(2.)

hfwei@nju.edu.cn

20221212/14

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.

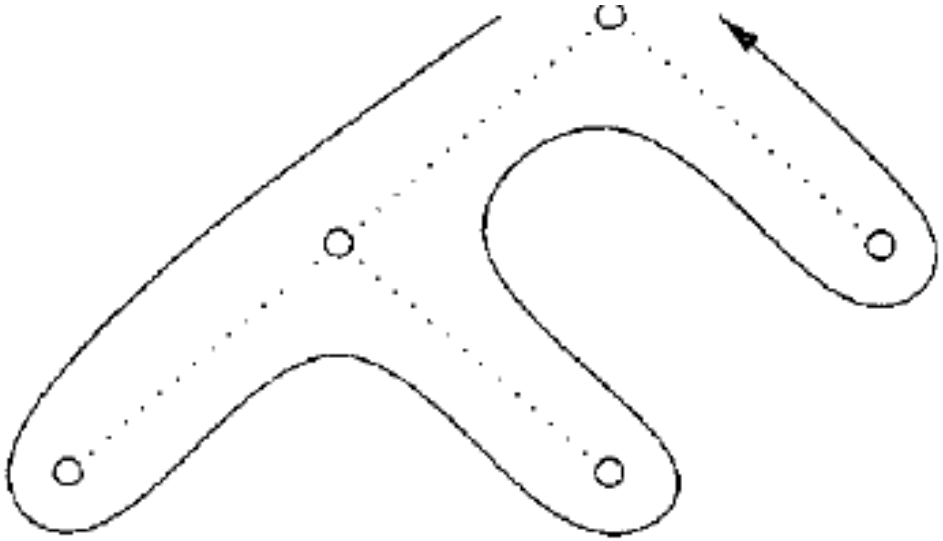
(Attribute Grammar):

: ?

DFS (around 1972)

Robert Tarjan (1948 $\sim$)

Offline :

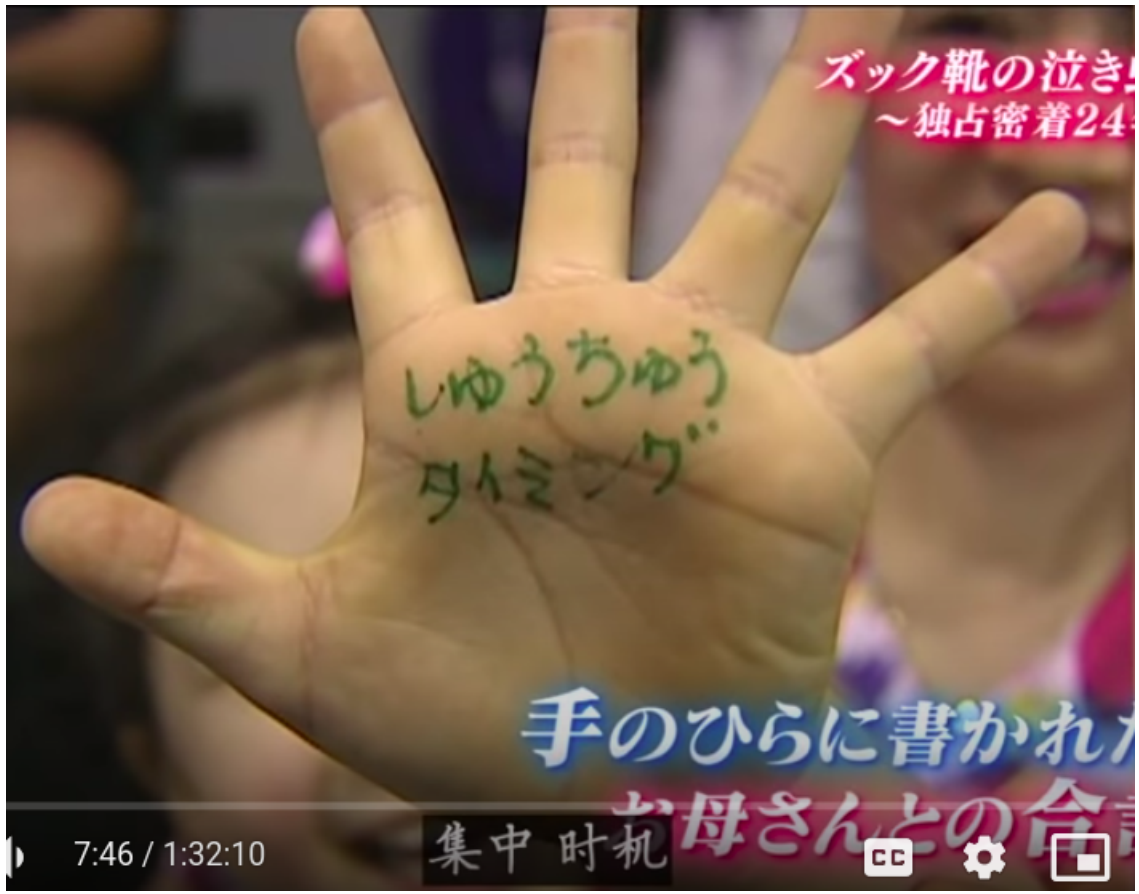


:

