Lecture 10: 语法制导的翻译-II

---L-属性SDD的实现

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

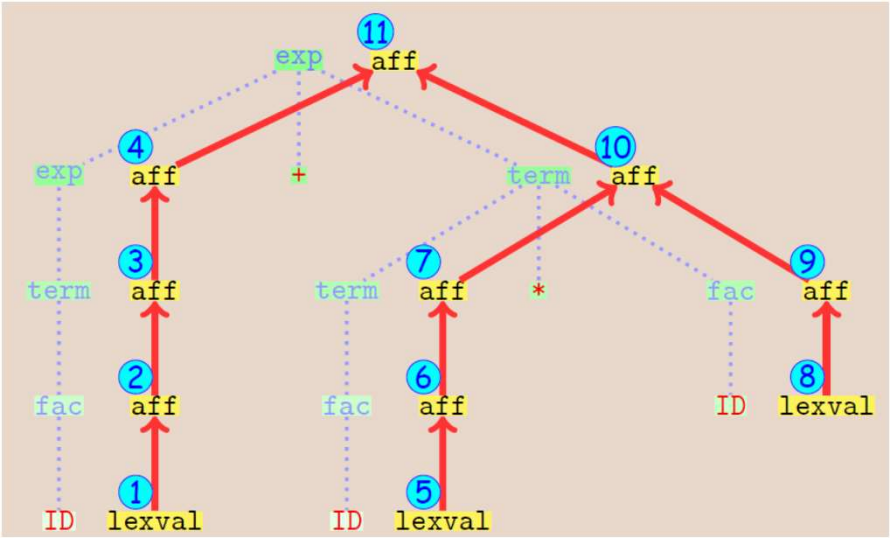
SDD的实现方法

- **先分析，后翻译---通用方法，适用于S属性SDD和L属性SDD**
  - 注释语法树（通用方法）
    - 建立语法分析树→添加注释成为注释语法分析树→画出依赖图→找出其中的拓扑顺序(无环)，按照拓扑顺序完成计算语法翻译
  - 添加语义动作虚拟结点（通用方法）
    - 建立语法分析树→将SDT语义动作作为虚拟结点添加至相应位置→前序遍历语法树，执行语义动作
    - 需要按照位置要求把SDD转换成SDT

SDD的实现方法

- **分析与翻译同步---非通用方法**
  - 如何能同步？
    - 属性的计算次序一定受分析方法所限定的分析树结点建立次序的限制
  - LR分析+S属性SDD
    - 从注释语法树来看

      计算顺序和LR分析天然相容
    - 后缀SDT+栈实现 (示例)



SDD的实现方法

- **分析与翻译同步---非通用方法**
  - 边分析边翻译的方式能否用于继承属性？
    - 分析树的结点是自左向右生成；如果属性信息是自左向右流动，那么就有可能在分析的同时完成属性计算
  - 递归下降分析+L属性SDD
  - LL分析 + L属性SDD
  - LR分析 + L属性SDD

10.1 递归下降过程中进行翻译

递归下降语法分析

- 例子

P:
(1) Z → aBd {a}
(2) B → d {d}
(3) B → c {c}
(4) B → bB {b}

a b c d

```
Z ( )
{
if (token == a)
 {   match(a);
     B( );
     match(d);
 }
else error();
}
```

```
B ( )
{
 case token of
  d:  match(d);break;
  c:   match(c); break;
  b:{  match(b);
       B( ); break;}
  other: error();
}
```

```
void main()
{read();
  Z();  }
```

递归下降语法分析

预测翻译器的设计:

把预测分析器的构造方法推广到翻译方案的实现

产生式$R \rightarrow +TR \mid \varepsilon$ 的分析过程

```
void R( ) {
if (lookahead == '+' ) {
        match ( '+' ); T( ); R( );
}
else  /* 什么也不做 */
}
```

L属性定义和递归下降语法分析天然相容

L属性定义的递归下降实现（基本方法）

- **使用递归下降的语法分析器**
  - 每个非终结符号对应一个函数
  - 函数的参数接受继承属性
  - 返回值包含了综合属性
- **在函数体中**
  - 首先选择适当的产生式
  - 使用局部变量来保存属性
  - 对于产生式体中的终结符号，读入符号并获取其（经词法分析得到的）综合属性
  - 对于非终结符号，使用适当的方式调用相应函数，并记录返回值

