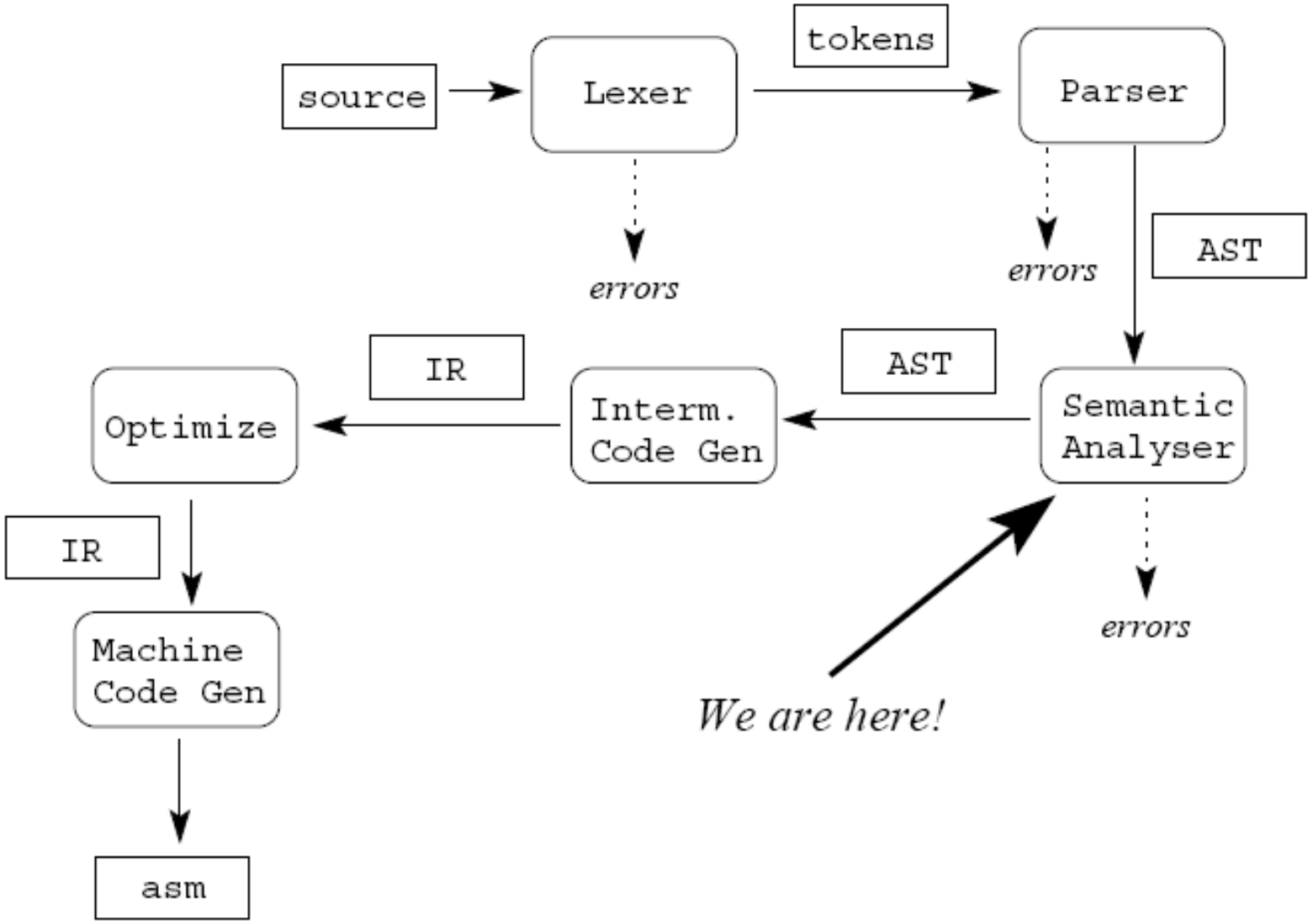
第5章 语法制导翻译

西北大学信息学院计算机科学系 付丽娜

语法制导翻译


source
Lexer
tokens
Parser
errors
errors
AST
IR
AST
Optimize
Interm.
Code Gen
Semantic
Analyser
errors
IR
Machine
Code Gen
We are here!
asm

Overview

- 本章讨论用上下文无关文法引导的语言翻译
  - 第6章继续讨论将翻译技术用于类型检查和中间代码生成
- 属性
  - 附加在文法符号上，把信息和语言构造联系起来
- 语法制导定义——翻译的规约
  - 通过与文法产生式相关的语义规则描述属性的值
- 语法制导的翻译方案——翻译的实现
  - 在产生式右部嵌入称为语义动作的程序片段
- 语法制导翻译方法
  - 构造语法分析树，通过访问这棵树的各个结点计算结点的属性值

SDD&SDT

	PRODUCTION	SEMANTIC RULES
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

```
S    :    S E '\n'      {printf("ans=%d\n",$2);}
     |    /*enpty*/      {/*enpty*/}
     ;
E    :    E '+' E       {$$ = $1+$3;}
     |    E '-' E       {$$ = $1-$3;}
     |    E '*' E       {$$ = $1*$3;}
     |    E '/' E       {$$ = $1/$3;}
     |    T_NUM         {$$ = $1;}
     |    '(' E ')'     {$$ = $2;}
     ;
```

Contents

语法制导定义

语法制导定义

- 语法制导定义（*Syntax-Directed Definition, SDD*）
  - 是一个上下文无关文法和属性及规则的结合。
  - 属性和文法符号相关联，规则和产生式相关联
- 属性的表示和实现
  - 如果X是一个符号而a是X的一个属性，那么我们用$X.a$来表示a在某个标号为X的分析树结点上的值
  - 属性可以有多种形式
    - 数字、类型、表格引用、串（代码序列）

继承属性和综合属性

- 非终结符号的两种属性
  - 综合属性（*synthesized attribute*）
    - 结点N上的综合属性只能通过N的子结点或N本身的属性值定义
  - 继承属性（*inherited attribute*）
    - 结点N上的继承属性只能通过N的父结点、N本身和N的兄弟结点上的属性值来定义

继承属性和综合属性

- 注意
  - 终结符号可以具有综合属性，但是不能有继承属性
    - 终结符号的综合属性值是由词法分析器提供的词法值
    - 在SDD中没有计算终结符号的属性值的语义规则

例……语法制导定义

❖ 例5.1 一个简单的桌面计算器的语法制导定义

	PRODUCTION	SEMANTIC RULES
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属性的SDD

- S属性（*S-attribute*）的SDD
  - 一个只包含综合属性的SDD称为S属性的SDD
  - 在一个S属性的SDD中，每个规则都根据相应产生式的右部中的属性值来计算产生式左部非终结符的一个属性
- 属性文法
  - 没有副作用的SDD也称为属性文法（*attribute grammar*）
  - 一个属性文法的规则仅仅通过其他属性值和常量值来定义一个属性值

在语法分析树的结点上对SDD求值

- 属性值的计算（语义处理）
  - 可以先构造一棵语法分析树，然后使用SDD的规则对这棵语法分析树的各个结点上的所有属性求值
    - 实际上编译器不需要显式构造语法分析树
    - 但实际上这个过程对应了一棵树的生长过程，而且在生长过程中每个节点的值将会被计算
- 注释语法分析树（*annotated parse tree*）
  - 一个显示了各个属性的值的语法分析树称为注释语法分析树
- 问题
  - 如何构造注释语法分析树？
  - 按照什么顺序计算各个属性？

