

语义分析

魏恒峰

hfwei@nju.edu.cn

2020 年 12 月 17 日





Donald Knuth

Semantics of Context-Free Languages

by

DONALD E. KNUTH

California Institute of Technology

ABSTRACT

“Meaning” may be assigned to a string in a context-free language by defining “attributes” of the symbols in a derivation tree for that string. The attributes can be defined by functions associated with each production in the grammar. This paper examines the implications of this process when some of the attributes are “synthesized”, i.e., defined solely in terms of attributes of the *descendants* of the corresponding nonterminal symbol, while other attributes are “inherited”, i.e., defined in terms of attributes of the *ancestors* of the nonterminal symbol. An algorithm is given which detects when such semantic rules could possibly lead to circular definition of some attributes. An example is given of a simple programming language defined with both inherited and synthesized attributes, and the method of definition is compared to other techniques for formal specification of semantics which have appeared in the literature.

属性文法 (Attribute Grammar): 为上下文无关文法赋予语义

关键问题：如何基于上下文无关文法做上下文相关分析？



语法分析树上的有序信息流动



一对概念



两类属性定义



三种实现方式

4

四大应用

表达式求值



类型系统 (语义分析)

抽象语法树

后缀表达式 (中间代码生成)

Definition (语法制导定义 (Syntax-Directed Definition; SDD))

SDD 是一个上下文无关文法和**属性**及**规则**的结合。

每个文法符号都可以关联多个属性

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

每个产生式都可以关联一组规则

Definition (语法制导定义 (Syntax-Directed Definition; SDD))

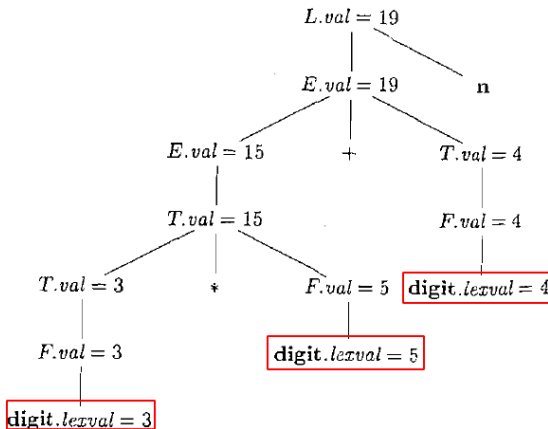
SDD 是一个上下文无关文法和**属性**及**规则**的结合。

SDD **唯一确定**了语法分析树上每个非终结符节点的属性值

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

SDD **没有**规定以什么方式、什么顺序计算这些属性值

注释 (annotated) 语法分析树: 显示了各个属性值的语法分析树



$3 * 5 + 4$

Definition (综合属性 (Synthesized Attribute))

节点 N 上的**综合属性**只能通过 N 的子节点或 N 本身的属性来定义。

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

Definition (综合属性 (Synthesized Attribute))

节点 N 上的**综合属性**只能通过 N 的子节点或 N 本身的属性来定义。

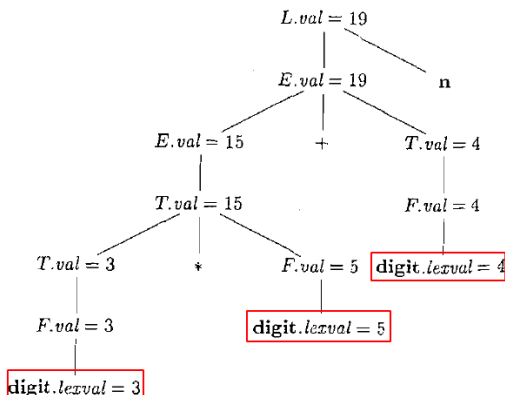
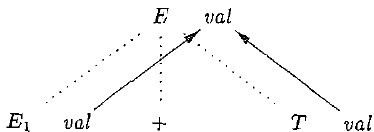
产生式	语义规则
1) $L \rightarrow E n$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow \text{digit}$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

Definition (S 属性定义 (S -Attributed Definition))

如果一个 SDD 的每个属性都是综合属性, 则它是 S 属性定义。

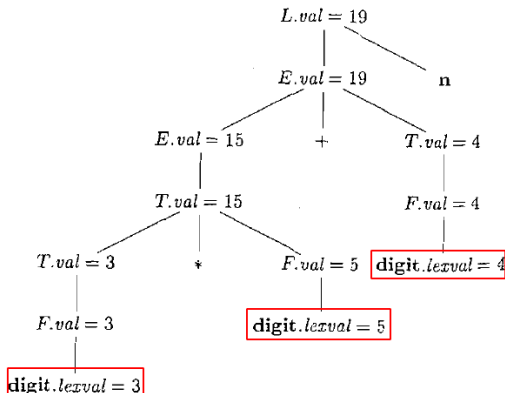
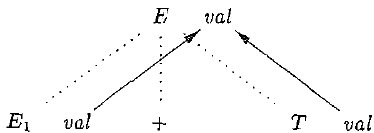
依赖图用于确定一棵给定的语法分析树中各个属性实例之间的依赖关系

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



依赖图用于确定一棵给定的语法分析树中各个属性实例之间的依赖关系

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



S 属性定义的依赖图描述了属性实例之间**自底向上**的信息流

S 属性定义的依赖图描述了属性实例之间自底向上的信息流

因此, 此类属性值的计算可以自然地在自底向上的语法分析过程中实现

S 属性定义的依赖图描述了属性实例之间自底向上的信息流

因此, 此类属性值的计算可以自然地在自底向上的语法分析过程中实现

当 LR 语法分析器进行归约时, 计算相应节点的综合属性值

S 属性定义的依赖图描述了属性实例之间自底向上的信息流

此类属性值的计算也可以在自顶向下的语法分析过程中实现

S 属性定义的依赖图描述了属性实例之间自底向上的信息流

此类属性值的计算也可以在自顶向下的语法分析过程中实现

在 LL 语法分析器中, 递归下降函数 A 返回时,
计算相应节点 A 的综合属性值

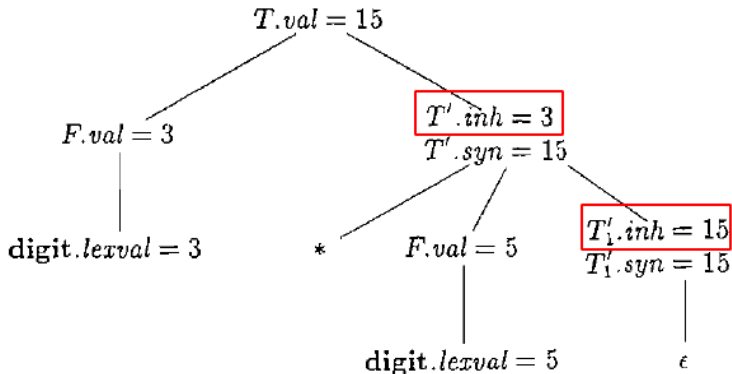
T' 有一个综合属性 syn 与一个继承属性 inh

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

Definition (继承属性 (Inherited Attribute))

