Semantic Analyzer
Syntax Directed Translation

Course : 2CS701 Compiler Construction

Prof Monika Shah

Nirma University

Ref : Ch.5 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

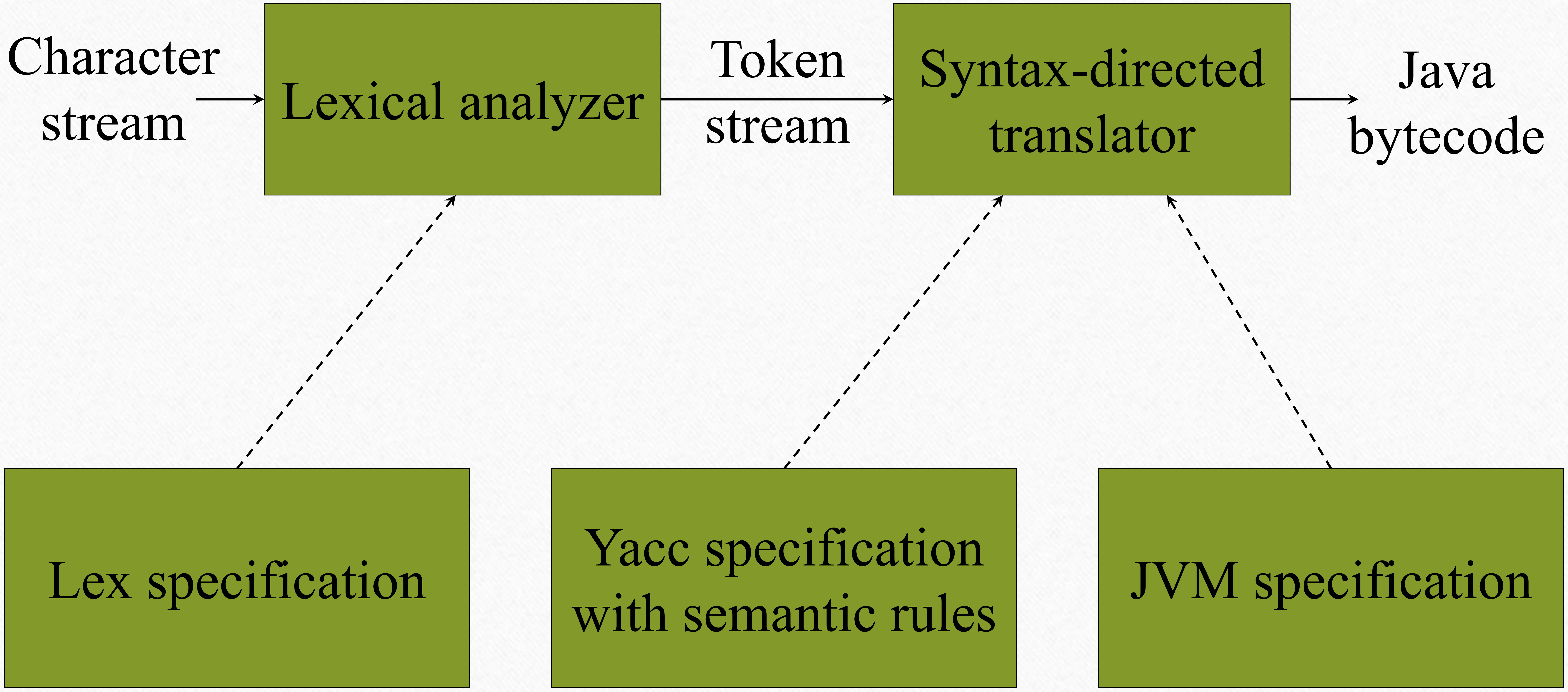
Glimpse

- Introduction to Syntax Directed Translation and applications
- Background : Important terminologies
- Type of Syntax Directed Definition
- Evaluation methods of syntax directed definition
- Translation Scheme
- Transformation of Syntax Directed definition to Translation Schemes
- Evaluation of Translation scheme
- Eliminating Left recursion from Translation scheme
- Implementing syntax directed definition at Top-down parsing, Bottom-up parsing
- Implementing SDD / Translation scheme in YACC

Conceptual View of Syntax Directed Translation

Character stream $\longrightarrow$ Token stream $\longrightarrow$ Parse tree $\longrightarrow$ Dependency Graph $\longrightarrow$ Evaluation Order of semantic rules

The Structure of our Compiler Revisited



What is Syntax Directed Translation ?

- Mapping between each Syntax production rule with Translation rule
- Bind Semantic rules to each Syntax production rule
- Applications :
  - Generate Intermediate Code
  - Generate Target Code
  - Semantic Check
  - Interpreter Design : Execute Syntax directed execution
  - Parse Tree Generation

Significance of Semantic Analyzer

- Analyze following pair of input string
  - Cow eats grass
  - Grass eats cow

	Lexical Analysis	Syntax Analysis	Semantic Analysis
Cow eats grass	✓	✓	✓
Grass eats cow	✓	✓	✗

- Different Target
  - 2 + 3 → Calculator → 5
  - 2 + 3 → Type Checker → “integer”
  - 2 + 3 → Infix-> Prefix → + 2 3

Realization from previous examples

- Lexically and Syntactically correct program may still contain other types of errors
- Lexical and Syntax analyzers are not powerful enough to ensure correct usage of variables, functions etc. For example,

```
int a;
a = 1.2345;
```

Example of Semantic Analysis to ensure program satisfy certain rules

- Variable must be defined before being used
- A variable should not be defined multiple times
- The same identifier cannot be use to denote different type of syntactic objects.
- In switch statement : Case label must be integer or character type
- IF statement, While statement should have condition expression of Boolean type
- In assignment statement, the LHS must be a variable and RHS must be expression of same data type

Background : Important Terminologies

- Syntax Directed Definition
- Attribute Grammar
- Annotated parse tree
- Attributes

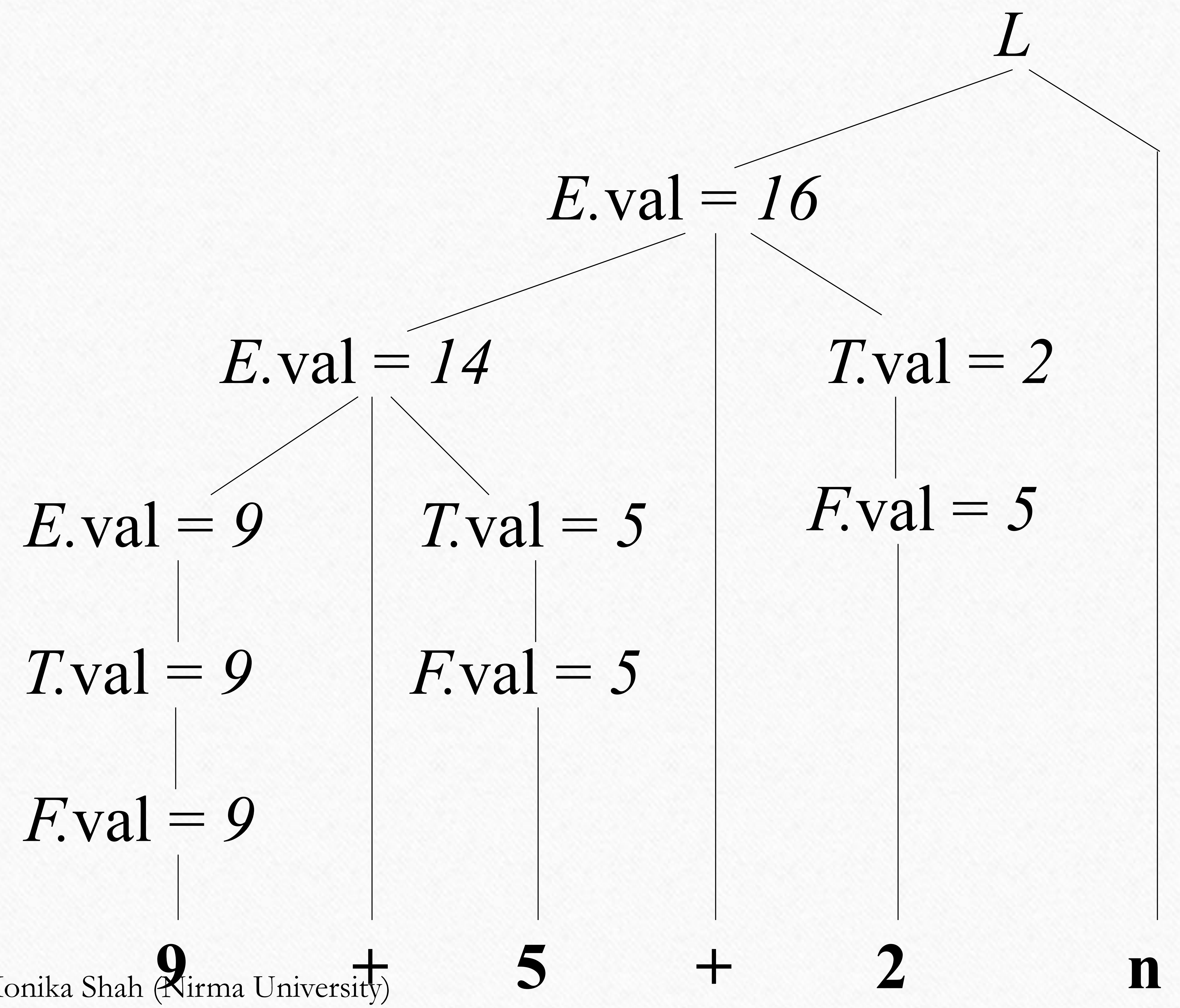
Syntax-Directed Definitions

- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes* holding values set by the semantic rules
- A *depth-first traversal* algorithm traverses the parse tree thereby executing semantic rules to assign attribute values
- After the traversal is complete the attributes contain the translated form of the input

Example Attribute Grammar

Production	Semantic Rule
$L \rightarrow E$ **n**	$print(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow$ **digit**	$F.\text{val} :=$ **digit**.lexval