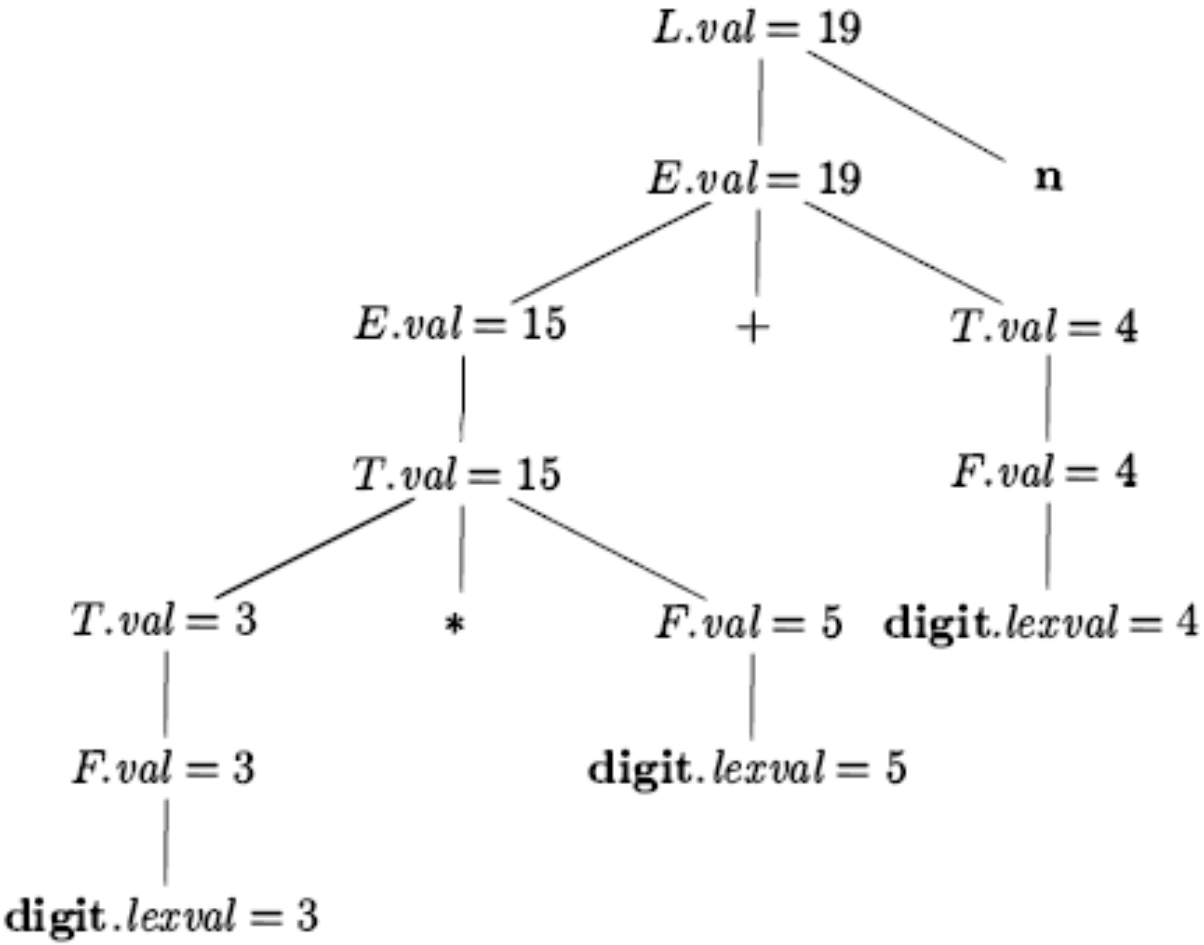
在语法分析树的结点上对SDD求值

- 属性的计算顺序
  - 在对一棵语法分析树的某个结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
- 综合属性的计算
  - 对于综合属性，可以按照任何自底向上的顺序计算它们的值
    - 比如对语法分析树进行后序遍历
- 同时有继承属性和综合属性的SDD
  - 不能保证有一个顺序对各个结点上的属性求值

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$ $B.i = A.s + 1$

例……注释语法分析树

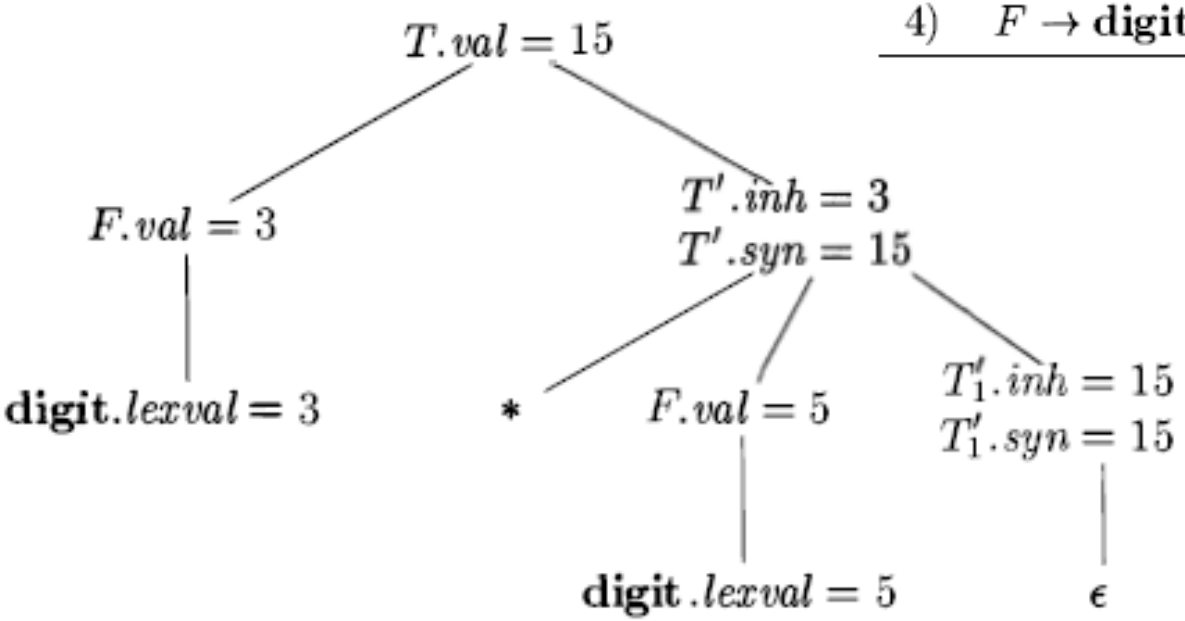
- 例5.2 对应输入串3*5+4n的注释语法分析树



例……继承属性的使用

- 例5.3
  - 一个基于适用于自顶向下的语法分析的文法的SDD
  - 输入串3*5的注释语法分析树

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



练习5.1（课堂练习）

练习 5.1.1：对于图 5-1 中的 SDD，给出下列表达式对应的注释语法分析树：

1）(3 + 4) * (5 + 6)n

2）1 * 2 * 3 * (4 + 5)n

3）(9 + 8 * (7 + 6) + 5) * 4n

练习 5.1.2：扩展图 5-4 中的 SDD，使它可以像图 5-1 所示的那样处理表达式。

练习 5.1.3：使用你在练习 5.1.2 中得到的 SDD，重复练习 5.1.1。

SDD的求值顺序

依赖图

- 属性的依赖关系
  - 已知一个SDD，对产生式$A \rightarrow \alpha$，有一组与其相关联的语义规则，规则的形式为：$b = f(c_1, c_2, \ldots, c_k)$，其中，$f$为一个函数：
    ① b是A的综合属性，且$c_1, c_2, \ldots, c_k$是产生式右部文法符号的属性，或者
    ② b是产生式右部某个文法符号的继承属性，且$c_1, c_2, \ldots, c_k$是A或者产生式右部任何文法符号的属性

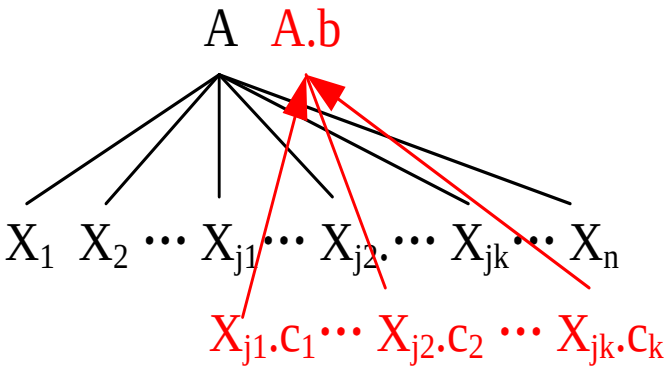
    在两种情况下，均称属性b依赖于$c_1, c_2, \ldots, c_k$
- 由语义规则所建立的属性依赖关系可使用依赖图表示
  - 依赖图中的边表示语义规则所蕴涵的约束
  - 对于每个语法分析树的结点X，和X关联的每个属性都在依赖图中有一个结点
  - 依赖图决定了语义规则的计算顺序

