952 + *$$

后缀表达式 S 属性定义

```
7   stat : expr { System.out.println($expr.postfix); }
8        ;
9
10  expr returns [String postfix]
11    : l = expr op = '*' r = expr { $postfix = $l.postfix + $r.postfix + $op.text; }
12    | l = expr op = '+' r = expr { $postfix = $l.postfix + $r.postfix + $op.text; }
13    | '(' expr ')'               { $postfix = $expr.postfix; }
14    | INT                        { $postfix = $INT.text; }
15    ;
```



$$9 * (5 + 2) \implies 952 + *$$

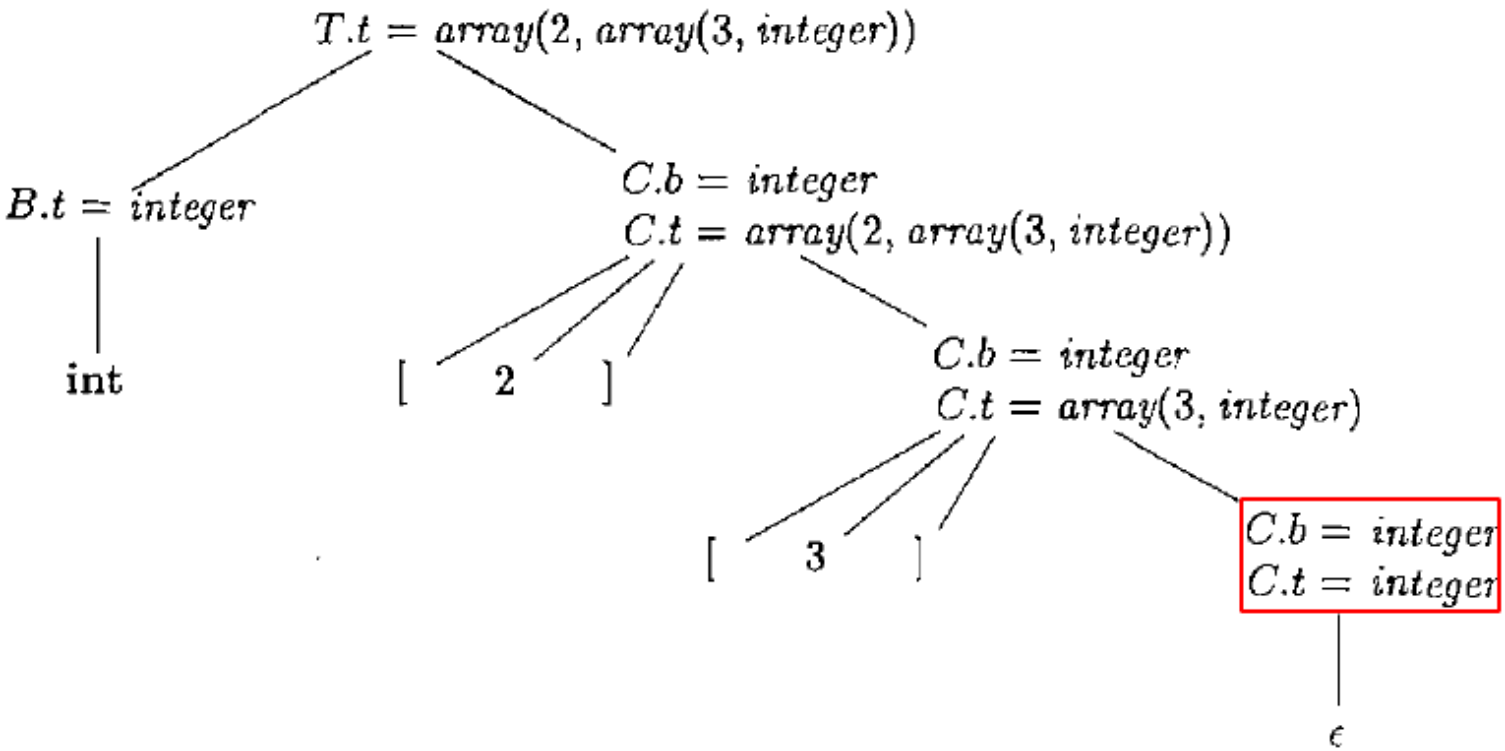
数组类型文法举例

产生式
$T \rightarrow B\ C$
$B \rightarrow \mathbf{int}$
$B \rightarrow \mathbf{float}$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$
$C \rightarrow \epsilon$

