



第二重要

# Principles of Compiler Construction

---

SH definition不重要，拓扑排序不重要

L-attribute = translation scheme 重要  
translation scheme就是SDT，关于一些详细的概念不是很清楚，需要再复习

yac 自底向上 能力更强 性能没有因此减弱

**Prof. Wen-jun LI**

School of Computer Science and Engineering

lnslwj@mail.sysu.edu.cn

语义分析和中间代码生成是同时实现的，因为这两部分没有形式化理论支撑，同时使用的工具都是编程工具

语义分析与中间代码生成一起讲，因为原理是一样的。

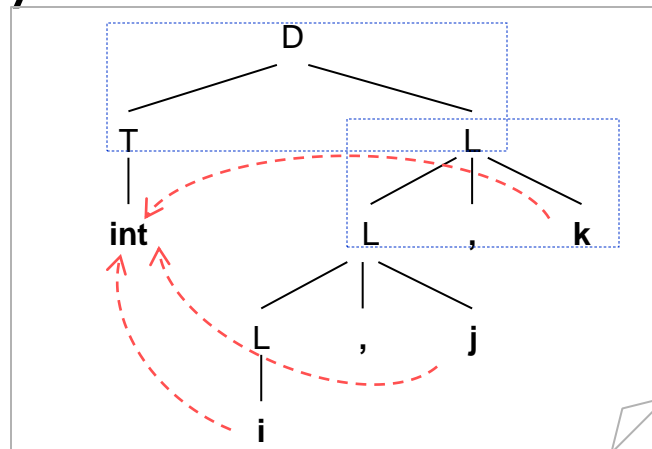
## Lecture 8. Syntax-Directed Translation

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

核心

# 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?



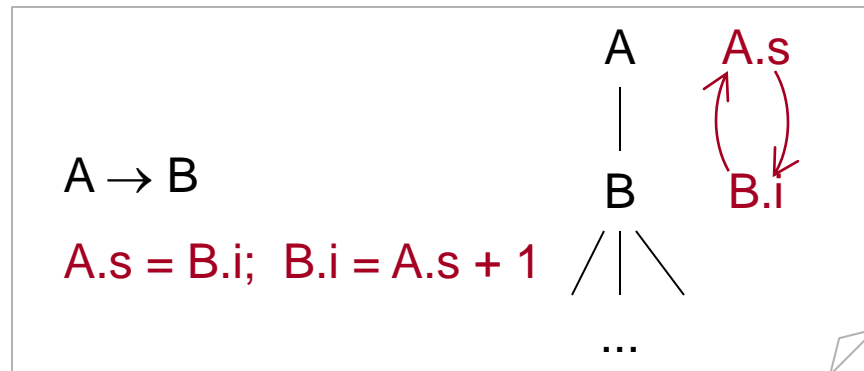
# Applications of Syntax-Directed Translation

---

- 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

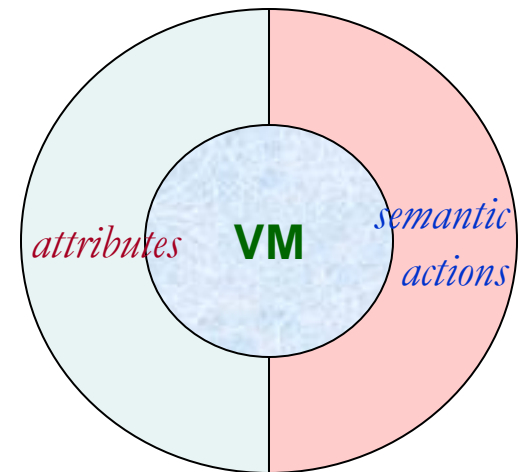
- Associate grammar symbols with attributes
  - Based on specific applications.
  - The **data** to be manipulated.
- Associate productions with semantic rules
  - Also named semantic actions.
  - The **operations** that manipulate the attributes.



# 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.





# 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

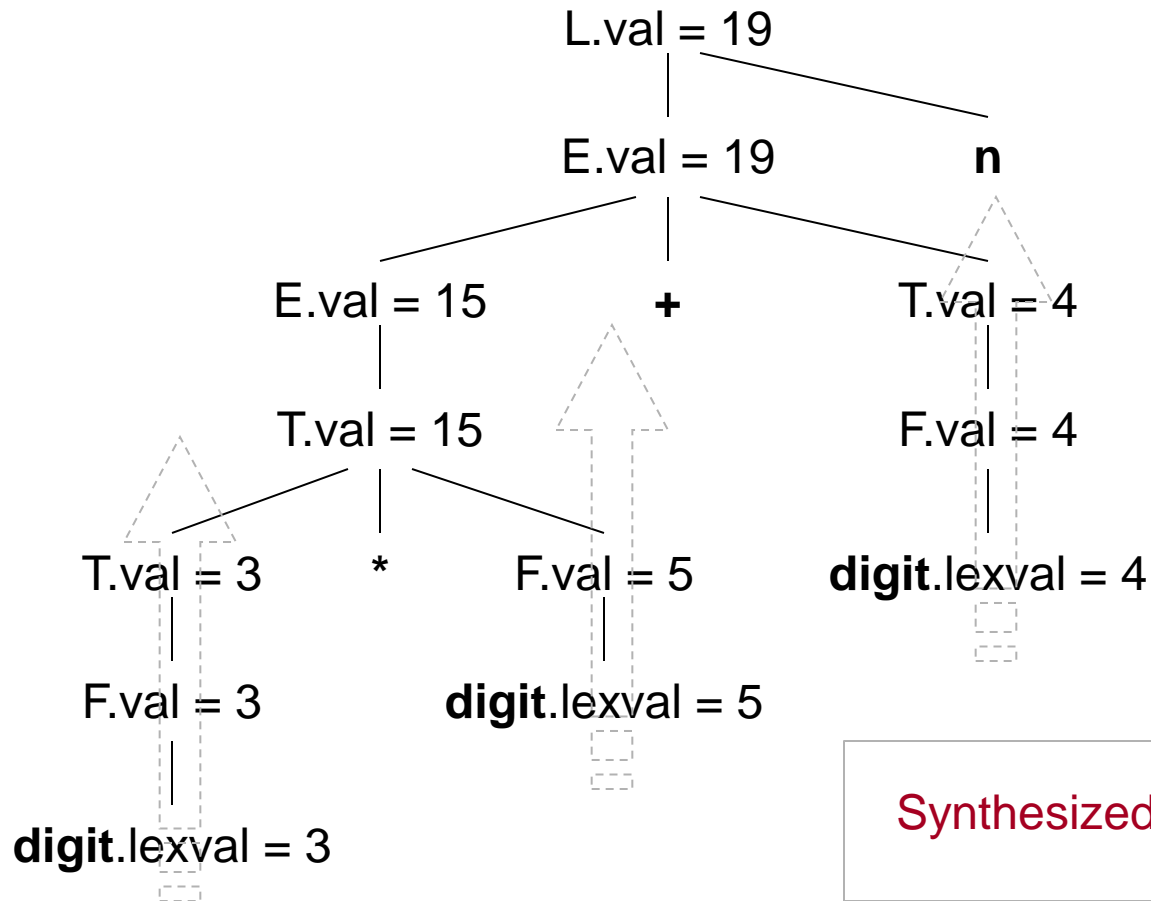
	Productions	Semantic Rules
1	$L \rightarrow E \mathbf{n}$ <span style="color: red;">回车</span>	$L.val = E.val$ $\text{print}(L.val)$ <span style="border: 1px solid red; padding: 2px;">Side-effects</span>
2	$E \rightarrow E_1 + T$ <span style="color: red;">下标是后面引用了两个不同的值, 用于区分</span>	$E.val = E_1.val + T.val$
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	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