例……属性的依赖关系

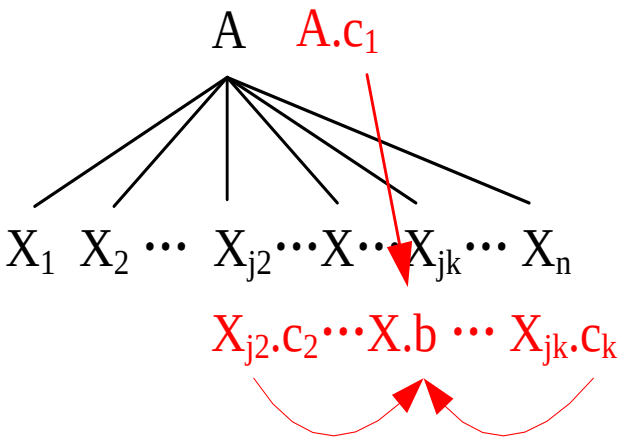
❖ 例：属性的依赖关系

1. A的综合属性 b 依赖于$c_1, c_2,\ldots,c_k$
2. X的继承属性 b 依赖于$c_1, c_2,\ldots,c_k$

1. $A \to \alpha$ $A.b = f(c_1, c_2,\cdots,c_k)$



2. $A \to \alpha$ $X.b = f(c_1, c_2,\cdots,c_k)$



假定 $\alpha = X_1X_2...X_n$

例……依赖图

❖ 例5.4

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

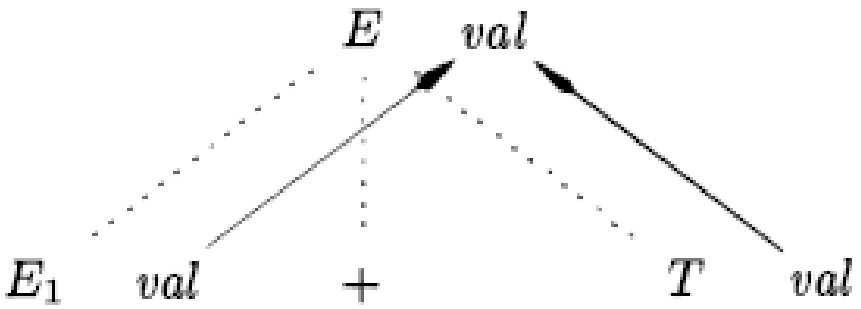


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

例……依赖图

❖ 例5.5 一个完整的依赖图

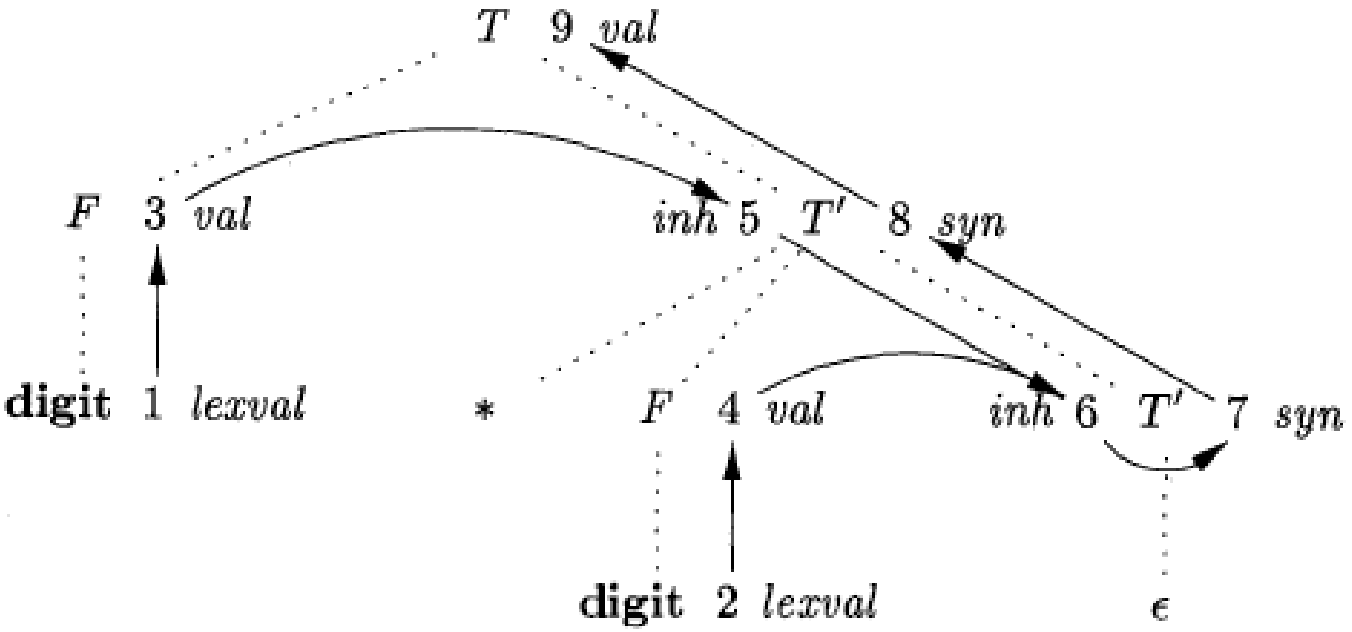


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

属性求值的顺序

- 有向无圈图的拓扑序是结点的一个任意顺序$m_1, m_2,\ldots,m_k$
  - 边都是从排序在前的结点指向在后的结点，即，

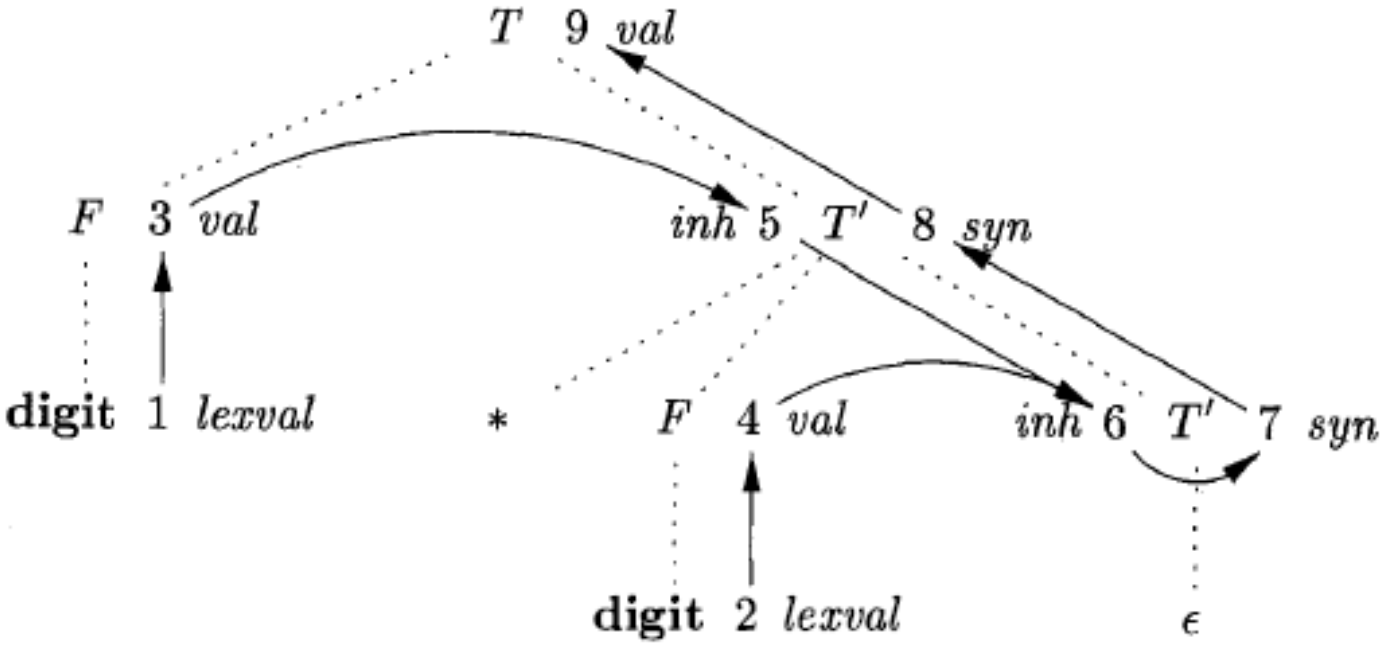
    若 $m_i \rightarrow m_j$ 是边，则在排序中 m_i 出现在 m_j 之前
- 依赖图的任意拓扑序均给出一个语义规则可被计算的有效顺序
  - 如果依赖图中有一条从结点M到结点N的边，那么先对M对应的属性求值，再对N对应的属性求值
  - 所有的可行求值顺序就是满足下列条件的结点序列 $N_1,N_2,\ldots,N_k$：如果有一条从结点N_i到N_j的依赖图的边，那么$i<j$。

属性求值的顺序

- 依赖图刻画了对一棵语法分析树中各结点上的属性求值时可能采取的顺序
  - 如果依赖图中没有环，那么总是至少存在一个拓扑排序
  - 如果存在循环依赖，则依赖图中有环，不存在拓扑排序
- 这表明可以精确（机械）地构造由属性文法所指定的翻译
  - 用基本文法构造输入串的语法树 → 构造依赖图 → 从依赖图的拓扑序中，可得到一个属性计算次序 → 按次序计算语义规则，产生输入串的一个翻译

例……依赖图的拓扑序

- 例5.6，
  - 图5-7中的依赖图没有环
  - 拓扑序①：1，2，3，……8，9
  - 拓扑序②：1，3，5，2，4，6，7，8，9



S属性的定义

- S属性定义
  - 如果一个SDD的每个属性都是综合属性，它就是S属性的。
- 例5.7
  - 图5-1中的SDD是一个S属性定义

	PRODUCTION	SEMANTIC RULES
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属性的定义

- S属性定义的属性求值顺序
  - 如果一个SDD是S属性的，那么可以按照语法分析树结点的任何自底向上顺序计算它的各个属性值。
  - 可以对语法分析树后序遍历并对属性求值，当遍历最后一次离开某个结点N时计算出N的各个属性值。
- S属性的定义可以在自底向上语法分析过程中实现
  - 一个自底向上的语法分析过程对应于一次后序遍历
  - 对于LR分析器，后序顺序精确对应于将一个产生式右部归约为左部的过程

L属性的定义

L属性定义

- SDD中的每个属性或者是

1. 综合属性，或者是
2. 继承属性，但它的规则具有如下限制：
   - 假设存在产生式$A \rightarrow X_1X_2...Xn$，并且有通过该产生式所关联的规则计算得到的继承属性$X_i.a$；那么这个规则只能使用：

① 和产生式左部A关联的继承属性；

② 位于X_i左边的文法符号实例$X_1, X_2, ... X_{i-1}$相关的继承属性或综合属性

③ 和这个X_i的实例本身相关的继承属性或综合属性，但是在由这个X_i的全部属性组成的依赖图中不存在环

例……L属性的定义

- 例5.8：图5-4中的SDD是L属性的

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

- 例5.9：包含下列产生式和规则的SDD不是L属性的

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

有受控副作用的语义规则

- 实践中翻译有副作用
  - 桌面计算器打印出结果
  - 代码生成器将标识符的类型加入符号表

例……语义规则的副作用

- 打印计算结果的桌面计算器
  - 语义规则*print(E.val)*的目的就是执行副作用
  - 可以将它看作是产生式左部符号L相关的哑综合属性的定义
  - 这个SDD在任何拓扑序下都产生相同的值

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$print(E.val)$
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