$\mathbf{int}[2][3]$

类型表达式 $(2, (3, \mathrm{int}))$

继承属性 $C.b$ 将一个基本类型沿着树向下传播

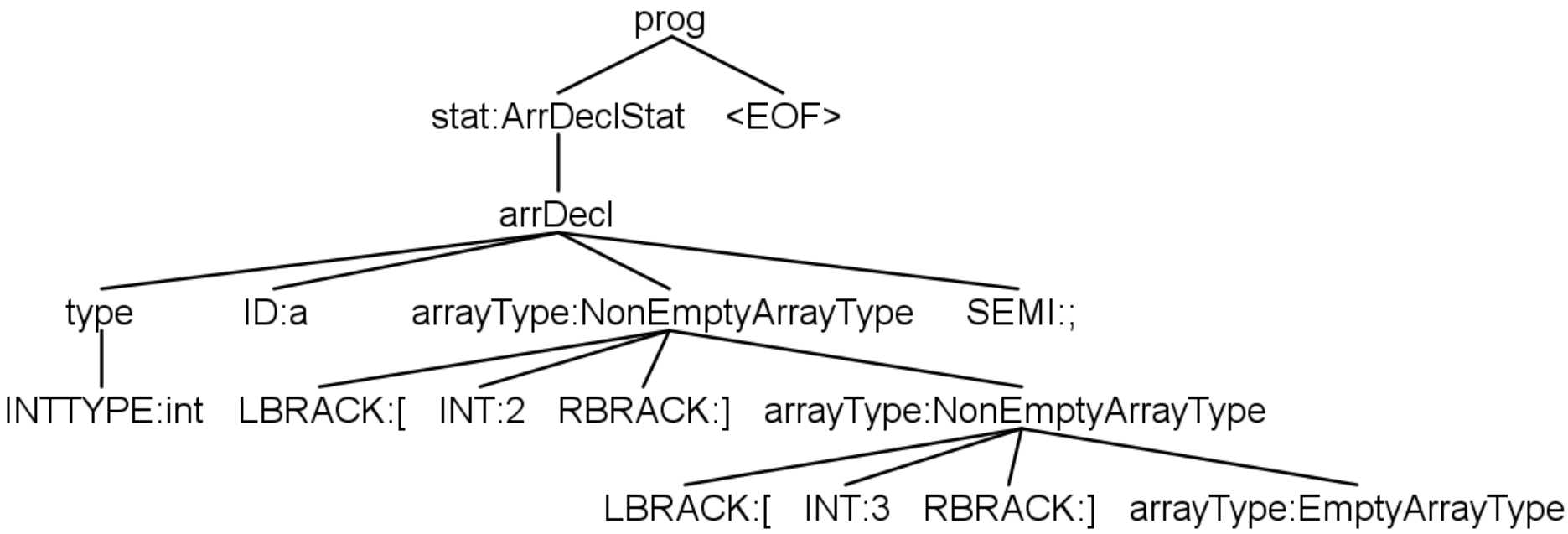


int[2][3]

综合属性 $C.t$ 收集最终得到的类型表达式

```
7  // OR: type ID ('[' INT ']')* ';'
8  arrDecl : basicType ID arrayType ';' ;
9  arrayType : '[' INT ']' arrayType       # NonEmptyArrayType
10           |                             # EmptyArrayType
11           ;
12
13 basicType : 'int' | 'double' ;
```

ArrayType.g4



$$\mathbf{int}[2][3]$$

```
7  // OR: type ID ('[' INT ']')* ';'
8  arrDecl : basicType ID arrayType[$basicType.text]
9    { System.out.println($ID.text + " : " + $arrayType.array_type); } ';' ;
10
11 arrayType[String basic_type]
12     returns [String array_type]
13   : '[' INT ']' arrayType[$basic_type]
14       { $array_type = "(" + $INT.int + ", " + $arrayType.array_type + ")"; }
15   |   { $array_type = $basic_type; }
16   ;
```