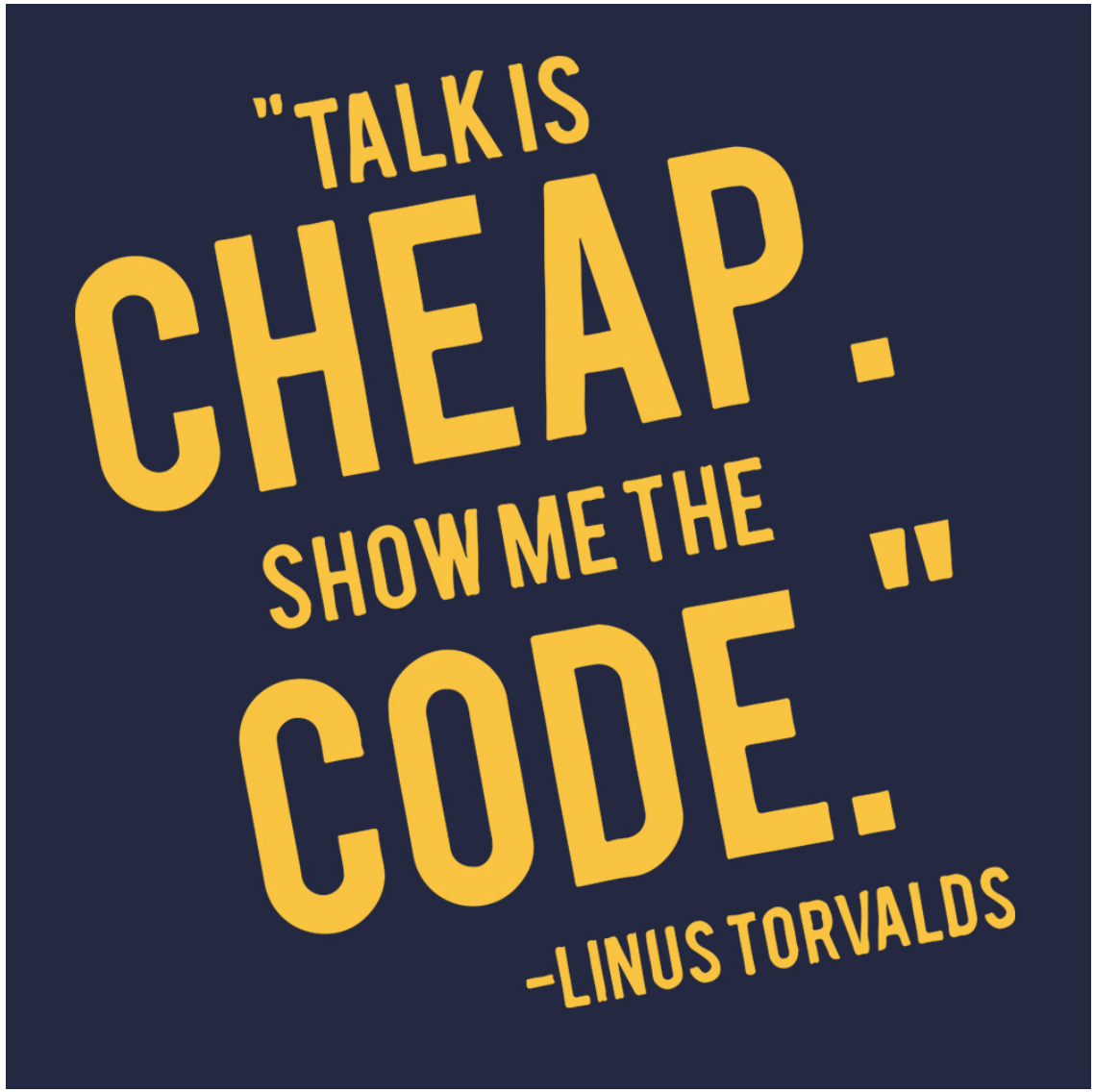
$$B \to X\{a\}Y$$

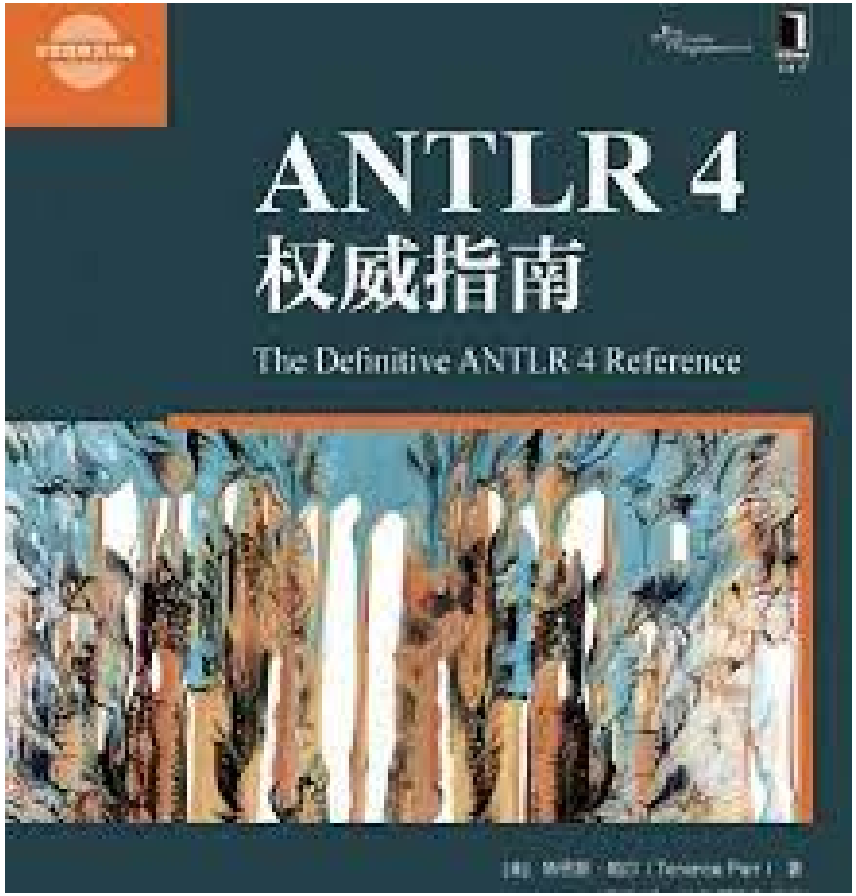
:

(Timing;)



?


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



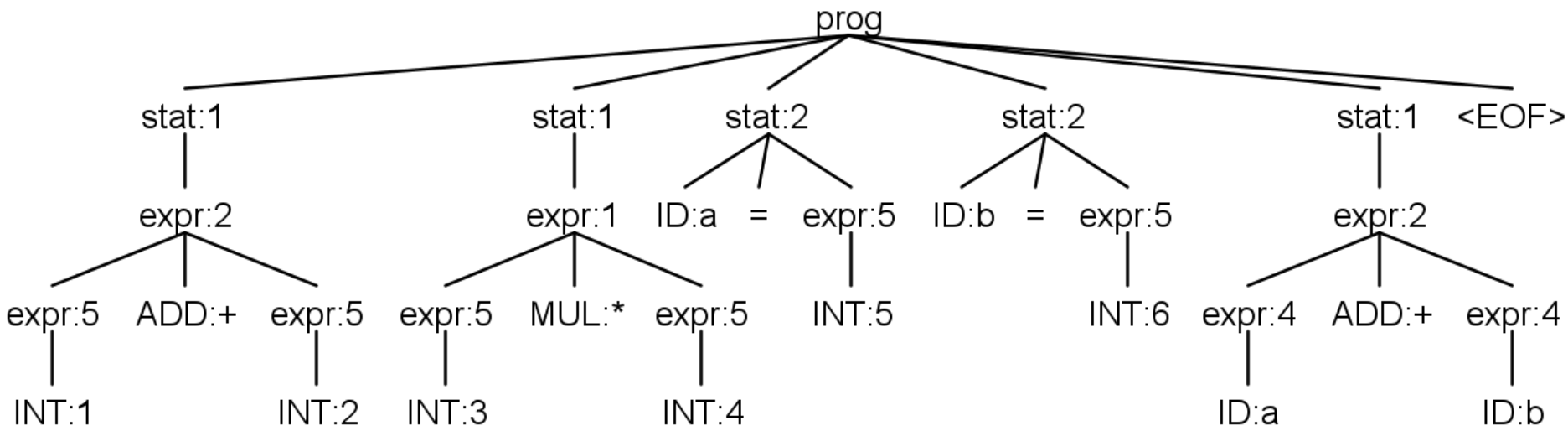
10 :

