第七章：语法制导的语义计算

提纲

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

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

两个概念

- 将语义规则同语法规则（产生式）联系起来要涉及两个概念
  - 语法制导定义(*Syntax-Directed Definitions, SDD*)
  - 语法制导翻译方案 (*Syntax-Directed Translation Scheme , SDT*)

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
- 如果*X*是一个文法符号，*a*是*X*的一个属性，则用*X.a*表示属性*a*在某个标号为*X*的分析树结点上的值

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
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow \text{int}$	$T.type = int$
$T \rightarrow \text{real}$	$T.type = real$
$L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$
…	…

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。按照惯例，语义动作放在花括号内
  - 例

$$D \rightarrow T \{ L.inh = T.type \} L$$
$$T \rightarrow \text{int} \{ T.type = int \}$$
$$T \rightarrow \text{real} \{ T.type = real \}$$
$$L \rightarrow \{ L_1.inh = L.inh \} L_1, \text{id}$$
$$\cdots$$

一个语义动作在产生式中的位置决定了这个动作的执行时间

SDD与SDT

- SDD
  - 是关于语言翻译的高层次规格说明
  - 隐蔽了许多具体实现细节，使用户不必显式地说明翻译发生的顺序
- SDT
  - 可以看作是对SDD的一种补充，是SDD的具体实施方案
  - 显式地指明了语义规则的计算顺序，以便说明某些实现细节

提纲

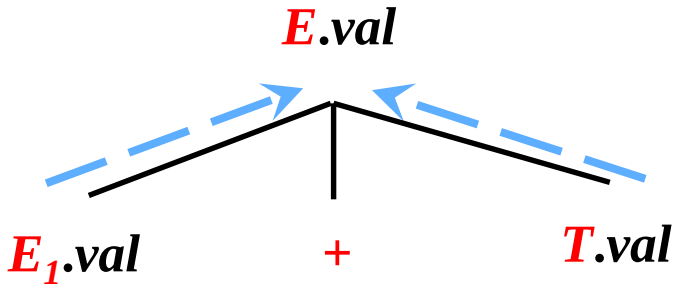
语法制导定义*SDD*

- 语法制导定义*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，用来计算该产生式中各文法符号的属性值
- 文法符号的属性
  - 综合属性 (*synthesized attribute*)
  - 继承属性 (*inherited attribute*)

综合属性(*synthesized attribute*)

- 在分析树结点N上的非终结符A的综合属性只能通过N的子结点或N本身的属性值来定义
  - 例

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

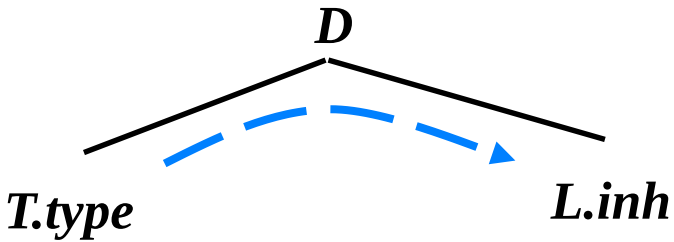


- 终结符可以具有综合属性。终结符的综合属性值是由词法分析器提供的词法值，因此在*SDD*中没有计算终结符属性值的语义规则

继承属性(*inherited attribute*)

- 在分析树结点N上的非终结符A的继承属性只能通过N的父结点、N的兄弟结点或N本身的属性值来定义
  - 例

产生式	语义规则
$D \rightarrow T\ L$	$L.inh = T.type$



- 终结符没有继承属性。终结符从词法分析器处获得的属性值被归为综合属性值

例：带有综合属性的*SDD*

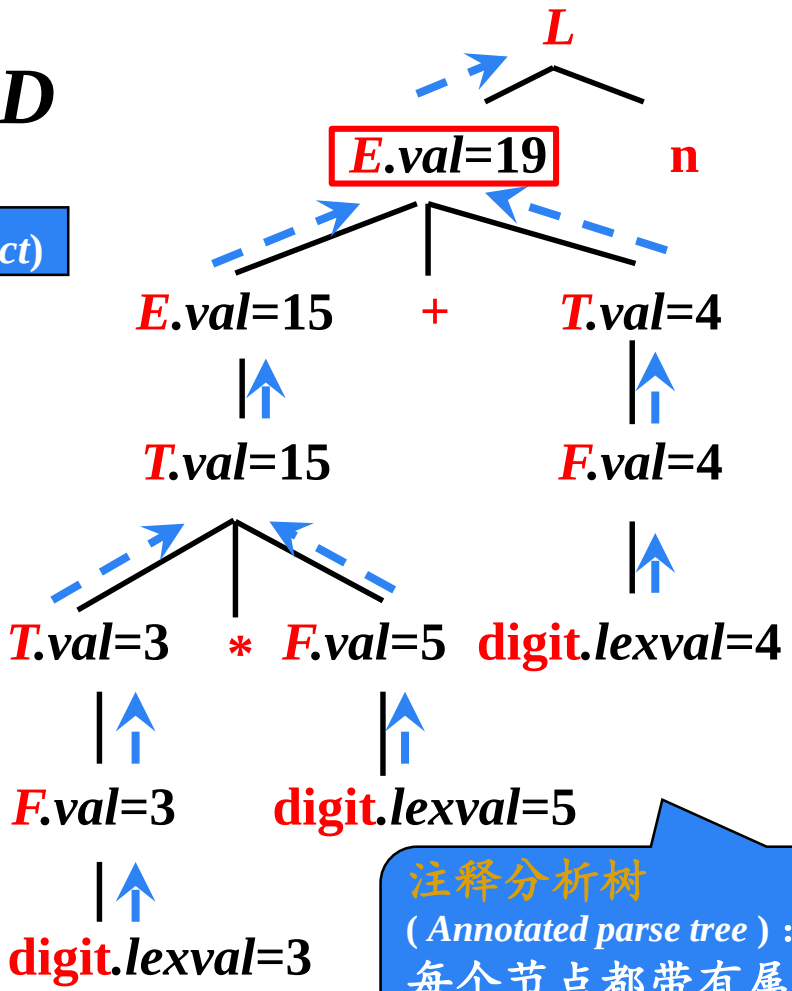
SDD：

副作用(*Side effect*)

产生式	语义规则
(1) $L \rightarrow E$ n	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) $F \rightarrow$ digit	$F.val =$ digit.*lexval*

输入：

3*5+4n

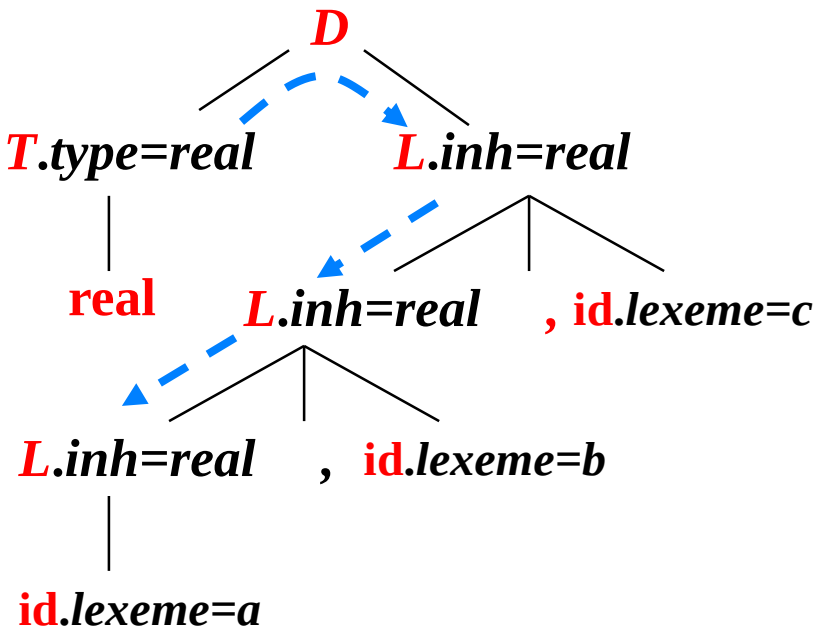


注释分析树
(*Annotated parse tree*)：
每个节点都带有属性值的分析树

例：带有继承属性*L.in*的*SDD*

***SDD*：**

	产生式	语义规则
(1)	$D \rightarrow T\ L$	*L.inh = T. type*
(2)	$T \rightarrow$ int	*T.type = int*
(3)	$T \rightarrow$ real	*T.type = real*
(4)	$L \rightarrow L_1$, id	L_1*.inh = L.inh*
		addtype(id.*lexeme*, *L.inh*)
(5)	$L \rightarrow$ id	*addtype*(id.*lexeme*, *L.inh*)



输入：

real *a* , *b* , *c*

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) $L \rightarrow E$ 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*

提纲

*SDD*的求值顺序

- *SDD*为*CFG*中的文法符号设置语义属性。对于给定的输入串x，应用语义规则计算分析树中各结点对应的属性值
- 按照什么顺序计算属性值?
  - 语义规则建立了属性之间的依赖关系，在对语法分析树节点的一个属性求值之前，必须首先求出这个属性值所依赖的所有属性值

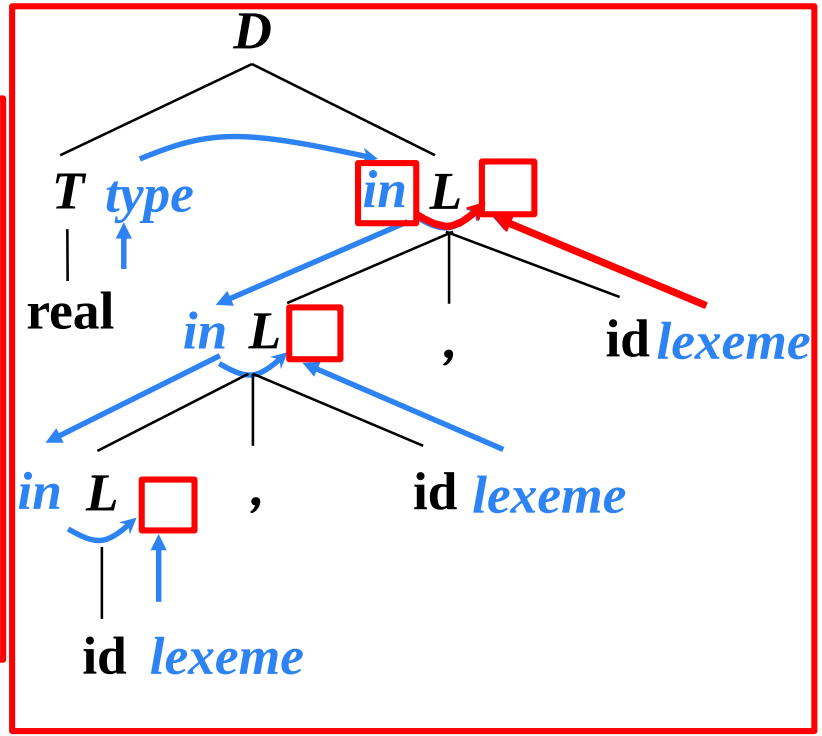
依赖图(*Dependency Graph*)

