程序设计语言的 GLR 优化分析*

李 虎+, 金茂忠, 许 福, 张 敏

(北京航空航天大学 计算机学院,北京 100083)

Optimized GLR Parsing for Programming Languages

LI Hu+, JIN Mao-Zhong, XU Fu, ZHANG Min

(School of Computer Science and Engineering, Beihang University, Beijing 100083, China)

+ Corresponding author: Phn: +86-10-82317640 ext 604, E-mail: lihu_602@hotmail.com, http://sei.buaa.edu.cn/students/tigerli

Received 2004-07-09; Accepted 2004-10-10

Li H, Jin MZ, Xu F, Zhang M. Optimized GLR parsing for programming languages. *Journal of Software*, 2005,16(2):174−183. http://www.jos.org.cn/1000-9825/16/174.htm

Abstract: The motivation of adopting the Generalized LR (GLR) parsing algorithm to construct parsers for programming languages is presented. A multi-level strategy for optimizing a GLR parser and the necessary runtime control mechanisms are introduced to the GLR parsers for flexible disambiguation and for invoking semantic actions attached to grammar rules while avoiding the "delayed semantic action" problem. The algorithm has been implemented in VPGE, a visual parser generation environment. Experimental results show that the speed of the generated GLR parsers is comparable to LALR(1) parsers generated by GNU's Bison when parsing deterministic programming languages.

Key words: LR parsing; generalized LR parsing; disambiguation; semantic action; parse forest

摘 要: 阐述了在程序设计语言语法分析器的构造中采用通用 LR(generalized LR,简称 GLR)分析算法的动机.提出了一个多层次的优化策略,加快了 GLR 分析器的分析速度.为基本的 GLR 算法增加了必要的运行时控制机制,以实现语法分析时调用文法规则附带的语义动作,化解输入串的二义性,同时避免 GLR 分析器可能存在的语义动作延迟问题.优化后的算法已在一个可视化语法分析器自动生成环境 VPGE 中实现.实验结果表明,在分析确定性的编程语言时,自动生成的 GLR 分析器的分析速度与自由软件基金会的 Bison 生成的 LALR(1)分析器的分析速度有可比性.

关键词: LR 分析;通用 LR 分析;二义性化解;语义动作;分析森林

中图法分类号: TP314 **文献标识码**: A

通用 LR(generalized LR,简称 GLR)分析算法,也称 Tomita 算法[1],是从确定性 LR 分析方法扩展而来的一种

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2001AA113100 (国家高技术研究发展计划(863))

作者简介: **李虎**(1974−),男,吉林长春人,博士生,主要研究领域为编译技术,软件测试;**金茂忠**(1941−),男,教授,博士生导师,主要研究领域为软件工程,软件测试,编译技术;**许福**(1979−),男,博士生,主要研究领域为编译技术,软件测试;**张敏**(1980−),女,硕士生,主要研究领域为编译技术,软件测试.

不确定性的自底向上语法分析算法.与其他通用上下文无关文法分析算法相比,GLR 算法具有较低的平均时间复杂度.这使得它在自然语言处理中得到了广泛应用.尽管如此,GLR 算法很少用于程序设计语言分析器的自动构造[2].这主要是由两方面原因引起的.首先是算法存在明显的性能问题.GLR 算法在最好、最坏和平均情况下的时间复杂度分别为 $O(n)$,$O(2^n)$和 $O(n^3)$,而 LR 分析算法总是具有线性的时间复杂度,其中 n 为输入串长度.一个典型的 GLR 分析器通常要比对应的 LR 分析器慢 10 倍~100 倍[3].其次是通用的分析算法缺乏必要的运行时控制机制,以在语法分析时调用语义动作,化解输入串的二义性,执行正确的语义值计算[2].

本文首先阐述在构造程序设计语言分析器时采用 GLR 分析算法的动机,然后剖析 GLR 分析算法的工作原理,并在此基础上引入若干优化技术,加快了 GLR 基本算法的执行速度.在设计实现 GLR 分析器自动生成器时,提出正确调用语义动作而解决文法二义性问题的实用途径.

为行文简洁,本文假设读者已经了解上下文无关文法、LR 分析、GLR 分析以及 LR 分析器自动生成技术的背景知识.文献[4]设立了专门篇幅介绍 LR 分析方法以及 YACC[5],Bison[6]等 LR 分析器自动生成系统.有关 GLR 算法的详细内容及文中出现的术语和符号的进一步解释可参考文献[1,7,8].