Different  
subfix style

# Annotated Parse Tree

*variables*

*constants*



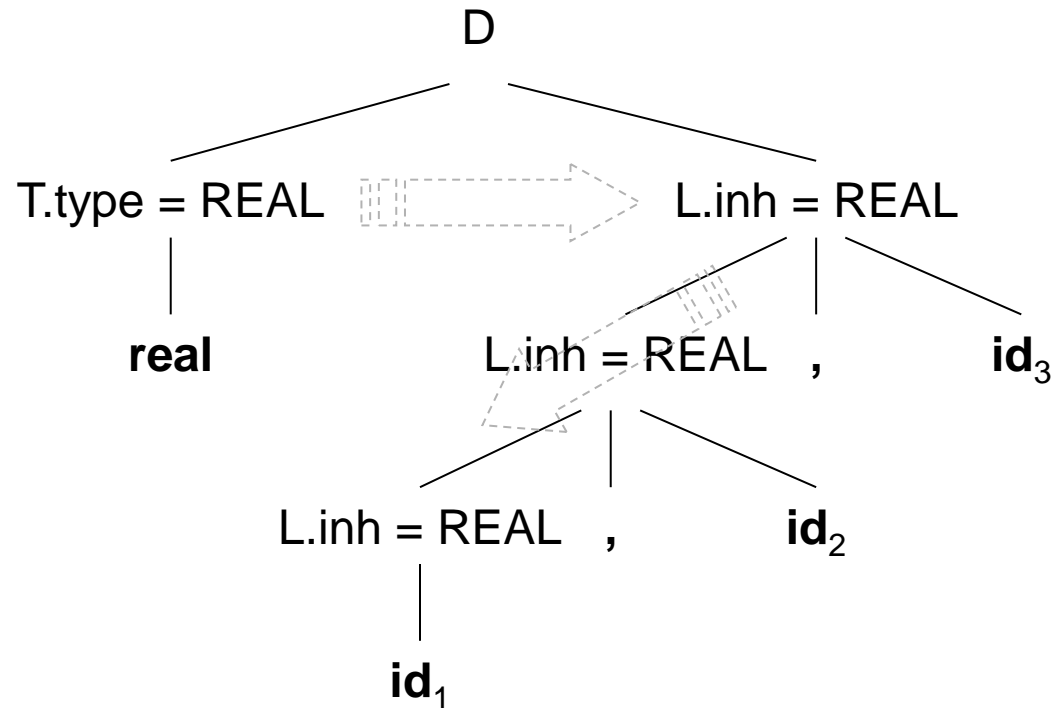
Synthesized attributes

# Inherited Attribute

左部 右部  
综合属性 继承属性

No.	Productions	Semantic Rules
1	$D \rightarrow T L$	$L.inh = T.type$
2	$T \rightarrow \mathbf{int}$	$T.type = \mathbf{INTEGER}$
3	$T \rightarrow \mathbf{real}$	$T.type = \mathbf{REAL}$
4	$L \rightarrow L_1, \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

