编译原理

第五章

语法制导翻译

哈尔滨工业大学　陈鄞　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语义信息并进行静态语义一致性和完整性的检查。
  - 语义信息：变量的类型、值和地址等
  - 静态检查：
    - 类型检查
    - 控制流检查
    - 一致性检查
    - ……

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

- 如何表示语义信息？
  - 为*CFG*中的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
- 如何计算语义属性？
  - 文法符号的语义属性值是用与文法符号所在产生式（语法规则）相关联的语义规则来计算的
  - 对于给定的输入串*x* ，构建*x*的语法分析树，并利用与产生式（语法规则）相关联的语义规则来计算分析树中各结点对应的语义属性值

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
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow$ int	$T.type = int$
$T \rightarrow$ float	$T.type = float$
$L \rightarrow L_1$, id	$L_1.inh = L.inh$ 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

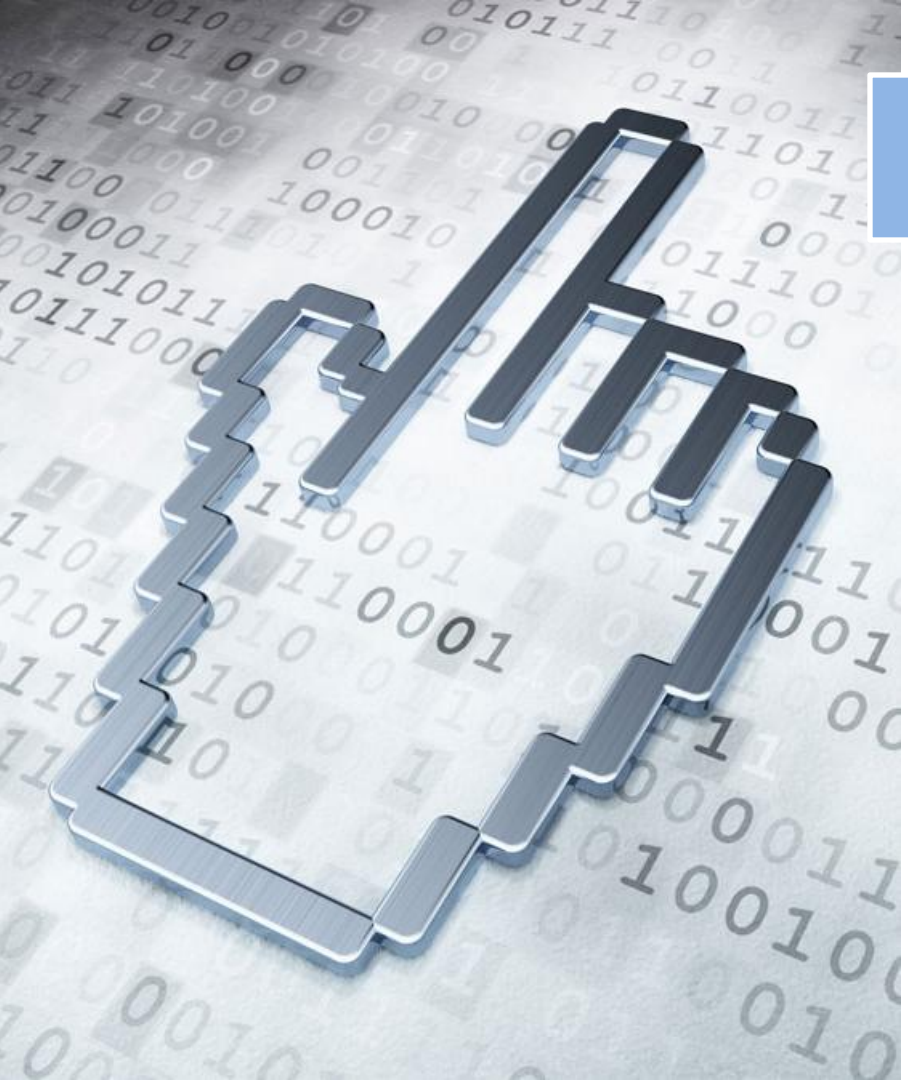
$D \rightarrow T \{ L.inh = T.type \} L$

$T \rightarrow \text{int} \{ T.type = int \}$

$T \rightarrow \text{float} \{ T.type = float \}$

$L \rightarrow \{ L_1.inh = L.inh \} L_1, \text{id} \{addType(id.entry, L.inh)\}$

$L \rightarrow \text{id} \{addType(id.entry, L.inh)\}$



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

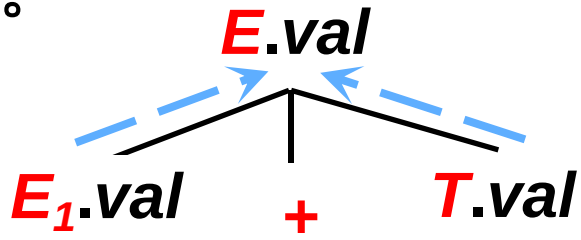
5.1 语法制导定义SDD

- 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
$E \quad E_1 + T$	$E.val = E_1.val + T.val$



- 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*

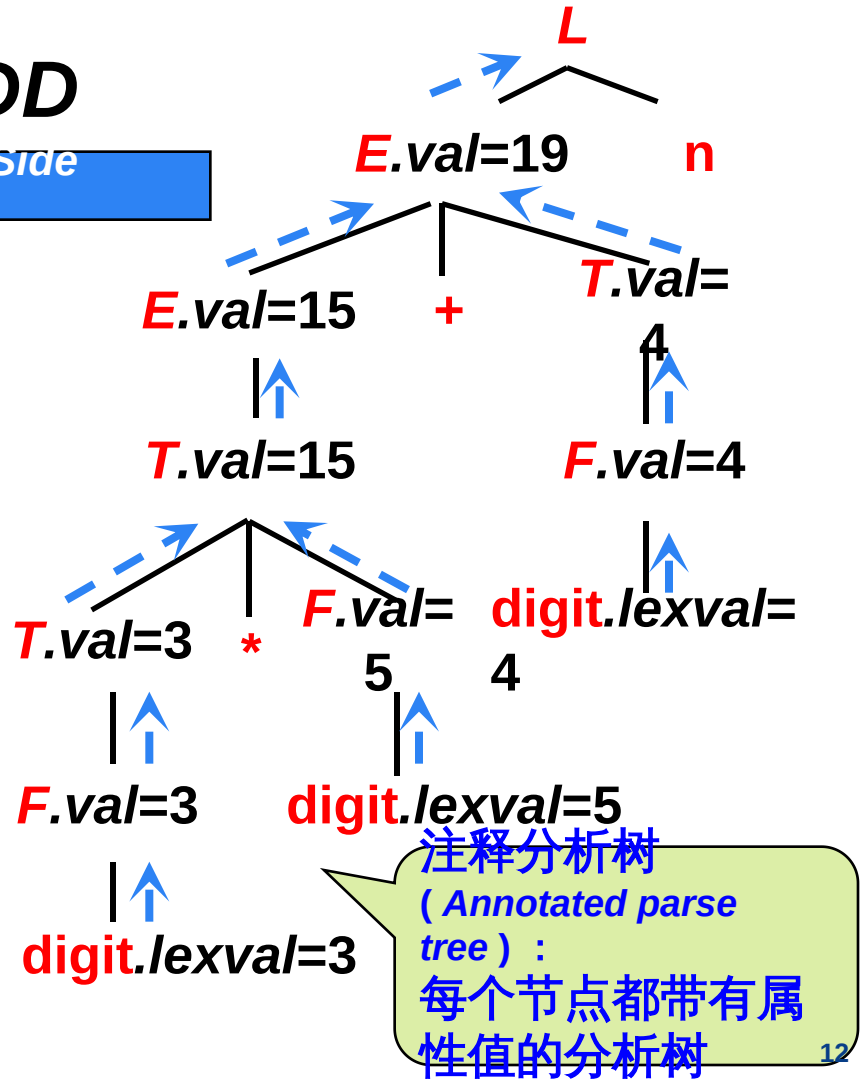
副作用(*Side effect*)

SDD：表达式求值

产生式	语义规则
(1) *L* → *E* n	print(*E.val*)
(2) *E* → E_1 + *T*	*E.val* = E_1.*val* + *T.val*
(3) *E* → *T*	*E.val* = *T.val*
(4) *T* → T_1 * *F*	*T.val* = T_1 *val* × *F.val*
(5) *T* → *F*	*T.val* = *F.val*
(6) *F* → (*E*)	*F.val* = *E.val*
(7) *F* → digit	*F.val* = digit.*lexval*

输入：3*5+4n

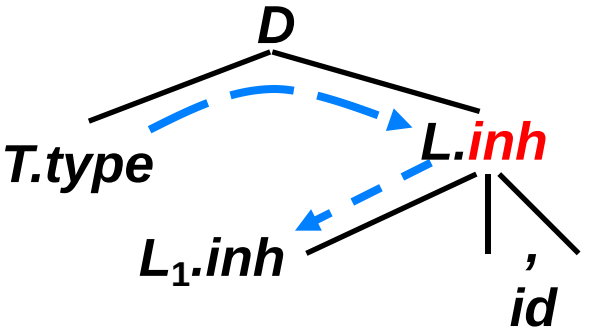
副作用可以看作是将函数的返回值赋给L的虚综合属性



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 → *T L*	*L. inh* = *T. type*
L → L_1, id	L_1.*inh* = *L.inh*

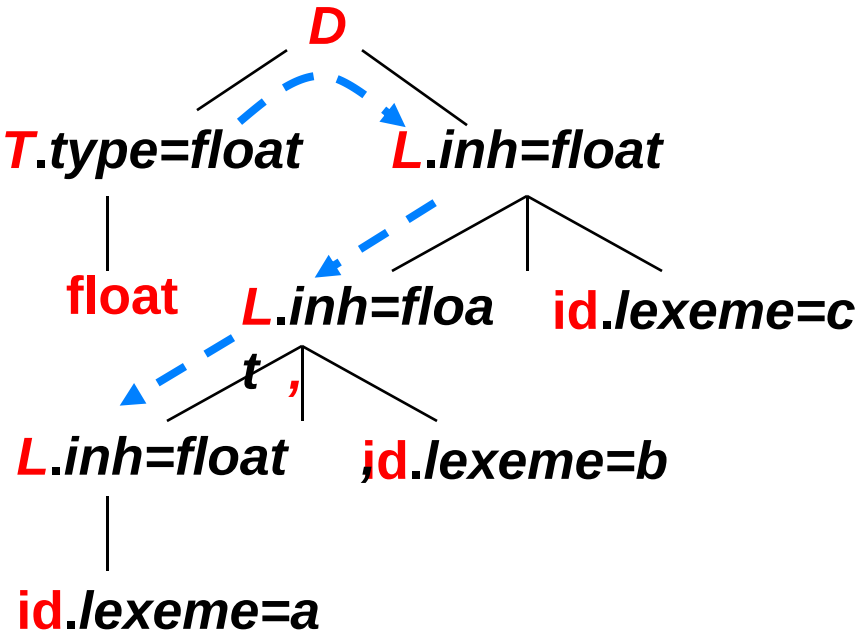


- 非终结符*A*的继承属性是由以*A*为右部符号之一的产生式所关联的语义规则来定义的。
- 终结符没有继承属性。

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*	*T L*	*L.inh = T. type*
(2)	*T*	int	*T.type = int*
(3)	*T*	float	*T.type = float*
(4)	*L*	L_1, id	L_1.*inh = L.inh* *addType*(id.*lexeme*, *L.inh*)
(5)	*L*	id	*addType*(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：
  addType(id的词法值, 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* → *E* n	*L.val* = *E.val*
(2) *E* → *E*$_1$ + *T*	*E.val* = *E*$_1$*.val* + *T.val*
(3) *E* → *T*	*E.val* = *T.val*
(4) *T* → *T*$_1$ * *F*	*T.val* = *T*$_1$*.val* × *F val*
(5) *T* → *F*	*T.val* = *F.val*
(6) *F* → (*E*)	*F.val* = *E.val*
(7) *F* → digit	*F.val* =