Annotated Parse Tree

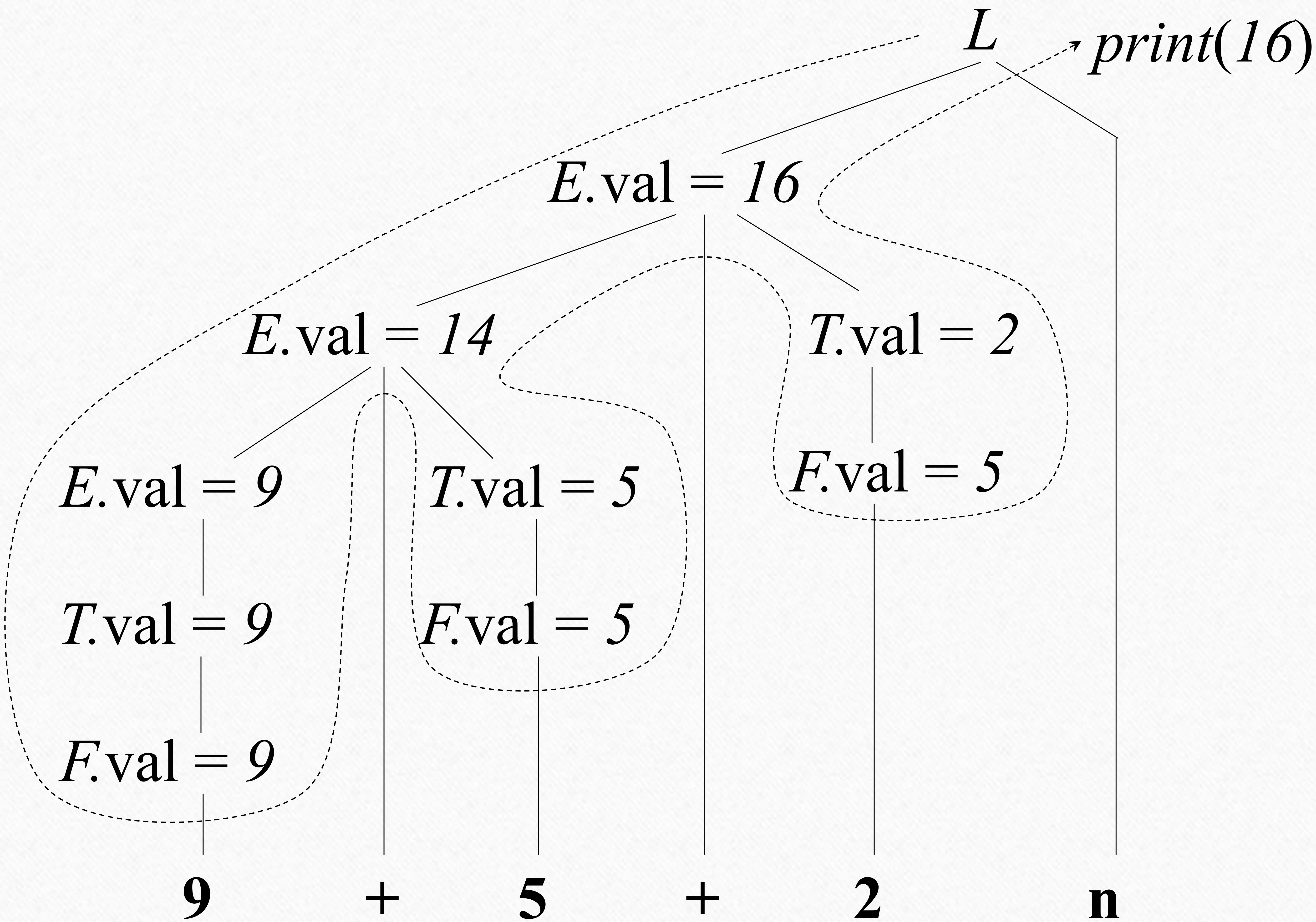


Attributes of Non-terminal nodes are **evaluated by applying attribute grammar** while traversing parse tree using **Depth First Search**

Annotating a Parse Tree With Depth-First Traversals

```
procedure visit(n : node);
begin
    for each child m of n, from left to right do
        visit(m);
    evaluate semantic rules at node n
end
```

Depth-First Traversals (Example)



Attributes

- Attribute values may represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Synthesized Versus Inherited Attributes

- Given a production
  $A \rightarrow \alpha$
  then each semantic rule is of the form
  $b := f(c_1, c_2, \ldots, c_k)$
  where f is a function and c_i are attributes of A and α, and either
  - b is a *synthesized* attribute of A
  - b is an *inherited* attribute of one of the grammar symbols in α

Synthesized Versus Inherited Attributes (cont'd)

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
…	…
$L \rightarrow$ **id**	… := L.in

inherited

synthesized

Types of Syntax Directed Definition

1. S-Attributed Definitions
2. L-Attributed Definitions
3. Other

S-Attributed Definitions

- A syntax-directed definition that **uses synthesized attributes exclusively** is called an *S-attributed definition* (or *S-attributed grammar*)
- A parse tree of an S-attributed definition is annotated with a single bottom-up traversal
- **Yacc/Bison only support S-attributed definitions**