L属性定义的递归下降实现（基本方法）

- 例1

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

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

L属性定义的递归下降实现（基本方法）

- 例2

$B \rightarrow B_1 B_2 \mid B_1 \textbf{sub} B_2 \mid (\ B_1\ ) \mid \textbf{text}$

	产生式		语义动作
1)	$S \rightarrow$	B	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow$	B_1	$\{\ B_1.ps = B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$ $B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$	B_1 **sub**	$\{\ B_1.ps = B.ps;\ \}$
		B_2	$\{\ B_2.ps = 0.7 \times B.ps;\ \}$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow$	(	$\{\ B_1.ps = B.ps;\ \}$
		B_1)	$\{\ B.ht = B_1.ht;$ $B.dp = B_1.dp;\ \}$
5)	$B \rightarrow$	**text**	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$ $B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

$$
\begin{array}{lcl}
S & \rightarrow & B \\
B & \rightarrow & T\ B_1 \mid T \\
T & \rightarrow & F\ \textbf{sub}\ T_1 \mid F \\
F & \rightarrow & (\ B\ ) \mid \textbf{text}
\end{array}
$$

```
(float, float) T(float ps) {
    float h1, h2, d1, d2; /* 用于存放高度和深度的局部变量*/
    /* F(ps) 代码开始 */
    if (当前输入 == '(' ) {
        读取下一个输入;
        (h1, d1) = B(ps);
        if (当前输入 != ')' ) 语法错误：期待 ')';
        读取下一个输入;
    }
    else if ( 当前输入 == text ) {
        令 t 等于词法值 text.lexval ;
        读取下一个输入;
        h1 = getHt(ps, t);
        d1 = getDp(ps, t);
    }
    else 语法错误 : 期待 text 或者 '(';
    /* F(ps) 代码结束 */
    if ( 当前输入 == sub ) {
        读取下一个输入;
        (h2, d2) = T(0.7 * ps);
        return (max(h1, h2 − 0.25 * ps), max(d1, d2 + 0.25 * ps));
    }
    return (h1, d1);
}
```

L属性定义的递归下降实现（边扫描边计算属性）

- **当属性值的体积很大时，对属性值进行运算的效率很低**
  - 比如code（代码）可能是一个上百K的串，对其进行并置等运算会比较低效
- **可逐步生成属性的各个部分，并增量式添加到最终的属性值中**
- **条件：**
  - 存在一个主属性，且主属性是综合属性
  - 在各产生式中，主属性是通过产生式体中各个非终结符号的主属性连接（并置）得到的，同时还会连接一些其它的元素
  - 各非终结符号的主属性的连接顺序和它在产生式体中的顺序相同

L属性定义的递归下降实现（边扫描边计算属性）

- **基本思想**
  - 只需要在适当的时候“发出”非主属性的元素，即把这些元素拼接到适当的地方
- **例**

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

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

L属性定义的递归下降实现（边扫描边计算属性）

- **基本思想**
  - 只需要在适当的时候“发出”非主属性的元素，即把这些元素拼接到适当的地方
- **例**

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

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

L属性定义的递归下降实现（边扫描边计算属性）

- **例　左递归的消除引起继承属性**

产　生　式	语　义　规　则
$E \to E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \to T$	$E.nptr = T.nptr$
$T \to T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \to F$	$T.nptr = F.nptr$
$F \to (E)$	$F.nptr = E.nptr$
$F \to$ id	$F.nptr = mkLeaf$ (id, id.*entry*)
$F \to$ num	$F.nptr = mkLeaf$ (num, num.*val*)

L属性定义的递归下降实现

产 生 式	语 义 规 则
$E \to E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \to T$	$E.nptr = T.nptr$
$T \to T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \to F$	$T.nptr = F.nptr$
$F \to (E)$	$F.nptr = E.nptr$
$F \to$ id	$F.nptr = mkLeaf$ (id, id.*entry*)
$F \to$ num	$F.nptr = mkLeaf$ (num, num.*val*)