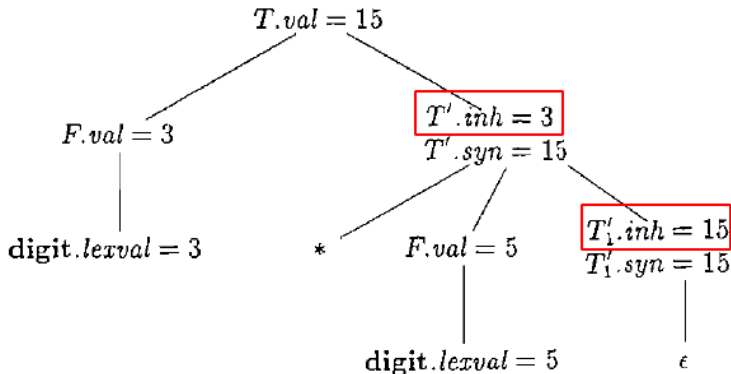
节点 N 上的**继承属性**只能通过 N 的父节点、 N 本身和 N 的兄弟节点上的属性来定义。

继承属性 $T'.inh$ 用于在表达式中从左向右传递中间计算结果



$3 * 5$

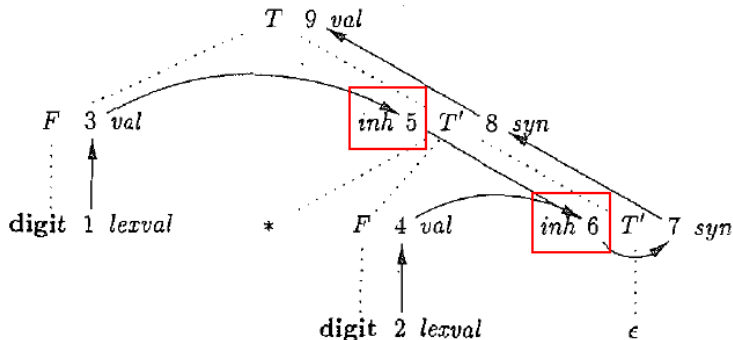
继承属性 $T'.inh$ 用于在表达式中从左向右传递中间计算结果



$3 * 5$

在右递归文法下实现了左结合

依赖图用于确定一棵给定的语法分析树中各个属性实例之间的依赖关系



信息流向: 先从左向右、从上到下传递信息, 再从下到上传递信息

Definition (L 属性定义 (L -Attributed Definition))

如果一个 SDD 的每个属性

- (1) 要么是综合属性,
- (2) 要么是继承属性, 但是它的规则满足如下限制:
对于产生式 $A \rightarrow X_1 X_2 \dots X_n$ 及其对应规则定义的继承属性 $X_i.a$, 则这个规则只能使用
 - (a) 和**产生式头** A 关联的**继承**属性;
 - (b) 位于 **X_i 左边**的文法符号实例 X_1, X_2, \dots, X_{i-1} 相关的**继承**属性或**综合**属性;
 - (c) 和**这个 X_i 的实例本身**相关的继承属性或综合属性, 但是在由这个 X_i 的全部属性组成的依赖图中**不存在环**。

则它是 L 属性定义。

非 L 属性定义

产生式
 $A \rightarrow B C$

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

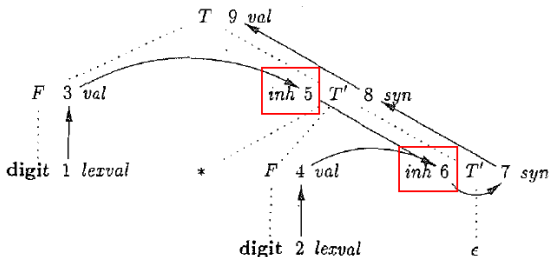
作为继承属性, $B.i$ 依赖了右边的 $C.c$ 属性与头部 $A.s$ 综合属性

L 属性文法: 依赖图是无环的

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

L 属性文法: 依赖图是无环的

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



(左递归) S 属性文法 \Rightarrow (右递归) L 属性文法

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

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

仍保持了操作的左结合性

(左递归) S 属性定义

$$A \rightarrow A_1 Y \quad A.a = g(A_1.a, Y.y)$$

$$A \rightarrow X \quad A.a = f(X.x)$$

(左递归) S 属性定义

$$A \rightarrow A_1 Y \quad A.a = g(A_1.a, Y.y)$$

$$A \rightarrow X \quad A.a = f(X.x)$$

(右递归) L 属性定义

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

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

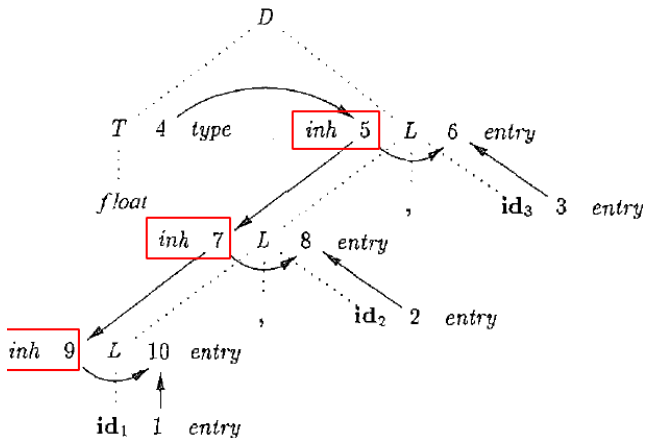
$$R \rightarrow \epsilon \quad R.s = R.i$$

类型声明文法举例

产生式	语义规则
1) $D \rightarrow T L$	$L.inh = T.type$
2) $T \rightarrow \text{int}$	$T.type = \text{integer}$
3) $T \rightarrow \text{float}$	$T.type = \text{float}$
4) $L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$ $\text{addType}(\text{id.entry}, L.inh)$
5) $L \rightarrow \text{id}$	$\text{addType}(\text{id.entry}, L.inh)$

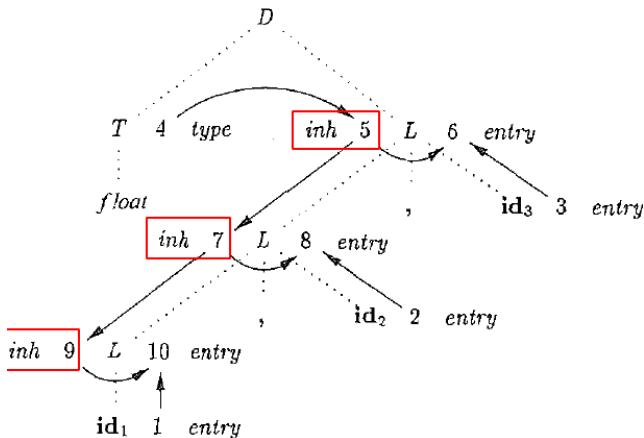
float $\text{id}_1, \text{id}_2, \text{id}_3$

L.inh 将声明的类型沿着标识符列表向下传递, 并被加入到符号表中



`float id1, id2, id3`

L.inh 将声明的类型沿着标识符列表向下传递, 并被加入到符号表中



float id_1, id_2, id_3

addType() 是一种受控的副作用, 使用全局符号表, 更易实现

数组类型文法举例

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

`int[2][3]`

数组类型文法举例

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

`int[2][3]`

语义规则用以生成**类型表达式** $\text{array}(2, \text{array}(3, \text{integer}))$

数组类型文法举例

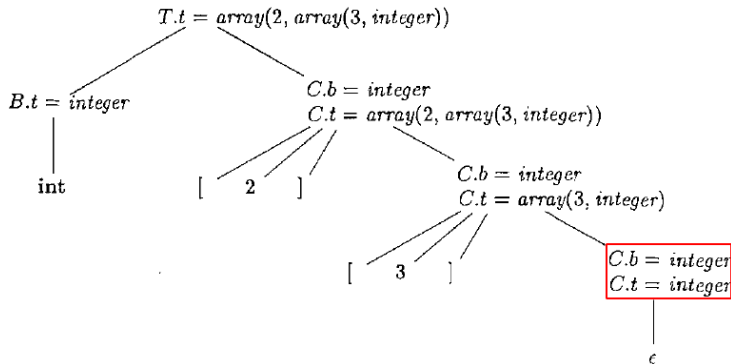
产生式	语义规则
$T \rightarrow B C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.t = \text{array}(\text{num.val}, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

