



编译原理
第五章
语法制导翻译

哈尔滨工业大学 陈歆 单丽莉



什么是语法制导翻译

➤ 编译的阶段

➤ 词法分析

➤ 语法分析

➤ 语义分析

➤ 中间代码生成

➤ 代码优化

➤ 目标代码生成

➤ 语义分析：收集或计算源程序中的上下文相关信息并进行静态语义一致性和完整性的检查。

➤ 类型检查

➤ 控制流检查

➤ 唯一性检查

➤ 一致性检查

➤ 函数调用时实参与形参的数目或类型不匹配

➤

什么是语法制导翻译

➤ 编译的阶段

➤ 词法分析

➤ 语法分析

➤ 语义分析

➤ 中间代码生成

➤ 代码优化

➤ 目标代码生成

语义翻译

语法制导翻译

(Syntax-Directed Translation)

语法制导翻译使用CFG来引导对语言的翻译，
是一种面向文法的翻译技术

什么是语法制导翻译

- 本章重点介绍通用的语法制导翻译技术
- 可用于实现各种语言的翻译任务
 - 编译器语义分析和生成中间代码
 - 排版系统中根据书写格式生成立体公式显示
 - 台式计算器，根据输入的算式，计算结果
 - HTML/XML格式分析生成结构化的网页展示
 - SQL查询语句的翻译
 - HTTP、SMTP等协议的翻译.....

语法制导翻译的基本思想

- 在进行语法分析的同时，完成相应的语义翻译。
 - 语法分析器识别出一个语法结构（选择一个产生式推导或归约）就立即执行该语法结构对应的语义规则来实现翻译。
 - 语义规则：根据语法规则和翻译任务设计。
 - 为每个产生式设置语义规则：收集、计算或传递语义信息、查填符号表、计算值、生成中间代码等。
 - 语义信息：为文法符号设置语义属性，用来表示语法成分对应的语义信息。

两种语义翻译模型

- 将语义规则同语法规则（产生式）联系起来
- 语法制导定义 (*Syntax-Directed Definitions, SDD*)
 - 一种基础的、抽象的语义翻译定义模型，适用于对语法制导翻译原理的理解。
- 语法制导翻译方案 (*Syntax-Directed Translation Scheme, SDT*)
 - 一种面向实现的语义翻译模型，有助于语义翻译程序的构造和实现。

语法制导定义(*SDD*)

➤ *SDD*是对*CFG*的推广

➤ 将每个文法符号和一个语义属性集合相关联

➤ 将每个产生式和一组语义规则相关联，这些规则用于计算该产生式中各文法符号的属性值

➤ 例：

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

语法制导翻译方案(*SDT*)

- *SDT*是在产生式右部嵌入了程序片段的*CFG*，这些程序片段称为语义动作，也称为翻译模式。
- 一个语义动作可以出现在产生式体的任何位置，这个位置决定了这个动作的执行时机

$$D \rightarrow T \{ L.inh = T.type \} L$$
$$T \rightarrow \text{int} \{ T.type = \text{int} \}$$
$$T \rightarrow \text{float} \{ T.type = \text{float} \}$$
$$L \rightarrow \{ L_1.inh = L.inh \} L_1, \text{id} \{ \text{addType}(\text{id.entry}, L.inh) \}$$
$$L \rightarrow \text{id} \{ \text{addType}(\text{id.entry}, L.inh) \}$$



本章内容

5.1 语法制导定义SDD

5.2 S-属性定义与L-属性定义

5.3 语法制导翻译方案SDT

5.4 L-属性定义的自顶向下翻译

5.5 L-属性定义的自底向上翻译

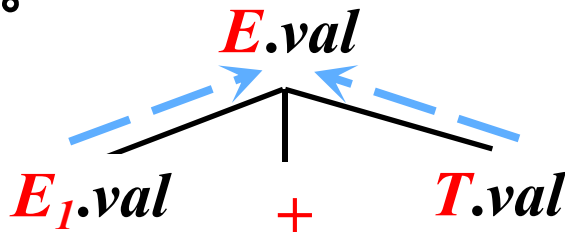
5.1 语法制导定义SDD

- 语法制导定义SDD是对CFG的推广
 - 为 文法符号 设置 一组语义属性，收集和传递语义信息。
 - 为 产生式 设置 一组语义规则，用于计算语义属性的值。
- 文法符号的属性的分类
 - 综合属性 (*synthesized attribute*) — 自底向上 传递信息
 - 继承属性 (*inherited attribute*) — 自顶向下 传递信息

综合属性(*synthesized attribute*) — 自底向上传递信息

- 语法分析树：结点 N 上的非终结符 A 的**综合属性**只能通过 N 子结点或 N 本身的属性值来定义。

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



- 非终结符 A 的**综合属性**是由以 A 为左部的产生式所关联的语义规则来定义的
- 终结符的**综合属性**（固有属性）：是由词法分析器提供的**词法值**。

例：带有综合属性的SDD

SDD: 翻译任务是计算算式的值

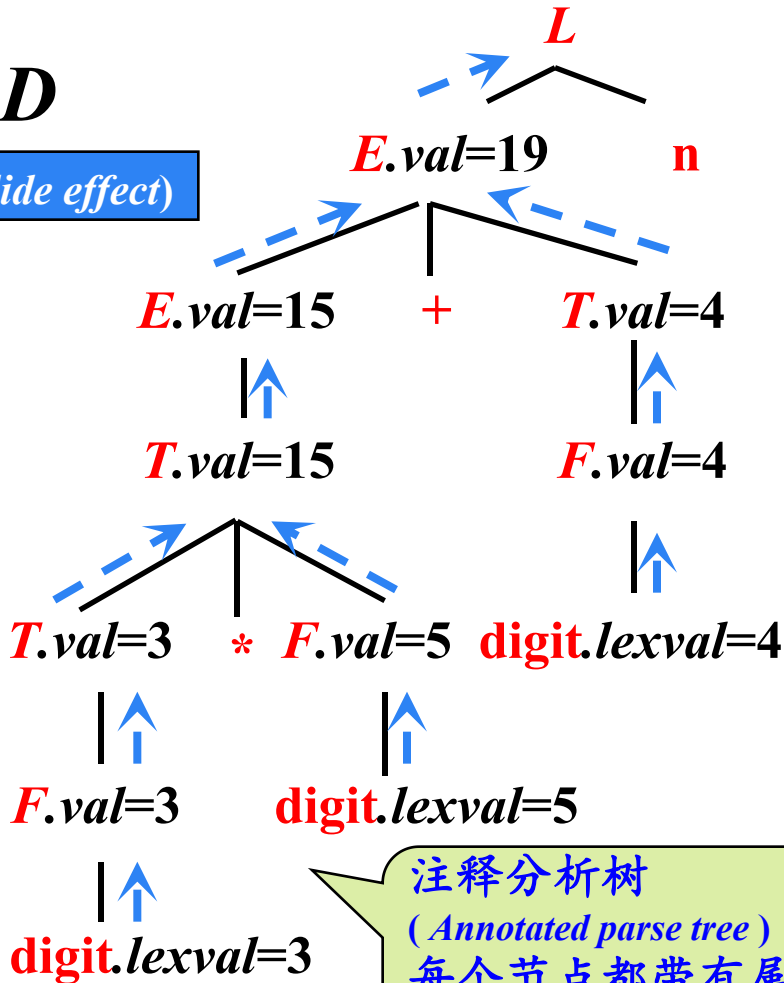
产生式	语义规则
(1) $L \rightarrow E n$	$\text{print}(E.val)$
(2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \rightarrow T$	$E.val = T.val$
(4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \rightarrow F$	$T.val = F.val$
(6) $F \rightarrow (E)$	$F.val = E.val$
(7) $F \rightarrow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

副作用 (Side effect)

输入:

$3 * 5 + 4n$

➤ 副作用可以看作是将函数的返回值赋给L的虚综合属性



注释分析树

(Annotated parse tree):
每个节点都带有属性值的分析树

继承属性(*inherited attribute*) — 自顶向下传递信息

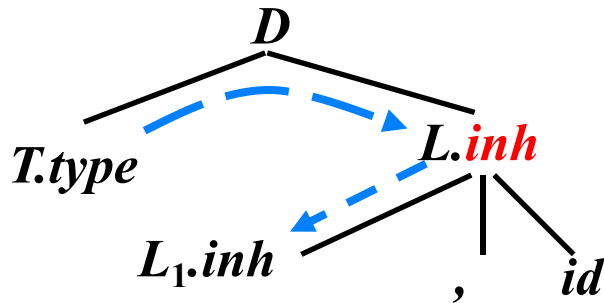
- 语法分析树：节点 N 上非终结符 A 的继承属性通过其父结点、兄弟结点或 N 本身的属性值来定义。

产生式

语义规则

$D \rightarrow T L$ $L.inh = T.type$

$L \rightarrow L_1, id$ $L_1.inh = L.inh$

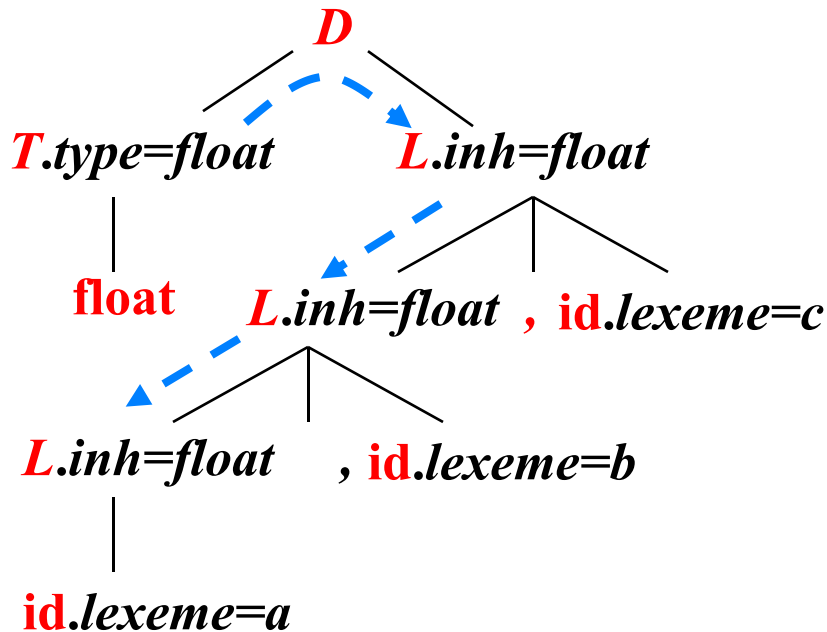


- 非终结符 A 的继承属性是由以 A 为右部符号之一的产生式所关联的语义规则来定义的。
- 终结符没有继承属性。

例：带有继承属性的SDD

SDD: 声明语句的翻译定义

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



输入:

`float a, b, c`

- 将标识符`id`和其数据类型`L.inh`存入符号表:
 $\text{addType}(\text{id的词法值}, \text{id的数据类型})$

属性文法 (*Attribute Grammar*)

- 一个没有副作用的 *SDD* 有时也称为 **属性文法**
- 属性文法的 **语义规则** 仅仅通过 **其它属性值和常量** 来定义一个属性值

➤ 例

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属性计算

通用性较好，无属性循环依赖就可以

➤ 基于分析树遍历的属性计算

➤ 语法分析和属性计算分别独立完成：多遍完成

➤ 生成语法分析树后，再进行属性计算

➤ 语法分析过程中同时完成语义翻译

对SDD有更多限制

➤ 语法分析和属性计算：一遍完成

➤ S属性SDD —— 适合在自底向上语法分析中实现

➤ L属性SDD —— 适合在自顶向下的语法分析中实现

基于分析树遍历的属性计算

基于分析树遍历的属性计算步骤：

- (1) 构造输入串的语法分析树;
- (2) 构造属性计算**依赖图**;
- (3) 按依赖图的一种**拓扑排序**对分析树进行遍历，同时按语义规则计算相关节点各属性的值。