$E \to T$ $\{R.i = T.nptr\}$ $T + T + T + \ldots$

R $\{E.nptr = R.s\}$

$R \to +$

T $\{R_1.i = mkNode(\text{‘+’}, R.i, T.nptr)\}$

R_1 $\{R.s = R_1.s\}$

$R \to \varepsilon$ $\{R.s = R.i\}$

$T \to F$ $\{W.i = F.nptr\}$

W $\{T.nptr = W.s\}$

$W \to *$

F $\{W_1.i = mkNode(\text{‘*’}, W.i, F.nptr)\}$

W_1 $\{W.s = W_1.s\}$

$W \to \varepsilon$ $\{W.s = W.i\}$

F 产生式部分不再给出

L属性定义的自上而下计算

```
syntaxTreeNode* R (syntaxTreeNode* i) {
    syntaxTreeNode *nptr, *i1, *s1, *s;
    char addoplexeme;

    if (lookahead == '+' ) { /* 产生式 R → +T R */
          addoplexeme = lexval;
          match('+' ); nptr = T();
          i1 = mkNode(addoplexeme, i , nptr);
          s1 = R (i1); s = s1;
    }
    else  s = i;  /* 产生式 R → ε */
    return s;
}
```

$R : i, s$

$T : nptr$

$+ : addoplexeme$

10.2 L属性定义的LL分析

L属性定义的LL实现

- **难点在哪里？**
  - LL分析和递归下降异同：都是top-down，但递归下降有回溯，而LL是一直向下
  - LL语法分析器实现了一个最左推导的过程。假设推导S =>* ωα 中：
    - ω 是已经匹配完成的输入串，而α的开头为A，A将以A →BC 产生式展开 (查表得)。那么，当前栈顶为A，下面将弹出A替换为BC (B为栈顶)
    - 如果C有继承属性C.i依赖于A的继承属性和B的属性，对于L属性文法，当计算到C时，A的继承属性和B的所有属性都应该可用了(思考why?)
    - 但是当分析到C的时候，A和B都早已不在栈内，其相关联的属性值也消失，因此无法计算C.i

L属性定义的LL实现

- **实现技巧**
  - 将所需的A的继承属性值拷贝到对C的继承属性求值的动作记录中
  - 将所需的B的继承/综合属性都临时拷贝到栈中B之下的一个记录中(top-1)，留待C.i计算的时候使用
- **为什么这种临时拷贝是可行的？**
  - 所有拷贝都发生在对某个非终结符号的一次展开时创造的不同文法符号记录之间，因为产生式可知，这些文法符号的相对位置也可知，所以他们在栈中的位置(相距于栈顶距离)也可以，即，可以正确寻址数据项，部分数据项是其他项 (文法符号)属性值的临时拷贝

L属性定义的LL实现

扩展语法栈

- 基础文法本身要是LL的，然后把SDD改写为内嵌的SDT
  - 计算X的继承属性的动作插入到产生式体中对应的X的左边；计算产生式头的综合属性的动作在产生式的最右边
- 扩展语法分析栈，存放语义动作，以及需要存储所有属性求值所需的某些数据项，部分数据项是其他项 (文法符号)属性值的临时拷贝

