第33卷 第3期
2010年3月

计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS

Vol. 33 No. 3
Mar. 2010

$LR(k)$任意文法位置的断点调试方法

许 福 金茂忠 李 虎 宋 淼

(北京航空航天大学软件工程研究所 北京 100191)

摘 要 $LR(k)$文法能描述所有确定型上下文无关语言,广泛应用于各类分析器生成器中.传统的$LR(k)$文法断点调试方法仅支持在产生式右部末尾设置断点(后文简称尾部断点),不支持在产生式右部中间位置设置断点(后文简称中间断点),这给分析器的开发和调试带来了不便.文中提出了一种新颖的$LR(k)$文法断点调试方法,不但支持传统的尾部断点,还支持中间断点.该方法可显著增加可利用的断点数量,可以跟踪到更细粒度的文法成分,从而帮助用户更好地进行文法调试,降低分析器的开发难度.

关键词 $LR(k)$文法;文法调试;断点调试

中图法分类号 TP311 **DOI号**:10.3724/SP.J.1016.2010.00483

A Breakpoint Debugging Method for $LR(k)$ Grammars

XU Fu JIN Mao-Zhong LI Hu SONG Miao

(*Software Engineering Institute*, *Beihang University*, *Beijing* 100191)

Abstract $LR(k)$, the most general category of deterministic linear-time parsing, is widely used in various parser generators. The traditional breakpoint debugging methods for $LR(k)$ grammars only support the breakpoints at the end of productions and do not support the breakpoints in the middle of productions. This brings lots of difficulties for grammar debugging. This paper proposes a new breakpoint debugging method, which supports breakpoint debugging for both middle and tail grammar positions. It can increase the debugging positions remarkably and can help debug grammars more effectively.

Keywords $LR(k)$ grammar; grammar debugging; breakpoint debugging

1 引 言

$LR(k)$文法广泛应用于计算机语言的语法定义,它只要求向前看k个符号即能做正确的自左至右的语法分解,在各类分析器生成器中有广泛应用.由于分析器开发的复杂性,目前绝大多数分析器都是通过分析器自动生成工具生成的,越来越多的普通软件开发者利用分析器生成器来构造各种程序分析工具[1].与分析器设计领域的专家不同,普通开发者一般并不精通程序分析理论,因此,良好的文法调试支持对他们是不可或缺的.

$LL(k)$文法采用自顶向下的递归下降分析方法,文法和分析器间有明显的结构对应关系,可以借助传统的源码调试工具(如GNU gdb[2])进行调试.然而,对于$LR(k)$文法,却没有类似的源码调试工具可用.这是因为$LR(k)$文法采用自底向上的分析方法,文法和分析器间没有明显的结构对应关系,利用源码调试工具难以实现分析器源码和对应文法的直观结构映射关系.$LL(k)$文法只是$LR(k)$文法的

收稿日期:2009-04-15;最终修改稿收到日期:2009-09-14.本课题得到国家自然科学基金(60703057,60573084)资助.本方法已获得国家知识产权局发明专利(专利号:ZL200610113288.9).**许 福**,男,1979年生,博士研究生,主要研究方向为编译技术、软件工程.E-mail:xufu@buaa.edu.cn.**金茂忠**,男,1941年生,教授,博士生导师,主要研究领域为编译技术、软件工程及软件测试.**李 虎**,男,1974年生,讲师,主要研究方向为编译技术、软件测试.**宋 淼**,女,1982年生,博士研究生,主要研究方向为编译技术、软件工程.

子类，与 $LL(k)$ 相比，$LR(k)$ 可以处理左递归，能描述所有确定型上下文无关语言，因而绝大多数的分析器生成器均基于 $LR(k)$（或其子类）文法，因此有必要研究适用于 $LR(k)$ 文法的调试技术.

2 基本定义

本文假定读者具有文法、语言和程序分析方面的基础知识，有关内容可参见文献[3-6]. 下面介绍本文用到的几个相关定义.

定义1. 上下文无关文法. 上下文无关文法是一个四元组 $G=(N,T,P,S)$. 其中，N 是非终结符的有限集，T 是终结符的有限集，$T\cap N=\varnothing$；P 是形如 $A\rightarrow\beta$ 的（重写）规则的有限集，A 称为规则左部，β 称为规则右部；$S\in N$，是文法的开始符号.

定义2. $LR(k)$ 文法. $LR(k)$ 文法是上下文无关文法的真子集，可以描述所有确定型上下文无关语言，它采用自底向上的移进-归约分析方法，只要求向前看 k 个符号即能做自左至右的语法分解.

定义3. 文法位置、主要位置、衍生位置. 文法位置是一个二元组 $[i,j]$，其中 i 表示产生式编号，j 表示产生式右部第 j 个符号后的位置，j 的取值范围为 $0\le j\le$（产生式 i 右部符号数目）. 文法开始符号所在的产生式编号为0，其它产生式采用非重复的递增顺序编号. 文法位置 $[0,0]$ 和 $[i,j]$（$j\ge1$）称为主要位置，文法位置 $[i,0]$（$i\ge1$）称为衍生位置. 图2给出了图1文法对应的文法位置.

规则0. $S\rightarrow E$
规则1. $E\rightarrow E+i$
规则2. $E\rightarrow i$

图1 一个 $LR(1)$ 文法 $G(S)$

规则0. $S\rightarrow[0,0]\ E\ [0,1]$
规则1. $E\rightarrow[1,0]\ E\ [1,1]+[1,2]\ i\ [1,3]$
规则2. $E\rightarrow[2,0]\ i\ [2,1]$

图2 $G(S)$ 文法的文法位置

定义4. 项目、初始项目、主要项目、衍生项目. 文法 G 的项目是一个三元组 $[i,j,\gamma]$，其中 $[i,j]$ 表示文法位置，γ 表示向前看符号集. 项目 $[i,j,\gamma]$ 也可以表示为 $A\rightarrow\alpha\cdot\beta\{\gamma\}$. $[0,0,\$]$ 称为文法的初始项目. 项目 $[i,j,\gamma]$ 是一个主要项目当且仅当 $[i,j]$ 是一个主要位置，项目 $[i,j,\gamma]$ 是一个衍生项目当且仅当 $[i,j]$ 是一个衍生位置.