`int[2][3]`

语义规则用以生成**类型表达式** `array(2, array(3, integer))`

`[]`是右结合的

继承属性 *C.b* 将一个基本类型沿着树向下传播



`int[2][3]`

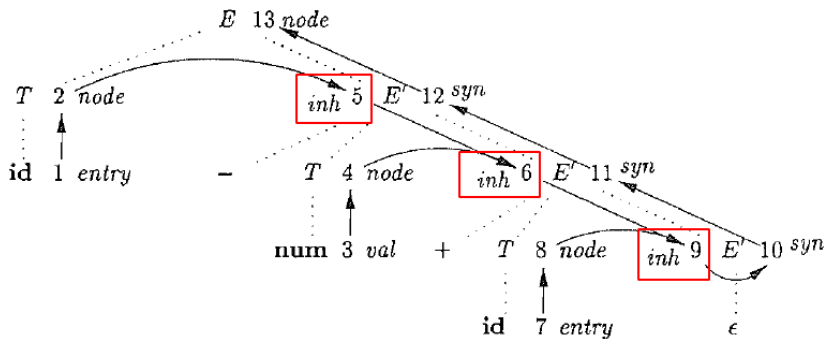
综合属性 *C.t* 收集最终得到的类型表达式

表达式的抽象语法树 S 属性定义

产生式	语义规则
1) $E \rightarrow E_1 + T$	$E.node = \text{new Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \text{new Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
6) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$

表达式的抽象语法树 L 属性定义

产生式	语义规则
1) $E \rightarrow T E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \rightarrow + T E'_1$	$E'_1.inh = \text{new Node}('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3) $E' \rightarrow - T E'_1$	$E'_1.inh = \text{new Node}('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4) $E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5) $T \rightarrow (E)$	$T.node = E.node$
6) $T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$
7) $T \rightarrow \text{num}$	$T.node = \text{new Leaf}(\text{num}, \text{num.val})$



$$a - 4 + c$$

Definition (后缀表示 (Postfix Notation))

- (1) 如果 E 是一个 **变量或常量**, 则 E 的后缀表示是 E 本身;
- (2) 如果 E 是形如 $E_1 \text{ op } E_2$ 的表达式, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 这里 E'_1 和 E'_2 分别是 E_1 与 E_2 的后缀表达式;
- (3) 如果 E 是形如 (E_1) 的表达式, 则 E 的后缀表示是 E_1 的后缀表示。

Definition (后缀表示 (Postfix Notation))

- (1) 如果 E 是一个 **变量或常量**, 则 E 的后缀表示是 E 本身;
- (2) 如果 E 是形如 $E_1 \text{ op } E_2$ 的表达式, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 这里 E'_1 和 E'_2 分别是 E_1 与 E_2 的后缀表达式;
- (3) 如果 E 是形如 (E_1) 的表达式, 则 E 的后缀表示是 E_1 的后缀表示。

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

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

Definition (后缀表示 (Postfix Notation))

- (1) 如果 E 是一个 **变量或常量**, 则 E 的后缀表示是 E 本身;
- (2) 如果 E 是形如 $E_1 \text{ op } E_2$ 的表达式, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 这里 E'_1 和 E'_2 分别是 E_1 与 E_2 的后缀表达式;
- (3) 如果 E 是形如 (E_1) 的表达式, 则 E 的后缀表示是 E_1 的后缀表示。

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

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

$$952 + -3* \implies$$

Definition (后缀表示 (Postfix Notation))

- (1) 如果 E 是一个 **变量或常量**, 则 E 的后缀表示是 E 本身;
- (2) 如果 E 是形如 $E_1 \text{ op } E_2$ 的表达式, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 这里 E'_1 和 E'_2 分别是 E_1 与 E_2 的后缀表达式;
- (3) 如果 E 是形如 (E_1) 的表达式, 则 E 的后缀表示是 E_1 的后缀表示。

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

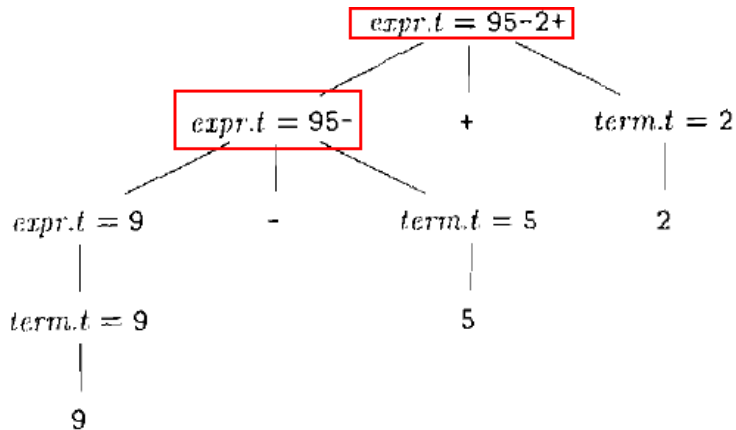
$$9 - (5 + 2) \implies 952 + -$$

$$952 + -3* \implies (9 - (5 + 2)) * 3$$

后缀表达式 S 属性定义

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

“ \parallel ” 表示字符串的连接



$9 - 5 + 2$

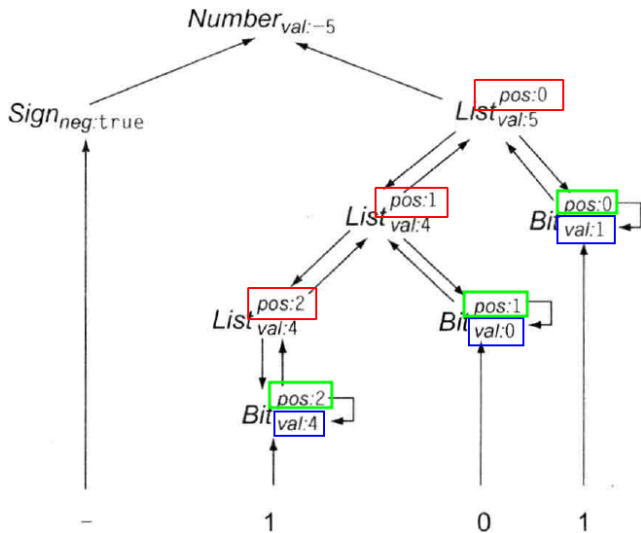
有符号二进制数文法

$$P = \left\{ \begin{array}{ll} \text{Number} & \rightarrow \text{Sign List} \\ \text{Sign} & \rightarrow \begin{array}{l} + \\ - \end{array} \\ \text{List} & \rightarrow \begin{array}{l} \text{List Bit} \\ \text{Bit} \end{array} \\ \text{Bit} & \rightarrow \begin{array}{l} 0 \\ 1 \end{array} \end{array} \right\}$$
$$T = \{+, -, 0, 1\}$$
$$NT = \{\text{Number}, \text{Sign}, \text{List}, \text{Bit}\}$$
$$S = \{\text{Number}\}$$