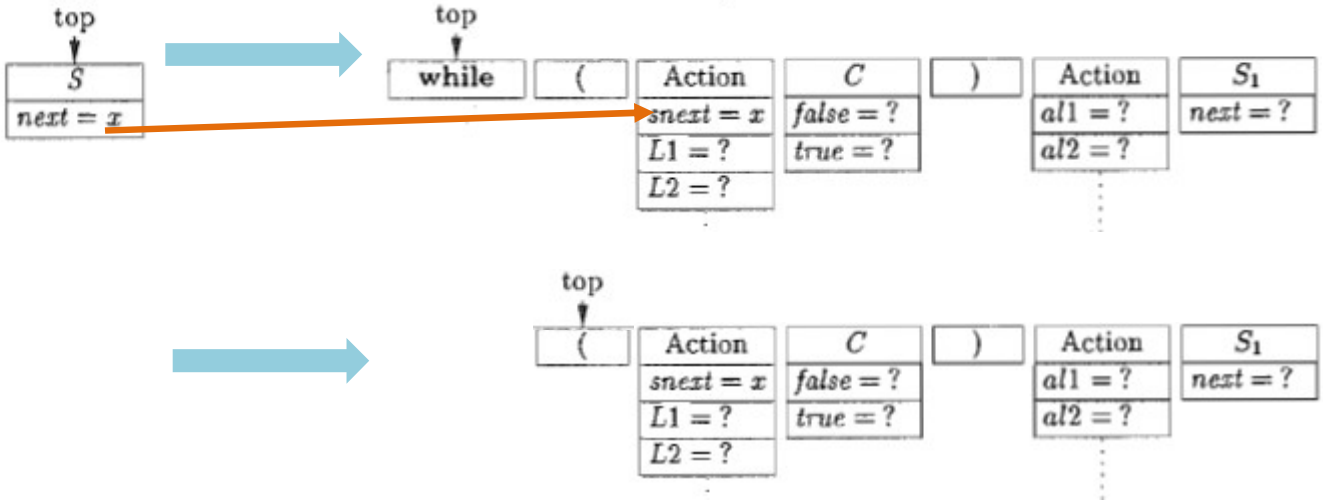
L属性定义的LL实现

- **扩展语法栈**
  - 扩展语法分析栈，除了保存对应于文法符号 (终结符和非终结符)的item之外，还增加以下两类Item:
    - 动作记录（action-record）：即将被执行的语义动作
    - 综合记录（synthesize-record）：非终结符的综合属性值
  - 这两类item在分析栈中的位置如下：
    - (非终结符X的继承属性位于它自身的记录item中)，而对这些属性求值的代码则使用仅靠在X的栈记录之上的动作记录项来表示
    - 非终结符X的综合属性放在单独的综合记录项中，在栈里面仅靠在X记录项之下
    - 与L属性文法内嵌式SDT的生成原则基本一致

L属性定义的LL实现

- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例1

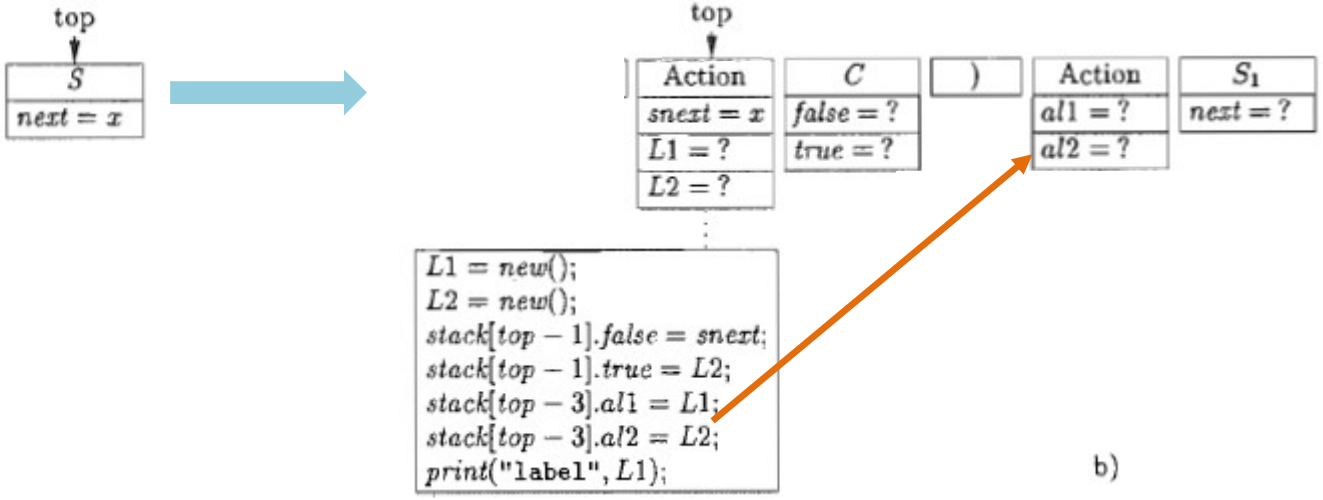
S	→	while (	{ $L1 = new()$; $L2 = new()$; $C.false = S.next$; $C.true = L2$; $print(\text{"label"}, L1)$; }
		C)	{ $S_1.next = L1$; $print(\text{"label"}, L2)$; }
		S_1	