# Annotated Parse Tree





## 2. Syntax-Directed Definition: Examples

---

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

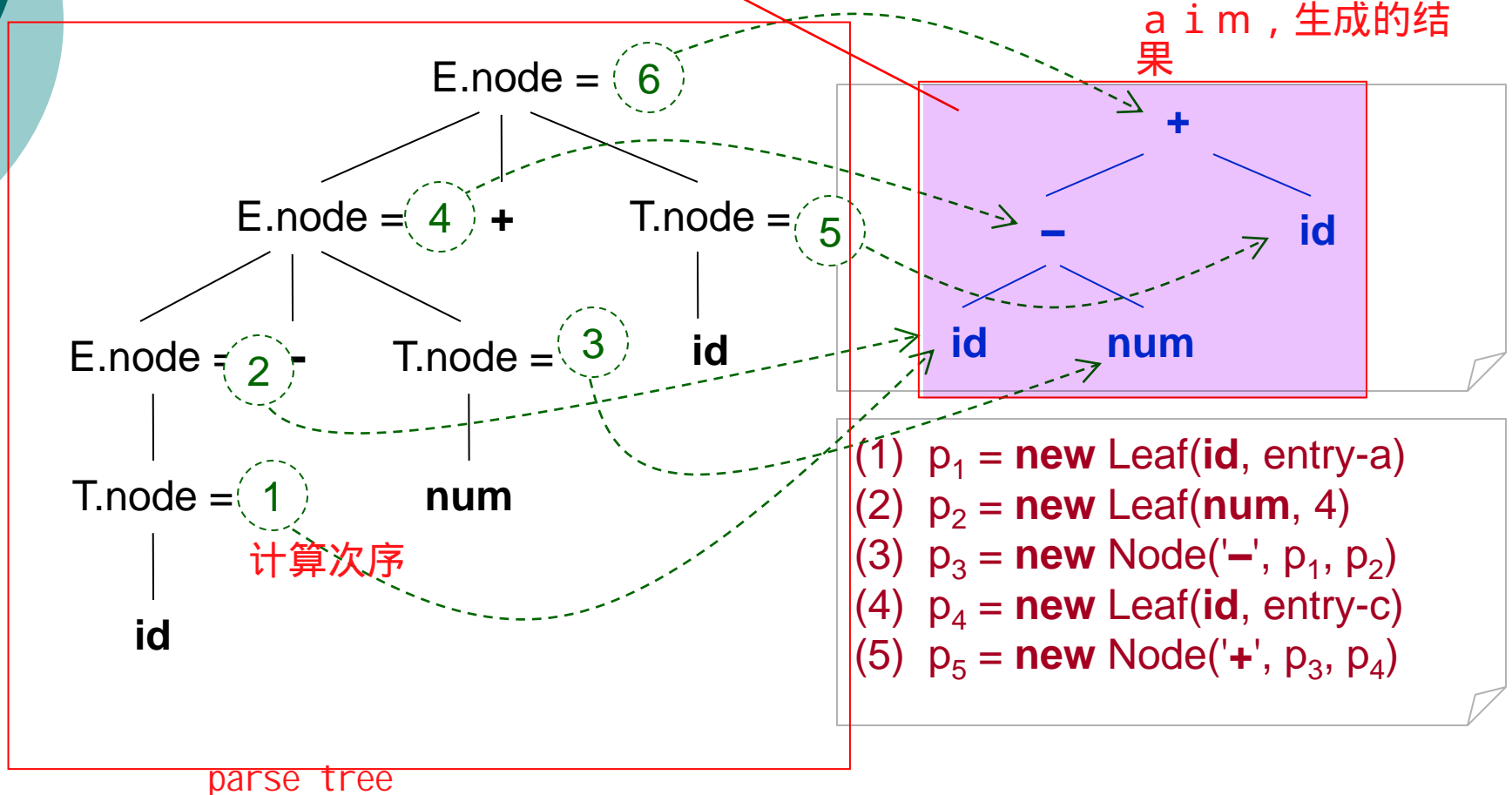
# Syntax Trees

SDD 语法制导定义

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

# Syntax Trees (cont')

## ○ Syntax tree for **a - 4 + c**



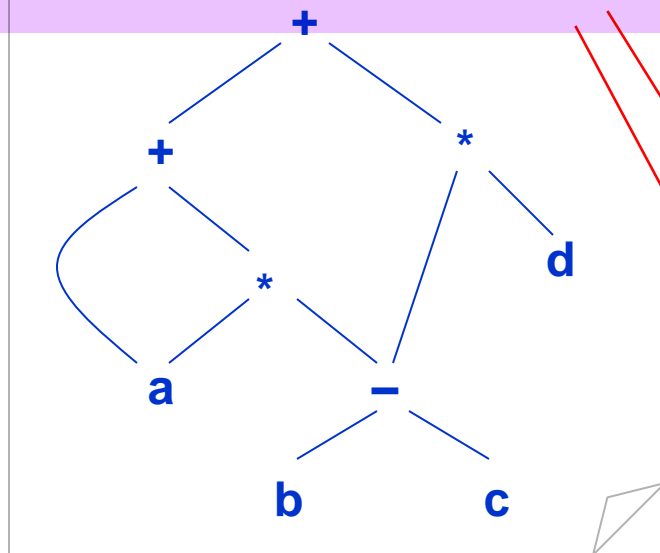
# DAG

有向无环图

## ○ DAG: Directed Acyclic Graph

●  $a + a * (b - c) + (b - c) * d$

使用现成节点，不再重复新建。从而形成DAG，  
好处是自动合并了重复的子表达式



```
(1) p1 = new Leaf(id, entry-a)
(2) → p2 = new Leaf(id, entry-a)
(3) p3 = new Leaf(id, entry-b)
(4) p4 = new Leaf(id, entry-c)
(5) p5 = new Node('-', p3, p4)
(6) p6 = new Node('*', p2, p5)
(7) p7 = new Node('+', p1, p6)
(8) → p8 = new Leaf(id, entry-b)
(9) → p9 = new Leaf(id, entry-c)
(10) → p10 = new Node('-', p8, p9)
(11) p11 = new Leaf(id, entry-d)
(12) p12 = new Node('*', p10, p11)
(13) p13 = new Node('+', p7, p12)
```