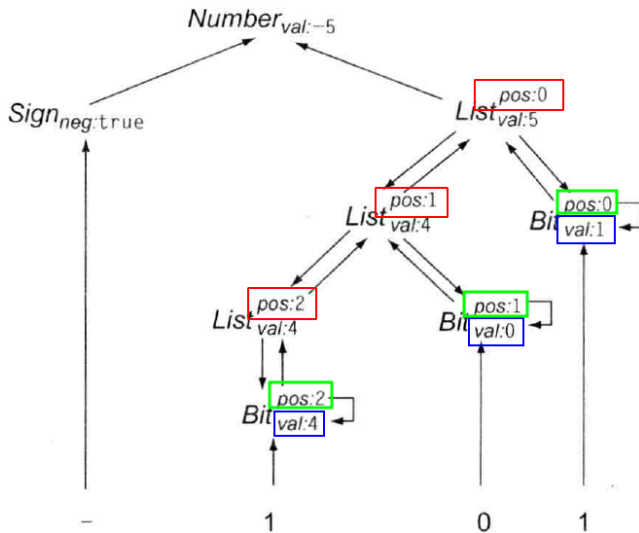
$$-101_2 = -5_{10}$$

有符号二进制数 L 属性定义

	产生式	属性规则
1	$Number \rightarrow Sign\ List$	$List.position \leftarrow 0$ if $Sign.negative$ then $Number.value \leftarrow -List.value$ else $Number.value \leftarrow List.value$
2	$Sign \rightarrow +$	$Sign.negative \leftarrow false$
3	$Sign \rightarrow -$	$Sign.negative \leftarrow true$
4	$List \rightarrow Bit$	$Bit.position \leftarrow List.position$ $List.value \leftarrow Bit.value$
5	$List_0 \rightarrow List_1\ Bit$	$List_1.position \leftarrow List_0.position + 1$ $Bit.position \leftarrow List_0.position$ $List_0.value \leftarrow List_1.value + Bit.value$
6	$Bit \rightarrow 0$	$Bit.value \leftarrow 0$
7	$Bit \rightarrow 1$	$Bit.value \leftarrow 2^{Bit.position}$



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



Q: 可否不用 *Bit.pos* 继承属性?



一对概念



两类属性定义



三种实现方式



四大应用

Definition (语法制导的翻译方案 (Syntax-Directed Translation Scheme; SDT))

SDT 是在其产生式体中嵌入**语义动作**的上下文无关文法。

语义动作可以嵌入在产生式体中的任何位置

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

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

语义动作可以嵌入在产生式体中的任何位置

- 1) $L \rightarrow E \mathbf{n}$
- 2) $E \rightarrow \{ \text{print}('+'); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}('*'); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$

前缀表达式 SDT

语义动作可以嵌入在产生式体中的任何位置

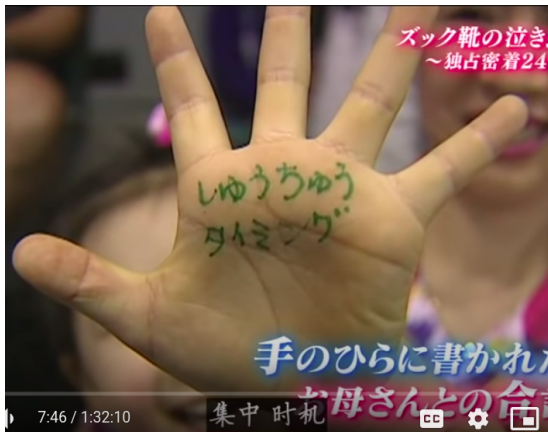
- 1) $L \rightarrow E \mathbf{n}$
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$

前缀表达式 SDT

语义动作嵌入的位置决定了**何时**执行该动作

基本思想: 一个动作在它**左边的**所有文法符号都**处理**过之后立刻执行

时机 (Timing; タイミング)



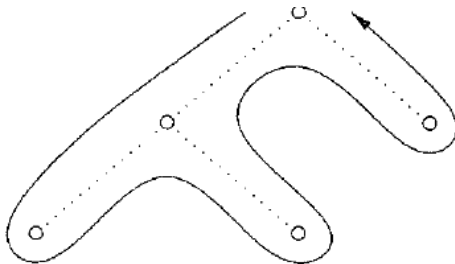
语义动作嵌入在什么地方？何时执行语义动作？

Q: 如何将 SDD 中的**语义规则**转换为带有**语义动作**的 SDT

	S 属性定义	L 属性定义
Offline		
LR		
LL		

Q: 如何以**三种方式**实现 SDT?

Offline 方式: 已有语法分析树



按照**从左到右**的**深度优先**顺序遍历语法分析树

基本思想: 一个动作在它**左边的**所有文法符号都**处理**过之后立刻执行

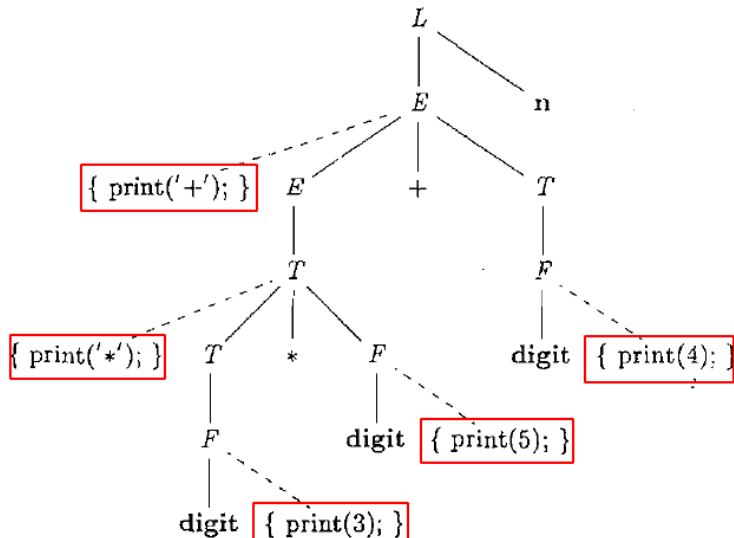
- 1) $L \rightarrow E \mathbf{n}$
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \mathbf{digit} \{ \text{print}(\mathbf{digit.lexval}); \}$

$3 * 5 + 4 \implies$

- 1) $L \rightarrow E \mathbf{n}$
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$

$3 * 5 + 4 \implies + * 354$

嵌入语义动作**虚拟节点**的语法分析树



$3 * 5 + 4 \Rightarrow + * 354$

	S 属性定义	L 属性定义
Offline	嵌入语义动作 虚拟节点	
LR		
LL		

基本思想: 一个动作在它**左边的**所有文法符号都**处理**过之后立刻执行

基本思想: 一个动作在它**左边的**所有文法符号都**处理**过之后立刻执行



Q: 是否所有的 SDT 都可以在 *LL/LR* 语法分析过程中实现?

该 SDT 无法在 $LL(1)/LR(1)$ 中实现

- 1) $L \rightarrow E n$
- 2) $E \rightarrow \{ \text{print}('+'); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}('*'); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$

前缀表达式 SDT

该 SDT **无法**在 $LL(1)/LR(1)$ 中实现

- 1) $L \rightarrow E n$
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$

前缀表达式 SDT

它需要在还不知道出现在输入中的运算符是 $*$ 还是 $+$ 时,
就执行打印这些运算符的操作

Q: 如何判断某 SDT 是否可以在 LL/LR 语法分析过程中实现?

Q: 如何判断某 SDT 是否可以在 LL/LR 语法分析过程中实现?