要求： *SDD*属性计算没有循环依赖关系

依赖图(*Dependency Graph*)

➤ 依赖图

➤ 用于描述分析树中结点属性间依赖关系的有向图

➤ 依赖图构造思想

➤ 对于某输入串的分析树中每个结点的每个属性 a 都对应着依赖图中的一个结点，如果属性 $X.a$ 的值依赖于属性 $Y.b$ 的值，则依赖图中有一条从 $Y.b$ 的结点指向 $X.a$ 的结点的有向边。

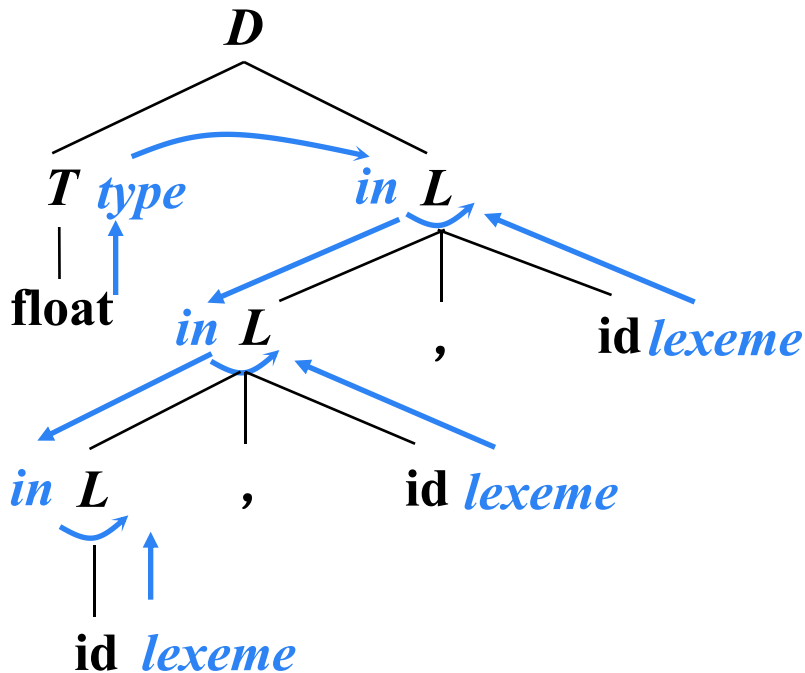
例

SDD : 声明语句的翻译定义

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

输入:

float a , b , c



副作用看作是对L的虚综合属性进行赋值
即: $L.value = \text{addtype}(\text{id.lexeme}, L.in)$

依赖图中属性值的计算顺序

- 依赖图上任意一个可行的求值顺序结点序列 N_1, N_2, \dots, N_k ，必满足条件：对于依赖图中任意一条从结点 N_i 到 N_j 的边 $(N_i \rightarrow N_j)$ ，那么 N_i 排在 N_j 前面，即 $i < j$ 。
- 拓扑排序：满足以上条件的排序，正好是有向图的拓扑排序(*topological sort*)。

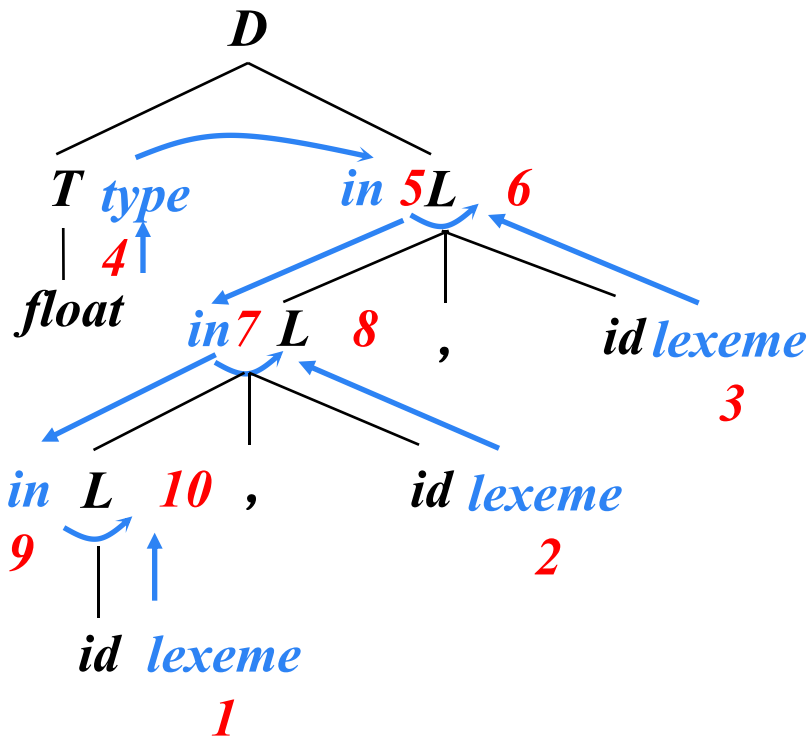
例

SDD : 声明语句的翻译定义

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

输入:

float a, b, c



拓扑排序:

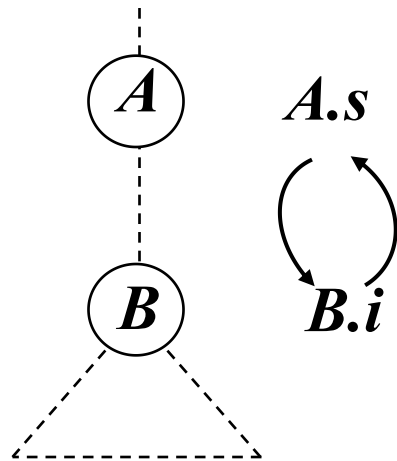
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

4, 3, 2, 1, 5, 7, 6, 9, 8, 10

有回路的依赖图

- 如果分析树的依赖图有回路，表明属性间存在循环依赖关系，就不存在一个拓扑排序来对所有属性进行求值。
- 从计算的角度看，给定一个SDD，很难确定是否存在某棵语法分析树使得属性间存在循环依赖关系。
- 幸运的是，存在SDD的有用子类，它们能够保证对每棵语法分析树都存在一个求值顺序，因为它们不允许产生带有回路的依赖图。

产生式	语义规则
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$



两类重要无回路的语法制导定义

- S -属性定义 (S -Attributed Definitions, S -SDD)

- L -属性定义 (L -Attributed Definitions, L -SDD)

- 首先，根据定义它们能够保证每棵注释分析树属性间都不会有循环依赖关系。—— 通用的基于树遍历计算
- 而且，不必显式地构造依赖图，只需要一次深度优先的分析树遍历就能计算出所有属性值。
- 更重要的，它们能够与自顶向下或自底向上的语法分析过程一起高效地实现。—— 一遍语法制导翻译



提纲

5.1 语法制导定义SDD

5.2 S-属性定义与L-属性定义

5.3 语法制导翻译方案SDT

5.4 L-属性定义的自顶向下翻译

5.5 L-属性定义的自底向上翻译

5.2 S -属性定义与 L -属性定义

- 只使用综合属性的 SDD 称为 S 属性的 SDD ，或 S -属性定义、 S - SDD

➤ 例

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) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

- 如果一个 SDD 是 S 属性的，可以按照语法分析树节点的任何自底向上顺序来计算它的各个属性值
- S -属性定义可以在自底向上的语法分析过程中实现

L -属性定义

- L -属性定义(也称为 L 属性的 SDD 或 L - SDD)的
直观含义: 在一个产生式所关联的各属性之间,
依赖图的边可以从左到右, 但不能从右到左
(因此称为 L 属性的, L 是 $Left$ 的首字母)

L -SDD的正式定义

- 一个SDD是 L -属性定义，当且仅当它的每个属性要么是一个综合属性，要么是满足如下条件的继承属性：假设存在一个产生式 $A \rightarrow X_1X_2 \dots X_n$ ，其右部符号 X_i ($1 \leq i \leq n$)的继承属性仅依赖于下列属性：
 - A 的继承属性
 - 产生式中 X_i 左边的符号 X_1, X_2, \dots, X_{i-1} 的属性
 - X_i 本身的属性，但 X_i 的全部属性不能在依赖图中形成环路

每个S-属性定义都是 L -属性定义

L-SDD的正式定义

➤ 一个SDD是L-属性定义，当且仅当它的每个属性要么是一个综合属性，要么是满足如下条件的继承属性：假设存在一个产生式 $A \rightarrow X_1 X_2 \dots X_n$ ，其右部符号 $X_i (1 \leq i \leq n)$ 的继承属性仅依赖于下列属性：

➤ A 的继承属性。 为什么不能依赖 A 的综合属性？

➤ 产生式中 X_i 左边的符号 X_1, X_2, \dots, X_{i-1} 的属性

➤ X_i 本身的属性，但 X_i 的全部属性不能在依赖图中形成环路

L-SDD的正式定义

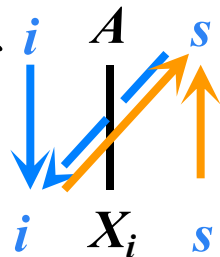
➤ 一个SDD是L-属性定义，当且仅当它的每个属性要么是一个综合属性，要么是满足如下条件的继承属性：假设存在一个产生式 $A \rightarrow X_1 X_2 \dots X_n$ ，其右部符号 $X_i (1 \leq i \leq n)$ 的继承属性仅依赖于下列属性

➤ A 的继承属性

为了避免依赖图出现回路

➤ 产生式中 X_i 左边的符号 X_1, X_2, \dots, X_{i-1} 的属性

➤ X_i 本身的属性，但 X_i 的全部属性不能在依赖图中形成环路



例：L-SDD

继承属性

	产生式	语义规则
(1)	$T \rightarrow F T'$	$\overline{T'.inh} = F.val$ $\overline{T.val} = \overline{T'.syn}$
(2)	$T' \rightarrow * F T_1'$	$\overline{T_1'.inh} = \overline{T'.inh} \times F.val$ $\overline{T'.syn} = \overline{T_1'.syn}$
(3)	$T' \rightarrow \varepsilon$	$\overline{T'.syn} = \overline{T'.inh}$
(4)	$F \rightarrow \text{digit}$	$\overline{F.val} = \overline{\text{digit.lexval}}$

综合属性

非 L 属性的 SDD

➤ 例

继承属性

产生式	语义规则
(1) $A \rightarrow LM$	$L.i = l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
(2) $A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s) \times$ $A.s = f(Q.s)$

综合属性



提纲

5.1 语法制导定义SDD

5.2 S-属性定义与L-属性定义

5.3 语法制导翻译方案SDT

5.4 L-属性定义的自顶向下翻译

5.5 L-属性定义的自底向上翻译

5.3 语法制导翻译方案SDT

- 语法制导翻译方案(SDT)是在产生式右部中嵌入了程序片段(称为语义动作)的CFG

➤ 例

$$\begin{aligned} D &\rightarrow T \{ L.inh = T.type \} L \\ T &\rightarrow \text{int} \{ T.type = \text{int} \} \\ T &\rightarrow \text{float} \{ T.type = \text{float} \} \\ L &\rightarrow \{ L_1.inh = L.inh \} L_1, \text{id} \{ ddType(\text{id.entry}, L.inh) \} \\ L &\rightarrow \text{id} \{ addType(\text{id.entry}, L.inh) \} \end{aligned}$$

