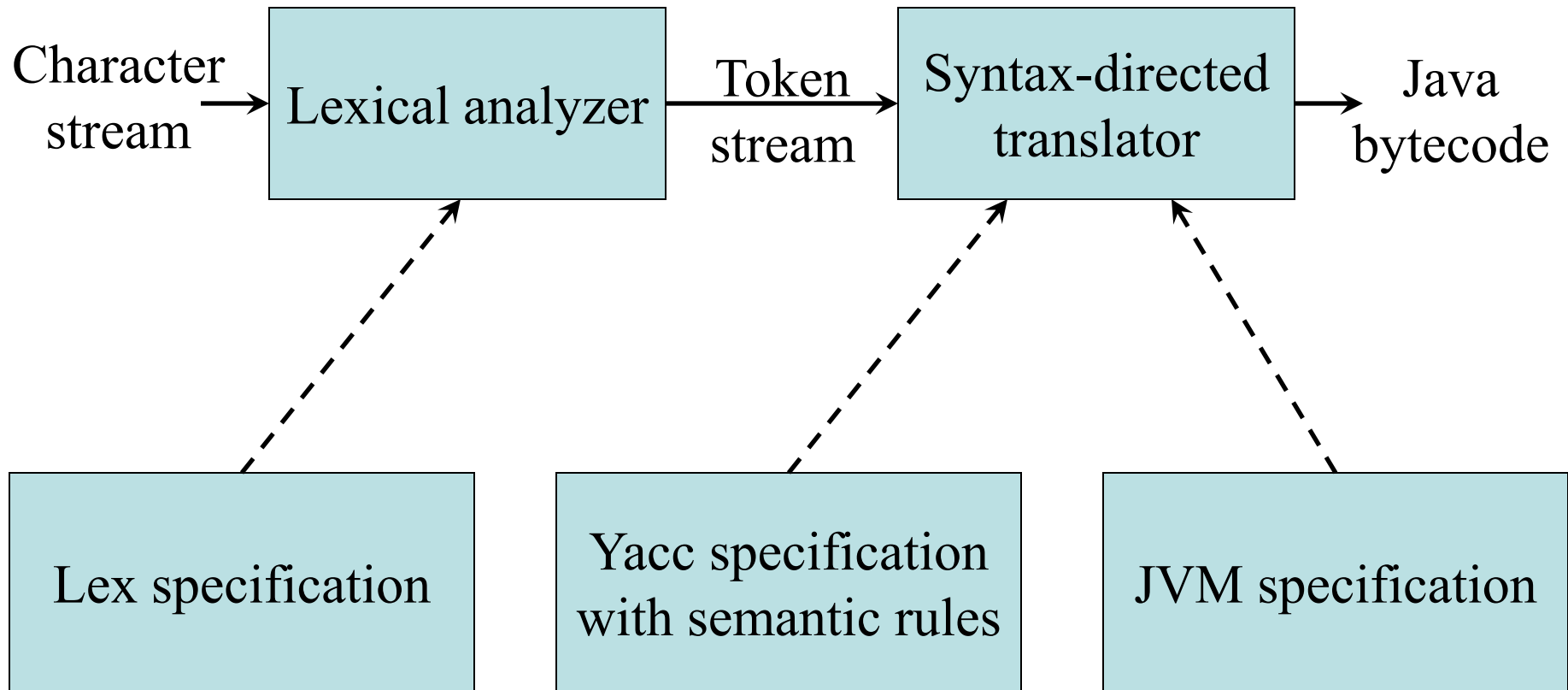


Syntax-Directed Translation

Part I

Chapter 5

The Structure of our Compiler Revisited



Syntax-Directed Definitions

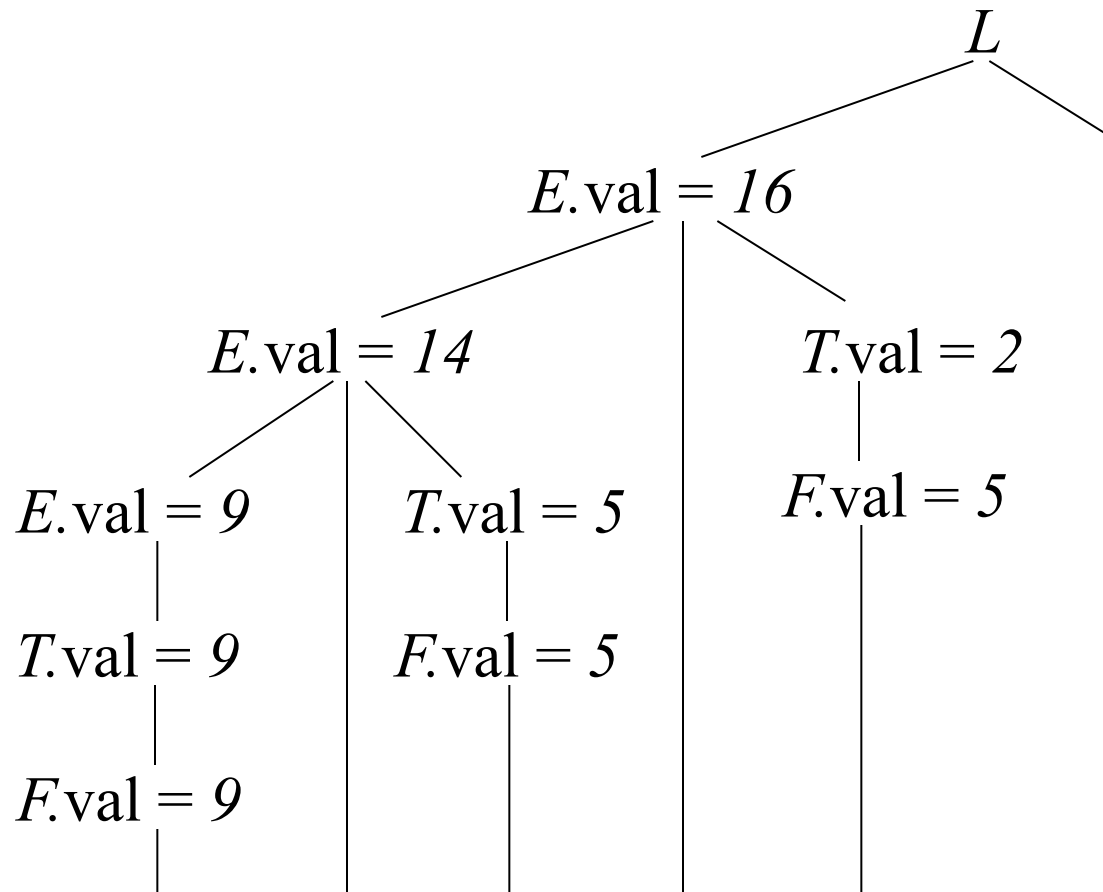
- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes* holding values set by the semantic rules
- A *depth-first traversal* algorithm traverses the parse tree thereby executing semantic rules to assign attribute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