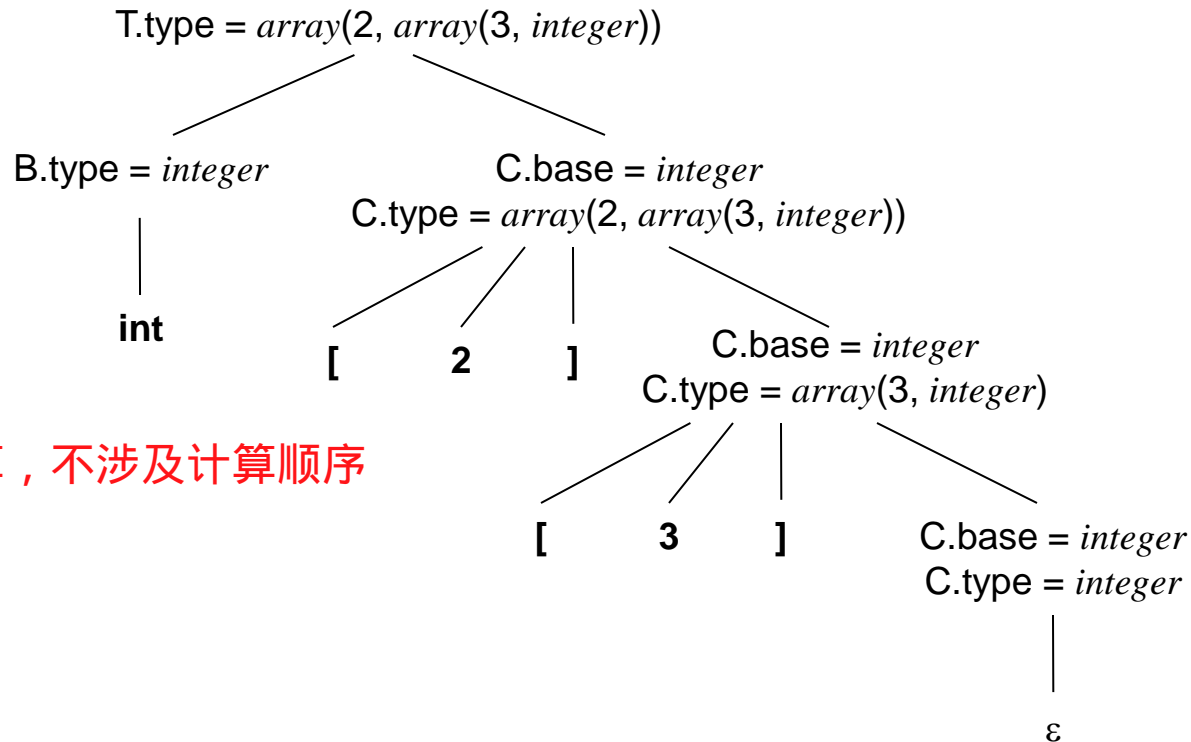


# Type Structures

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

# Type Structures (cont')

- Type structure for **int[2][3]**



SDD是说明怎么算，不涉及计算顺序

# 3. Evaluation Order and Dependency Graphs

general but useless

---

- 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

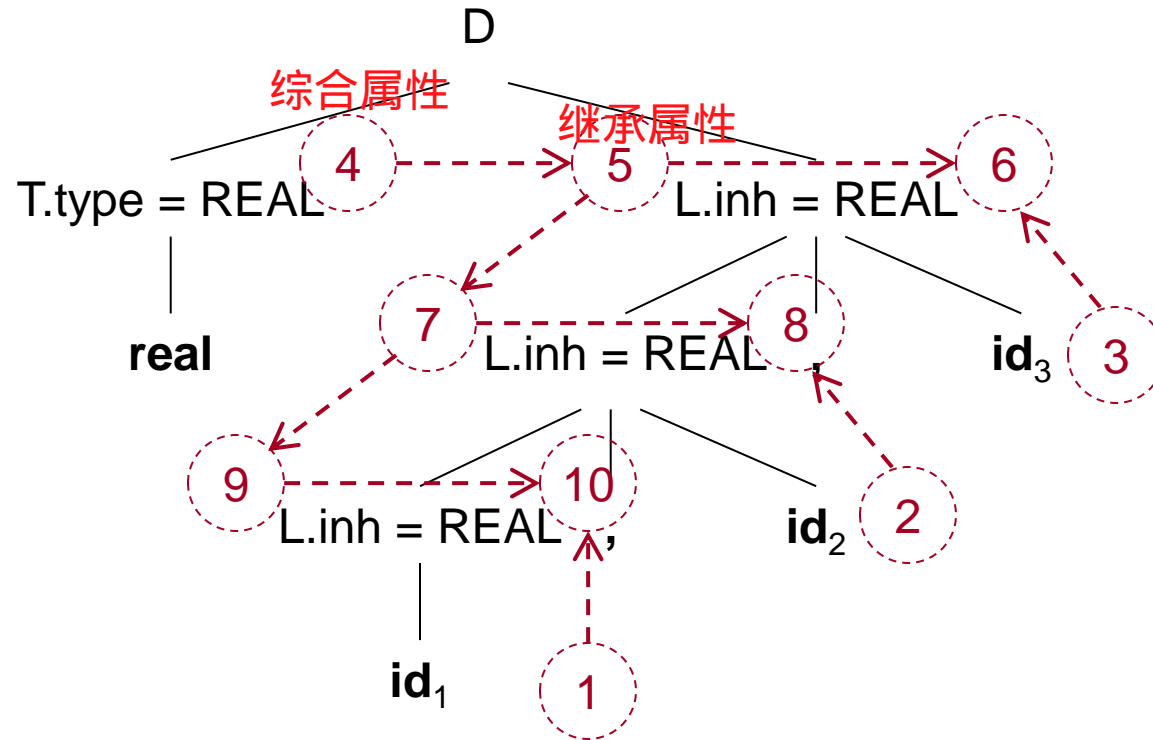
- Explicit parse tree.
- Low efficiency and impractical.

## The Previous Syntax-Directed Definition

No.	Productions	Semantic Rules
1	$D \rightarrow T L$	$L.inh = T.type$
2	$T \rightarrow \mathbf{int}$	$T.type = \text{INTEGER}$
3	$T \rightarrow \mathbf{real}$	$T.type = \text{REAL}$
4	$L \rightarrow L_1 , \mathbf{id}$	$L_1.inh = L.inh$ $\text{addType}(\mathbf{id}.entry, L.inh)$
5	$L \rightarrow \mathbf{id}$	$\text{addType}(\mathbf{id}.entry, L.inh)$

之后考虑：LR、递归下降预测分析器中如何实现

# Dependency Graph: An Example



adj. 假的, 仿真的

6, 8, 10 are dummy attributes

addType(id.entry, L.inh) 没有计算出值, 但是会产生影响

# Evaluation Order

---

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

# Implicit Parse Trees

- **S-attributed definitions** 都是综合属性，顺序都是 upward
  - Every attribute is synthesized.
  - Guarantee an evaluation order that is the same order of the output of LR parsing.
- **L-attributed definitions** 综合 或 非右的继承属性
  - 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.

SDD不能随便写，可能因为依赖有回路导致不能实现加约束，成为以上两者之一



## 4. S-Attributed Definitions

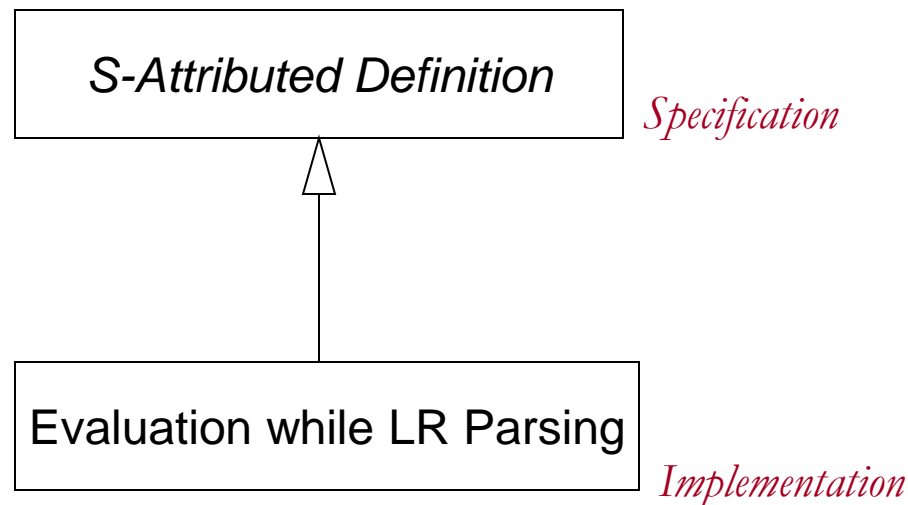
---

- 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



# The Previous Calculator Example

No.	Productions	Semantic Rules
1	$L \rightarrow E \mathbf{n}$	$L.val = E.val$ $\text{print}(L.val)$
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	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