将每个内嵌的语义动作 A 替换为一个独有的非终结符 M

添加新产生式 $M \rightarrow \epsilon$

判断新产生的文法是否可用 LL/LR 进行分析

前缀表达式 SDT

- 1) $L \rightarrow E \mathbf{n}$ **M2**
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$ **M4**
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.} \textit{lexval}); \}$ **M7**

$$M_2 \rightarrow \epsilon$$

$$M_4 \rightarrow \epsilon$$

$$M_7 \rightarrow \epsilon$$

- 1) $L \rightarrow E \text{ n}$ **M2**
- 2) $E \rightarrow \{ \text{print}(' + '); \} E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}(' * '); \} T_1 * F$ **M4**
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \{ \text{print}(\text{digit.lexval}); \}$ **M7**

$$M_2 \rightarrow \epsilon$$

$$M_4 \rightarrow \epsilon$$

$$M_7 \rightarrow \epsilon$$

$$[A \rightarrow \alpha \cdot B\beta, a] \in I \implies \forall b \in \text{FIRST}(\beta a). [B \rightarrow \cdot \gamma, b] \in I$$

1)	L	\rightarrow	$E \text{ n}$	M2
2)	E	\rightarrow	$\{ \text{print}(' + '); \}$	$E_1 + T$
3)	E	\rightarrow	T	M4
4)	T	\rightarrow	$\{ \text{print}(' * '); \}$	$T_1 * F$
5)	T	\rightarrow	F	
6)	F	\rightarrow	(E)	
7)	F	\rightarrow	$\text{digit } \{ \text{print}(\text{digit.lexval}); \}$	M7

$$M_2 \rightarrow \epsilon$$

$$M_4 \rightarrow \epsilon$$

$$M_7 \rightarrow \epsilon$$

$$[A \rightarrow \alpha \cdot B\beta, a] \in I \implies \forall b \in \text{FIRST}(\beta a). [B \rightarrow \cdot \gamma, b] \in I$$

- | | | | | |
|----|-----|---------------|---|-----------|
| 1) | L | \rightarrow | $E \mathbf{n}$ | M2 |
| 2) | E | \rightarrow | $\{ \text{print}(' + '); \}$ | $E_1 + T$ |
| 3) | E | \rightarrow | T | M4 |
| 4) | T | \rightarrow | $\{ \text{print}(' * '); \}$ | $T_1 * F$ |
| 5) | T | \rightarrow | F | |
| 6) | F | \rightarrow | (E) | |
| 7) | F | \rightarrow | $\text{digit} \{ \text{print}(\text{digit.lexval}); \}$ | M7 |

$$M_2 \rightarrow \epsilon$$

$$M_4 \rightarrow \epsilon$$

$$M_7 \rightarrow \epsilon$$

$$L \rightarrow \cdot E \mathbf{n}, \quad \$$$

$$E \rightarrow \cdot \mathbf{M_2} E + T, \quad \mathbf{n}$$

$$E \rightarrow \cdot T, \quad \mathbf{n}$$

$$\mathbf{M_2} \rightarrow \cdot, \quad (/ \text{digit}$$

$$T \rightarrow \cdot \mathbf{M_4} T * F, \quad \mathbf{n}$$

$$T \rightarrow \cdot F, \quad \mathbf{n}$$

$$\mathbf{M_4} \rightarrow \cdot, \quad (/ \text{digit}$$

$$F \rightarrow \cdot (E), \quad \mathbf{n}$$

$$F \rightarrow \cdot \text{digit } \mathbf{M_7}, \quad \mathbf{n}$$

遇到 **digit**, 产生移入/归约冲突

	S 属性定义	L 属性定义
Offline	嵌入语义动作 虚拟节点	
LR	✓	
LL		✓

S 属性定义

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

后缀翻译方案

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

语义动作

S 属性定义

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

后缀翻译方案

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

语义动作

后缀翻译方案: 所有动作都在产生式的最后

在 LR 中, 按某个产生式**归约**时, 执行相应动作

$$A \rightarrow XYZ$$

	X	Y	Z
	$X.x$	$Y.y$	$Z.z$

状态 / 文法符号

综合属性

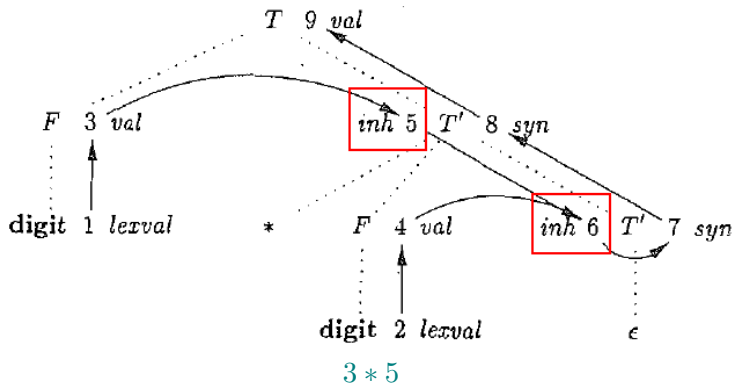
栈顶

移入时,携带终结符的属性

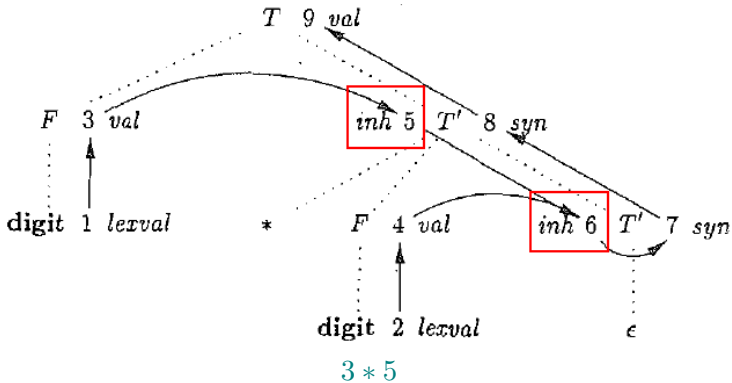
归约时, 计算 A 的属性值并入栈

	S 属性定义	L 属性定义
Offline	嵌入语义动作 虚拟节点	
LR	后缀 翻译方案	
LL		✓

L 属性定义 与 LL 语法分析



L 属性定义 与 LL 语法分析



$$A \rightarrow X_1 \cdots X_i \cdots X_n$$

原则: 从左到右处理各个 X_i 符号

对每个 X_i , 先计算继承属性, 后计算综合属性

递归下降子过程 $A \rightarrow X_1 \cdots X_i \cdots X_n$

- ▶ 在调用 X_i 子过程之前, 计算 X_i 的**继承属性**
- ▶ 以 X_i 的继承属性为**参数**调用 X_i 子过程
- ▶ 在 X_i 子过程返回之前, 计算 X_i 的**综合属性**
 - ▶ 在 X_i 子过程中**返回** X_i 的综合属性