1 采用 GLR 分析算法的动机

随着程序设计语言的丰富和发展,无论是设计实现一种新语言的原型,还是开发已有语言的分析器,LALR(1)方法均存在明显缺陷.在设计和描述一种语言的文法时,首要目标是为了阅读者理解语言的语法和语义.因此,语言参考手册上的文法通常不是 LALR(1)文法,而是二义性文法.实践经验表明,对 C++,PROLOG,ROSIE 等复杂语言,以及现实世界不断出现的各种特定于领域的语言(domain-specific language),将手册上的文法改写成等价的 LALR(1)文法或者是不可能完成的[8],或者是极其困难的[9].文法经过改写后,它所生成的语言可能发生了变化.文法的等价性在形式语言理论中属于不可判定问题[10],因而没有工具和算法能够自动判断改写后的文法与改写前的文法所定义的是否为同一语言.此外,文法改写会破坏语言中的许多概念性结构.改写过的文法,非终结符不再对应语言原有的语法成分,而是对应语言的分析过程,不仅难以理解和维护,还会给语义动作的编写带来额外的复杂性.

Brand MVD 对当前主流的程序设计语言语法分析方法和支持工具进行了广泛调查和评价后指出:当前主流的确定性语法分析方法对软件再工程是有害的[11].原因是这些语法分析方法对文法的限制过于苛刻,满足这些限制的实用文法不易获得.此外,由于 LR 文法在组合和划分操作下不具有封闭性质,因而 LR 分析方法不支持文法的组合分析和模块化分析,不能满足软件再工程和逆向工程领域的需要[11].

GLR 算法能够接受任意上下文无关文法,容忍文法中的二义性,同时,上下文无关文法在组合和划分操作下具有封闭性质[10],因而采用 GLR 算法构造分析器可以避免确定性 LR 分析方法存在的上述问题.但算法是否可行和实用,取决于算法的实现效率和算法所需要的运行时控制机制是否得到正确和灵活的实现.

2 GLR 分析算法

GLR 分析器与 LR 分析器都由分析表、分析栈和分析引擎 3 部分组成.表 1 总结了一个典型的 GLR 分析器与典型的 LR 分析器之间的主要区别.GLR 算法的基本思想是,通过在分析过程中采用穷举方法,对输入语句在所有的可能路径上进行分析,从而实现了对语句的识别与翻译.下面结合例子说明 GLR 算法的基本原理.

例 1:设有文法 $G=(N,\Sigma,P,S)$,其中 $N=\{A\}$,$\Sigma=\{a,b,c\}$,$S=A$,$P=\{A\rightarrow AbA,A\rightarrow AcA,A\rightarrow a\}$.其增广文法的产生式集为$\{r_1:S\rightarrow A\#,r_2:A\rightarrow a,r_3:A\rightarrow AbA,r_4:A\rightarrow AcA\}$.文法 G 的 LR(0)分析表见表 2,其中 S_i 表示移进符号 i,R_i 表示应用第 i 条规则执行归约.

GLR 分析器使用多个并行栈用来记录一个输入串的所有可能的最右推导序列.在各栈上,分析器必须同时移进每个位置上的输入符号,保持执行上的同步.GLR 分析算法在同步位置上将这组平行栈的相同状态节点合并,这组合并了相同状态节点的并行分析栈被称为图结构栈,简称图栈(graphic-structured stack,简称 GSS).文献[7]给出了一个基于 LR(0)分析表的标准 GLR 分析算法的伪代码描述.限于篇幅,这里不再引述.

Table 1 典型 GLR 分析器与典型 LR 分析器的主要区别

表 1 Differences between a typical GLR parser and a typical LR parser

Features	GLR parser	LR parser
Typical parse table	LR(0) table allowing conflicts	LALR(1) table free of conflicts
Parse stack	GSS	Linear
Semantic value	No semantic value	In stack node
Parsing algorithm	GLR algorithm	LR algorithm
Typical output	Parse forest	Invoking actions or parse tree
Lookahead	Infinite	Limited, usually one
Disambiguation	Parser generation, parsing time, after parsing	Parser generation

Table 2 例 1 中文法 G 的 LR(0)分析表

表 2 LR(0) parse table for grammar G in example 1

State	Action				Goto	
	a	*b*	*c*	#	*S*	*A*
0	S_2					1
1		S_5	S_4	S_3		
2		R_2	R_2	R_2		
3	---------- Accept ----------					
4	S_2					6
5	S_2					7
6		S_5,R_4	S_4,R_4	R_4		
7		S_5,R_3	S_4,R_3	R_3		

图 1 以输入串 *abaca#*为例,展示了 GLR 分析算法运行时图栈的动态变化和分析森林的生成过程.