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

# Implementation in LR Parsing

No.	Productions	Code	Notes
1	$L \rightarrow E \mathbf{n}$	<code>stack[ntop].val = stack[top - 1].val; print(stack[ntop].val);</code>	
2	$E \rightarrow E_1 + T$	<code>stack[ntop].val = stack[top - 2].val + stack[top].val;</code>	<code>stack[top - 1].val = '+'</code>
3	$E \rightarrow T$	<b>n e w t o p</b>	
4	$T_1 \rightarrow T_2 * F$	<code>stack[ntop].val = stack[top - 2].val * stack[top].val;</code>	<code>stack[top - 1].val = '*'</code>
5	$T \rightarrow F$		
6	$F \rightarrow ( E )$	<code>stack[ntop].val = stack[top - 1].val;</code>	<code>stack[top].val = ')'  stack[top - 2].val = '('</code>
7	$F \rightarrow \mathbf{digit}$		

Setup an attribute stack with the same height as parsing (state) stack:  
 **$ntop = top - | \text{right-side} | + 1$**

# Evaluation While LR Parsing

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

---

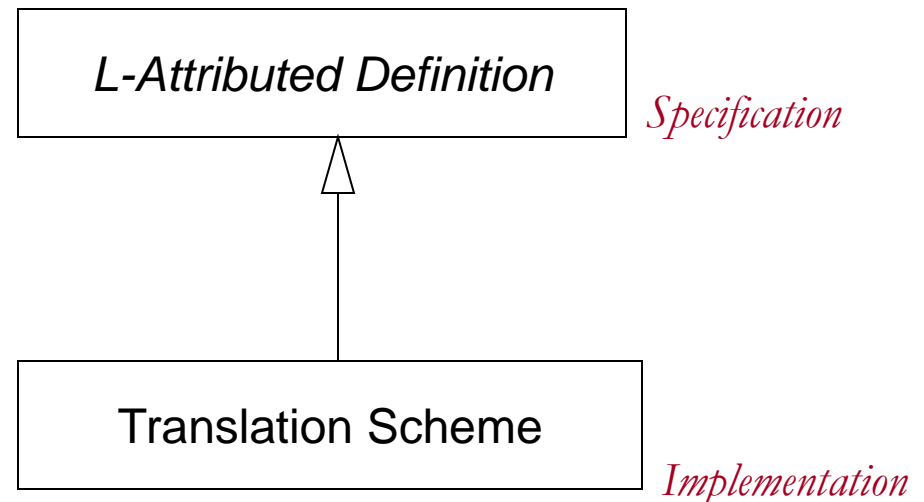
- The evaluation order of an L-attributed definition
  - The same as depth-first visiting of the parse tree.
  - Also the same order as top-down parsing.

```
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.





# Translation Scheme: Example 1

翻译模式与语法制导翻译SDD的区别：用花括号写，里面的可以动作可以插入表达式，从而说明计算顺序

- **Postfix translation scheme** for an L-attributed definition: > 都在最右/后

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

# Translation Scheme: Example 2

## ○ Translation scheme for LR parsing:

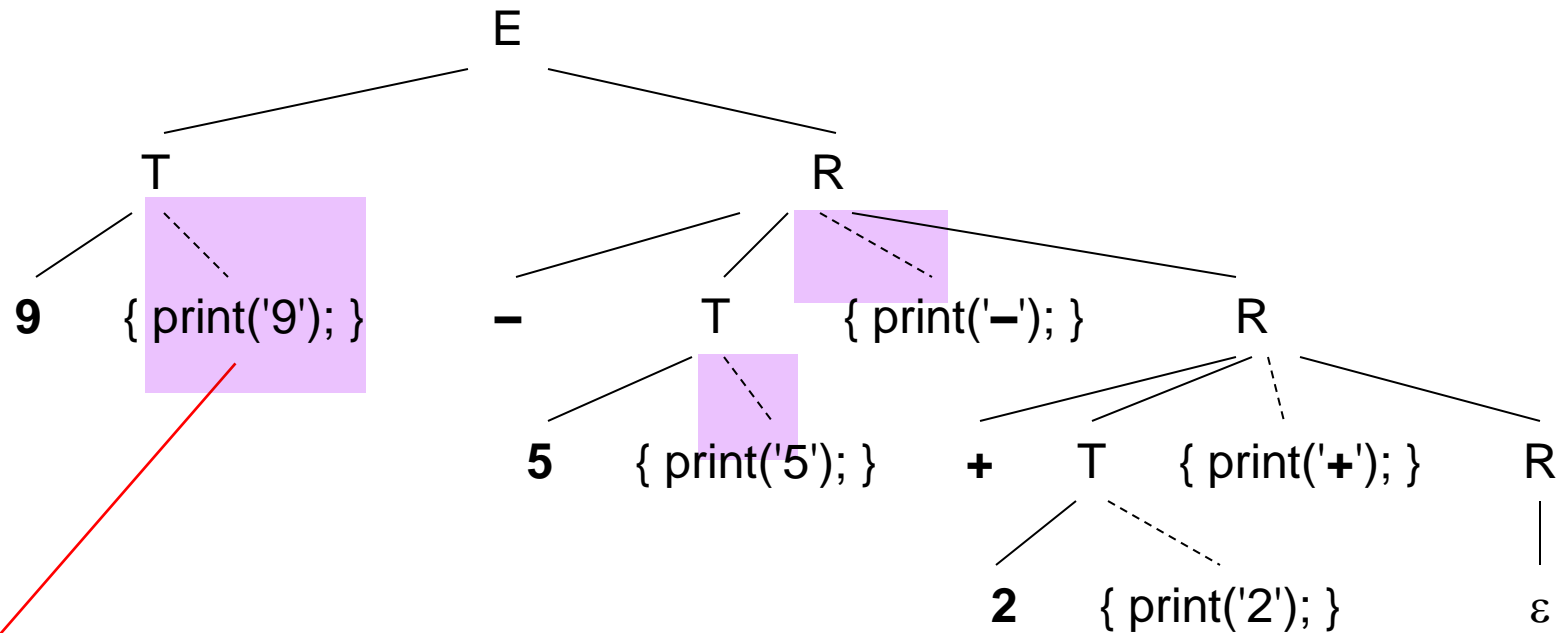