L属性定义的LL实现

- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例1

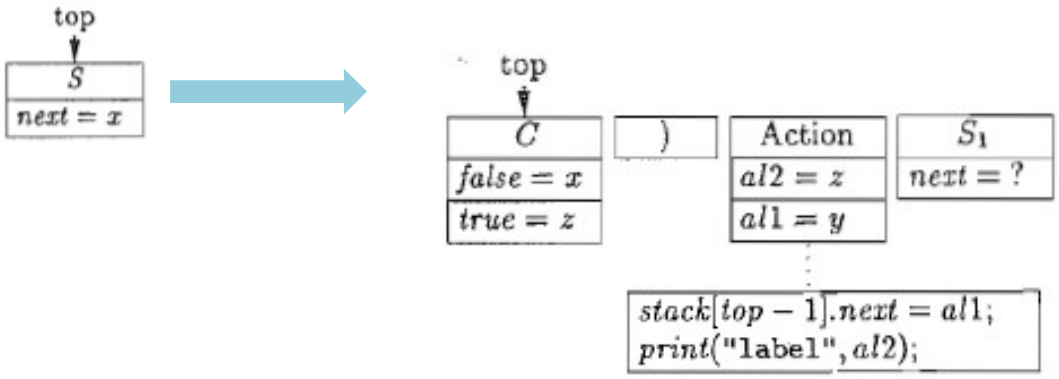
$S \rightarrow$ **while** (	{ $L1 = new()$; $L2 = new()$; $C.false = S.next$; $C.true = L2$; }
C)	{ $S_1.next = L1$; }
S_1	{ $S.code =$ **label** $\|\|$ $L1$ $\|\|$ $C.code$ $\|\|$ **label** $\|\|$ $L2$ $\|\|$ $S_1.code$; }



L属性定义的LL实现

- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例1

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

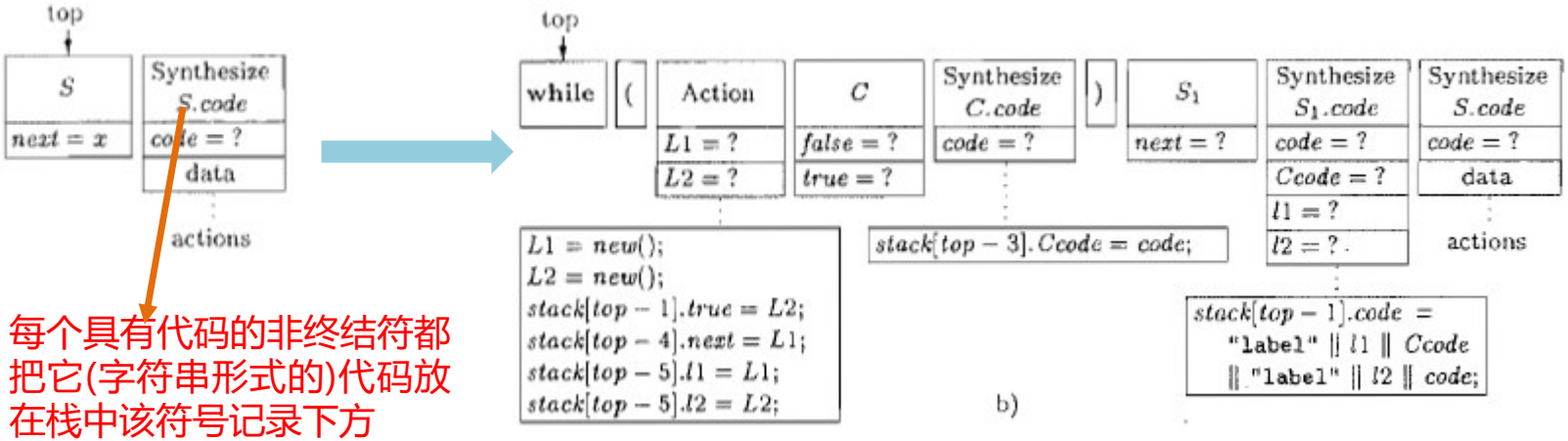


L属性定义的LL实现

- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例2

$S \rightarrow$ **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** $\|$ $L2$ $\|$ $S_1.code$; }



10.3 L属性定义的LR分析

L属性定义的LR实现

我们可以处理 LR 文法上的 L 属性 SDD 吗?