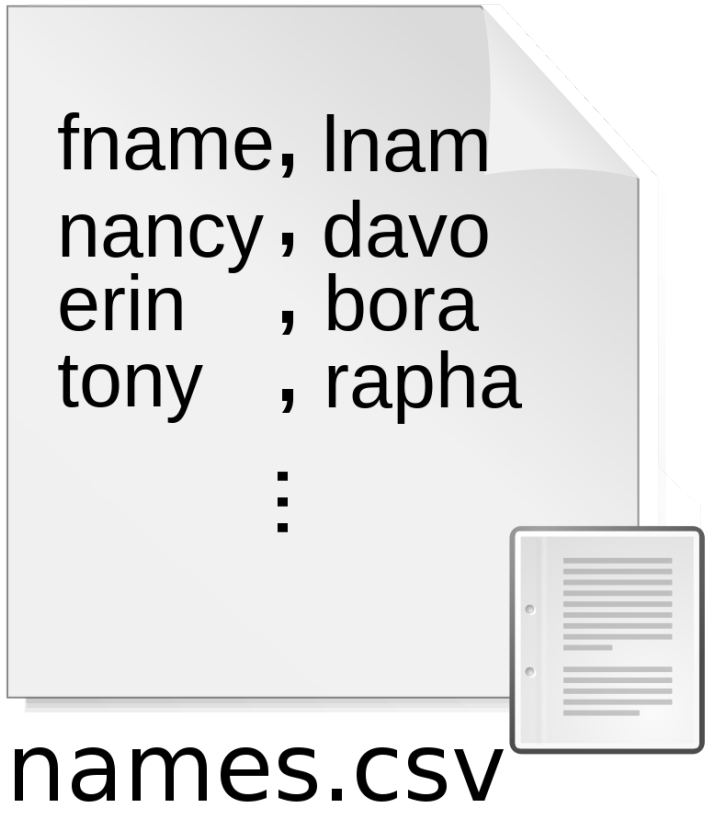

ON/AC
M+
M−
MRC
GT
OFF
7
8
9
÷
4
5
6
×
%
1
2
3
−
MU
0
00
.
+
=

()

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

```
3
12
11
```

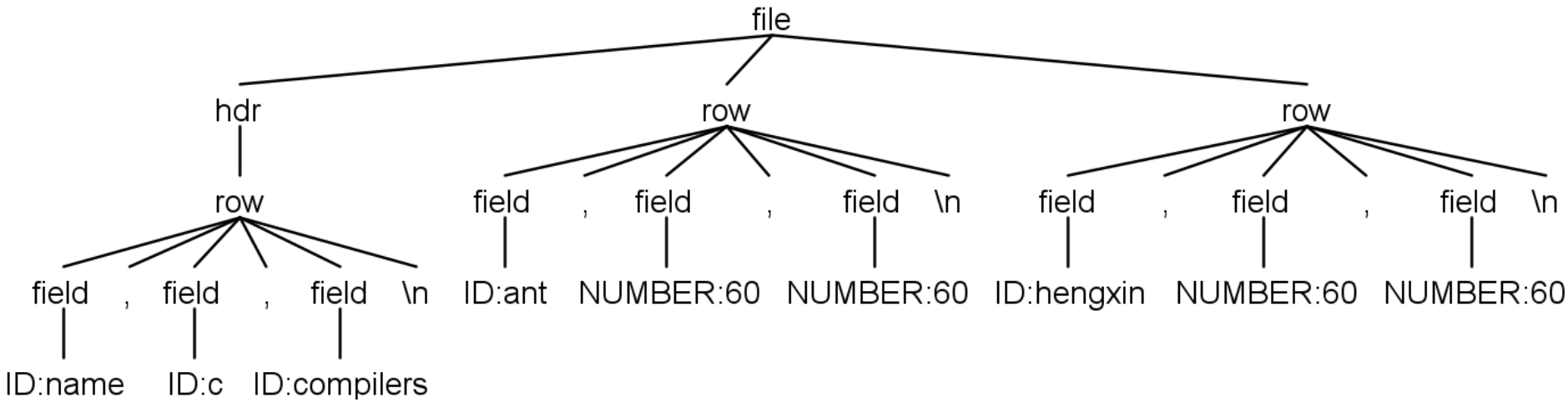




Comma-Separated Values

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

```
header: name,c,compilers
values = {name=ant, c=60, compilers=60}
values = {name=hengxin, c=60, compilers=60}
Totally 2 rows
Row token interval : 7..13
Row token interval : 14..20
```




```
file : hdr row+ ;
hdr : row ;
row : field (',' field)* '\r'? '\n' ;
```

Definition ((Syntax-Directed Definition; SDD))

SDD

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 ((Syntax-Directed Definition; SDD))

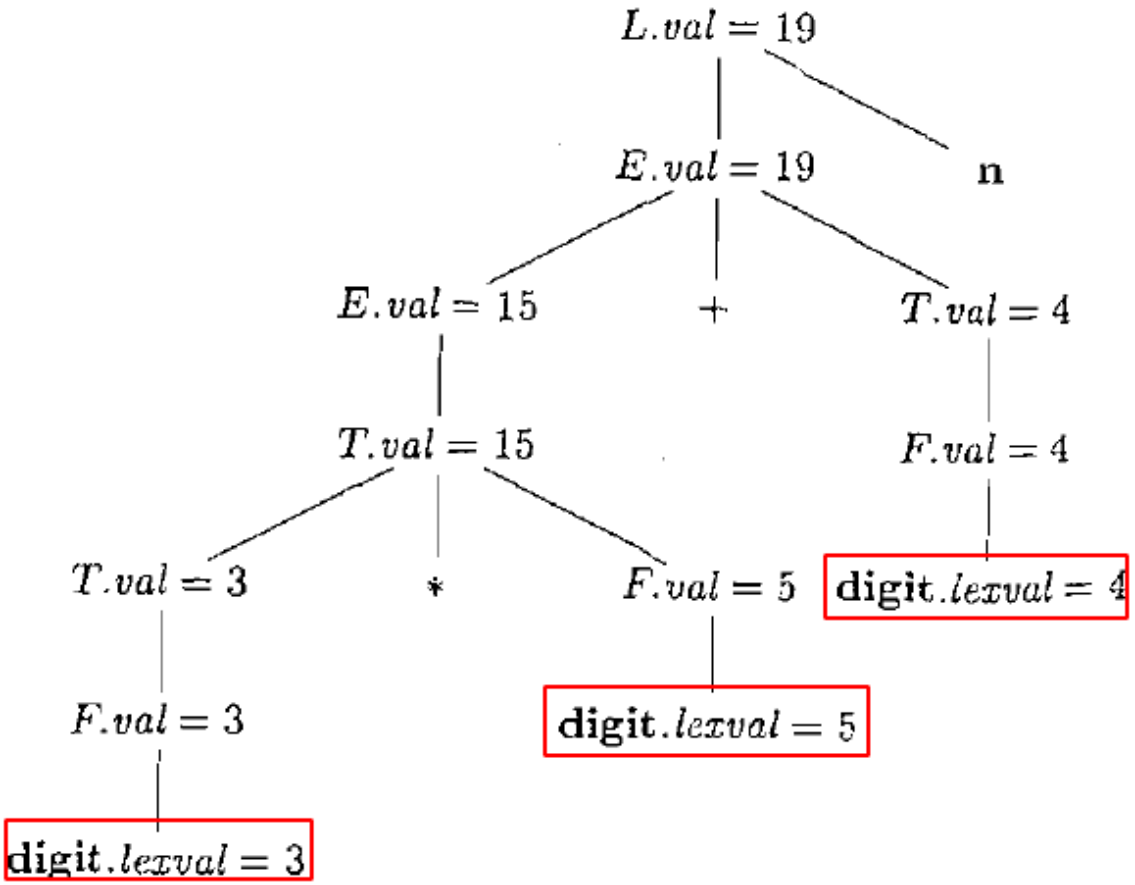
SDD

SDD

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

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



$3 * 5 + 4$

Definition ((Synthesized Attribute))