- 依赖图是一个描述了分析树中结点属性间依赖关系的有向图
- 分析树中每个标号为*X*的结点的每个属性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：

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



输入：

real *a* , *b* , *c*

属性值的计算顺序

- 可行的求值顺序是满足下列条件的结点序列$N_1, N_2, \ldots, N_k$：如果依赖图中有一条从结点N_i到N_j的边$(N_i \rightarrow N_j)$，那么$i < j$（即：在节点序列中，N_i排在N_j前面）
- 这样的排序将一个有向图变成了一个线性排序，这个排序称为这个图的拓扑排序(*topological sort*)

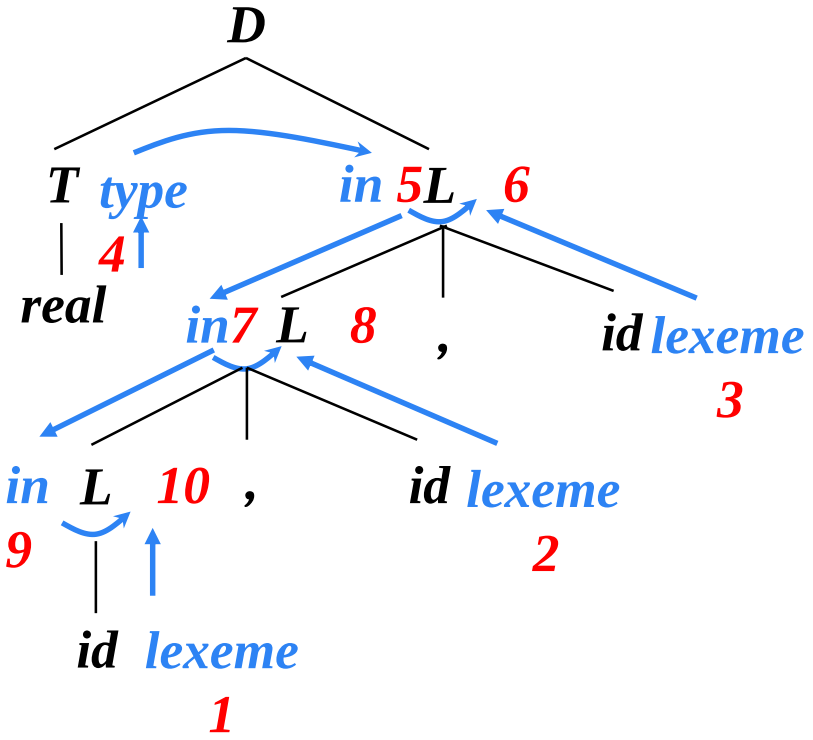
例

SDD：

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

输入：

real *a* , *b* , *c*



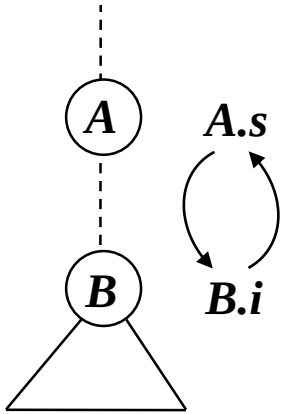
拓扑排序：

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

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

- 对于只具有综合属性的*SDD*，可以按照任何自底向上的顺序计算它们的值
- 对于同时具有继承属性和综合属性的*SDD*，不能保证存在一个顺序来对各个节点上的属性进行求值
- 例

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



如果图中没有环，那么至少存在一个拓扑排序

- 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*SDD*的属性之间存在循环依赖关系
- 幸运的是，存在一个*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环的依赖图
- 不仅如此，接下来介绍的两类*SDD*可以和自顶向下及自底向上的语法分析过程一起高效地实现
  - *S*-属性定义 (*S-Attributed Definitions, S-SDD*)
  - *L*-属性定义 (*L-Attributed Definitions, L-SDD*)

提纲

S-属性定义

- 仅仅使用综合属性的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) $L \rightarrow E$ 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.$lexval$

- 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$A \rightarrow X_1 X_2 \ldots X_n$，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 $X_1, X_2, \ldots, X_{i-1}$ 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

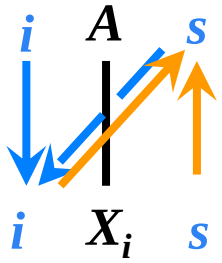
L-SDD的正式定义

- 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$A \rightarrow X_1 X_2 \ldots X_n$，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 $X_1, X_2, \ldots, X_{i-1}$ 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

为什么不能是综合属性？

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$A \rightarrow X_1 X_2 \ldots X_n$，其右部符号$X_i (1 \le i \le n)$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



例： *L-SDD*

继承属性

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

综合属性

非L属性的SDD

➢例

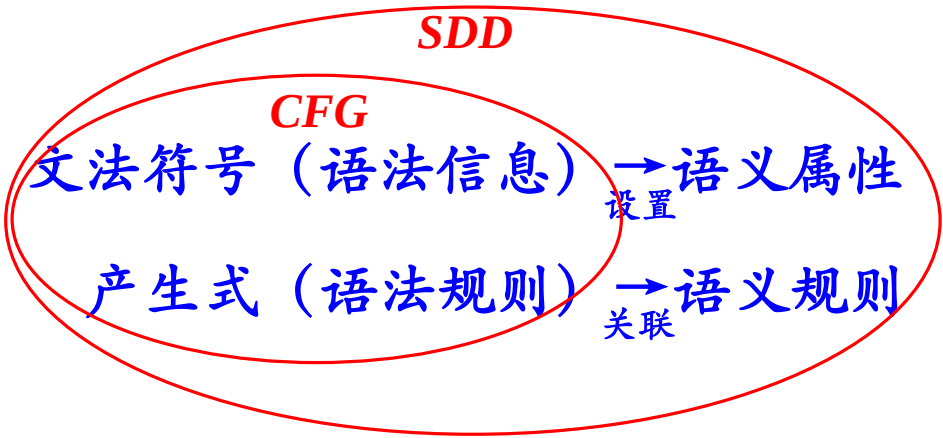
产生式	语义规则
(1) $A \rightarrow LM$	$L.i = l(A.i)$
	$M.i = m(L.s)$
	$A.s = f(M.s)$
(2) $A \rightarrow QR$	$R.i = r(A.i)$
	$Q.i = q(R.s)$ ×
	$A.s = f(Q.s)$

继承属性

综合属性

SDD要点小结-语义分析要解决的问题

- 如何表示语义信息?
- 如何计算语义信息(语义属性)?



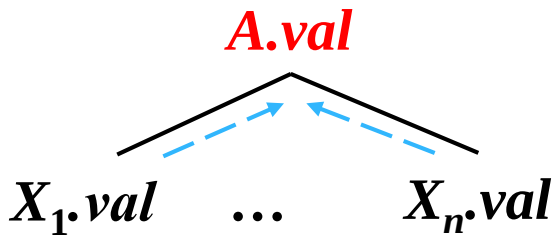
SDD是对CFG的扩展

产生式	语义规则
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow$ int	$T.type = int$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.inh = L.inh$
…	…

语义属性

- 综合属性

  A：

  a：词法值　token：〈种别码，属性值〉

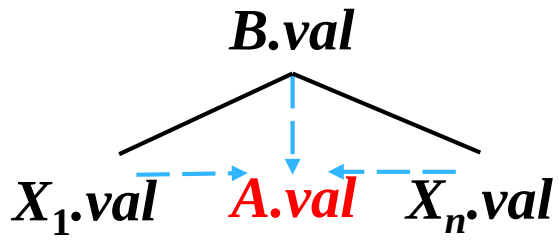
  语法分析关注

  语义分析关注

  词法值

- 继承属性

  A：

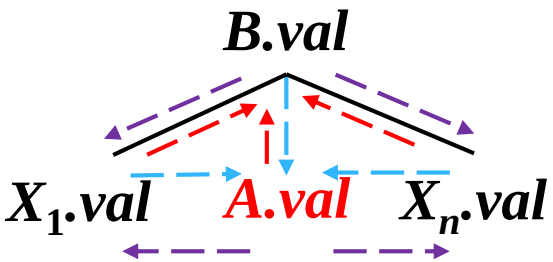
  

  a：无

*SDD*属性的求值顺序

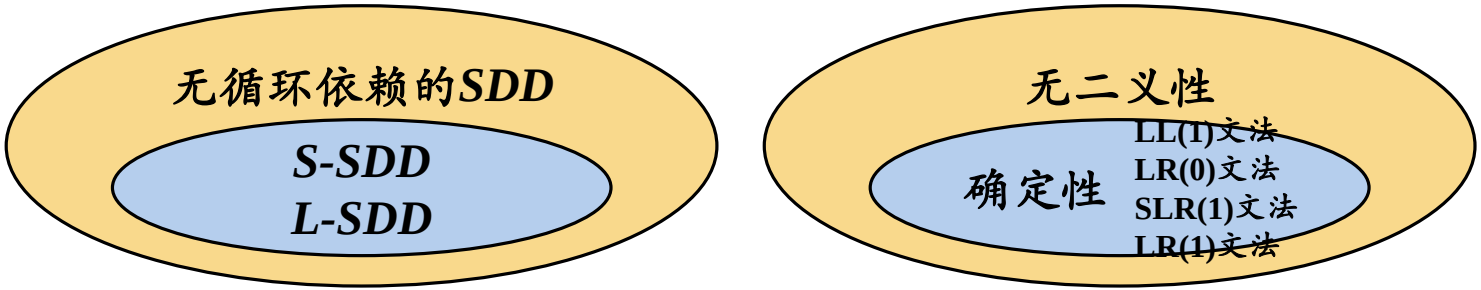
不含继承属性：　∀bottom_up顺序
（只有综合属性）

包含继承属性：　不保证存在一个拓扑排序



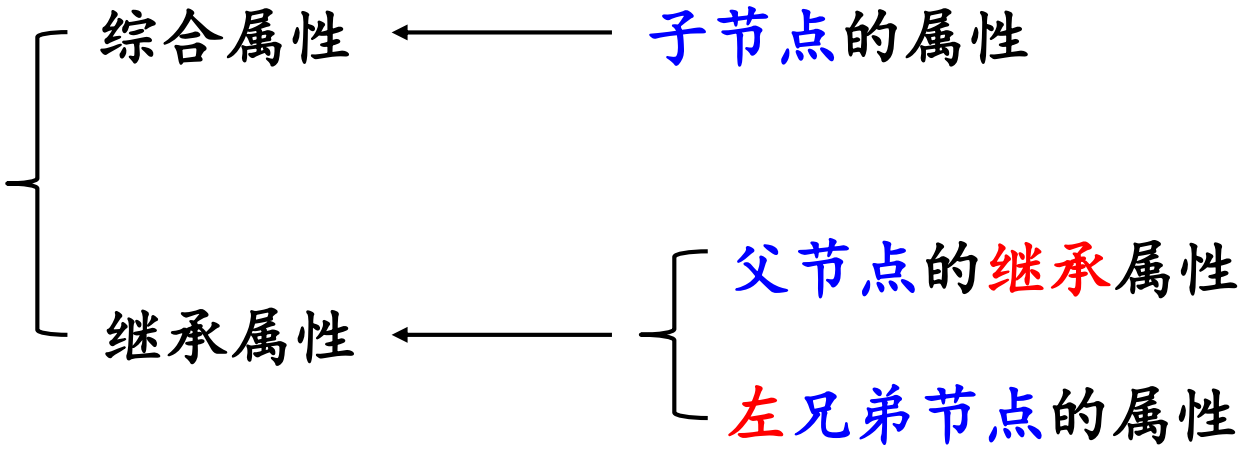
循环依赖*SDD*的判定

- 很难（计算角度）
- 但能给出一组充分条件，满足这组充分条件的*SDD*是无循环依赖的



S-SDD与L-SDD

- S-SDD：仅仅使用综合属性的SDD
- L-SDD：



如何判断语义规则中定义的一个属性是综合属性还是继承属性？

➢ 例

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

Exercise

1文法G[S]及其语法制导翻译定义如下：

产生式	语义动作
S’ → S	print(S.num)
S → (L)	S.num = L.num +1
S → a	S.num = 0
L →L(1), S	L.num = L(1).num + S.num
L →S	L.num = S.num