Example Attribute Grammar

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

Note: all attributes in this example are of the synthesized type

Example Annotated Parse Tree

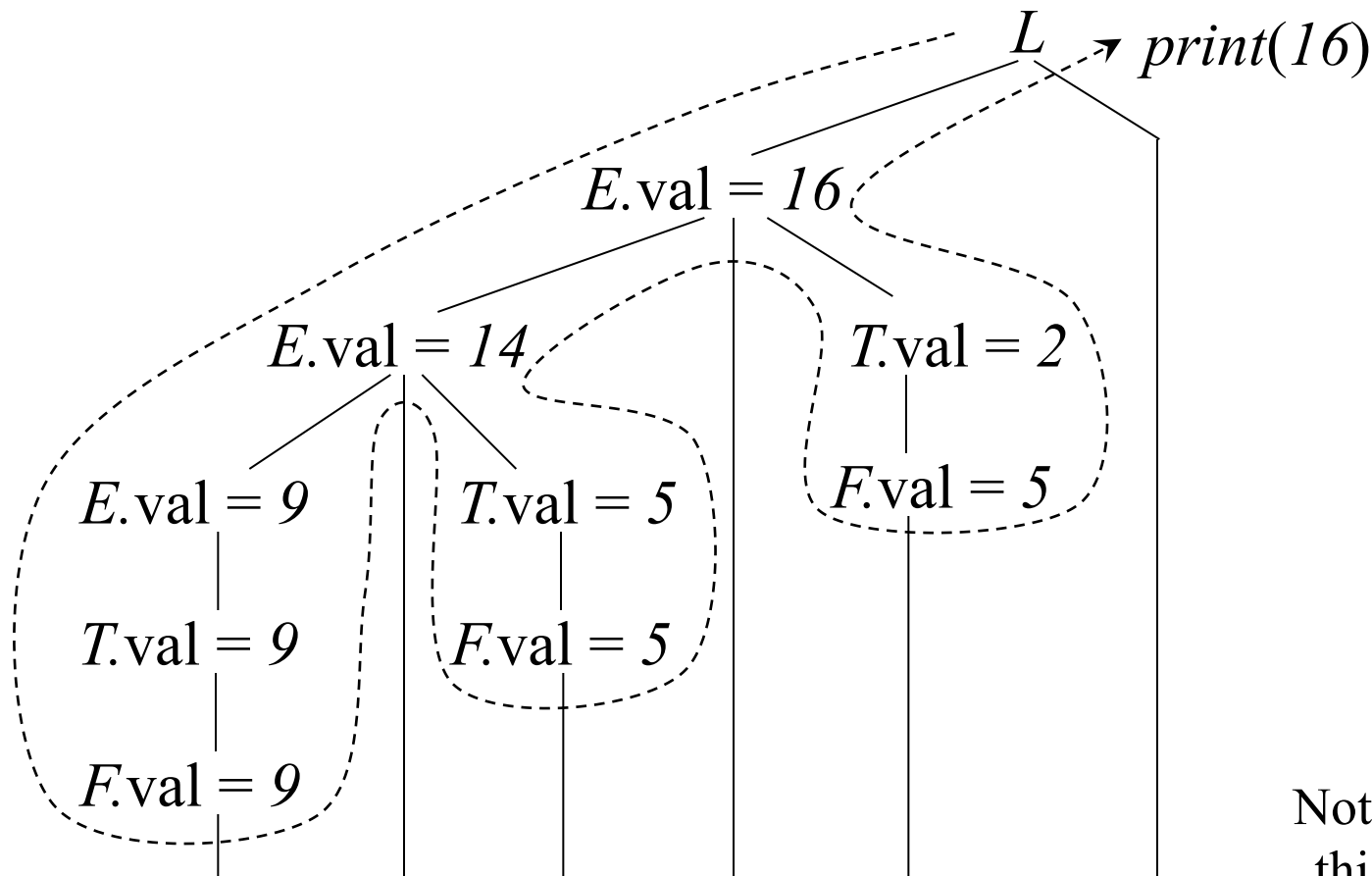


Note: all attributes in this example are of the synthesized type

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)



Note: all attributes in this example are of the synthesized type

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

$$b := f(c_1, c_2, \dots, c_k)$$

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	
$D \rightarrow T L$	$L.in := T.type$	inherited
$T \rightarrow \mathbf{int}$	$T.type := \text{'integer'}$	
...	...	
$L \rightarrow \mathbf{id}$	$\dots := L.in$	synthesized