5.3 语法制导翻译方案SDT

- 语法制导翻译方案(SDT)是在产生式右部中嵌入了程序片段(称为语义动作)的CFG
- SDT可以看作是SDD的具体实施方案
- 本节主要关注如何使用SDT来实现两类重要的SDD, 因为在这两种情况下, SDT可在语法分析过程中实现
 - 基本文法可以使用LR分析技术, 且SDD是S属性的
 - 基本文法可以使用LL分析技术, 且SDD是L属性的

将S-SDD转换为SDT

➤ 将一个S-SDD转换为SDT的方法：将每个语义动作都放在产生式的最后

➤ 例

S-SDD

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) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

SDT

(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) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

S-属性定义的SDT实现

- 如果一个S-SDD的基本文法可以使用LR分析技术，那么它的SDT可以在LR语法分析过程中实现

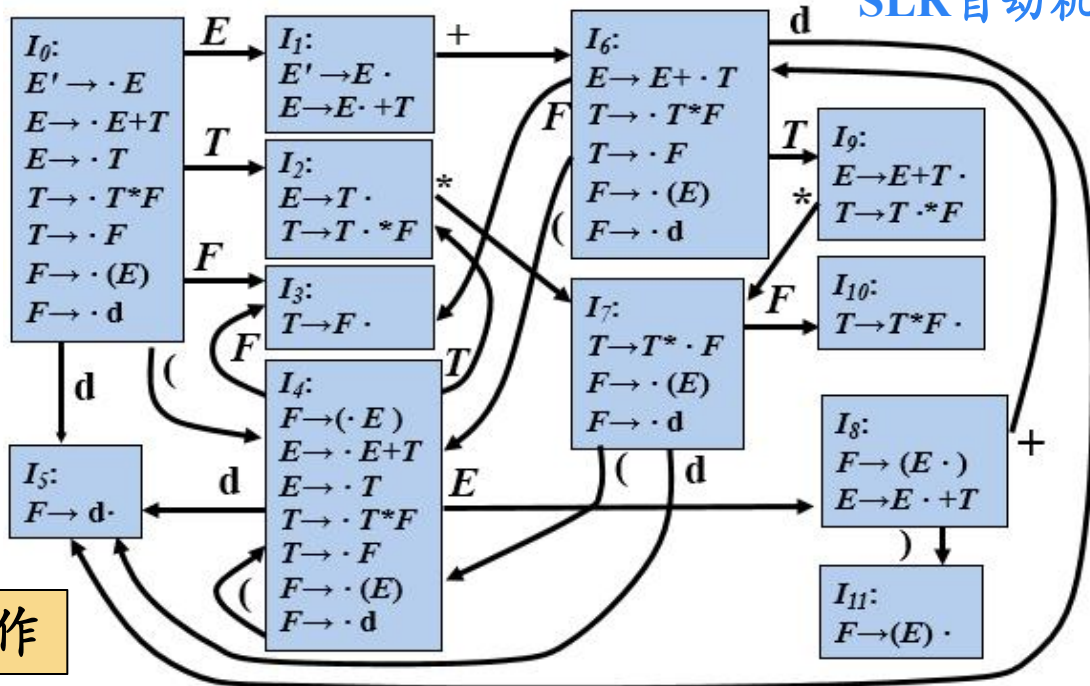
➤ 例

SDT

SLR自动机

- (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) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

当归约发生时执行相应的语义动作



扩展的LR语法分析栈

在分析栈中使用一个附加的域来存放综合属性值

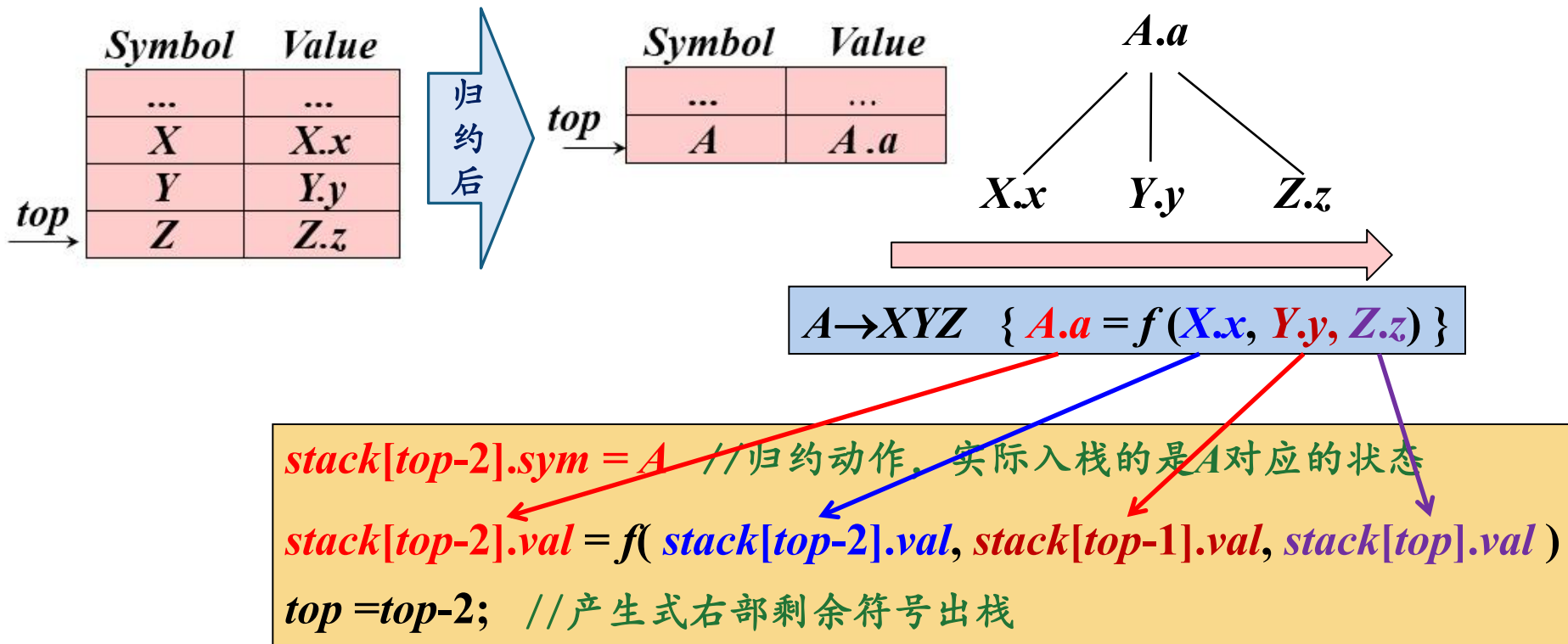
状态/文法符号	综合属性
$S_0 / \$$	
...	...
S_{m-2} / X	$X.x$
S_{m-1} / Y	$Y.y$
S_m / Z	$Z.z$
...	...

$top \rightarrow$

- 若支持多个属性
 - 使栈记录变得足够大
 - 在栈记录中存放指针

将语义动作中的抽象定义式改写成具体可执行的栈操作

stack 数组



例：在自底向上语法分析栈中实现桌面计算器

产生式	语义动作	
(1) $E' \rightarrow E$	$\text{print}(E.val)$	{ $\text{print}(\text{stack}[\text{top}.val]$); }
(2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	{ $\text{stack}[\text{top}-2].val = \text{stack}[\text{top}-2].val + \text{stack}[\text{top}.val]$; $\text{top}=\text{top}-2$; }
(3) $E \rightarrow T$	$E.val = T.val$	// $E.val$ 入栈还在 $T.val$ 的位置，且值相等，无需操作
(4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$	{ $\text{stack}[\text{top}-2].val = \text{stack}[\text{top}-2].val \times \text{stack}[\text{top}.val]$; $\text{top}=\text{top}-2$; }
(5) $T \rightarrow F$	$T.val = F.val$	// $T.val$ 入栈还在 $F.val$ 的位置，且值相等，无需操作
(6) $F \rightarrow (E)$	$F.val = E.val$	{ $\text{stack}[\text{top}-2].val = \text{stack}[\text{top}-1].val$; $\text{top}=\text{top}-2$; }
(7) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$	// $F.val$ 入栈还在 digit.lexval 的位置且值相等无需操作

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

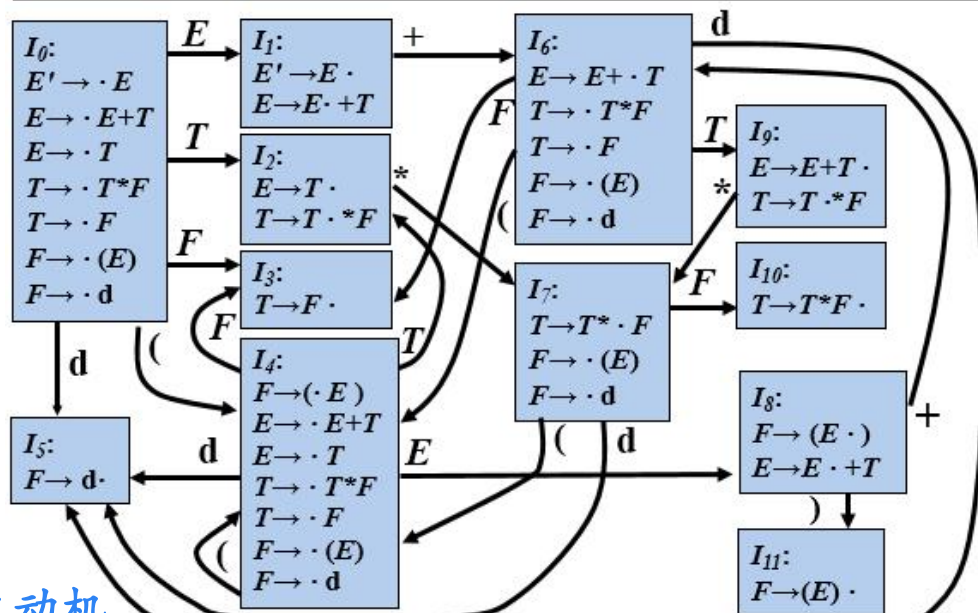
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑

状态 / 符号 属性

0	\$	-
5	d	3



X	$FOLLOW(X)$
E	$)$, $+$, $\$$
T	$)$, $+$, $\$$, $*$
F	$)$, $+$, $\$$, $*$

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

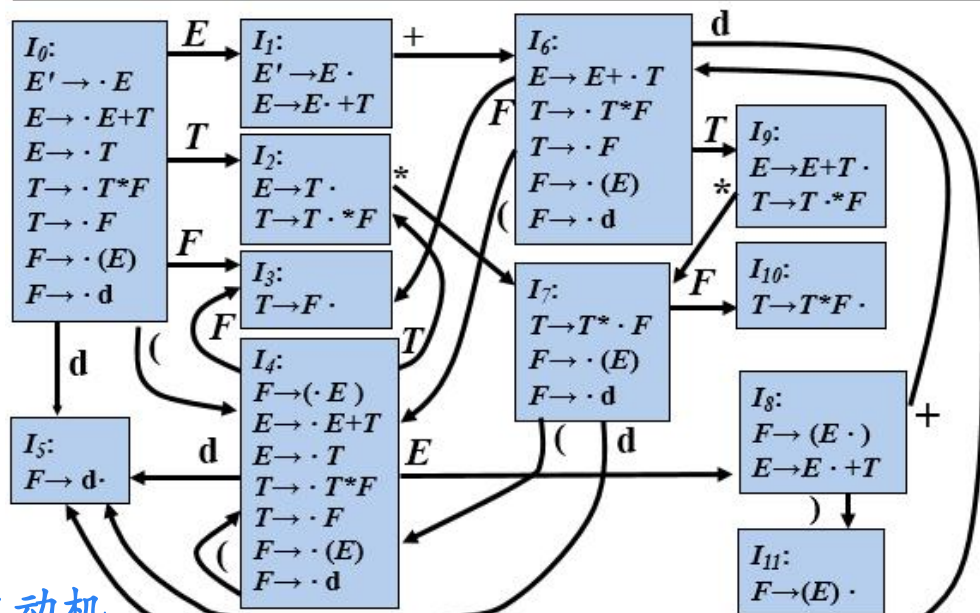
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑

状态 / 符号 属性

0	\$	-
3	F	3



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

(1) $E' \rightarrow E$

{ print (stack[top].val); }

(2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }(3) $E \rightarrow T$ (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }(5) $T \rightarrow F$ (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }(7) $F \rightarrow \text{digit}$

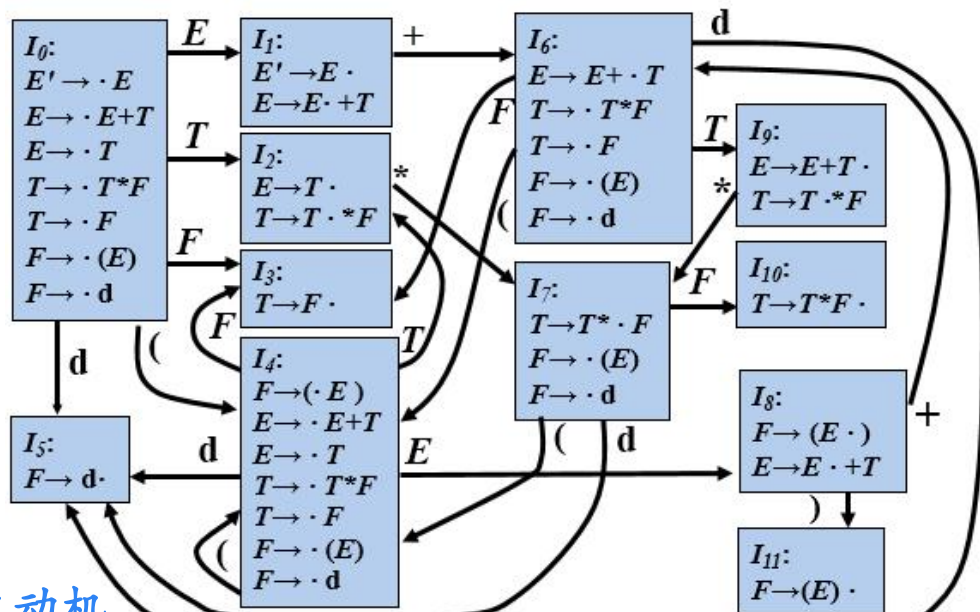
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑

状态 / 符号 属性

0	\$	-
2	T	3
7	*	-
5	d	5



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

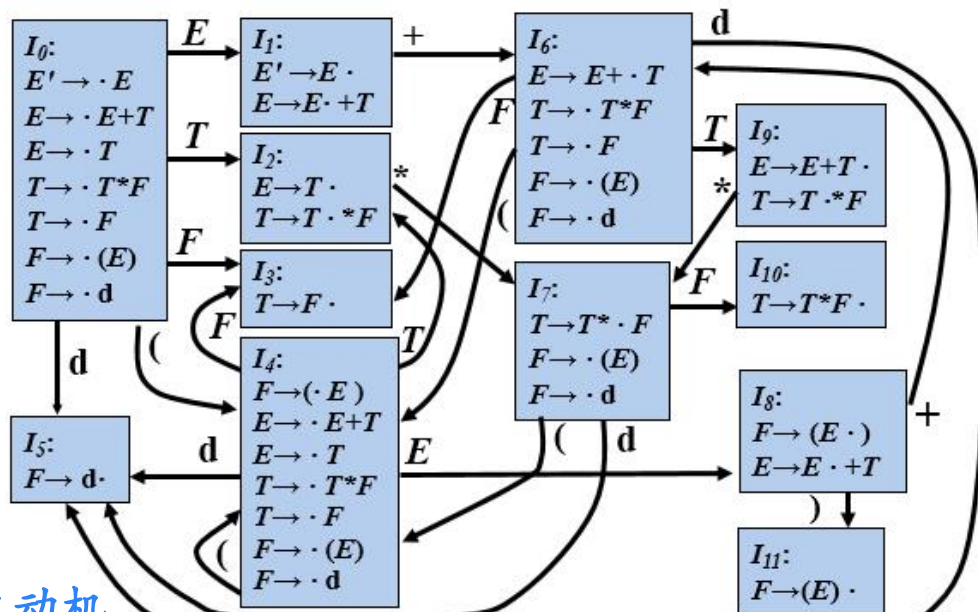
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑

状态 / 符号 属性

0	\$	-
2	T	15
7	*	-
10	F	5



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

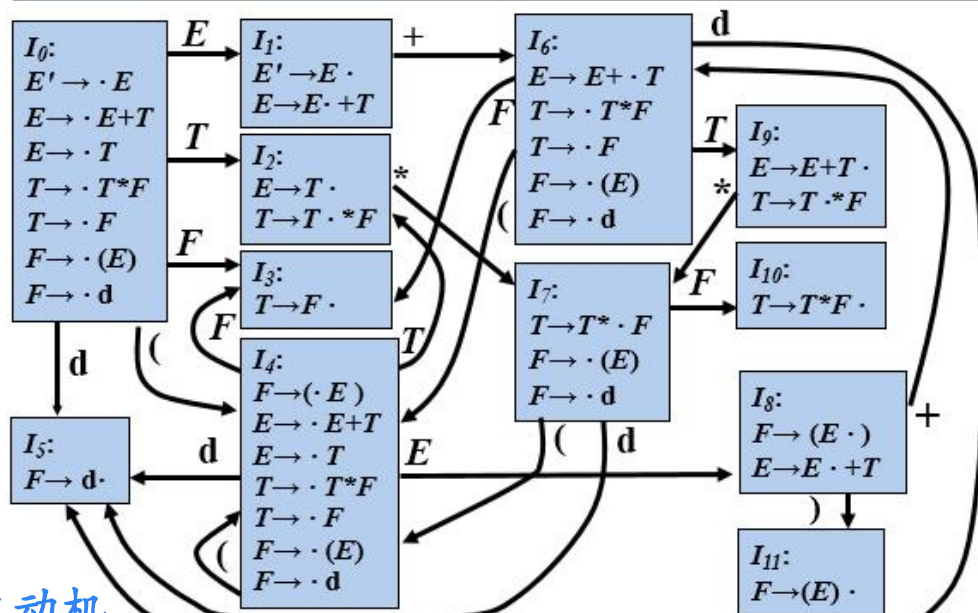
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑

状态 / 符号 属性

0	\$	-
2	T	15



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

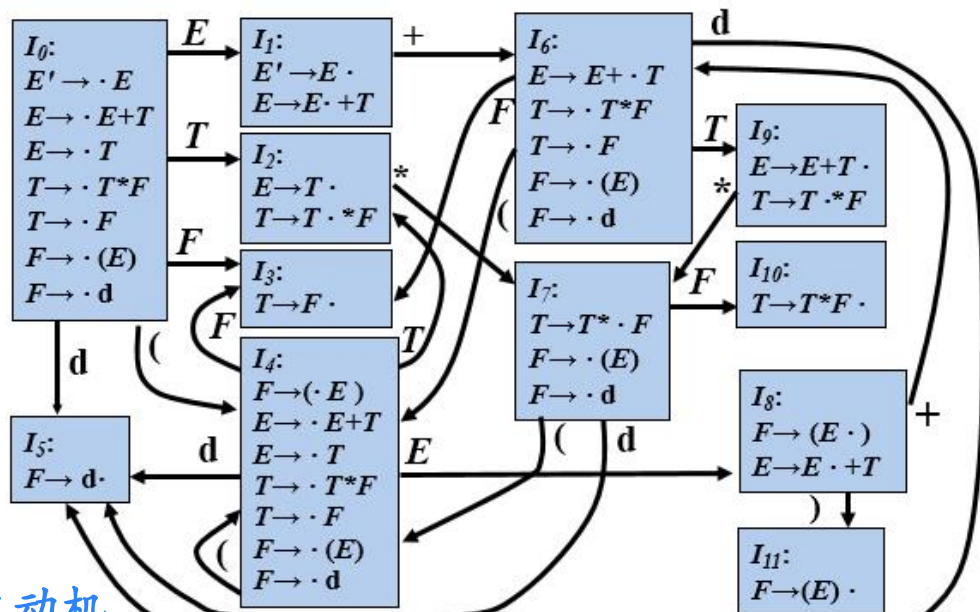
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑↑↑↑

状态 / 符号 属性

0	\$	-
1	E	15
6	+	-
5	d	4



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

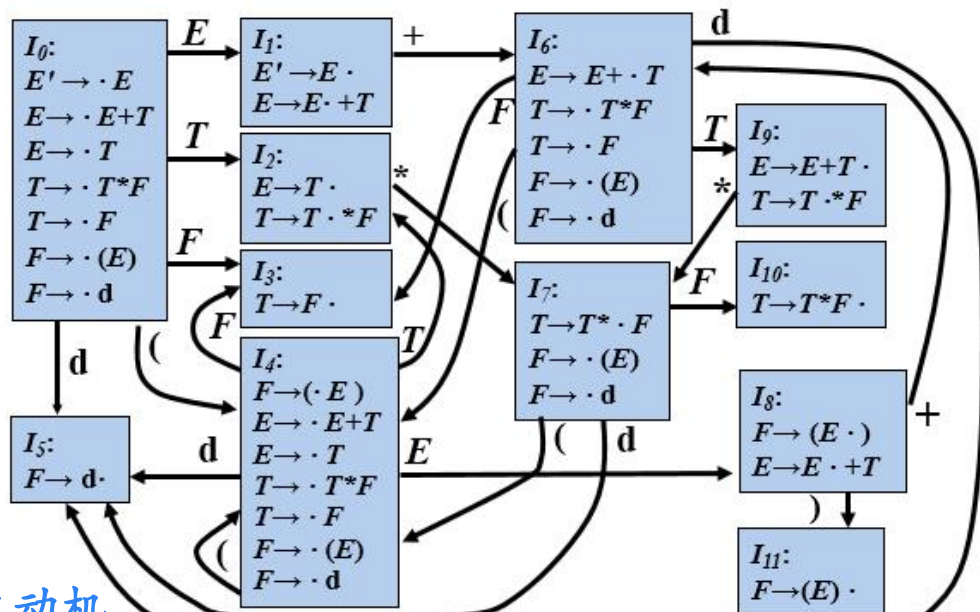
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑↑↑↑