若输入为(a,(a))，且采用自底向上的分析方法，则输出为(　　)。

A.0　　B.1　　C.2　　D.4

Exercise

2使用(　)可以定义一个程序的意义。
A.语义规则
B.词法规则
C.产生规则
D.词法规则

3以下说法正确的是(　)。
A.语义规则中的属性有两种：综合属性与继承属性
B.终结符只有继承属性，它由词法分析器提供
C.非终结符可以有综合属性，但不能有继承属性
D.属性值在分析过程中可以进行计算，但不能传递

提纲

回顾-语法制导翻译

- 语义分析要解决的两个问题
  - 语义信息的表示：语义属性（表达式的值val、变量的类型type、…）
  - 语义信息的计算：语义规则

SDD = CFG + 语义属性 + 语义规则

- 无循环依赖SDD的判定
  - 充分条件　继承属性不依赖右兄弟节点的属性和父节点的综合属性
    - S-SDD　if LR文法　then LR分析+语义翻译
    - L-SDD　if LL(1)文法　then { LL(1)分析+语义翻译 / bottom_up分析+语义翻译

语法制导翻译方案*SDT*

- 语法制导翻译方案（*SDT*）　*SDD*的具体实施方案

  *SDD*定义了各属性的计算方法（计算规则）怎么算？

  *SDT*进一步明确了各属性的计算时机（计算顺序）怎么算？＋何时算？

- 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：

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

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

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

$L \rightarrow \{ L_1.inh = L.inh \} L_1, \text{id}$

…

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) $L \rightarrow E$ 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.$lexval$}

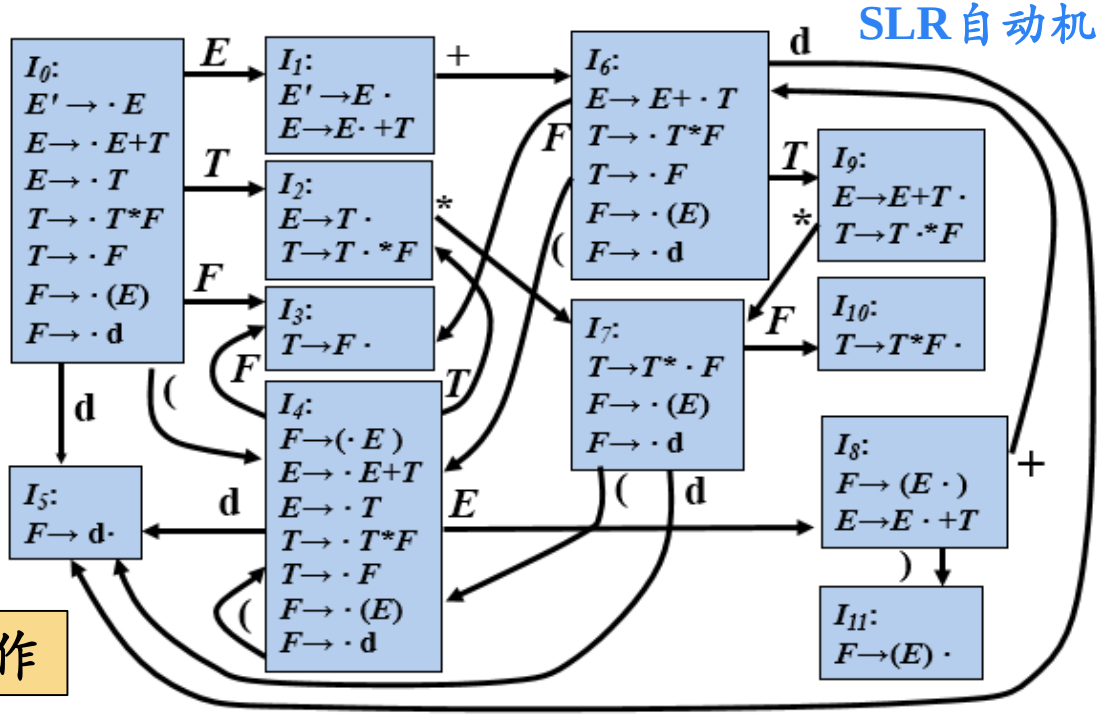
S-属性定义的SDT实现

- 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

S-SDD

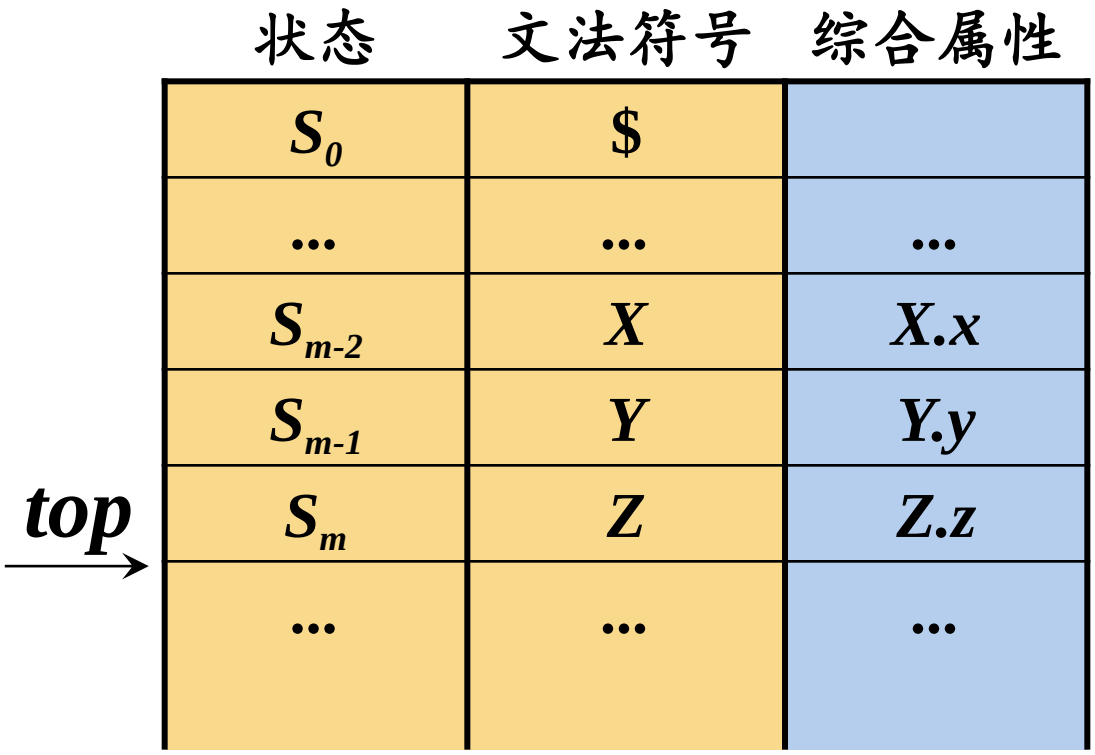
产生式	语义规则
(1) $L \to E$ n	$L.val = E.val$
(2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \to T$	$E.val = T.val$
(4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \to F$	$T.val = F.val$
(6) $F \to (E)$	$F.val = E.val$
(7) $F \to$ digit	$F.val =$ digit.*lexval*

当归约发生时执行相应的语义动作



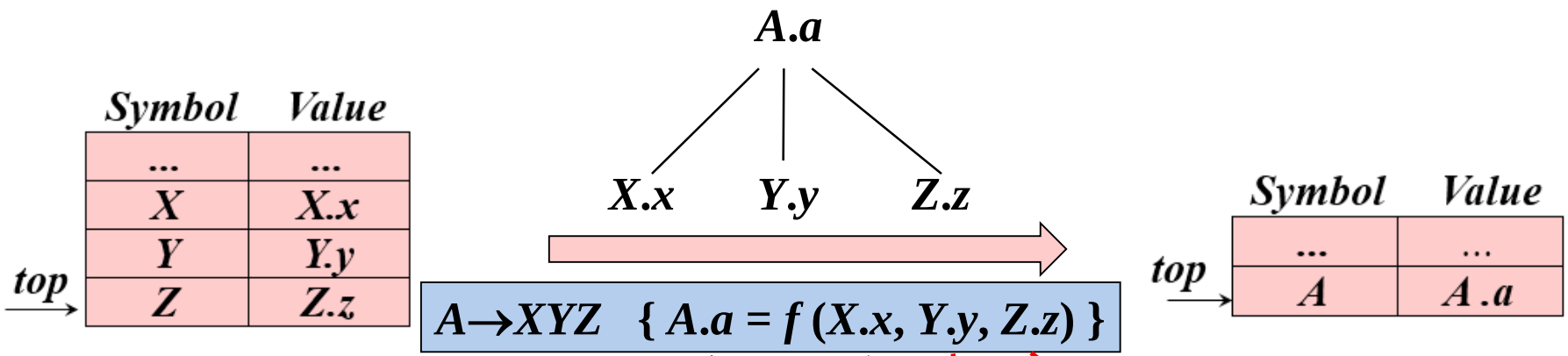
扩展的LR语法分析栈

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[*top*-2].*symb* =*A*

stack[*top*-2].*val* = *f*(*stack*[*top*-2].*val*, *stack*[*top*-1].*val*, *stack*[*top*].*val*)

top =*top*-2;