N NN

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 ((Synthesized Attribute))

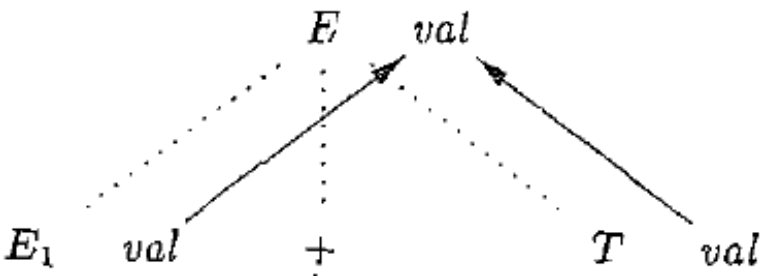
N NN

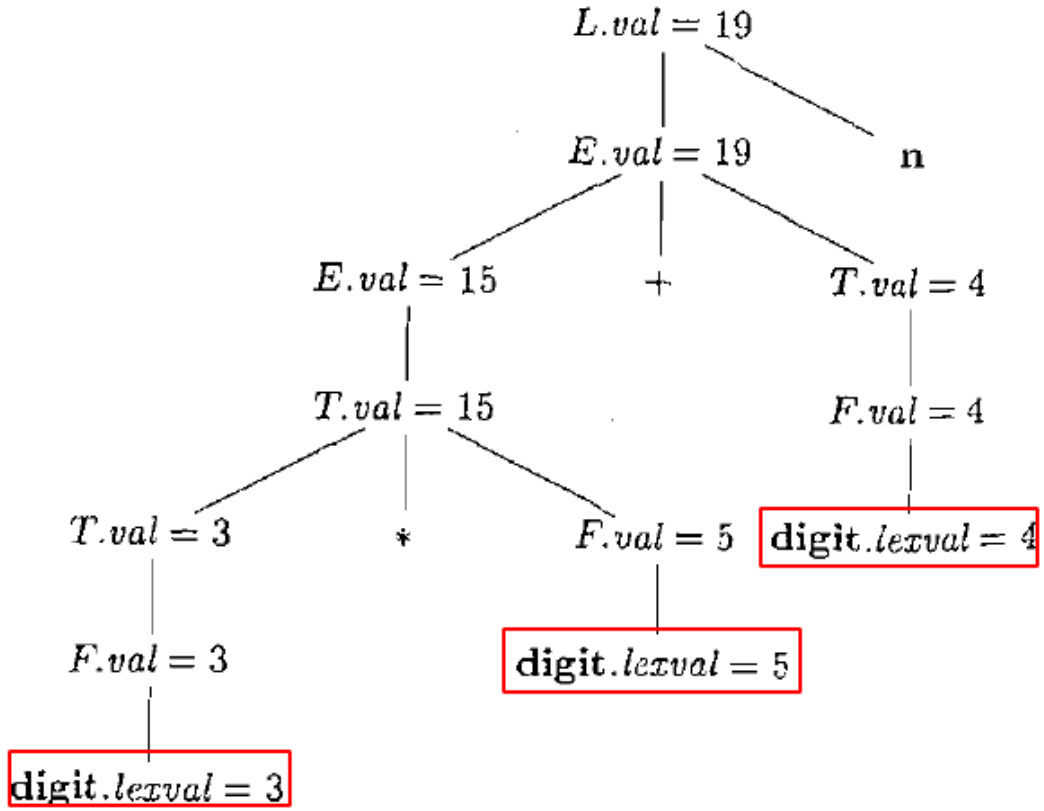
产生式	语义规则
1) $L \to E$ n	$L.val = E.val$
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to$ **digit**	$F.val =$ **digit**.lexval

Definition (S (S-Attributed Definition))

SDD, S

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

S

LL

S

LL

LL , A ,
A

$T'syninh$

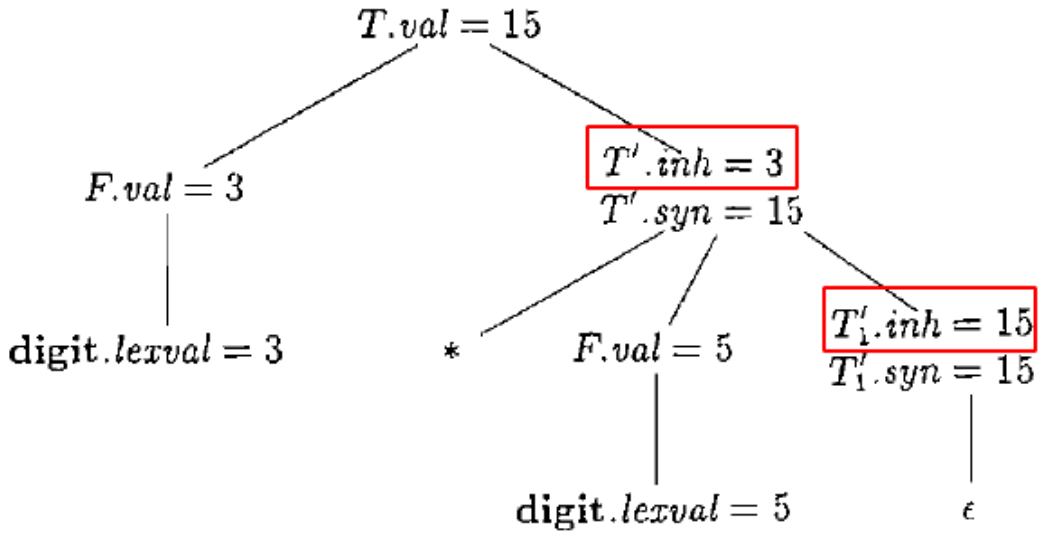
产生式	语义规则
1) $T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Definition ((Inherited Attribute))

$NNNN$

$T'.inh$

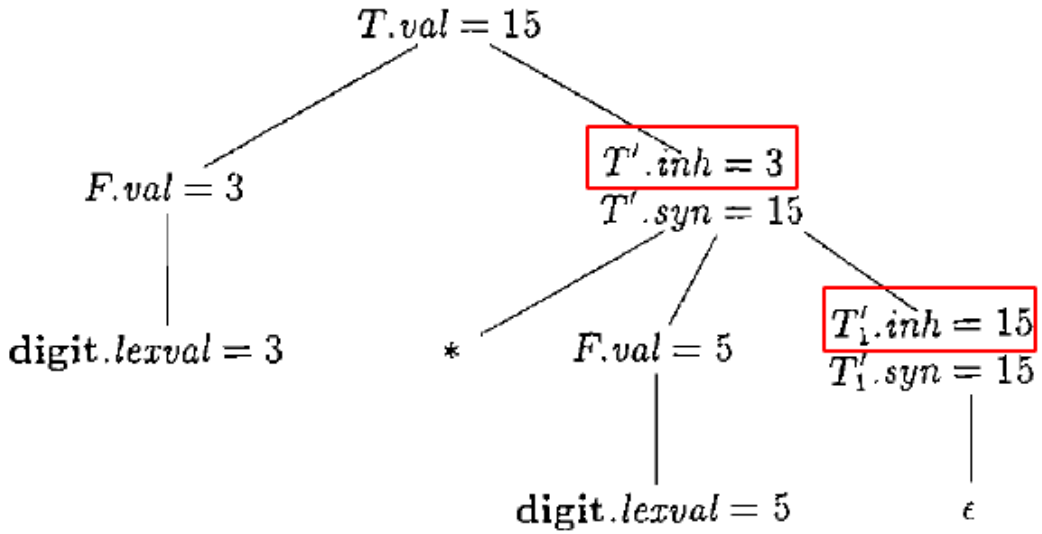
产生式		语义规则
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