(左递归) S 属性定义

$$A \rightarrow A_1 Y \quad A.a = g(A_1.a, Y.y)$$

$$A \rightarrow X \quad A.a = f(X.x)$$

$XY Y$

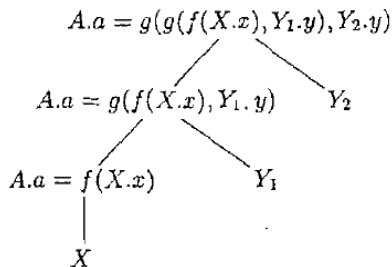
(右递归) L 属性定义

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

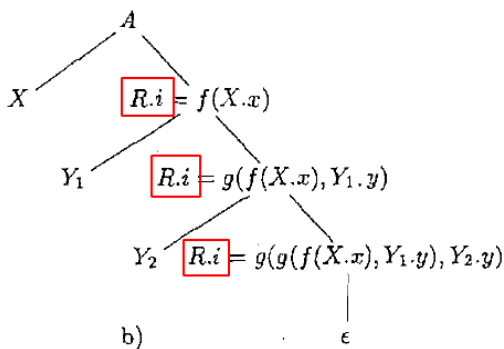
$$R \rightarrow Y R_1 \quad R_1.i = g(R.i, Y.y); \quad R.s = R_1.s$$

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

继承属性 $R.i$ 用于计算并传递中间结果



a)



b)

c

先计算继承属性, 再计算综合属性

(右递归) L 属性定义

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

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

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

原则：继承属性在处理文法符号之前，综合属性在处理文法符号之后

(右递归) L 属性定义

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

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

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

原则：继承属性在处理文法符号之前，综合属性在处理文法符号之后

L 属性定义的 SDT

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

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

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

$$A \rightarrow X \quad \{\textcolor{red}{R}.i = f(X.x)\} \quad R \quad \{A.a = R.s\}$$
$$R \rightarrow Y \quad \{\textcolor{red}{R}_1.i = g(R.i, Y.y)\} \quad R_1 \quad \{R.s = R_1.s\}$$
$$R \rightarrow \epsilon \quad \{\textcolor{blue}{R}.s = \textcolor{blue}{R}.i\}$$

-
- | | |
|--|-----------------------------------|
| 1: procedure $A()$ | ▷ A 是开始符号, 无需继承属性做参数 |
| 2: if token = ? then | ▷ 假设选择 $A \rightarrow XR$ 产生式 |
| 3: $X.x \leftarrow \text{MATCH}(X)$ | ▷ 假设 X 是终结符, 返回综合属性 |
| 4: $\textcolor{red}{R}.i \leftarrow f(X.x)$ | ▷ 先计算 $\textcolor{red}{R}.i$ 继承属性 |
| 5: $\textcolor{blue}{R}.s \leftarrow R(R.i)$ | ▷ 递归调用子过程 $R(R.i)$ |
| 6: return $\textcolor{red}{R}.s$ | ▷ 返回 $A.a \leftarrow R.s$ 综合属性 |
-

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

1: procedure $R(R.i)$	▷ R 使用继承属性 $R.i$ 做参数
2: if token = ? then	▷ 假设选择 $R \rightarrow YR$ 产生式
3: $Y.y \leftarrow \text{MATCH}(Y)$	▷ 假设 Y 是终结符, 返回综合属性
4: $R.i \leftarrow g(R.i, Y.y)$	▷ 先计算 $R.i$ 继承属性
5: $R.s \leftarrow R(R.i)$	▷ 递归调用子过程 $R(R.i)$
6: return $R.s$	▷ 返回综合属性
7: else if token = ? then	▷ 假设选择 $R \rightarrow \epsilon$ 产生式
8: return $R.i$	▷ 返回 $R.s \leftarrow R.i$ 综合属性

L 属性定义转换为 SDT

$$A \rightarrow X_1 \cdots X_i \cdots X_n$$

计算 X_i **继承属性**的动作放在产生式体中 X_i 的**左边**



计算产生式头部 A **综合属性**的动作放在产生式体的**最右边**

$$\begin{aligned}
 E &\rightarrow E_1 + T \{ \text{print}(' + '); \} \\
 E &\rightarrow T
 \end{aligned}$$

特例: 语义动作不涉及属性的计算

$$\begin{aligned}
 E &\rightarrow T R \\
 R &\rightarrow + T \{ \text{print}(' + '); \} R \\
 R &\rightarrow \epsilon
 \end{aligned}$$

	S 属性定义	L 属性定义
Offline	嵌入语义动作 虚拟节点	
LR	后缀 翻译方案	
LL		先 继承 , 后 综合

	S 属性定义	L 属性定义
Offline	嵌入语义动作 虚拟节点	
LR	后缀 翻译方案	
LL		先 继承 , 后 综合

Additional Report (PPT + Video)

while 语句的翻译

类型声明与使用 (符号表)

类型检查

要有大局观!!!



认清“你”在语法分析树中所处的位置

```

 $S \rightarrow \text{while} ( C ) S_1 \quad \begin{array}{l} L1 = \text{new}(); \\ L2 = \text{new}(); \\ S_1.\text{next} = L1; \\ C.\text{false} = S.\text{next}; \\ C.\text{true} = L2; \\ S.\text{code} = \text{label} \parallel L1 \parallel C.\text{code} \parallel \text{label} \parallel L2 \parallel S_1.\text{code} \end{array}$ 

```

图 5-27 while 语句的 SDD

$S \rightarrow$	while ({	$L1 = new();$	$L2 = new();$	$C.false = S.next;$	$C.true = L2;$	}
$C)$		{	$S_1.next = L1;$	}			
S_1		{	$S.code = \text{label} \parallel L1 \parallel C.code \parallel \text{label} \parallel L2 \parallel S_1.code;$	}			

图 5-28 while 语句的 SDT

```

string S(label next) {
    string Scode, Ccode; /* 存放代码片段的局部变量 */
    label L1, L2; /* 局部标号 */
    if (当前输入 == 词法单元 while) {
        读取输入;
        检查 '(' 是下一个输入符号, 并读取输入;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2); C(c.false, c.true)
        检查 ')' 是下一个输入符号, 并读取输入;
        Scode = S(L1); S(S.next)
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* 其他语句类型 */
}

```

图 5-29 用一个递归下降语法分析器实现 while 语句的翻译

$S \rightarrow$	while ({	$L1 = new();$	$L2 = new();$	$C.false = S.next;$	$C.true = L2;$	}
$C)$		{	$S_1.next = L1;$	}			
S_1		{	$S.code = \text{label} \parallel L1 \parallel C.code \parallel \text{label} \parallel L2 \parallel S_1.code;$				
		}					

图 5-28 while 语句的 SDT

$$\begin{array}{ll}
 S \rightarrow \text{while} (& \{ L1 = \text{new}(); L2 = \text{new}(); C.\text{false} = S.\text{next}; C.\text{true} = L2; \} \\
 C) & \{ S_1.\text{next} = L1; \} \\
 S_1 & \{ S.\text{code} = \text{label} \parallel L1 \parallel C.\text{code} \parallel \text{label} \parallel L2 \parallel S_1.\text{code}; \}
 \end{array}$$

图 5-28 while 语句的 SDT

$$\begin{array}{ll}
 S \rightarrow \text{while} (& \{ L1 = \text{new}(); L2 = \text{new}(); C.\text{false} = S.\text{next}; \\
 & C.\text{true} = L2; \text{print}(\text{"label"}, L1); \} \\
 C) & \{ S_1.\text{next} = L1; \text{print}(\text{"label"}, L2); \} \\
 S_1 &
 \end{array}$$

图 5-32 边扫描边生成 while 语句的代码的 SDT


```

void S(label next) {
    label L1, L2; /* 局部标号 */
    if ( 当前输入 == 词法单元 while ) {
        读取输入;
        检查 '(' 是下一个输入符号, 并读取输入;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        检查 ')' 是下一个输入符号, 并读取输入;
        print("label", L2);
        S(L1);
    }
    else /* 其他语句类型 */
}

```

图 5-31 while 语句的 on-the-fly 的递归下降代码生成

Definition (符号表 (Symbol Table))