定义5. 空非终结符. 空非终结符是一个特殊的非终结符，它唯一出现在文法中的某条规则左部，该规则右部为空字符串 ε.

3 断点类型判定算法

传统的 $LR(k)$ 文法断点调试方法仅支持尾部断点，不支持中间断点，这是 $LR(k)$ 分析方法的一个固有属性，因为断点可被视为特殊的语义动作，而标准的 $LR(k)$ 分析方法只支持在产生式末尾（即在执行归约动作时）调用语义动作，不支持在产生式右部中间位置调用语义动作[3]，这给分析器的开发和调试带来了不便. 在产生式右部中间位置设置断点，是很多程序设计语言分析和处理中的一个常见需求，例1给出了一个例子.

例1. 图3是一段典型的C语言if判断语句，其对应的汇编代码如图4所示. 其中的 $a\parallel b$ 为逻辑表达式语句，C语义规定，先计算 a 的值，如果 a 为真，则逻辑表达式 $a\parallel b$ 也为真，不再进一步计算 b 的值，图4地址0040D43D和0040D441处的语句说明了这点. “cmp dword ptr [ebp-4],0”，首先比较“dword ptr [ebp-4]”（即变量 a）的值是否为0，如果不为0，则表达式 $a\parallel b$ 的值为真，直接执行“jne main+39h(0040d449)”，跳转到地址0040D449处，执行“mov dword ptr [ebp-0Ch],1”（即执行语句 $c=1$）.

```
if (a || b)
    c=1;
else
    c=2;
```

图3 一段C语言if判断语句

```
12:    if (a || b)
0040D43D   cmp    dword ptr [ebp-4],0
0040D441   jne    main+39h (0040d449)
0040D443   cmp    dword ptr [ebp-8],0
0040D447   je     main+42h (0040d452)
13:          c=1;
0040D449   mov    dword ptr [ebp-0Ch],1
14:    else
0040D450   jmp    main+49h (0040d459)
15:          c=2;
0040D452   mov    dword ptr [ebp-0Ch],2
```

图4 if判断语句对应的汇编代码

图5给出了 $a\parallel b$ 对应的逻辑表达式语句的产生式规则. 若要开发一个C语言动态分析程序，计算逻辑表达式 $a\parallel b$ 的值，就需要在 $\parallel$ 前设置断点（图6）.

expr_stmt → expr ‖ expr

图5 逻辑表达式语句

expr_stmt →expr ● ‖ expr

图 6　在 ‖ 前设置断点

通过对文法进行简单变换，给其引入空非终结符，则可以使标准的 $LR(k)$ 分析方法支持产生式右部中间位置的断点调试. 比如，对于图 5 所示文法，可以在第一个 expr 后插入一个空非终结符 X，然后在新添加的产生式（图 7 中的规则 2）末尾设置断点.

规则 1. expr_stmt →expr X ‖ expr
规则 2. $X \to \varepsilon$ ●

图 7　在产生式右部中间设置断点

空非终结符只推导出空串 ε，因此插入空非终结符后得到的文法与原始文法识别的语言相同. 然而，并不是所有的文法位置都可以插入空非终结符，在 $LR(k)$ 文法的某些文法位置插入空非终结符，可能会引入分析冲突，导致变换后的文法不是 $LR(k)$ 文法，造成分析算法识别失败，例 2 给出了一个例子.

例 2.　图 1 所示文法是 $LR(1)$ 文法，如果在"规则 1"右部的符号 E 之前插入空非终结符 X，将导致变换后的文法不再是 $LR(k)$ 文法（图 8）.

规则 0. $S \to E$
规则 1. $E \to X\,E+i$
规则 2. $E \to i$
规则 3. $X \to \varepsilon$ ●

图 8　插入空终结符导致分析冲突

由语义处理需要而插入的空非终结符，如果导致了语法分析冲突，一般有两种处理策略：

（1）只执行与分析树（分析森林）构造有关的语义动作，延迟执行其它语义动作，待分析树（分析森林）构造完毕后，进行第 2 遍处理，执行其它语义动作. 有文献称这种策略为"两遍编译"（two-pass compiling）[7]；

（2）改写文法消除分析冲突. YACC[8] 采用这种处理策略. 如果插入语义动作后引入了分析冲突，YACC 要求改写文法消除冲突，否则采用 YACC 的缺省冲突处理规则处理：① 移进-归约冲突，选择移进；② 归约-归约冲突，选择先声明的产生式执行归约.

对断点调试来讲，这两种处理策略均不合适. 第 1 种策略的缺陷在于，必须在分析树（分析森林）生成完毕后才能执行相应的语义动作，而断点调试要求在语法分析过程中执行语义动作，这时分析树（分析森林）尚未生成完毕；第 2 种策略的缺陷在于，普通软件开发者难以掌握并熟练应用文法改写技巧. 文法改写是一项高度专业并十分困难的软件开发活动，改写过程中要遵循复杂的等价变换规则，并且改写文法极易出错，导致改写前和改写后的文法定义的语言不一致，而这种不一致性无法通过程序自动验证判断，因为文法的等价性在形式语言理论中属于不可判定问题[4]. 此外，断点本质上不是文法的有效组成部分，调试完毕后，由于断点调试需要而插入的空非终结符，还需要从文法中删除，这需要再次修改文法. 从上面的分析可看出，通过修改文法解决插入断点造成的分析冲突，在断点插入和删除后需要两次修改文法，工作量很大，因而是不可取的.

综上，通过简单的文法变换，插入空非终结符，并在新添加的产生式末尾添加断点，不能解决中间断点的文法调试问题. 因此，需要设计相应的算法来区分不同的断点位置，以判断哪些位置能插入断点，哪些位置不能插入断点. 本文引入了两个新的概念：有效断点位置和无效断点位置，来区分不同的断点类型.