SDD属性计算

- 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
  - 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

通用性较好，无属性循环依赖就可以

对SDD有更多限制

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

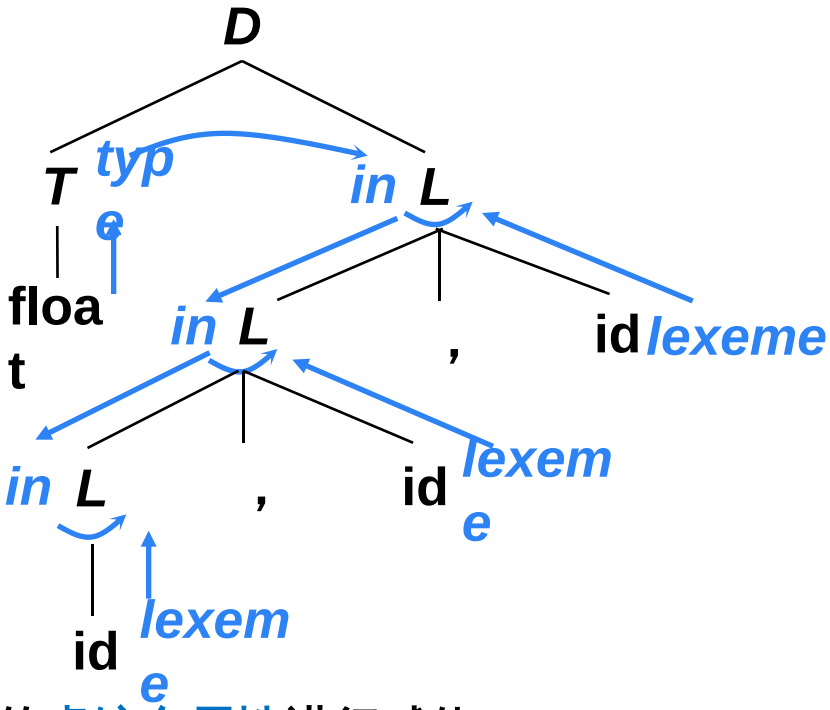
依赖图(*Dependency Graph*)

- **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*	*T L*	*L.in = T. type*
(2)	*T*	int	*T.type = int*
(3)	*T*	float	*T.type = float*
(4)	*L*	L_1, id	L_1*.in = L.in* *addtype*(id.*lexeme*, *L.in*)
(5)	*L*	id	*addtype*(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 = addtype(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, … , 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*)。

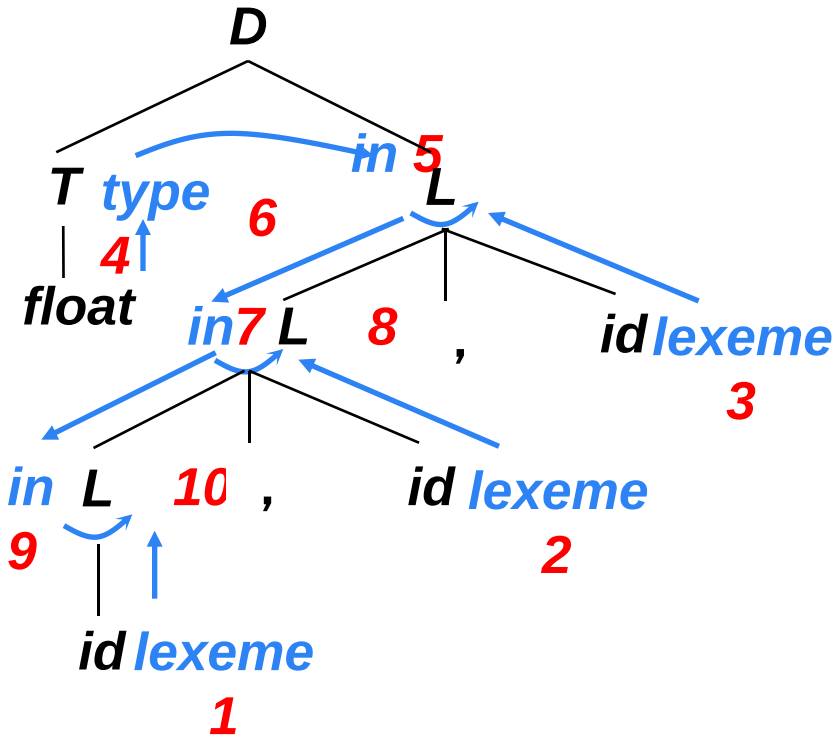
例

SDD ：声明语句的翻译定义

	产生式		语义规则
(1)	*D*	*T L*	*L.in = T. type*
(2)	*T*	int	*T.type = int*
(3)	*T*	float	*T.type = float*
(4)	*L*	L_1, id	L_1*.in = L.in* *addtype*(id.*lexeme*, *L.in*)
(5)	*L*	id	*addtype*(id.*lexeme*, *in*)

输入：

float *a* , *b* , *c*



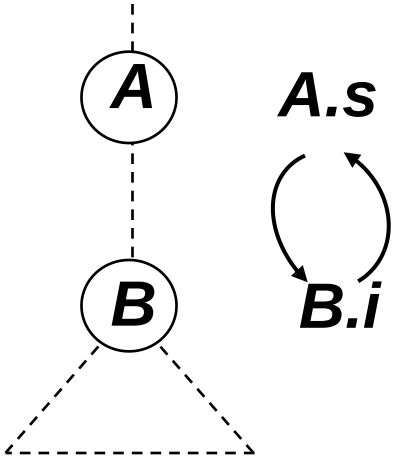
拓扑排序：

1, 2, 3, 4, 5, 6, 7, 8, 9, 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。

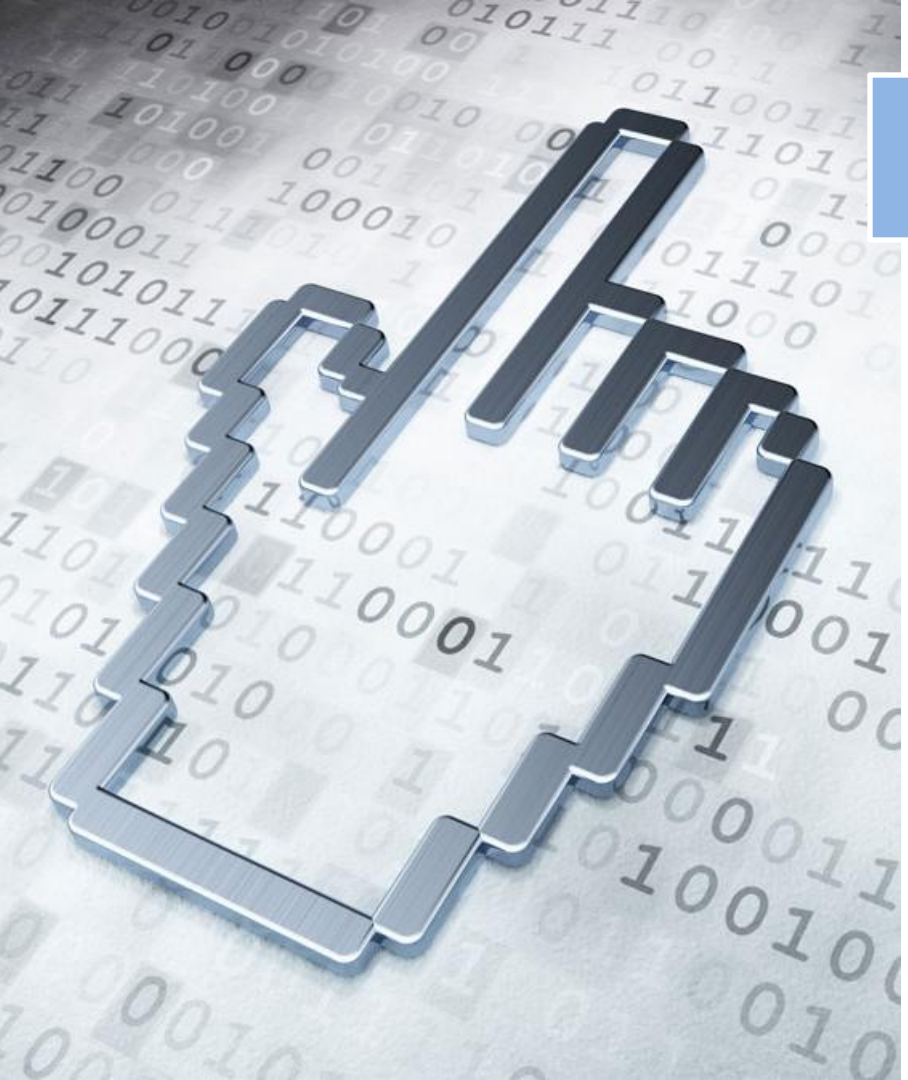
产生式	语义规则
~~*A B*~~	~~*A.s = B.i*~~
+1	*B.i = A.s*



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) $L \to E$ n	$L.val = E.val$
(2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \to T$	$E.val = T.val$
(4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \to F$	$T.val = F.val$
(6) $F \to (E)$	$F.val = E.val$
(7) $F \to$ digit	$F.val =$ 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$A \rightarrow X_1X_2\ldots X_n$，其右部符号$X_i$ (1 i 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 $X_1, X_2, \ldots, X_{i-1}$ 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

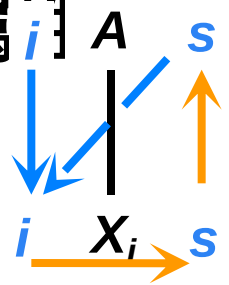
- 一个SDD是L-属性定义，当且仅当它的每个属性要么是一个综合属性，要么是满足如下条件的继承属性：假设存在一个产生式$A \rightarrow X_1X_2...X_n$，其右部符号$X_i$ (1 i n)的继承属性仅依赖于下列属性：
  - A的继承属性。。。
  - 产生式中X_i左边的符号 $X_1, X_2, \ldots, X_{i-1}$ 的属性
  - X_i本身的属性，但X_i 的全部属性不能在依赖图中形成环路

为什么不能依赖A的综合属性？

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$A \rightarrow X_1X_2...X_n$，其右部符号$X_i$ (1 i 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 $X_1, X_2, \ldots, X_{i-1}$ 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