符号表是用于保存各种信息的**数据结构**。

标识符: 词素、类型、大小、存储位置等

Definition (符号表 (Symbol Table))

符号表是用于保存各种信息的**数据结构**。

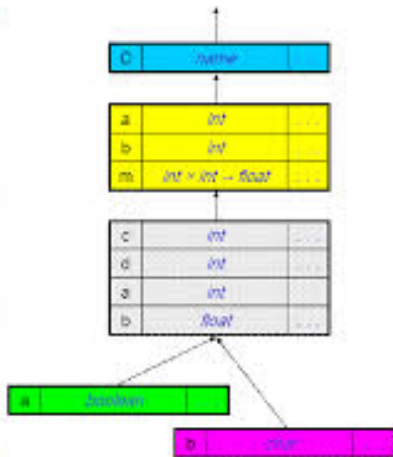
标识符: 词素、类型、大小、存储位置等

```
public class ST<Key extends Comparable<Key>, Value>
```

ST()	<i>create an empty symbol table</i>
void put(Key key, Value val)	<i>associate val with key</i>
Value get(Key key)	<i>value associated with key</i>
void remove(Key key)	<i>remove key (and its associated value)</i>
boolean contains(Key key)	<i>is there a value associated with key?</i>
int size()	<i>number of key-value pairs</i>
Iterable<Key> keys()	<i>all keys in the symbol table</i>

通常使用 HashTable

为每个作用域建立单独的符号表



可以使用符号表栈实现树形的嵌套关系

```

1) package symbols;                                // 文件 Env.java
2) import java.util.*;
3) public class Env {
4)     private Hashtable table;
5)     protected Env prev;
6)     public Env(Env p) { 创建新的, 旧的“压栈”
7)         table = new Hashtable(); prev = p;
8)     } 放入当前符号表
9)     public void put(String s, Symbol sym) {
10)         table.put(s, sym);
11)     } 倒序搜索符号表链
12)     public Symbol get(String s) {
13)         for( Env e = this; e != null; e = e.prev ) {
14)             Symbol found = (Symbol)(e.table.get(s));
15)             if( found != null ) return found;
16)         }
17)         return null;
18)     }
19) }

```

```
{ int x; char y; { bool y; x; y; } x; y; }
```

```
{ int x; char y; { bool y; x; y; } x; y; }
```

翻译任务：忽略**声明**部分，为每个标识符标明**类型**

```
{ { x:int; y:bool; } x:int; y:char; }
```

Definition (类型表达式 (Type Expressions))

- ▶ **基本类型**是类型表达式;
 - ▶ char, bool, int, float, double, void, ...
- ▶ **类名**是类型表达式;

Definition (类型表达式 (Type Expressions))

- ▶ **基本类型**是类型表达式;
 - ▶ char, bool, int, float, double, void, ...
- ▶ **类名**是类型表达式;
- ▶ 如果 t 是类型表达式, 则 **array(num, t)** 是类型表达式;
- ▶ **record(<id : t, ...>)** 是类型表达式;

Definition (类型表达式 (Type Expressions))

- ▶ **基本类型**是类型表达式;
 - ▶ char, bool, int, float, double, void, ...
- ▶ **类名**是类型表达式;
- ▶ 如果 t 是类型表达式, 则 **array(num, t)** 是类型表达式;
- ▶ **record(<id : t, ...>)** 是类型表达式;
- ▶ 如果 s 和 t 是类型表达式, 则 $s \times t$ 是类型表达式;
- ▶ 如果 s 和 t 是类型表达式, 则 $s \rightarrow t$ 是类型表达式;

Definition (类型表达式 (Type Expressions))

- ▶ **基本类型**是类型表达式;
 - ▶ char, bool, int, float, double, void, ...
- ▶ **类名**是类型表达式;
- ▶ 如果 t 是类型表达式, 则 **array(num, t)** 是类型表达式;
- ▶ **record(<id : t, ...>)** 是类型表达式;
- ▶ 如果 s 和 t 是类型表达式, 则 $s \times t$ 是类型表达式;
- ▶ 如果 s 和 t 是类型表达式, 则 $s \rightarrow t$ 是类型表达式;
- ▶ 类型表达式可以包含取值为类型表达式的变量。

<i>program</i>	→	block	{ <i>top</i> = null; }
<i>block</i>	→	'{'	{ <i>saved</i> = <i>top</i> ;
			<i>top</i> = new <i>Env</i> (<i>top</i>);
		decls stmts '}'	print("{ "); }
			{ <i>top</i> = <i>saved</i> ;
			print("} "); }
<i>decls</i>	→	<i>decls decl</i>	
			ε
<i>decl</i>	→	type id ;	{ <i>s</i> = new <i>Symbol</i> ;
			<i>s.type</i> = type.lexeme ;
			<i>top.put</i> (id.lexeme , <i>s</i>); }
<i>stmts</i>	→	<i>stmts stmt</i>	
			ε
<i>stmt</i>	→	block	{ print("; "); }
			<i>factor</i> ;
<i>factor</i>	→	id	{ <i>s</i> = <i>top.get</i> (id.lexeme);
			print(id.lexeme);
			print(": ");
			print(<i>s.type</i>); }

```
{ char y; { bool y; y; } y; }
```

```
{ { y : bool; } y:char; }
```

<i>program</i> →	block	{ top = null; }
<i>block</i> →	'{'	{ saved = <i>top</i> ;
		<i>top</i> = new Env(<i>top</i>);
		print("{ "); }
	decls stmts '}'	{ <i>top</i> = saved
		print("} "); }
<i>decls</i> →	<i>decls decl</i>	
		ε
<i>decl</i> →	type id ;	{ <i>s</i> = new Symbol;
		<i>s.type</i> = type.lexeme ;
		top.put (<i>id.lexeme</i> , <i>s</i>); }
<i>stmts</i> →	<i>stmts stmt</i>	
		ε
<i>stmt</i> →	block	
		<i>factor</i> ;
<i>factor</i> →	id	{ <i>s</i> = top.get (<i>id.lexeme</i>);
		print(<i>id.lexeme</i>);
		print(": ");
		print(<i>s.type</i>); }

类型声明

```
float x;  
record { float x; float y; } p;  
record { int tag; float x; float y; } q;
```

符号表中记录标识符的类型、宽度 (width)、偏移地址 (offset)

$$\begin{aligned}
 D &\rightarrow T \text{ id } ; D \mid \epsilon \\
 T &\rightarrow B C \mid \text{record } \{ D \} \\
 B &\rightarrow \text{int} \mid \text{float} \\
 C &\rightarrow \epsilon \mid [\text{num}] C
 \end{aligned}$$