在5.4.1 节中,我们看到在 LR 文法上的每个 S 属性 SDD 都可以在自底向上语法分析过程中实现。根据 5.3.5 节,LL 文法上的每个 L 属性都可以在自顶向下语法分析中实现。因为 LL 文法类是 LR 文法类的一个真子集,并且 S 属性 SDD 类是 L 属性 SDD 类的一个真子集,那么我们能否以自底向上的方式处理每个 LR 文法和每个 L 属性 SDD 呢?

如下面的直观论述指出的,我们不能这么做。假设我们有一个 LR 文法的产生式 $A \rightarrow BC$,并且有一个继承属性 $B.i$,它依赖于 A 的继承属性。当我们规约到 B 的时候,我们还没有看到由 C 生成的输入,因此不能确定会扫描到产生式 $A \rightarrow BC$ 的体。因此,我们在此时还不能计算 $B.i$,因为我们不能确定是否使用和这个产生式相关联的规则。

也许我们可以等到已经归约得到 C,并且知道必须把 BC 归约到 A 时才进行计算。然而,即使到那个时候,我们仍然不知道 A 的继承属性,因为即使在归约之后,我们仍然不能确定包含这个 A 的是哪个产生式的体。我们可以说这个决定也应该推迟,因此也需要将 $B.i$ 的计算进一步推迟。如果我们继续这样推迟,我们很快会发现必须把所有的决定推迟到对整个输入的语法分析完成之后再进行。实质上,这就是"先构造语法分析树,再执行翻译"的策略。

L属性定义的LR实现

- 它能实现任何基于LL(1)文法的*L*属性定义
- 也能实现许多（但不是所有的）基于LR(1) 的*L*属性定义

删除翻译方案中嵌入的动作

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{print\ (\text{'+'})\}R_1 \mid -\ T\ \{print\ (\text{'–'})\}R_1 \mid \varepsilon$

$T \rightarrow \text{num}\ \{print\ (\text{num}.val)\}$

在文法中加入产生ε的标记非终结符，让每个嵌入动作由不同标记非终结符M代表，并把该动作放在产生式$M \rightarrow \varepsilon$的右端

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R_1 \mid -\ T\ N\ R_1 \mid \varepsilon$

$T \rightarrow \text{num}\ \{print\ (\text{num}.val)\}$

$M \rightarrow \varepsilon\ \{print\ (\text{'+'})\}$

$N \rightarrow \varepsilon\ \{print\ (\text{'–'})\}$

这些动作的一个重要特点：

没有引用原来产生式文法符号的属性

L属性定义的LR实现

- **Case 1: 分析栈上的继承属性---位置可预测**

例 int p, q, r

$D \rightarrow T$ {*L.in = T.type*}

L

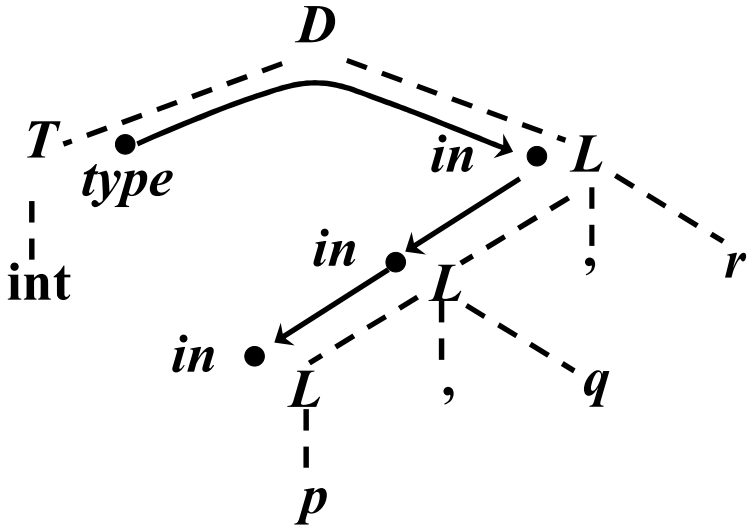
$T \rightarrow$ int {*T. type = integer*}