$L \rightarrow E n$	<pre>{ print(stack[top - 1].val);   top = top - 1; }</pre>
$E \rightarrow E_1 + T$	<pre>{ stack[top - 2].val = stack[top - 2].val + stack[top].val;   top = top - 2; }</pre>
$E \rightarrow T$	
$T \rightarrow T_1 * F$	<pre>{ stack[top - 2].val = stack[top - 2].val * stack[top].val;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

# Translation Scheme: Example 3

- Translation scheme for transformation from infix to postfix expressions:
  - Actions inside productions.

E	→	T R	
R	→	<b>addop</b> T	{ print( <b>addop.lexeme</b> ); }
		R <sub>1</sub>	还是R, 为了区分多加了1
R	→	ε	
T	→	<b>num</b>	{ print( <b>num.val</b> ); }

# Translation Scheme: Example 3 (cont')



嵌入的原因：深度优先，从左到右遍历的时候发挥作用，与执行顺序保持一致。

## Parse Tree with Actions

动作要用虚线、动作用括号括起来，动作过长的时候标上序号，树上挂序号

# Translation Scheme: Example 4

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

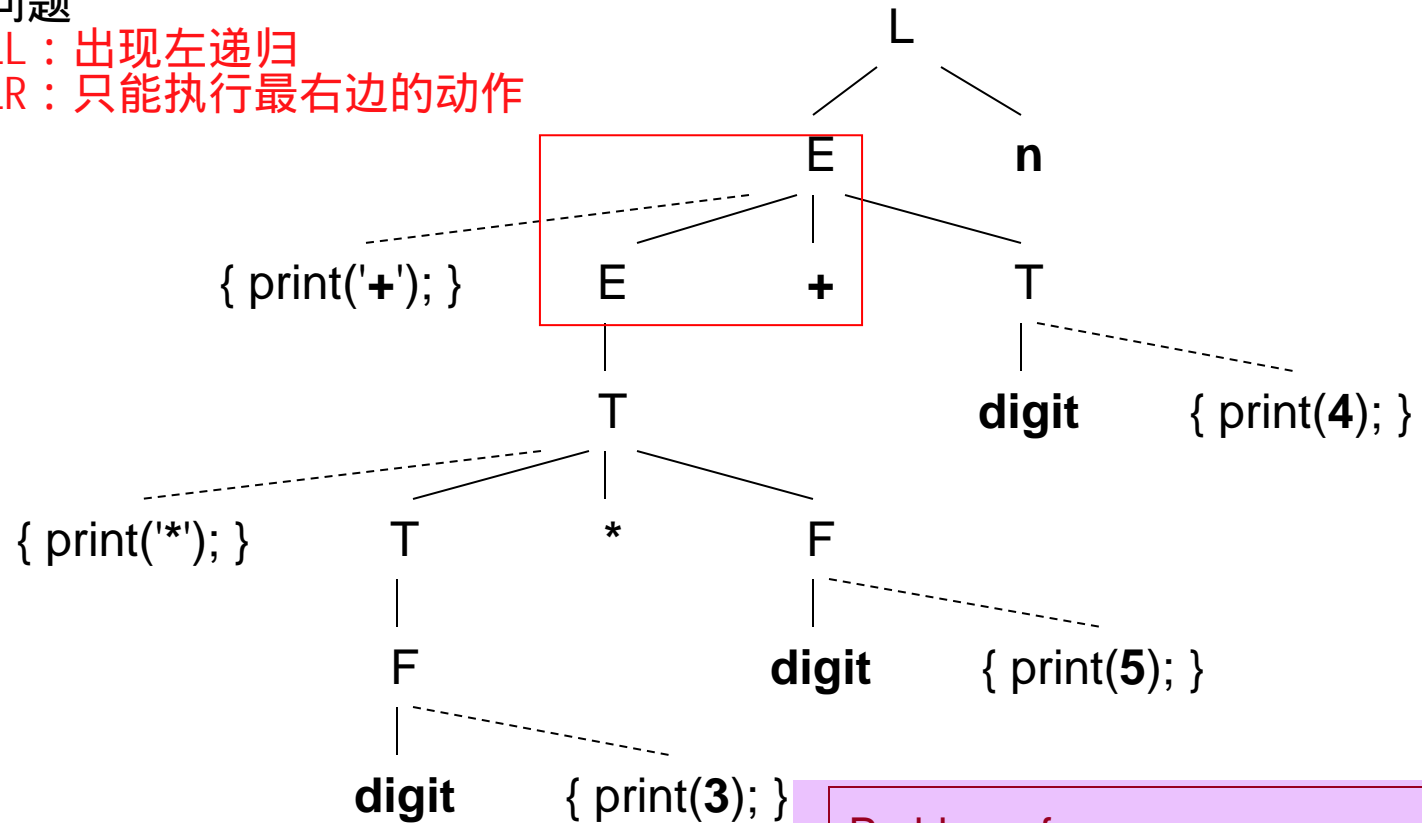
L	→	E n	
E	→		{ print('+'); }
		E <sub>1</sub> + T	→
E	→	T	
T	→		{ print('*'); }
		T <sub>1</sub> * F	
T	→	F	
F	→	( E )	
F	→	digit	{ print( <b>digit</b> .lexval); }

# Translation Scheme: Example 4 (cont')

问题

LL : 出现左递归

LR : 只能执行最右边的动作



Problems for  
top-down parsing: left-recursion  
bottom-up parsing: embedded actions

真正能起作用

# From L-Attributed Definitions to Translation Schemes

→只能从左兄弟或者父节点继承

## ○ 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.

右边的属性都不能碰

Does not satisfy the requirements:

```
S → A1 A2 { A1.in = 1; A2.in = 2; }  
A → a { print(A.in); }
```

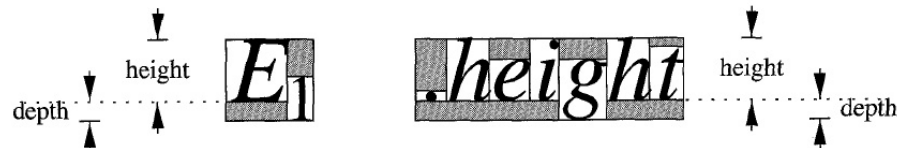
错误，需要改动位置

# Typesetting Boxes: EQN and T<sub>E</sub>X

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

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 <sub>1</sub> .ps = B.ps; }	
	B <sub>1</sub>	{ B <sub>2</sub> .ps = B.ps; }	
	B <sub>2</sub>	{ B.ht = max(B <sub>1</sub> .ht, B <sub>2</sub> .ht); B.dp = max(B <sub>1</sub> .dp, B <sub>2</sub> .dp); }	
B	→	{ B <sub>1</sub> .ps = B.ps; }	
	B <sub>1</sub> <b>sub</b>	{ B <sub>2</sub> .ps = B.ps × 70%; }	
	B <sub>2</sub>	{ B.ht = max(B <sub>1</sub> .ht, B <sub>2</sub> .ht – B.ps × 25%); B.dp = max(B <sub>1</sub> .dp, B <sub>2</sub> .dp + B.ps × 25%); }	
B	→ (	{ B <sub>1</sub> .ps = B.ps; }	
	B <sub>1</sub> )	{ B.ht = B <sub>1</sub> .ht; B.dp = B <sub>1</sub> .dp; }	综合属性放最后
B	→ <b>text</b>	{ B.ht = getHight(B.ps, <b>text</b> .lexval); B.dp = getDepth(B.ps, <b>text</b> .lexval); }	

## 6. L-Attributed Definitions in Predictive Parsing

### ○ Development steps

May have  
left-recursions

- Write a possibly LL(1) grammar for syntax rules. 可能含有左递归
- 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). 递归下降预测分析器，实际上不仅是parser了，而是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.

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

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

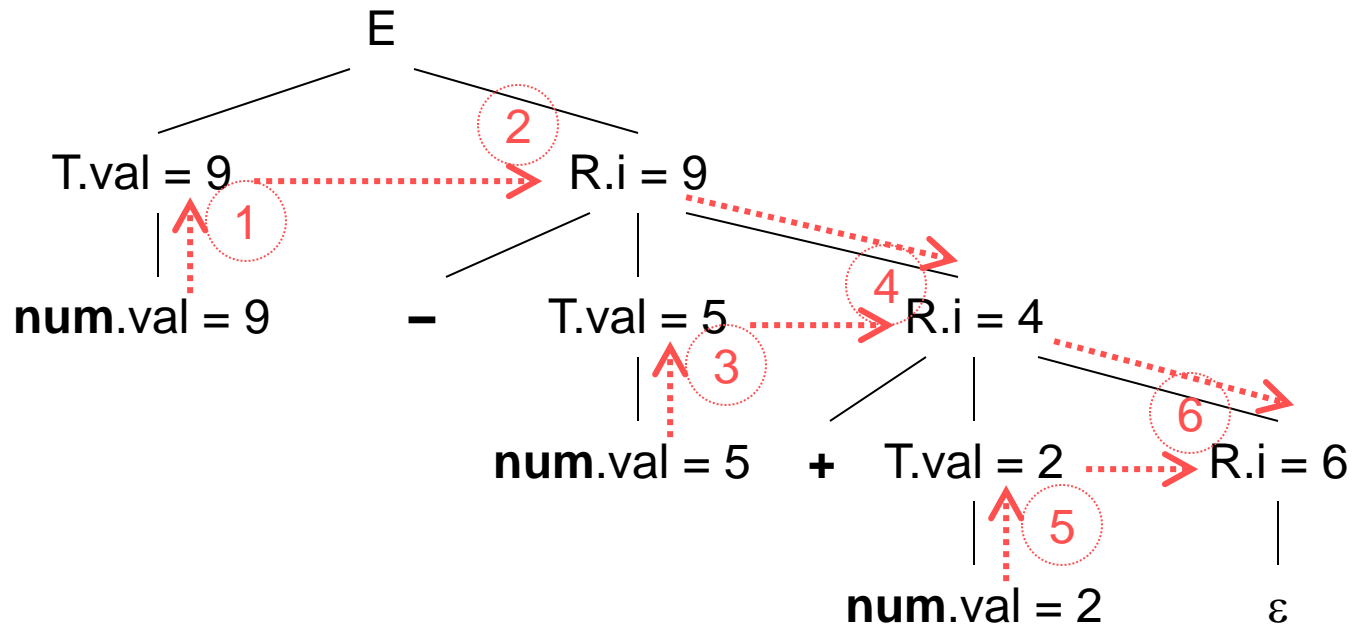
# Eliminating Left-Recursion: More Examples

## ○ A more complex case

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

$E \rightarrow T$	$\{ R.i = T.val; \}$
$R$	$\{ E.val = R.s; \}$
$R \rightarrow + T$	$\{ R_1.i = R.i + T.val; \}$
$R_1$	$\{ R.s = R_1.s; \}$
$R \rightarrow - T$	$\{ R_1.i = R.i - T.val; \}$
$R_1$	$\{ R.s = R_1.s; \}$
$R \rightarrow \varepsilon$	$\{ R.s = R.i; \}$
$T \rightarrow ( E )$	$\{ T.val = E.val; \}$
$T \rightarrow \text{num}$	$\{ T.val = \text{num.val}; \}$

# Eliminating Left-Recursion: More Examples (cont')



Evaluation Order of Input Expression  $9 - 5 + 2$

# Eliminating Left-Recursion

---

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

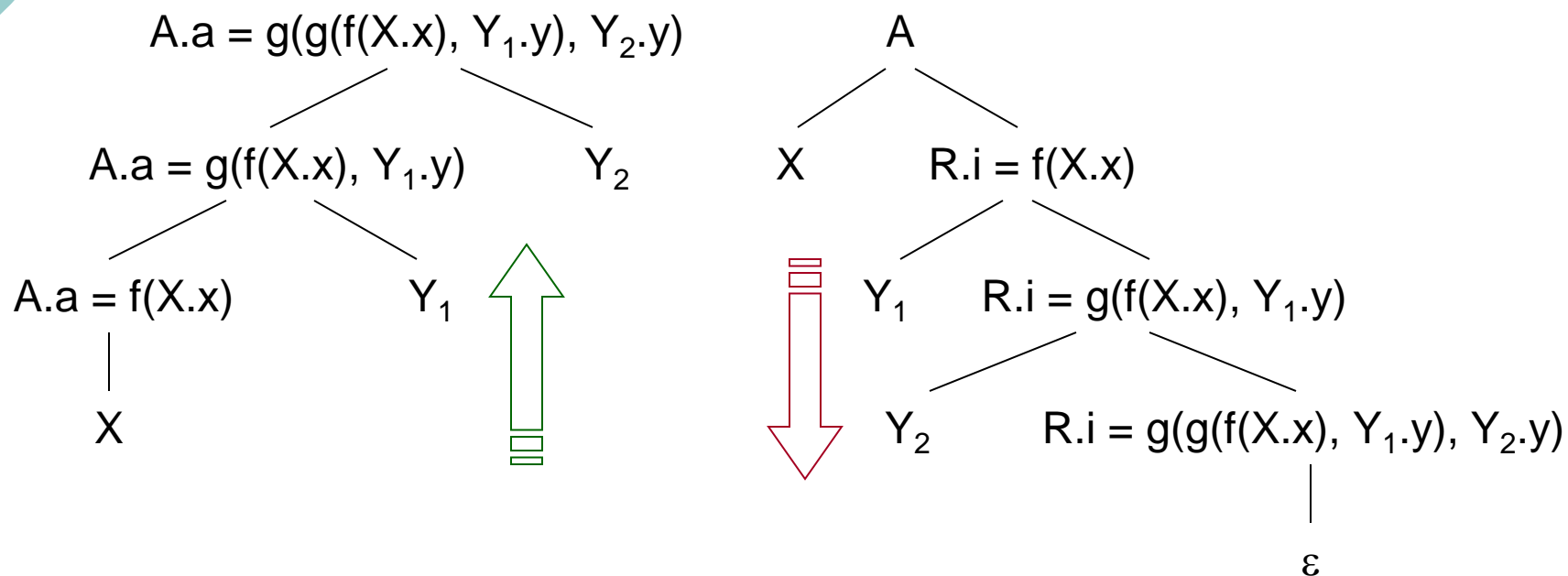
# 通式

$$A \rightarrow A_1 Y \{ A.a = g(A_1.a, Y.y); \}$$

$$A \rightarrow X \{ A.a = f(X.x); \}$$

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

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

$$R \rightarrow \varepsilon \{ R.s = R.i; \}$$


# 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.

原来的做法

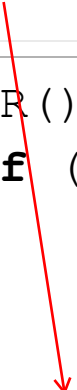


# Writing a Predictive Parser: An Example

## ○ For parsing only

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