状态 / 符号 属性

0	\$	-
1	E	15
6	+	-
3	F	4



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

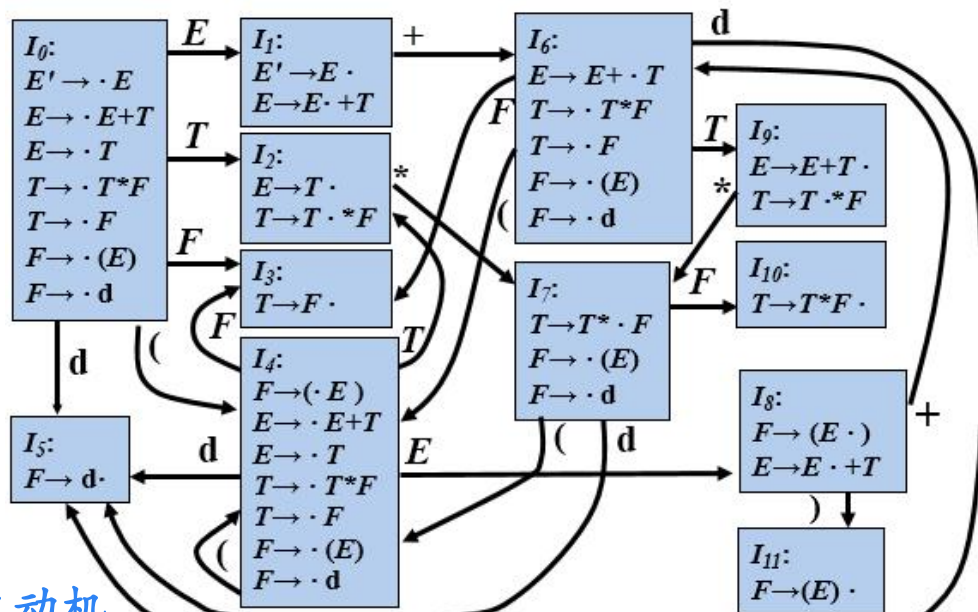
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑↑↑↑

状态 / 符号 属性

0	\$	-
1	E	19
6	+	-
9	T	4



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

产生式

语义动作

- (1) $E' \rightarrow E$ { print (stack[top].val); }
- (2) $E \rightarrow E_1 + T$ { stack[top-2].val = stack[top-2].val + stack[top].val ;
top=top-2; }
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$ { stack[top-2].val = stack[top-2].val \times stack[top].val ;
top=top-2; }
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$ { stack[top-2].val = stack[top-1].val;
top=top-2; }
- (7) $F \rightarrow \text{digit}$

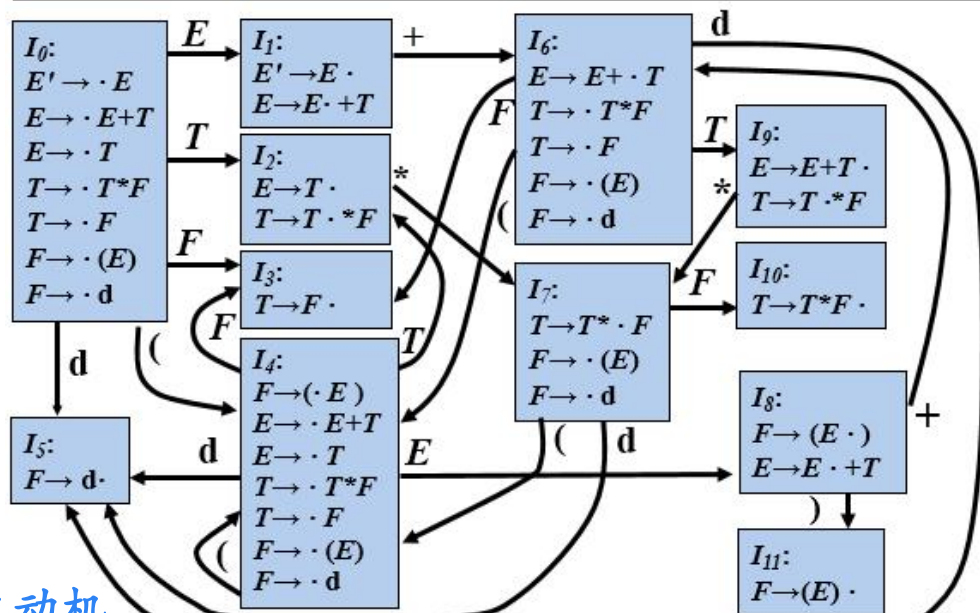
digit* digit+ digit \$

输入: 3*5+4 \$

↑↑↑↑↑↑↑

状态 / 符号 属性

0	\$	-
1	E	19



X	FOLLOW(X)
E), +, \$
T), +, \$, *
F), +, \$, *

SLR 自动机

将 L - SDD 转换为 SDT

➤ 将 L - SDD 转换为 SDT 的规则

- 将计算某个非终结符号 A 的继承属性的动作插入到产生式右部中紧靠在 A 的本次出现之前的位置上
- 将计算一个产生式左部符号的综合属性的动作放置在这个产生式右部的最右端

例

➤ *L-SDD*

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
(4)	$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

?思考题：如果把符号A的继承属性的计算，放在符号A的后面可不可以，为什么？

➤ *SDT*

- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

L-属性定义的SDT实现

➤ 如果一个L-SDD的基本文法可以使用LL分析技术，那么它的SDT可以在LL语法分析过程中实现，经过改造后，可以在LR语法分析中实现。

➤ 例

- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

SELECT (1) = { digit }

SELECT (2) = { * }

SELECT (3) = { \$ }

SELECT (4) = { digit }

L -属性定义的 SDT 实现

- 如果一个 L - SDD 的基本文法可以使用 LL 分析技术，那么它的 SDT 可以在 LL 语法分析过程中实现，经过改造后，可以在 LR 语法分析中实现。
- 在非递归的预测分析过程中进行语义翻译
- 在递归的预测分析过程中进行语义翻译
- 改造文法和 SDT 并在 LR 分析过程中进行语义翻译

练习1

➤ 下面的文法生成了含“小数点”的二进制数：

$$S \rightarrow L_1 . L_2 \mid L \qquad L \rightarrow L_1 B \mid B \qquad B \rightarrow 0 \mid 1$$

(1) 设计一个S属性的SDT来计算S.val，即输入串的十进制数值。比如，串101.101应该被翻译成十进制数5.625。

(2) 参考图5-20，将(1)中的SDT改写为在自底向上语法分析栈中实现的栈操作。

A 3D metallic staircase structure, composed of several rectangular blocks, is positioned on the left side of the slide. The background is a light blue surface covered with a pattern of binary code (0s and 1s) in a darker blue color. The staircase has a modern, industrial look with a brushed metal texture.

提纲

5.1 语法制导定义SDD

5.2 S-属性定义与L-属性定义

5.3 语法制导翻译方案SDT

5.4 L-属性定义的自顶向下翻译

5.5 L-属性定义的自底向上翻译

5.4 L-属性定义的自顶向下翻译

SDT: 桌面计算器, 计算算式的值

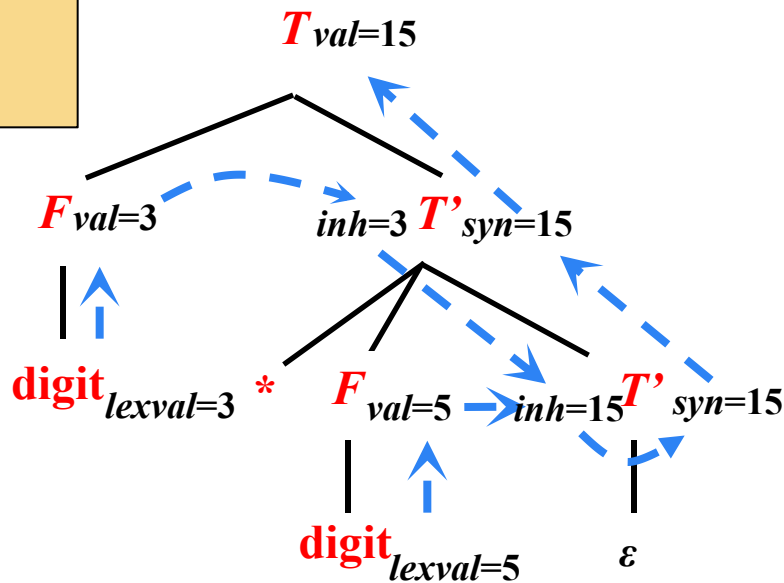
- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入:

3 * 5 \$

digit * digit \$

↑ ↑ ↑ ↑



5.4 L-属性定义的自顶向下翻译

- 5.4.1 在非递归的预测分析过程中进行翻译
- 5.4.2 在递归的预测分析过程中进行翻译

5.4.1 在非递归的预测分析过程中进行翻译

➤ 扩展语法分析栈

动作记录	符号	综合记录	
action	A	Asyn	\$
			-

Symbol

Value

- A的继承属性与符号A存放在一处
- A的综合属性紧挨着符号A记录之下
- 语义动作当作动作符号存放指向语义动作代码的指针，随所在的产生式体一同入栈，出现在栈顶时就执行语义动作，并出栈。

A的综合属性，关联复制属性值的语义动作

A的继承属性，关联复制属性值的语义动作

指向将被执行的语义动作代码的指针，
以及临时存放计算需要的其它属性值

例

- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$



- | | |
|---|--|
| 1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$ | $a_1 : T'.inh = F.val$ |
| 2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$ | $a_2 : T.val = T'.syn$ |
| 3) $T' \rightarrow \varepsilon \{ a_5 \}$ | $a_3 : T_1'.inh = T'.inh \times F.val$ |
| 4) $F \rightarrow \text{digit} \{ a_6 \}$ | $a_4 : T'.syn = T_1'.syn$ |
| | $a_5 : T'.syn = T'.inh$ |
| | $a_6 : F.val = \text{digit.lexval}$ |

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1: T'.inh = F.val$

$a_2: T.val = T'.syn$

$a_3: T_1'.inh = T'.inh \times F.val$

$a_4: T'.syn = T_1'.syn$

$a_5: T'.syn = T'.inh$

$a_6: F.val = \text{digit}.lexval$

输入: 3 * 5 \$



符号记录入栈时，
符号的综合属性记录先入栈。

符号记录出栈时，符号的综合属性记录，不出栈，因还未计算。

T	T_{syn}	\$
	val	

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T_1'.inh = T'.inh \times F.val$

$a_4 : T'.syn = T_1'.syn$

$a_5 : T'.syn = T'.inh$

$a_6 : F.val = \text{digit}.lexval$

输入: 3 * 5 \$
↑

actions			actions		actions		
F	$Fsyn$	$\{a_1\}$	T'	$T'syn$	$\{a_2\}$	$Tsyn$	
	val		inh	syn		val	\$

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T'_1 \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T'_1.inh = T'.inh \times F.val$

$a_4 : T'.syn = T'_1.syn$

$a_5 : T'.syn = T'.inh$

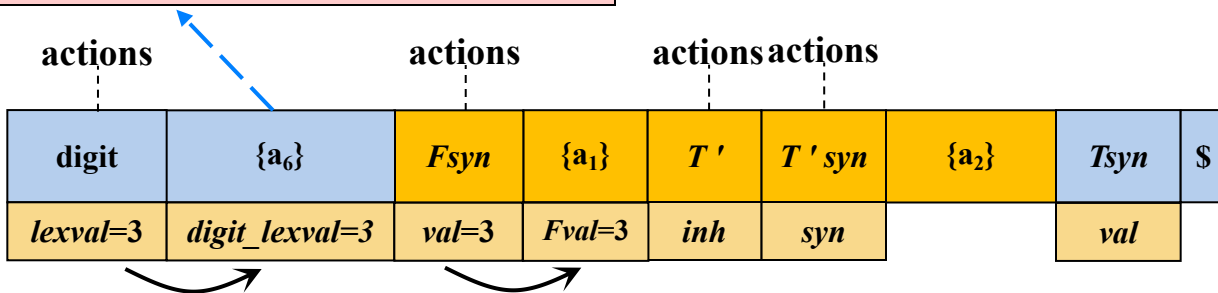
$a_6 : F.val = \text{digit.lexval}$

输入: 3 * 5 \$



➤综合属性记录出栈时,与综合记录关联的语义动作负责将综合属性复制到需要它的栈中位置。

$stack[top-1].val = stack[top].digit_lexval$



例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1: T'.inh = F.val$

$a_2: T.val = T'.syn$

$a_3: T_1'.inh = T'.inh \times F.val$

$a_4: T'.syn = T_1'.syn$

$a_5: T'.syn = T'.inh$

$a_6: F.val = \text{digit.lexval}$

输入: 3 * 5 \$
↑ ↑

当包含继承属性值的符号记录出栈时，与它关联的语义动作要将继承属性值复制到即将入栈的产生式体中需要此值的动作记录中

$stack[top-1].inh = stack[top].Fval$

actions actions

$\{a_1\}$	T'	$T'.syn$	$\{a_2\}$	$Tsyn$	\$
$Fval=3$	$inh=3$	syn		val	

$stack[top+3].T'.inh = stack[top].inh$

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T_1'.inh = T'.inh \times F.val$