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

$L \rightarrow$ {L_1*.in = L.in* }

L_1, id {*addtype* (id.*entry*, *L.in*)}

$L \rightarrow$ id {*addtype* (id.*entry*, *L.in*)}



L属性定义的LR实现

- **Case 1: 分析栈上的继承属性---位置可预测**

例 int p, q, r

$D \rightarrow T$ {*L.in = T.type*}

L

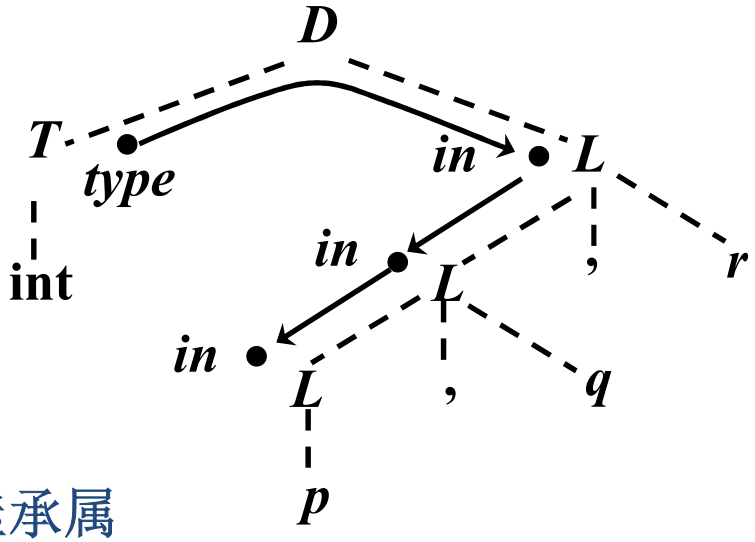
$T \rightarrow$ int {*T. type = integer*}

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

$L \rightarrow$ {L_1*.in = L.in* }

L_1, id {*addtype* (id.*entry*, *L.in*)}

$L \rightarrow$ id {*addtype* (id.*entry*, *L.in*)}



继承属性的计算可以略去，引用继承属性的地方改成引用其他符号的综合属性

L属性定义的LR实现

■ Case 1: 分析栈上的继承属性---位置可预测

例 int p, q, r

$D \rightarrow T$ {*L.in = T.type*}

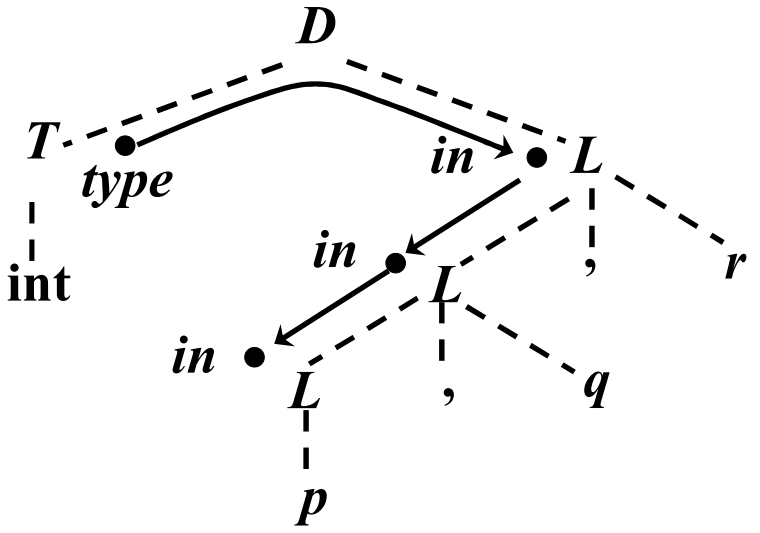
$T \rightarrow$ int {*T. type = integer*}

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

$L \rightarrow L_1$, id {*addtype* (id.*entry*, *L.in*)}

$L \rightarrow$ id {*addtype* (id.*entry*, *L.in*)}

产 生 式	代 码 段
$D \rightarrow TL$	
$T \rightarrow$ int	$val[top] = integer$
$T \rightarrow$ real	$val[top] = real$
$L \rightarrow L_1$, id	$addType(val[top], val[top-3])$
$L \rightarrow$ id	$addType(val[top], val[top-1])$



L属性定义的LR实现

- **Case 2: 分析栈上的继承属性---位置不可预测**

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABC$	$C.i = A.s$
$C \rightarrow c$	$C.s = g(C.i)$

增加标记非终结符，使得位置可以预测

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABMC$	$M.i = A.s$; $C.i = M.s$
$C \rightarrow c$	$C.s = g(C.i)$
$M \rightarrow \varepsilon$	$M.s = M.i$