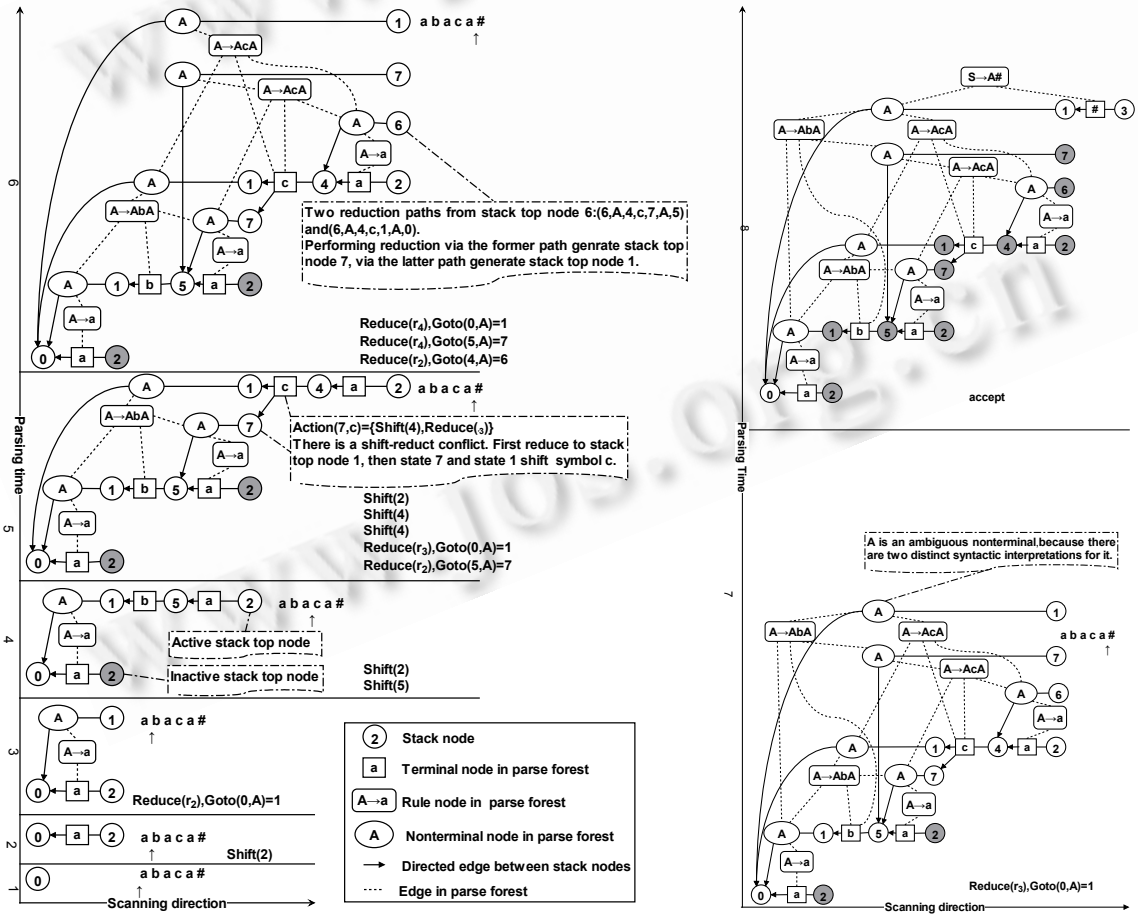


Fig.1 The behaviors of graphic-structured stack and parse forest in GLR parsing

图 1 GLR 分析算法运行时图栈的动态变化和分析森林的生成过程

在实现这个算法时,我们总结出关于算法运行时性质的一组观察结果.这些观察结果详细剖析了 GLR 算法的工作原理,揭示了算法的性能瓶颈,这也是我们进行算法优化的前提和基础.在下面的观察描述中,用六元组 $LRA(G)=\langle K,V,P,s_0,Goto,Action\rangle$ 表示文法 $G(N,\Sigma,P,S)$ 的 LR(0)自动机[4],用状态编号表示图栈中的状态节点,$w=(a_1,\ldots,a_i,\ldots,a_n)$ 为输入串,$S_{top}=\{s_1,\ldots,s_j,\ldots,s_m\}$ 为图栈的活动栈顶节点(active stack top node)集.