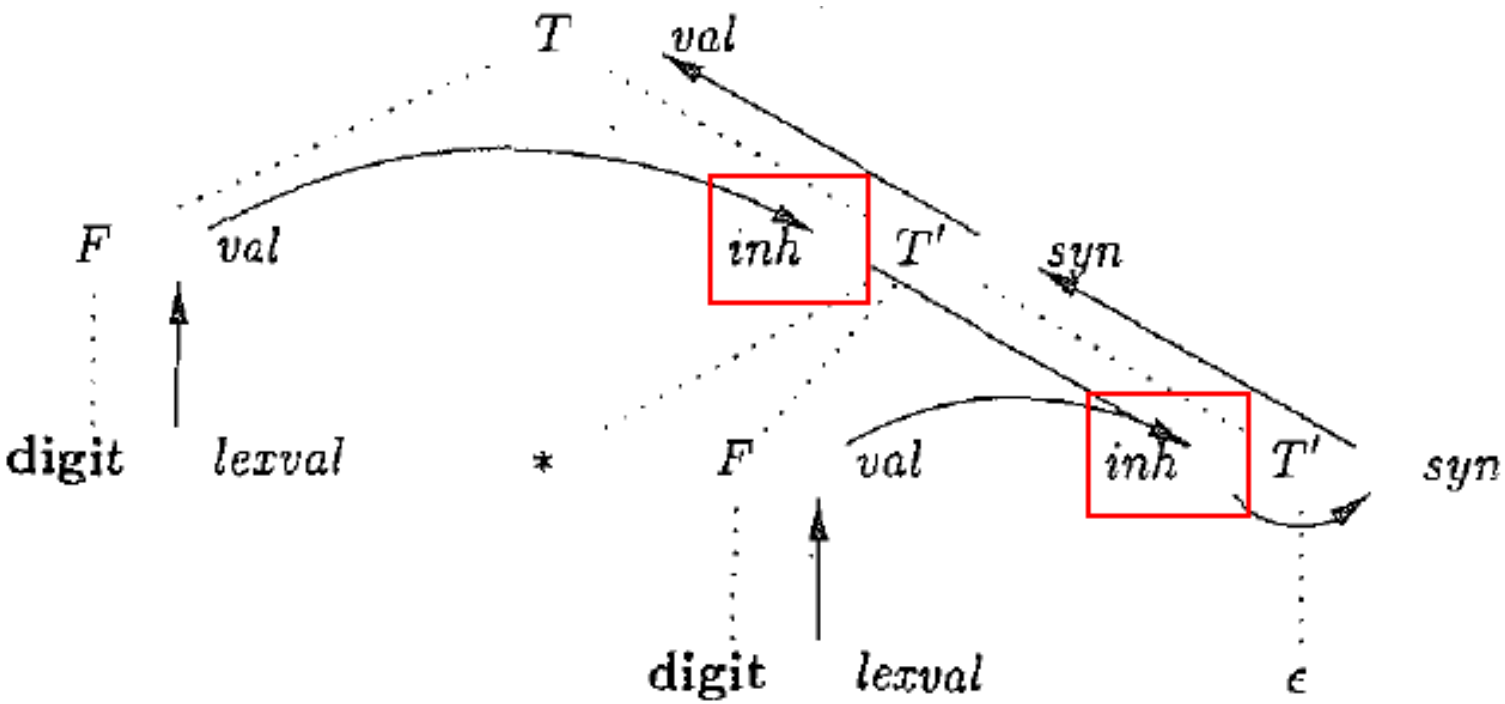
$3 * 5$

$T'.inh$

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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



$3 * 5$


T
val
F
val
inh
T'
syn
digit
lexval
*
F
val
inh
T'
syn
digit
lexval
ϵ

:

Definition (L (L-Attributed Definition))

SDD

(1) ,

(2) , :

$A \to X_1 X_2 \ldots X_n$ $X_i.a$,

(a) A;

(b) X_i $X_1 X_2 \ldots X_{i-1}$;