如果考虑*M*.s的可计算性

L属性定义的LR实现

- **Case 2: 分析栈上的继承属性---位置不可预测**

继承属性是某个综合属性的一个函数

$S \rightarrow aAC$	$C.i = f(A.s)$
$C \rightarrow c$	$C.s = g(C.i)$

增加标记非终结符，把$f(A.s)$的计算移到对标记非终结符归约时进行

$S \rightarrow aANC$	$N.i = A.s;\ C.i = N.s$
$N \rightarrow \varepsilon$	$N.s = f(N.i)$
$C \rightarrow c$	$C.s = g(C.i)$

L属性定义的LR实现

- **例 数学排版语言EQN**

$S \rightarrow$ {$B.ps = 10$ }
B {$S.ht = B.ht$ }

$B \rightarrow$ {$B_1.ps = B.ps$ }
B_1 {$B_2.ps = B.ps$ }
B_2 {$B.ht = max(B_1.ht, B_2.ht)$ }

$B \rightarrow$ { $B_1.ps = B.ps$ }
B_1
sub { $B_2.ps = shrink(B.ps)$ }
B_2 {$B.ht = disp\ (B_1.ht, B_2.ht)$ }

$B \rightarrow$ text {$B.ht = \text{text}.h \times B.ps$ }

L属性定义的LR实现

产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht\ )$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht\ )$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht =$ text.$h \times B.ps$

L属性定义的LR实现

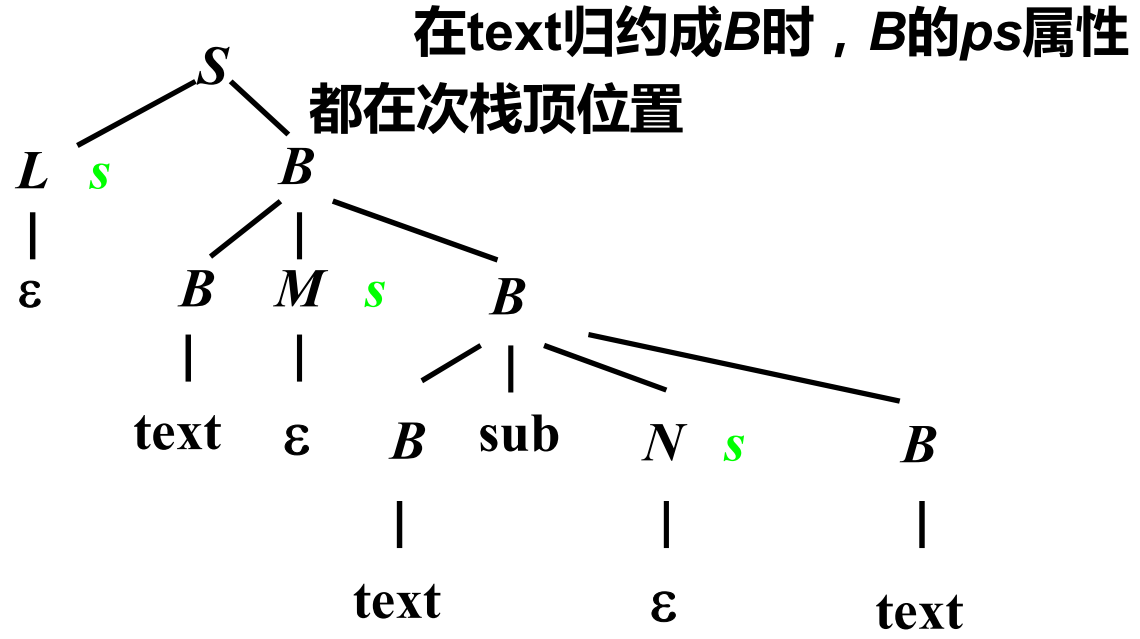
产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht =$ text.$h \times B.ps$

L属性定义的LR实现

产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$ 兼有计算功能
$B \to$ text	$B.ht =$ text.$h \times B.ps$

L属性定义的LR实现

举例说明



L属性定义的LR实现

产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$; $B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

4.4 L属性的自下而上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$val[top] = val[top] \times val[top-1]$

Thank you!