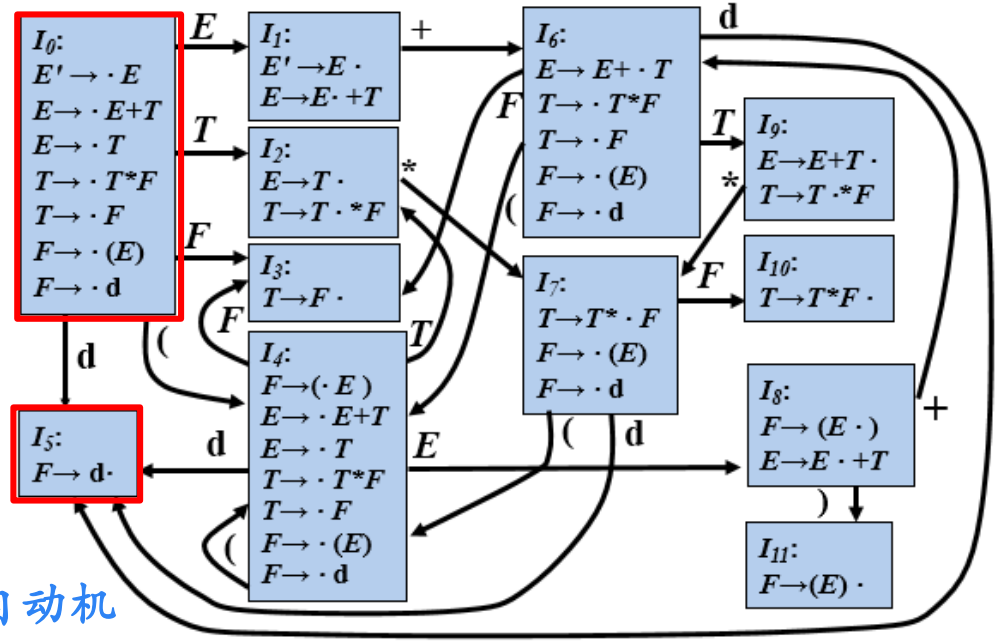
例：在自底向上语法分析栈中实现桌面计算器

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*	
(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*	
(6)$F \rightarrow (E)$	*F.val* = *E.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-1].*val*; *top*=*top*-2; }
(7)$F \rightarrow$ digit	*F.val* = digit.*lexval*	

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	

输入： 3*5+4

状态	符号	属性
0	$	_
5	d	3

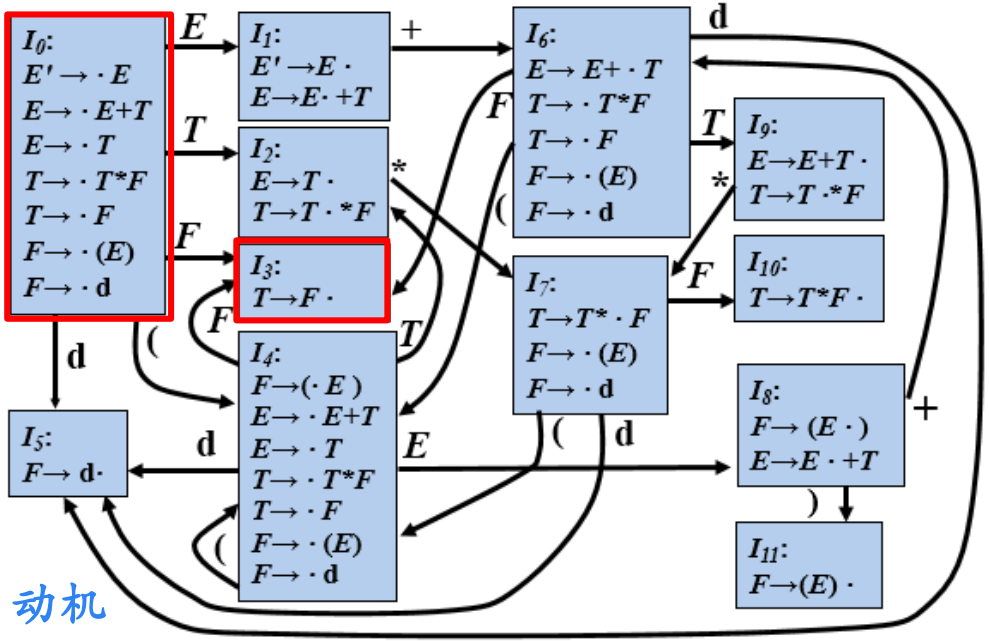


SLR自动机

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* × *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	

输入： 3*5+4

状态	符号	属性
0	$	_
3	F	3

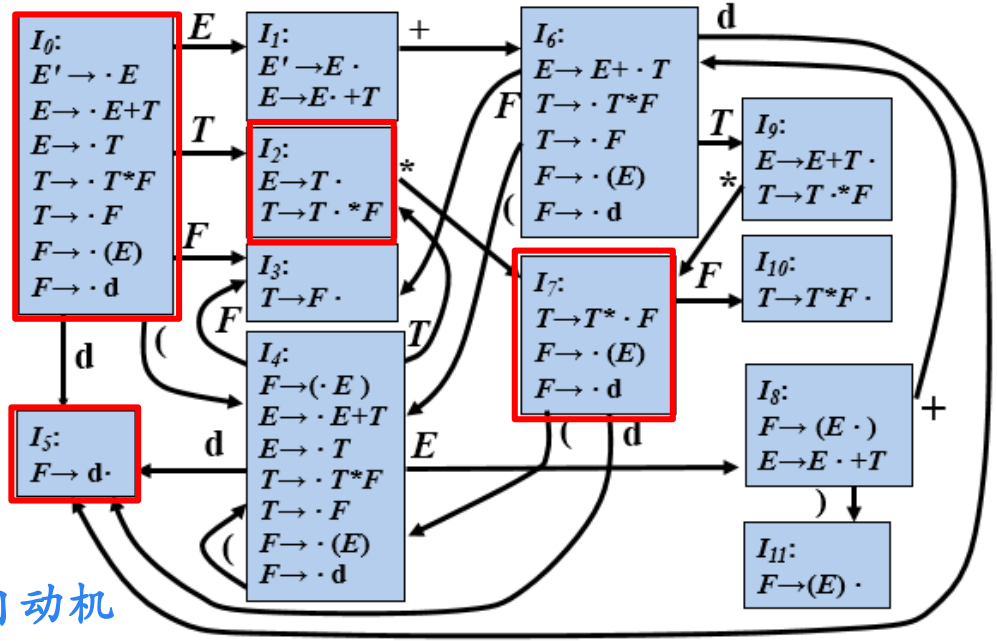


SLR自动机

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* × *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	

输入： 3*5+4

状态	符号	属性
0	$	_
2	*T*	3
7	*	_
5	d	5

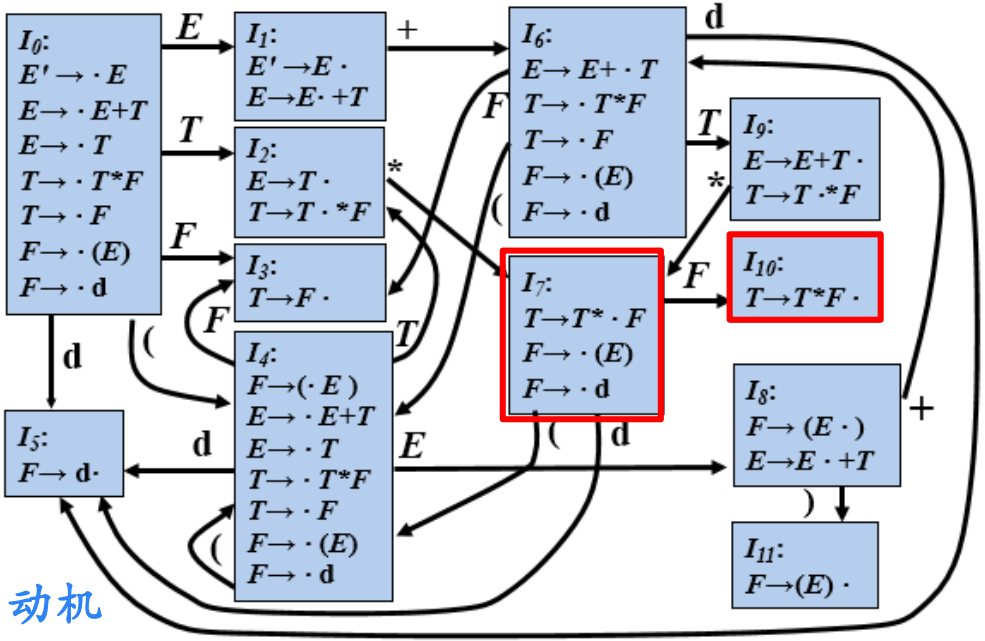


SLR自动机

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* × *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	

输入： 3*5+4

状态	符号	属性
0	$	_
2	*T*	15
7	*	_
10	*F*	5

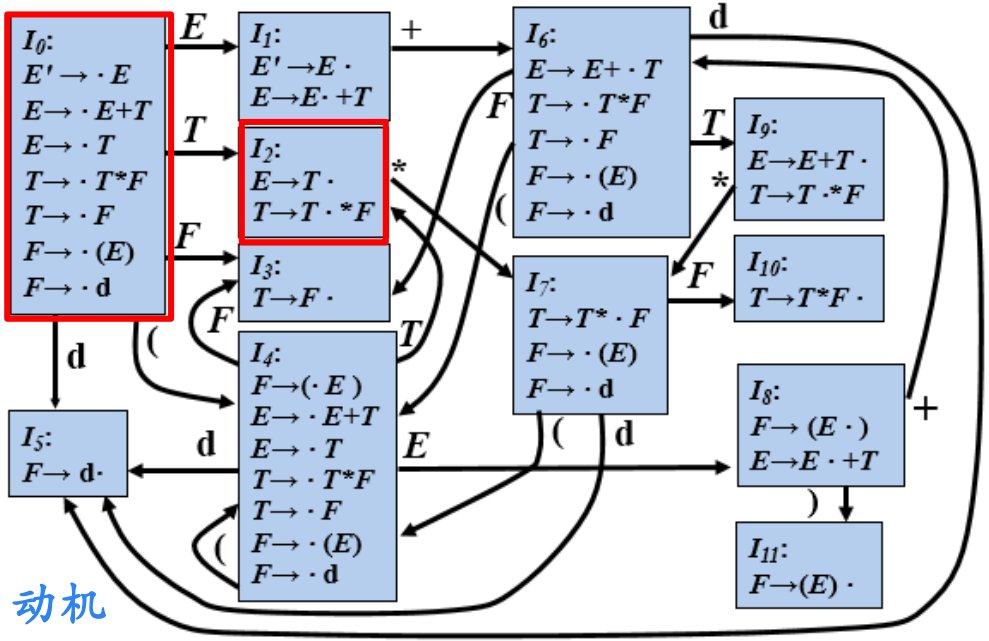


SLR自动机

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* × *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	