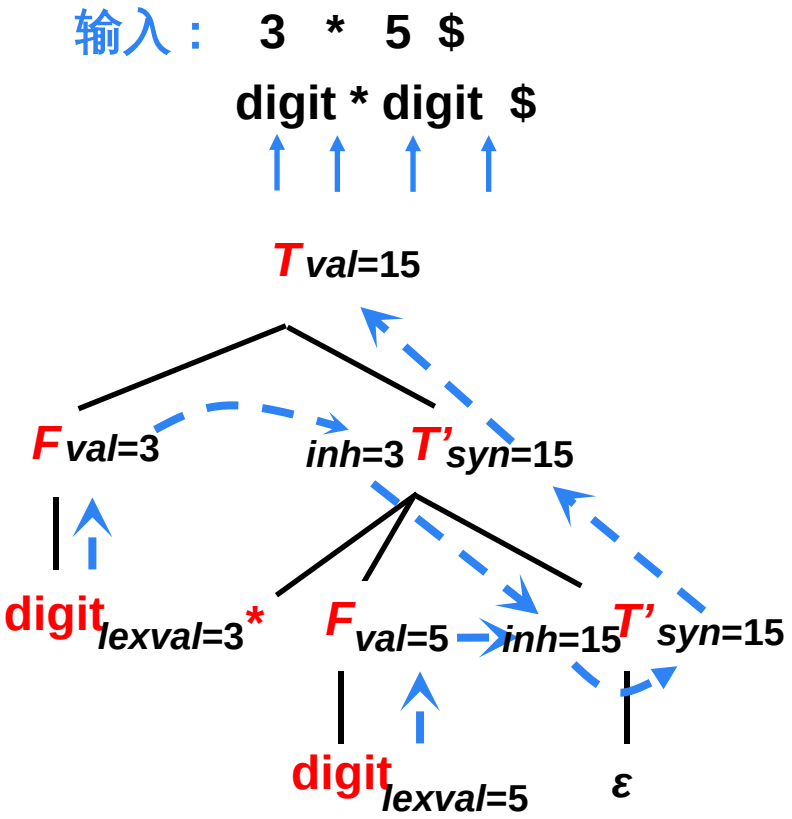
为了避免依赖图出现回路



例：*L-SDD*

为*LL*文法设计L-SDD，
实现表达式求值

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



思考题：为什么表达式的值要传递回根结点T？

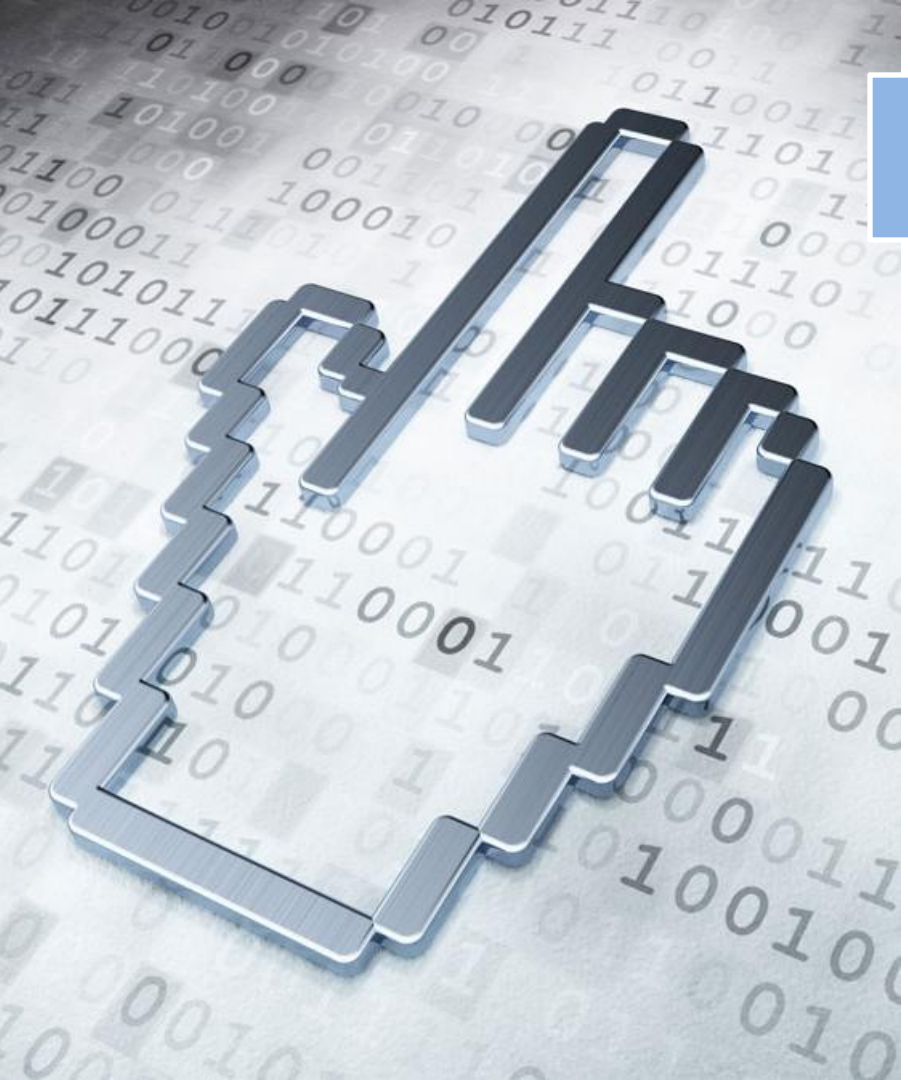
非L属性的SDD

- 例

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

继承属性

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

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

$T \rightarrow$ int $\{ T.type = int \}$

$T \rightarrow$ float $\{ T.type = float \}$

$L \rightarrow \{ L_1.inh = L.inh \} L_1$, id $\{ ddType(id.entry, L.inh) \}$

$L \rightarrow$ id $\{ addType(id.entry, L.inh) \}$

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*
- 任意*SDT*都可以先构建语法分析树，再先根深度优先遍历语法分析树中的动作节点来完成计算。(5.4.3)
- 能在语法分析中计算的两类重要的*SDD*的*SDT*
  - 基础文法可以使用*LR*分析技术，且*SDD*是*S*属性的
  - 基础文法可以使用*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) $L \rightarrow E$ 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) $F \rightarrow$ digit	$F.val =$ digit.*lexval*

SDT

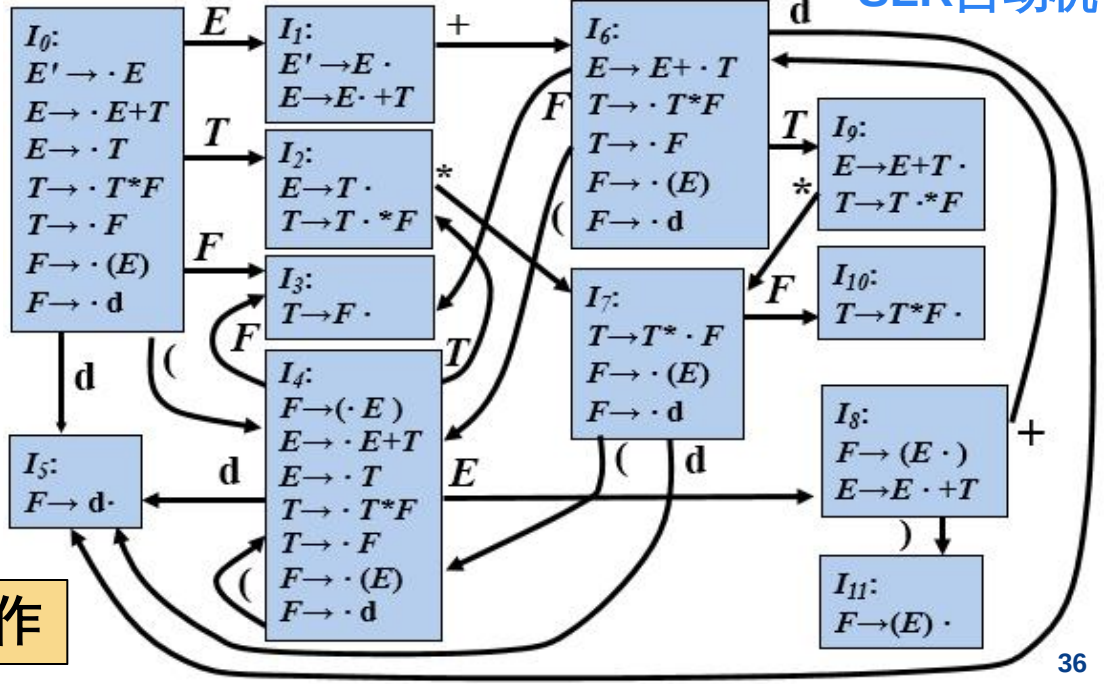
(1) $L \rightarrow E$ n { $L.val = E.val$ }
(2) $E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }
(3) $E \rightarrow T$ { $E.val = T.val$ }
(4) $T \rightarrow T_1 * F$ { $T.val = T_1.val \times F.val$ }
(5) $T \rightarrow F$ { $T.val = F.val$ }
(6) $F \rightarrow (E)$ { $F.val = E.val$ }
(7) $F \rightarrow$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

S-属性定义的SDT 实现

- 如果一个S-SDD的基础文法可以使用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

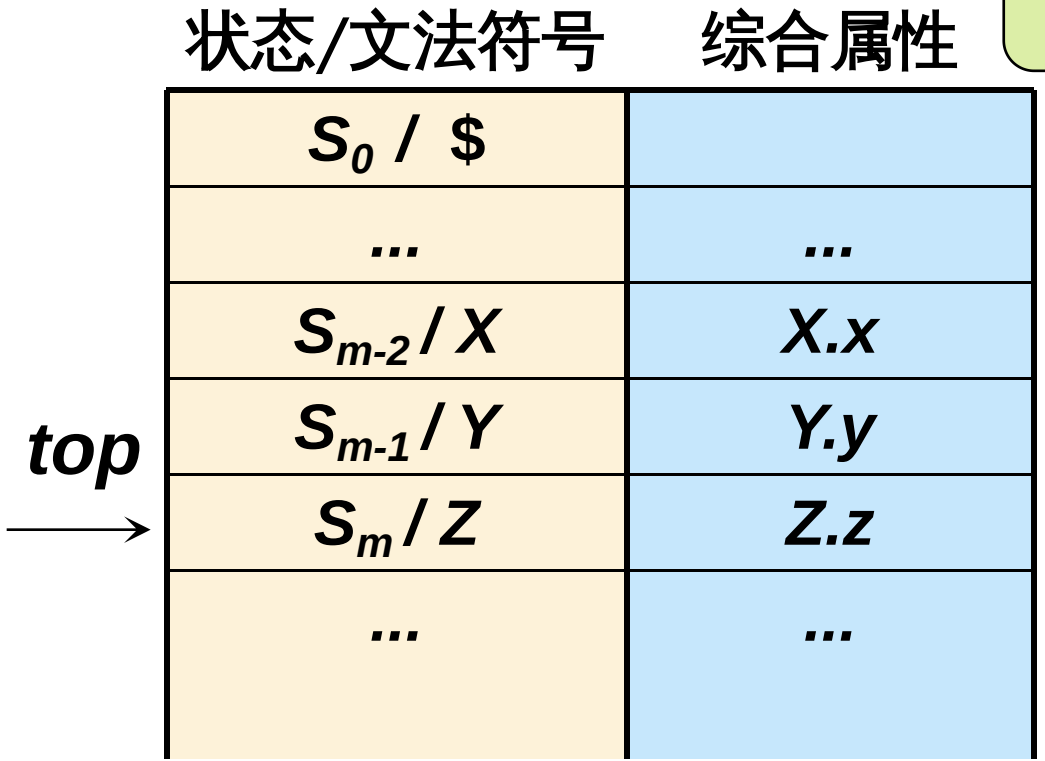
(1) L E n { $L.val = E.val$}
(2) E $E_1 + T${$E.val = E_1.val + T.val$}
(3) E T { $E.val = T.val$}
(4) T $T_1 * F$ { $T.val = T_1.val \times F.val$}
(5) T F { $T.val = F.val$}
(6) F (E) { $F.val = E.val$}
(7) F digit { $F.val$ = digit.$lexval$}