观察 1:对栈顶节点 $s^-_j\in S_{top}$ 和面临的输入符号 a_i,若 $Action(s^-_j,a_i)\supseteq Reduce(p:A\rightarrow\alpha\in P)$ 且 $|\alpha|=k$,算法要搜索和保存所有从 s^-_j 出发的包含 $k+1$ 个状态节点的归约路径(reduction path).然后沿每一条归约路径 $(s^-_j,t_j,\ldots,s_{j-k+1},t_{j-k+1},s\_)$ 执行一个归约动作:生成一个新的栈顶状态节点 s,为产生式的左部符号 A 生成一个新的规则节点 $t_{A\rightarrow\alpha}$ 和符号节点 t_A,使 $t_{j,\ldots,}t_{i-k+1}$ 成为 t_A 的孩子,$t_{A\rightarrow\alpha}$ 成为 t_A 的孩子,并且从节点 s 到节点 $s\_$ 增加一条有向边 $(s,t_A,s\_)$.

观察 2:所有活动栈顶节点上的归约动作全部执行完毕后,才能执行移进动作.

观察 3:设 $S_{top}=\{s_1,\ldots,s_i,\ldots,s_j,\ldots,s_m\}$ 为任意时刻的活动栈顶节点集,如果 $i\neq j$,则有 $s_i\neq s_j$.

观察 4:设 $s^-_j\in S_{top}$ 是一个活动的栈顶节点,算法沿着一条归约路径 $(s^-_j,t_j,\ldots,s_{j-k+1},t_{j-k+1},s\_)$ 执行一个归约动作 $Reduce(A\rightarrow\alpha)$ 前(设 $|\alpha|=k,s=Goto(s\_,A)$),如果 $s\in S_{top}$ 且从节点 s 到节点 $s\_$ 之间还没有建立有向边,那么这两个节点之间新增的有向边有可能是从栈顶节点 s 出发的一条归约路径的一部分.

观察 5:设 $s^-_j\in S_{top}$ 是一个活动的栈顶节点,算法沿着一条归约路径 $(s^-_j,t_j,\ldots,s_{j-k+1},t_{j-k+1},s\_)$ 执行一个归约动作 $Reduce(A\rightarrow\alpha)$ 前(设 $|\alpha|=k,s=Goto(s,A)$),如果 $s\in S_{top}$ 且 s 到节点 $s\_$ 之间已经建立了有向边,这说明当前已经移进的子串 $v\in w$ 是文法 G 的一个二义性句子的前缀,非终结符 A 是文法 G 的一个二义性非终结符(ambiguous nonterminal).文法 G 对 A 存在两种不同的解释(参考图 1 的第 7 步),即存在 $\alpha,\beta\in V^*$ 和从 $\alpha A\beta$ 到 $v\beta$ 的两个不同的最右推导序列 π_1 和 π_2,使得 $S\Rightarrow^*\alpha A\beta\Rightarrow^{\pi 1}v\beta$ 且 $S\Rightarrow^*\alpha A\beta\Rightarrow^{\pi 2}v\beta$.

3 GLR 基本算法的优化

GLR 基本算法的优化可以在 3 个层次上同时进行,即分析表的优化、图栈操纵算法优化和通用源代码优化.

3.1 分析表

Billot 和 Lang 等人的研究结果发现,采用 LR(0)分析表的 GLR 分析器适用于自然语言处理[12].文献中对 GLR 算法的介绍也大都基于 LR(0)分析表.这是因为自然语言的二义性绝大多数是全局二义性,LR(0)分析表中的冲突一般不能通过增加向前看符号的个数予以解决.从理论上讲,GLR 分析器可以使用任何 LR 分析表.Parr TJ 统计了 22 种经过改写后的常见程序设计语言文法(不含 C++等复杂语言的文法)的确定性分析器对向前看符号个数的需求.在需要若干个向前看符号才能确定的全部分析决策中,约 98%的分析决策只需一个向前看符号便可确定[13].根据 Parr TJ 的统计结果,我们在设计 GLR 分析器自动生成器时没有采用自然语言处理中常用的 LR(0)表,而是采用了 LALR(1)表,这样可以使 GLR 分析器在分析程序设计语言时遇到的语法分析冲突较少,图栈接近于线性,减少了用于维护和操纵图栈的时间、空间开销.