定义 6.　有效断点位置、无效断点位置. 对于 $LR(k)$ 文法 G，令 $\pounds'$ 为不包含文法位置 $[i,j]$ 的其它任意文法位置的集合，$\pounds''=\pounds'\cup[i,j]$，在 $\pounds'$ 插入空非终结符后得到文法 G'，在 $\pounds''$ 插入空非终结符后得到文法 G''. 文法位置 $[i,j]$ 是一个有效断点位置，当且仅当对所有的 $\pounds'$，要么文法 G' 和 G'' 都是 $LR(k)$ 文法，要么都不是. 如果文法位置 $[i,j]$ 不是有效断点位置，则称其为无效断点位置. 简单来说，如果在文法位置 $[i,j]$ 处插入断点，不会造成分析冲突，则 $[i,j]$ 是有效断点位置，否则 $[i,j]$ 是无效断点位置.

根据定义 6，算法 1 给出了一种直观的断点类型判定算法.

算法 1.　直观断点类型判断算法（计算所有文法位置）.

输入：$LR(k)$ 文法 $G=(N,T,P,S)$

输出：文法 G 的每个文法位置的断点类型判定，即有效断点还是无效断点

步骤：

1. 在文法 $G[S]$ 的文法位置 $[i,j]$ 处，插入空非终结符 X，得到扩展文法 $G[S']$；

2. 为文法 $G[S']$ 生成 $LR(k)$ 分析表. 检查分析表中是否存在冲突，如果不存在冲突，则说明位置 $[i,j]$ 为有效断点位置，否则为无效断点位置；

3. 对所有文法位置，重复执行步 1 和步 2，计算出所有文法位置的断点类型.

根据定义 6 可知，算法 1 是正确的. 算法 1 的缺

陷在于速度太慢,因为对每个文法位置,都要构造一次 $LR(k)$ 分析表.文法规模一大,计算开销将迅速增加.以本文第5部分提及的Java 1.4文法为例,其 $LALR(1)$ 文法包含278条产生式,2690个文法位置,生成一次 $LALR(1)$ 分析表平均耗时0.057s.根据算法1,计算完2690个文法位置共需要:2690×0.057s=153.33s.考虑到Java文法很规整,在程序设计语言文法中属中等复杂度文法,如果是C++等复杂语言文法,耗时会更长,因此,算法1不能满足交互式断点调试的需要.

如果文法是固定不变的,可以仅执行一次计算,把断点类型信息存储到一个表中.以后每次调试时,不再执行计算,直接查询该表,就可以获知每个位置是否是有效断点位置.然而,文法调试的主要动机就是修正文法中可能存在的错误,这个阶段,文法一般还未定型,需要频繁修改.因此,这种方法也不可行.

另外一种可能的解决办法是,只对用户实际设置断点的文法位置进行计算判断,而不是像算法1一样,对所有的文法位置进行计算,如算法2所示.

算法2. 直观断点类型判断算法(计算用户实际插入断点的文法位置).

输入:$LR(k)$ 文法 $G=(N,T,P,S)$

输出:文法 G 中用户插入断点的文法位置类型判定,即有效断点还是无效断点

步骤:

1. 在用户设置断点的文法位置 $[i,j]$ 处插入空非终结符 X,得到扩展文法为 $G[S']$;
2. 为文法 $G[S']$ 生成 $LR(k)$ 分析表.检查分析表中是否存在冲突,如果不存在冲突,则说明位置 $[i,j]$ 为有效断点位置,否则为无效断点位置;
3. 对所有插入断点的文法位置,重复执行步1和步2,即可判定出所有插入断点位置的断点类型.

算法2仅计算实际设置断点的文法位置,不计算所有文法位置,因而,算法2比算法1执行速度快.算法2的不足有两点:(1)当用户设置的断点数目较多时,计算开销也会很大,因为每个断点都需要生成一次 $LR(k)$ 分析表,难以满足交互式调试的时间响应需求;(2)断点只能在调试开始前预先设置好,调试过程中不能动态添加断点.这是因为,调试前仅判断了用户插入断点的文法位置,没判断其它文法位置,如果调试过程中添加断点,可能引入新的分析冲突,造成分析失败.

本文算法可以解决算法1和算法2的不足,仅生成一次 $LR(k)$ 分析表即可计算出所有文法位置的断点类型,调试前和调试过程中可以动态添加、删除断点,拥有较高的时间性能,能满足交互式调试的时间需求.

本文算法借鉴了Purdom等提出的部分状态位置图[9],利用部分状态位置图来区分不同的断点类型.在正式介绍算法前,先介绍几个相关的定义.

定义7. 前驱决定结点、直接前驱决定结点.对于有向图 $G=(V,A,r)$,其中 V 表示顶点集合,A 表示有向边集合,r 表示开始结点,V 中除了开始结点 r 外的其它结点都可以从 r 经过一条或多条有向边到达.如果从结点 r 到结点 v 的每一条路径都经过结点 w,则称结点 w 为结点 v 的前驱决定结点.如果 p 和 q 是 G 的结点,p 是 q 的前驱决定结点当且仅当从 r 到 q 的每一条路径都经过 p.结点 p 是结点 q 的直接前驱决定结点当且仅当 p 是 q 的前驱决定结点,并且 q 除了 p 以外的其它任何前驱决定结点也是 p 的前驱决定结点.

定义8. 部分状态 (x,α).对于给定的状态 x,(x,α) 表示状态 x 的部分状态,其中 x 是状态编号,α 是向前看符号,(x,α) 表示状态 x 中以 α 为向前看符号的所有项目的集合.任何一个状态都由一个或多个部分状态组成.每个部分状态 (x,α) 包括下述项目:(1)任何可移入某个终结符或者用某条产生式归约的项目 d;(2)任何可推导出 d 的主要项目(令这些主要项目的集合为 M);(3)任何在 M 的闭包[5-6]中并且可以推导出 d 的项目.