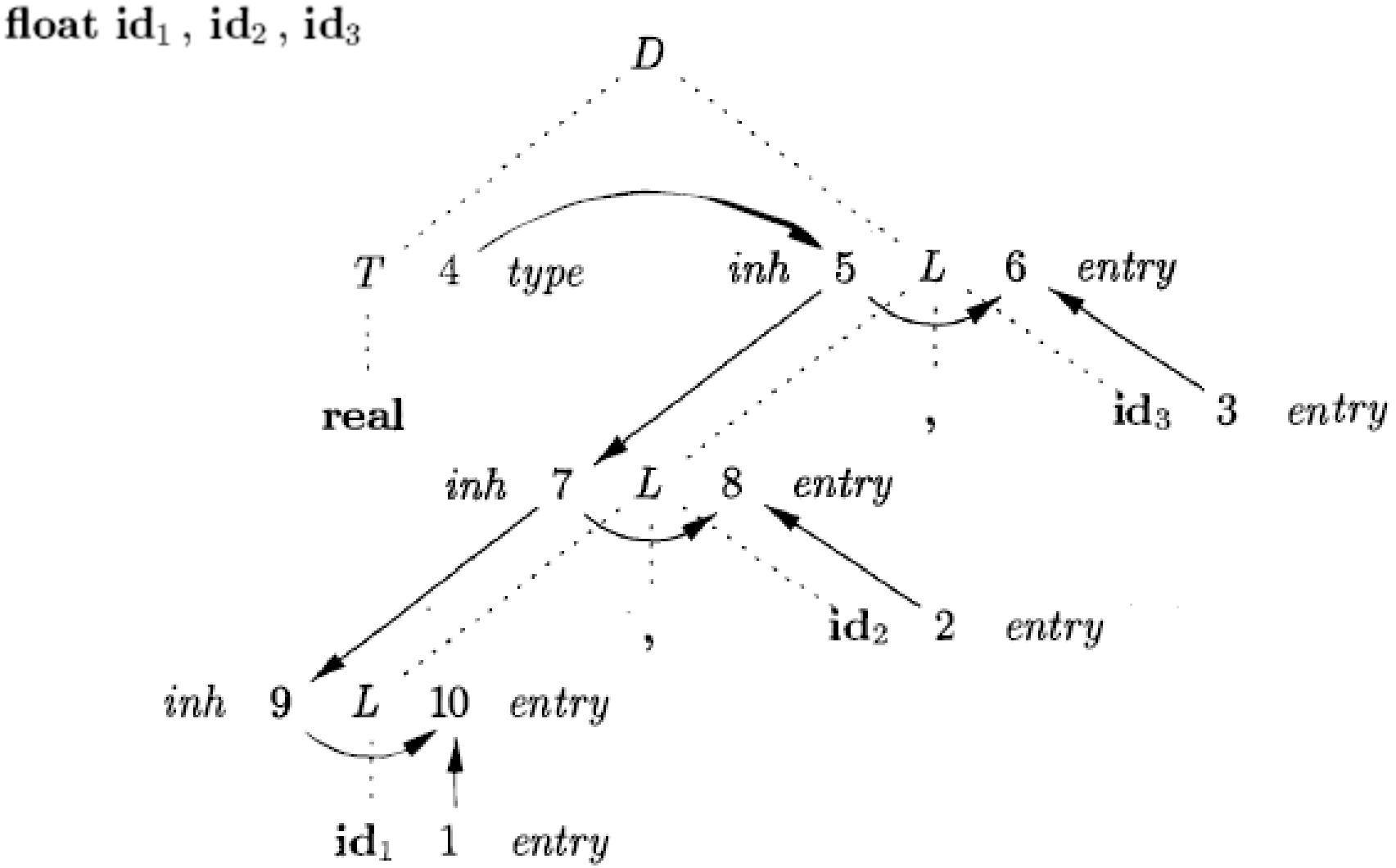
例……语义规则的副作用

- 例5.10：简单类型声明
  - 将声明标识符的类型信息加入符号表条目中

addType(id.entry, L.inh)

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

例……语义规则的副作用


float id1 , id2 , id3
D
T
4
type
inh
5
L
6
entry
real
,
id3
3
entry
inh
7
L
8
entry
,
id2
2
entry
inh
9
L
10
entry
id1
1
entry

例……设计SDD

- 为文法：

  S → aBS | bAS |ε

  B →aBB | b

  A→bAA | a

  写一个语法制导定义，统计句子中a的个数和b的个数

- 分析：
  - 需要两个综合属性分别记录a的个数和b的个数

产生式	语义规则
$S \rightarrow aBS_1$	S.an = 1 + B.an + S_1.an S.bn = B.bn + S_1.bn
$S \rightarrow bAS_1$	S.an = A.an + S_1.an S.bn = 1 + A.bn + S_1.bn
$S \rightarrow \varepsilon$	S.an = 0 S.bn = 0
$B \rightarrow aB_1B_2$	B.an = 1 + B_1.an + B_2.an B.bn = B_1.bn + B_2.bn
$B \rightarrow b$	B.an = 0 B.bn = 1
$A \rightarrow bA_1A_2$	A.an = A_1.an + A_2.an A.bn = 1 + A_1.bn + A_2.bn
$A \rightarrow a$	A.an = 1 A.bn = 0

例……设计SDD(课后练习)

- 为文法：

$S \rightarrow (L) \mid a \quad L \rightarrow L , S \mid S$

写一个语法制导定义，打印出句子中括号的对数

产生式	语义规则
$S' \rightarrow S$	print(S.h)
$S \rightarrow (L)$	S.h = L.h + 1
$S \rightarrow a$	S.h = 0
$L \rightarrow L_1 , S$	$L.h = L_1.h + S.h$
$L \rightarrow S$	L.h = S.h

练习5.2

练习5.2.2：对于图5-8中的SDD，给出下列表达式对应的注释语法分析树：

1) `int a, b, c`

2) `float w, x, y, z`

练习5.2.3：假设我们有一个产生式 $A \rightarrow BCD$。A、B、C、D 这四个非终结符号都有两个属性：s 是一个综合属性，而 i 是一个继承属性

对于下面的每组规则，指出(i)这些规则是否满足S属性定义的要求。

(ii)这些规则是否满足L属性定义的要求。

(iii)是否存在和这些规则一致的求值过程？

1) $A.s = B.i + C.s$

2) $A.s = B.i + C.s$ 和 $D.i = A.i + B.s$

3) $A.s = B.s + D.s$

!4) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$ 和 $D.i = B.i + C.i$

练习5.2

! 练习5.2.4：这个文法生成了含“小数点”的二进制数：

$$S \rightarrow L \,.\, L \mid L$$
$$L \rightarrow L\ B \mid B$$
$$B \rightarrow 0 \mid 1$$

设计一个L属性的SDD来计算 $S.val$，即输入串的十进制数值。

比如，串101.11应该被翻译为十进制数5.635。

提示：使用一个继承属性 $L.side$ 来指明一个二进制位在小数点的哪一边。

!! 练习5.2.5：为练习5.2.4中描述的文法和翻译设计一个S属性的SDD。

语法制导翻译的应用

- 语法制导翻译的应用
  - 类型检查
  - 中间代码生成
- 应用示例
  - 抽象语法树的构造：抽象语法树常用作编译器的中间表示形式。

抽象语法树的构造

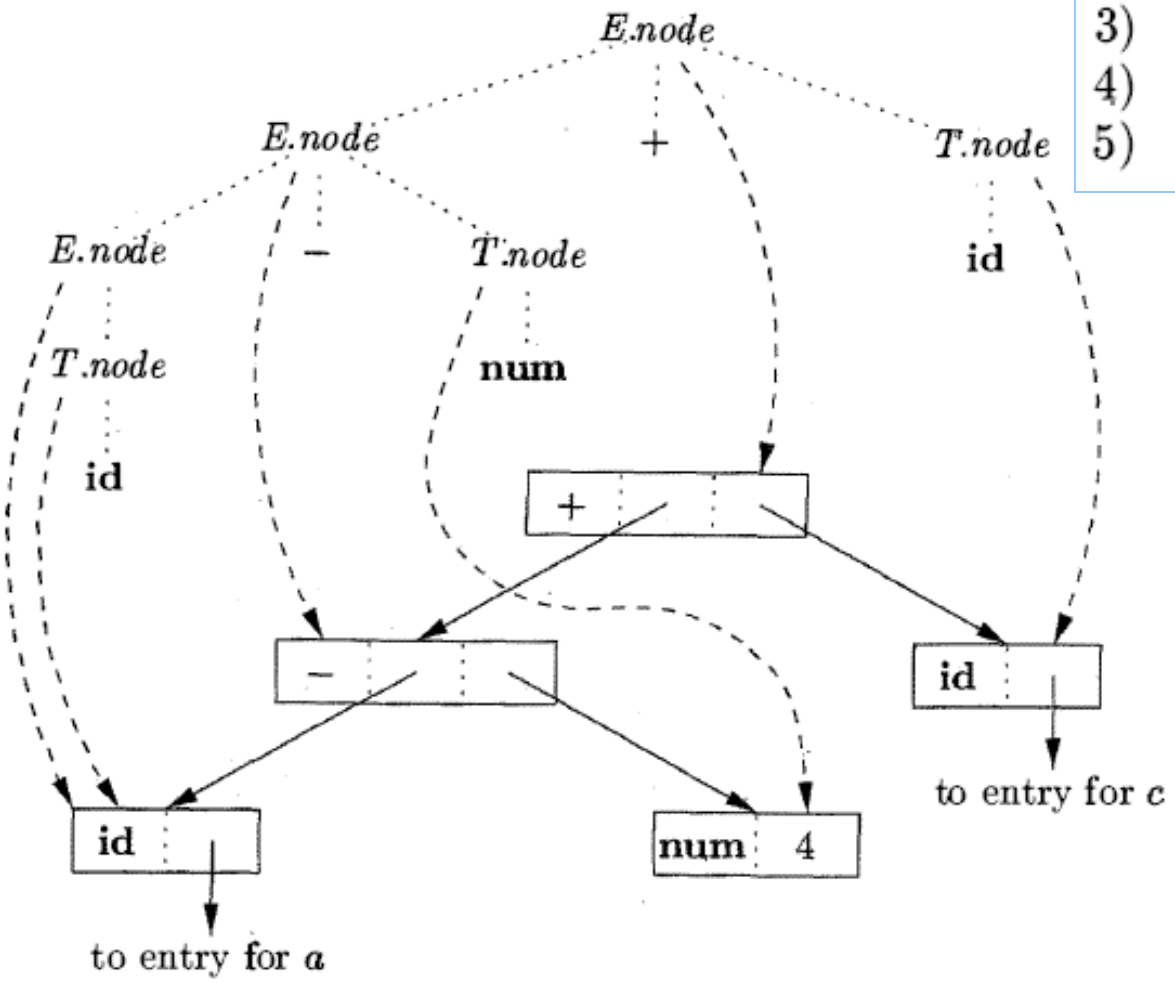
- S属性定义
  - *Leaf(op, val)*创建一个叶子结点，op是该结点的标号，val存放词法值
  - *Node(op, c_1, c_2, ..., c_k)*创建一个内部结点，标号为op
  - 非终结符的综合属性*node*表示相应的抽象语法树结点

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** $Leaf($**id**, **id**$.entry)$
6)	$T \to$ **num**	$T.node =$ **new** $Leaf($**num**, **num**$.val)$