(c) X_i, X_i

L

L

产生式	语义规则
$A \to B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

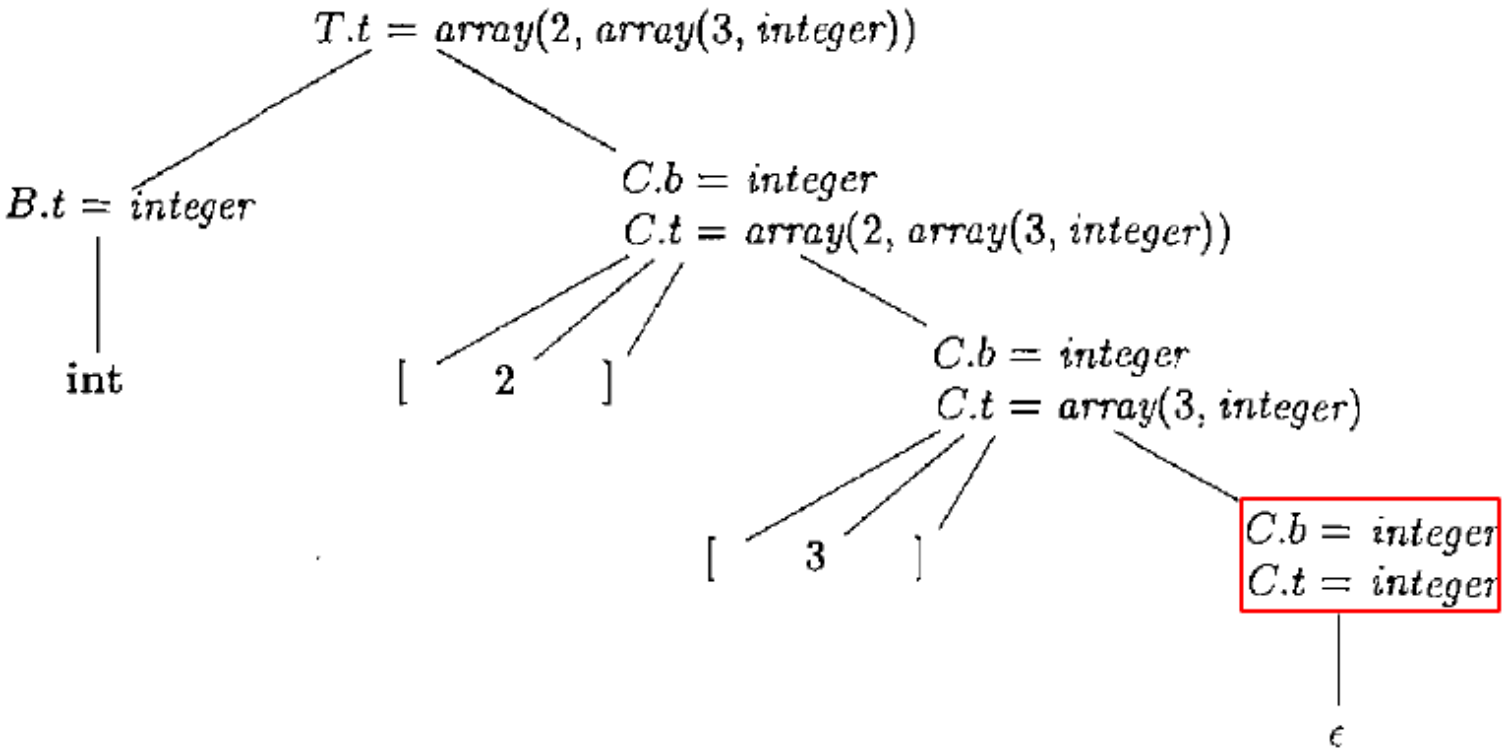
$B.i$ $C.c$

产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

int[2][3]

$array(2, array(3, integer))$

$C.b$


T.t = array(2, array(3, integer))
B.t = integer
C.b = integer
C.t = array(2, array(3, integer))
int
[2]
C.b = integer
C.t = array(3, integer)
[3]
C.b = integer
C.t = integer
ϵ

int[2][3]

$C.t$

Definition ((Postfix Notation))

(1) E , EE;

(2) E E_1 **op** E_2 , $EE_1'E_2'\mathbf{op}$, E_1' E_2' E_1 E_2 ;

(3) E (E_1) , EE_1

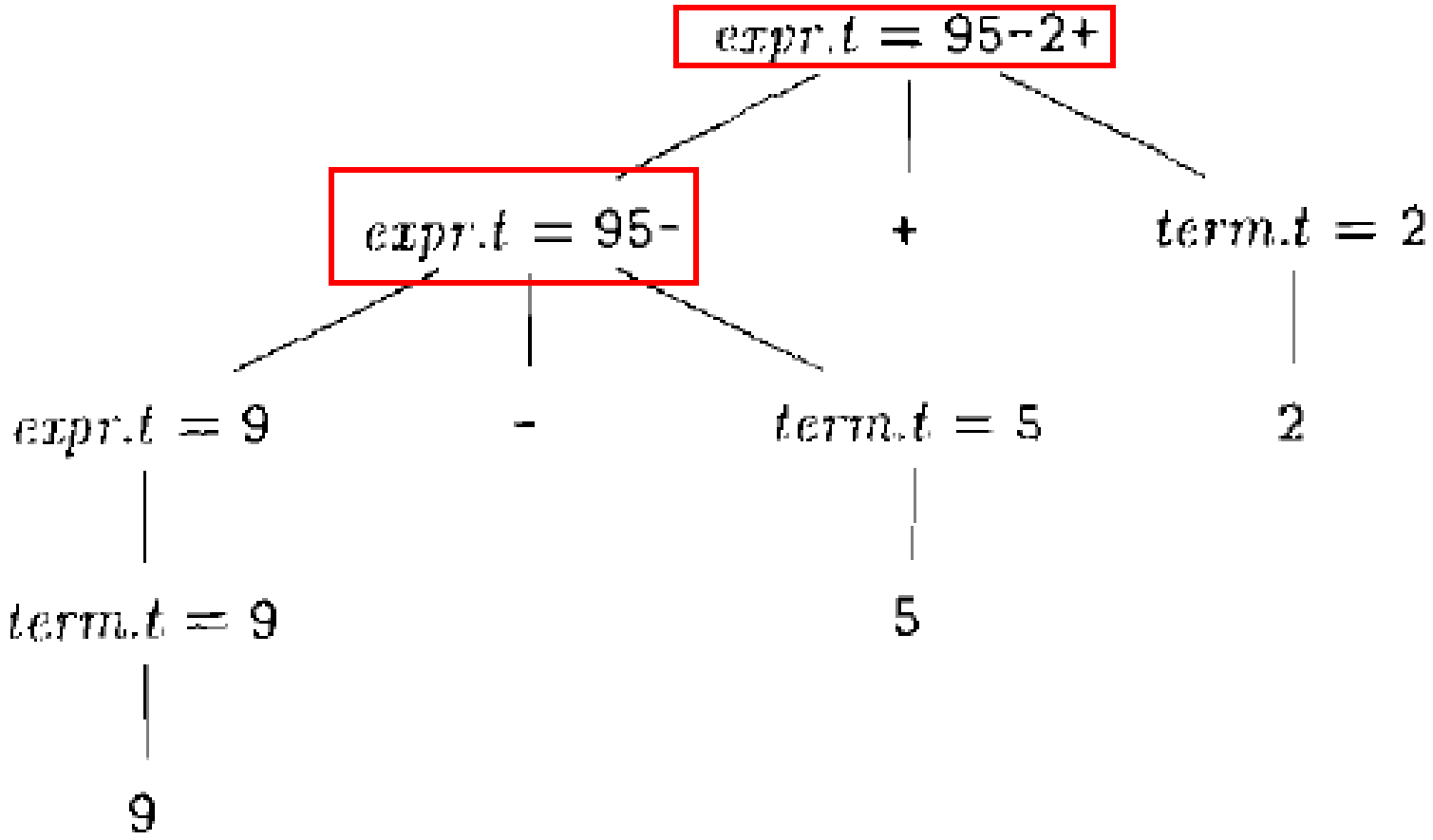
$$(9-5)+2 \implies 95-2+$$

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

S

产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

“_____”