定义9. 部分状态位置图.对于每个部分状态 (x,α),都存在一个对应的部分状态位置图(Partial State Position Graph,PSPG).PSPG由一个初始结点(用I表示),一个或多个终止结点(用E表示),一个或多个动作结点以及一个或多个中间结点组成.PSPG是一个有向图,其中的有向边表示项目间的 ε 转换关系.比如,有向边 $a\rightarrow b$(其中 a、b 既不是初始结点,也不是终止结点,而是PSPG中的中间结点或动作结点),表示结点 a 通过一次 ε 转换可变换到 b.从初始结点I到 M(M 的含义见定义8)中的每个元素都有一条有向边,从 d(d 的含义见定义8)到终止结点E也有一条有向边.除初始结点和终止结点外的每个结点(即中间结点和动作结点)都对应一个文法位置.

图9是一个算术表达式文法,表1给出了该文法的初始状态0对应的 $LR(1)$ 分析表.从表1可看出,状态0共包含两个部分状态:$(0,i)$ 和 $(0,()$,其对应的部分状态位置图如图10和图11所示.

规则 0. $S \rightarrow E\ \$$
规则 1. $E \rightarrow E+T$
规则 2. $E \rightarrow T$
规则 3. $T \rightarrow T * P$
规则 4. $T \rightarrow P$
规则 5. $P \rightarrow i$
规则 6. $P \rightarrow (E)$

图 9　一个算术表达式文法 $G[S]$

表 1　$G[S]$文法初始状态的 $LR(1)$分析表

State	项		动作						转移			
			$	+	*	i	(	)	S	E	T	P
	$S \rightarrow .E\ \$$	{ε}										
	$E \rightarrow .E+T$	{$ +}										
	$T \rightarrow .T$	{$ +}										
0	$T \rightarrow .T*P$	{$ + *}				S5	S6			1	7	4
	$T \rightarrow .P$	{$ + *}										
	$T \rightarrow .i$	{$ + *}										
	$E \rightarrow .(E)$	{$ + *}										

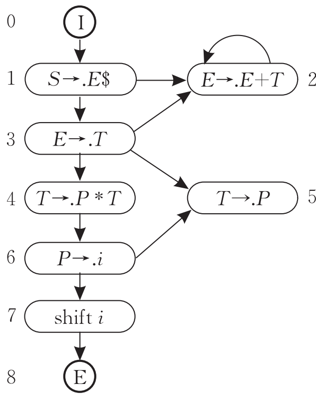


图 10　部分状态(0,i)对应的部分状态位置图

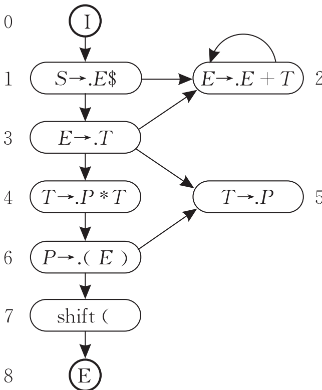


图 11　部分状态(0,()对应的部分状态位置图

部分状态位置图中的结点可分为两大类：特殊结点和中间结点.

(1) 特殊结点

包括 3 个子类：① 初始结点 I，图 10 中对应结点 0；② 终止结点 E，图 10 中对应结点 8；③ 动作结点，可能的执行动作包括 3 类：shift、reduce、accept，图 10 中的动作结点为结点 7.

(2) 中间结点

根据中间结点和动作结点的关系不同，可把中间结点分为两个子类：① 动作结点的前驱决定结点，图 10 中对应结点 1,3,6；② 动作结点的非前驱决定结点，图 10 中对应结点 2,4,5.

根据结点是否包含自旋(可以推导出自身的结点为自旋结点，否则为非自旋结点)，可把中间结点分为两个子类：① 自旋结点，图 10 中对应结点 2；② 非自旋结点，图 10 中对应结点 1,3,4,5,6.

定理 1.　对于 $LR(k)$ 文法 $G[S]$，其部分状态位置图中的自旋结点对应的文法位置为无效断点位置.

证明.　根据定义，自旋结点可以推导出自身，因此自旋结点对应的项目可以抽象为 $E \rightarrow .E\gamma$(其中 γ 为一个、多个终结符或非终结符的组合)，令该产生式编号为 i，则自旋结点对应的文法位置为 $[i,0]$. 在文法位置 $[i,0]$ 处设置断点，需要插入空非终结符 X，变换后得到的产生式为 $E \rightarrow .XE\gamma$. 产生式 $E \rightarrow .XE\gamma$ 迭代 n 次后可得到句型 $X^n \beta \gamma^n \eta$，其中 β、η 为一个、多个终结符或非终结符的组合.

若 β 恒为空串 ε，说明 γ 和 E 能且只能推导出 ε，因此，规则 $E \rightarrow E\gamma$ 蜕化为 $E \rightarrow E$ 和 $E \rightarrow \varepsilon$，$E \rightarrow E$ 是有害规则[6]，不应出现在文法定义中，因此，β 不恒为 ε. 同理，γ 也不恒为 ε，否则 $E \rightarrow E\gamma$ 蜕化为 $E \rightarrow E$，为有害规则.

令添加空非终结符 X 后得到的文法为 $G[S']$. 因为 $X^n \beta \gamma^n \eta$ 是文法 $G[S']$ 的句型，若 $G[S']$ 是 $LR(k)$ 文法，则可通过向前看 k 个符号识别句型 $X^n \beta \gamma^n \eta$ 中的 X^n. 而句型 $X^n \beta \gamma^n \eta$ 要求 X^n 后必须要跟着 $\beta \gamma^n$，因为 β、γ 不总为 ε，因此，至少要向后看 $n+1$ 个符号才能识别 X^n. 产生式 $E \rightarrow .XE\gamma$ 迭代次数 n 可为任意次，而 k 为固定值. 令 $n>k$，则实际向前看符号长度 k 小于识别 X^n 必须的 $n+1$，因而 $G[S']$ 不是 $LR(k)$ 文法. 因此，自旋结点对应的文法位置为无效断点位置.　证毕.

对于 $LR(k)$ 文法，Purdom 证明了在 PSPG 的动作结点的非前驱决定结点位置插入语义动作，可能会引入分析冲突，也可能不会引入分析冲突[9]. 根据本文的断点类型定义，这些位置都是无效断点位置(定理 2).

定理 2.　部分状态位置图中，动作结点的非前驱决定结点对应的文法位置为无效断点位置.