$a_4 : T'.syn = T_1'.syn$

$a_5 : T'.syn = T'.inh$

$a_6 : F.val = \text{digit.lexval}$

输入: 3 * 5 \$
↑ ↑ ↑

$stack[top+3].T'.inh = stack[top].inh$

*	F	Fsyn	{a ₃ }	T ₁ '	T ₁ '.syn	{a ₄ }	T' syn	{a ₂ }	Tsyn	\$
		val	T' inh=3	inh	syn		syn		val	



例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T_1'.inh = T'.inh \times F.val$

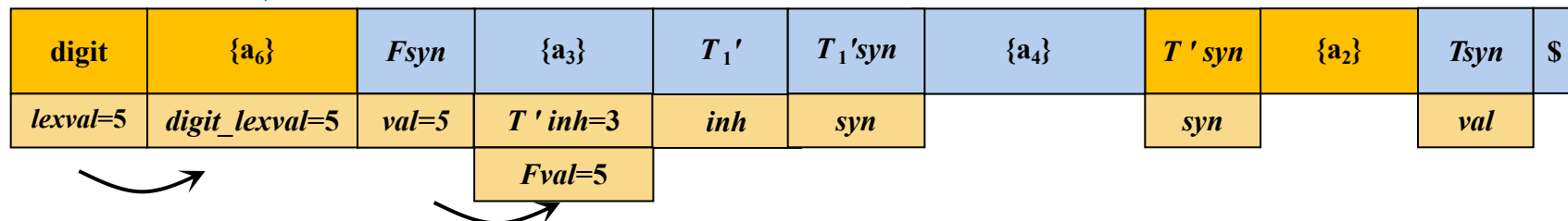
$a_4 : T'.syn = T_1'.syn$

$a_5 : T'.syn = T'.inh$

$a_6 : F.val = \text{digit}.lexval$

输入: 3 * 5 \$
↑ ↑ ↑ ↑

$stack[top-1].val = stack[top].digit_lexval$



例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T_1'.inh = T'.inh \times F.val$

$a_4 : T'.syn = T_1'.syn$

$a_5 : T'.syn = T'.inh$

$a_6 : F.val = \text{digit.lexval}$

输入: 3 * 5 \$



➤T'要替换为空，T'出栈前需将其继承属性复制到将要入栈的a5动作记录中保存，a5刚好在T'的位置。

$stack[top-1].inh = stack[top].T'.inh \times stack[top].Fval$

{a ₃ }	T ₁ '	T ₁ '.syn	{a ₄ }	T'.syn	{a ₂ }	Tsyn	\$
T'.inh=3	inh=15	syn		syn		val	
Fval=5							

$stack[top].T'.inh = stack[top].inh;$

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1: T'.inh = F.val$

$a_2: T.val = T'.syn$

$a_3: T_1'.inh = T'.inh \times F.val$

$a_4: T'.syn = T_1'.syn$

$a_5: T'.syn = T'.inh$

$a_6: F.val = \text{digit.lexval}$

输入: 3 * 5 \$
↑ ↑ ↑ ↑

$stack[top-1].syn = stack[top].T_1'.inh$

$\{a_5\}$	$T_1'.syn$	$\{a_4\}$	$T'.syn$	$\{a_2\}$	$T.syn$	\$
$T_1'.inh=15$	$syn=15$	$T_1'.syn=15$	syn		val	



例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1: T'.inh = F.val$

$a_2: T.val = T'.syn$

$a_3: T_1'.inh = T'.inh \times F.val$

$a_4: T'.syn = T_1'.syn$

$a_5: T'.syn = T'.inh$

$a_6: F.val = \text{digit.lexval}$

输入: 3 * 5 \$
↑ ↑ ↑ ↑

$stack[top-1].syn = stack[top].T_1'.syn$

$\{a_4\}$	$T'.syn$	$\{a_2\}$	$T.syn$	\$
$T_1'.syn=15$	$syn=15$	$T'.syn=15$	val	

例

SDT

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1 : T'.inh = F.val$

$a_2 : T.val = T'.syn$

$a_3 : T_1'.inh = T'.inh \times F.val$

$a_4 : T'.syn = T_1'.syn$

$a_5 : T'.syn = T'.inh$

$a_6 : F.val = \text{digit}.lexval$

输入: 3 * 5 \$
↑ ↑ ↑ ↑

$stack[top-1].val = stack[top].T'syn$

$\{a_2\}$	$T'syn$	\$
$T' syn=15$	$val=15$	

分析栈中的每一个记录都关联着语义动作

- **综合记录**出栈时，要将**综合属性值**复制给栈中需要此值的语义动作记录。
 - 根据符号所在的产生式关联的语义动作确定需要此值的语义动作记录在栈中的位置。
- **符号本身的记录**出栈（即推导）时，如果其含有继承属性，通常要将**继承属性值**复制给即将入栈的产生式体中需要此值的语义动作记录。
 - 根据推导选择的产生式来确定需要此值的语义动作记录未来在栈顶方向出现的位置。

例

- 1) $T \rightarrow F \{a_1\} T' \{a_2\}$
- 2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$
- 3) $T' \rightarrow \varepsilon \{a_5\}$
- 4) $F \rightarrow \text{digit} \{a_6\}$

- $a_1: T'.inh = F.val$
- $a_2: T.val = T'.syn$
- $a_3: T_1'.inh = T'.inh \times F.val$
- $a_4: T'.syn = T_1'.syn$
- $a_5: T'.syn = T'.inh$
- $a_6: F.val = \text{digit.lexval}$

1) $T \rightarrow F \{a_1: T'.inh = F.val\} T' \{a_2: T.val = T'.syn\}$

符号	属性	执行代码	动作
F			
$Fsyn$	val	$stack[top-1].Fval = stack[top].val; top=top-1;$	复制到{a1}
a_1	$Fval$	$stack[top-1].inh = stack[top].Fval; top=top-1;$	
T'	inh	根据当前输入符号选择产生式进行推导 若选 2): $stack[top+3].T'inh = stack[top].inh; top=top+6;$ 若选 3): $stack[top].T'inh = stack[top].inh;$	复制到{a3} 复制到{a5}
$T'syn$	syn	$stack[top-1].T'syn = stack[top].syn; top=top-1;$	复制到{a2}
a_2	$T'syn$	$stack[top-1].val = stack[top].T'syn; top=top-1;$	

例

$$1) T \rightarrow F \{a_1\} T' \{a_2\}$$

$$2) T' \rightarrow *F \{a_3\} T_1' \{a_4\}$$

$$3) T' \rightarrow \varepsilon \{a_5\}$$

$$4) F \rightarrow \text{digit} \{a_6\}$$

$$a_1: T'.inh = F.val$$

$$a_2: T.val = T'.syn$$

$$a_3: T_1'.inh = T'.inh \times F.val$$

$$a_4: T'.syn = T_1'.syn$$

$$a_5: T'.syn = T'.inh$$

$$a_6: F.val = \text{digit.lexval}$$

$$2) T' \rightarrow *F\{a_3: T_1'.inh = T'.inh \times F.val\} T_1'\{a_4: T'.syn = T_1'.syn\}$$

符号	属性	执行代码	动作
*			
F			
Fsyn	val	stack[top-1].Fval = stack[top].val; top=top-1;	复制到{a3}
a ₃	T'.inh; Fval	stack[top-1].inh = stack[top].T'.inh × stack[top].Fval; top=top-1;	
T ₁ '	inh	根据当前输入符号选择产生式进行推导 若选2): stack[top+3].T'.inh = stack[top].inh; top=top+6; 若选3): stack[top].T'.inh = stack[top].inh;	复制到{a3} 复制到{a5}
T ₁ '.syn	syn	stack[top-1].T ₁ '.syn = stack[top].syn; top=top-1;	复制到{a4}
a ₄	T ₁ '.syn	stack[top-1].syn = stack[top].T ₁ '.syn; top=top-1;	

例

1) $T \rightarrow F \{a_1\} T' \{a_2\}$

2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$

3) $T' \rightarrow \varepsilon \{a_5\}$

4) $F \rightarrow \text{digit} \{a_6\}$

$a_1: T'.inh = F.val$

$a_2: T.val = T'.syn$

$a_3: T_1'.inh = T'.inh \times F.val$

$a_4: T'.syn = T_1'.syn$

$a_5: T'.syn = T'.inh$

$a_6: F.val = \text{digit}.lexval$

3) $T' \rightarrow \varepsilon \{a_5: T'.syn = T'.inh\}$

符号	属性	执行代码
a_5	$T'.inh$	$stack[top-1].syn = stack[top].T'.inh;$ $top=top-1;$

例

- 1) $T \rightarrow F \{a_1\} T' \{a_2\}$
- 2) $T' \rightarrow *F \{a_3\} T_1' \{a_4\}$
- 3) $T' \rightarrow \varepsilon \{a_5\}$
- 4) $F \rightarrow \text{digit} \{a_6\}$

- $a_1: T'.inh = F.val$
 $a_2: T.val = T'.syn$
 $a_3: T_1'.inh = T'.inh \times F.val$
 $a_4: T'.syn = T_1'.syn$
 $a_5: T'.syn = T'.inh$
 $a_6: F.val = \text{digit.lexval}$

4) $F \rightarrow \text{digit} \{a_6: F.val = \text{digit.lexval}\}$

符号	属性	执行代码	动作
digit	<i>lexval</i>	<i>stack[top-1].digitlexval = stack[top].lexval;</i> <i>top=top-1;</i>	复制到{a6}
a_6	<i>digitlexval</i>	<i>stack[top-1].val = stack[top].digitlexval;</i> <i>top=top-1;</i>	

5.4.2 在递归的预测分析过程中进行翻译

为每个非终结符 A 构造一个函数， A 的每个继承属性对应该函数的一个形参，函数的返回值是 A 的综合属性值

► 例

SDT

1) $T \rightarrow F \{ T'.inh = F.val \} T'$
 $\{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1'$
 $\{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

$\text{FOLLOW}(T') = \{\$, \}$

对于每个动作，将其代码复制到语法分析器，并把对属性的引用改为对相应变量的引用

```
 $T'.syn$   $T'(token, T'.inh)$ 
{
   $D: Fval, T_1'.inh, T_1'.syn$ ;
  if token = "*" then
  {
    Getnext(token);
     $Fval = F(token)$ ;
     $T_1'.inh = T'.inh \times Fval$ ;
     $T_1'.syn = T_1'(token, T_1'.inh)$ ;
     $T'.syn = T_1'.syn$ ;
    return  $T'.syn$ ;
  }
  else if token = "$" then
  {
     $T'.syn = T'.inh$ ;
    return  $T'.syn$ ;
  }
  else Error;
}
```

5.4.2 在递归的预测分析过程中进行翻译

➤ 例

SDT

1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1'$
 $\{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

Tval T(token)

{

D: Fval, T'inh, T'syn;

if token ≠ “digit” then Error;

Fval = F(token);

T'inh = Fval;

T'syn = T' (token, T'inh);

Tval = T'syn;

return Tval;