Example Attribute Grammar in Yacc

```
%token DIGIT
%%
L : E '\n'       { printf("%d\n", $1); }
  ;
E : E '+' T      { $$ = $1 + $3; }
  | T         { $$ = $1; }
  ;
T : T '*' F      { $$ = $1 * $3; }
  | F         { $$ = $1; }
  ;
F : '(' E ')'    { $$ = $2; }
  | DIGIT        { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \rightarrow F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *****	*3* _	**5+4n$**	shift	
$ *T* *** 5**	*3* _ *5*	**+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *T* ***** *F*	*3* _ *5*	**+4n$**	reduce $T \rightarrow T * F$	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \rightarrow T$	$$ = $1
$ *E* **+**	*15* _	**+4n$**	shift	
$ *E* **+ 4**	*15* _ *4*	**4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *E* **+** *F*	*15* _ *4*	**n$**	reduce $T \rightarrow F$	$$ = $1
$ *E* **+** *T*	*15* _ *4*	**n$**	reduce $E \rightarrow E + T$	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19* _	**$**	reduce $L \rightarrow E$ **n**	*print* $1
$ *L*	*19*	**$**	accept	

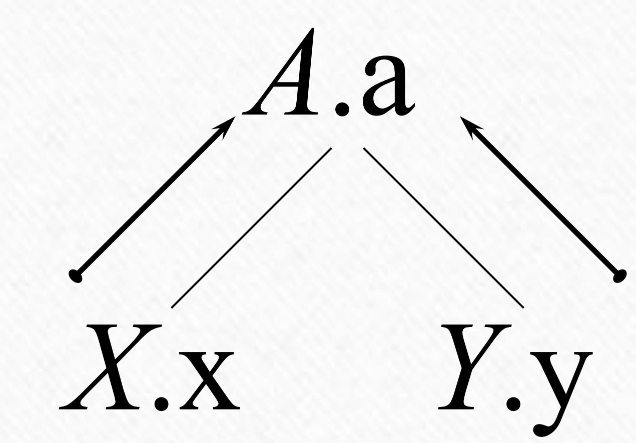
Example Attribute Grammar with Synthesized+Inherited Attributes

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

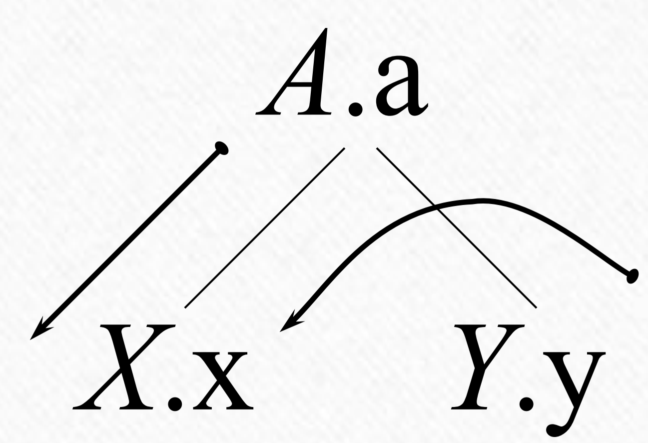
Synthesized: T.type, **id**.entry
Inherited: L.in

Acyclic Dependency Graphs for Attributed Parse Trees

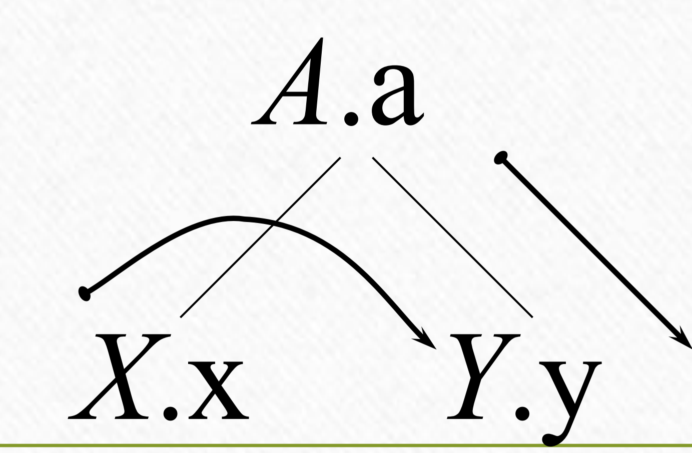
$A \rightarrow X\ Y$



$A.\mathrm{a} := f(X.\mathrm{x}, Y.\mathrm{y})$



$X.\mathrm{x} := f(A.\mathrm{a}, Y.\mathrm{y})$



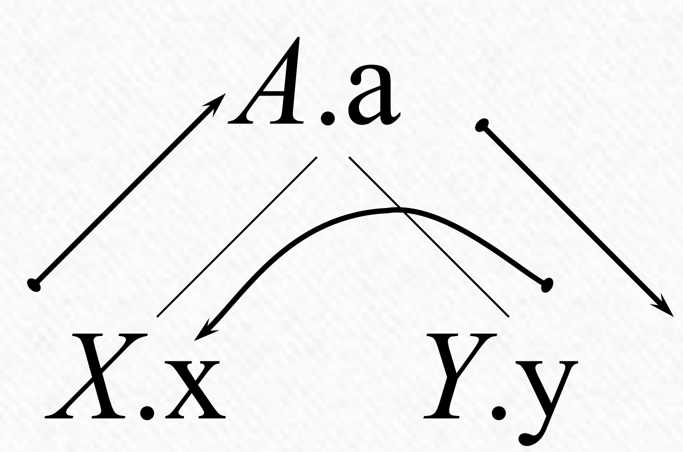
$Y.\mathrm{y} := f(A.\mathrm{a}, X.\mathrm{x})$

Direction of

value dependence

Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- **<u>Dependency graphs cannot be cyclic</u>**



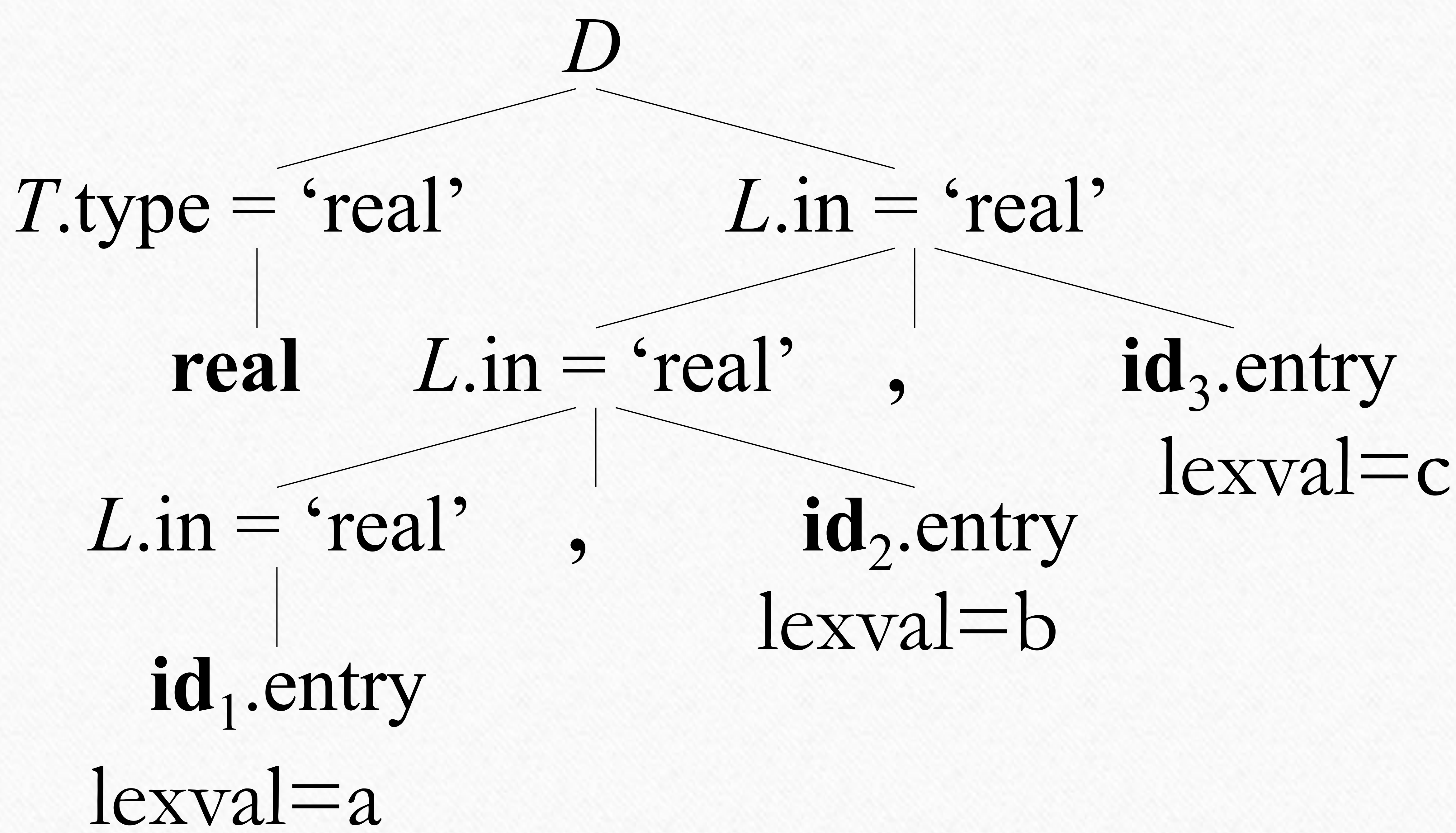
$$A.\mathrm{a} := f(X.\mathrm{x})$$
$$X.\mathrm{x} := f(Y.\mathrm{y})$$
$$Y.\mathrm{y} := f(A.\mathrm{a})$$

Error: cyclic dependence

Example Annotated Parse Tree

Input character stream : real a, b, c

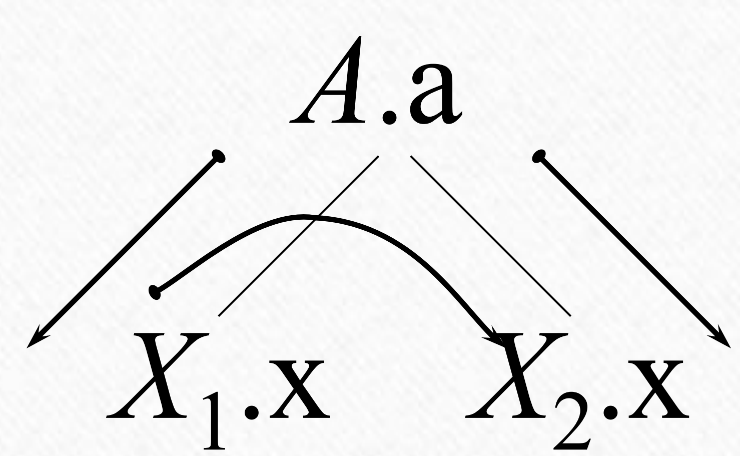
Input token stream : DT ID, ID, ID



L-Attributed Definitions

- The example parse tree on slide 18 is traversed "in order", because the direction of the edges of inherited attributes in the dependency graph point top-down and from left to right
- More precisely, a **syntax-directed definition is *L-attributed*** if each <u>inherited</u> attribute of X_j on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

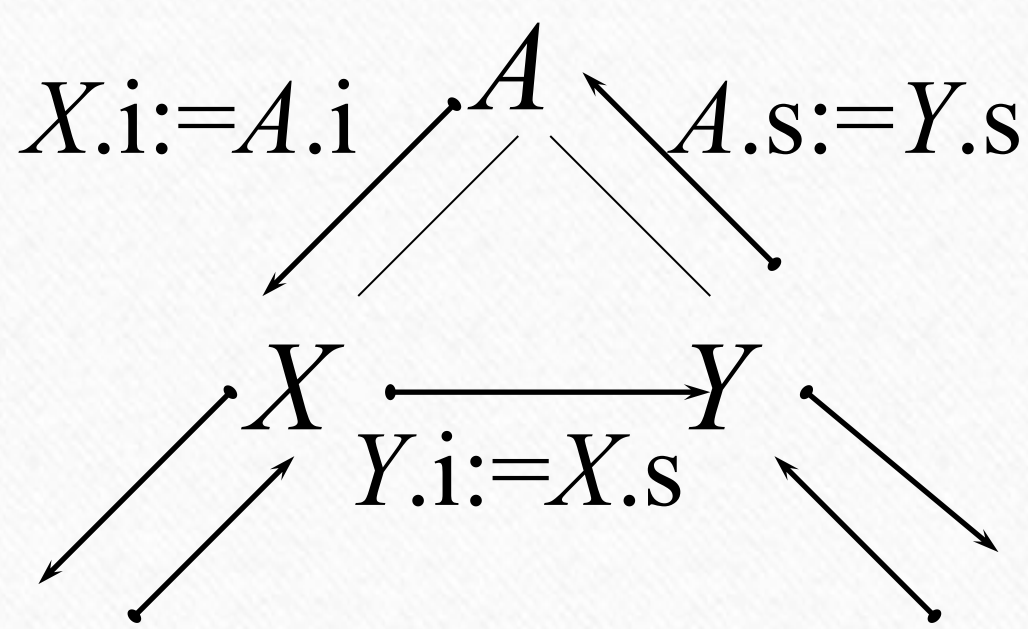
Shown: dependences of inherited attributes