输入： 3*5+4

状态	符号	属性
0	$	_
2	*T*	15

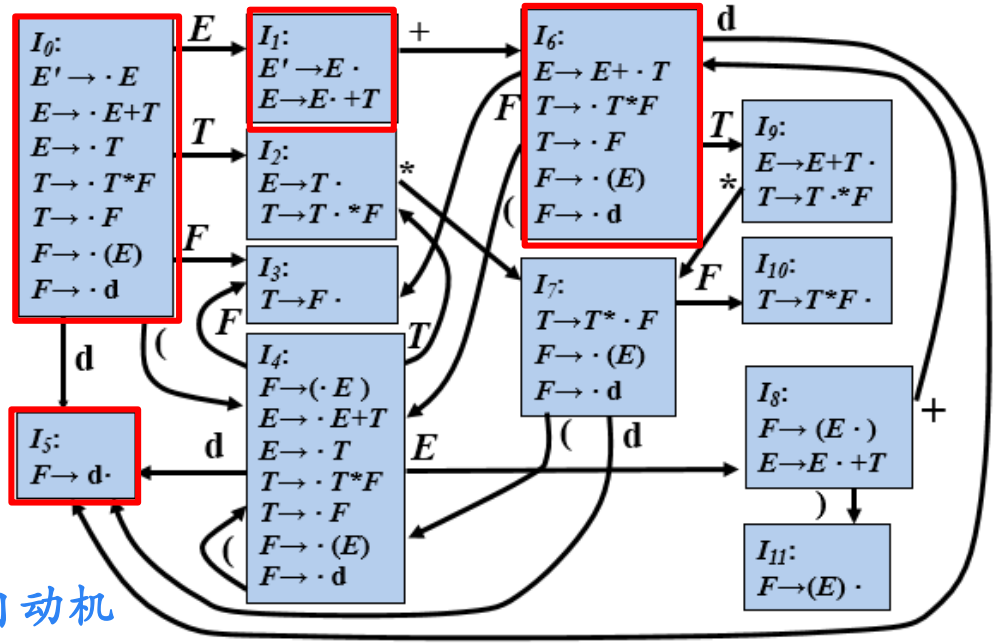


SLR自动机

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* × *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	

输入： 3*5+4

状态	符号	属性
0	$	_
1	*E*	15
6	+	_
5	d	4

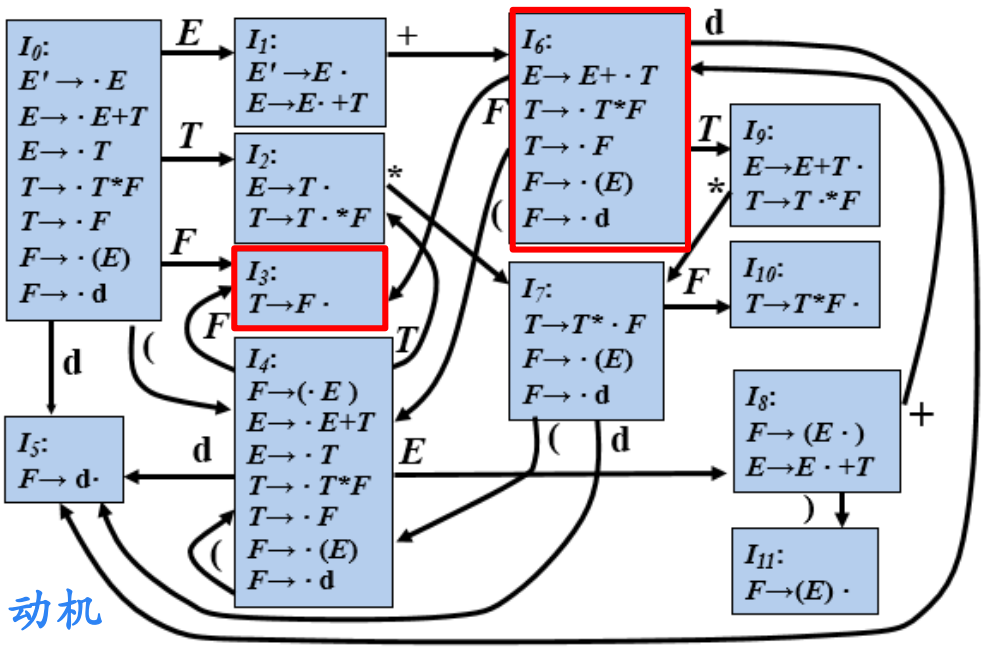


SLR自动机

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* × *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	

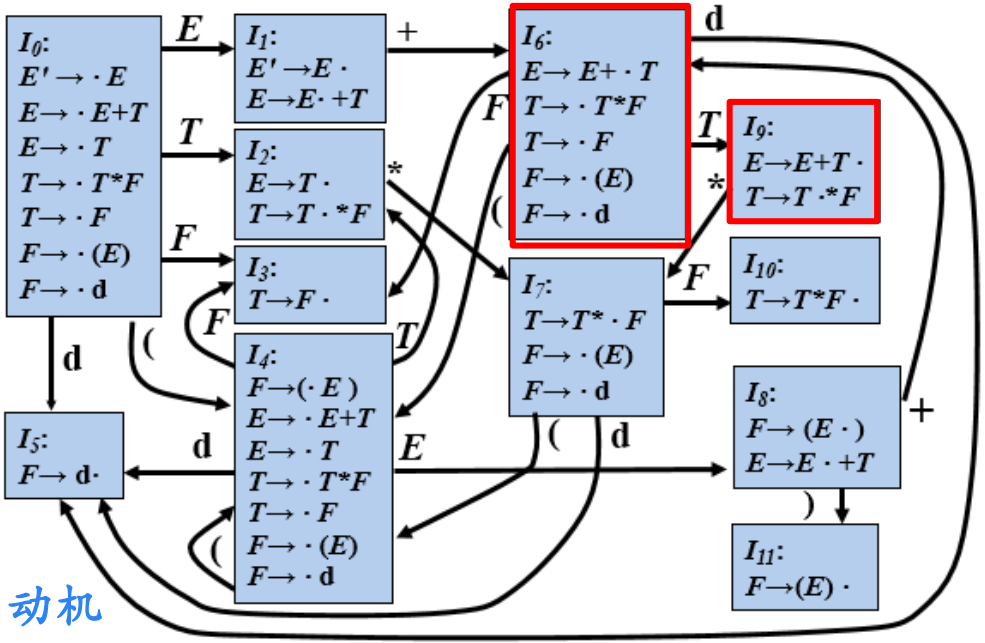
输入： 3*5+4

状态	符号	属性
0	$	_
1	*E*	15
6	+	_
3	*F*	4



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* × *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

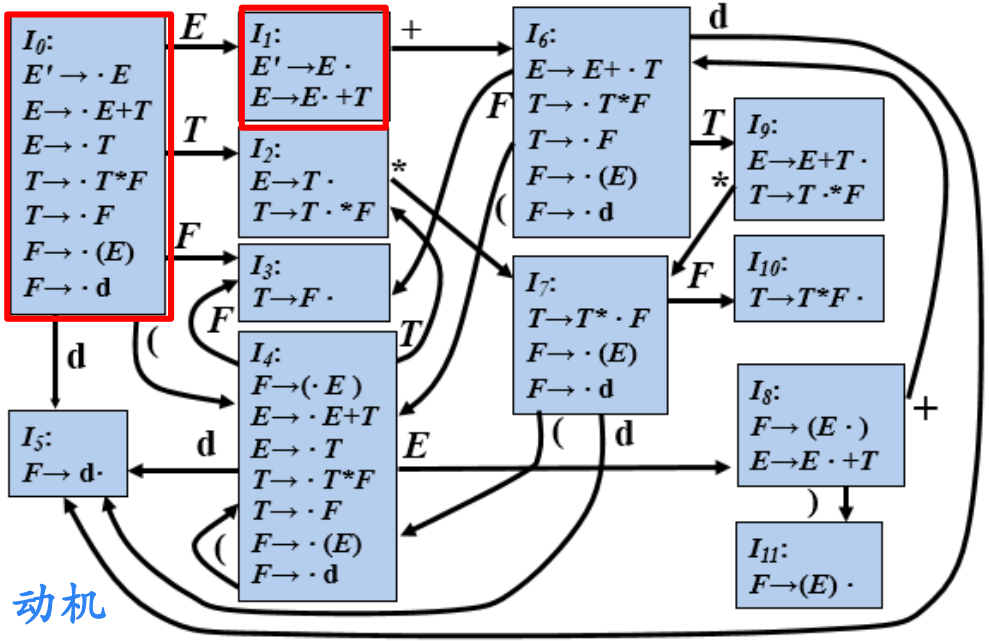
输入： 3*5+4

状态	符号	属性
0	$	_
1	*E*	19
6	+	_
9	*T*	4

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	

输入： 3*5+4

状态	符号	属性
0	$	_
1	*E*	19



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

➢ *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* $\}$

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*或*LR*语法分析过程中实现
  - 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 }

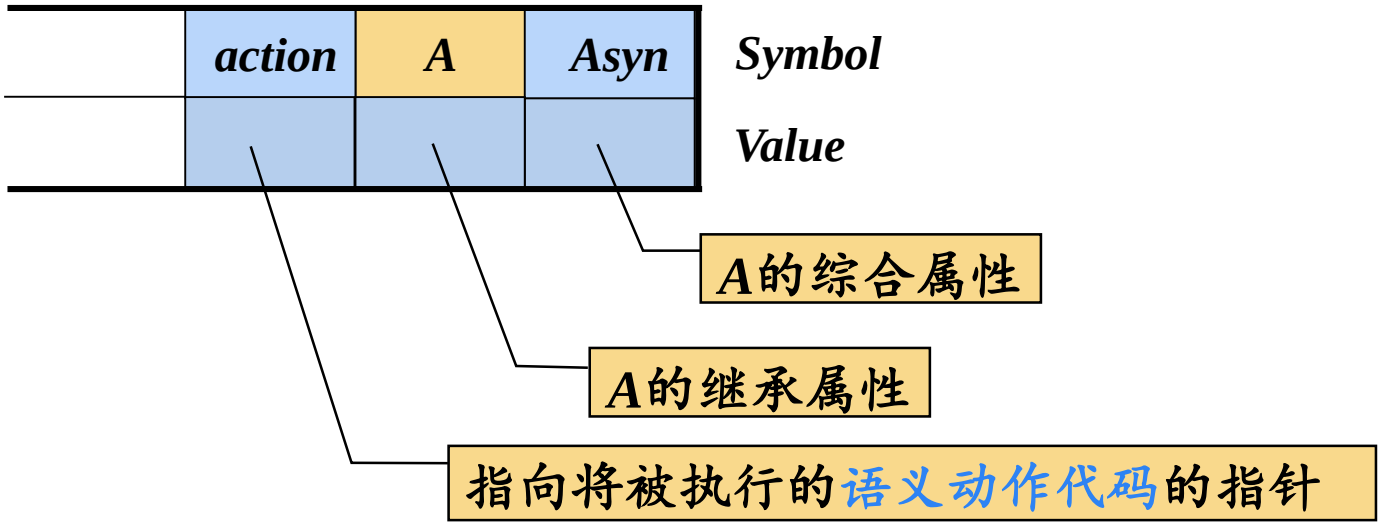
L-属性定义的*SDT*实现

- 如果一个*L-SDD*的基本文法可以使用*LL*分析技术，那么它的*SDT*可以在*LL*或*LR*语法分析过程中实现
  - 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
  - 在*LR*分析过程中进行语义翻译

提纲

在非递归的预测分析过程中进行翻译

- 扩展语法分析栈



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.$lexval$

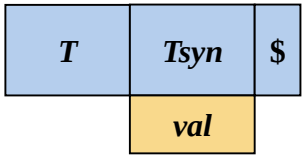
例

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) $F \rightarrow$ 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