当归约发生时执行相应的语义动作



扩展的LR语法分析栈

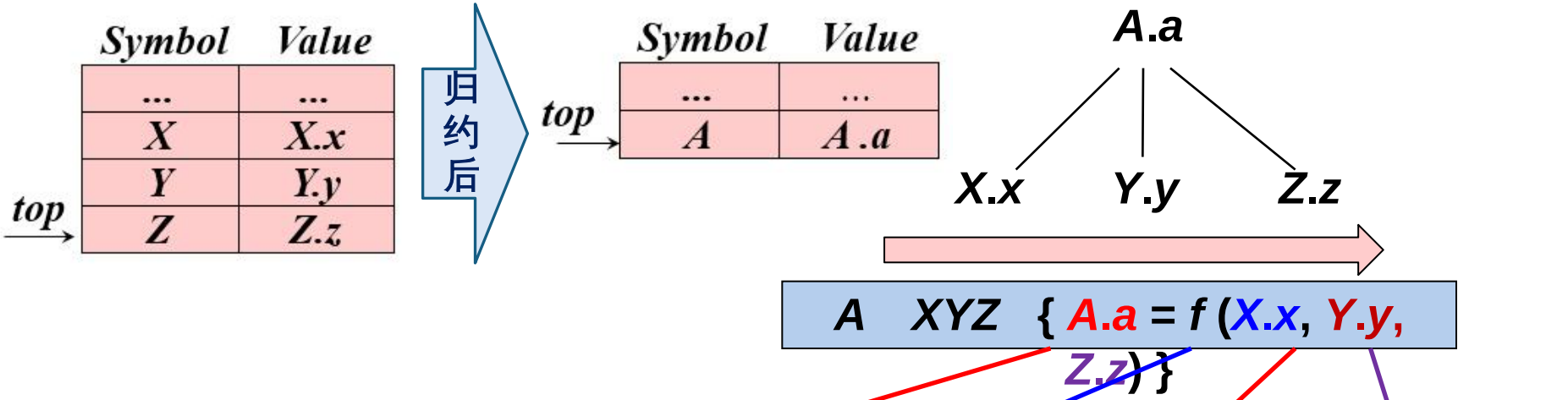
在分析栈中使用一个附加的域来存放综合属性值



- 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

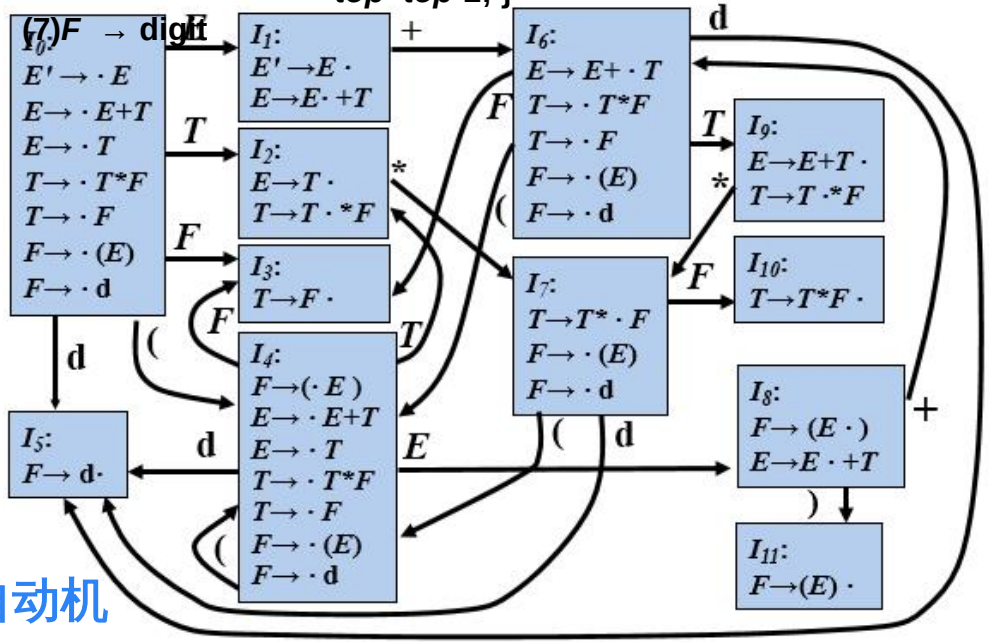



```
stack[top-2].sym = A   //归约动作，实际入栈的是A对应的状态
stack[top-2].val = f( stack[top-2].val, stack[top-1].val, stack[top].val )
top =top-2;   //产生式右部剩余符号出栈
```

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)$E' \rightarrow E$	print(*E.val*)	{ print (*stack*[*top*].*val*);}
(2)$E \rightarrow E_1 + T$	*E.val* = E_1.*val* + *T.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* + *stack*[*top*].*val* ; *top*=*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* × *F.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* × *stack*[*top*].*val* ; *top*=*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*	{ *stack*[*top*-2].*val* = *stack*[*top*-1].*val*; *top*=*top*-2; }