ArrayTypeAG.g4

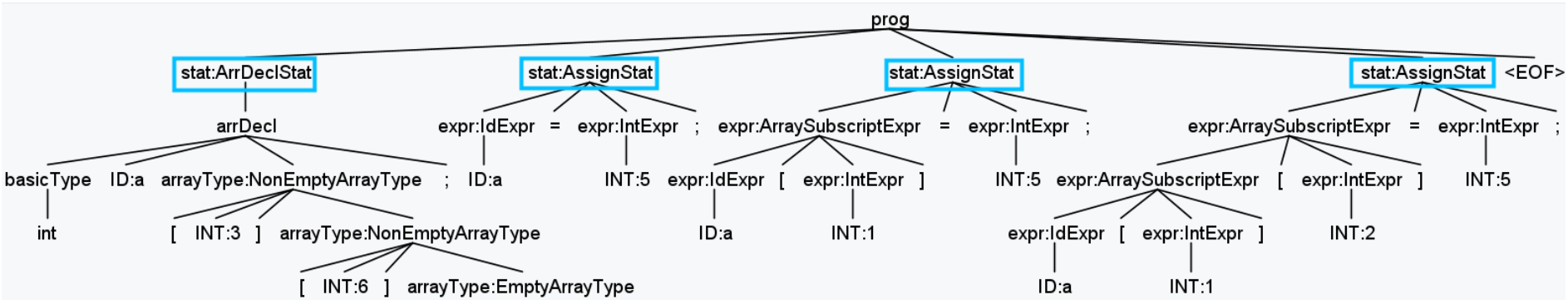
数组声明、数组引用、类型检查

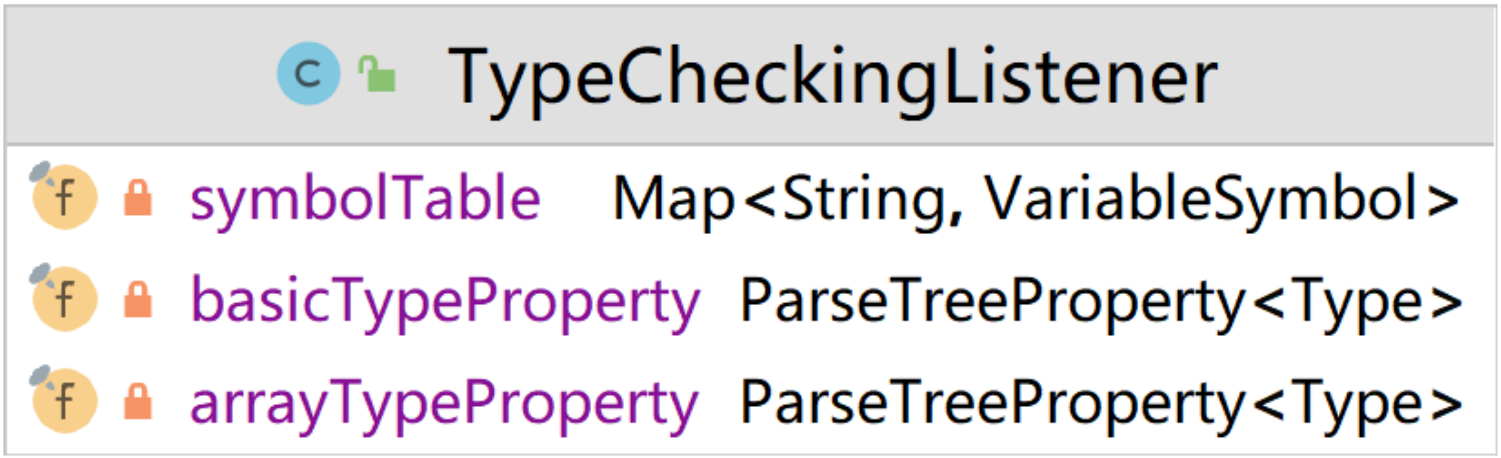
```
7  prog : stat* EOF ;
8
9  stat : varDecl                          # VarDeclStat
10      | arrDecl                          # ArrDeclStat
11      | lhs = expr '=' rhs = expr ';'    # AssignStat
12      ;
13
14 varDecl : basicType ID ';' ;
15 basicType : 'int' | 'double' ;
16
17 // OR: type ID ('[' INT ']')* ';'
18 arrDecl : basicType ID arrayType ';' ;
19 arrayType : '[' INT ']' arrayType      # NonEmptyArrayType
20           |                           # EmptyArrayType
21           ;
22
23 expr: primary = expr '[' subscript = expr ']'   # ArraySubscriptExpr
24     | ID                                       # IdExpr
25     | INT                                      # IntExpr
26     ;
```

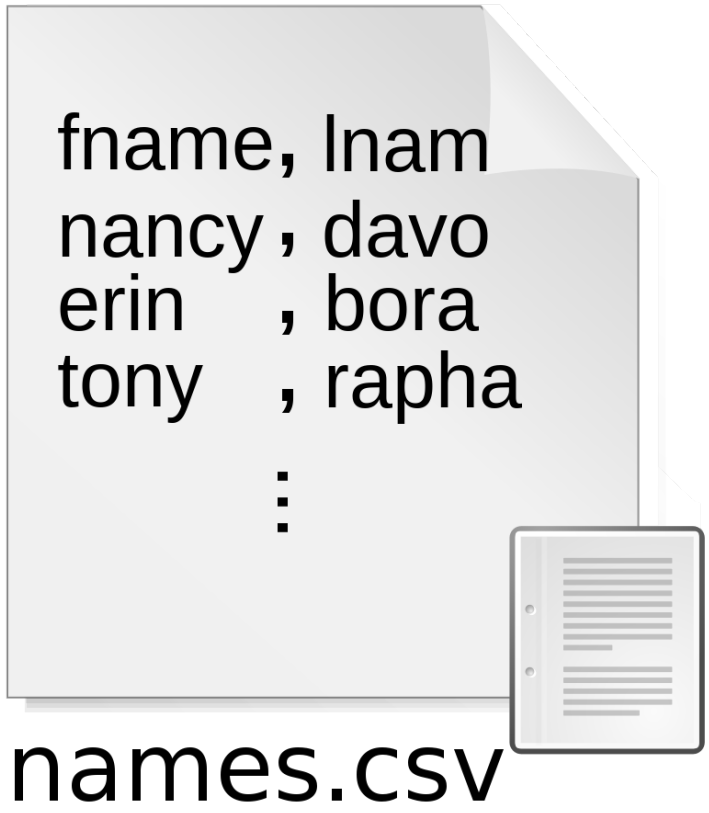
Array.g4

```
1  int a[3][6];
2  a = 5;
3  a[1] = 5;
4  a[1][2] = 5;
```

```
a : (3, (6, int))
(3, (6, int)) <=> int
 (6, int) <=> int
 int <=> int
```