$9 - 5 + 2$

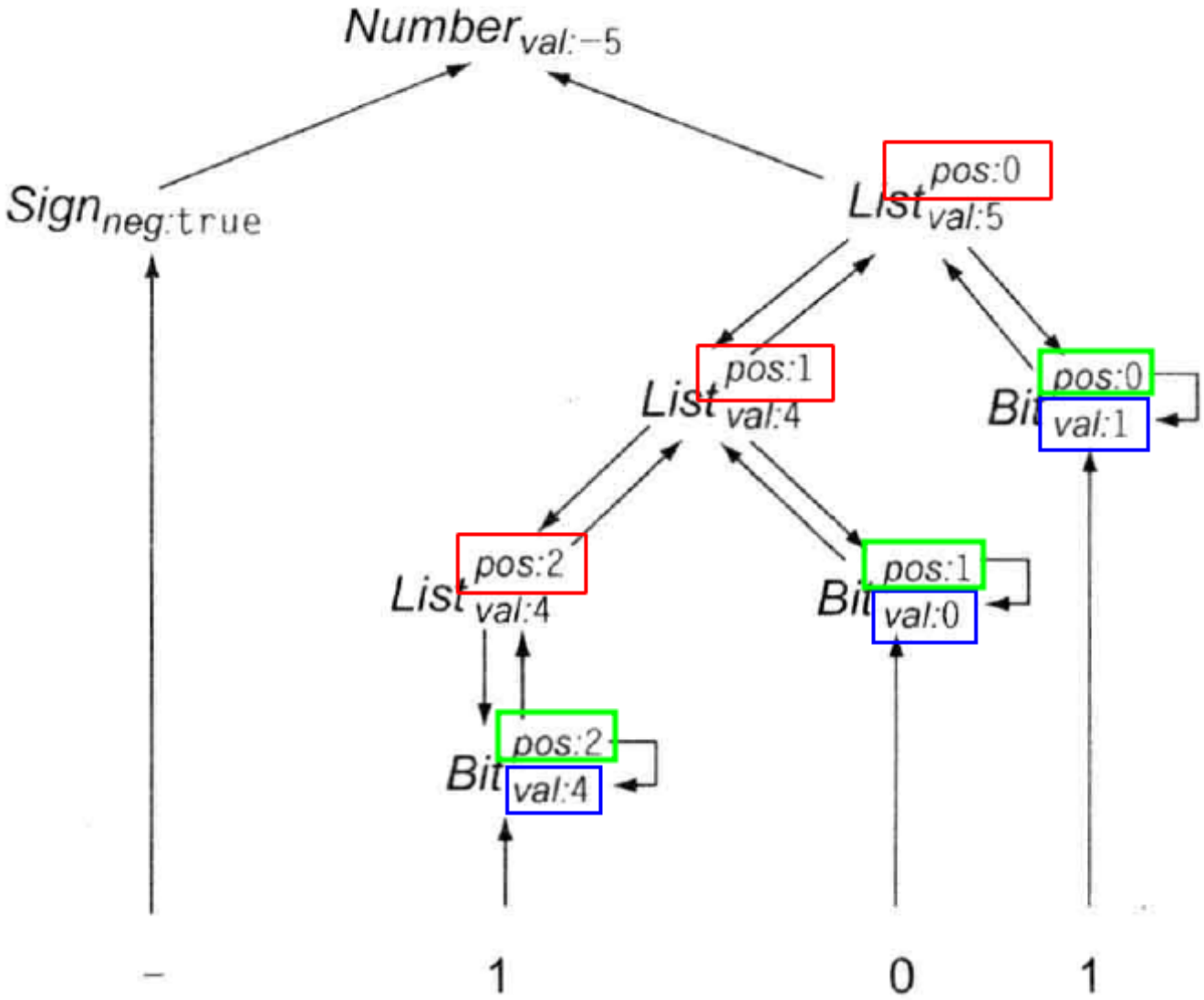
$$
P = \left\{
\begin{array}{lcl}
\textit{Number} & \rightarrow & \textit{Sign List} \\
\textit{Sign} & \rightarrow & + \\
 & | & - \\
\textit{List} & \rightarrow & \textit{List Bit} \\
 & | & \textit{Bit} \\
\textit{Bit} & \rightarrow & 0 \\
 & | & 1
\end{array}
\right\}
$$

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

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

L

	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 ((Syntax-Directed Translation Scheme; SDT))

SDT

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

Definition ((Syntax-Directed Translation Scheme; SDT))

SDT

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 : SDD SDT

S

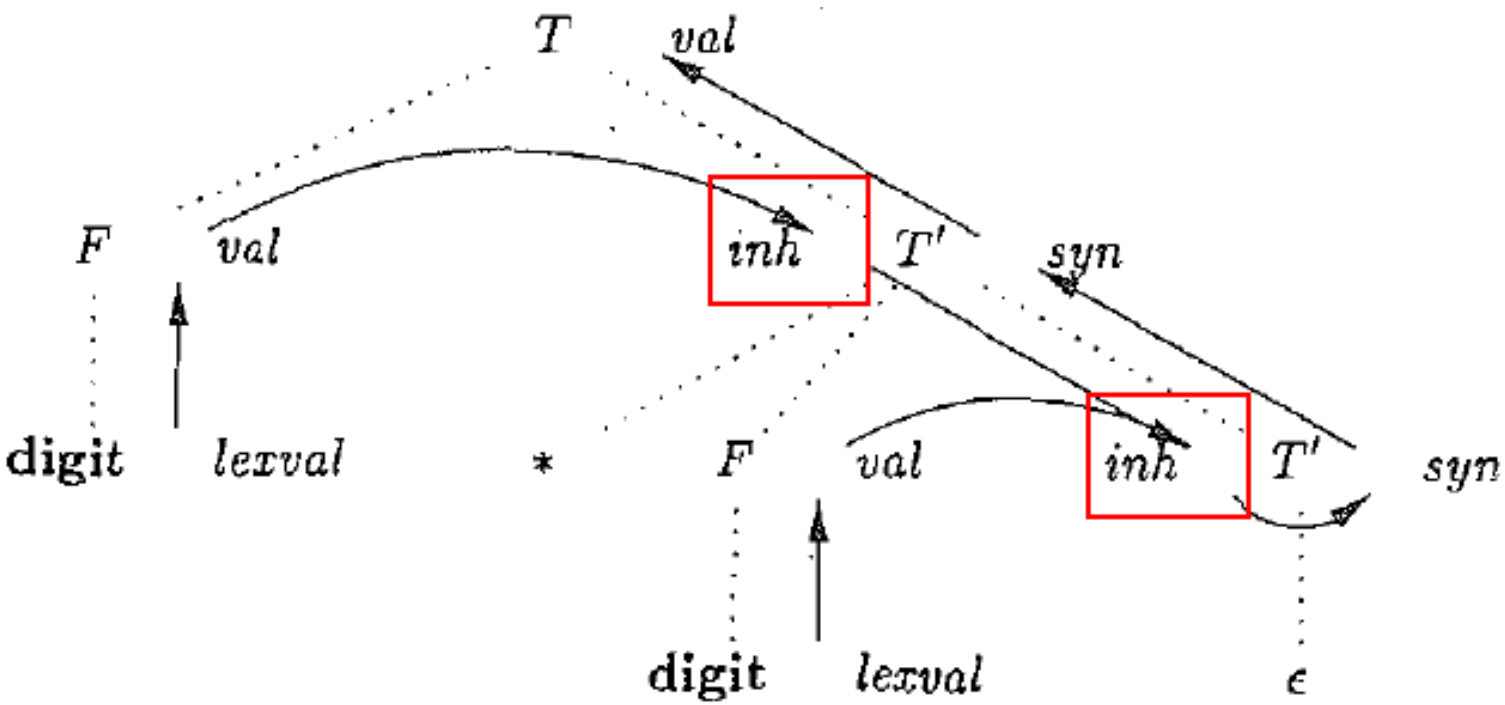
产生式		语义规则
1)	$L \to E$ n	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

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

:

L LL



$3 * 5$

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

: X_i

X_i, ,

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

- X_i , X_i
- X_i X_i

- X_i , X_i
- X_i X_i

```
file
locals [int i = 0]
    : hdr ( rows += row[$hdr.text.split(",")] { $i++; })+ {
            System.out.println("Totally " + $i + " rows");
            for (RowContext r : $rows) {
                System.out.println("Row token interval : " + r.getSourceInterval());
            }
        }
    ;
```

```
row[String[] columns] returns [Map<String, String> values]
locals [int col = 0]
@init {
    $values = new LinkedHashMap<>();
}
@after {
    if ($values.size() > 0) {
        System.out.println("values = " + $values);
    }
}
    : field {
        if ($columns != null) {
            $values.put($columns[$col++].trim(), $field.text.trim());
        }
    } (',' field
        {
            if ($columns != null) {
                $values.put($columns[$col++].trim(), $field.text.trim());
            }
        }
    )* '\r'? '\n'
    ;
```