3.2 图栈操作

第 2 节描述的观察结果说明,图栈拆分、搜索和维护归约路径集以及归约动作施加在图栈上的复杂操作,是 GLR 分析算法的性能瓶颈.

为了减少图栈的拆分,我们将 GLR 分析算法设计成一种交互式算法.遇到语法分析冲突时,交互式算法使用冲突信息作为参数,回调(callback)用户编写的冲突解决例程.一般情况下,用户都可以根据发生冲突时的前文,即当前已经分析过的语法成分和建立的语义信息,如符号表和标识符的类型信息等,确定如何从发生冲突的分析动作中正确地选择一个.用户没有编写冲突解决例程或所需要的前文信息不易收集时,按 GLR 分析算法的规则执行所有可能的分析动作.图 2 给出了一个例子和有关算法的伪代码描述.

对于没有二义性的输入(子)串,GLR 分析算法的时间复杂度是线性的.但是,由于 GLR 分析器要维护比 LR 分析器复杂得多的数据结构,因此它的常系数比 LR 分析算法大得多.为了降低常系数,我们将移进动作分为两类:确定性移进和标准 GLR 移进.将归约动作分为 3 类:确定性归约(deterministic reduction)、简单 GLR 归约

(simple GLR reduction)和标准 GLR 归约(standard GLR reduction).

在当前的活动栈顶所代表的 LR 分析栈上执行一个归约动作之前,首先判断图栈是否满足下列结构特征:

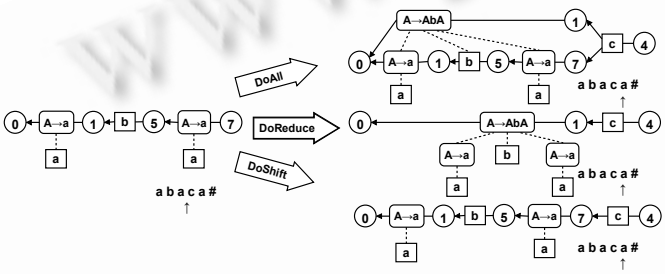
(1) 图栈只有一个活动栈顶节点;

(2) 从一个活动栈顶节点出发的归约路径中,除最后一个状态节点外所有状态节点的出度均为 1.

如果上述两个条件均满足,则执行确定性归约动作.在只满足条件(2)而不满足条件(1)时,执行简单 GLR 归约.两个条件均不满足时,执行标准 GLR 分析算法的归约动作.图 3 直观地说明了 3 类归约动作的区别.

对于移进动作,如果条件(1)满足,执行一个确定性移进动作,否则执行标准 GLR 分析算法的移进动作.执行确定性移进动作只需要增加新的栈顶节点,不需要在活动栈顶节点集中搜索和比较新增状态(参考第 2 节的观察 3).执行确定性归约动作只需从栈中弹出若干个状态节点(出栈的状态数取决于被归约产生式的右部长度),新增一个活动栈顶节点.同样不需要在活动栈顶节点集中搜索和比较新增状态,也不需要搜索和保存归约路径集(参考第 2 节的观察 1).增加从活动栈顶节点到图栈内部节点的有向边所引发的对大量节点的搜索和比较也无须进行(参考第 2 节的观察 4).

优化后的算法在分析无二义性的输入(子)串时,具有和 LR 分析算法相同的时间复杂度和接近的常系数.仅增加了一小部分时间开销用于判断各类移进和归约动作的执行条件.程序设计语言文法的二义性成分较少,一般可以改写为 LALR(1)文法或近似 LALR(1)文法.因此,文法经过简单改写后,其 LALR(1)分析表中往往不含语法分析冲突或只含有少量未解决的冲突.在分析过程的大部分时间内,图栈近似于线性,满足条件(1)和(2),从而使得分析器在大部分时间内执行时间复杂度最低的确定性移进和确定性归约动作.




```
/* Interactive conflict resolving*/
if(Action(state,lookhead)>1)
 begin/*Callback user-defined routine to resolve conflicts*/
      selected=UsrSolving(state,lookhead);
      case selected
      begin
        /*Select a parsing action */
          Ri: DoReduce(rule i);
          Si: DoShift(state i);
          default:
          /*Perform standard GLR algorithm*/
                DoAll (state,lookhead);
      end;
 end;
```