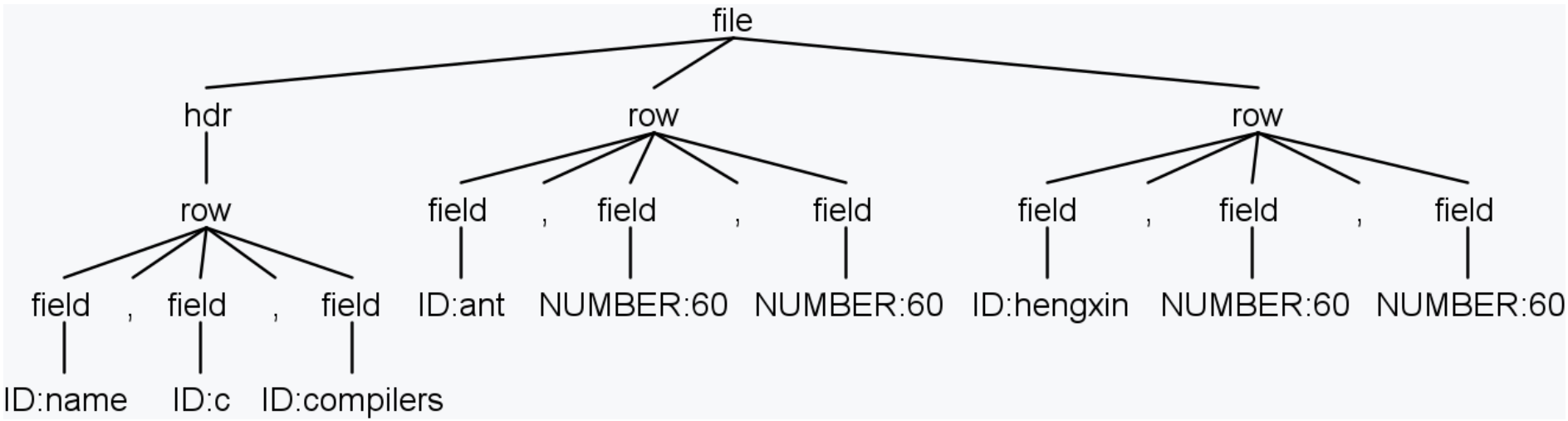

TypeCheckingListener
symbolTable Map<String, VariableSymbol>
basicTypeProperty ParseTreeProperty<Type>
arrayTypeProperty ParseTreeProperty<Type>



Comma-Separated Values

```
name, c, compilers
ant, 60, 60
hengxin, 60, 60
```

```
header: name,c,compilers
{c=60, name=ant, compilers=60}
{c=60, name=hengxin, compilers=60}
Totally 2 rows
Row token interval: 5..9
Row token interval: 10..14
```




```
file : hdr row+ ;
hdr : row ;
row : field (',' field)* ;
```

```
9   file
10  locals [int i = 0]
11    : hdr
12      { System.out.println("header: " + $hdr.text); }
13      (rs += row[$hdr.text.split(",")] { $i++; System.out.println($row.vals); } )+
14      { System.out.println("Totally " + $i + " rows");
15        for (RowContext r : $rs) {
16          System.out.println("Row token interval: " + r.getSourceInterval());
17        }
18      };
19  hdr : row[null] ;
20  row[String[] columns] returns [Map<String, String> vals = new HashMap<String, String>()]
21    locals [int col = 0]
22      : field { if ($columns != null) $vals.put(columns[$col++], $field.text); }
23        (',' field { if ($columns != null) $vals.put(columns[$col++], $field.text); } )* ;
```