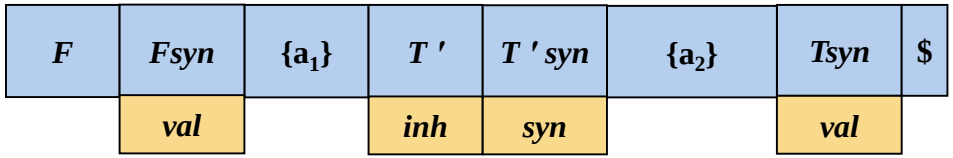
例

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



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: $T.val = T'.syn$

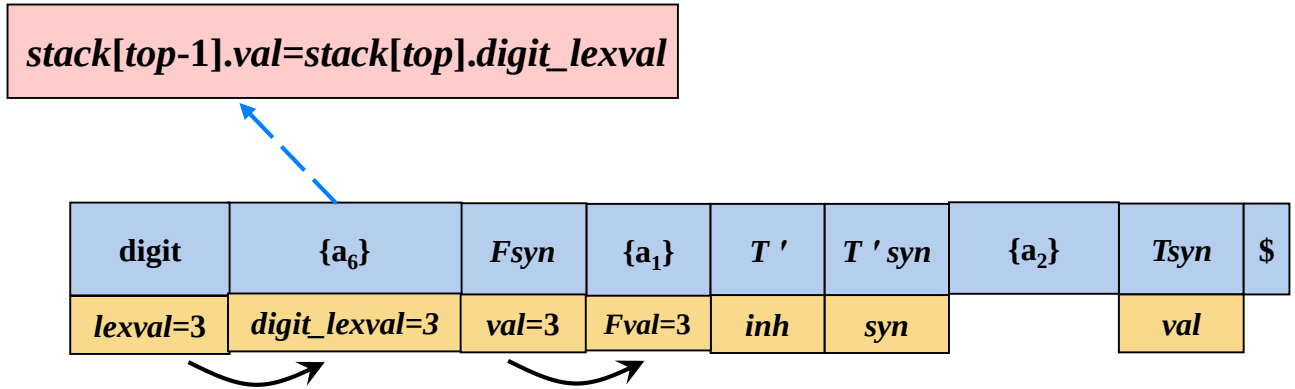
a_3: $T_1'.inh = T'.inh \times F.val$

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

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

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

输入: 3 * 5



例

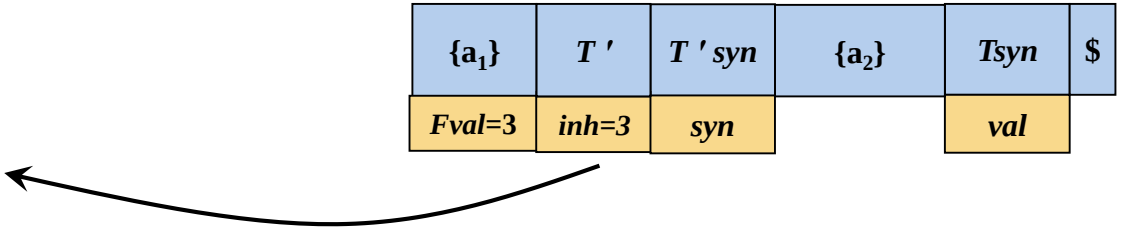
SDT

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



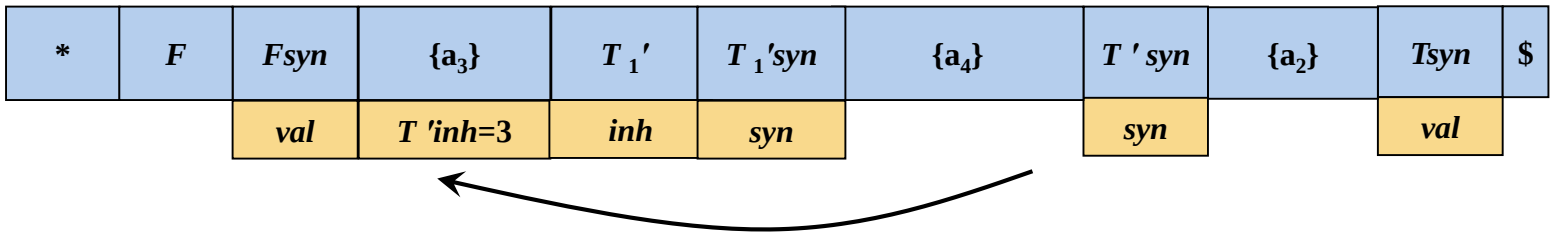
例

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



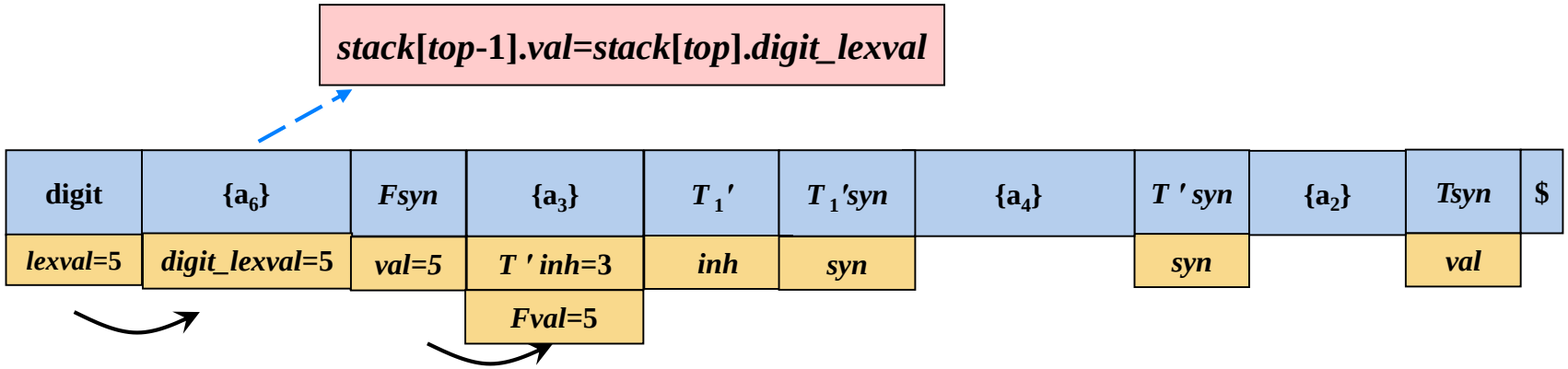
例

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



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

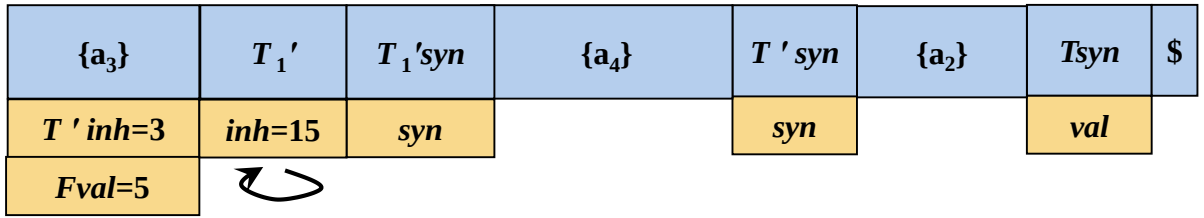
a_4 : $T'.syn = T_1'.syn$

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

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

输入: 3 * 5

stack[*top*-1].*inh*=*stack*[*top*].*T'inh*×*stack*[*top*].*Fval*



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 : $T.val = T'.syn$

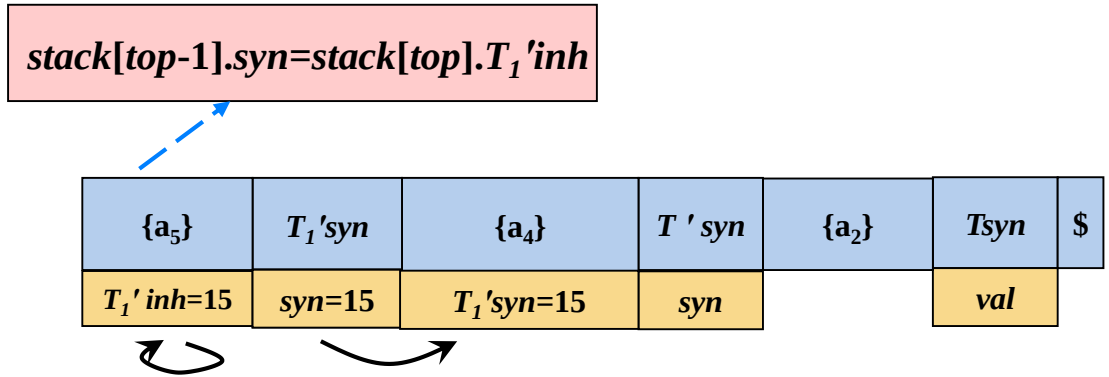
a_3 : $T_1'.inh = T'.inh \times F.val$

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

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

a_6 : $F.val =$ digit.*lexval*

输入: 3 * 5



例

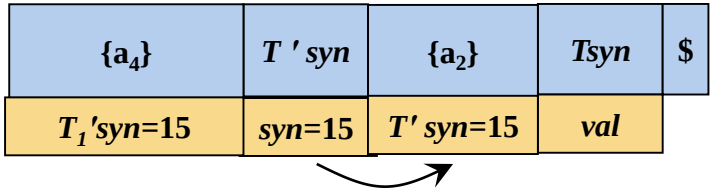
SDT

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

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

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

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

分析栈中的每一个记录都对应着一段执行代码

- 综合记录出栈时，要将综合属性值复制给后面特定的语义动作
- 变量展开时（即变量本身的记录出栈时），如果其含有继承属性，则要将继承属性值复制给后面特定的语义动作

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

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
F		
Fsyn	*val*	*stack*[*top*-1].*Fval* = *stack*[*top*].*val*； *top*=*top*-1;
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*;
T'syn	*syn*	*stack*[*top-1*].*T'syn* = *stack*[*top*].*syn*； *top*=*top*-1;
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) $F \rightarrow$ 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
*		
F		
$Fsyn$	val	$stack[top\text{-}1].Fval = stack[top].val$； $top=top\text{-}1$;
a_3	$T'inh$; $Fval$	$stack[top\text{-}1].inh = stack[top].T'inh \times stack[top].Fval$； $top=top\text{-}1$;
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): $stack[top].T'inh = stack[top].inh$;
$T_1'syn$	syn	$stack[top\text{-}1].T_1'syn = stack[top].syn$； $top=top\text{-}1$;
a_4	$T_1'syn$	$stack[top\text{-}1].syn = stack[top].T_1'syn$； $top=top\text{-}1$;

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*

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) $F \rightarrow$ 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.$lexval$

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

符号	属性	执行代码
digit	*lexval*	*stack*[*top*-1].*digitlexval* = *stack*[*top*].*lexval*; *top*=*top*-1;
a_6	*digitlexval*	*stack*[*top*-1].*val* = *stack*[*top*].*digitlexval*; *top*=*top*-1;

提纲

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;
    Getnext(token);
    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;
}
```

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;
    Fval = F(token);
     T'inh = Fval;
    Getnext(token);
    T'syn = T1'(token, T'inh);
    Tval = T'syn;
    return Tval;
}
```

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 ≠ digit then Error;
    Fval=token.lexval;
    return Fval;
}
```

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