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

输入： 3*5+4 $

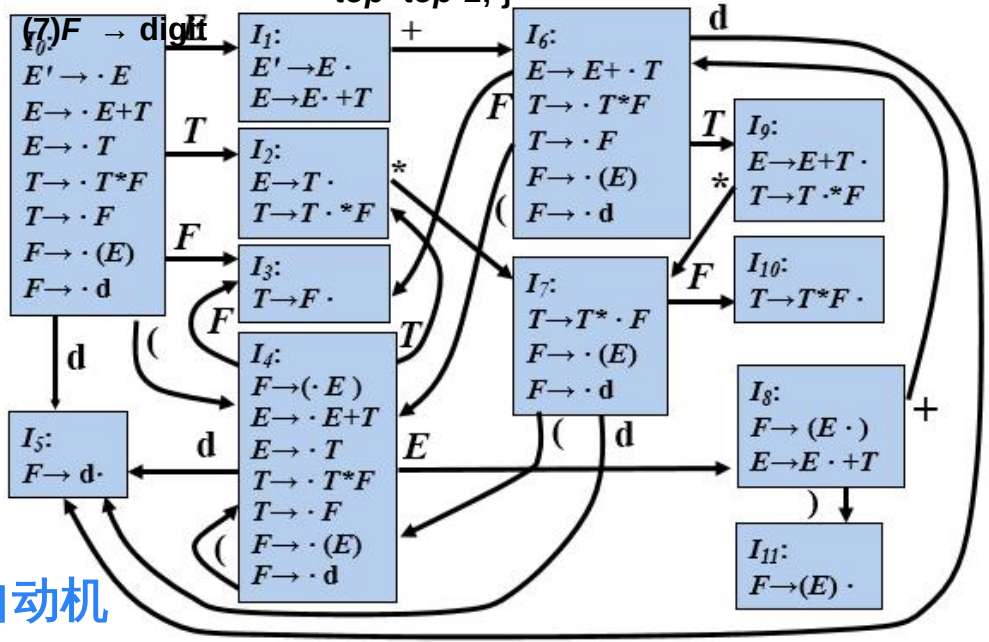
↑↑

状态 /符号 属

0	$	_
5	d	3

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

输入： 3*5+4 $

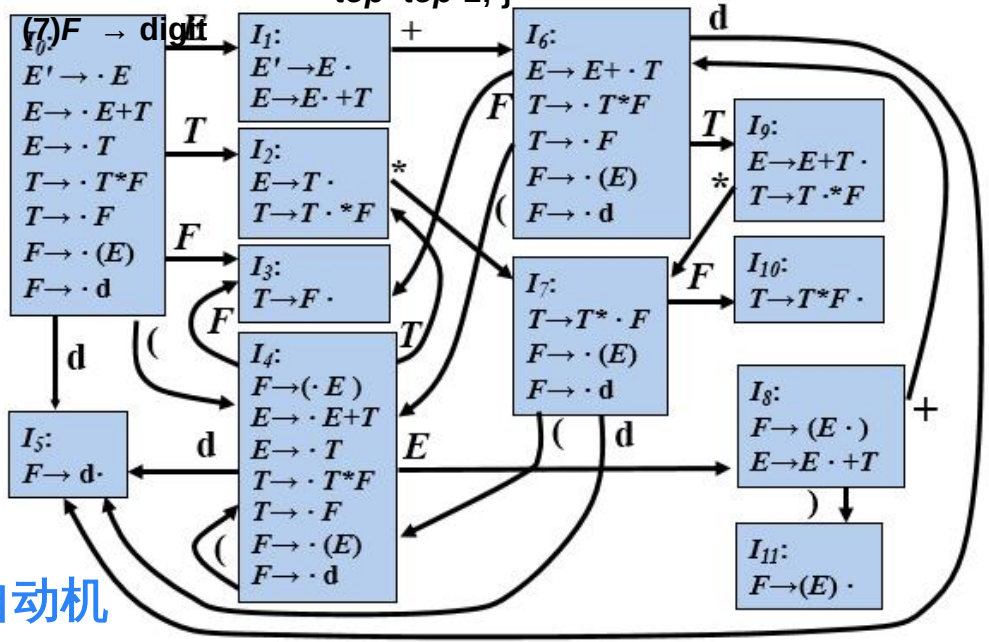
↑↑

状态 /符号 属

0	$	_
3	*F*	3

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

输入： 3*5+4 $

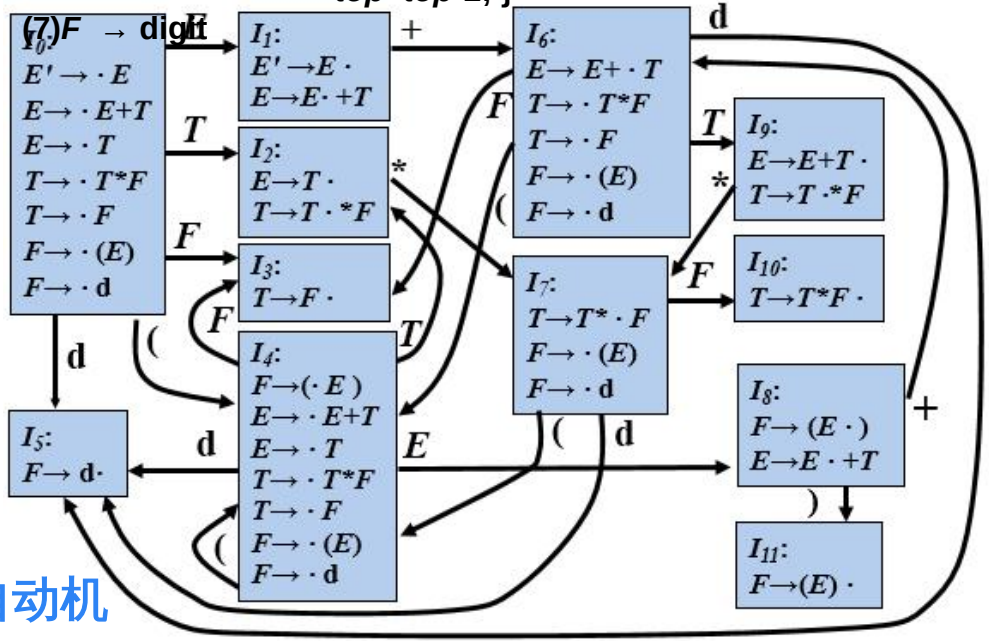
↑↑↑↑

状态 /符号 属

0	$	_
2	*T*	3
7	*	_
5	d	5

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

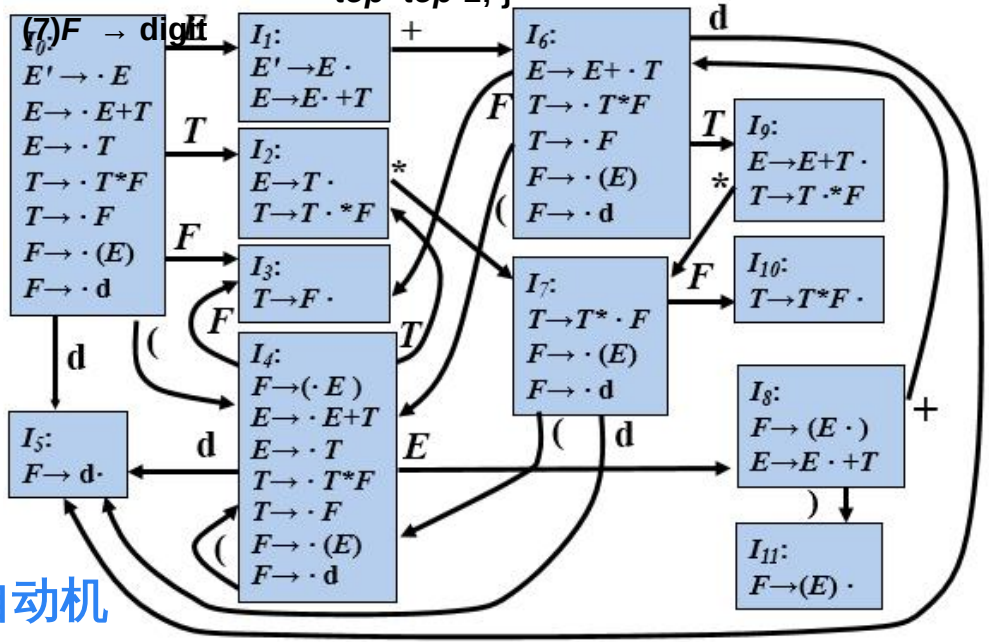
输入： 3*5+4 $

状态 /符号 属

0	$	_
2	*T*	15
7	*	_
10	*F*	5

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

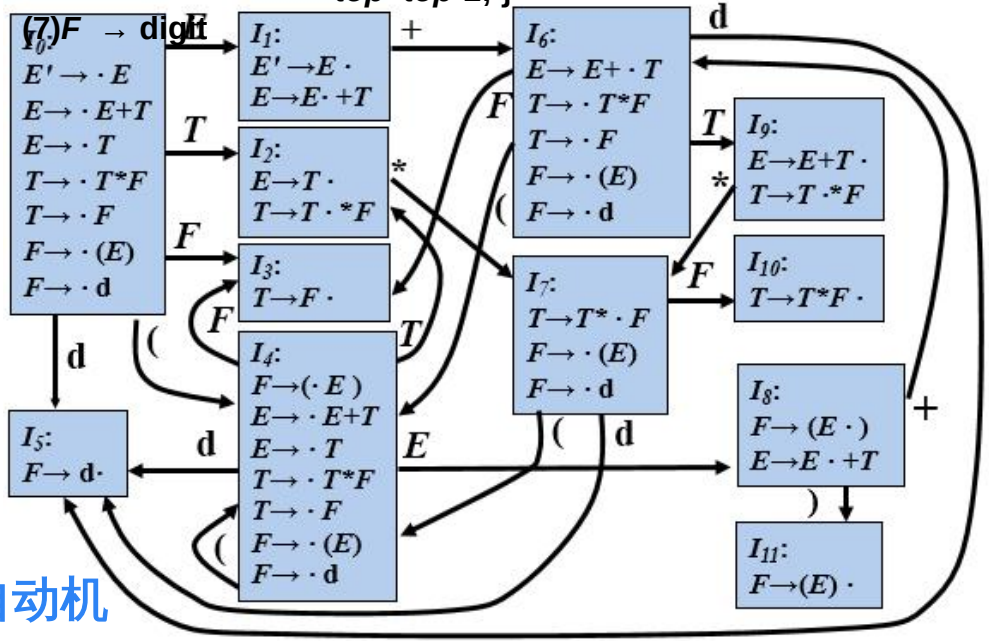
输入： 3*5+4 $

状态 /符号 属

0	$	_
2	*T*	15

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

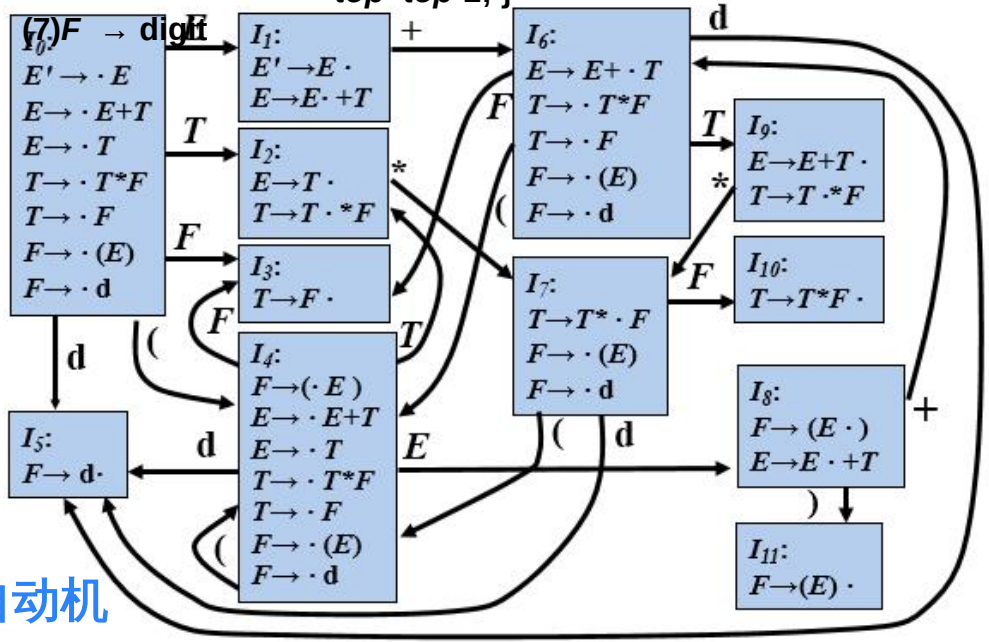
输入： 3*5+4 $

状态 /符号 属

0	$	_
1	*E*	15
6	+	_
5	d	4

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

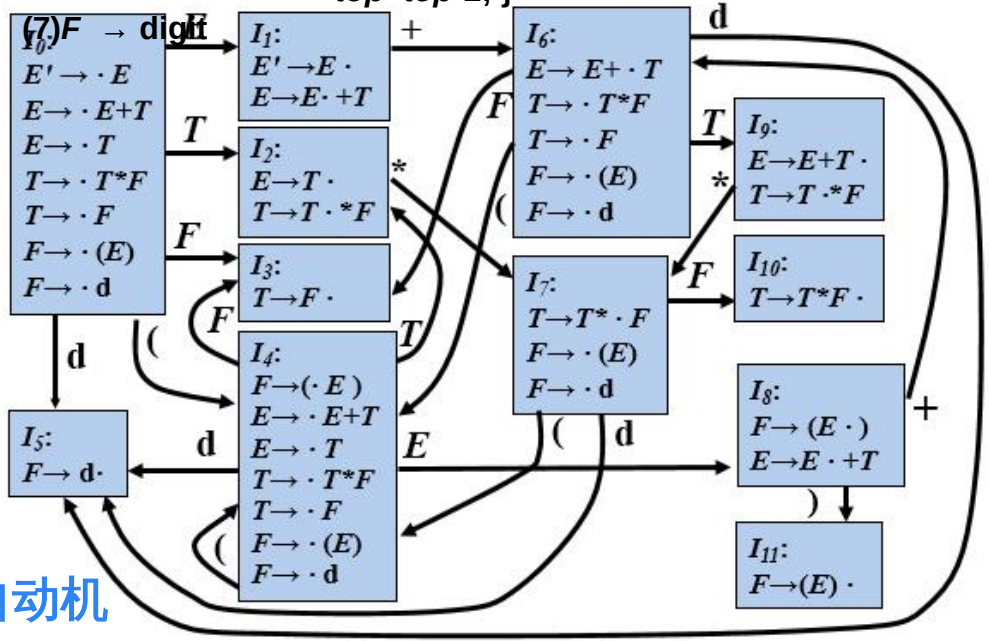
输入： 3*5+4 $

状态 /符号 属

0	$	_
1	*E*	15
6	+	_
3	*F*	4

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

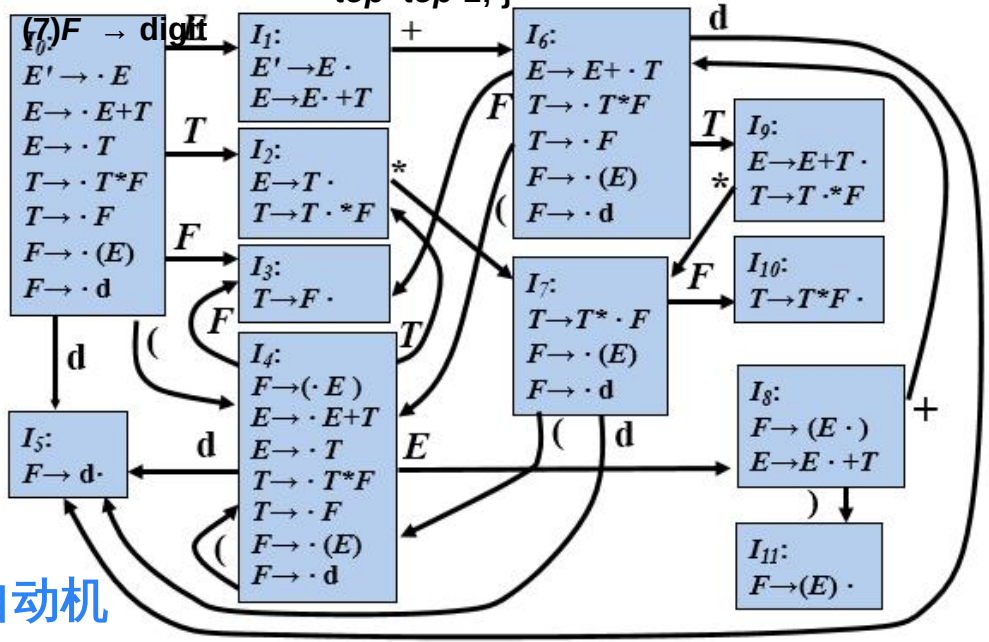
输入： 3*5+4 $

状态 /符号 属

0	$	_
1	*E*	19
6	+	_
9	*T*	4

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

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* × *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)$F \rightarrow$ digit	



SLR自动机

digit*digit+digit $

输入： 3*5+4 $

状态 /符号 属

0	$	_
1	*E*	19

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

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)	$T \to F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
(2)	$T' \to *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$
		$T'.syn = T_1'.syn$
(3)	$T' \to \varepsilon$	$T'.syn = T'.inh$
(4)	$F \to$ digit	$F.val =$ digit.*lexval*

思考题：可不可以把符号A的继承属性的计算放在A之后 ？

- *SDT*

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

L-属性定义的*SDT* 实现

- 如果一个*L-SDD*的基础文法可以使用*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) $F \rightarrow$ digit { $F.val$ = 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*的基础文法可以使用*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