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; }
```

```
;
```

```
%%
```

```
AVAILABLE AT:
```

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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 $F \rightarrow \text{digit}$	$$$ = \1
\$ F	3	*5+4n\$	reduce $T \rightarrow F$	$$$ = \1
\$ T	3	*5+4n\$	shift	
\$ T *	3	5+4n\$	shift	
\$ T * 5	3 5	+4n\$	reduce $F \rightarrow \text{digit}$	$$$ = \1
\$ T * F	3 5	+4n\$	reduce $T \rightarrow T * F$	$$$ = \$1 * \$3$
\$ T	15	+4n\$	reduce $E \rightarrow T$	$$$ = \1
\$ E	15	+4n\$	shift	
\$ E +	15	4n\$	shift	
\$ E + 4	15 4	n\$	reduce $F \rightarrow \text{digit}$	$$$ = \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 $L \rightarrow E n$	<i>print</i> \$1
\$ L	19	\$	accept	

Example Attribute Grammar with Synthesized+Inherited Attributes

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

Synthesized: $T.type, \mathbf{id}.entry$

Inherited: $L.in$

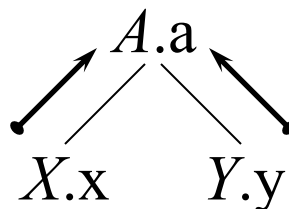
AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

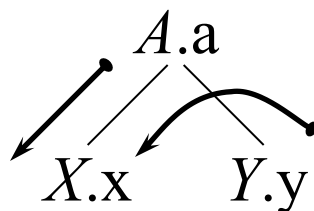
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Acyclic Dependency Graphs for Parse Trees

$$A \rightarrow XY$$



$$A.a := f(X.x, Y.y)$$

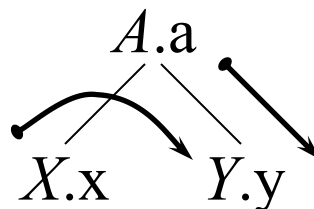


$$X.x := f(A.a, Y.y)$$

Direction of



value dependence



$$Y.y := f(A.a, X.x)$$

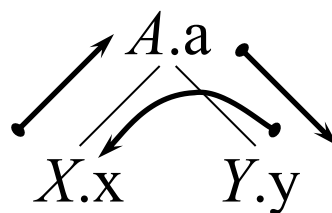
AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic



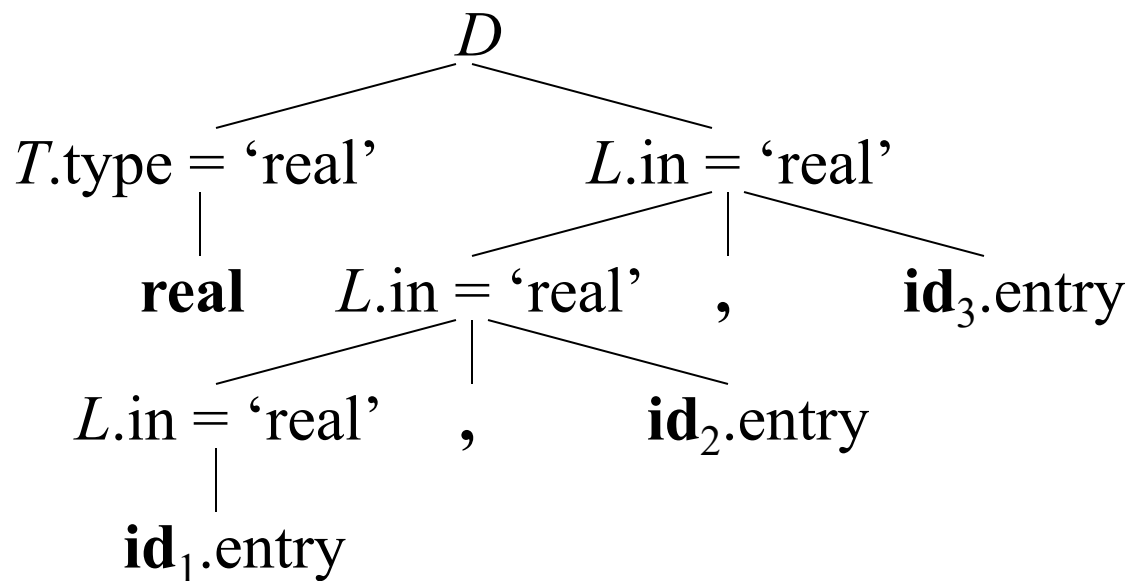
$$A.a := f(X.x)$$

$$X.x := f(Y.y)$$

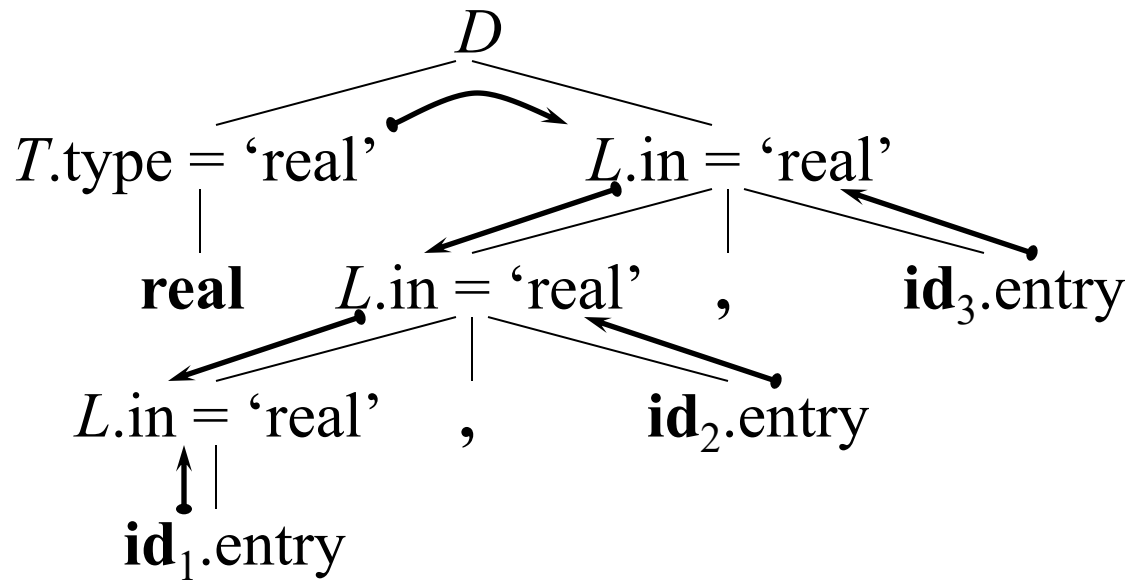
$$Y.y := f(A.a)$$

Error: cyclic dependence