需要为每个 **record** 生成单独的符号表

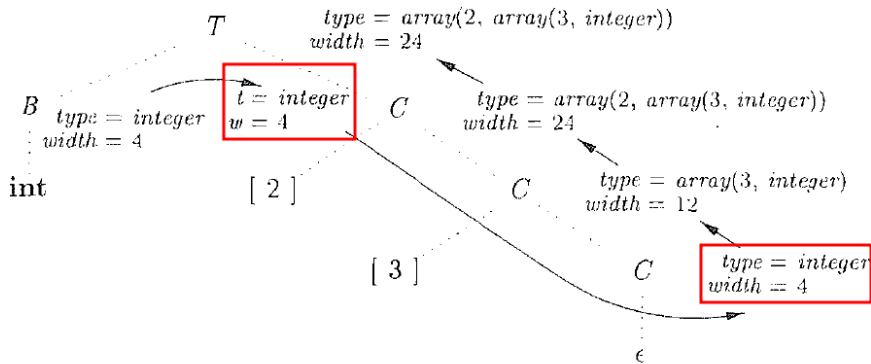


图 6-16 数组类型的语法制导翻译

`int [2] [3]`

全局变量 `offset` 表示变量的相对地址

全局变量 `top` 表示当前的符号表

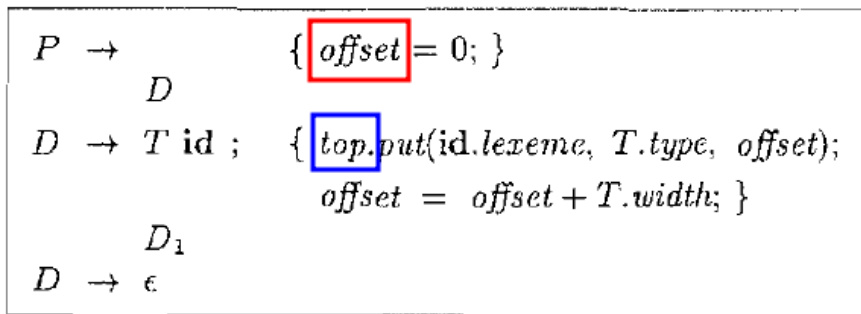


图 6-17 计算被声明变量的相对地址

```
float x; float y;
```

```

T → record '{' { Env.push(top); top = new Env();
                  Stack.push(offset); offset = 0; }

D '}' { T.type = record(top); T.width = offset;
        top = Env.pop(); offset = Stack.pop(); }

```

图 6-18 处理记录中的字段名

```

T → record '{'  { Env.push(top); top = new Env();
                  Stack.push(offset); offset = 0; }

D '}'           { T.type = record(top); T.width = offset;
                  top = Env.pop(); offset = Stack.pop(); }

```

图 6-18 处理记录中的字段名

record 类型表达式: record(top)

```

T → record '{'   { Env.push(top); top = new Env();
                  Stack.push(offset); offset = 0; }

D '}'           { T.type = record(top); T.width = offset;
                  top = Env.pop(); offset = Stack.pop(); }

```

图 6-18 处理记录中的字段名

record 类型表达式: record(top)

全局变量 top 表示当前的符号表

```

T → record '{' { Env.push(top); top = new Env();
                  Stack.push(offset); offset = 0; }

D '}' { T.type = record(top); T.width = offset;
        top = Env.pop(); offset = Stack.pop(); }

```

图 6-18 处理记录中的字段名

record 类型表达式: record(top)

全局变量 top 表示当前的符号表

全局变量 Env 表示符号表栈

```

T → record '{' { Env.push(top); top = new Env();
                  Stack.push(offset); offset = 0; }

                  D '}' { T.type = record(top); T.width = offset;
                        top = Env.pop(); offset = Stack.pop(); }

```

图 6-18 处理记录中的字段名

record 类型表达式: record(top)

全局变量 top 表示当前的符号表

全局变量 Env 表示符号表栈

```
record { float x; float y; } p;
```



```
float x;  
record { float x; float y; } p;  
record { int tag; float x; float y; } q;
```

类型检查的常见形式

```
if two type expressions are equivalent  
then return a given type  
else return type_error
```

```
typedef struct {  
    int data[100];  
    int count;  
} Stack;
```

```
typedef struct {  
    int data[100];  
    int count;  
} Set;
```

```
Stack x, y;  
Set r, s;
```

Definition (结构等价 (Structurally Equivalent))

两种类型**结构等价**当且仅当以下任一条件为真:

- ▶ 它们是相同的基本类型;
- ▶ 它们是将相同的类型构造算子应用于结构等价的类型而构造得到;
- ▶ 一个类型是另一个类型表达式的名字。

Definition (结构等价 (Structurally Equivalent))

两种类型**结构等价**当且仅当以下任一条件为真:

- ▶ 它们是相同的基本类型;
- ▶ 它们是将相同的类型构造算子应用于结构等价的类型而构造得到;
- ▶ **一个类型是另一个类型表达式的名字。**

Definition (名等价 (Name Equivalent))

两种类型**名等价**当且仅当以下任一条件为真:

- ▶ 它们是相同的基本类型;
- ▶ 它们是将相同的类型构造算子应用于结构等价的类型而构造得到。

结构等价中的“结构”又是什么意思？

`array(n, t)` `array(m, t)`

`record(a : int)` `record(b : int)`

结构等价中的“结构”又是什么意思？

`array(n, t)` `array(m, t)`

`record(<a : int>)` `record(<b : int>)`

取决于语言设计者的“大局观”

类型综合: 根据子表达式的类型确定表达式的类型

if f 的类型为 $s \rightarrow t$ 且 x 的类型为 s
then 表达式 $f(x)$ 的类型为 t

$$E_1 + E_2$$


```
void err() { ... }  
void err(String s) { ... }
```

重载函数的类型综合规则

if f 可能的类型为 $s_i \rightarrow t_i$ ($1 \leq i \leq n$), 其中, $s_i \neq s_j$ ($i \neq j$)
and x 的类型为 s_k ($1 \leq k \leq n$)
then 表达式 $f(x)$ 的类型为 t_k

类型推导: 根据某语言结构的使用方式确定表达式的类型

if $f(x)$ 是一个表达式,
then 对某些 α 和 β , f 的类型为 $\alpha \rightarrow \beta$ 且 x 的类型为 α

$null(x) : x$ 是一个列表, 它的元素类型未知

类型转换

```
t1 = (float) 2  
t2 = t1 * 3.14
```

类型转换

```
t1 = (float) 2  
t2 = t1 * 3.14
```

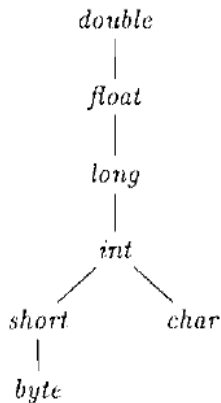
```
if ( E1.type = integer and E2.type = integer ) E.type = integer;  
else if ( E1.type = float and E2.type = integer ) ...  
...
```

类型转换

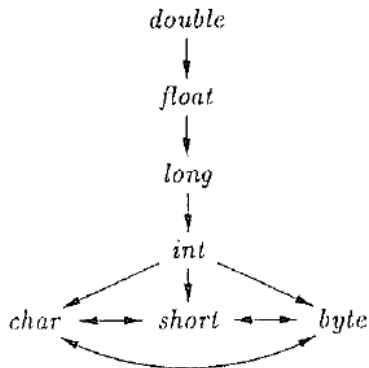
```
t1 = (float) 2  
t2 = t1 * 3.14
```

```
if ( E1.type = integer and E2.type = integer ) E.type = integer;  
else if ( E1.type = float and E2.type = integer ) ...  
...
```

不要写这样的代码!!!



a) 拓宽类型转换



b) 窄化类型转换

```

E → E1 + E2 { E.type = max(E1.type, E2.type);
                    a1 = widen(E1.addr, E1.type, E.type);
                    a2 = widen(E2.addr, E2.type, E.type);
                    E.addr = new Temp();
                    gen(E.addr '=' a1 '+' a2); }

```

图 6-27 在表达式求值中引入类型转换

```

Addr widen(Addr a, Type t, Type w)
    if (t = w) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' (float) a);
        return temp;
    }
    else error;
}

```

Thank
You!



Office 926

hfwei@nju.edu.cn