}

5.4.2 在递归的预测分析过程中进行翻译

➤ 例

SDT

1) $T \rightarrow F \{ T'.inh = F.val \} T'$
 $\{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1'$
 $\{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

Fval $F(token)$

{

 if $token \neq \text{“digit”}$ then Error;

Fval = $token.lexval$;

 Getnext(token);

 return *Fval*;

}

5.4.2 在递归的预测分析过程中进行翻译

► 例

SDT

1) $T \rightarrow F \{ T'.inh = F.val \} T'$

$\{ T.val = T'.syn \}$

2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1'$

$\{ T'.syn = T_1'.syn \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

Descent()

{

D: Tval;

Getnext(token);

Tval = T(token);

if token ≠ “\$” then Error;

return ;

}

算法

- 为每个非终结符 A 构造一个函数， A 的每个继承属性对应该函数的一个形参，函数的返回值是 A 的综合属性值。对出现在 A 产生式中的每个文法符号的每个属性都设置一个局部变量
- 非终结符 A 的代码根据当前的输入决定使用哪个产生式

算法 (续)

- 与每个产生式有关的代码执行如下动作：从左到右考虑产生式右部的词法单元、非终结符及语义动作
- 对于带有综合属性 x 的词法单元 X ，把 x 的值保存在局部变量 $X.x$ 中；然后产生一个匹配 X 的调用，并继续输入
- 对于非终结符 B ，产生一个右部带有函数调用的赋值语句 $c := B(b_1, b_2, \dots, b_k)$ ，其中， b_1, b_2, \dots, b_k 是代表 B 的继承属性的变量， c 是代表 B 的综合属性的变量
- 对于每个语义动作，将其代码复制到语法分析器，并把对属性的引用改为对相应变量的引用

练习2

下面的文法生成了含“小数点”的二进制数：

$$S \rightarrow L_1 . L_2 \mid L \qquad L \rightarrow L_1 B \mid B \qquad B \rightarrow 0 \mid 1$$

消除左递归和左公因子后，为LL(1) 文法：

$$\begin{aligned} S &\rightarrow LS' & S' &\rightarrow .L \mid \varepsilon & L &\rightarrow BL' \\ L' &\rightarrow BL' \mid \varepsilon & B &\rightarrow 0 \mid 1 \end{aligned}$$

请为其设计L属性SDD，并转换为SDT，最后写出SDT在自顶向下分析中的栈操作

(参考第5章PPT73-76页)



提纲

5.1 语法制导定义SDD

5.2 S-属性定义与L-属性定义

5.3 语法制导翻译方案SDT

5.4 L-属性定义的自顶向下翻译

5.5 L-属性定义的自底向上翻译

5.5 L -属性定义的自底向上翻译

- L -SDD无法直接在 LR 分析中同时计算，继承属性的计算需延迟到整个分析完成时才可计算。
- 给定一个以 LL 文法为基础的 L -SDD，可以修改这个文法和 L -SDD为类 S -SDD，并在 LR 语法分析过程中计算这个新文法之上的SDD
- 可以证明， LL 文法经过这样的修改之后一定是 LR 文法。
- 但 LR 文法经过这样的改造后不能保证还是 LR 文法。

修改文法和L-SDD

Step1: 先构造
L-SDD的SDT

- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

Step2: 使用互不相同的标记非终结符
(Marker Nonterminal)
代替内嵌的语义动作

Step3: 为标记非终结符构造空产生式,
并将内嵌语义动作改造后置于末尾。

➤ 使用标记非终结符的继承属性复制依赖属性值; 使用综合属性计算并保存右侧符号的继承属性;





- 1) $T \rightarrow F M T' \{ T.val = T'.syn \}$
 $M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$
- 2) $T' \rightarrow *F N T_1' \{ T'.syn = T_1'.syn \}$
 $N \rightarrow \varepsilon \{ N.i1 = T'.inh;$
 $N.i2 = F.val;$
 $N.s = N.i1 \times N.i2 \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit}.lexval \}$

例

- 1) $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$
- 2) $T' \rightarrow *F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

互不相同的
标记非终结符
(Marker Nonterminal)

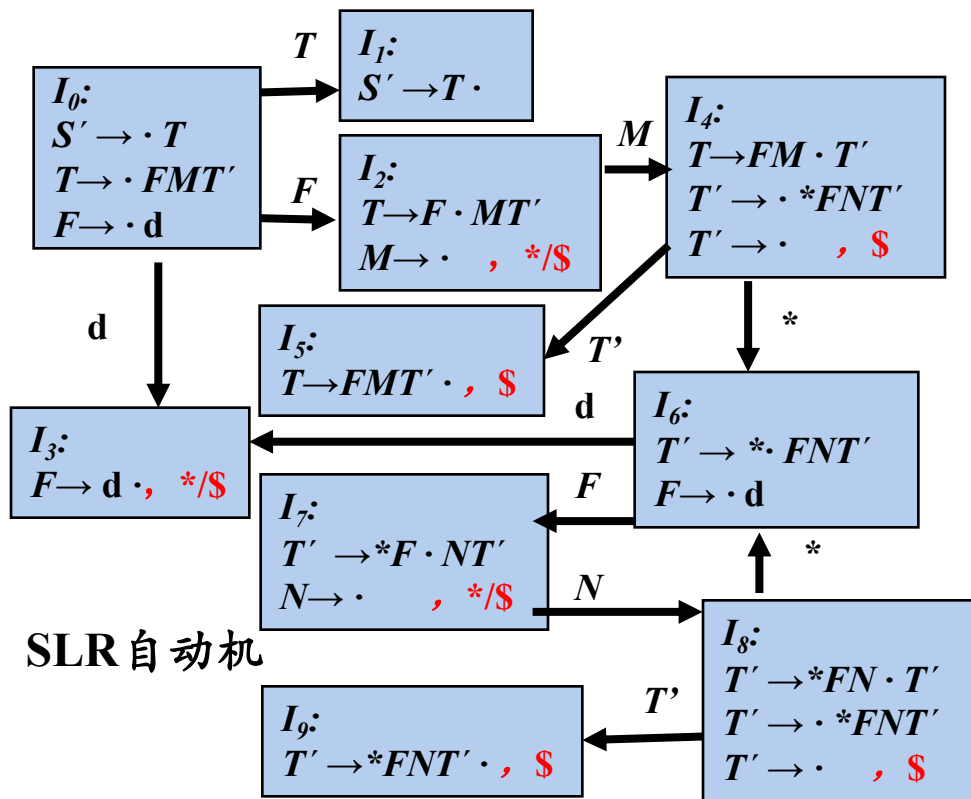
- 
- 
- 1) $T \rightarrow F \mathbf{M} T' \{ T.val = T'.syn \}$
 $\mathbf{M} \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$
 - 2) $T' \rightarrow *F \mathbf{N} T_1' \{ T'.syn = T_1'.syn \}$
 $\mathbf{N} \rightarrow \varepsilon \{ N.i1 = T'.inh;$
 $\quad N.i2 = F.val;$
 $\quad N.s = N.i1 \times N.i2 \}$
 - 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
 - 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

修改后的SDT，所有语义动作都位于产生式末尾

如何保证正确访问未出现在该产生式中的符号的属性？

F和N在产生式体中的相对位置保证：归约出M或N时，栈底一定已归约出F

例



1) $T \rightarrow F M T' \{ T.val = T'.syn \}$

$M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$

2) $T' \rightarrow * F N T'_1 \{ T'.syn = T'_1.syn \}$

$N \rightarrow \varepsilon \{ N.i1 = T'.inh;$

$N.i2 = F.val;$

$N.s = N.i1 \times N.i2 \}$

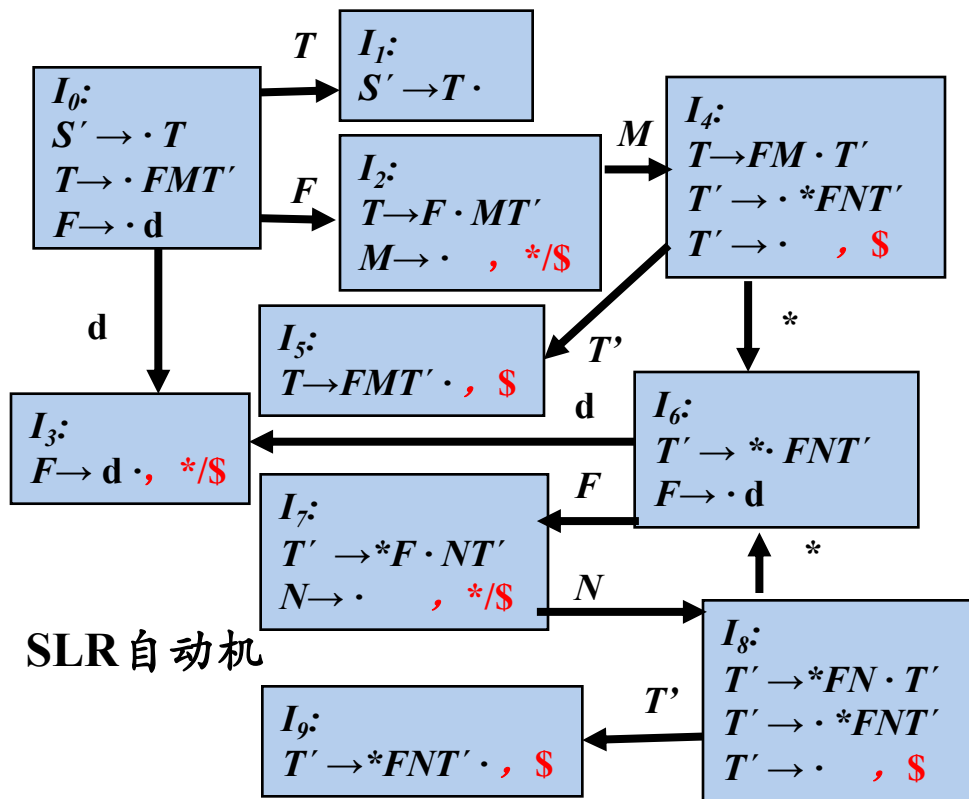
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入: 3 * 5 \$
↑ ↑

0	3
\$	d
	3

例



1) $T \rightarrow F M T' \{ T.val = T'.syn \}$

$M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$

2) $T' \rightarrow * F N T_1' \{ T'.syn = T_1'.syn \}$

$N \rightarrow \varepsilon \{ N.i1 = T'.inh;$

$N.i2 = F.val;$

$N.s = N.i1 \times N.i2 \}$

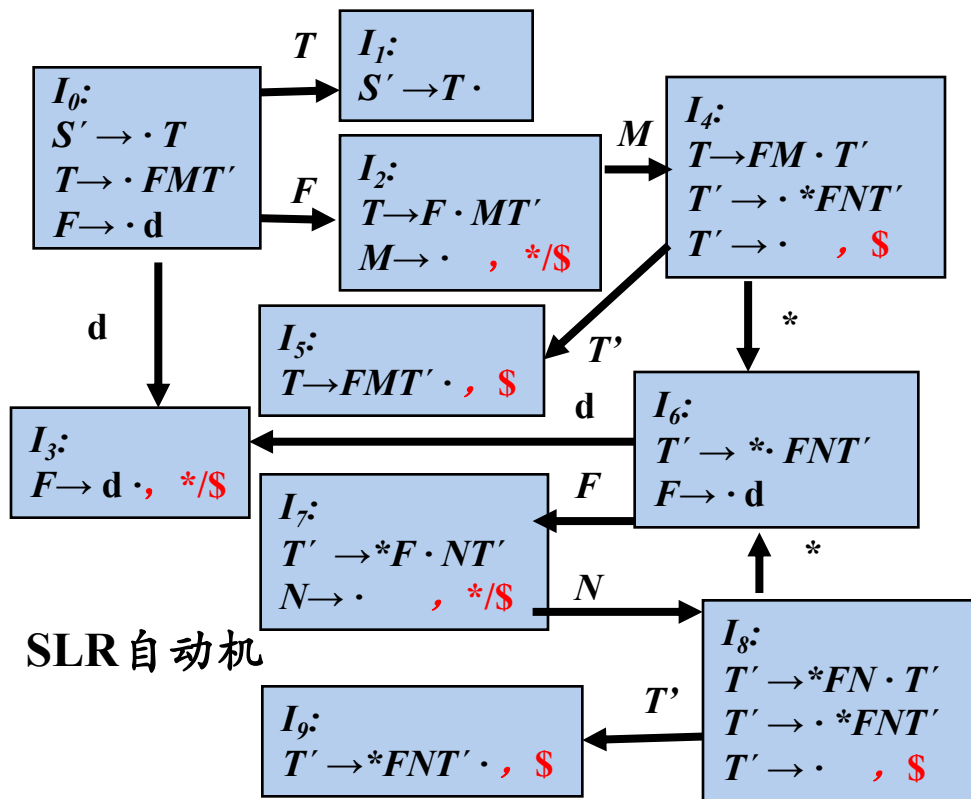
3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入: 3 * 5 \$
 ↑ ↑ ↑