例……构造抽象语法树

❖ *a-4+c*的抽象语法树和构造步骤

1) $p_1 = \mathbf{new}\ Leaf(\mathbf{id}, entry\text{-}a);$
2) $p_2 = \mathbf{new}\ Leaf(\mathbf{num}, 4);$
3) $p_3 = \mathbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \mathbf{new}\ Leaf(\mathbf{id}, entry\text{-}c);$
5) $p_5 = \mathbf{new}\ Node('+', p_3, p_4);$



语法制导的翻译方案

语法制导的翻译方案

- 语法制导的翻译方案（*syntax-directed translation scheme*）
  - 在产生式右部嵌入了程序片段的上下文无关文法
  - 这些程序片段称为语义动作，可以出现在产生式右部任何地方；语义动作用花括号括起，文法符号的花括号则加引号
  - SDT是对SDD的一种补充；SDD可以用SDT实现
- 本节讨论用SDT实现两类重要的SDD
  - 基本文法可以用LR技术分析，且SDD是S属性的
  - 基本文法可以用LL技术分析，且SDD是L属性的

语法制导的翻译方案

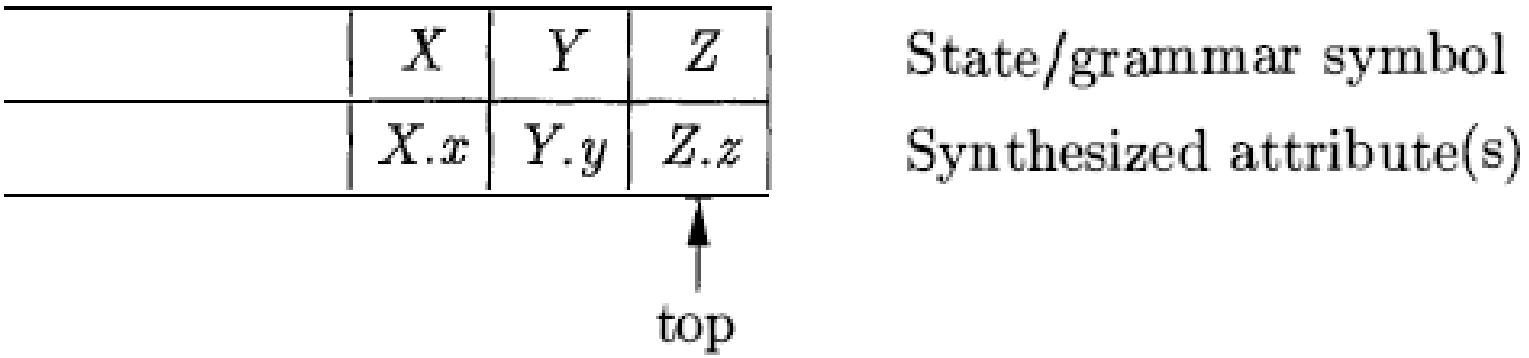
- 要解决的问题
  - 如何将SDD中的语义规则转换为带语义动作的SDT
  - 在语法分析过程中何时执行语义动作
    - 产生式右部的动作在它左边的所有文法符号都被匹配之后立刻执行
- 如何识别可以在语法分析过程中实现的SDT？
  - 将每个内嵌的语义动作替换为一个标记（*marker*）非终结符
  - 每个标记非终结符号M只有一个产生式，$M \rightarrow \varepsilon$
  - 如果带有标记非终结符号的文法可以使用某个方法进行语法分析，那么这个SDT就可以在语法分析过程中实现

后缀翻译方案

- 实现SDD最简单的情况
  - 文法用自底向上方法分析，SDD是S属性定义
  - 可以构造一个SDT，每个动作都在产生式的最后，将产生式右部归约为产生式左部的时候执行这个动作
- 后缀SDT
  - 所有动作都在产生式最右端的SDT称为后缀翻译方案
- 例5.14
  - 实现图5-1中的桌面计算器的SDD
  - 基本文法是LR的，SDD是S属性的

后缀SDT的语法分析栈实现

- 后缀SDT在LR语法分析的过程中实现
  - 归约时执行产生式相应的语义动作
  - 文法符号的属性和文法符号（或表示文法符号的LR状态）一起放在栈中的记录里，使得归约时可以找到它们
  - 所有属性都是综合属性，所有动作位于产生式的末尾，那么可以在把产生式右部归约为左部时计算各个属性的值



例……后缀SDT的语法分析栈实现

- 例5.15：在自底向上的语法分析栈中实现桌面计算器

PRODUCTION	ACTIONS
$L \to E$ **n**	{ print($stack[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (\ E\ )$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \to$ **digit**	

产生式内部带有语义动作的SDT

- SDT产生式内部的语义动作
  - 动作可以放在产生式右部任何位置
  - 当一个动作左边的所有符号都被处理过后，该动作立刻执行
  - 对产生式 $B \rightarrow X\{a\}Y$，当识别到X（X是终结符）或者所有从X推导出的终结符号（X是非终结符）之后，执行动作a，更准确地讲：
    ① 如果是自底向上的语法分析，在X的此次出现位于语法分析栈栈顶时，立刻执行语义动作a
    ② 如果是自顶向下的语法分析，在试图展开Y的本次出现（Y是非终结符）或者在输入中检测到Y（Y是终结符）之前执行语义动作a

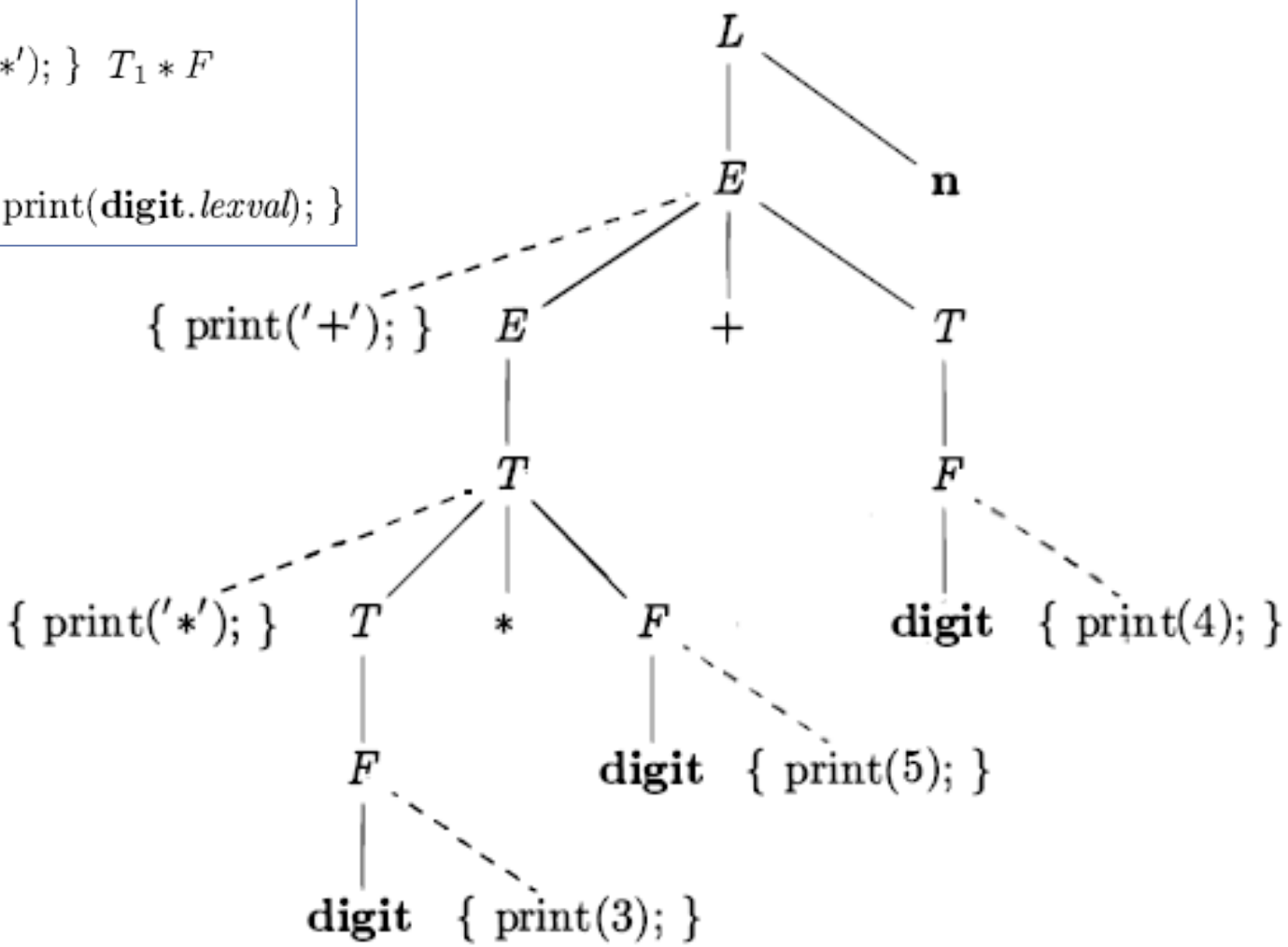
任意SDT的实现方法

- 任何SDT都可以按照下列方法实现
  1. 忽略语义动作，对输入进行语法分析，产生语法分析树
  2. 检查每个内部结点N，假设它的产生式是A→α；将α中的各个动作当作N的附加子结点加入，使得N的子结点从左到右和α中的动作完全一致
  3. 对这棵语法分析树进行前序遍历，并且当访问到一个以某动作为标号的结点时立刻执行这个动作

例……任意SDT的实现

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

$3 * 5 + 4$



从SDT中消除左递归

- 只考虑动作执行顺序的情况
  - 消除文法左递归时，将动作当成终结符号处理
  - 因为文法转换保持了由文法生成的符号串中终结符的顺序，因此，这些动作在任何从左到右的语法分析过程中都按照相同的顺序执行，无论分析是自顶向下还是自底向上的。