Fig.2 Interactive algorithm for resolving conflicts

图 2 交互式冲突解决算法

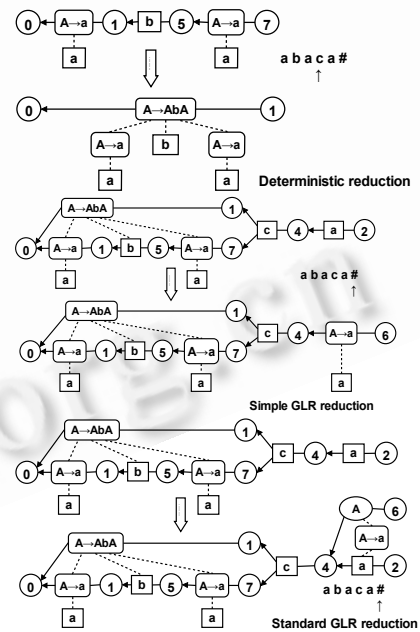


Fig.3 Three types of reductions

图 3 3 类归约动作

3.3 通用的源代码优化技术

第 2 节的观察 3 说明,图栈的活动栈顶节点集在任意时刻都不含有相同状态.根据这一性质,栈顶节点集可采用以状态编号为关键字的 Hash 表存储,而且 Hash 表不会存在冲突.使用 Hash 表存储栈顶节点集,可以明显地改善算法的性能.

图栈的维护要涉及大量动态对象的创建与销毁.在初期的算法实现中我们发现,动态对象的创建与销毁平均占据约 50%的算法运行时间.为了减少这部分操作的时间复杂度,GLR 优化分析器使用了对象池设计模式和

垃圾回收机制.每次动态创建一个新对象后,就在对象池中注册该对象的存储地址和活动状态.销毁一个对象时,并不物理删除它,而是在对象池中将该对象设置为非活动状态.在算法执行到外层循环时,执行垃圾回收,将对象池中的非活动对象顺序删除.对象池设计模式和垃圾回收机制特别适用于由大量动态对象组成的复杂系统,可以有效防止内存泄露,提高系统的运行效率.一些常用的数据结构和代码优化技术也会显著加快算法的运行速度.例如,优先采用顺序存储结构,尽量不使用链式存储结构.避免在程序的内层循环中申请和释放内存,将这些操作移到外层循环中,以便内存的申请和释放可以在内存空间中同一段连续区域内进行,等等.

4 语义动作的调用和二义化解

在自然语言处理中,GLR 分析器输出以某种形式压缩存储的分析森林,作为语义构造程序的标准输入,语法分析过程中不调用语义动作.这是因为在分析森林中如何选择最佳分析树与分析森林中各种语法结构实际出现的概率有关,只有语法分析完成后,相关统计信息才能获得.显然,这种语义处理策略意味着语法分析阶段与语义处理阶段各自需要对输入进行一遍扫描.如何让自动生成的 GLR 分析器与 LR 分析器一样,在语法分析的同时调用规则附带的语义动作,在一次扫描过程中实现 S 属性翻译文法,是实际应用中必须解决的问题.

4.1 语义动作延迟

一种直观的解决方案是让 GLR 分析器模仿 LR 分析器的语义值存储方式,在图栈的状态节点中开辟一个语义值域,这个语义值域专门用于存储文法符号的语义值.实践表明,这种设计方案会引起语义动作延迟问题.