L-Attributed Definitions (cont'd)

- L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

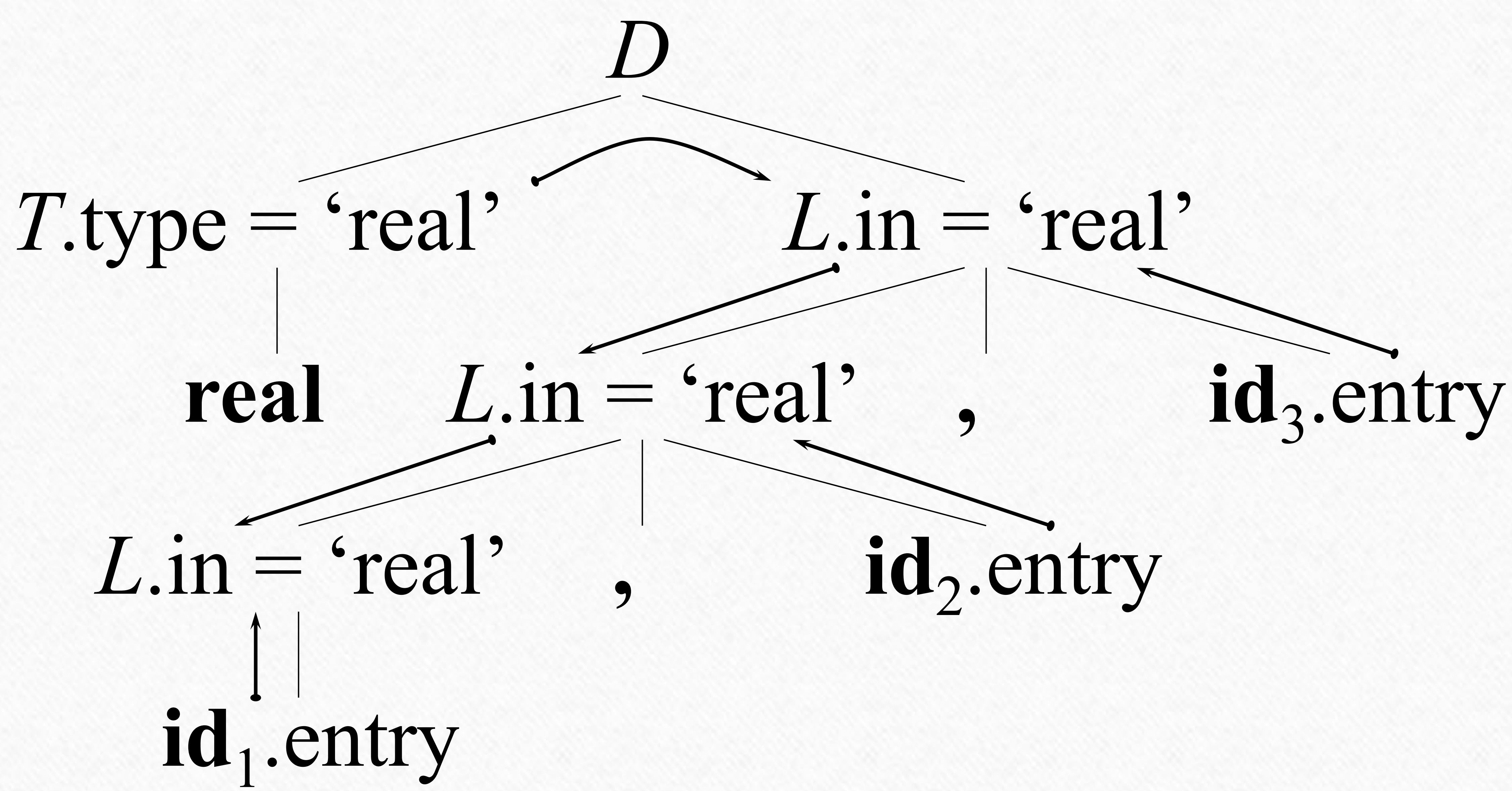
$A \rightarrow X\ Y$



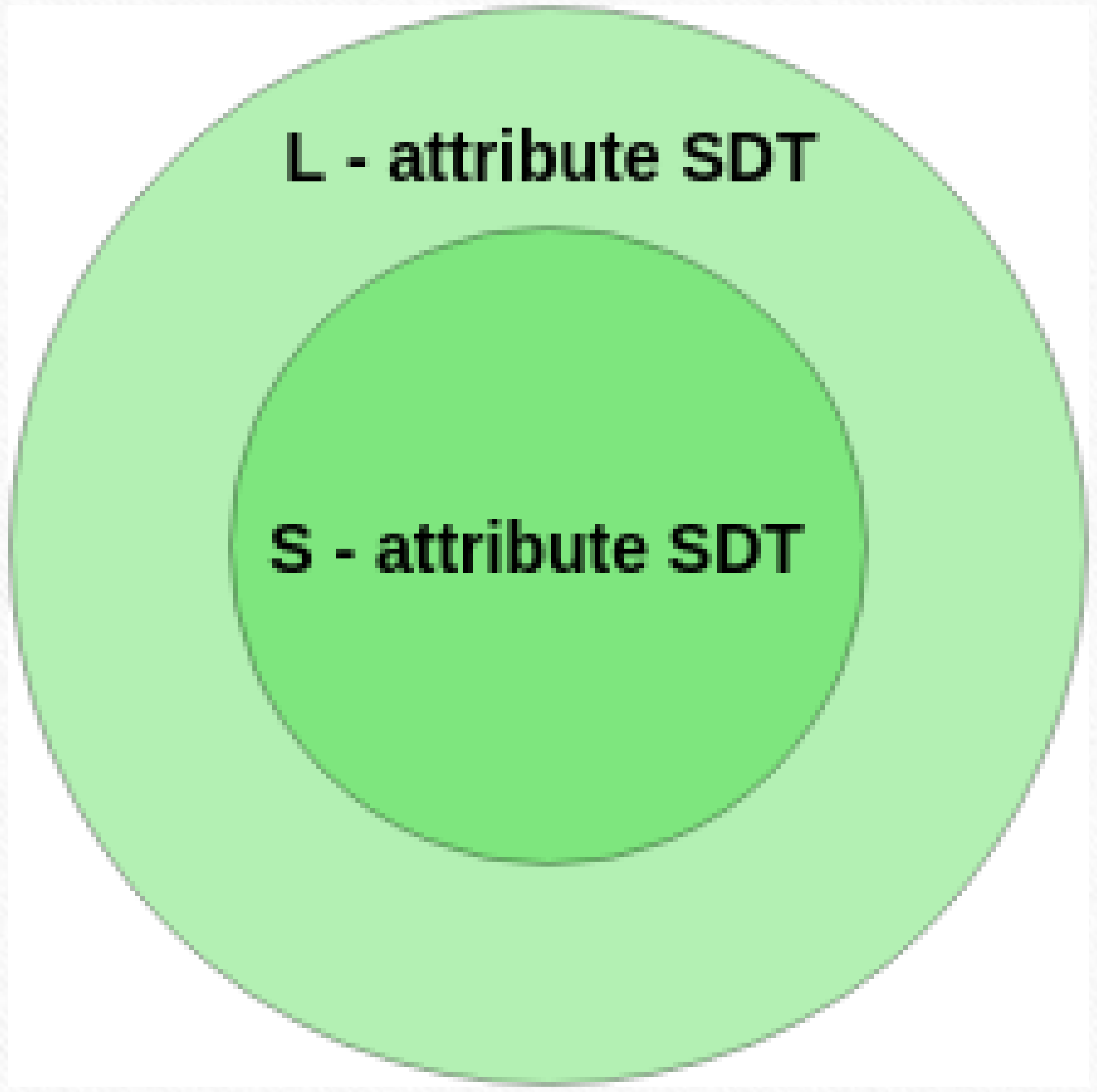
$X.\mathrm{i} := A.\mathrm{i}$
$Y.\mathrm{i} := X.\mathrm{s}$
$A.\mathrm{s} := Y.\mathrm{s}$

- **<u>Note</u>**: every S-attributed syntax-directed definition is also L-attributed

Example Annotated Parse Tree with Dependency Graph



S-attributed SDD and L-attributed SDD



Every S-attributed SDD is L-attributed , but not Vise Versa

S -> MN {S.val= M.val + N.val}

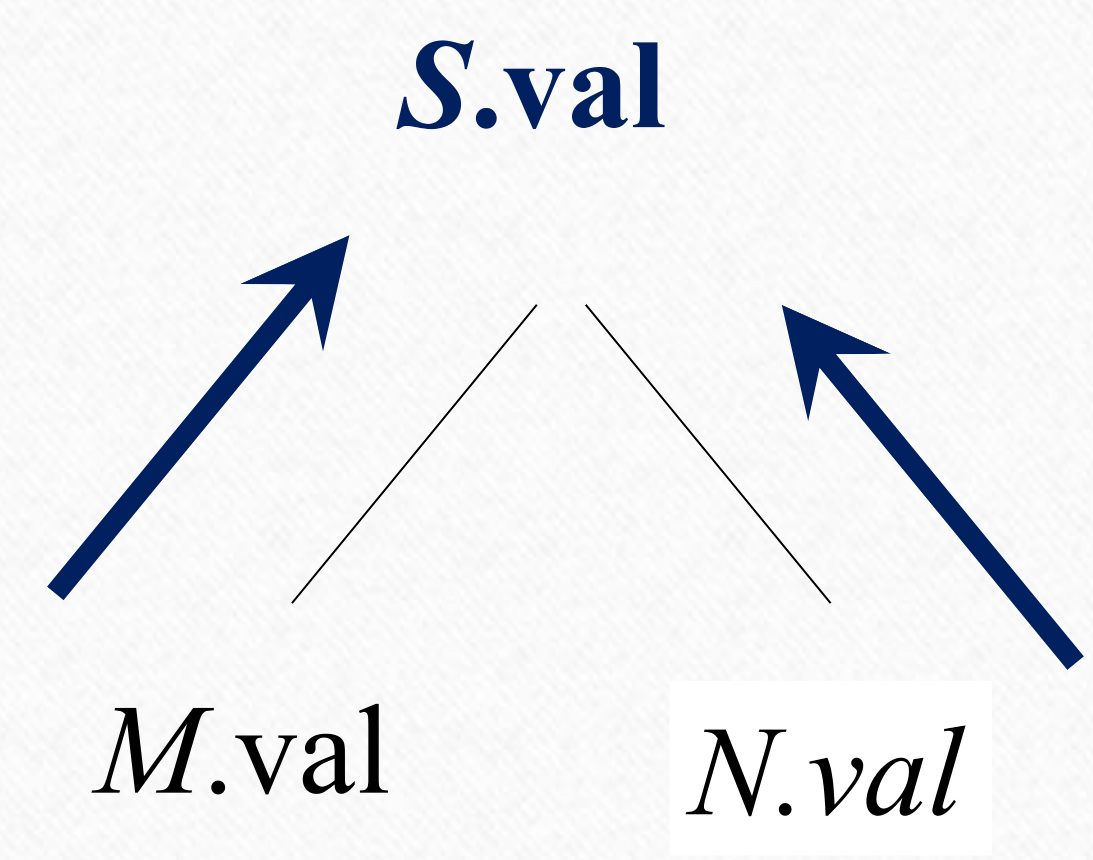
Example of SDD

S → MN { S.val = M.val + N.val}

What type of attribute S.val ? **synthesized**

Is this SDD S-attributed SDD ? ✓

Is this SDD L-attributed SDD ? ✓



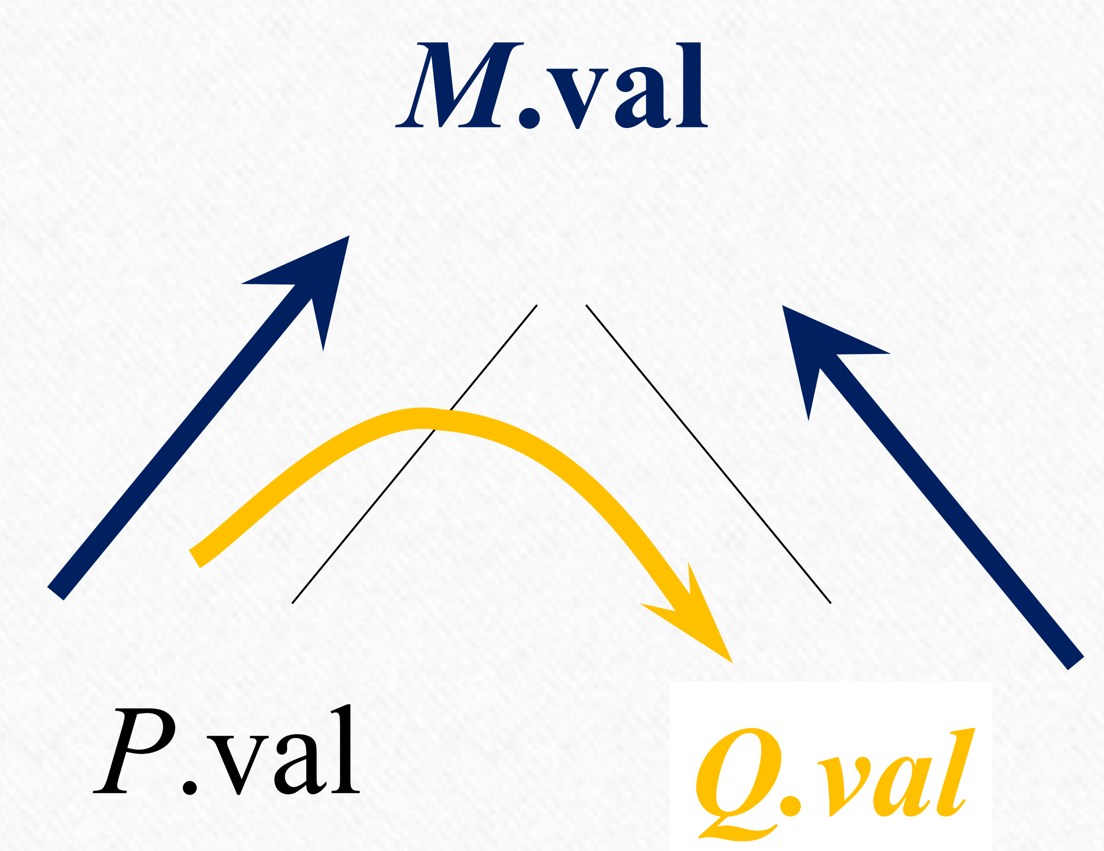
Example of SDD

M → PQ { M.val = P.val * Q.val , Q.val = P.val}

What type of attribute M.val ? **Inherited**

Is this SDD S-attributed SDD ?

Is this SDD L-attributed SDD ?



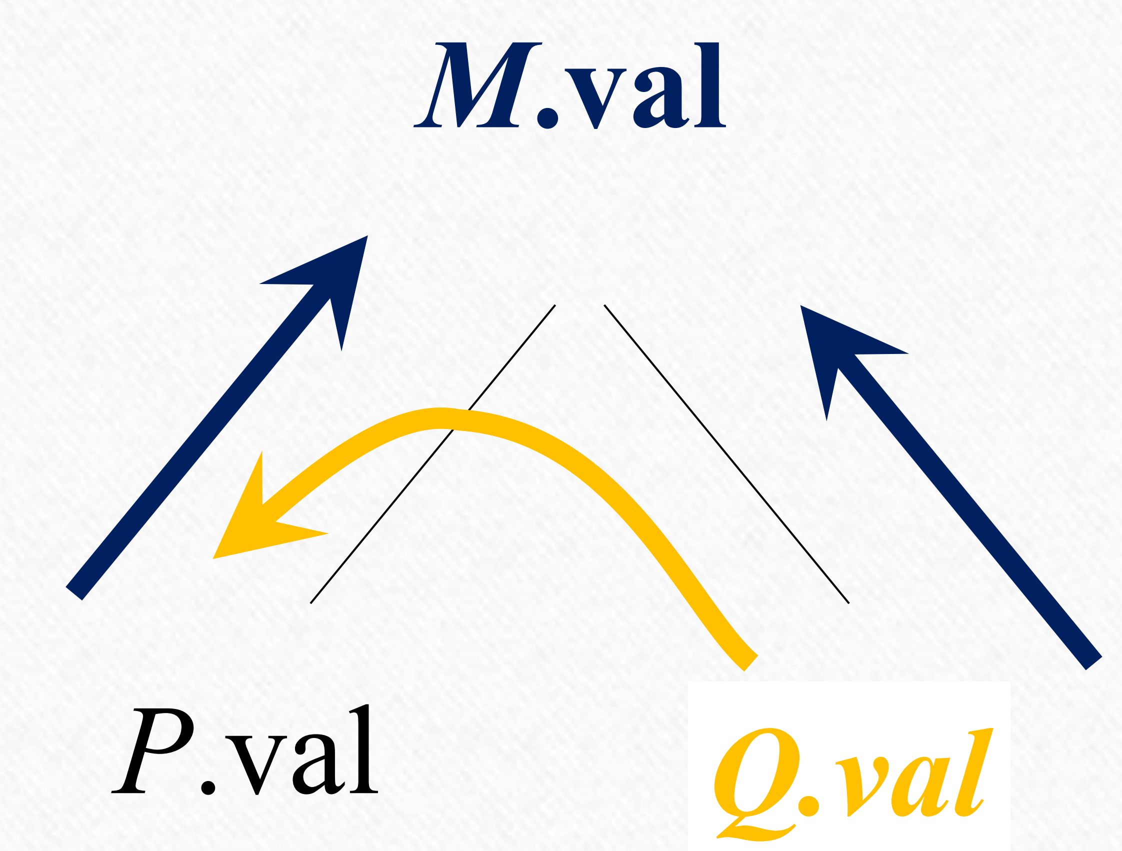
Example of SDD

M → PQ { M.val = P.val * Q.val , P.val = Q.val}

What type of attribute Q.val? **Inherited**

Is this SDD S-attributed SDD ?

Is this SDD L-attributed SDD ?



Example of SDD

A → XYZ { Y.S = A.S , Y.S = X.S, Y.S = Z.S}

What type of attribute Y.S ? **Inherited**

Is this SDD S-attributed SDD ? ✗

Is this SDD L-attributed SDD ? ✗

Exercise

- Write SDD to translate **binary number to decimal** number using following Grammar:

Syntax Rules	**Semantic rules**
BN → B	BN.Val = B.Val
B → B D	B.Val = B1.Val * 2 + D.Val
B → D	B.Val = D.Val
D → 0	D.Val = 0
D → 1	D.Val =1

Exercise

- Write SDD to translate **fractional binary number to decimal** number using following Grammar:

Syntax Rules	Semantic rules
BN → B '.' B	BN.Val = B1.Val + B2.Val / power (2, B2.count)
BN → B	BN.Val = B.Val
B → B D	B.Val = B1.Val * 2 + D.Val ; B.Count = B1.count + D.count
B → D	B.Val = D.Val , B.count=D.count
D → 0	D.Val = 0 , D.count=1
D → 1	D.Val =1, D.count=1

Exercise

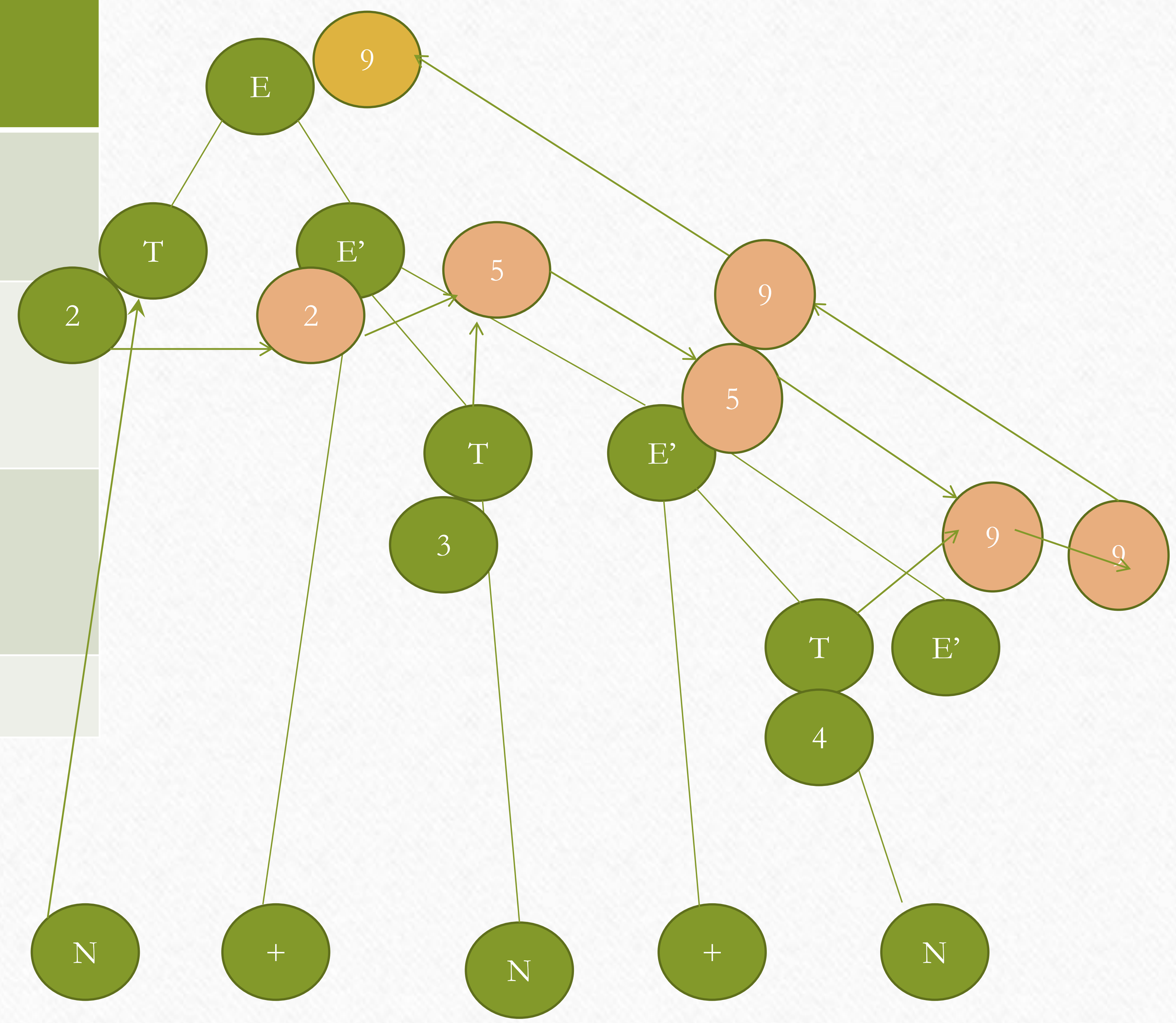
Write SDD to evaluate an arithmetic expression using following Grammar:
i.e. a= b= 2+3 should assign 2+3 to b , b to a

Syntax Rules	Semantic rules
A→ ID '=' A	Update(ID.entry, Value=A1.Value), A.Value=A1.Value
A → E	A.Value = E.Value
E → E '+' T	E.Value = E1. Value + T.Value
E → T	E.Value = T. Value
T → NUM	T.Value = atoi (NUM.lexval)
T → ID	T.Value = Lookup(ID.entry, Value)
T → '(' E ')'	T,.Value = E.Value

Exercise

Write SDD to evaluate an arithmetic expression using following Grammar: N + N + N