- 例5.17

$$A \to A\alpha \mid \beta$$

$$A \to \beta R$$
$$R \to \alpha R \mid \epsilon$$

$$\begin{array}{lcl} E & \to & E_1 + T \quad \{ \text{print}('+'); \} \\ E & \to & T \end{array}$$

$$\begin{array}{lcl} E & \to & T\ R \\ R & \to & +\ T\ \{ \text{print}('+'); \}\ R \\ R & \to & \epsilon \end{array}$$

实现L属性的SDD

实现L属性的SDD

- 遍历语法分析树（多趟）
  1. 建立语法分析树并注释。
     - 适用于任何非循环定义的SDD
  2. 构造语法分析树，加入动作，按前序顺序执行这些动作
     - 可以处理任何L属性定义
- 在语法分析过程中进行翻译（单趟）
  3. 使用递归下降语法分析器，为每个非终结符A建立一个函数，以参数的方式接收A的继承属性并返回A的综合属性
  4. 使用递归下降语法分析器，边扫描边生成代码
  5. 与LL语法分析器结合，实现SDT
  6. 与LR分析器结合，实现SDT

在递归下降语法分析过程中进行翻译

- 将递归下降语法分析器扩展为翻译器
  1. 函数A的参数是非终结符A的继承属性
  2. 函数A的返回值是非终结符A的综合属性的集合
  3. 在A的函数体中，进行语法分析并处理属性：
     - ① 决定用哪一个产生式展开A
     - ② 需要读入终结符号时，在输入中检查这些符号是否出现
     - ③ 在局部变量中保存所有必要的属性值，这些值将用于计算产生式右部非终结符的继承属性，或产生式左部非终结符的综合属性
     - ④ 调用对应于被选定产生式右部的非终结符的函数，向它们提供正确的参数

例……递归下降翻译器

- 例5.20 *while*语句的递归下降翻译器
- *while*语句的SDD和SDT

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

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

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

```
string S(label next) {
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                check ')' is next on the input, and advance;
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* other statement types */
}
```

L属性的SDD和LL语法分析

- 关于SDD的假设
  - SDD的基础文法是LL(1)文法，已经转换得到了SDT，语义动作嵌入到了各产生式中
  - 要在LL语法分析过程中完成翻译
- LL语法分析器的扩展
  - 扩展语法分析栈，以存放语义动作和属性求值所需的某些数据项；这些数据项一般是属性值的复制
  - 语法分析栈中保存

① 文法符号记录：终结符和非终结符

② 动作记录：表示即将被执行的语义动作

③ 综合记录：保存非终结符号的综合属性值

L属性的SDD和LL语法分析

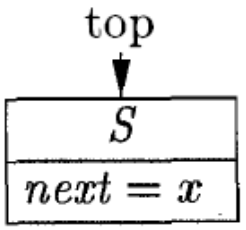
- 分析栈中的属性管理
  - 非终结符A的继承属性放在表示这个非终结符号的栈记录中
    - 对这些属性求值的代码通常用紧靠A的栈记录之上的动作记录表示
  - 非终结符A的综合属性放在一个单独的综合记录中，在栈中紧靠在A的记录之下
  - 所有属性复制都发生在对某个非终结符的一次展开时创建的不同记录之间；因此记录中的每一个都知道其他各个记录在栈中距离自己有多远，可以安全地把值写入它下面的记录中

例……L属性的SDD的LL语法分析

- 例5.23：实现图5.32 的SDT
  - 边扫描边生产while语句代码
  - 除了表示标号的哑属性之外，没有综合属性

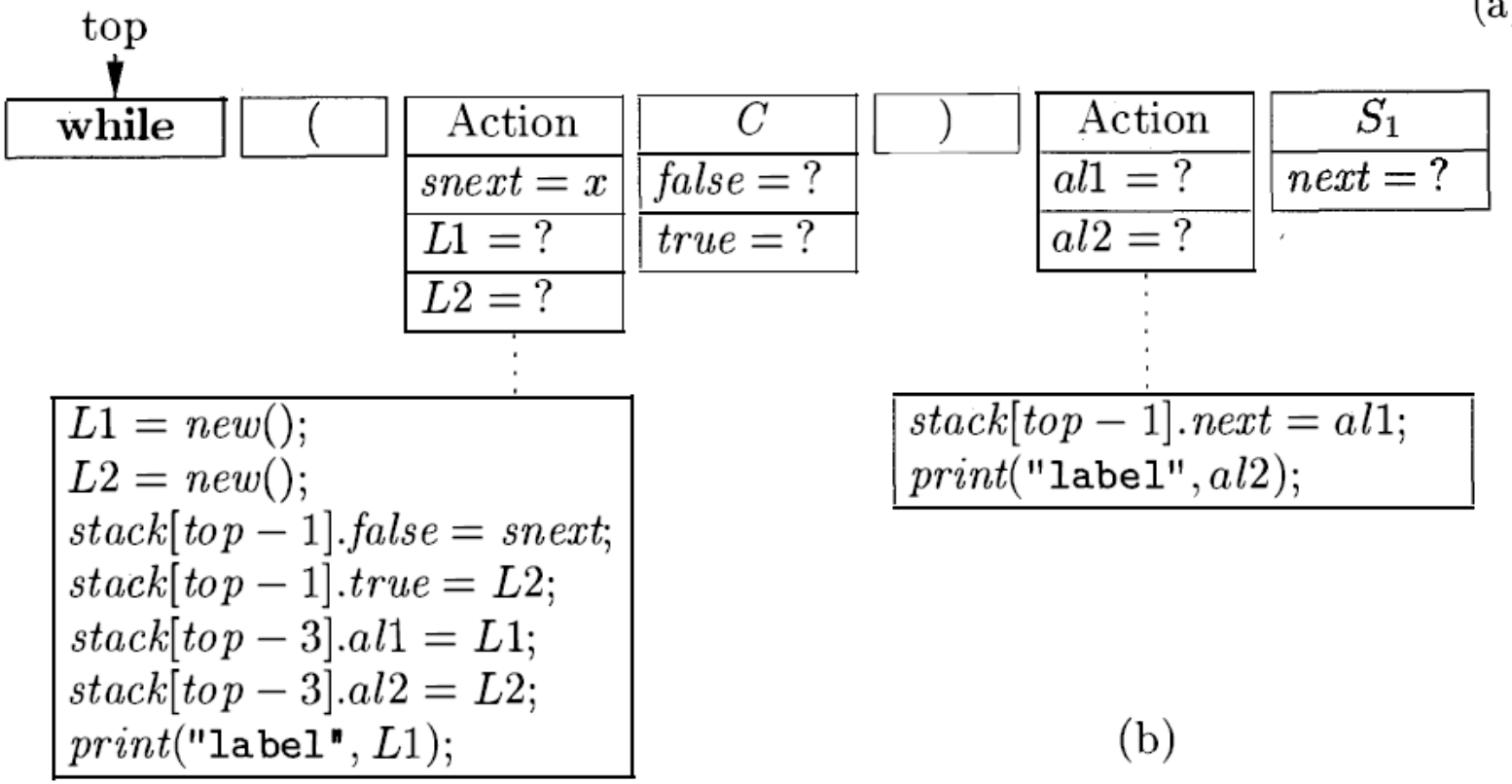
$$
\begin{array}{lcll}
S & \rightarrow & \textbf{while} \; (& \{ \; L1 = new(); \; L2 = new(); \; C.false = S.next; \\
 & & & \quad C.true = L2; \; print(\texttt{"label"}, L1); \; \} \\
 & & C \;) & \{ \; S_1.next = L1; \; print(\texttt{"label"}, L2); \; \} \\
 & & S_1 &
\end{array}
$$

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

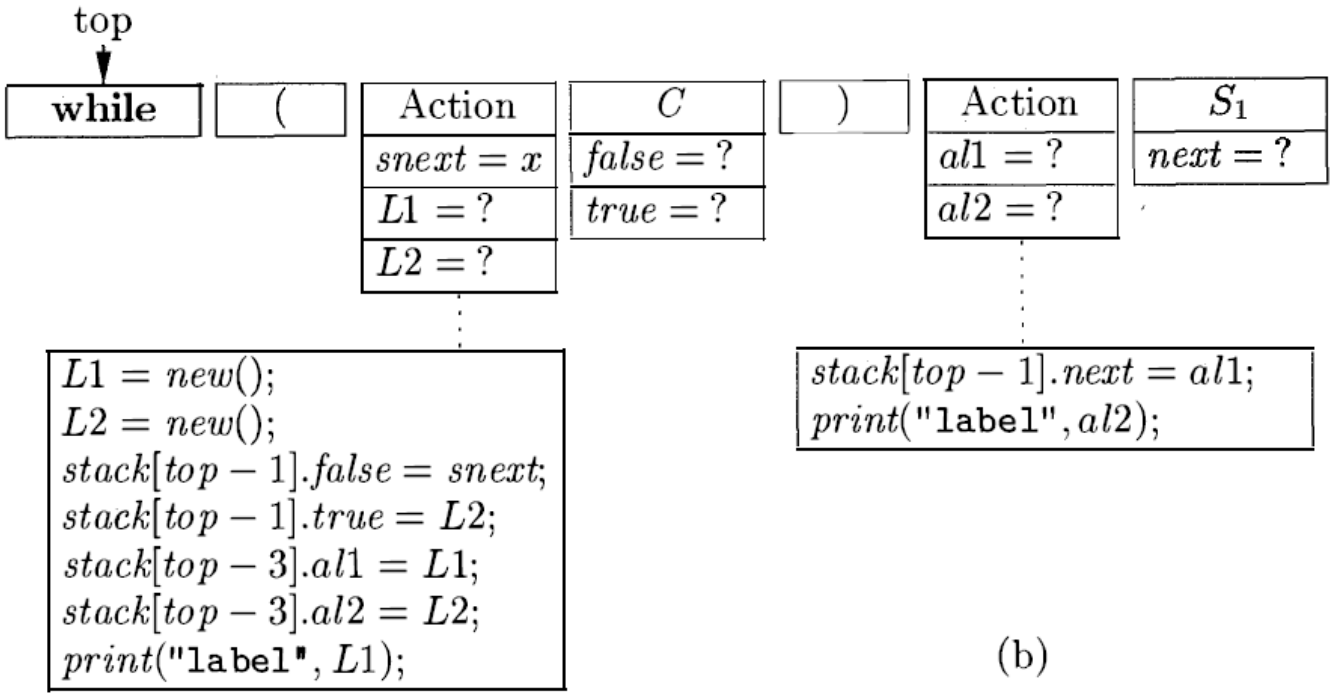


(a)