0	2	4	6	3
\$	F	M	*	d
	3	$T'.inh=3$		5

例



1) $T \rightarrow F M T' \{ T.val = T'.syn \}$

$M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$

2) $T' \rightarrow * F N T_1' \{ T'.syn = T_1'.syn \}$

$N \rightarrow \varepsilon \{ N.i1 = T'.inh;$

$N.i2 = F.val;$

$N.s = N.i1 \times N.i2 \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

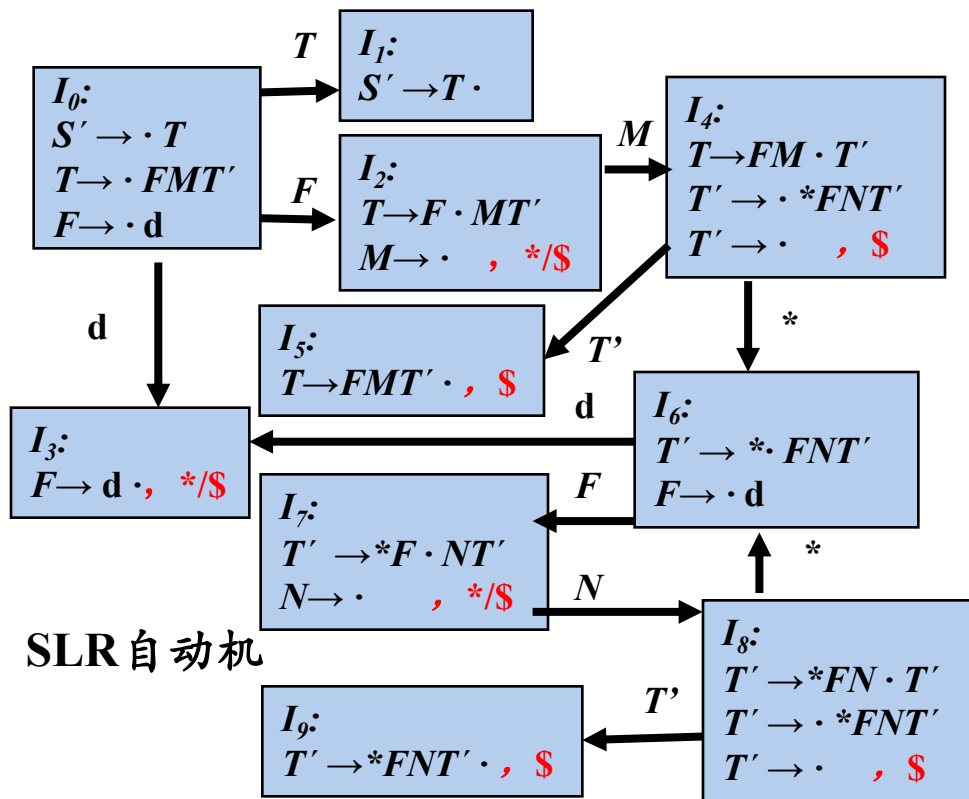
4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入: 3 * 5 \$



0	2	4	6	7	8	9
\$	F	M	*	F	N	T'
	3	$T'.inh=3$		5	$T_1'.inh=15$	$syn=15$

例



1) $T \rightarrow F M T' \{ T.val = T'.syn \}$

$M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$

2) $T' \rightarrow * F N T'_1 \{ T'.syn = T'_1.syn \}$

$N \rightarrow \varepsilon \{ N.i1 = T'.inh; \}$

$N.i2 = F.val; \}$

$N.s = N.i1 \times N.i2 \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

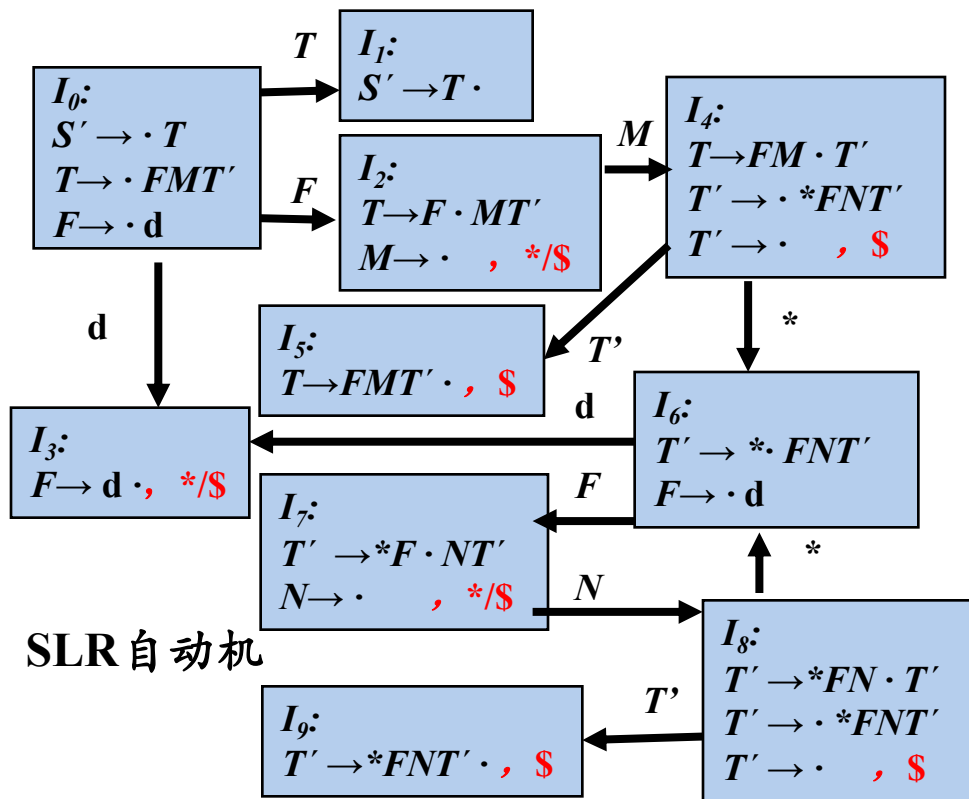
4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入: 3 * 5 \$



0	2	4	5
\$	F	M	T'
	3	$T'.inh=3$	$syn=15$

例



1) $T \rightarrow F M T' \{ T.val = T'.syn \}$

$M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$

2) $T' \rightarrow * F N T'_1 \{ T'.syn = T'_1.syn \}$

$N \rightarrow \varepsilon \{ N.i1 = T'.inh; \}$

$N.i2 = F.val; \}$

$N.s = N.i1 \times N.i2 \}$

3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$

4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

输入: 3 * 5 \$



0	1
\$	T
	val=15

将语义动作改写为可执行的栈操作

- 1) $T \rightarrow F M T' \{ T.val = T'.syn \}$
 $M \rightarrow \varepsilon \{ M.i = F.val; M.s = M.i \}$
- 2) $T' \rightarrow^* F N T_1' \{ T'.syn = T_1'.syn \}$
 $N \rightarrow \varepsilon \{ N.i1 = T'.inh;$
 $N.i2 = F.val;$
 $N.s = N.i1 \times N.i2 \}$
- 3) $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$
- 4) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

- 1) $T \rightarrow F M T' \{ \text{stack}[top-2].val = \text{stack}[top].syn; top = top-2; \}$
 $M \rightarrow \varepsilon \{ \text{stack}[top+1].T'.inh = \text{stack}[top].val; top = top+1; \}$
- 2) $T' \rightarrow^* F N T_1' \{ \text{stack}[top-3].syn = \text{stack}[top].syn; top = top-3; \}$
 $N \rightarrow \varepsilon \{ \text{stack}[top+1].T'.inh = \text{stack}[top-2].T'.inh \times \text{stack}[top].val; top = top+1; \}$
- 3) $T' \rightarrow \varepsilon \{ \text{stack}[top+1].syn = \text{stack}[top].T'.inh; top = top+1; \}$
- 4) $F \rightarrow \text{digit} \{ \text{stack}[top].val = \text{stack}[top].lexval; \}$

给定一个以 LL 文法为基础的 L -属性定义, 可以修改这个文法, 并在 LR 语法分析过程中计算这个新文法之上的 SDD

- 首先构造 SDT , 在各个非终结符之前放置语义动作来计算它的继承属性, 并在产生式后端放置语义动作计算综合属性
- 对每个内嵌的语义动作, 向文法中引入一个标记非终结符来替换它。每个这样的位置都有一个不同的标记, 并且对于任意一个标记 M 都有一个产生式 $M \rightarrow \epsilon$
- 如果标记非终结符 M 在某个产生式 $A \rightarrow \alpha\{a\}\beta$ 中替换了语义动作 a , 对 a 进行修改得到 a' , 并且将 a' 关联到 $M \rightarrow \epsilon$ 上。动作 a'
 - (a) 将动作 a 需要的 A 或 α 中符号的任何属性作为 M 的继承属性进行复制
 - (b) 按照 a 中的方法计算各个属性, 但是将计算得到的这些属性作为 M 的综合属性

练习3

下面的文法生成了含“小数点”的二进制数：

$$S \rightarrow L_1 . L_2 \mid L \qquad L \rightarrow L_1 B \mid B \qquad B \rightarrow 0 \mid 1$$

消除左递归和左公因子后，为LL(1) 文法：

$$\begin{aligned} S &\rightarrow LS' & S' &\rightarrow .L \mid \varepsilon & L &\rightarrow BL' \\ L' &\rightarrow BL' \mid \varepsilon & B &\rightarrow 0 \mid 1 \end{aligned}$$

请为其设计L属性SDD，并转换为SDT.

将此L属性的SDT改造成可以在自底向上语法分析中实现的SDT。

本章小结

- 语法制导定义
- S-属性定义与L-属性定义
- 语法制导翻译方案SDT
- L-属性定义的自顶向下翻译
 - 在非递归的预测分析过程中进行翻译
 - 在递归的预测分析过程中进行翻译
- L-属性定义的自底向上翻译

本章小结

- 任何基于LR文法的S-SDD都能够在自底向上的语法分析中实现
- 任何基于LL文法的L-SDD都能够在自顶向下的语法分析中实现
- 任何基于LL文法的L-SDD经过改造，都能够在自底向上的语法分析中实现
- 可以证明，任何LL文法经过教材5.5.4中方法改造后为LR文法。



结束