CSVAG.g4

有符号二进制数文法

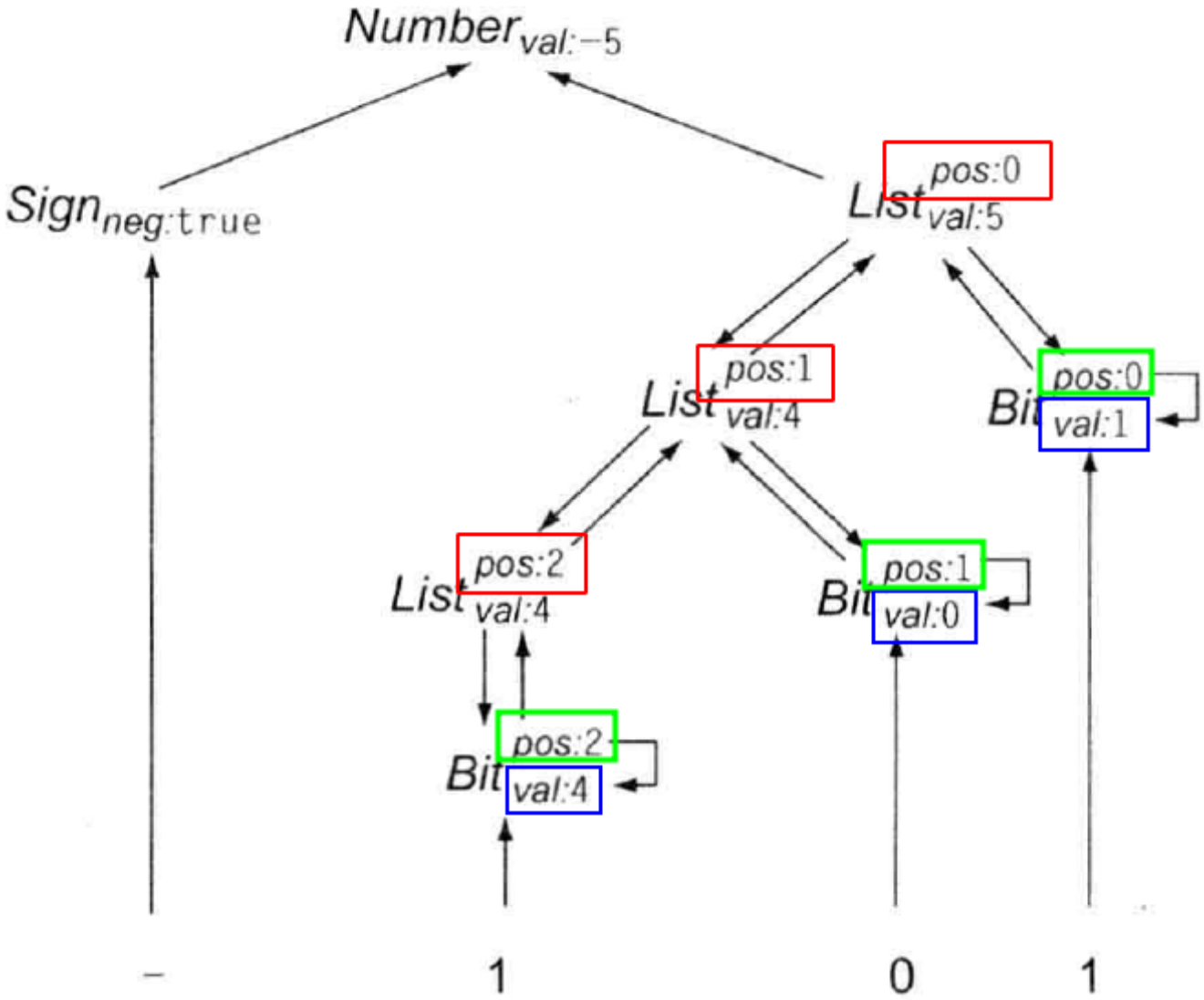
$$
P = \left\{
\begin{array}{lcl}
\mathit{Number} & \rightarrow & \mathit{Sign\ List} \\
\mathit{Sign} & \rightarrow & + \\
 & | & - \\
\mathit{List} & \rightarrow & \mathit{List\ Bit} \\
 & | & \mathit{Bit} \\
\mathit{Bit} & \rightarrow & 0 \\
 & | & 1
\end{array}
\right\}
$$

$$
\begin{aligned}
T &= \{+, -, 0, 1\} \\
NT &= \{\mathit{Number}, \mathit{Sign}, \mathit{List}, \mathit{Bit}\} \\
S &= \{\mathit{Number}\}
\end{aligned}
$$

$$-101_2 = -5_{10}$$

有符号二进制数 L 属性定义

	产生式	属性规则
1	$Number \rightarrow Sign\ List$	$List.position \leftarrow 0$ if $Sign.negative$ then $Number.value \leftarrow -List.value$ else $Number.value \leftarrow List.value$
2	$Sign \rightarrow +$	$Sign.negative \leftarrow false$
3	$Sign \rightarrow -$	$Sign.negative \leftarrow true$
4	$List \rightarrow Bit$	$Bit.position \leftarrow List.position$ $List.value \leftarrow Bit.value$
5	$List_0 \rightarrow List_1\ Bit$	$List_1.position \leftarrow List_0.position + 1$ $Bit.position \leftarrow List_0.position$ $List_0.value \leftarrow List_1.value + Bit.value$
6	$Bit \rightarrow 0$	$Bit.value \leftarrow 0$
7	$Bit \rightarrow 1$	$Bit.value \leftarrow 2^{Bit.position}$


Number val:−5
Sign neg:true
List val:5
pos:0
List val:4
pos:1
Bit
pos:0
val:1
List val:4
pos:2
Bit
pos:1
val:0
Bit
pos:2
val:4
−
1
0
1

$$-101_2 = -5_{10}$$

Definition (语法制导的翻译方案 (Syntax-Directed Translation Scheme; SDT))