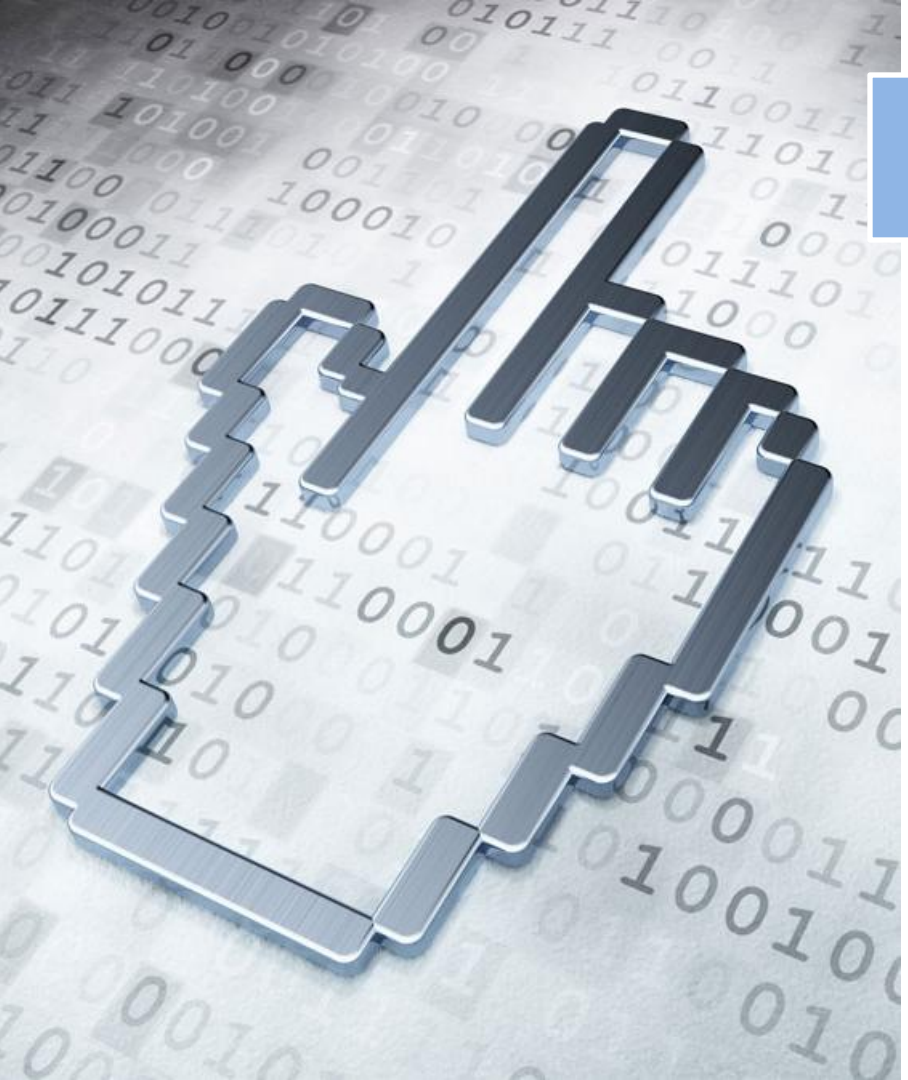
练习1

- 下面的文法生成了含“小数点”的二进制数：

$S \rightarrow L_1 . L_2 | L$　　$L \rightarrow L_1 B | B$　　$B \rightarrow 0 | 1$

(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。



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

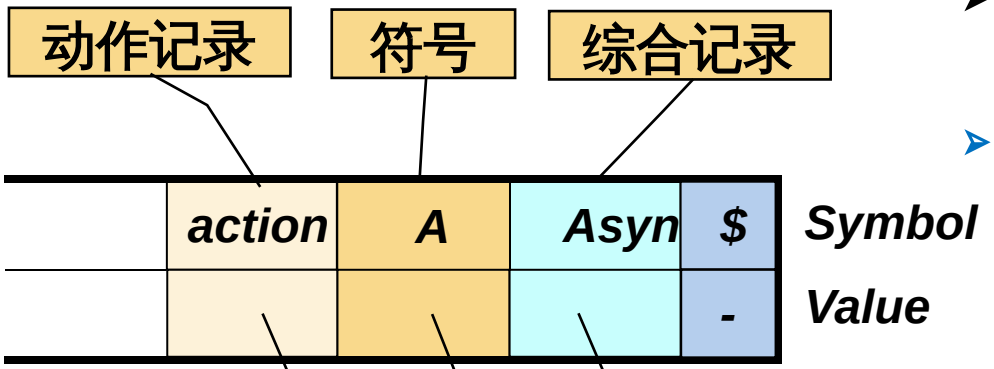
5.5 L-属性定义的自底向上翻译

5.4 L-属性定义的自顶向下翻译

- 5.4.1 在非递归的预测分析过程中进行翻译
  - 表驱动的预测分析
  - 显示维护一个栈
- 5.4.2 在递归的预测分析过程中进行翻译
  - 为每个非终结符定义一个处理函数

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



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

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) $F \rightarrow$ 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 : $F.val =$ digit.*lexval*

例

SDT

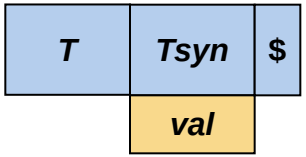
1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow \text{digit} \{ a_6 \}$

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。



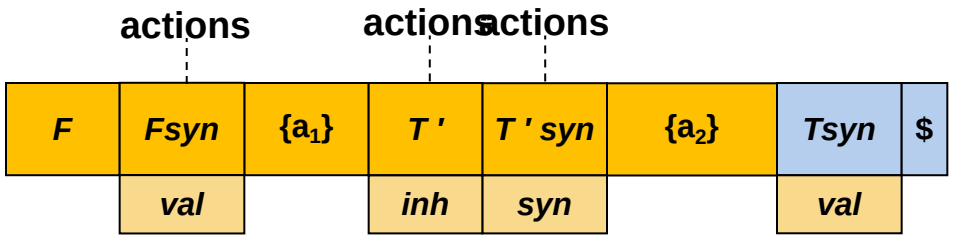
例

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) $F \rightarrow$ 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 : $F.val =$ digit.*lexval*

输入: 3 * 5 $



例

SDT

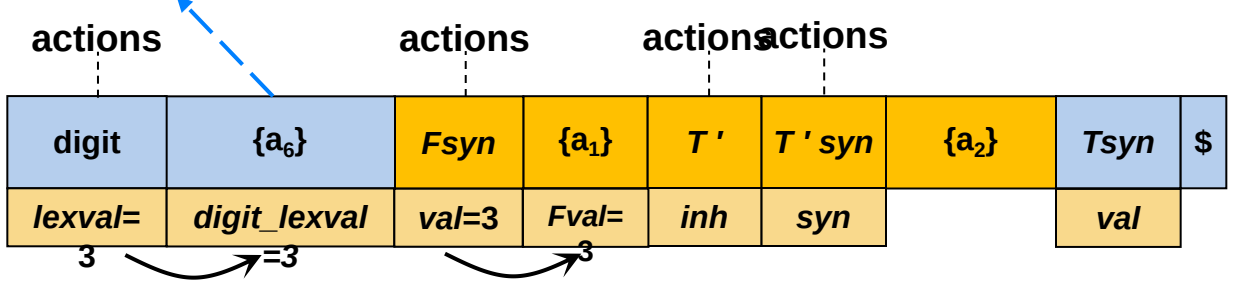
1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow$ digit $\{ a_6 \}$

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 : $F.val =$ 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

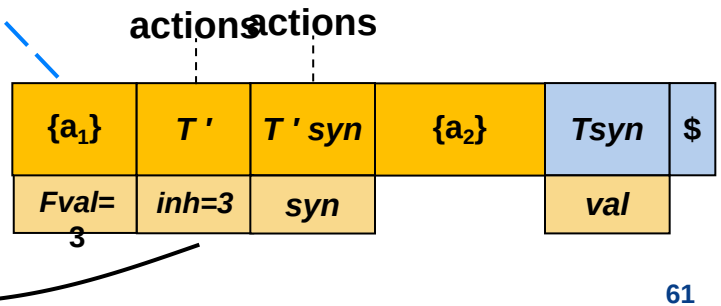
SDT	
1) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	a_1 : $T'.inh = F.val$
2) $T' \rightarrow *F\ \{a_3\}\ T_1'\ \{a_4\}$	a_2 : $T.val = T'.syn$
3) $T' \rightarrow \varepsilon\ \{a_5\}$	a_3 : $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$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

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



例

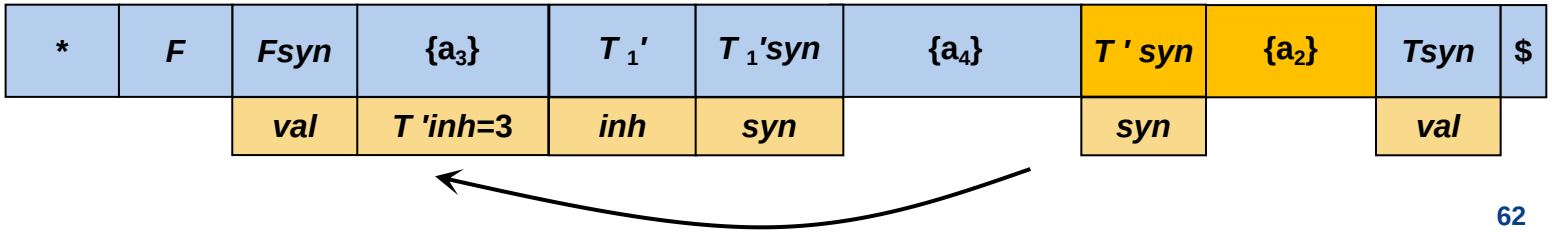
SDT

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) $F \rightarrow$ 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 : $F.val$ = digit.*lexval*

输入: 3 * 5 $

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



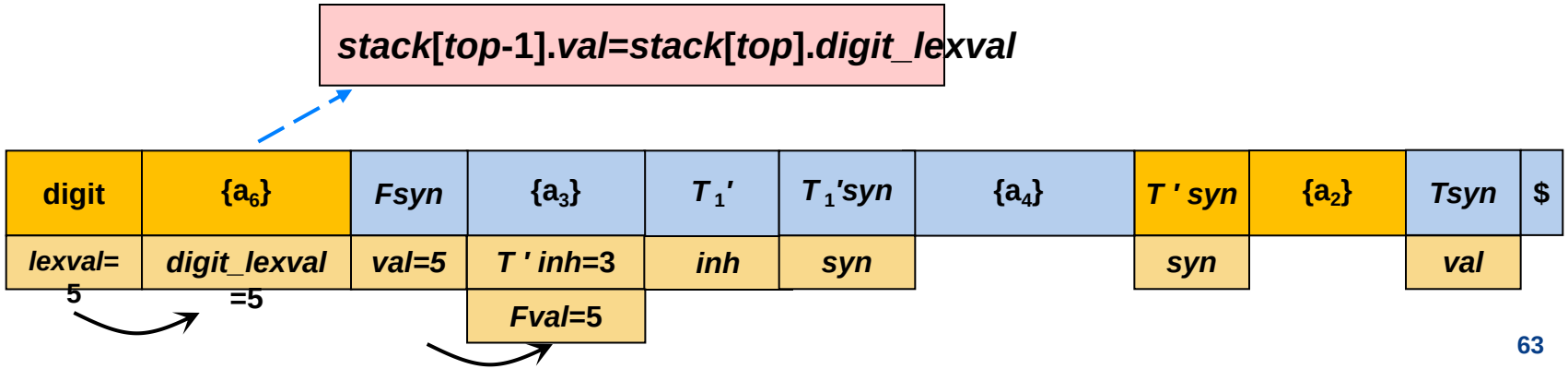
例

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) $F \rightarrow$ 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 : $F.val$ = digit.*lexval*

输入: 3 * 5 $



例

SDT

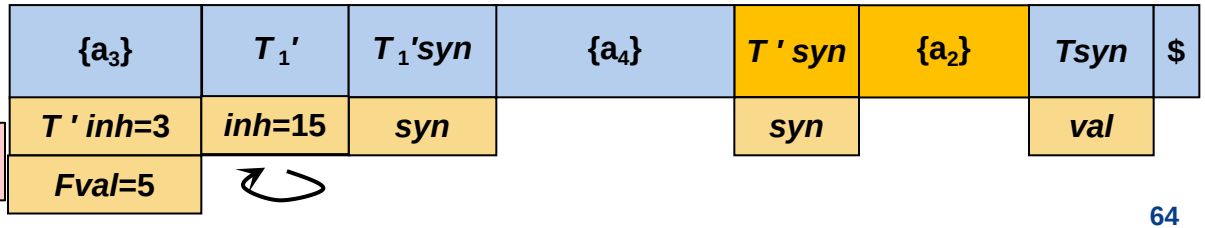
1) $T \rightarrow F \{ a_1 \} T' \{ a_2 \}$
2) $T' \rightarrow *F \{ a_3 \} T_1' \{ a_4 \}$
3) $T' \rightarrow \varepsilon \{ a_5 \}$
4) $F \rightarrow$ digit $\{ a_6 \}$

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 : $F.val$ = 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*×*stack*[*top*].*Fva*