Syntax Rules	Semantic rule
E → TE'	E'.ival=T.sval E.sval=E'.sval
E' → '+' TE'	E1'.ival=E.ival+T.val E'.sval=E1'.sval
E' → e	E'.sval=E'.ival
T → NUM	



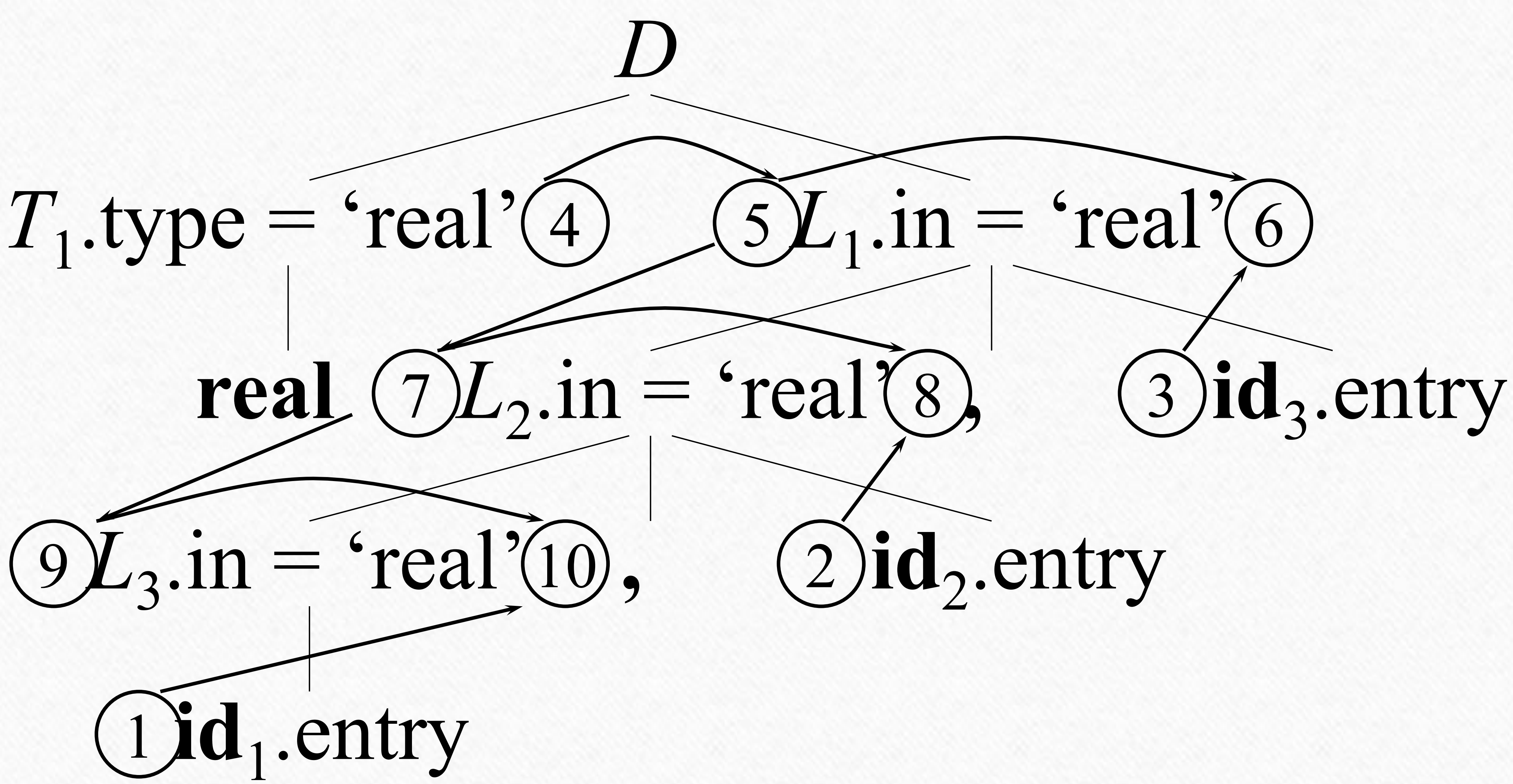
Practice questions

- "Every S-attributed SDD is L-attributed SDD" Write your opinion about this statement with proper justification
- Write SDD for counting number of variables in a program
- Write SDD to assign data types to variable
- Writes SDD to verify variables are defined before its use

Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \to m_j$ is an edge, then m_i appears before m_j
- Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules

Example Parse Tree with Topologically Sorted Actions



Topological sort:

1. Get **id**$_1$.entry
2. Get **id**$_2$.entry
3. Get **id**$_3$.entry
4. T_1.type='real'
5. L_1.in=T_1.type
6. *addtype*(**id**$_3$.entry, L_1.in)
7. L_2.in=L_1.in
8. *addtype*(**id**$_2$.entry, L_2.in)
9. L_3.in=L_2.in
10. *addtype*(**id**$_1$.entry, L_3.in)

Evaluation Methods

- *Parse-tree methods* determine an evaluation order from a topological sort of the dependence graph constructed from the parse tree for each input
- *Rule-base methods* the evaluation order is pre-determined from the semantic rules
- *Oblivious methods* the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)

Translation Scheme

- Shows evaluation order of semantic rules
- Semantic rules are embedded within production rules on RHS

Transformation of Syntax Directed Definition to Translation Scheme

- Transformation of L-attributed definition
  - Place semantic rule of Synthesis attribute at end of Syntax rule
  - Place semantic rule of Inherited attribute just before the attribute
- Transformation of S-attributed definition
  - No change

Using Translation Schemes for L-Attributed Definitions

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Translation Scheme

$D \rightarrow T$ { L.in := T.type } L

$T \rightarrow$ **int** { T.type := 'integer' }

$T \rightarrow$ **real** { T.type := 'real' }

$L \rightarrow$ { L_1.in := L.in } L_1 **, id** { *addtype*(**id**.entry, L.in) }

$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Implementing L-Attributed Definitions in Top-Down Parsers

- Inherited attributes are arguments
- Synthesis are return

$$D \rightarrow T \ \{ \ L.\mathrm{in} := T.\mathrm{type} \ \} \ L$$
$$T \rightarrow \mathbf{int} \ \{ \ T.\mathrm{type} := \text{‘integer’} \ \}$$
$$T \rightarrow \mathbf{real} \ \{ \ T.\mathrm{type} := \text{‘real’} \ \}$$