SDT 是在其产生式体中嵌入**语义动作**的上下文无关文法。

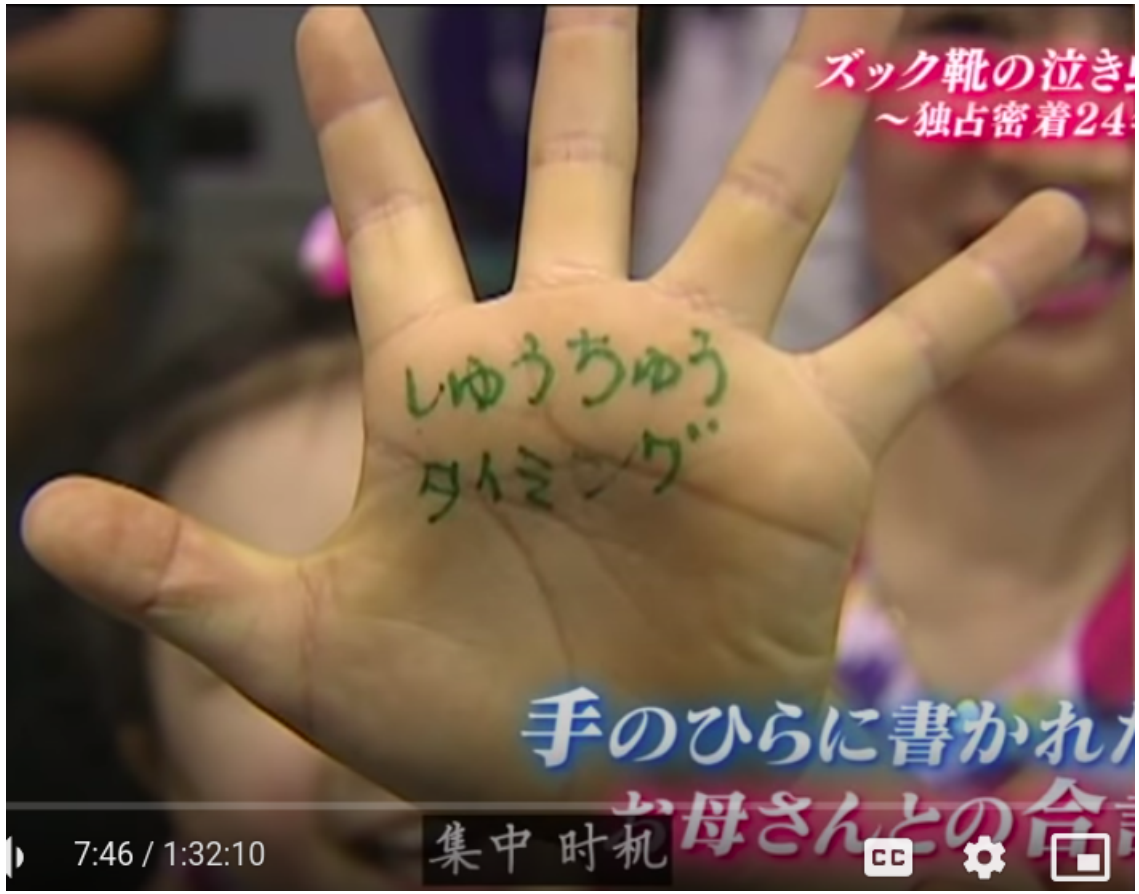
产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

L	$\to$	E n	{ print($E.val$); }
E	$\to$	$E_1 + T$	{ $E.val = E_1.val + T.val$; }
E	$\to$	T	{ $E.val = T.val$; }
T	$\to$	$T_1 * F$	{ $T.val = T_1.val \times F.val$; }
T	$\to$	F	{ $T.val = F.val$; }
F	$\to$	(E)	{ $F.val = E.val$; }
F	$\to$	**digit**	{ $F.val =$ **digit**.$lexval$; }

语义动作

Q：如何将带有**语义规则**的 SDD 转换为带有**语义动作**的 SDT

时机 (Timing; タイミング)