Purdom 方法中，除自旋结点和动作结点的非前驱决定结点外，其它位置都归类为自由位置，可自

由插入语义动作而不会引入分析冲突.这个结论是不正确的,因为一个状态可能有多个 PSPG,在这些 PSGP 中独立应用 Purdom 方法,有时会得出矛盾的结论.比如,图 12 文法的 $LR(1)$ 分析表初始状态有两个 PSPG (图 13).根据 Purdom 方法,图 13(a)中的结点 2 不是动作结点的前驱决定结点,因而结点 2 处不能插入语义动作;图 13(b)中,结点 10 不是自旋结点,且结点 10 是动作结点的前驱决定结点,因此结点 10 处可插入语义动作.实际上,结点 2 和结点 10 是同一结点,但从同一状态的两个 PSPG 中,却得出了矛盾的结论,其原因在于,没有把结点 2 和结点 10 在所有 PSPG 中比较判断.

对于某个结点 n,若 n 为非自旋结点,且 n 在所有 PSPG 中均为动作结点的前驱决定结点,结点 n 处才可插入语义动作,即有效断点位置.合并某一状态的所有部分状态位置图中的相同结点得到的图称为状态位置图(State Position Graph,SPG)[17].比如,合并图 13 中的两个 PSPG,可得到图 14 的 SPG.

规则 0. $S \to A$ 规则 1. $A \to B$
规则 2. $A \to C$ 规则 3. $B \to C\,a$
规则 4. $B \to b$ 规则 5. $C \to c$

图 12 文法 $G[S]$

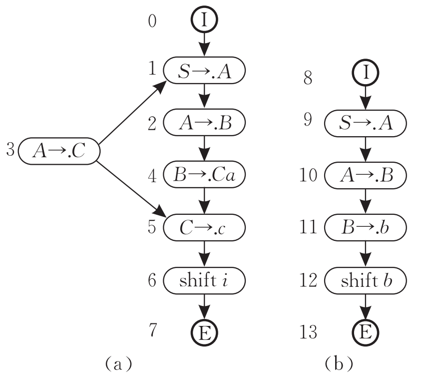


图 13 $G(S)$的初始状态对应的两个 PSPG

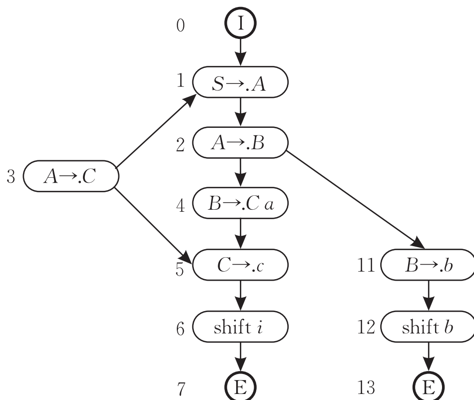


图 14 合并 PSPG 中的相同结点得到 SPG 的过程

定理 3 证明了,结点 n 在所有 PSPG 中均为动作结点的前驱决定结点,当且仅当 n 在 SPG 中是所有动作结点的前驱决定结点.这样,计算有效断点位置转化为查找 SPG 中的所有动作结点的前驱决定结点.

定理 3. 对于 $LR(k)$ 文法 $G[S]$,结点 n 在 SPG 对应的所有 PSPG 中都是动作结点的前驱决定结点,当且仅当结点 n 是 SPG 中动作结点的前驱决定结点.

证明.

充分性.若 n 是 SPG 中动作结点的前驱决定结点,则在 SPG 对应的 PSPG 中 n 也是各个动作结点的前驱决定结点.

根据 SPG 定义,SPG 由各个 PSPG 合并得到,因而各个 PSPG 是 SPG 的子图.若 n 是 SPG 中动作结点的前驱决定结点,则 n 也必然是各个子图动作结点的前驱决定结点,PSPG 是 SPG 的子图,因而 n 也是各个 PSPG 中的动作结点的前驱决定结点.

必要性.若 n 是各个 PSPG 中动作结点的前驱决定结点,则在 SPG 中 n 也是动作结点的前驱决定结点.

根据前提条件,n 是各个 PSPG 中动作结点的前驱决定结点,因此,在某个 PSPG 中,从初始结点 I 到动作结点 A,存在一条路径 p_1,且 n 是路径 p_1 上的一个结点.SPG 由各个 PSPG 合并得到,因此,动作结点 A 也是 SPG 的动作结点,且 p_1 路径存在于 SPG 中.假定 n 不是 SPG 中动作结点 A 的前驱决定结点,则从 SPG 的初始结点 I 到动作结点 A 存在另外一条路径 p_2,且 p_2 不经过结点 n.因为 p_1 和 p_2 都包含动作结点 A,因此它们在同一个 PSPG 中.因为 n 不是 p_2 路径上的结点,路径 p_2 中从初始结点 I 到动作结点 A 不经过 n,因此,n 不是该 PSPG 中动作结点 A 的前驱决定结点,这与前提条件矛盾.因此,n 是 SPG 中动作结点的前驱决定结点. 证毕.

从定理 1～3 可看出,可采用算法 3 来计算 $LR(k)$ 文法的断点类型.

算法 3. 断点类型判断算法(基于 SPG).

输入:$LR(k)$ 文法 $G=(N,T,P,S)$

输出:文法 G 中所有文法位置的断点类型判定,即有效断点还是无效断点

步骤:

1. 为 $LR(k)$ 文法生成 $LR(k)$ 分析表;
2. 为 $LR(k)$ 分析表中的每个状态构造 SPG;
3. 对步 2 中每个 SPG,重复执行步 4～6;

4. 对某个 SPG(令其为 M)，计算 M 中的自旋结点，把自旋结点对应的位置标记为无效断点位置；

5. 计算 M 中动作结点的前驱决定结点，把动作结点的非前驱决定结点位置标记为无效断点位置；

6. 把 M 中其它未标记位置标记为有效断点位置.

下面对算法 3 各步骤采用的具体算法及每个具体算法的时间复杂度做一下介绍和分析.