```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute

Input:inherited attribute

Example showing Transformation of SDD to Translation scheme

Syntax Rules	Semantic rule
E → TE’	E’.ival=T.sval E.sval=E’.sval
E’ → ‘+’ TE’	E1’.ival=E.ival+T.val E’.sval=E1’.sval
E’ → e	E’.sval=E’.ival
T → NUM	T → NUM {T.val = atoi(N.lexval)}

Translation Scheme →

Translation Scheme
E → T {E’.ival=T.sval} E’ {E.sval=E’.sval}
E’ → ‘+’ T {E1’.ival=E’.ival+T.sval} E’ {E’.sval=E1’.sval}
E’ → e {E’.sval=E’.ival}
T → NUM {T.sval = atoi(N.lexval)}

Implementing L-Attributed Definitions in Bottom-Up Parsers

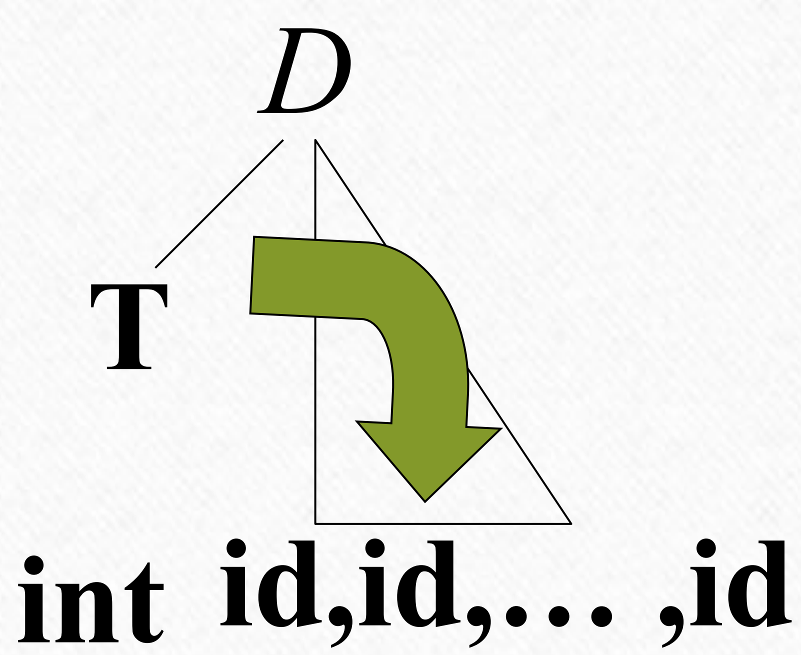
- More difficult and also requires rewriting L-attributed definitions into translation schemes
- Insert marker nonterminals to remove embedded actions from translation schemes, that is
  $A \rightarrow X$ { actions } Y
  is rewritten with marker nonterminal N into
  $A \rightarrow X\,N\,Y$
  $N \rightarrow \varepsilon$ { actions }
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