语义动作嵌入在什么地方？这决定了何时执行语义动作。

S 属性定义

产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

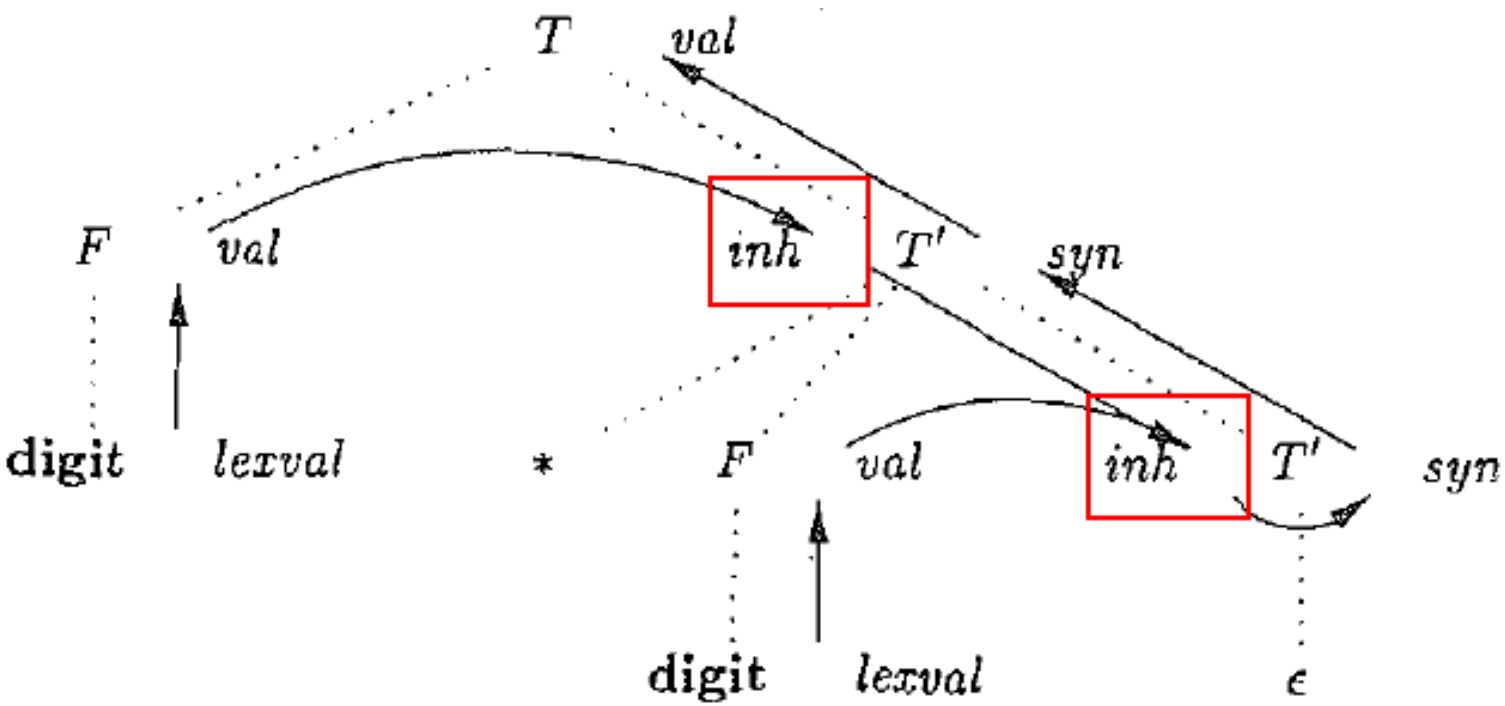
后缀翻译方案

L	$\to$	E n	{ print($E.val$); }
E	$\to$	$E_1 + T$	{ $E.val = E_1.val + T.val$; }
E	$\to$	T	{ $E.val = T.val$; }
T	$\to$	$T_1 * F$	{ $T.val = T_1.val \times F.val$; }
T	$\to$	F	{ $T.val = F.val$; }
F	$\to$	(E)	{ $F.val = E.val$; }
F	$\to$	**digit**	{ $F.val =$ **digit**.$lexval$; }

语义动作

后缀翻译方案: 所有动作都在产生式的最后

L 属性定义 与 LL 语法分析



$3 * 5$

$$A \to X_1 \cdots X_i \cdots X_n$$

原则: 从左到右处理各个 X_i 符号

对每个 X_i, 先计算**继承属性**, 后计算**综合属性**

递归下降子过程 $A \to X_1 \cdots X_i \cdots X_n$

- 在调用 X_i 子过程之前, 计算 X_i 的**继承属性**
  - 以 X_i 的继承属性为**参数**调用 X_i 子过程
- 在 X_i 子过程返回之前, 计算 X_i 的**综合属性**
  - 在 X_i 子过程结束时**返回** X_i 的综合属性