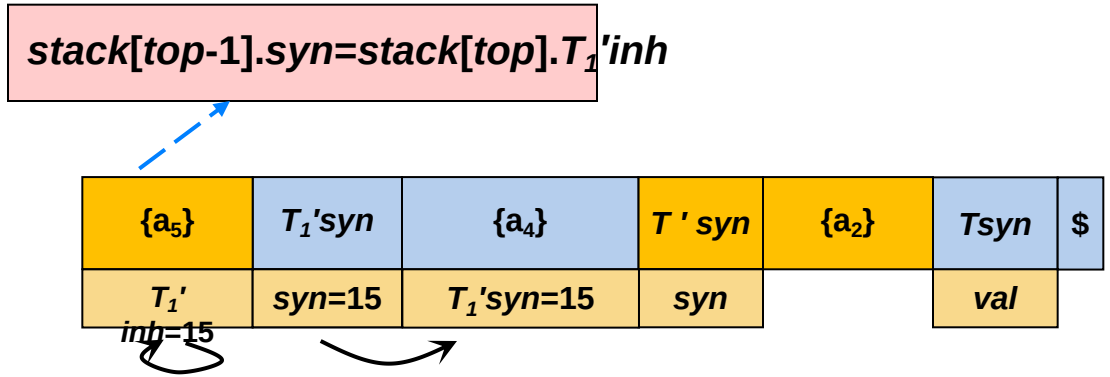


stack[*top*].*T'inh* = *stack*[*top*].*inh*;

例

SDT	
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) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val$ = digit.$lexval$

输入: 3 * 5 $

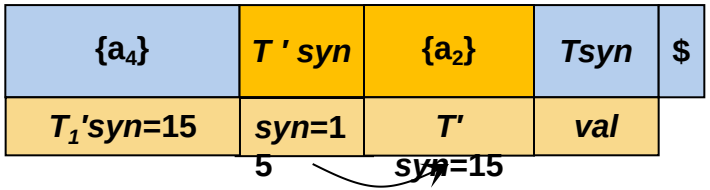


例

SDT	
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) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val$ = digit.*lexval*

输入: 3 * 5 $

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



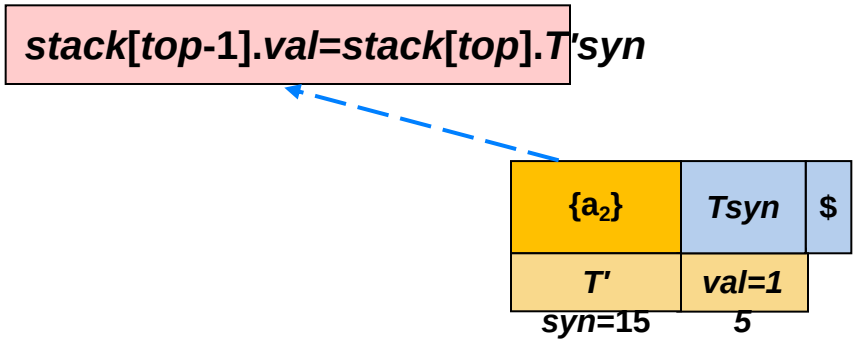
例

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) $F \rightarrow$ 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 : $F.val$ = digit.*lexval*

输入: 3 * 5 $



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。
  - 根据需要此值的语义动作在产生式中的位置，来确定对应的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) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	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) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val$ = 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; 制到{a1}	复
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; 到{a3} 若选 3): *stack*[*top*].*T'inh* =*stack*[*top*].*inh*; 到{a5}	复制 复制

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) $F \rightarrow$ 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 : $F.val$ = 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; 制到{a3}	复
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; 制到{a3} 若选3): *stack*[*top*].*T'inh* = *stack*[*top*].*inh*; 到{a5}	复 复制

例

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) $F \rightarrow$ digit $\{ a_6 \}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val =$ 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) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	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) $F \rightarrow$ digit $\{a_6\}$	a_4 : $T'.syn = T_1'.syn$
	a_5 : $T'.syn = T'.inh$
	a_6 : $F.val$ = digit.$lexval$