步 1，生成 $LR(k)$ 分析表，具体算法可参考文献[5]. 随着 k 的增大，分析表的状态空间在最坏状态下为指数增长[4]，分析速度也会变慢. 因此，实际中一般取 $k=1$，生成 $LR(1)$ 分析表可在线性时间内完成[4].

步 2，为每个状态生成 SPG，SPG 的具体构造算法见算法 4，算法 4 也可以在线性时间内完成.

步 4，查找 SPG 中的自旋结点，执行速度取决于图中的顶点数 $|A|$ 和有向边的数目 $[V]$，本质上是侦测图中的环路，是一个线性复杂度算法[10].

步 5，计算动作结点的前驱决定结点，可采用图论中的决定结点查找算法(dominator searching)[11]. Purdom 使用了一个近线性时间(almost linear-time)算法，速度较慢. Harel[12]、Alstrup[13]、Buchsbaum[14] 等提出了一系列改进算法，然而，这些算法仍存在一定缺陷. 2004 年 Georgiadis 和 Tarjan 提出了一个改进的线性时间算法[15]，修正了以前算法的不足，本文采用该算法作为动作结点的决定结点查找算法.

步 6，标记其它所有未标记结点，需要的时间取决于 SPG 中的结点数，也可以在线性时间内完成.

通过上述分析可看出，算法 3 的每个步骤均可在线性时间内完成，算法 3 是一个线性时间复杂度算法.

算法 4.　状态位置图(SPG)构造算法.

输入：$LR(k)$ 文法 $G=(N,T,P,S)$ 及其 $LR(k)$ 分析表 M

输出：每个状态对应的 SPG

步骤：

1. 对 M 中的每个状态 n，执行步 2～4；

2. 为状态 n 创建 SPG 的初始结点 I_n；

3. 为状态 n 中的每个主要项目 mi 创建新结点 N_{mi}，从初始结点 I_n 到 N_{mi} 添加一条有向边；

4. 检查步 3 中的每个主要项目 mi，

① 若 mi 形如 $A\rightarrow\alpha\cdot X\beta,\{\gamma\},\alpha,\beta\in(N\cup T)^*,X\in T$. 生成动作结点 $N_{\text{shift } X}$，从结点 N_{mi} 添加一条有向边到 $N_{\text{shift } X}$；生成终止结点 E，从 $N_{\text{shift } X}$ 添加一条有向边到 E；

② 若 mi 形如 $A\rightarrow\alpha\cdot,\{\gamma\},\alpha\in(N\cup T)^*$. 生成动作结点 $N_{\text{reduce}(A\rightarrow\alpha)}$，从结点 N_{mi} 添加一条有向边到 $N_{\text{reduce}(A\rightarrow\alpha)}$；生成终止结点 E，从 $N_{\text{reduce}(A\rightarrow\alpha)}$ 添加一条有向边到 E；

③ 若 mi 形如 $A\rightarrow\alpha\cdot X\beta,\{\gamma\},\alpha,\beta\in(N\cup T)^*,X\in N$. 对项目 $A\rightarrow\alpha\cdot X\beta$ 执行闭包运算，对闭包运算得到的每个项目创建一个新结点，为两个结点间的每个 ε 转换添加一条有向边. 检查闭包运算得到每个项目的 N_{current}：

a) 若 N_{current} 形如 $Y\rightarrow\cdot\varepsilon,\{\Gamma\}$. 生成动作结点 $N_{\text{reduce}(Y\rightarrow\varepsilon)}$，从 N_{current} 添加一条有向边到 $N_{\text{reduce}(Y\rightarrow\varepsilon)}$；生成终止结点 E，从 $N_{\text{reduce}(Y\rightarrow\varepsilon)}$ 添加一条有向边到 E；

b) 若 N_{current} 形如 $Y\rightarrow\cdot B\eta,\{\Gamma\},B\in T,\eta\in(N\cup T)^*$. 生成动作结点 $N_{\text{shift } B}$，从 N_{current} 添加一条有向边到 $N_{\text{shift } B}$；生成终止结点 E，从 $N_{\text{shift } B}$ 添加一条有向边到 E.

从算法 4 可看出，构造 SPG 的过程和 $LR(k)$ 分析表的生成过程类似，只是增加了生成初始结点、动作结点和终止结点的过程. 如果把 SPG 的构造过程合并到 $LR(k)$ 分析表的生成过程，则一些公共步骤，如计算项目(集)闭包等，只需要计算一次，这样可获得更好的时间性能. 从算法 4 可看出，除了增加特殊结点外，构造 SPG 的时间，与生成 $LR(k)$ 的分析表生成时间相当.

4　支持中间断点和尾部断点的文法断点调试方法

标准的 $LR(k)$ 分析方法不支持在产生式右部中间调用语义动作，要支持产生式右部中间位置的断点调试，必须在相应位置插入空非终结符. 这些空非终结符应该由调试器自动添加. 考虑到添加空非终结符可能会引入分析冲突，造成转换后的文法不是 $LR(k)$ 文法，导致 $LR(k)$ 分析方法失效，因此，调试器应该利用算法 3 来判别断点类型，计算哪些位置可以插入断点，哪些位置不可以插入断点. 算法 5 给出了同时支持中间断点和尾部断点的 $LR(k)$ 文法断点调试算法.

算法 5.　支持中间断点和尾部断点的文法断点调试算法.

输入：$LR(k)$ 文法 $G=(N,T,P,S)$ 和用户设置的断点位置集合 B

步骤：

1. 利用算法 3 计算所有有效断点位置集合 M；

2. 在集合 M 的每个中间断点位置插入空非终结符，为了区分不同的断点位置，插入的空非终结符名称应各不相同，令插入非终结符后得到的文法为 G'；

3. 为 G' 生成 $LR(k)$ 分析器；

4. 调试器加载步 3 生成的分析器执行调试，对输入串进行分析，直至遇到归约动作. 检查归约动作对应的产生式 p 和用户设置的断点集合 B，若产生式 p 中的文法位置不在集合 B 中，则说明用户没有在 p 中设置断点，继续分析过程. 若 p 中的某个文法位置 $[i,j]$ 在集合 B 中，检查 $[i,j]$ 是否在集合 M 中，若 $[i,j]$ 在 M 中，则说明其为有效断点位置，调试器暂停分析过程，显示调试信息. 若 $[i,j]$ 不在 M 中，则说明用户设置了一个无效断点，调试器提示用户，同时忽略该断点，继续分析过程；

5. 调试过程中，根据需要用户可动态添加或删除断点，调试器应根据实际断点设置动态更新集合 B.