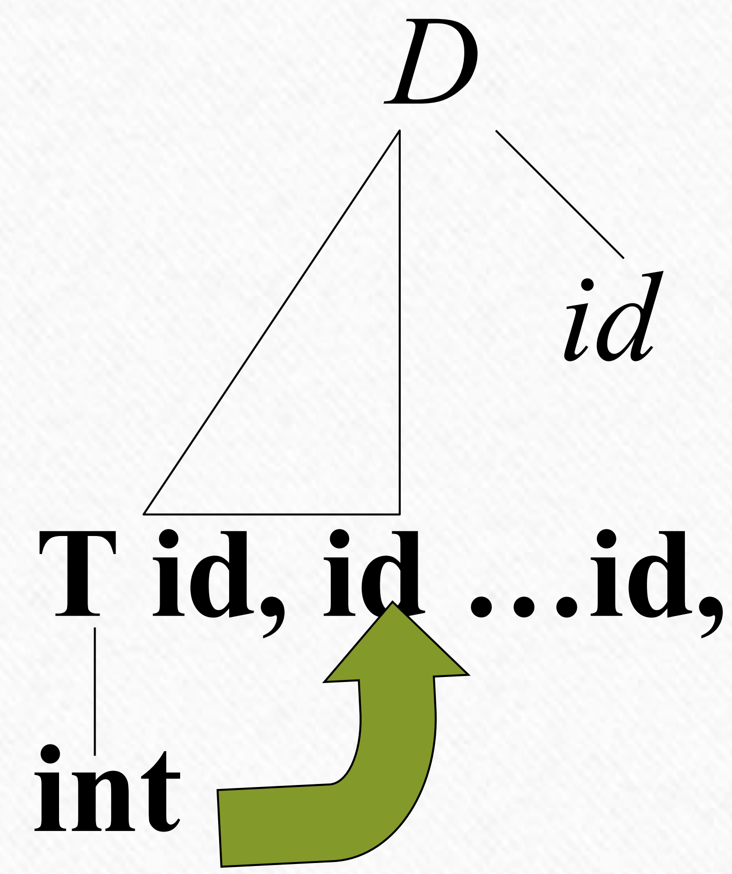
Emulating the Evaluation of L-Attributed Definitions in Yacc

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id**
{ *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

```
%{
Type Lin; /* global variable */
%}
%%
D   : Ts L
    ;
Ts  : T           { Lin = $1; }
    ;
T   : INT         { $$ = TYPE_INT; }
    | REAL        { $$ = TYPE_REAL; }
    ;
L   : L ',' ID { addtype($3, Lin);}
    | ID          { addtype($1, Lin);}
    ;
%%
```

Rewriting a Grammar to Avoid Inherited Attributes



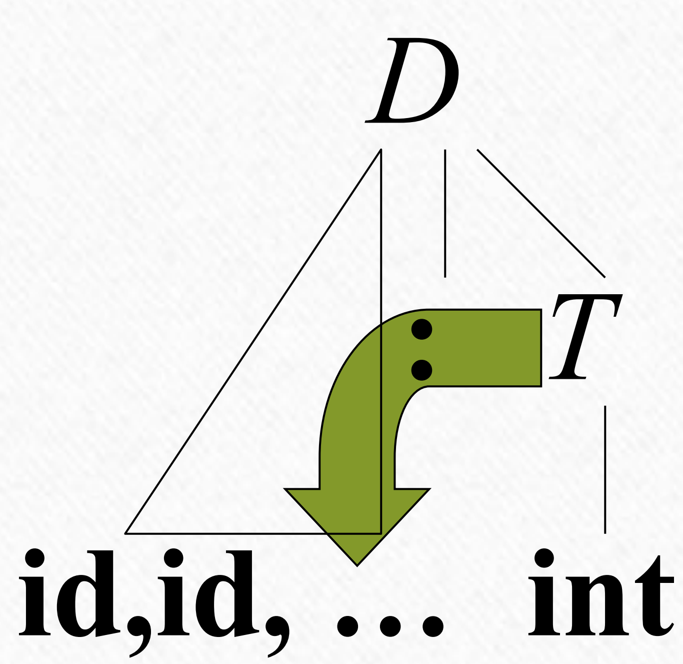


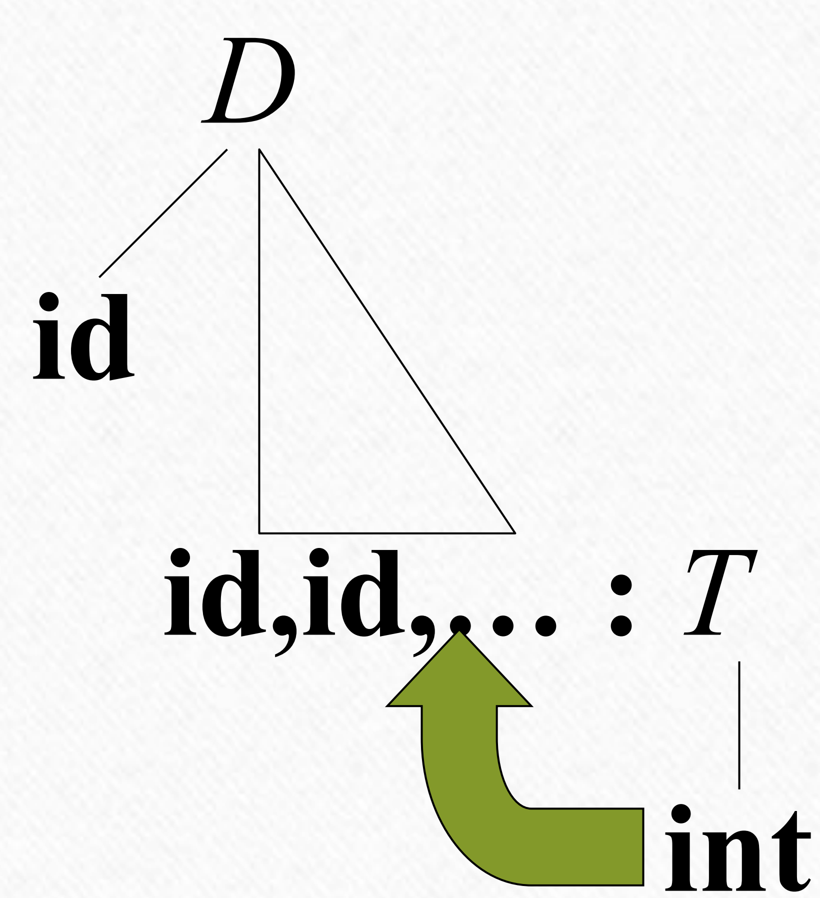
$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := ‘integer’ }
$T \rightarrow$ **real** { T.type := ‘real’ }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id**
{ *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

$D \rightarrow$ L id { addtype(id.entry, L.type)}
$T \rightarrow$ **int** { T.type := ‘integer’ }
$T \rightarrow$ **real** { T.type := ‘real’ }
$L \rightarrow L_1$ **id**, { *addtype*(**id**.entry, L_1.type)
L.type=L_1.type }
$L \rightarrow$ **T** { L.type = T.type }

Rewriting a Grammar to Avoid Inherited Attributes

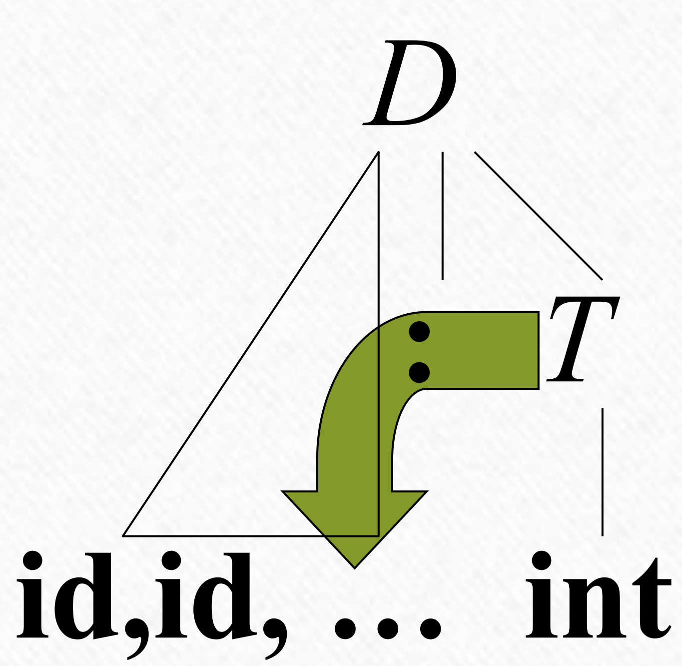
Production	Production	Semantic Rule
$D \rightarrow L : T$	$D \rightarrow \textbf{id}\ L$	$addtype(\textbf{id}.\text{entry}, L.\text{type})$
$T \rightarrow \textbf{int}$	$T \rightarrow \textbf{int}$	$T.\text{type} :=$ 'integer'
$T \rightarrow \textbf{real}$	$T \rightarrow \textbf{real}$	$T.\text{type} :=$ 'real'
$L \rightarrow L_1\ , \textbf{id}$	$L \rightarrow\ , \textbf{id}\ L_1$	$addtype(\textbf{id}.\text{entry}, L.\text{type})$
$L \rightarrow \textbf{id}$	$L \rightarrow : T$	$L.\text{type} := T.\text{type}$