- 根据while语句的产生式扩展S

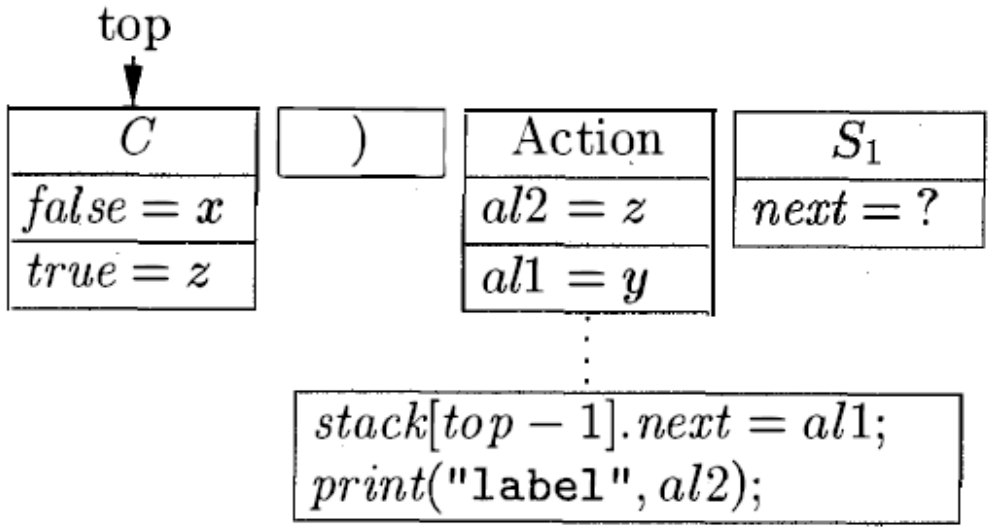


(b)



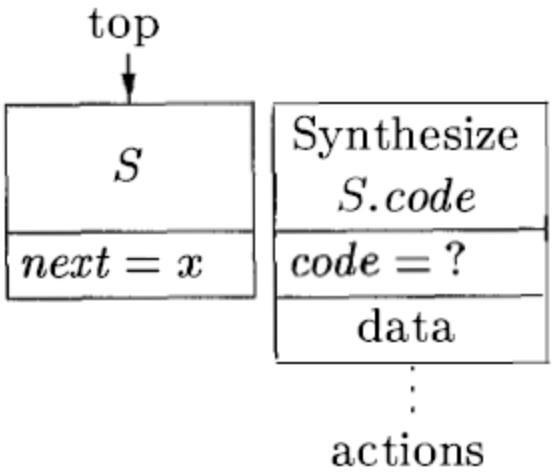
(b)

❖ C之上的动作执行之后



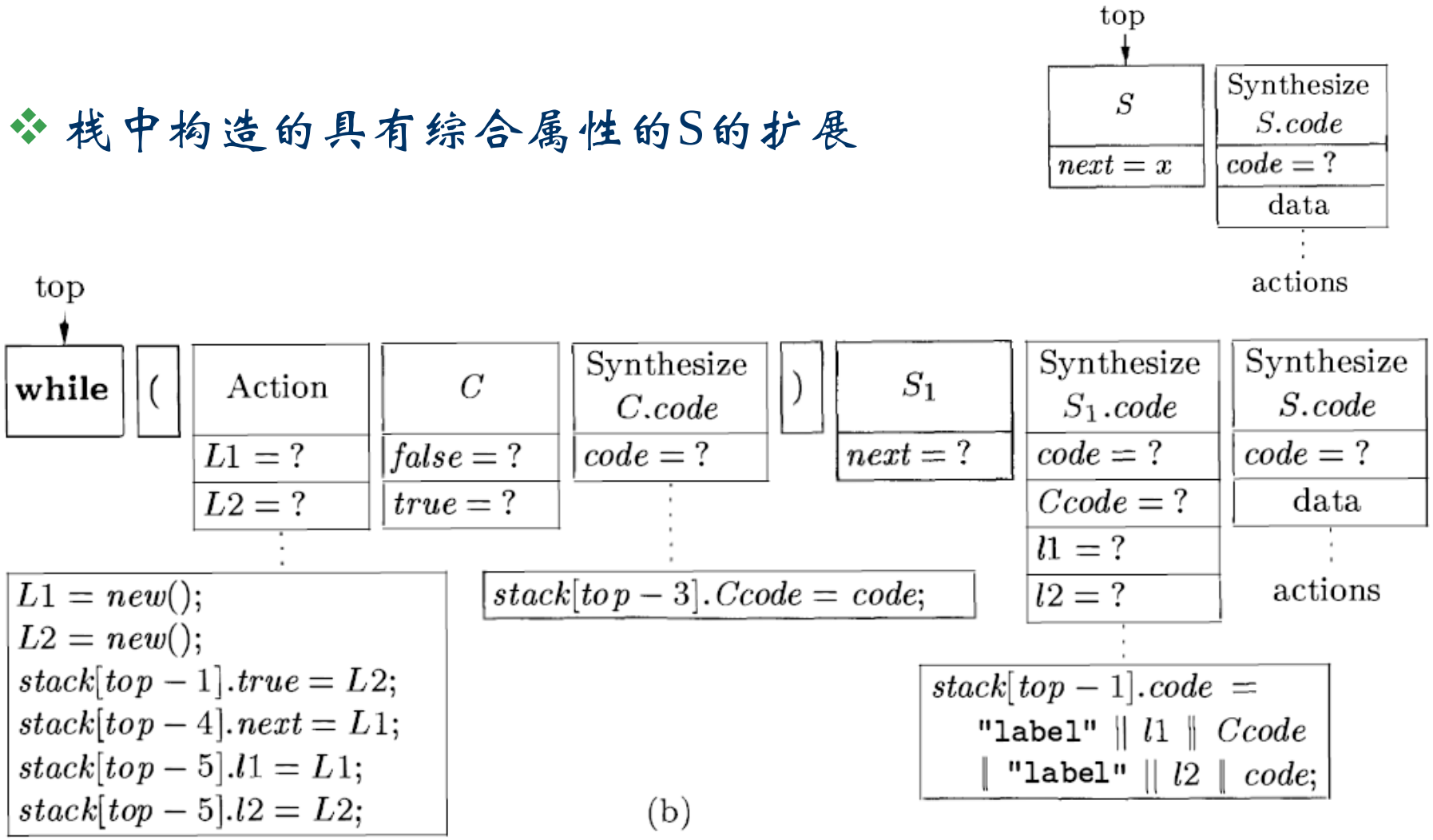
例……L属性的SDD的LL语法分析

- 例5.24：将S.code作为综合属性的翻译方法，假设：
  - 每个具有代码的非终结符都把它的（字符串形式的）代码存放在栈中该符号的记录下方的综合记录中
  - 展开S之前
    - 栈顶是S的记录，有一个存放S.next的字段
    - 紧靠这个记录之下是S的本次出现的综合记录，有一个存放S.code的字段



$S \rightarrow$ **while** (C) S_1

{ $L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;$ }
{ $S_1.next = L1;$ }
{ $S.code =$ **label** $\|\ L1\ \|\ C.code\ \|$ **label** $\|\ L2\ \|\ S_1.code;$ }



❖ 栈中构造的具有综合属性的S的扩展

L属性的SDD的自底向上语法分析

- 给定一个基础文法为LL文法的L属性的SDD，我们可以修改该文法，并在LR语法分析过程中计算这个新文法之上的SDD
  1. 构造SDD的SDT：计算非终结符继承属性的语义动作在该非终结符号之前，在产生式末尾放置计算综合属性的动作
  2. 对每个内嵌的语义动作，向这个文法中引入一个标记非终结符替换它；每个这样的位置都有一个不同的标记，对每个标记M都有一个产生式$M \rightarrow \varepsilon$
  3. 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

L属性的SDD的自底向上语法分析

- 产生式A→α{a}β的语义动作a到M→ε 的语义动作a'的变换：语义动作a'
  1. 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
  2. 按照a中的方法计算各个属性，将计算得到的这些属性作为M的综合属性
  - 因为是在LR分析栈上实现各个语义动作，所以必要的属性总是可用的，它们位于栈顶之下的已知位置上