Example Annotated Parse Tree



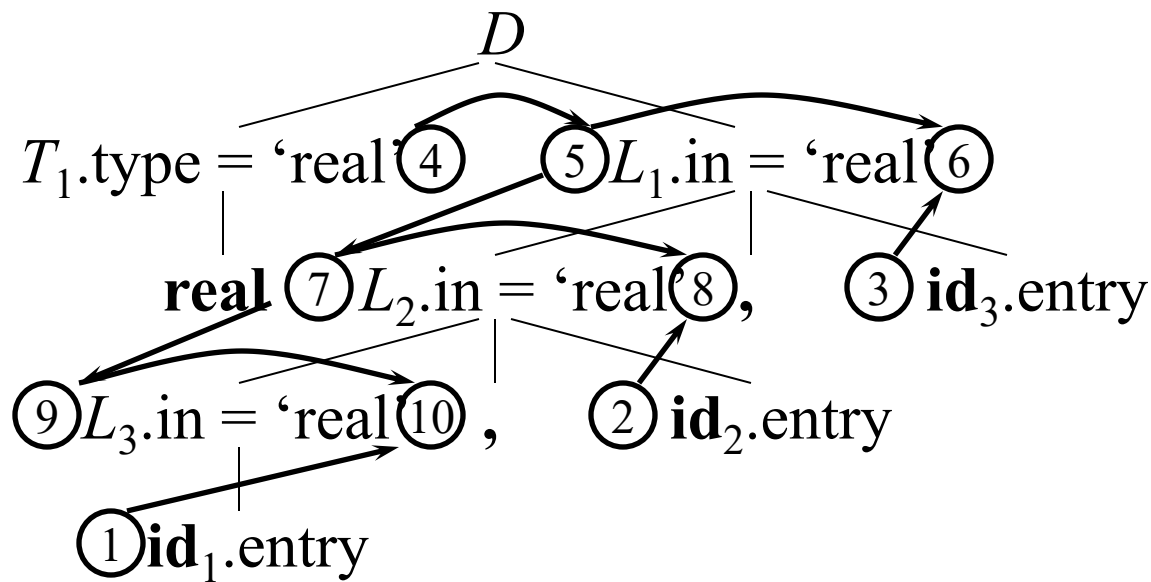
Example Annotated Parse Tree with Dependency Graph



Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \dots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ 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. $addtype(id_1.entry, L_3.in)$

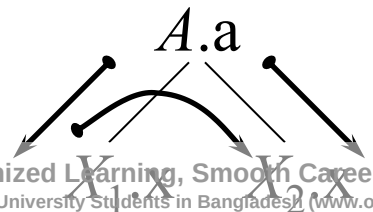
Evaluation Methods

- *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)

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 inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \dots X_n$ depends only on
 1. the attributes of the symbols $X_1, X_2, \dots, X_{j-1}$
 2. the inherited attributes of A

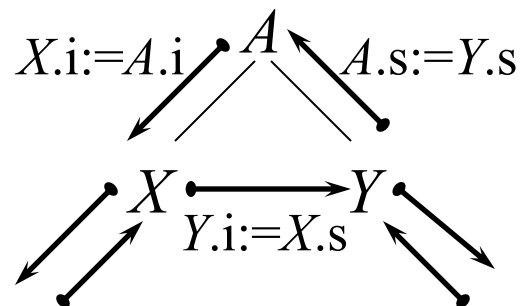
Shown: dependences
of inherited attributes



L-Attributed Definitions (cont'd)

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

$A \rightarrow X Y$



$X.i := A.i$
 $Y.i := X.s$
 $A.s := Y.s$

- Note: every S-attributed syntax-directed definition is also L-attributed

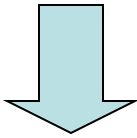
AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Using Translation Schemes for L-Attributed Definitions

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



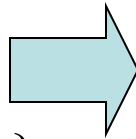
Translation Scheme

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

Implementing L-Attributed Definitions in Top-Down Parsers

Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

$D \rightarrow T \{ L.in := T.type \} L$
 $T \rightarrow \text{int} \{ T.type := \text{'integer'} \}$
 $T \rightarrow \text{real} \{ T.type := \text{'real'} \}$