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

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

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

对出现在A产生式右部中的每个文法符号的每个属性都设置一个局部变量

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, T1'inh, T1'syn ;
  if token="*" then
  { Getnext(token) ;
    Fval=F(token) ;
    T1'inh= T'inh×Fval ;
    T1'syn=T1'(token, T1'inh) ;
    T'syn=T1'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) $F \rightarrow$ digit { *F.val* = 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) $F \rightarrow$ digit $\{ F.val =$ digit.*lexval* $\}$

```
Fval F(token)
{
   if token ≠ "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) $F \rightarrow$ digit $\{ F.val =$ 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$c := B(b_1, b_2, \cdots, b_k)$，其中， $b_1, b_2, \cdots, b_k$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：

$S \rightarrow L_1 . L_2 | L$　　$L \rightarrow L_1 B | B$　　$B \rightarrow 0 | 1$

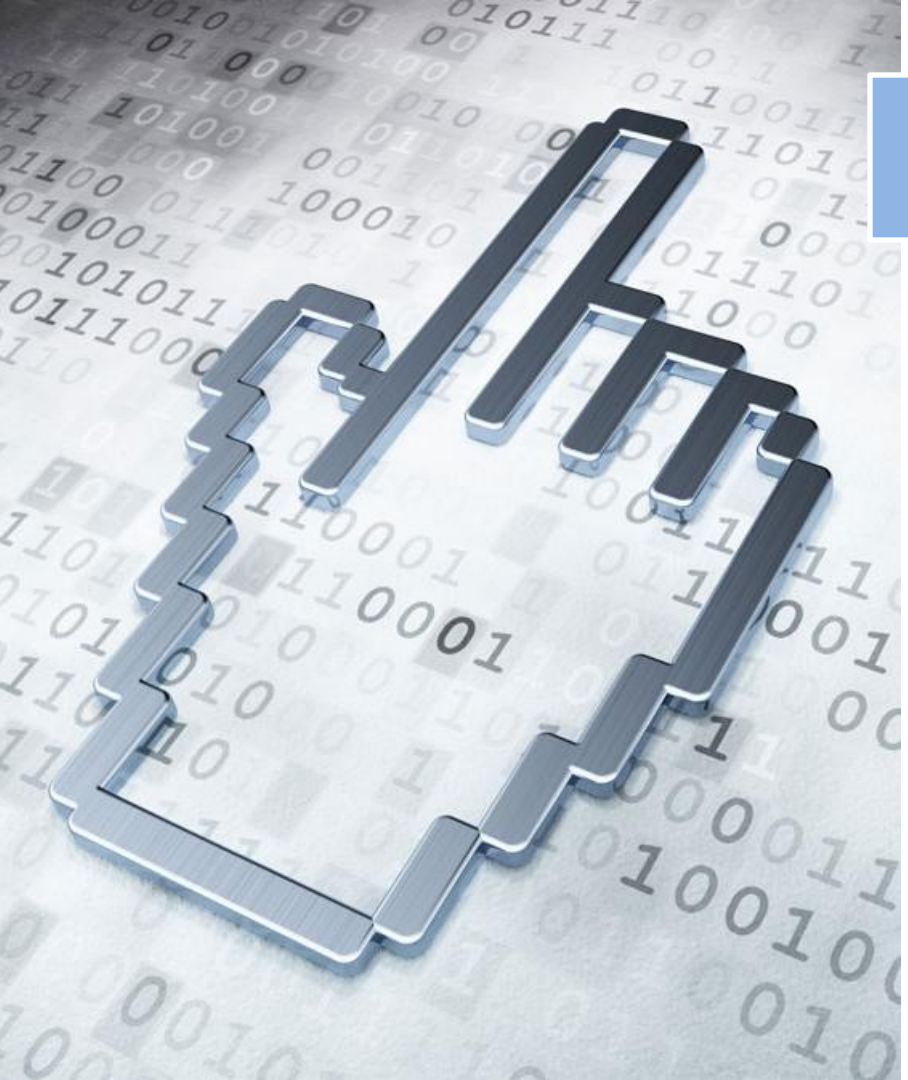
消除左递归和左公因子后，为LL(1) 文法：

$S \rightarrow LS'$　　$S' \rightarrow . L | \epsilon$　　$L \rightarrow BL'$

$L' \rightarrow BL' | \epsilon$　　$B \rightarrow 0 | 1$

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PPT72-75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***，可以改造这个文法及其***L-SDD***成为后缀***SDT***，并在***LR***语法分析过程中计算这个新文法之上的***SDT***。
- 可以证明，***LL***文法经过这样的改造之后一定是***LR***文法。
- 但***LR***文法经过这样的改造后不能保证还是***LR***文法，如果不是，只能采用通用方法。

改造文法和*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) $F \rightarrow$ digit $\{ F.val =$ digit $.lexval \}$

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

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

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

$N.i2 = F.val;$

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

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

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

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

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

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

$N.i2 = F.val;$

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

修改后的*SDT*，所有语义动作都位于产生式末尾

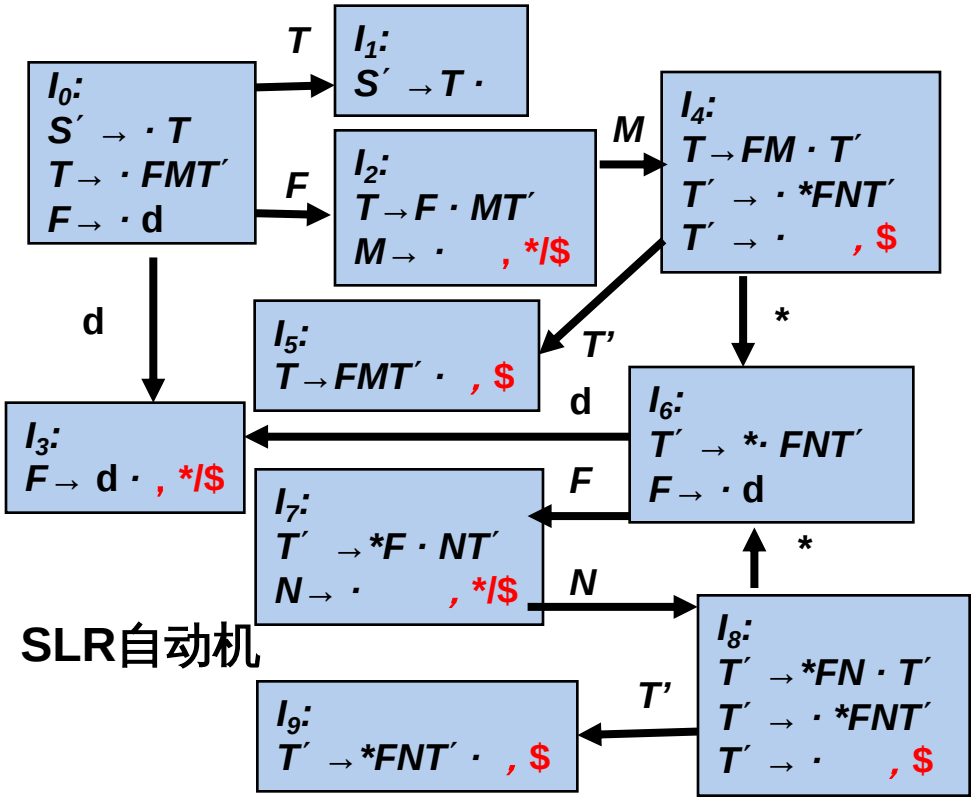
F和N在产生式体中的相对位置保证：归约出N时，栈底一定已归约出F

M和T'在产生式体中的相对位置保证在T'的右部入栈时，M已经在栈下面的某个位置了

思考：如何保证正确访问未出现在该产生式中的符号的属性？

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

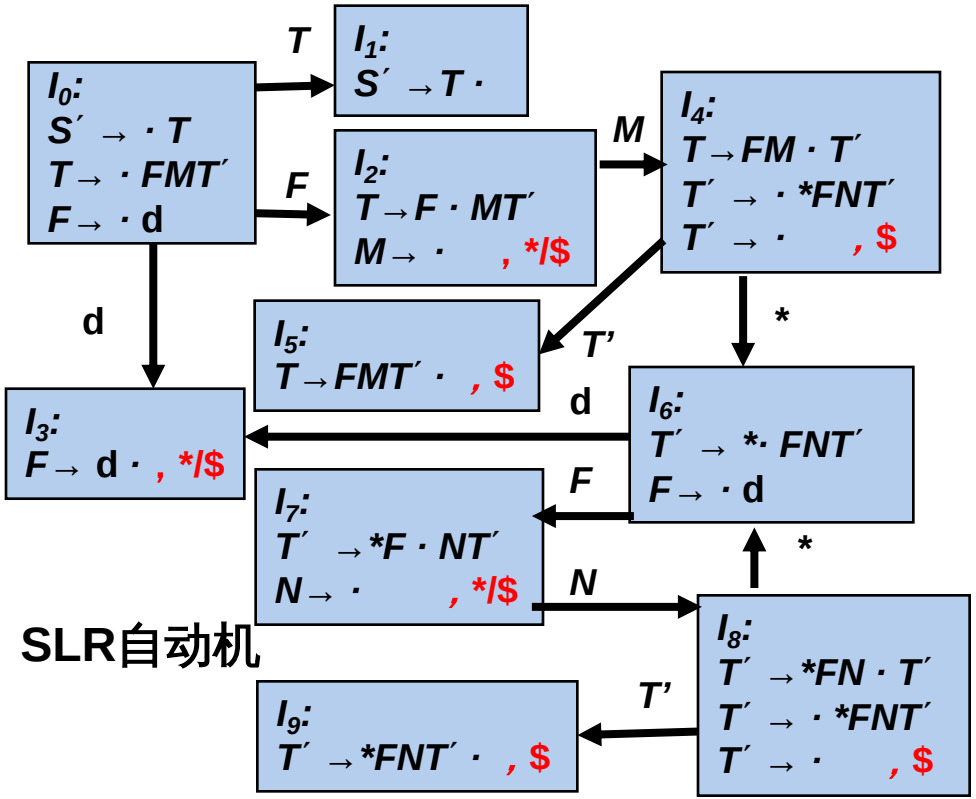


SLR自动机

输入：３＊５＄

0	3
$	d
	3

例



SLR自动机

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

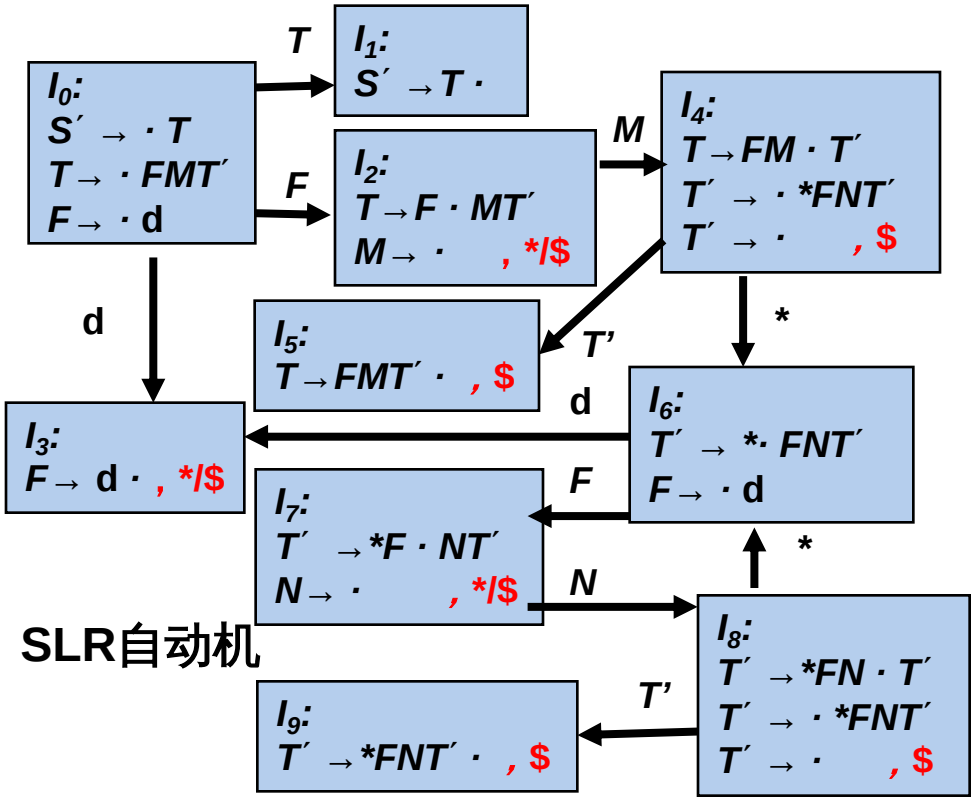
输入：3 * 5 $

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

输入：３＊５＄

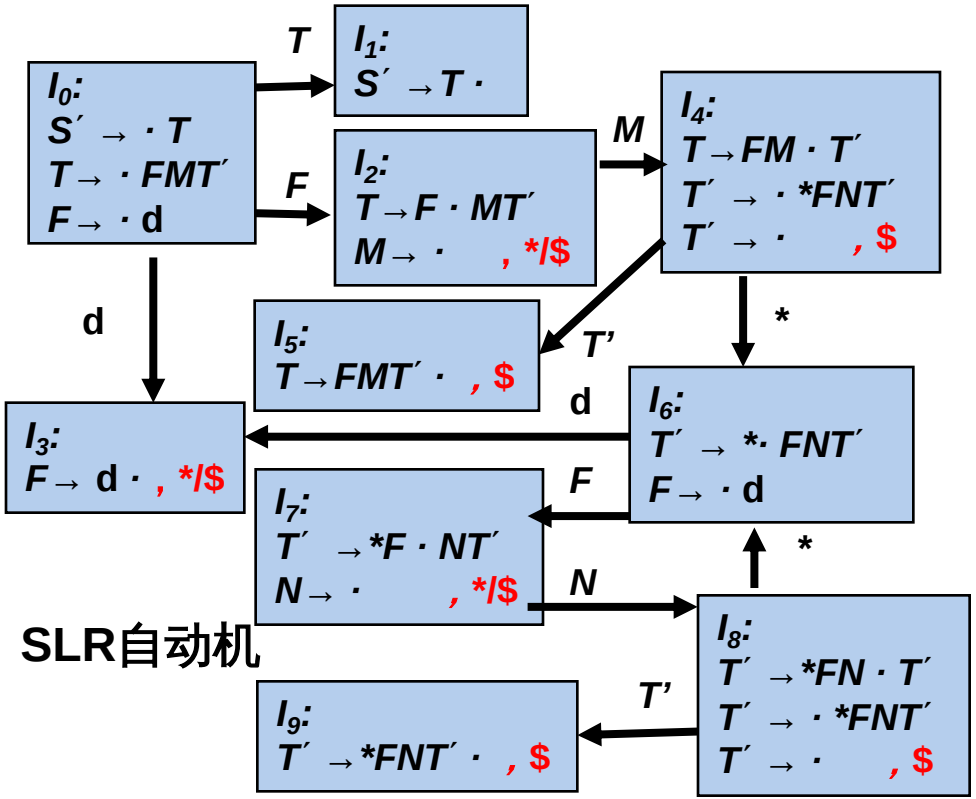


SLR自动机

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



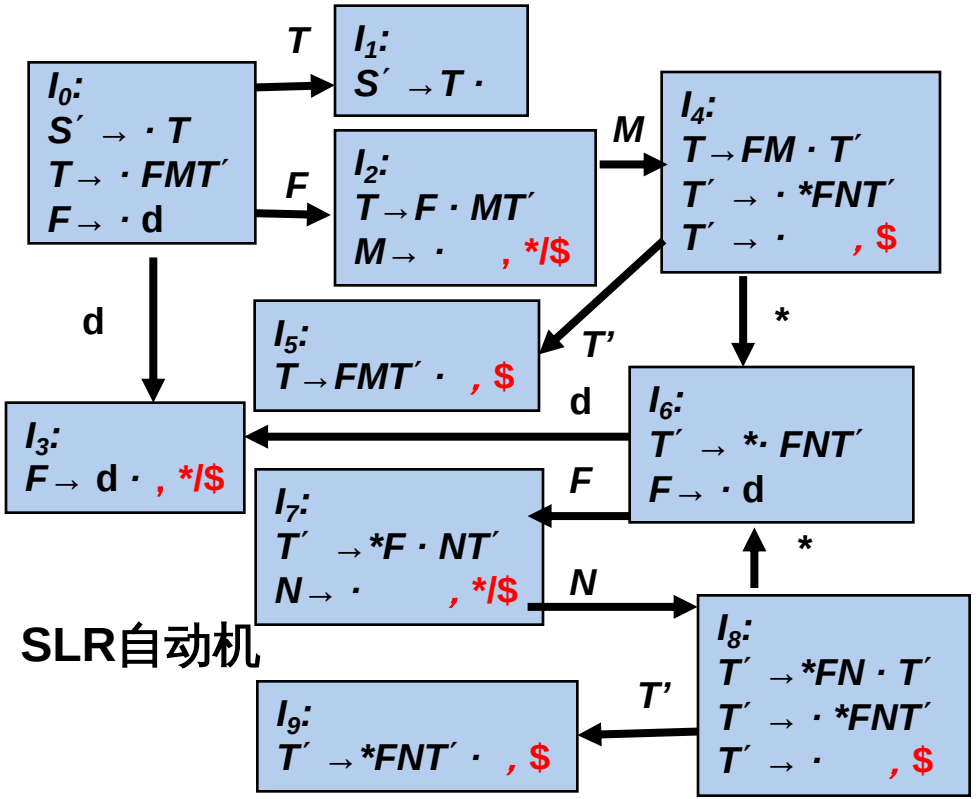
SLR自动机

输入：3 * 5 $

0	2	4	5
$	F	M	T'
	3	T'inh=3	syn=1
			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) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }



SLR自动机

输入：3 * 5 $

0	1
$	T
	val=1

5

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

1) $T \rightarrow F\ M\ T'$ {$stack[top-2].val = stack[top].syn$; $top = top-2$;}

   $M \rightarrow \varepsilon$ {$stack[top+1].T'inh = stack[top].val$; $top = top+1$;}

2) $T' \rightarrow *F\ N\ T_1'$ {$stack[top-3].syn = stack[top].syn$; $top = top-3$;}

   $N \rightarrow \varepsilon$ { $stack[top+1].T'inh = stack[top-2].T'inh \times stack[top].val$; $top = top+1$; }

3) $T' \rightarrow \varepsilon$ { $stack[top+1].syn = stack[top].T'inh$; $top = top+1$; }

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：

$S \rightarrow L_1 . L_2 | L$　　$L \rightarrow L_1 B | B$　　$B \rightarrow 0 | 1$

消除左递归和左公因子后，为LL(1) 文法：

$S \rightarrow LS'$　　$S' \rightarrow . L | \epsilon$　　$L \rightarrow BL'$

$L' \rightarrow BL' | \epsilon$　　$B \rightarrow 0 | 1$

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。

小结

- 通用方法
  - SDD -> 构建依赖图 –>按依赖顺序计算　条件：无回路
  - 任意SDT -> 构建语法分析树 ->遍历其中的动作节点
- 特殊的SDD: S-SDD 和L-SDD

基础文法	SDD	SDT	语法分析实现
LR	S-SDD	后缀	LR
LL	L-SDD	非后缀	LL
LL	L-SDD	改造成基于LR文法的后缀SDT	LR
LR	L-SDD	改造后若基于LR 若不是基于LR	LR 通用方法

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。

课程主要内容

1．绪论	（2学时）
2. 语言及其文法	（2学时）
3．词法分析	（3学时）
4．语法分析	（9学时）
5．语法制导翻译	（6学时）
6．中间代码生成	（7学时）
7．运行时的存贮组织	（3学时）
8．代码优化	（6学时）
9．代码生成	（2学时）

结束