CSVAGParser.java

$$() \ S$$

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

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

$$\boxed{XY^*}$$

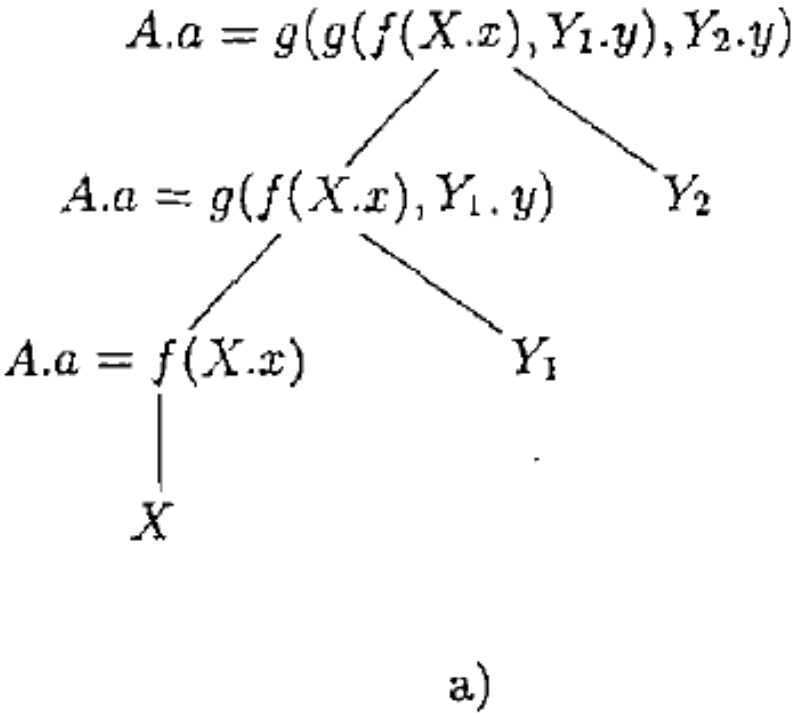
$$() \ L$$

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

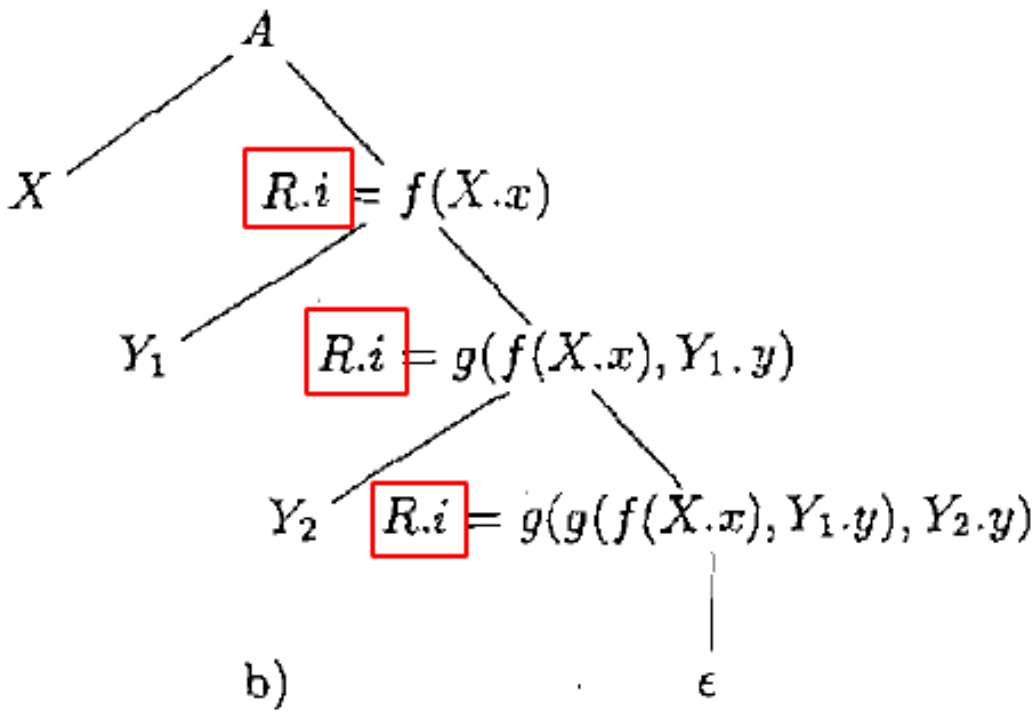
$$R \to YR_1 \qquad R_1.i = g(R.i, Y.y); \quad R.s = R_1.s$$

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

$R.i$


A.a = g(g(f(X.x), Y1.y), Y2.y)
A.a = g(f(X.x), Y1.y)
Y2
A.a = f(X.x)
Y1
X

a)


A
X
R.i = f(X.x)
Y1
R.i = g(f(X.x), Y1.y)
Y2
R.i = g(g(f(X.x), Y1.y), Y2.y)
ϵ

b)

,

() L

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

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

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

: ,

() L

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

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

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

: ,

L SDT

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

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

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

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

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

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

1: **procedure** $A()$ ▷ A ,
2: **if** `token` $=$ `?` **then** ▷ $A \to XR$
3: $X.x \leftarrow \text{MATCH}(X)$ ▷ X ,
4: $R.i \leftarrow f(X.x)$ ▷ $R.i$
5: $R.s \leftarrow R(R.i)$ ▷ $R(R.i)$
6: **return** $R.s$ ▷ $A.a \leftarrow R.s$

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

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

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

```
1: procedure R(R.i)                                   ▷ R  R.i
2:     if token = ? then                              ▷ R → Y R
3:         Y.y ← MATCH(Y)                             ▷ Y ,
4:         R.i ← g(R.i, Y.y)                          ▷ R.i
5:         R.s ← R(R.i)                               ▷ R(R.i)
6:         return R.s                                 ▷
7:     else if token = ? then                         ▷ R → ϵ
8:         return R.i                                 ▷ R.s ← R.i
```

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.

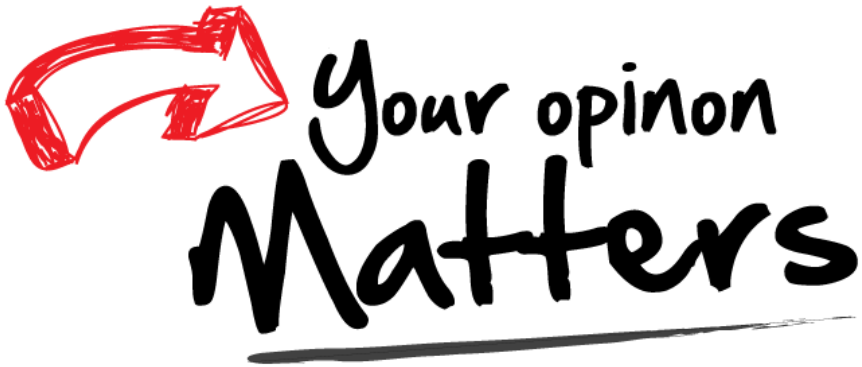
What is the difference between ANTLR 3 and 4?

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