算法步 2 之所以在每个有效的中间断点位置都插入空非终结符，主要是为了支持调试过程中动态添加断点. 从算法 5 可看出，插入空非终结符后得到的文法是 G'，调试用的分析器是根据文法 G' 生成的，在所有有效的中间断点位置都插入空非终结符，可以确保用户在调试前或调试过程中设置在有效断点位置的断点，总能被有效暂停. 在调试开始前，若只在用户实际设置断点中的有效中间断点处插入空非终结符(令变换得到的文法为 G'')，调试过程中将不能动态添加断点，因为若动态添加断点，则新得到的变换文法为 G'''，G''' 不同于 G''，而调试器加载的是 G'' 对应的分析器，因此需要停止分析过程，重新为 G''' 生成分析器. 除步 2 外，算法的其它步骤均直观易懂，此处不再赘述.

5 实验结果

一个实用的调试器必须有足够快的响应速度. 从算法 3、算法 4 和算法 5 可看出，$LR(k)$ 分析表生成，SPG 构造和断点类型判定是任意文法位置断点调试的 3 个核心子算法，本文进行了 3 组实验来测试其性能①.

5.1 *LR*(*k*)分析表自动生成性能

与 $LALR(k)$ 分析表相比，$LR(k)$ 分析表往往包含数目巨大的状态空间. 尽管 $LALR(k)$ 识别能力比 $LR(k)$ 弱，但状态空间小得多，且识别能力接近，因此，$LALR(k)$ 在实际中应用更广. 我们选择实际应用最广的 $LALR(1)$ 分析表，来测试其自动生成性能.

Earley 证明了图 15 文法的 $LR(k)$ 分析表状态数随着 n 的增大呈指数级增长[16]. 该文法很适合测试分析表的自动生成性能，表 2 给出了该组文法在 $n=2\sim8$ 时的 $LALR(1)$ 分析表的自动生成时间. 表 3 给出了 Java 1.4 文法的 $LALR(1)$ 分析表自动生成时间.

$$
\begin{array}{ll}
S\rightarrow A_i & (1\le i\le n)\\
S\rightarrow a_jA_i & (1\le i\ne j\le n)\\
S\rightarrow a_iB_i\,|\,b_i & (1\le i\le n)\\
S\rightarrow a_jB_i\,|\,b_i & (1\le i\ne j\le n)
\end{array}
$$

图 15 一个指数增长文法 $G(S)$

表 2 图 15 文法的 *LALR*(1)分析表生成

n	状态	CPU 时钟	生成时间/s
2	25	741344	0.0005
3	78	3977074	0.0026
4	255	24461032	0.0160
5	868	175078296	0.1142
6	2997	2789569909	1.8197
7	10314	68143093218	44.4508
8	35131	888599946893	579.6477

表 3 Java 1.4 文法的 *LALR*(1)分析表生成

状态	CPU 时钟	生成时间/s
448	87337681	0.0570

典型的程序设计语言文法，其 $LALR(1)$ 分析表一般包含几百个状态，即使是比较复杂的 C++ 文法，其 $LALR(1)$ 分析表也仅包含约 1200 个状态[17]，与表 2 中文法 $n=5$ 和 $n=6$ 的状态数相当，与 Java 文法的状态数基本在同一数量级，其分析表的自动生成时间与 Java 文法具有可比性. 从表 2 和表 3 可推断出，典型的程序设计语言文法，其 $LALR(1)$ 分析表的自动生成时间大约在 0.1s 左右，可以满足交互式调试的时间响应需求.

5.2 SPG 构造性能

表 4 给出了图 15 文法在 $n=2\sim8$ 时的 SPG 构造时间，表 5 给出了 Java 1.4 文法的 SPG 构造时间. 从表 4 和表 5 可看出，与 Java 1.4 文法规模相当的程序设计语言文法，其 SPG 构造时间大约在 0.005s 左右，可以满足交互式调试的时间响应需求.

表 4 图 15 文法的 SPG 构造时间

n	状态	CPU 时钟	构造时间/s
2	25	583613	0.0004
3	78	1119128	0.0007
4	255	5960010	0.0039
5	868	25990414	0.0170
6	2997	116758983	0.0762
7	10314	589713020	0.3847
8	35131	2176746289	1.4199

① 3 组数据均在我们研制的 VPGE 分析器自动生成器原型系统上实验得到. VPGE 官方网站：http://www.VPGE.org. 实验机配置：CPU AMD Athlon XP 1800+，512MB 内存，Windows XP 专业版(sp2).

表 5　Java 1.4 文法的 SPG 构造时间

状态	CPU 时钟	构造时间/s
448	7010831	0.0046

5.3　断点类型判定算法性能

在断点类型判定算法中，有两个问题是重要的：一个是算法的执行速度有多快；另一个是对于典型的程序设计语言文法，该算法可以增加多少可用断点.

对于第 1 个问题，本文测试了 Java、C、Pascal 3 种常用程序设计语言文法的断点类型分类算法执行时间，表 6 给出了实验结果. 因为断点类型判定采用的决定结点查找算法是线性的，从表 6 可推断出，和 Java 规模类似的文法，断点类型判定时间大约在 0.03s，可以满足交互式断点调试的时间响应需求.

表 6　3 种常用文法的断点类型判断算法性能

	状态	规则	整个断点	有效断点	无效断点	CPU 时钟	判定时间/s
Java	448	278	2690	2326	364	4836431	0.0032
C	487	241	2178	1460	718	5638129	0.0037
Pascal	327	195	1592	1281	311	3745186	0.0025

对于第 2 个问题，从表 6 可看出，在 Java 文法中，有效断点和无效断点的百分比分别是 86.47% 和 13.53%. 一般情况下，每条产生式右部平均至少有两个符号，即每条产生式平均至少有 3 个断点位置. 传统的断点调试方法只支持尾部断点，因此最多有 33% 的可用断点位置. 考虑到平均情况下产生式右部符号一般远多于两个，因此传统断点调试方法可利用的断点数量一般远低于 33%. 以表 6 的 Java 文法为例，其 *LALR*(1) 文法共包含 278 条产生式，利用传统断点调试方法，最多有 278 个可用断点，有效断点位置占所有断点位置的百分比为 10.33%. 采用本文断点调试方法，可利用的断点数目是 2326，远高于传统方法的可用断点数量.