```

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

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

$$A \rightarrow X \{ \text{actions} \} Y$$

is rewritten with marker nonterminal N into

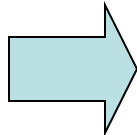
$$A \rightarrow X N Y$$

$$N \rightarrow \varepsilon \{ \text{actions} \}$$

- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

Emulating the Evaluation of L-Attributed Definitions in Yacc

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



```

%{
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

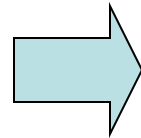
Production

$$D \rightarrow L : T$$

$$T \rightarrow \text{int}$$

$$T \rightarrow \text{real}$$

$$L \rightarrow L_1, \text{id}$$

$$L \rightarrow \text{id}$$


Production

$$D \rightarrow \text{id } L$$

$$T \rightarrow \text{int}$$

$$T \rightarrow \text{real}$$

$$L \rightarrow , \text{id } L_1$$

$$L \rightarrow : T$$

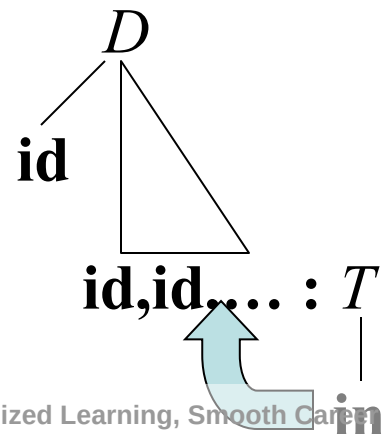
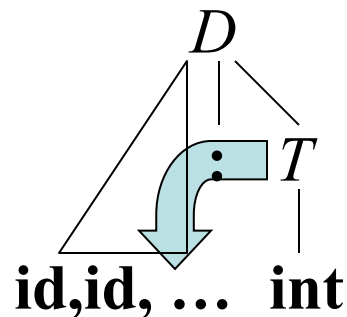
Semantic Rule

$$\text{addtype}(\text{id.entry}, L.\text{type})$$

$$T.\text{type} := \text{'integer'}$$

$$T.\text{type} := \text{'real'}$$

$$\text{addtype}(\text{id.entry}, L.\text{type})$$

$$L.\text{type} := T.\text{type}$$


AVAILABLE AT: