Principles of Compiler Construction

Prof. Wen-jun LI

School of Computer Science and Engineering

lnslwj@mail.sysu.edu.cn

Lecture 8. Syntax-Directed Translation

1. Introduction
2. Syntax-Directed Definition: Examples
3. Evaluation Order and Dependency Graphs
4. S-Attributed Definitions
5. L-Attributed Definitions and Translation Schemes
6. L-Attributed Definitions in Predictive Parsing
7. L-Attributed Definitions in LR Parsing

1. Introduction

- What is syntax-directed translation?
  - E.g. denotational semantics
    - Every component has its denotation.
    - The denotation of a composite component depends only on the denotations of its sub-components.
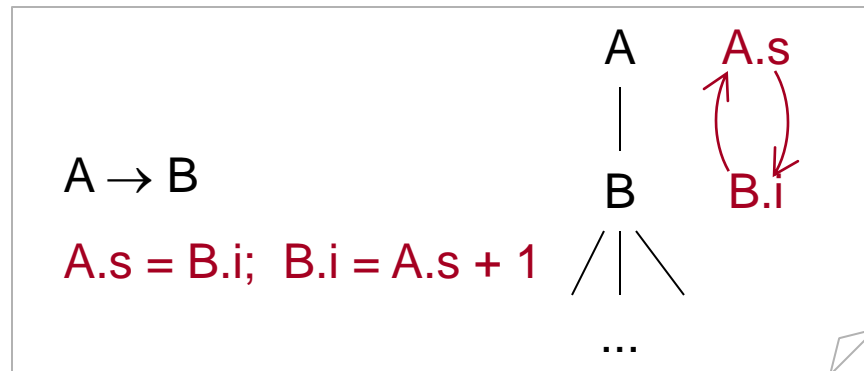- Why syntax-directed?
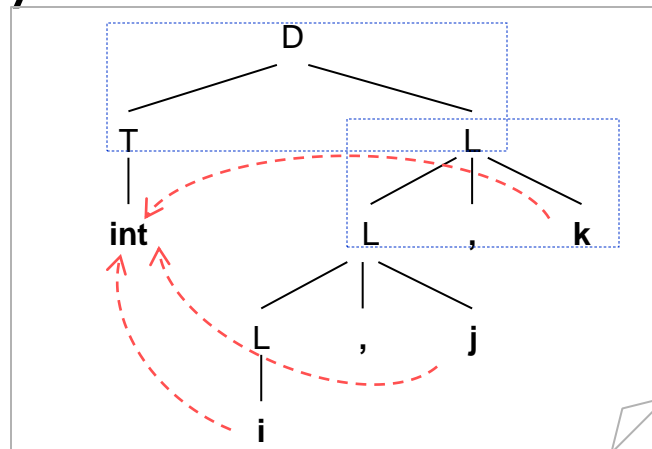
Applications of Syntax-Directed Translation

- Two compiling phases are covered
    - Semantic analysis
    - Intermediate code generation
- Our learning steps
    1. The general concepts and framework of syntax-directed translation
    2. Application of these concepts and framework to semantic analysis and intermediate code generation.

Basic Ideas

- Associate grammar symbols with attributes
  - Based on specific applications.
  - The **data** to be manipulated.
- Associate productions with semantic rules
  - Also named semantic actions.
  - The **operations** that manipulate the attributes.

$A \rightarrow B$

$A.s = B.i; \quad B.i = A.s + 1$

More Insights: A Virtual Machine

- Syntax-directed translation can be explained as a virtual machine
  - What to do: semantic actions.
  - How to do: evaluation order, i.e. the order in which the actions are performed.

attributes **VM** *semantic actions*
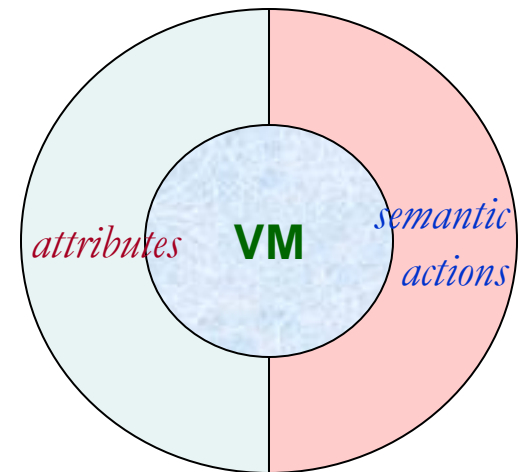
Challenges

○ Efficiency of the decision of semantic actions execution order

- The best way is to execute the semantic actions while parsing.

- I.e. the evaluation order of actions is the same as the order of parsing output.

- Trade-off: capability vs. efficiency

Concepts

- Syntax-directed definition
  - Translation Scheme
- Annotated parse tree
  - Annotated parse tree with actions
- Synthesized Attribute
- Inherited Attribute

Syntax-Directed Definition

	Productions	Semantic Rules
1	L $\rightarrow$ E **n**	L.val = E.val print(L.val)
2	E $\rightarrow$ E_1 **+** T	E.val = E_1.val + T.val
3	E $\rightarrow$ T	E.val = T.val
4	T_1 $\rightarrow$ T_2 ***** F	T_1.val = T_2.val * F.val
5	T $\rightarrow$ F	T.val = F.val
6	F $\rightarrow$ **(** E **)**	F.val = E.val
7	F $\rightarrow$ **digit**	F.val = **digit**.lexval

Side-effects

Different
subfix style

Annotated Parse Tree

variables

constants

Synthesized attributes
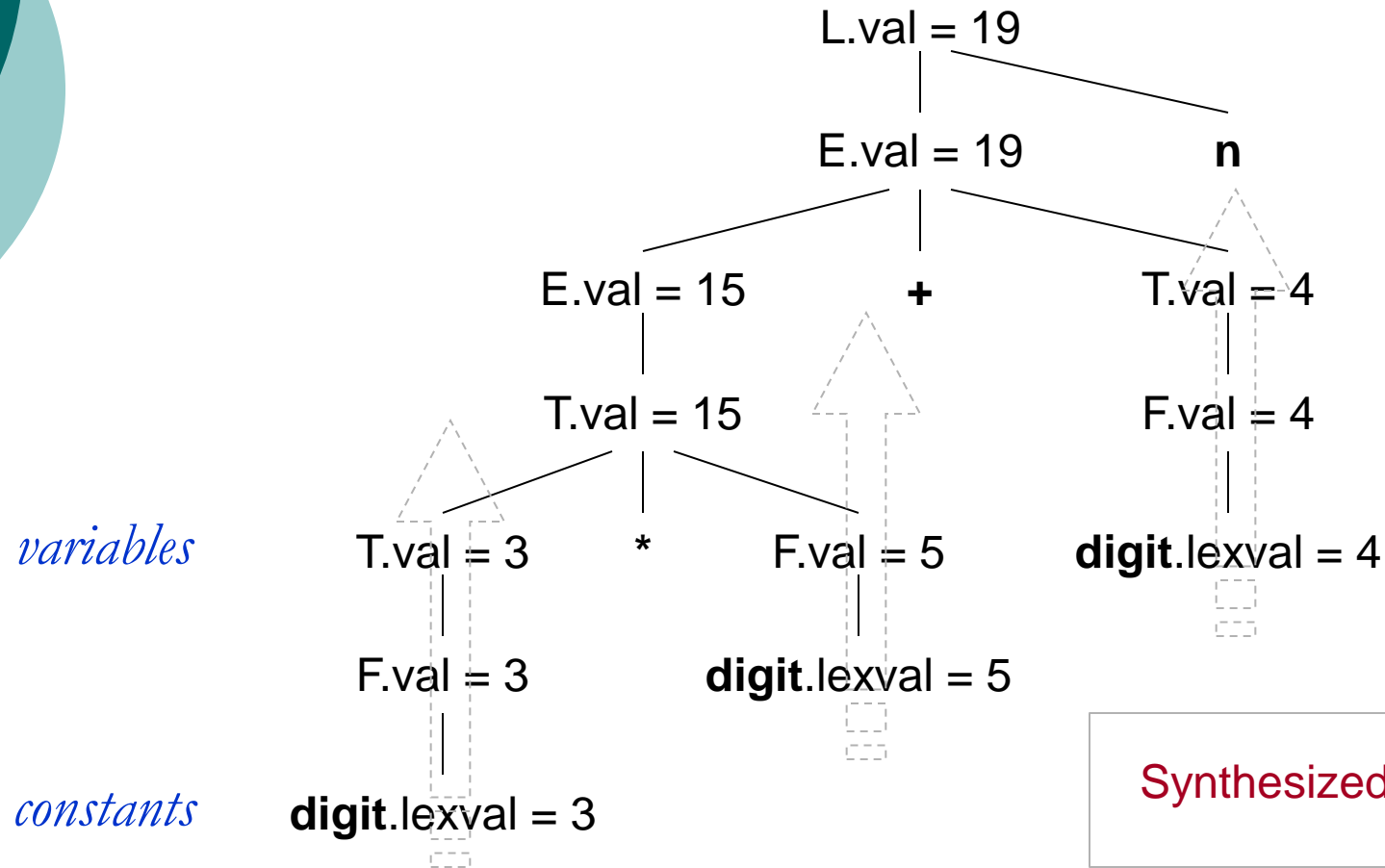
Inherited Attribute

No.	Productions	Semantic Rules
1	D $\rightarrow$ T L	L.inh = T.type
2	T $\rightarrow$ **int**	T.type = INTEGER
3	T $\rightarrow$ **real**	T.type = REAL
4	L $\rightarrow$ L$_1$ **, id**	L$_1$.inh = L.inh addType(**id**.entry, L.inh)
5	L $\rightarrow$ **id**	addType(**id**.entry, L.inh)

Annotated Parse Tree

D

$T.type = REAL$ ⟶ $L.inh = REAL$

real

$L.inh = REAL$, **id$_3$**

$L.inh = REAL$, **id$_2$**

id$_1$
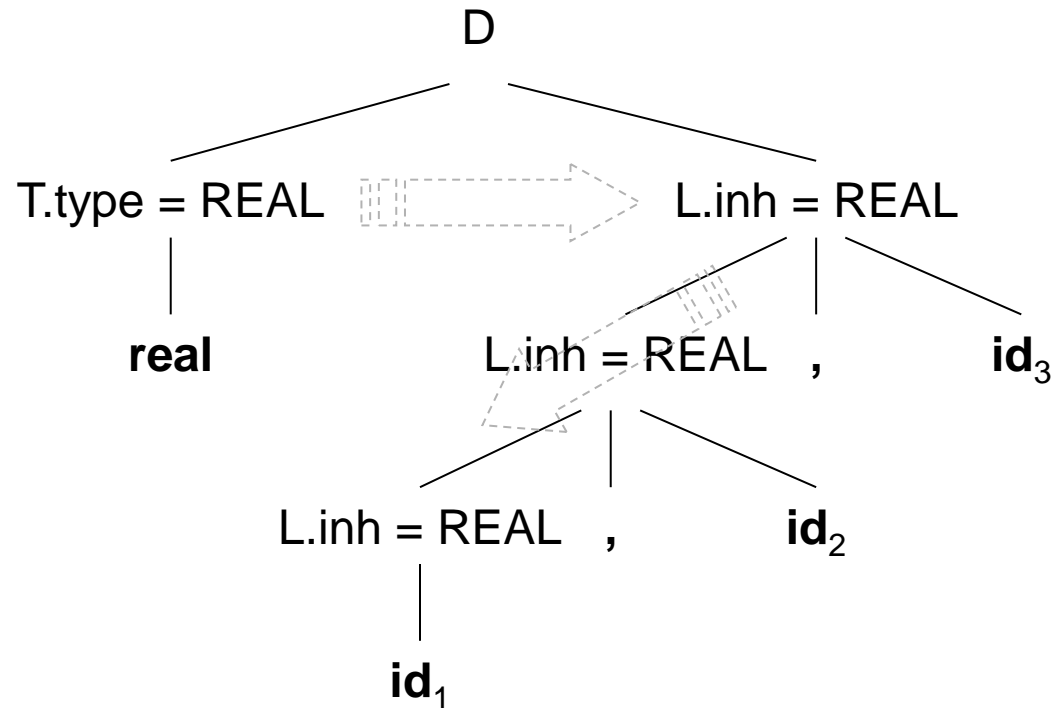
2. Syntax-Directed Definition: Examples

○ Construction of Syntax Trees
○ Construction of DAG
○ Type Structure of Arrays

Syntax Trees

No.	Productions	Semantic Rules
1	$E \rightarrow E_1$ **+** T	E.node = **new** Node('+', E_1.node, T.node)
2	$E \rightarrow E_1$ **–** T	E.node = **new** Node('–', E_1.node, T.node)
3	$E \rightarrow T$	E.node = T.node
4	$T \rightarrow$ **(** E **)**	T.node = E.node
5	$T \rightarrow$ **id**	T.node = **new** Leaf(**id**, **id**.entry)
6	$T \rightarrow$ **num**	T.node = **new** Leaf(**num**, **num**.val)

Syntax Trees (cont')

○ Syntax tree for **a − 4 + c**

(1) $p_1 =$ **new** Leaf(**id**, entry-a)
(2) $p_2 =$ **new** Leaf(**num**, 4)
(3) $p_3 =$ **new** Node('**−**', p_1, p_2)
(4) $p_4 =$ **new** Leaf(**id**, entry-c)
(5) $p_5 =$ **new** Node('**+**', p_3, p_4)
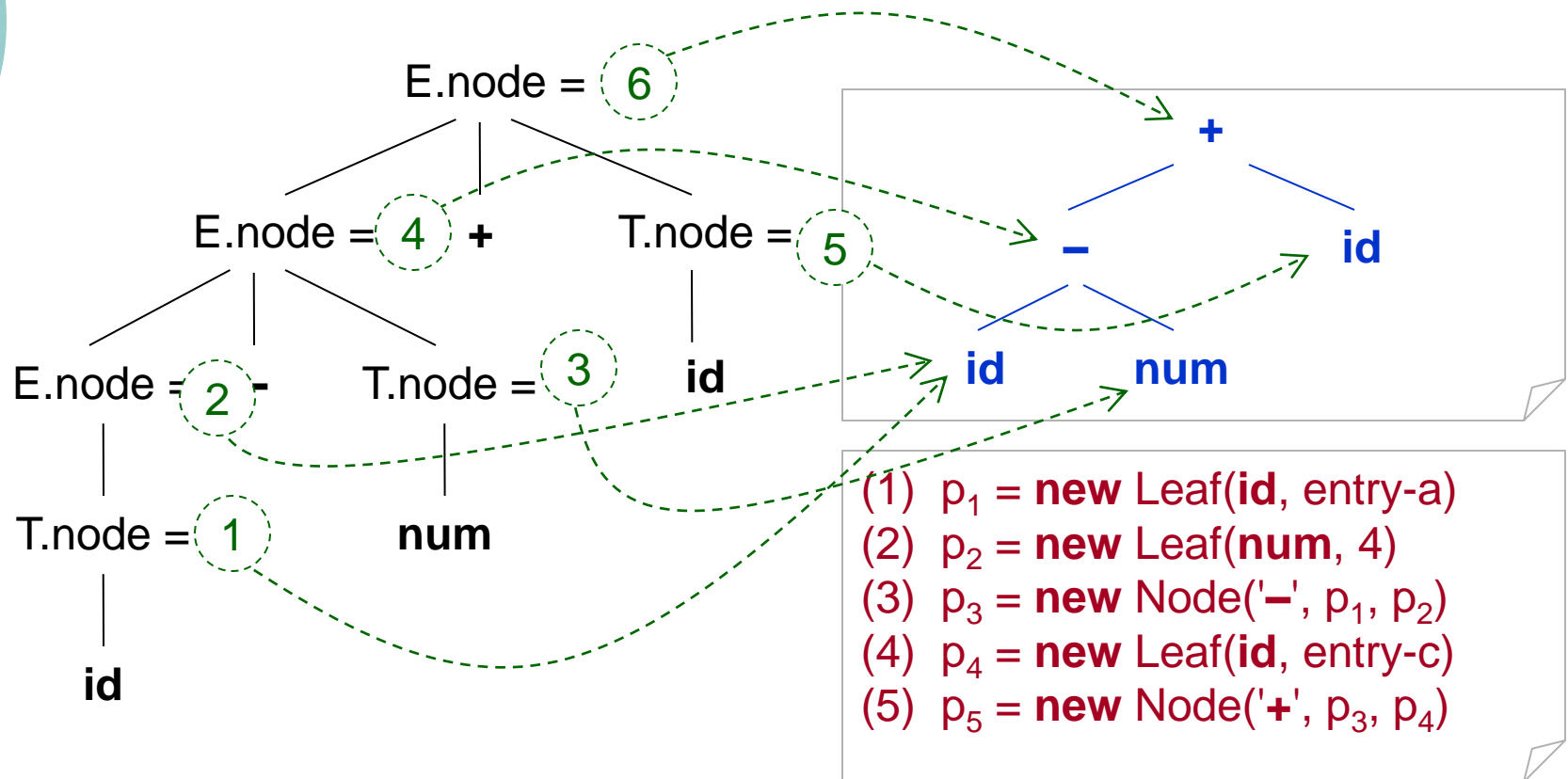
DAG

- DAG: Directed Acyclic Graph
  - **a + a * (b – c) + (b – c) * d**
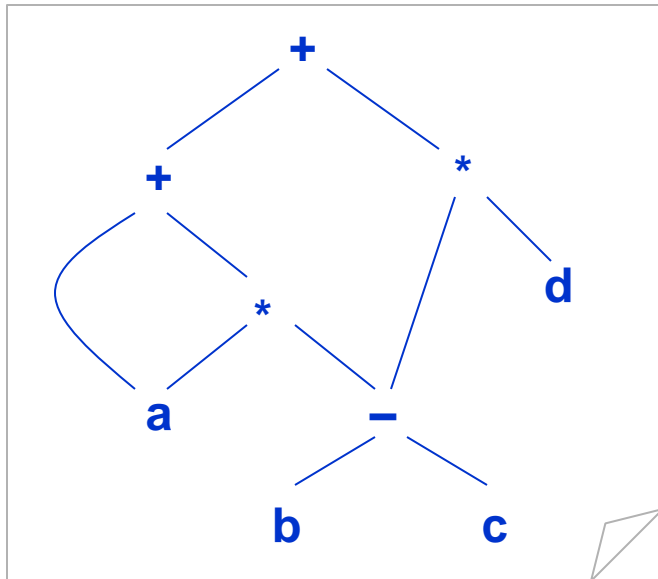
(1) p_1 = **new** Leaf(**id**, entry-a)

(2) p_2 = **new** Leaf(**id, entry-a**)

(3) p_3 = **new** Leaf(**id**, entry-b)

(4) p_4 = **new** Leaf(**id**, entry-c)

(5) p_5 = **new** Node('**–**', p_3, p_4)

(6) p_6 = **new** Node('*', p_2, p_5)

(7) p_7 = **new** Node('**+**', p_1, p_6)

(8) p_8 = **new** Leaf(**id**, entry-b)

(9) p_9 = **new** Leaf(**id**, entry-c)

(10) p_{10} = **new** Node('**–**', p_8, p_9)

(11) p_{11} = **new** Leaf(**id**, entry-d)

(12) p_{12} = **new** Node('*', p_{10}, p_{11})

(13) p_{13} = **new** Node('**+**', p_7, p_{12})

Type Structures

No.	Productions	Semantic Rules
1	T $\rightarrow$ B C	T.type = C.type C.base = B.type
2	B $\rightarrow$ **int**	B.type = int
3	B $\rightarrow$ **float**	B.type = $float$
4	C $\rightarrow$ **[num]** C_1	C.type = $array$(**num**.val, C_1.type) C_1.base = C.base
5	C $\rightarrow$ ε	C.type = C.base

Type Structures (cont')

○ Type structure for **int[2][3]**

$$T.type = array(2, array(3, integer))$$

$$B.type = integer$$

$$C.base = integer$$
$$C.type = array(2, array(3, integer))$$

int

[**2**   **]**

$$C.base = integer$$
$$C.type = array(3, integer)$$

[**3**   **]**

$$C.base = integer$$
$$C.type = integer$$

ε
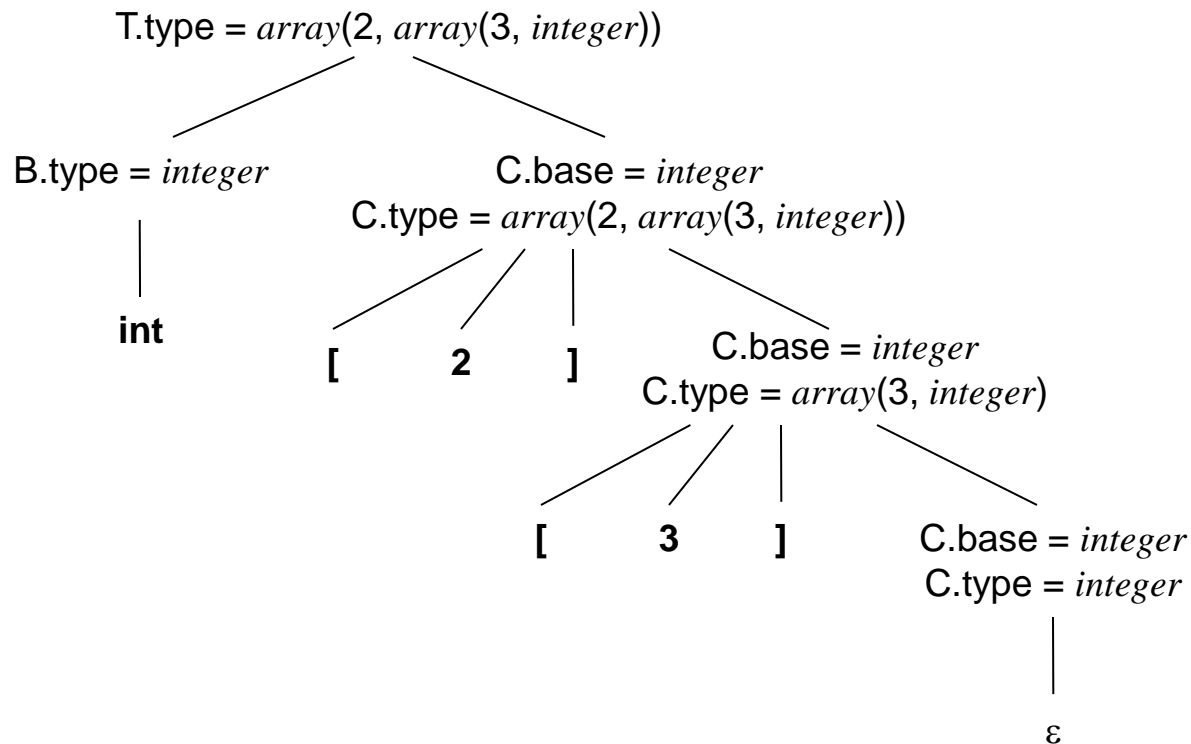
3. Evaluation Order and Dependency Graphs

○ A general framework of working steps:
- Introduce attributes to grammar symbols.
- Define semantic rules for each production.
- Draw the **dependency graph** based on the parse tree.
- Determine the **evaluation order** by topological sorting of the dependency graph.
- Execute the semantic rules according to the evaluation order.

Work with **explicit** parse tree

Pros and Cons

○ Pros
  - A general and powerful approach.
  - Can be used to demonstrate the principles of evaluation order of syntax-directed definitions.

○ Cons
  - Explicit parse tree.
  - Low efficiency and impractical.

The Previous Syntax-Directed Definition

No.	Productions	Semantic Rules
1	D $\rightarrow$ T L	L.inh = T.type
2	T $\rightarrow$ **int**	T.type = INTEGER
3	T $\rightarrow$ **real**	T.type = REAL
4	L $\rightarrow$ L$_1$ **, id**	L$_1$.inh = L.inh addType(**id**.entry, L.inh)
5	L $\rightarrow$ **id**	addType(**id**.entry, L.inh)

Dependency Graph: An Example
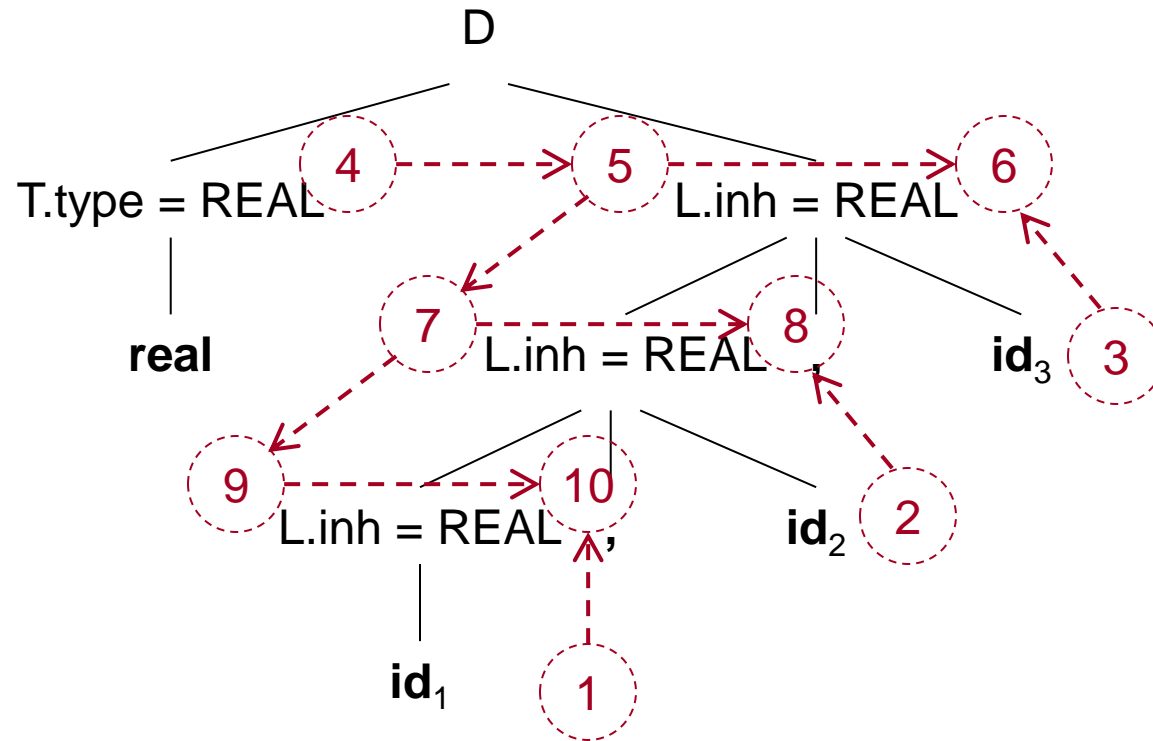
6, 8, 10 are dummy attributes
addType(id.entry, L.inh)

Evaluation Order

- Topological sorting
  - Specification: dependency graph
  - Implementation: evaluation order
- Mapping a specification to an implementation
  - 1 : 1 -- without any cycles in the graph
  - 1 : n -- without any cycles in the graph
  - 1 : 0 -- with cycles in the graph

Implicit Parse Trees

- S-attributed definitions
  - Every attribute is synthesized.
  - Guarantee an evaluation order that is the same order of the output of LR parsing.

- L-attributed definitions
  - Every attribute is synthesized, or only inherited from parent or left siblings (not from right siblings).
  - Guarantee an evaluation order that is the same order of recursive descent predictive parsing.

4. S-Attributed Definitions

- A syntax-directed definition with only synthesized attributes

- Evaluation order for S-attributed definitions
  - Upwards, and only upwards
  - The same order as LR parsing!

Specification vs. Implementation

- Evaluate S-attributed definitions while bottom-up parsing

```
┌─────────────────────────────────┐
│     S-Attributed Definition     │      Specification
└─────────────────────────────────┘
                 △
                 │
                 │
┌─────────────────────────────────┐
│   Evaluation while LR Parsing   │
└─────────────────────────────────┘      Implementation
```
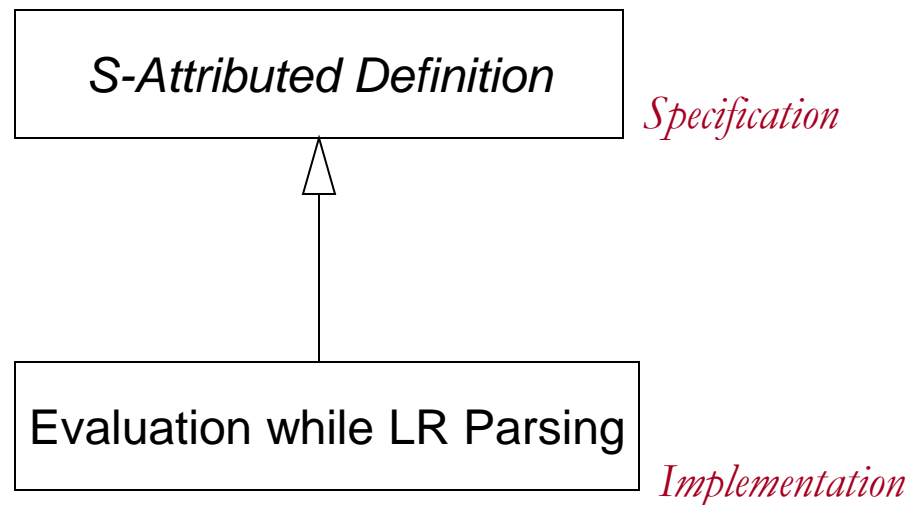
The Previous Calculator Example

No.	Productions	Semantic Rules
1	$L \rightarrow E$ **n**	$L.val = E.val$ print($L.val$)
2	$E \rightarrow E_1$ **+** T	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	$E.val = T.val$
4	$T_1 \rightarrow T_2$ ***** F	$T_1.val = T_2.val * F.val$
5	$T \rightarrow F$	$T.val = F.val$
6	$F \rightarrow$ **(** E **)**	$F.val = E.val$
7	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Associate attributes of grammar symbols with positions in the parsing stack.

Implementation in LR Parsing

No.	Productions	Code	Notes
1	$L \rightarrow E$ **n**	stack[ntop].val = stack[top − 1].val; print(stack[ntop].val);	
2	$E \rightarrow E_1$ **+** T	stack[ntop].val = stack[top − 2].val + stack[top].val;	stack[top − 1].val = '**+**'
3	$E \rightarrow T$		
4	$T_1 \rightarrow T_2$ ***** F	stack[ntop].val = stack[top − 2].val * stack[top].val;	stack[top − 1].val = '*****'
5	$T \rightarrow F$		
6	$F \rightarrow$ **(** E **)**	stack[ntop].val = stack[top − 1].val;	stack[top].val = '**)**' stack[top − 2].val = '**(**'
7	$F \rightarrow$ **digit**		

Setup an attribute stack with the same height as parsing (state) stack:
ntop = top − | right-side | + 1

Evaluation While LR Parsing

Step	States (Illustrative)	Attributes	Input	Code	Output
1	$	$	3 * 5 + 4 n $		
2	$ 3	$ 3	* 5 + 4 n $	--	F → **digit**
3	$ F	$ 3	* 5 + 4 n $	--	T → F
4	$ T	$ 3	* 5 + 4 n $		
5	$ T *	$ 3 *	5 + 4 n $		
6	$ T * 5	$ 3 * 5	+ 4 n $	--	F → **digit**
7	$ T * F	$ 3 * 5	+ 4 n $	3 * 5	T → T ***** F
8	$ T	$ 15	+ 4 n $	--	E → T
9	$ E	$ 15	+ 4 n $		
10	$ E +	$ 15 +	4 n $		
11	$ E + 4	$ 15 + 4	n $	--	F → **digit**
12	$ E + F	$ 15 + 4	n $	--	T → F
13	$ E + T	$ 15 + 4	n $	15 * 4	E → E **+** T
14	$ E	$ 19	n $		
15	$ E **n**	$ 19 **n**	$	print(19)	L → E **n**
16	$ L	$ 19	$	accept	

5. L-Attributed Definitions and Translation Schemes

○ A syntax-directed definition is L-attributed if each inherited attribute depends only on attributes of its **left** siblings or **inherited** attributes of its parent.

- Synthesized attributes are supported.

- Can NOT depend on any synthesized attributes of its parent! (Why?)

Depth-First Evaluation Order

- The evaluation order of an L-attributed definition
  - The same as depth-first visiting of the parse tree.
  - Also the same order as top-down parsing.

```
void dfvisit(n: Node) {
    for (each child m of n, from left to right) {
        evaluate inherited attributes of m;
        dfvisit(m)
    }
    evaluate synthesized attributes of n
}
```

Translation Schemes

- Translation scheme vs. L-attributed definition
  - **Explicit** evaluation order in a translation scheme.
  - Perform semantic actions in a left-to-right depth-first order.

L-Attributed Definition   *Specification*

Translation Scheme   *Implementation*
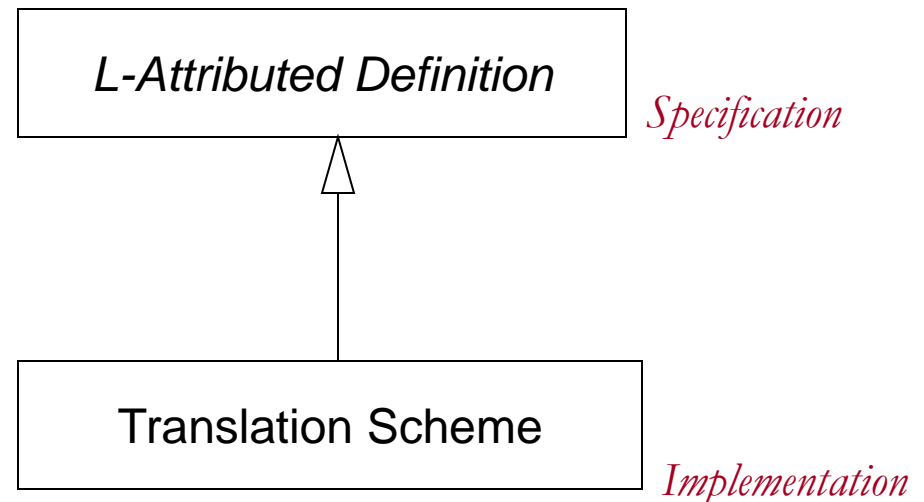
Translation Scheme: Example 1

- **Postfix translation scheme** for an L-attributed definition:

L → E **n**	{ print(E.val); }
E → E$_1$ **+** T	{ E.val = E$_1$.val + T.val; }
E → T	{ E.val = T.val; }
T → T$_1$ ***** F	{ T.val = T$_1$.val * F.val; }
T → F	{ T.val = F.val; }
F → **(** E **)**	{ F.val = E.val; }
F → **digit**	{ F.val = **digit**.lexval; }

Translation Scheme: Example 2

○ Translation scheme for LR parsing:

$L \rightarrow E$ **n**	{ print(stack[top − 1].val);
	top = top − 1; }
$E \rightarrow E_1$ **+** T	{ stack[top − 2].val = stack[top − 2].val + stack[top].val;
	top = top − 2; }
$E \rightarrow T$	
$T \rightarrow T_1$ ***** F	{ stack[top − 2].val = stack[top − 2].val * stack[top].val;
	top = top − 2; }
$T \rightarrow F$	
$F \rightarrow$ **(** E **)**	{ stack[top − 2].val = stack[top − 1].val;
	top = top − 2; }
$F \rightarrow$ **digit**	

Translation Scheme: Example 3

○ Translation scheme for transformation from infix to postfix expressions:

  ● Actions inside productions.

E	$\rightarrow$	T R	
R	$\rightarrow$	**addop** T	{ print(**addop**.lexeme); }
		R_1	
R	$\rightarrow$	ε	
T	$\rightarrow$	**num**	{ print(**num**.val); }

Translation Scheme: Example 3 (cont')

```
                            E
                ┌───────────┴───────────┐
                T                        R
            ┌───┊───┐          ┌───┬─────┊──────────────┐
            9  { print('9'); }  –   T   { print('–'); }  R
                              ┌───┊───┐        ┌───┬────┊─────────┐
                              5  { print('5'); } +   T  { print('+'); }  R
                                              ┌─┊─┐                   │
                                              2 { print('2'); }        ε
```
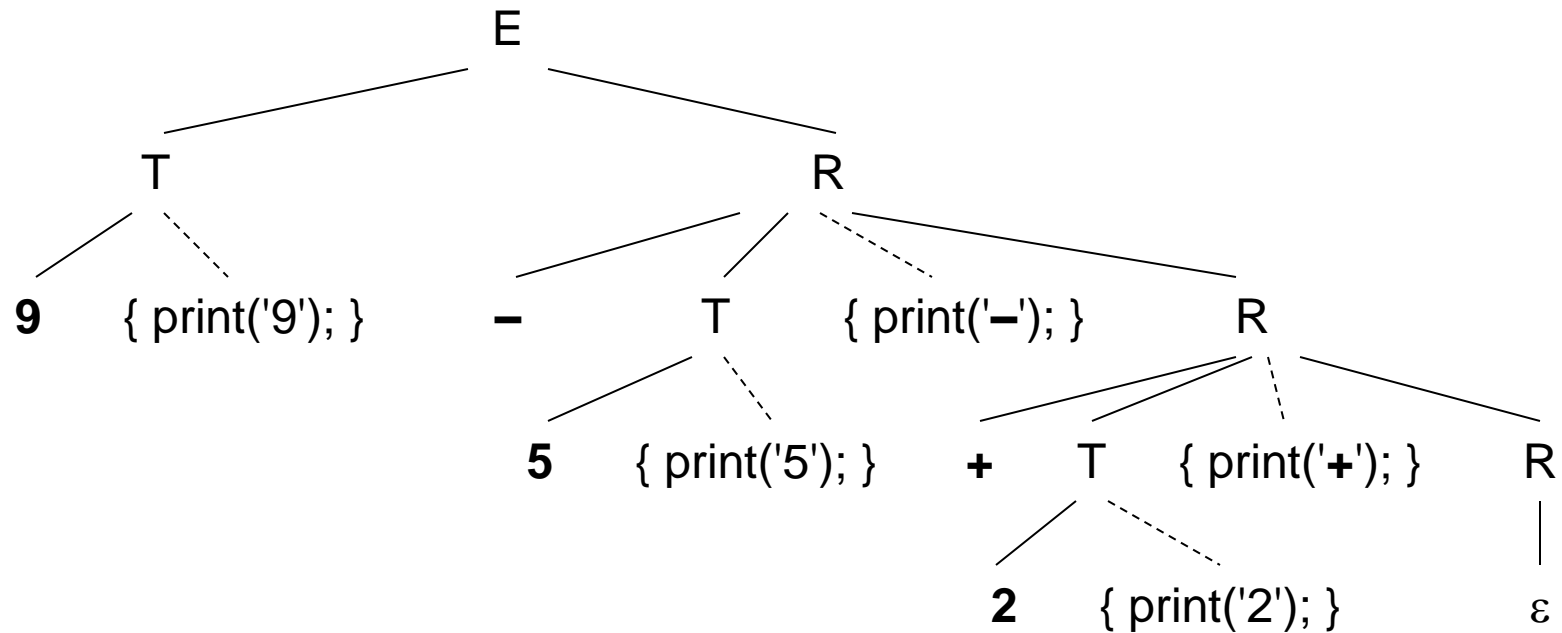
Parse Tree with Actions

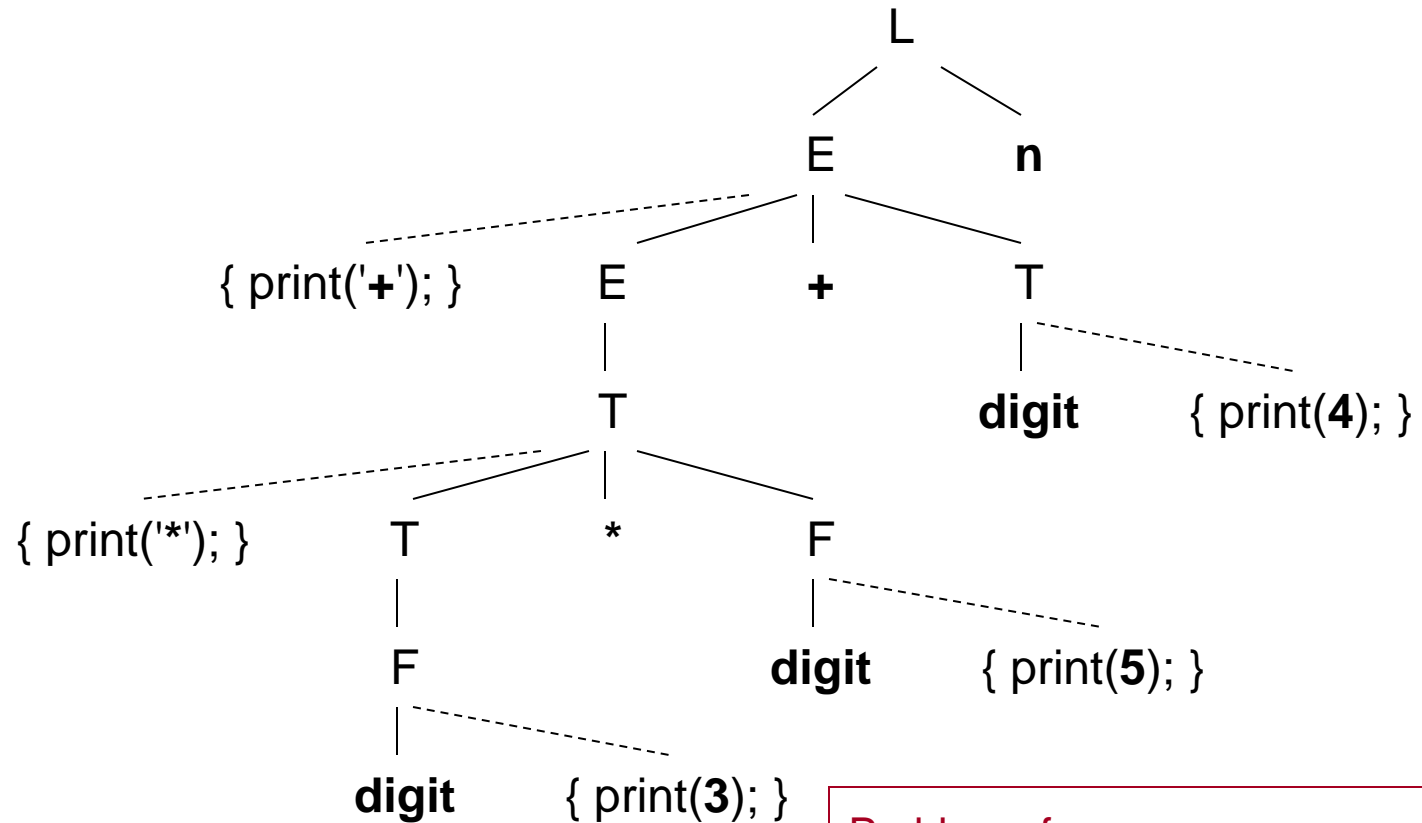# Translation Scheme: Example 4

- Problematic translation scheme for prefix expressions:
  - In both top-down and bottom-up parsing.

L	→	E **n**	
E	→		{ print('+'); }
		E_1 **+** T	
E	→	T	
T	→		{ print('*'); }
		T_1 * F	
T	→	F	
F	→	**(** E **)**	
F	→	**digit**	{ print(**digit**.lexval); }

Translation Scheme: Example 4 (cont')

```
                              L
                             / \
                            E   n
              _____ /|_____
  { print('+'); }        E  +           T
                         |               |_____
                         T            digit   { print(4); }
          _____  /|_____
  { print('*'); }     T  *           F
                      |               |_____
                      F            digit   { print(5); }
           _____|
        digit   { print(3); }
```

From L-Attributed Definitions to Translation Schemes

○ Three transformation rules

1. Inherited attributes of A must be calculated before A.

2. An action must not refer to a synthesized attribute of a symbol to the right of the action.

3. A synthesized attribute must be computed after all attributes it references have been computed.
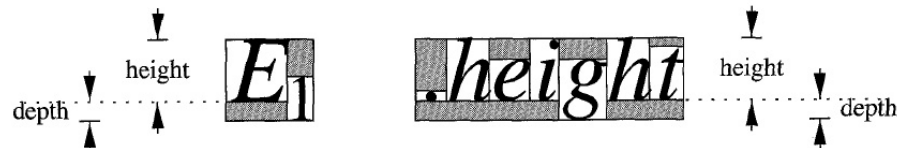
Does not satisfy the requirements:

S	→	$A_1 A_2$	{ A_1.in = 1; A_2.in = 2; }
A	→	**a**	{ print(A.in); }

Typesetting Boxes: **EQN** and **T$_E$X**

No.	Productions	Semantic Actions
1	$S \rightarrow B$	$B.ps = 10$
2	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = max(B_1.ht, B_2.ht)$ $B.dp = max(B_1.dp, B_2.dp)$
3	$B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = B.ps \times 70\%$ $B.ht = max(B_1.ht, B_2.ht - B.ps \times 25\%)$ $B.dp = max(B_1.dp, B_2.dp + B.ps \times 25\%)$
4	$B \rightarrow$ **(** B_1 **)**	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5	$B \rightarrow$ **text**	$B.ht = getHight(B.ps, \textbf{text}.lexval)$ $B.dp = getDepth(B.ps, \textbf{text}.lexval)$

ps = point size
ht = height
dp = depth

Higher precedence, right associativity

From L-Attributed Definition to Translation Scheme

S	→	{ B.ps = 10; }
	B	
B	→	{ B_1.ps = B.ps; }
	B_1	{ B_2.ps = B.ps; }
	B_2	{ B.ht = max(B_1.ht, B_2.ht);
		B.dp = max(B_1.dp, B_2.dp); }
B	→	{ B_1.ps = B.ps; }
	B_1 **sub**	{ B_2.ps = B.ps $\times$ 70%; }
	B_2	{ B.ht = max(B_1.ht, B_2.ht − B.ps $\times$ 25%);
		B.dp = max(B_1.dp, B_2.dp + B.ps $\times$ 25%); }
B	→	**(** { B_1.ps = B.ps; }
	B_1 **)**	{ B.ht = B_1.ht;
		B.dp = B_1.dp; }
B	→	**text** { B.ht = getHight(B.ps, **text**.lexval);
		B.dp = getDepth(B.ps, **text**.lexval); }

6. L-Attributed Definitions in Predictive Parsing

- Development steps

    May have
    left-recursions

    - Write a possibly LL(1) grammar for syntax rules.

    - Define an L-attributed definition by appending semantic rules.

    - Transform the L-attributed definition to a translation scheme.

    - Eliminate **left-recursion** in the translation scheme.

    - Write a recursive descent predictive parser (translator).

Eliminating Left-Recursion: A Simple Example

○ A motivating example: a simple case
  - Trick: treating actions as terminals if they do not calculate any attributes.

$E \rightarrow E_1$ **+** T { print('+'); }
$E \rightarrow T$

$E \rightarrow T R$
$R \rightarrow$ **+** T { print('+'); } R
$R \rightarrow \varepsilon$

Eliminating Left-Recursion: More Examples

○ A more complex case

$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val; }
$E \rightarrow E_1 - T$	{ E.val = E_1.val – T.val; }
$E \rightarrow T$	{ E.val = T.val; }
$T \rightarrow (E)$	{ T.val = E.val; }
$T \rightarrow num$	{ T.val = num.val; }

$E \rightarrow T$	{ R.i = T.val; }
	R	{ E.val = R.s; }
$R \rightarrow + T$	{ R_1.i = R.i + T.val; }
	R_1	{ R.s = R_1.s; }
$R \rightarrow - T$	{ R_1.i = R.i – T.val; }
	R_1	{ R.s = R_1.s; }
$R \rightarrow \varepsilon$	{ R.s = R.i; }
$T \rightarrow (E)$	{ T.val = E.val; }
$T \rightarrow num$	{ T.val = num.val; }
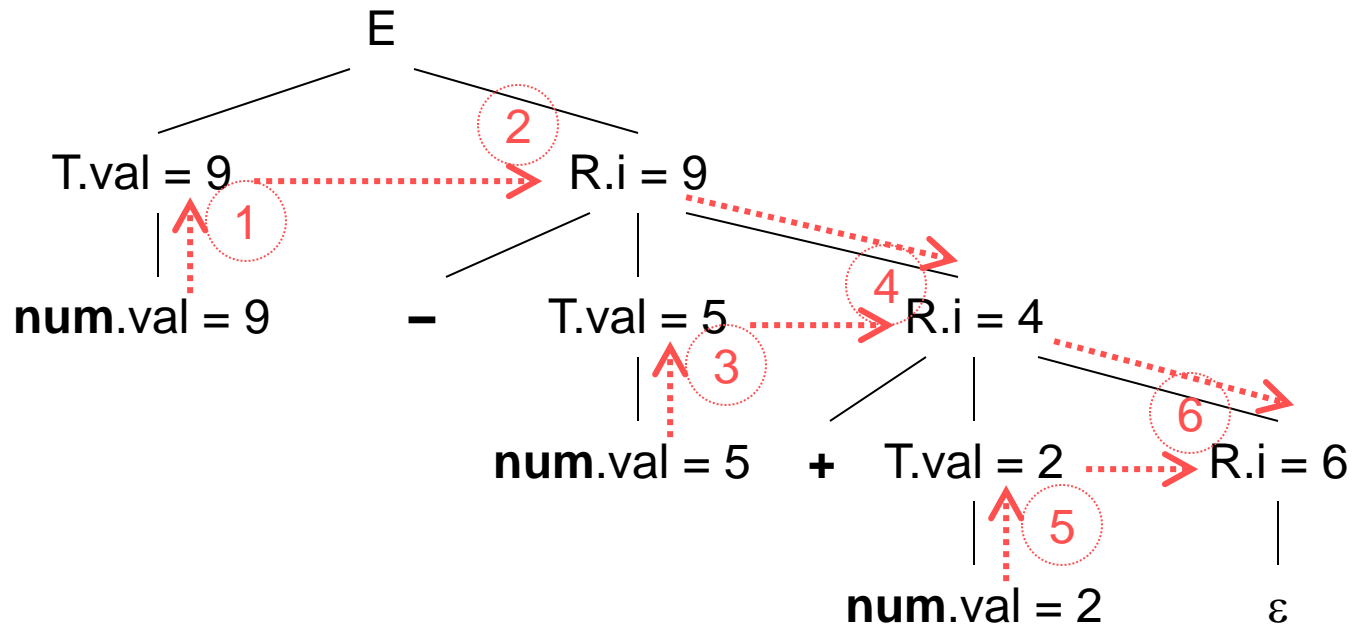
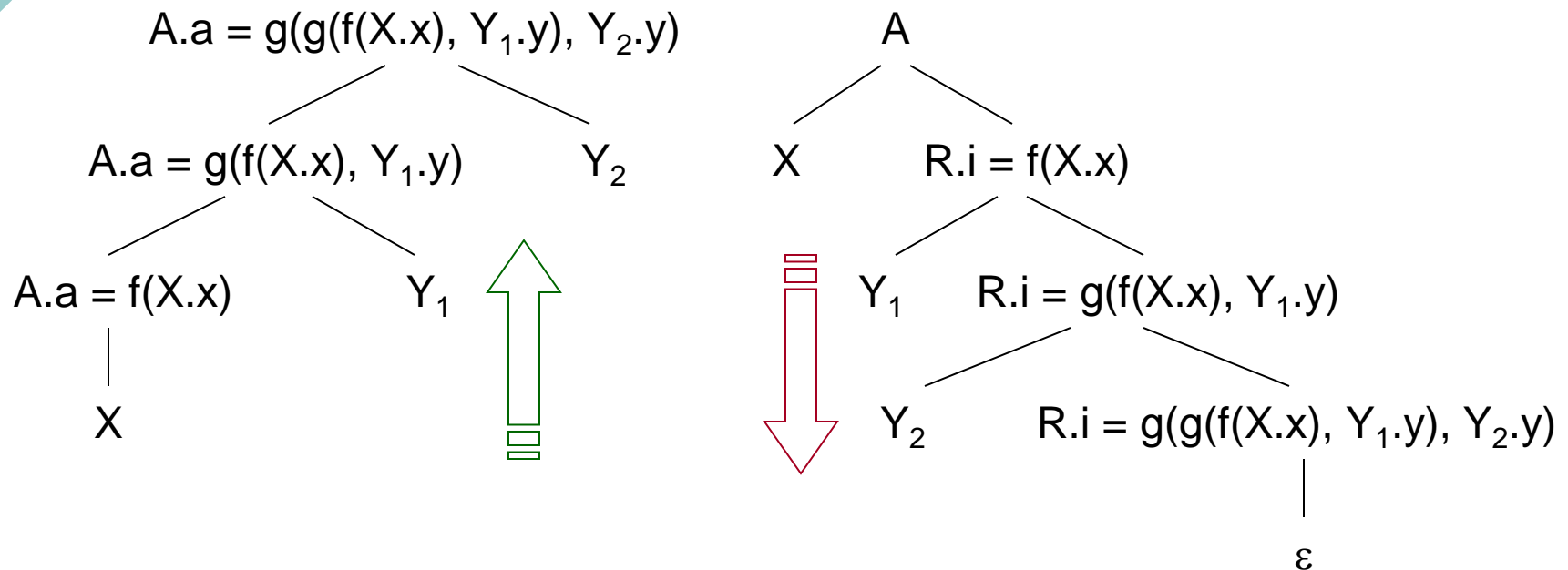Evaluation Order of Input Expression **9 − 5 + 2**

Eliminating Left-Recursion

- General rules
  - Only available for S-attributed definitions (postfix translation schemes).

A   →   A₁ Y { A.a = g(A₁.a, Y.y); }
A   →   X { A.a = f(X.x); }

$$A \rightarrow X \{ R.i = f(X.x); \} R \{ A.a = R.s; \}$$
$$R \rightarrow Y \{ R_1.i = g(R.i, Y.y); \} R_1 \{ R.s = R_1.s; \}$$
$$R \rightarrow \varepsilon \{ R.s = R.i; \}$$

A.a = g(g(f(X.x), Y₁.y), Y₂.y)   A

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

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

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

ε

Writing a Predictive Parser

○ Review: writing a recursive descent predictive parser (only for parsing)

- Each grammar symbol corresponds to a (recursive) subprogram.

- The start symbol corresponds to the main entry subprogram.

- In each subprogram, branching actions with regard to the lookahead.

Writing a Predictive Parser: An Example

○ For parsing only

$$R \rightarrow \textbf{addop } T \quad \{ R_1.i = mknode(\textbf{addop}.lexeme, R.i, T.nptr); \}$$
$$R_1 \quad \{ R.s = R_1.s; \}$$
$$R \rightarrow \varepsilon \quad \{ R.s = R.i; \}$$

```
void R() {
    if (lookahead == addop) {
        match(addop);
        T();
        R()
    } else {
        // do nothing
    }
}
```

Writing a Predictive Translator

- Writing a translator with semantic actions
  - Each inherited attribute corresponds to a formal parameter.
  - All synthesized attributes correspond to the return value
    - Multiple synthesized attributes may be merged in a single record.
  - Each attribute of the child nodes corresponds to a local variable.
  - Process the right side of the production
    - Terminals: match()
    - Nonterminals: procedure call
    - Actions: direct execution (copy)

From a Parser to a Translator

$R \rightarrow$ **addop** T   { R_1.i = mknode(**addop**.lexeme, R.i, T.nptr); }
        R_1 { R.s = R_1.s; }
$R \rightarrow \varepsilon$ { R.s = R.i; }

```
SyntaxTreeNode R(SyntaxTreeNode i) {
    SyntaxTreeNode s;  // synthesized attributes
    SyntaxTreeNode t_nptr, r1_i, r1_s;  // for children
    char addopLexeme;  // temporary

    if (lookahead == addop) {
        addopLexeme = lookahead.lexval;
        match(addop);
        t_nptr = T();
        r1_i = mknode(addopLexeme, i, t_nptr);
        r1_s = R(r1_i);
        s = r1_s;
    } else {
        s = i;
    }
    return s;
}
```

7. L-Attributed Definitions in LR Parsing

○ S-attributed definitions are easy to be evaluated by LR parsing.

● See section 4 in these slides.

○ What are the challenges for evaluating L-attributed definitions in LR parsing?

1. Not all actions are on the right-most of a production body (postfix translation scheme).

2. Inherited attributes are not stored in the parsing stack.

Make Use of Tricks

- Using **markers** to move all embedded actions to the right-most of a production body.

- **Tracing** inherited attributes in the parsing stack.

  1. The simplest case: locating inherited attributes which are calculated by **copy rules**.

  2. Introduce **markers** to help to locate inherited attributes in the stack.

  3. Also make use of new **markers** to locate inherited attributes which are not calculated by copy rules.

Move Embedded Actions to Right-Most

○ A marker is an ε-production that acts as a place-holder.

```
E   →   T R
R   →   + T { print('+'); } R
    |   – T { print('–'); } R
    |   ε
T   →   num { print(num.val); }
```

```
E   →   T R
R   →   + T M R
    |   – T N R
    |   ε
T   →   num              { print(num.val); }
M   →   ε                { print('+'); }
N   →   ε                { print('–'); }
```

Trace a Copied Inherited Attribute

- Calculations of inherited attributes are the biggest source of embedded actions.

$$
\begin{array}{lll}
D & \rightarrow & T \qquad\qquad \{ \text{L.inh = T.type; } \} \\
& & L \\
T & \rightarrow & \textbf{int} \qquad\quad\ \{ \text{T.type = INTEGER; } \} \\
T & \rightarrow & \textbf{real} \qquad\ \{ \text{T.type = REAL; } \} \\
L & \rightarrow & \qquad\qquad \{ L_1\text{.inh = L.inh; } \} \\
& & L_1 \textbf{, id} \qquad \{ \text{addType}(\textbf{id}.\text{entry, L.inh}); \} \\
L & \rightarrow & \textbf{id} \qquad\qquad \{ \text{addType}(\textbf{id}.\text{entry, L.inh}); \}
\end{array}
$$

Trace a Rewriting Inherited Attribute (cont')

D	→	T	{ L.inh = T.type; }
		L	
T	→	**int**	{ T.type = INTEGER; }
T	→	**real**	{ T.type = REAL; }
L	→		{ L_1.inh = L.inh; }
		L_1 **, id**	{ addType(**id**.entry, L.inh); }
L	→	**id**	{ addType(**id**.entry, L.inh); }

Productions	Code
D → T L	
T → **int**	stack[ntop].val = INTEGER;
T → **real**	stack[ntop].val = REAL;
L → L_1 **, id**	addType(stack[top].val, stack[top − 3].val);
L → **id**	addType(stack[top].val, stack[top − 1].val);

Predict Positions of Inherited Attributes

○ Introduce new markers

S	→	**a** A	{ C.i = A.s; }
		C	
S	→	**b** A B	{ C.i = A.s; }
		C	
C	→	**c**	{ C.s = g(C.i); }

S	→	**a** A	{ C.i = A.s; }
		C	
S	→	**b** A B	{ M.i = A.s; }
		M	{ C.i = M.s; }
		C	
C	→	**c**	{ C.s = g(C.i); }
M	→	ε	{ M.s = M.i; }

Store Calculated Inherited Attributes

- Also make use of new markers

$$S \rightarrow \mathbf{a}\,A \qquad \{\ C.i = f(A.s);\ \}$$
$$\phantom{S \rightarrow \mathbf{a}\,}C$$
$$C \rightarrow \mathbf{c} \qquad \{\ C.s = g(C.i);\ \}$$

$$S \rightarrow \mathbf{a}\,A \qquad \{\ M.i = A.s;\ \}$$
$$\phantom{S \rightarrow \mathbf{a}\,}M \qquad \{\ C.i = M.s;\ \}$$
$$\phantom{S \rightarrow \mathbf{a}\,}C$$
$$C \rightarrow \mathbf{c} \qquad \{\ C.s = g(C.i);\ \}$$
$$M \rightarrow \varepsilon \qquad \{\ M.s = f(M.i);\ \}$$

A Practical Example

Productions	Semantic Actions
S → K B	B.ps = K.s
K → ε	K.s = 10
B → B₁ L B₂	B_1.ps = B.ps L.i = B.ps B_2.ps = L.s B.ht = max(B_1.ht, B_2.ht) B.dp = max(B_1.dp, B_2.dp)
L → ε	L.s = L.i
B → B₁ **sub** M B₂	B_1.ps = B.ps M.i = B.ps B_2.ps = M.s B.ht = max(B_1.ht, B_2.ht − B.ps × 25%) B.dp = max(B_1.dp, B_2.dp + B.ps × 25%)
M → ε	M.s = M.i × 70%
B → **(** N B₁ **)**	B_1.ps = B.ps N.i = B.ps B.ht = B_1.ht B.dp = B_1.dp
N → ε	N.s = N.i
B → **text**	B.ht = getHight(B.ps, **text**.lexval) B.dp = getDepth(B.ps, **text**.lexval)

Anytime when B is reduced, **B.ps** is immediately under B

A Practical Example (cont')

Productions	Code
S → K B	
K → ε	stack[ntop].ps = 10;
B → B₁ L B₂	stack[ntop].ht = max(stack[top − 2].ht, stack[top].ht); stack[ntop].dp = max(stack[top − 2].dp, stack[top].dp);
L → ε	stack[ntop].ps = stack[top − 1].ps;
B → B₁ **sub** M B₂	stack[ntop].ht = max(stack[top − 3].ht, stack[top].ht − stack[top − 4].ps × 25%); stack[ntop].dp = max(stack[top − 3].dp, stack[top].dp + stack[top − 4].ps × 25%);
M → ε	stack[ntop].ps = stack[top − 2].ps × 70%
B → **(** N B₁ **)**	stack[ntop].ht = stack[top − 1].ht stack[ntop].dp = stack[top − 1].dp
N → ε	stack[ntop].ps = stack[top − 1].ps;
B → **text**	stack[ntop].ht = getHight(stack[top − 1].ps, stack[top].lexval); stack[ntop].dp = getDepth(stack[top − 1].ps, stack[top].lexval);

ps is treated as a synthesized attribute

Remove all calculations for inherited attributes

Exercise 8.1

○ Given the translation scheme for the EQN language (see pp.40 in this lecture), calculate the height and depth of the input: **text sub text sub text**.

- Suppose that for each **text**,
  - ○ getHeight(ps, **text**.lexval) = 8 * ps
  - ○ getDepth(ps, **text**.lexval) = 0

Further Reading

- Dragon Book, 2nd Edition (DBv2)
  - Comprehensive Reading:
    - Section 5.1-5.3 on introduction to syntax-directed translation.
    - Section 5.4-5.5 on the implementation of translation schemes in top-down and bottom-up LR parsing.
  - Skip Reading:
    - DBv1 Section 5.8-5.10 on advanced implementation of translation schemes.

Enjoy the Course!