例……L属性的SDD的LR分析

- 例5.25
  - 假设LL文法中一个产生式A→BC，继承属性B.i是根据继承属性A.i利用公式*B.i=f(A.i)*计算得到的
  - 相应的SDT片段是

$$A \rightarrow \{B.i = f(A.i);\}\ B\ C$$

  - 引入标记M，M有继承属性M.i和综合属性M.s
    - M.i是A.i的一个复制，M.s将称为B.i
  - SDT被写作

$$A \rightarrow M\ B\ C$$
$$M \rightarrow \{M.i = A.i;\ M.s = f(M.i);\}$$

例……L属性的SDD的LR分析

- 属性在LR分析栈中的安排
  - 可以安排LR分析栈，如果即将进行A的归约，那么A的每个继承属性都出现在栈中之下这个归约的位置下方，从该处就可以读到这些继承属性。
  - 将ε用$M \rightarrow \varepsilon$归约为M时，直接在它的下方找到$A.i$，并读取
  - $M.s$的值和M一起存放在栈中，实际上是$B.i$，以后在进行B的归约时可以在下方找到这个值

$$A \rightarrow M\ B\ C$$
$$M \rightarrow \{M.i = A.i;\ M.s = f(M.i);\}$$

例……L属性的SDD的LR分析

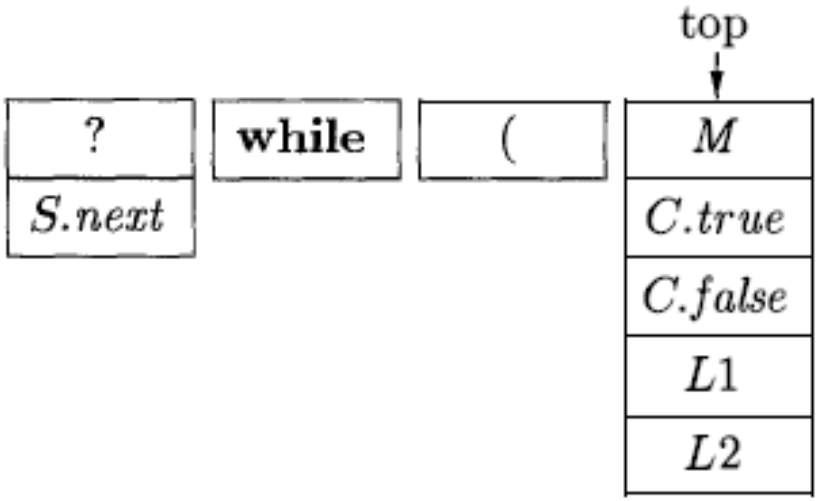
- 例5.26 *while*语句的LR分析和翻译
  - 原SDT

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

  - 修改后的LR文法

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{while} \; (\; M \; C \;) \; N \; S_1 \\
M & \rightarrow & \epsilon \\
N & \rightarrow & \epsilon
\end{array}
$$

例……L属性的SDD的LR分析



$$M \rightarrow \epsilon$$

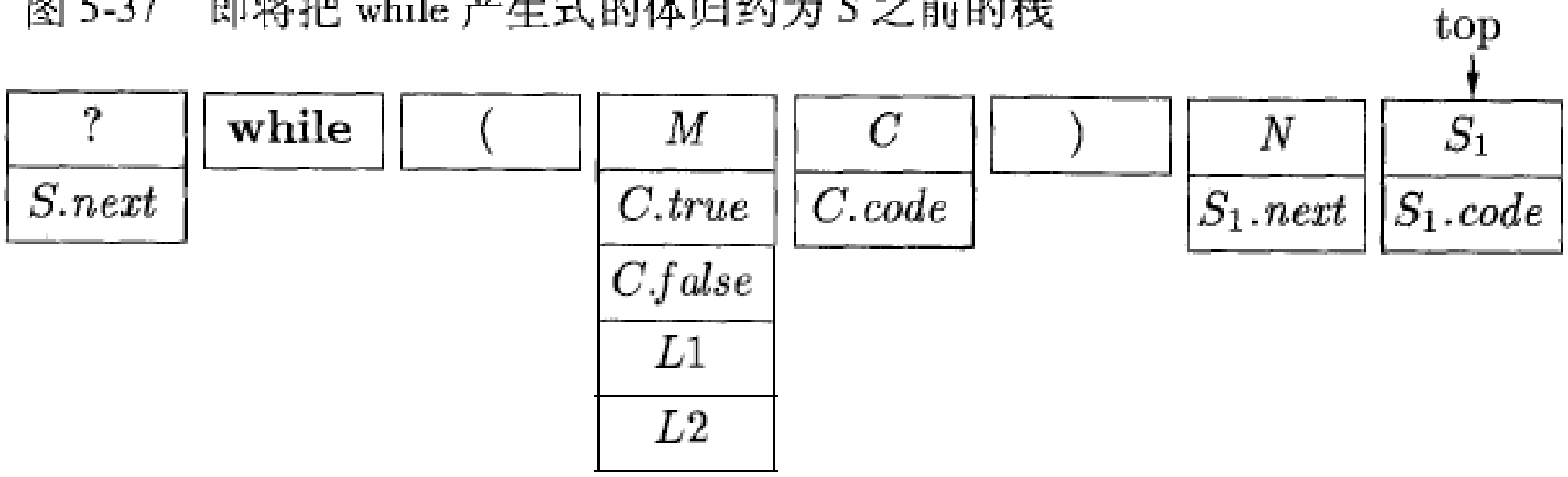
Code executed during reduction of ϵ to M

$L1 = new();$
$L2 = new();$
$C.true = L2;$
$C.false = stack[top - 3].next;$

图 5-36　在将 ϵ 归约为 M 之后的 LR 语法分析栈

例……L属性的SDD的LR分析

图 5-37　即将把 while 产生式的体归约为 S 之前的栈



$$N \rightarrow \epsilon$$

计算 $S_1.next$ 的值的代码是

$$S_1.next = stack[top - 3].L1;$$

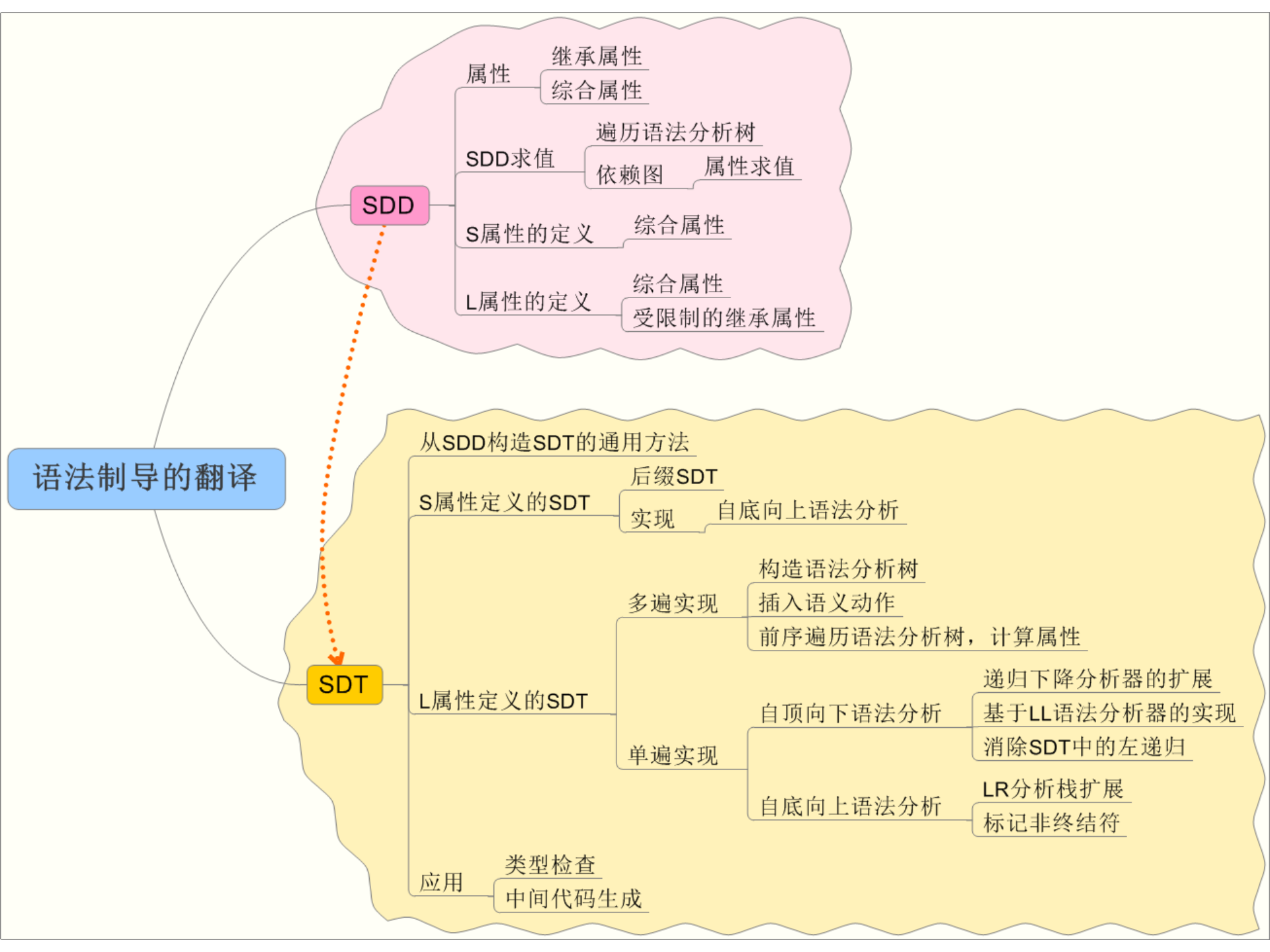
例……L属性的SDD的LR分析

$S \rightarrow \textbf{while}\ (\ M\ C\ )\ N\ S_1$

把从 **while** 到 S_1 的全部内容归约为 S 执行的代码是：

$$
\begin{aligned}
&tempCode = \textbf{label} \parallel stack[top-4].L1 \parallel stack[top-3].code \parallel \\
&\quad \textbf{label} \parallel stack[top-4].L2 \parallel stack[top].code; \\
&top = top - 5; \\
&stack[top].code = tempCode;
\end{aligned}
$$

本章小结


语法制导的翻译
SDD
属性
继承属性
综合属性
SDD求值
遍历语法分析树
依赖图
属性求值
S属性的定义
综合属性
L属性的定义
综合属性
受限制的继承属性
SDT
从SDD构造SDT的通用方法
S属性定义的SDT
后缀SDT
实现
自底向上语法分析
L属性定义的SDT
多遍实现
构造语法分析树
插入语义动作
前序遍历语法分析树，计算属性
单遍实现
自顶向下语法分析
递归下降分析器的扩展
基于LL语法分析器的实现
消除SDT中的左递归
自底向上语法分析
LR分析栈扩展
标记非终结符
应用
类型检查
中间代码生成