```
Desent()
{
    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$c := B(b_1, b_2, \ldots, b_k)$，其中，$b_1, b_2, \ldots, b_k$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

提纲

L-属性定义的自底向上翻译

- 给定一个以*LL*文法为基础的*L-SDD*，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*

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 $.lexval$ }

修改后的SDT，所有语义动作都位于产生式末尾

标记非终结符
(*Marker Nonterminal*)

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.$lexval$ }

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

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

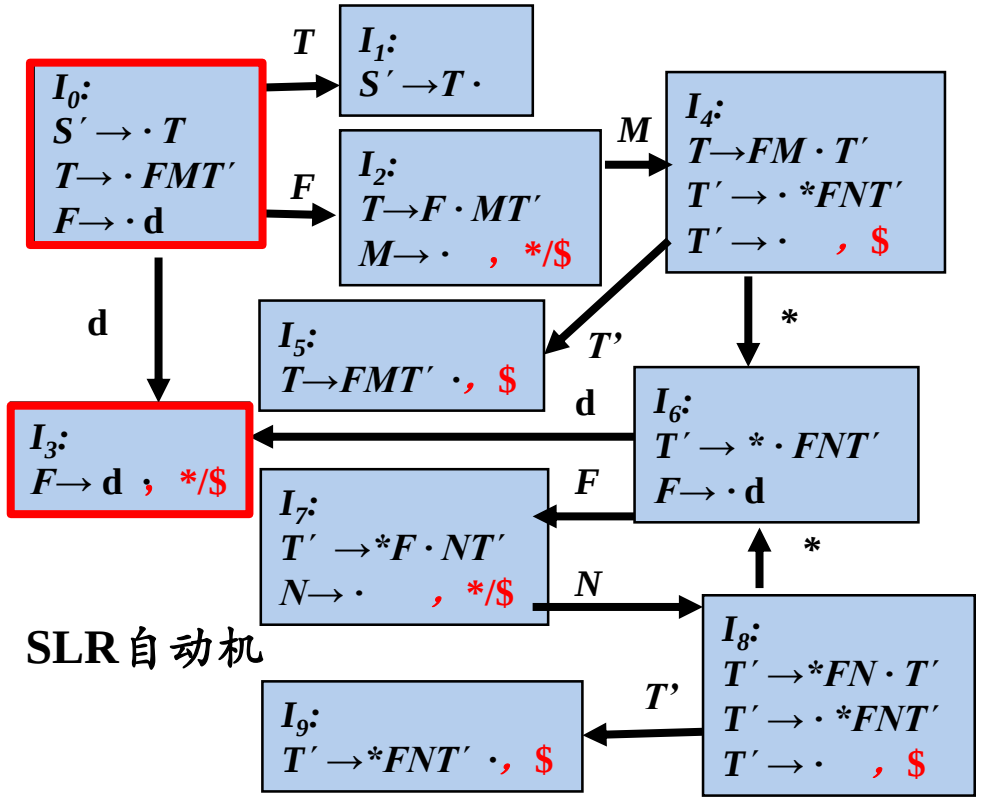
$N.i2 = F.val;$

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

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

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

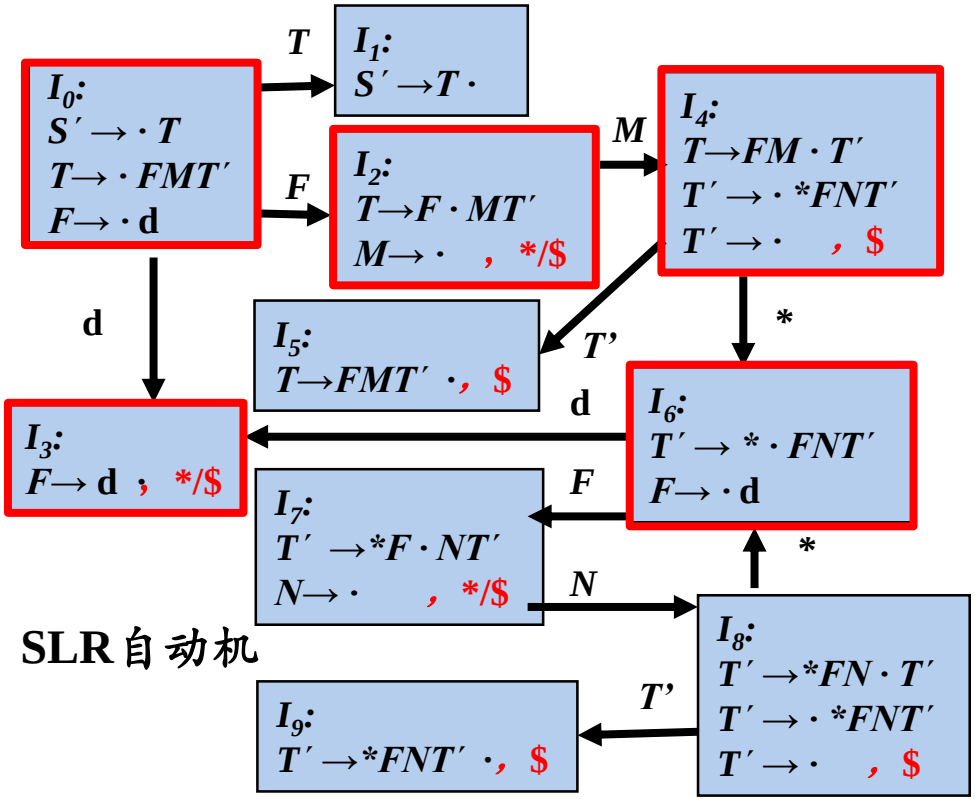
输入： 3 * 5



SLR自动机

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

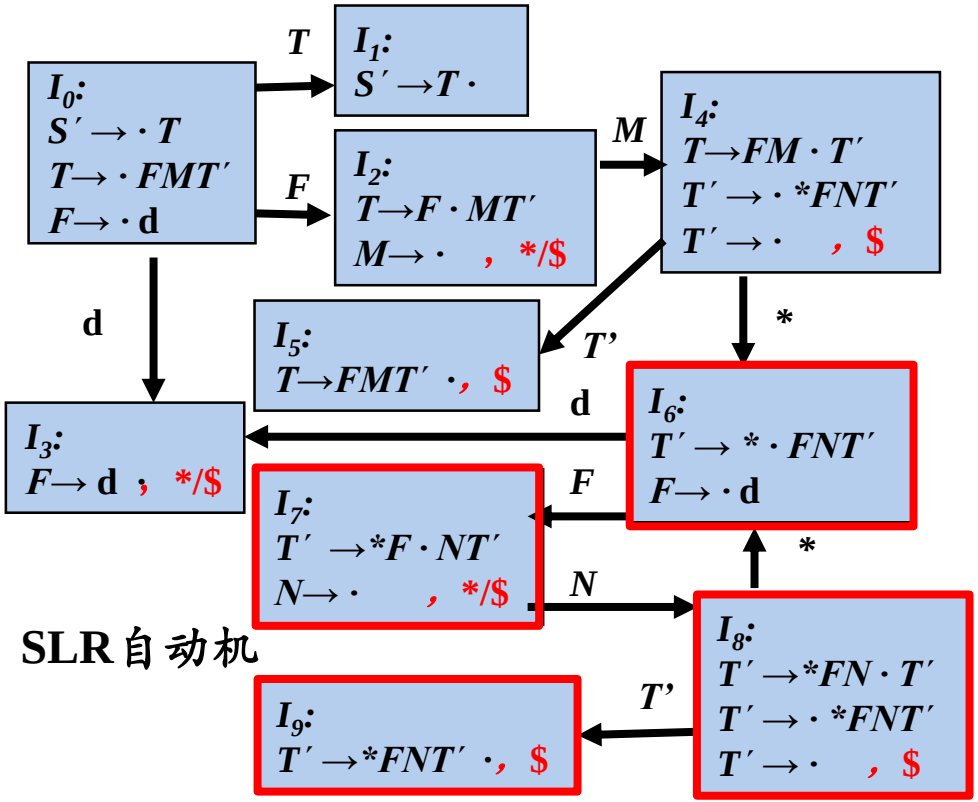
$M \rightarrow \varepsilon\ \{M.i = F.val;\ M.s = M.i\ \}$

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 $\}$



SLR自动机

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

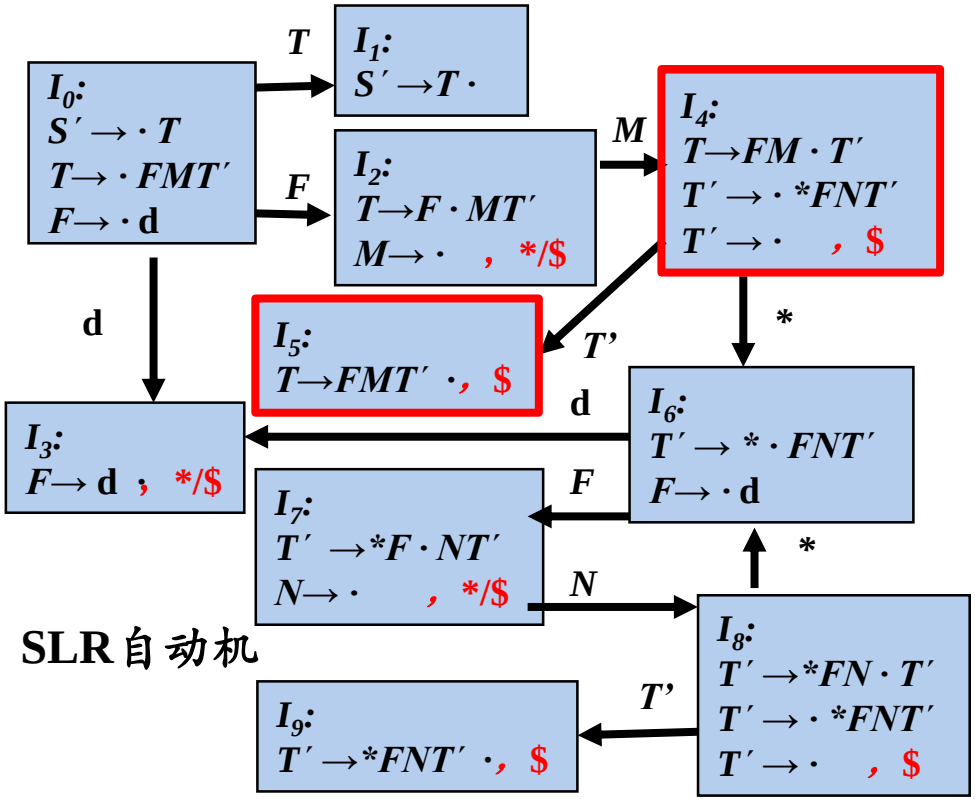
   $N \rightarrow \varepsilon\ \{N.i1 = T'.inh;$