```
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	→	<b>addop</b>	T	{ R <sub>1</sub> .i = mknode( <b>addop</b> .lexeme, R.i, T.nptr); }
		R <sub>1</sub>		{ R.s = R <sub>1</sub> .s; }
R	→	ε		{ 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;  
}
```

局部变量

## 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  $\varepsilon$ -production that acts as a place-holder.

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

处理不在最右边动作的方法

```
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.

```
D → T           { L.inh = T.type; }
      L
T → int         { T.type = INTEGER; }
T → real        { T.type = REAL; }
L →             { L1.inh = L.inh; }
      L1 , id    { addType(id.entry, L.inh); }
L → id          { addType(id.entry, L.inh); }
```

# Trace a Rewriting Inherited Attribute (cont')

```

D → T      { L.inh = T.type; }
    L
T → int     { T.type = INTEGER; }
T → real    { T.type = REAL; }
L →         { L1.inh = L.inh; }
    L1, id { addType(id.entry, L.inh); }
L → id      { addType(id.entry, L.inh); }
  
```

Productions	Code
D → T L	
T → <b>int</b>	stack[ntop].val = INTEGER;
T → <b>real</b>	stack[ntop].val = REAL;
L → L <sub>1</sub> , <b>id</b>	addType(stack[top].val, stack[top - 3].val);
L → <b>id</b>	addType(stack[top].val, stack[top - 1].val);

一个栈是状态，一个是综合属性，继承属性通过综合属性追踪



# 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); }
```

保证C的值从栈的正下方一个处来

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

# Store Calculated Inherited Attributes

- Also make use of new markers

```
S → a A      { C.i = f(A.s); }  
      C  
C → c      { C.s = g(C.i); }
```

```
S → a A      { M.i = A.s; }  
      M      { C.i = M.s; } 存储计算结果  
      C  
C → c      { C.s = g(C.i); }  
M → ε      { M.s = f(M.i); }
```

虽然嵌在中间，但是  
只是记录继承属性从哪里来，实际上  
在执行的时候会被删除

通用的解决方式：加marker存储值

# A Practical Example

Productions	Semantic Actions
$S \rightarrow K B$	$B.ps = K.s$
$K \rightarrow \varepsilon$	$K.s = 10$
$B \rightarrow B_1 L B_2$	$B_1.ps = B.ps$ $L.i = B.ps$ $B_2.ps = L.s$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
$L \rightarrow \varepsilon$	$L.s = L.i$
$B \rightarrow B_1 \text{ sub } M B_2$	$B_1.ps = B.ps$ $M.i = B.ps$ $B_2.ps = M.s$ $B.ht = \max(B_1.ht, B_2.ht - B.ps \times 25\%)$ $B.dp = \max(B_1.dp, B_2.dp + B.ps \times 25\%)$
$M \rightarrow \varepsilon$	$M.s = M.i \times 70\%$
$B \rightarrow ( N B_1 )$	$B_1.ps = B.ps$ $N.i = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
$N \rightarrow \varepsilon$	$N.s = N.i$
$B \rightarrow \text{text}$	$B.ht = \text{getHight}(B.ps, \text{text.lexval})$ $B.dp = \text{getDepth}(B.ps, \text{text.lexval})$

Anytime when B is reduced, **B.ps** is immediately under B  
 一般都是这样处理，也就是保证继承属性位于正下方

new top, 应该是此时B1的位置, 也就是top-2; 可以看哈工大PPT

## A Practical Example (cont')

Productions	Code
$S \rightarrow K B$	
$K \rightarrow \varepsilon$	stack[ntop].ps = 10; 直接赋值
$B \rightarrow B_1 L B_2$	stack[ntop].ht = max(stack[top - 2].ht, stack[top].ht); stack[ntop].dp = max(stack[top - 2].dp, stack[top].dp);
$L \rightarrow \varepsilon$	stack[ntop].ps = stack[top - 1].ps;
$B \rightarrow B_1 \text{ sub } M B_2$	stack[ntop].ht = max(stack[top - 3].ht, stack[top].ht - stack[top - 4].ps $\times$ 25%); stack[ntop].dp = max(stack[top - 3].dp, stack[top].dp + stack[top - 4].ps $\times$ 25%);
$M \rightarrow \varepsilon$	stack[ntop].ps = stack[top - 2].ps $\times$ 70%
$B \rightarrow ( N B_1 )$	stack[ntop].ht = stack[top - 1].ht stack[ntop].dp = stack[top - 1].dp
$N \rightarrow \varepsilon$	stack[ntop].ps = stack[top - 1].ps;
$B \rightarrow \text{text}$	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**,
    - $\text{getHeight}(\text{ps}, \text{text.lexval}) = 8 * \text{ps}$
    - $\text{getDepth}(\text{ps}, \text{text.lexval}) = 0$

# Further Reading

---

- Dragon Book, 2<sup>nd</sup> 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!

---