(左递归) S 属性定义

$$A \to A_1 Y \qquad A.a = g(A_1.a, Y.y)$$

$$A \to X \qquad A.a = f(X.x)$$

$$\boxed{XY^*}$$

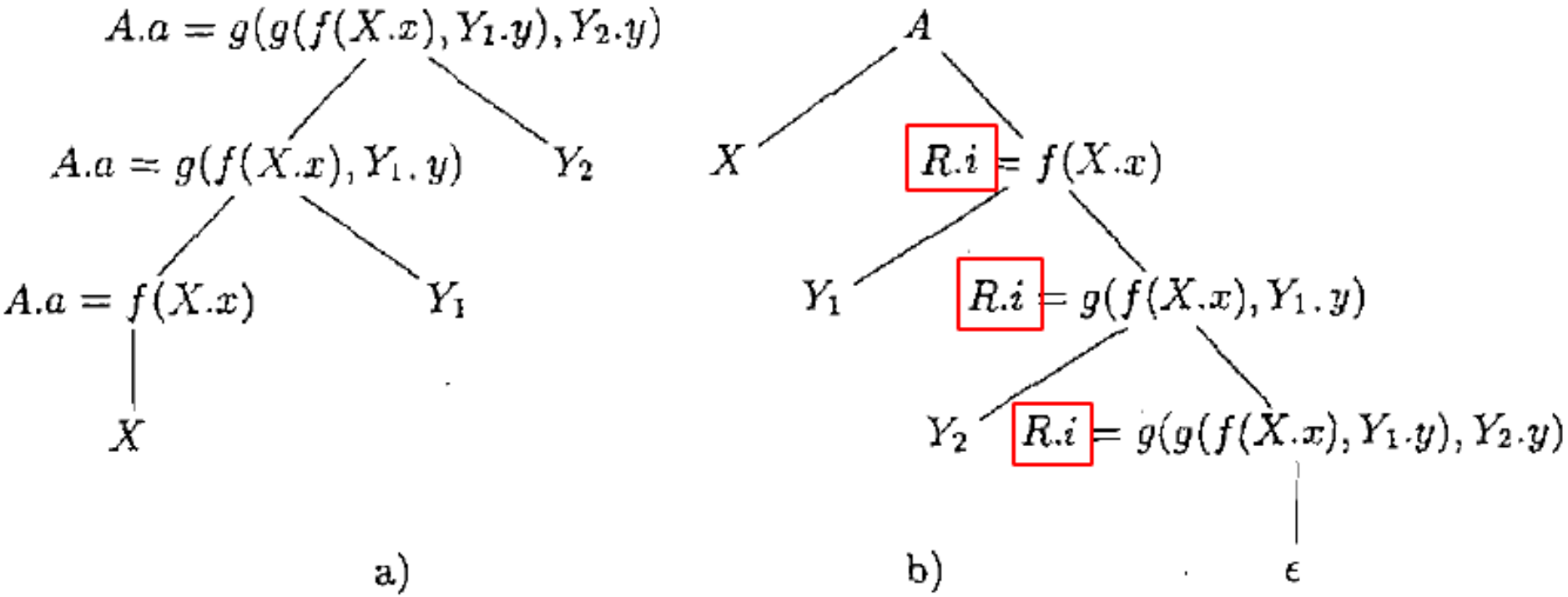
(右递归) L 属性定义

$$A \to XR \qquad R.i = f(X.x); \quad A.a = R.s$$

$$R \to YR_1 \qquad R_1.i = g(R.i, Y.y); \quad R.s = R_1.s$$

$$R \to \epsilon \qquad R.s = R.i$$

继承属性 $R.i$ 用于计算并传递中间结果



先计算继承属性，再计算综合属性

(右递归) L 属性定义

$A \to XR \qquad R.i = f(X.x); \quad A.a = R.s$

$R \to YR_1 \qquad R_1.i = g(R.i, Y.y); \quad R.s = R_1.s$

$R \to \epsilon \qquad R.s = R.i$

原则：继承属性在处理文法符号之前，综合属性在处理文法符号之后

L 属性定义的 SDT

$A \to X \quad \{R.i = f(X.x)\} \quad R \quad \{A.a = R.s\}$

$R \to Y \quad \{R_1.i = g(R.i, Y.y)\} \quad R_1 \quad \{R.s = R_1.s\}$

$R \to \epsilon \quad \{R.s = R.i\}$

$$A \to X \quad \{R.i = f(X.x)\} \quad R \quad \{A.a = R.s\}$$

$$R \to Y \quad \{R_1.i = g(R.i, Y.y)\} \quad R_1 \quad \{R.s = R_1.s\}$$

$$R \to \epsilon \quad \{R.s = R.i\}$$

1: **procedure** $A()$ ▷ A 是开始符号, 无需继承属性做参数
2: **if** token $= ?$ **then** ▷ 假设选择 $A \to XR$ 产生式
3: $X.x \leftarrow \text{MATCH}(X)$ ▷ 假设 X 是终结符, 返回综合属性
4: $R.i \leftarrow f(X.x)$ ▷ 先计算 $R.i$ 继承属性
5: $R.s \leftarrow R(R.i)$ ▷ 递归调用子过程 $R(R.i)$
6: **return** $R.s$ ▷ 返回 $A.a \leftarrow R.s$ 综合属性

$$A \to X \quad \{R.i = f(X.x)\} \quad R \quad \{A.a = R.s\}$$

$$R \to Y \quad \{R_1.i = g(R.i, Y.y)\} \quad R_1 \quad \{R.s = R_1.s\}$$

$$R \to \epsilon \quad \{R.s = R.i\}$$

```
1: procedure R(R.i)                          ▷ R 使用继承属性 R.i 做参数
2:     if token = ? then                     ▷ 假设选择 R → YR 产生式
3:         Y.y ← MATCH(Y)                    ▷ 假设 Y 是终结符, 返回综合属性
4:         R.i ← g(R.i, Y.y)                 ▷ 先计算 R.i 继承属性
5:         R.s ← R(R.i)                      ▷ 递归调用子过程 R(R.i)
6:         return R.s                        ▷ 返回综合属性
7:     else if token = ? then                ▷ 假设选择 R → ϵ 产生式
8:         return R.i                        ▷ 返回 R.s ← R.i 综合属性
```

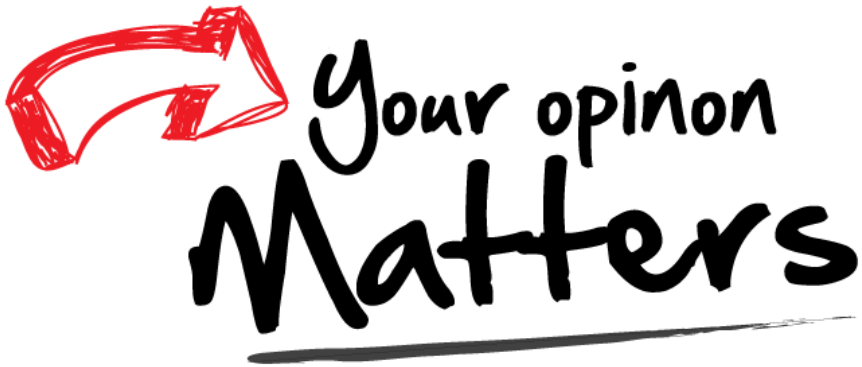
What is the difference between ANTLR 3 and 4?

Another big difference is that we discourage the use of actions directly within the grammar because ANTLR 4 automatically generates listeners and visitors for you to use that trigger method calls when some phrases of interest are recognized during a tree walk after parsing. See also Parse Tree Matching and XPath.

Q: What are the main design decisions in ANTLR4?

Ease-of-use over performance. I will worry about performance later. Simplicity over complexity. For example, I have taken out explicit/manual AST construction facilities and the tree grammar facilities. For 20 years I've been trying to get people to go that direction, but I've since decided that it was a mistake. It's much better to give people a parser generator that can automatically build trees and then let them use pure code to do whatever tree walking they want. People are extremely familiar and comfortable with visitors, for example.


Thank
You!



Office 926

hfwei@nju.edu.cn