   $N.i2 = F.val;$

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

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

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

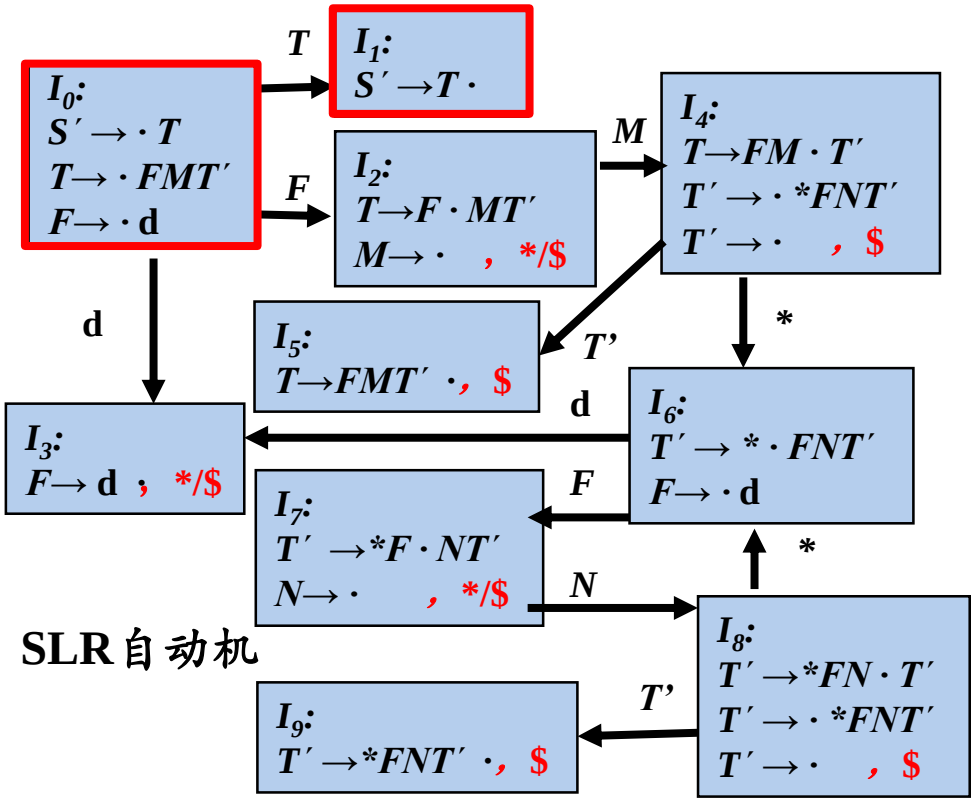


SLR自动机

输入： 3 * 5

0	2	4	5
$	F	M	T′
	3	T′inh=3	syn=15

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

4) $F \rightarrow$ digit {$stack[top].val = stack[top].lexval$;}

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$M \rightarrow \varepsilon$
- 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$M \rightarrow \varepsilon$ 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

本章小结

- 语义分析要解决的两个问题
  - 语义信息的表示：语义属性（表达式的值val、变量的类型type、…）
  - 语义信息的计算：语义规则

SDD = CFG + 语义属性 + 语义规则

- 无循环依赖SDD的判定
  - 充分条件　继承属性不依赖右兄弟节点的属性和父节点的综合属性
    - S-SDD　if LR文法　then LR分析+语义翻译
    - L-SDD　if LL(1)文法　then { LL(1)分析+语义翻译 { 递归的 / 非递归的 ; bottom_up分析+语义翻译 }

本章小结

➢ 语法制导翻译方案（*SDT*）

SDD的具体实施方案

*SDD*定义了各属性的计算方法（计算规则）怎么算？

*SDT*进一步明确了各属性的计算时机（计算顺序）怎么算？＋何时算？

无循环依赖SDD的SDT

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*

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* $\}$

S-SDD的SDT

- 综合属性的计算时机
  - 所有子节点分析完
  - 语义动作置于产生式末尾

L-SDD的SDT

- 继承属性的计算时机
  - 即将分析A之前
  - 语义动作置于紧靠在A的本次出现之前的位置上
- 综合属性的计算时机
  - 所有子节点分析完毕
  - 语义动作置于产生式末尾

S-SDD的自底向上翻译

***S-SDD*： if *LR*文法 then *LR*分析+语义翻译**

if *LR*文法 + *S-SDD* then *LR*分析+语义翻译

- 语法分析器的扩展
  - 为每个栈记录增加属性值字段，存放文法符号的综合属性值
  - 在每次归约时调用计算综合属性值的语义子程序

L-SDD的自顶向下翻译　　LL(1)文法 + *L-SDD*

L-SDD： if *LL*(1)文法 then *LL*(1)分析+语义翻译

- LL(1)分析+语义翻译
  - 非递归的
  - 递归的
- bottom_up分析+语义翻译

➢在预测分析的同时实现语义翻译

➢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

在非递归的预测分析过程中进行翻译

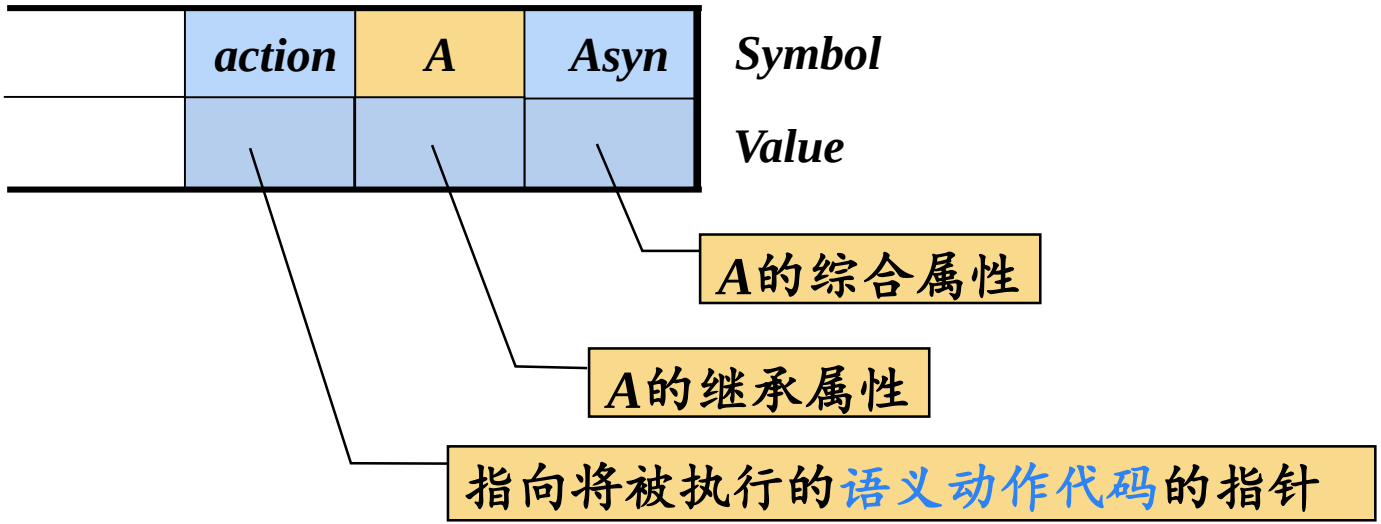
- 扩展语法分析栈

(1)增加属性值（value）字段

(2)将继承属性和综合属性存放在不同的记录中

(3)增加动作记录用来存放语义动作代码的指针

(4)不只是动作记录，分析栈中的每一个记录都对应着一段执行代码



分析栈中的每一个记录都对应着一段执行代码

- 综合记录出栈时，要将综合属性值复制给后面特定的语义动作
- 变量展开时（即变量本身的记录出栈时），如果其含有继承属性，则要将继承属性值复制给后面特定的语义动作

L-SDD的自顶向下翻译　　LL(1)文法 + *L-SDD*

L-SDD： if *LL*(1)文法 then *LL*(1)分析+语义翻译

- LL(1)分析+语义翻译
  - 非递归的
  - 递归的
- bottom_up分析+语义翻译

➢在预测分析的同时实现语义翻译

➢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

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

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;
}
```

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 = T1'(token, T'inh);
    Tval = T'syn;

    return Tval;
}
```

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 ≠ "digit" then Error;
    Fval=token.lexval;
    Getnext(token);
     return Fval;
}
```

L-SDD的自底向上翻译　　LL(1)文法 + *L-SDD*

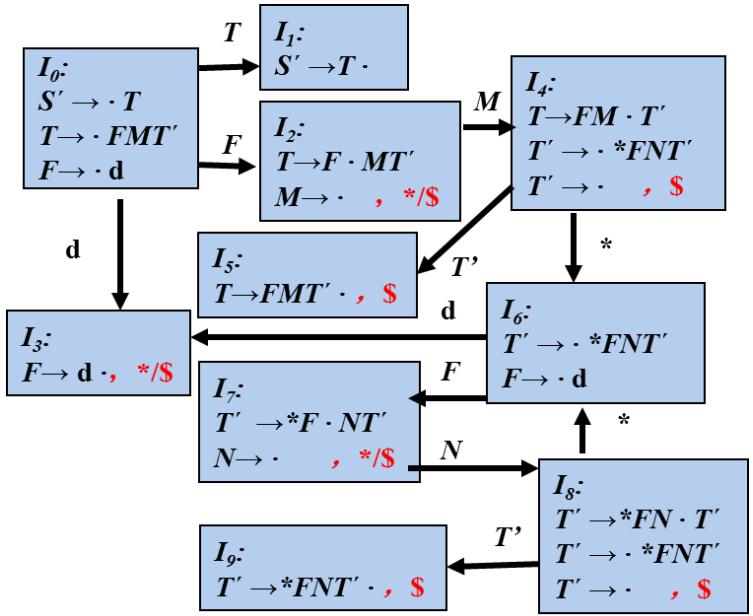
***L-SDD*：　if *LL*(1)文法　then *LL*(1)分析+语义翻译**

- LL(1)分析+语义翻译
  - 非递归的
  - 递归的
- bottom_up分析+语义翻译

L-属性定义的自底向上翻译

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



本章小结

- 语法制导定义SDD
- S-属性定义与L-属性定义
- 语法制导翻译方案SDT
- S-属性定义的自底向上翻译
- 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

Exercise

1在递归的预测分析过程中进行翻译，以下说法不正确的是(　　)。

A.可以将一个递归的预测分析器扩展为一个翻译器

B.在语法分析器中，每个非终结符A对应一个过程，在做语义分析时，要将过程扩展成一个函数

C.以继承属性作为函数的参数，以综合属性作为函数的返回值

D.以综合属性作为函数的参数，以继承属性作为函数的返回值

Exercise

2在递归的预测分析过程中进行翻译，以下说法不正确的是(　)。

A.在语法分析器中，每个非终结符A对应一个过程，在做语义分析时，要将过程扩展成一个函数
B.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
C.如果非终结符含有继承属性，需要将函数调用的返回值赋给相应的局部变量
D.对于产生式右部的每个动作，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

Exercise

3以下说法不正确的是(　)。

A.语法制导翻译方案只限自底向上的分析方法
B.给定一个以LL文法为基础的L-SDD，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
C.对于LL文法中内嵌的语义动作，向文法中引入一个标记非终结符M来替换它，进行自底向上的翻译
D.每个标记非终结符M对应着一个空产生式M→ε，该产生式对应着一段语义子程序，它的任务就是完成M所替换的那个语义动作要完成的工作