6　结　论

本文的文法断点调试方法，不仅支持传统的尾部断点，还支持中间断点，可显著增加可供调试的断点数目. 可用断点数目增加带来的益处是可以跟踪到更细粒度的文法成分，从而帮助开发者更好地理解语法分析过程，有效降低分析器的开发难度. 本方法的另一个优点是很容易集成到现有的各种 *LR*(*k*)(及子类)分析器生成器中，具有较为广阔的实际应用价值.

7　进一步工作

当用户启动调试器调试文法时，与前次调试过程相比，即使只添加、删除或修改了一条产生式，*LR*(*k*)分析表生成、SPG 构造和断点类型判定也必须重新执行. 对于复杂的大型文法，每次的计算开销会很大. 因此，在后续的工作中，需要开发增量的 *LR*(*k*)分析表生成、SPG 构造和断点类型判定算法. 这样，当变更少量文法产生式时，采用增量算法可以获得更好的时间性能.

参　考　文　献

[1] Brown C A, Purdom P W. Methodology and notation for compiler front end design. Software-Practice and Experience, 1984, 14(4): 335-346

[2] Stallman Richard M, Pesch Roland, Shebs Stan. Debugging with GDB: The GNU Source-Level Debugger. 9th Edition. Boston, USA: Free Software Foundation, 2002

[3] Grune Dick, Jacobs Ceriel J H. Parsing Techniques: A Practical Guide. Chichester, England: Ellis Horwood Limited, 1990

[4] Sippu S. Soisalen-Soininen E. Parsing Theory. Berlin: Springer, 1990

[5] Aho Alfred V, Lam Monica S, Sethi Ravi, Ullman Jeffrey D. Compilers: Principles, Techniques, and Tools. Boston: Addison Wesley, USA, 1986

[6] Gao Zhong-Yi, Jin Mao-Zhong. The Theory and Construction of Compilers. Beijing: Beijing University of Aeronautics and Astronautics Press, 1990(in Chinese)
(高仲仪，金茂忠. 编译原理及编译程序构造. 北京：北京航空航天大学出版社，1990)

[7] Mössenböck H. A convenient way to incorporate semantic actions in two-pass compiling schemes. Software-Practice & Experience, 1988, 18(7): 691-700

[8] Johnson Stephen C. YACC—Yet another compiler-compiler. Bell Laboratories, New-Jersey: Computer Science Technical Report 32, 1974

[9] Purdom P, Brown C A. Semantic routines and *LR*(*k*) parsers. Acta Informatica, 1980, 14(4): 299-315

[10] Tarjan Robert. Depth-first search and linear graph algorithms//Proceedings of the 12th Annual Symposium on Switching and Automata Theory. East Lansing, Michigan, USA, 1971: 114-121

[11] Lengauer T, Tarjan R E. A fast algorithm for finding dominators in a flow graph. ACM Transactions on Programming Languages and Systems, 1979, 1(1): 121-141

[12] Harel D. A linear time algorithm for finding dominators in flow graphs and related problems//Proceedings of the 17th Annual ACM Symposium on Theory of Computing(STOC'85). Providence, RI, USA, 1985: 185-194

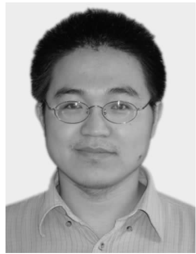
[13] Alstrup S, Harel D, Lauridsen P W, Thorup M. Dominators in linear time. SIAM Journal on Computing, 1999, 28(6): 2117-2132

[14] Buchsbaum A L, Kaplan H, Rogers A, Westbrook J R. Linear-time pointer-machine algorithms for least common ancestors, MST verification and dominators//Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98). Dallas, TX, USA, 1998: 279-288

[15] Georgiadis L, Tarjan R E. Finding dominators revisited: Extended abstract//Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms. New Orleans, Louisiana, USA, 2004: 869-878

[16] Earley J. An efficient context-free parsing algorithm. Assn Computing Machy-Communications, 1970, 13(2): 94-102

[17] Ouyang S T, Wu P C, Wang F J. Locating free positions in *LR*(*k*) grammars. Journal of Information Science and Engineering, 2002, 15(3): 411-423



XU Fu, born in 1979, Ph.D. candidate. His current research interests include compiler theory and software engineering.

JIN Mao-Zhong, born in 1941, professor, Ph.D. supervisor. His research interests include compiler theory, software engineering and software testing.

LI Hu, born in 1974, lecturer. His research interests include compiler theory and software engineering.

SONG Miao, born in 1982, Ph.D. candidate. Her research interests include compiler theory and software testing.

Background

Automated parsing has been widely used in constructing parsers or compiler front ends since the introduction of *LALR*(1) parser generator YACC (Yet Another Compiler-Compiler). Although automated parsing can reduce some workload in constructing parsers, it is still difficult and time-consuming to build parsing tools for many popular programming languages even using the latest parser generators. An important reason is the lack of a powerful debugger.

The authors began to develop an *LALR*(1) and GLR hybrid parser generator—VPGE (Visual Parser Generation Environment) since 2005, and this research got the fund support from NSFC (National Natural Science Foundation of China) with the title "Research on Parsing Theory and Key Techniques in Software Reengineering and Reverse Engineering" (Grant No. 60573084). The debugging method described in this study is part of that project. The debugger used in VPGE can illustrate the detailed parsing process effectively, for example, state stack, symbol stack, lookahead, parse tree, etc. The most important feature of this debugger is that it supports breakpoint debugging for both middle and tail grammar positions. It can help track to much smaller slice of the grammars and can help diagnose errors and conflicts effectively.