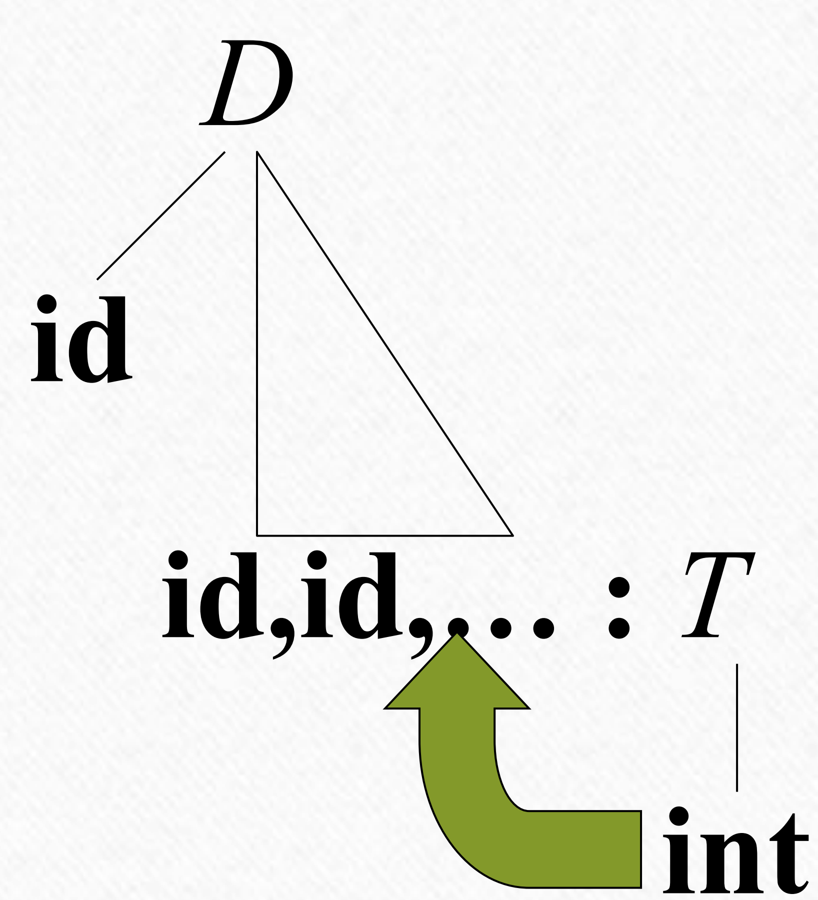




Rewriting a Grammar to Avoid Inherited Attributes

Production	Production	Semantic Rule
$D \to L : T$	$D \to$ **id** L	*addtype*(**id**.entry, L.type)
$T \to$ **int**	$T \to$ **int**	T.type := 'integer'
$T \to$ **real**	$T \to$ **real**	T.type := 'real'
$L \to L_1$, **id**	$L \to$, **id** L_1	*addtype*(**id**.entry, L.type)
$L \to$ **id**	$L \to : T$	L.type := T.type





Summary of Topics discussed

- Difference between Translation Scheme and Syntax Directed Definition
- Types of Syntax Directed Definitions
- **Eliminating Leftmost recursion from Translation Scheme/SDD**
- Implementation of S-attributed definition in Bottom-up Parser
- Implementation of L-attributed definition in Bottom-up parser
- Implementation of S-attributed definition in Top-down Parser
- Implementation of L-attributed definition in Top-down parser
- Implementing semantic rules in YACC

Eliminating Leftmost recursion from Translation Scheme/ SDD

- E : E +T { E.val = f(E1.val,T.val)=(E.val+T.val)}
- E : T { E.val =g(T.val)=T.val}

- After Eliminate leftmost recurson
- E : T E’ { E’.ival =g(T.val) =T.val ; E.sval=E’ .sval}
- E’ : + T E’ { E1’.ival=f(E’.ival,T.val)=E’.ival+T.ival ; E’.sval=E1’.sval}
- E’ : null { E’.sval = E’.ival }

Eliminating Leftmost recursion from SDD

SDD Before Left recursion elimination	SDD After Left recursion elimination	Translation scheme
E : E + T { E.val=E1.val + T.val} OR {E.val= f(E1.val,T.val)}	E : TE' {E.val= E'.sval E'.ival=g(T.sval)}	E: T {E.ival=g(T.sval)}E' {E.sval=E'.sval}
E : T {E.val=T.val} or {E.val=g(T.val)}	E' : +TE' { E1'.ival = f(E'.ival, T.sval) E'.sval =E1.sval}	E': + T {E1'.ival=f(E'.ival,T.sval)} E1' {E'.sval = E1'.sval}
	E' : e {E'.sval = E'.ival}	E' : e {E'.sval = E'.ival}

Implementing Semantic rules in YACC

<u>SDD:</u>

E:E+T {E.val=E1.val+T.val}
E:E-T {E.val=E1.val-T.val}
E: T {E.val=T.val}
T: NUM {T.val=NUM.val}

Expr.y
======

```
%{
%}
%token NUM
%%
E : E '+' T    {$$=$1+$3;}
E : E '-' T    {$$=$1-$3;}
E : T          {$$=$1;}
T : NUM        {$$=$1;}
%%
```

Expr.l
======

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[+-]        {return yytext[0];}
[0-9]+      {yylval=atoi(yytext);
             return NUM;}
%%
```

Implementing Semantic rules(with symbol table) in YACC

<u>SDD:</u>

E:E+T {E.val=E1.val+T.val}
E:E-T {E.val=E1.val-T.val}
E: T {E.val=T.val}
T: NUM {T.val=NUM.val}
T: ID {T.val=lookup(ID.entry, value)}

Expr.y

```
======
%{
int symbols[26]={0};
%}
%token NUM
%%
E : E '+' T    {$$=$1+$3;}
E : E '-' T    {$$=$1-$3;}
E : T          {$$=$1;}
T : NUM        {$$=$1;}
T : ID   {$$= symbols[$1];}
%%
```

Expr.l

```
======
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[+-]        {return yytext[0];}
[0-9]+      {yylval=atoi(yytext);
             return NUM;}
[a-z]       {yylval=yytext[0]-'a';
             return ID;}
%%
```

Implementing Semantic rules(with symbol table) in YACC

SDD:

A: ID = A {A.val=A1.val
update(ID.entry,value)=A1.value}
A: E ';' {A.val=E.val;
print E.val}
E:E+T {E.val=E1.val+T.val}
E:E-T {E.val=E1.val-T.val}
E: T {E.val=T.val}
T: NUM {T.val=NUM.val}
T: ID {T.val=lookup(ID.entry,
value)}

Expr.y
======

```
%{
int symbols[26]={0};
%}
%token NUM
%%
A: ID '=' A   {symbols[$1]=$3;
               $1=$3;}
A: E ';'      {$$=$1;
  printf("ans=%d\n",$1);}
E : E '+' T   {$$=$1+$3;}
E : E '-' T   {$$=$1-$3;}
E : T         {$$=$1;}
T : NUM       {$$=$1;}
T : ID  {$$= symbols[$1];}
%%
```

Expr.l
======

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[;=+-]       {return yytext[0];}
[0-9]+     {yylval=atoi(yytext);
            return NUM;}

[a-z]       {yylval=yytext[0]-'a';
            return ID;}

%%
```

Implement Calculator using Symbols of variable length in YACC

Symbol Table = Linklist of node (name, value, ptr)

Name	Value	Next

Expr.l

```
======
%{
#include "y.tab.h"
#include "mystruct.h"
%}
%%
[;=+-]     {return yytext[0];}
[0-9]+     {yylval=atoi(yytext);  return NUM;}
([_a-zA-Z][0-9]*)+  {yylval=lookup(yytext);
return ID;}
%%
```

```
struct node *lookup( char str[])
{
struct node *temp=head;
while(temp->next!=NULL)
        {
        if(strcmp(temp->name,str)==0)
                return temp;
        temp=temp->next
        }
newnode=(struct node *) malloc(sizeof(struct node));
newnode->value =0;
strcpy(newnode=>name,str);
newnode->next=NULL;
temp->next=newnode
Return newnode; }
```

Implement Calculator using Symbols of variable length in YACC

Symbol Table = Linklist of node (name, value, ptr)

Name	Value	Next

Expr.y

```
======
%{
int symbols[26]={0};
%}
%union {
   int ival;
  struct node *nval;
}
%token <ival> NUM
%token <nval> ID
%type <ival> T E A
```

```
%%
A: ID '=' A   {$1->value=$3;
                $1=$3;}
A: E ';'       {$$=$1;
   printf("ans=%d\n",$1);}
E : E '+' T    {$$=$1+$3;}
E : E '-' T    {$$=$1-$3;}
E : T          {$$=$1;}
T : NUM        {$$=$1;}
T : ID   {$$= $1->value;}
%%
```

Implement Datatype allocation and type verification

E.g.

Name	Data Type	Value
a	1	
f	1	
c	0	

```
 {
int a;
float   f;
char  a; //YACC report error by(semantic analysis ) Mulitple declaration of  a
f  = a+ c;   // YACC report error by (semantic analysis) undeclared c
```

Implementation data type allocation and verification

<u>SDD</u>

```
SS : SS S | S
S   : DS    | E ';'
DS : T  {L.in=T.val} L;
L:  L ',' ID  {L1.in=L.in,
                  if(ID.entry, type)!=NULL
                     error=Multiple declaration
                  else
                      update(ID.entry,type)=L.in}
  | ID {update(ID.entry,type)=L.in}
E : E '+' F
   | F
F : ID   {if lookup(ID.entry,type)==NULL)
                        print undeclared;}
    | NUM
```

<u>Expr.y : rule section</u>

```
%%
SS: SS S
    | S
S : DS
  | E ';'
DS : Ts L
Ts  : T        { Type=$1;}
L :  L ',' ID { if($1->type!=0)  $1->type=Type;}
   else printf("Multple dec of %s", $1->name);}
  | ID        {if($1->type!=0)  $1->type=Type;}
   else printf("Multple dec of %s", $1->name);}
E : E '+' F
    |F
F: ID   {if($1->type==0) printf("undeclared");}
    | NUM
```