例 2:回顾本文第 2 节的例 1,在图 1 的第 5 步,假设当前一条活动的 LR 分析器处于格局(0157 • *ca*#),其中圆点前的数字串代表状态栈中的内容,圆点后的符号串是待分析的输入串.在该格局下,分析器会遇到一个移进-归约冲突.分析器在执行归约动作 *Reduce*(r_3)时不能调用规则 r_3 附带的语义动作,因为此时分析器无法确定最终的选择应该是归约动作还是移进动作.对程序设计语言来说,正确的分析结果只能有一个.这个语义动作被延迟调用后,为了保证 *S* 属性翻译文法的正确语义,分析过程中后面遇到的所有语义动作都必须被延迟调用.当分析过程进行到某一时刻,图栈中只留下一条活动的 LR 分析栈,或者算法检测到二义性非终结符(参考第 2 节的观察 5),分析器才能确定此前被延迟了的语义动作是否应该被调用.在后一种情况下,由于一个二义性非终结符可能存在几种不同的语法解释,那么它将同时存在多个可能的语义值.GLR 分析器首先依次调用此前被延迟了的所有语义动作,然后调用由用户编写的非终结符语义值合并(merge)例程,将多个可能的语义值合并为一个,保存在该非终结符对应的状态节点中.被延迟调用的语义动作信息和临时产生的语义值都需要暂存在分析器中,使用完后才能被释放.图 4 描述了第 2 节中如图 1 所示的分析过程中语义值在状态节点之间的传播.

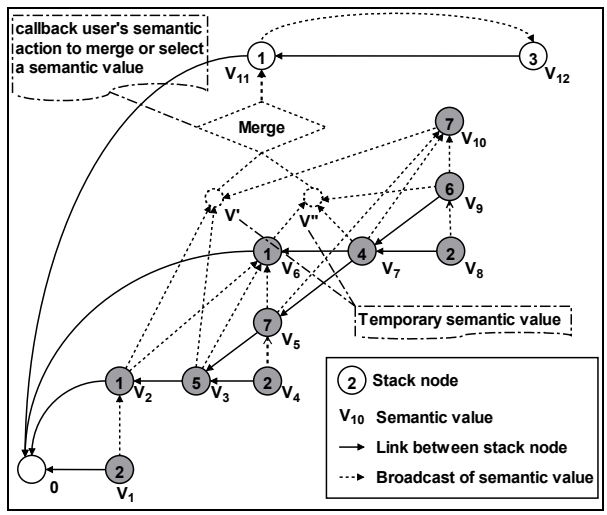


Fig.4 The broadcast of semantic values in Fig.1

图 4 图 1 所示的分析过程中语义值的传播过程

上述设计方案的主要优点是分析过程中不构造分析森林,不保留输入串的二义性,因而分析速度较快,存储开销也较小.然而,如果自动生成的 GLR 分析器采用这一语义处理方案,则语义动作延迟问题将会给分析器的实际应用带来诸多副作用.首先,语法分析器与语言处理器中的其他模块(如词法分析模块、符号表管理模块、类型检查模块等)之间存在着复杂的数据流和控制流依赖关系.分析过程中哪些语义动作可能被延迟调用,以及

每个语义动作被延迟了多长时间才能被正确调用等问题与文法的二义性问题一样,理论上属于不可判定问题.由此造成语法分析器与其他模块之间的通信和同步也将具有不确定性.如果分析器运行在多个进程或线程中,这一问题将变得更为复杂.语义动作的延迟可能会引起线(进)程的等待、阻塞甚至死锁.这些问题的预防和解决要求用户定义的语义动作必须避免因动作延迟而带来的副作用,前提是用户应能准确理解 GLR 分析器的工作原理并能够事先预测出哪些语义动作可能被延迟调用以及语义动作被延迟了多长时间才会被调用.

4.2 基于分析森林剪枝的交互式语义处理和二义性化解

我们在设计GLR分析器生成器时采用了另一种设计方案.首先,文法符号的语义值附加在状态节点之间的有向边上,相当于存储在分析森林的节点中,而不是存储于图栈的状态节点中.其次,在语法分析过程中,让GLR 分析器既构造分析森林,又调用规则附带的语义动作.当遇到暂时不能解决的语法分析冲突时,语义动作无须延迟,而是被正常调用.一个符号所有可能的语义值都被保存在对应的分析森林节点中.每当分析器检测到二义性的非终结符时,分析器自动回调(callback)用户自定义的例程.用户在其例程中可以调用分析器提供给用户的编程接口,对已经构造的部分分析森林进行剪枝,选择可能的语义值中的一个,删除和取消其他的语义值,也可以将多个语义值合并为一个.如果根据当前已经收集到的信息还不能确定二义性非终结符最终的语法解释,分析器将同时保留这两种解释下的语义值,继续分析直至分析器搜集到的信息能够化解该非终符的二义性为止.图 5 示意了这种语义处理和二义性化解方案,仍采用第 2 节图 1 所示的例子以便和图 4 对比.

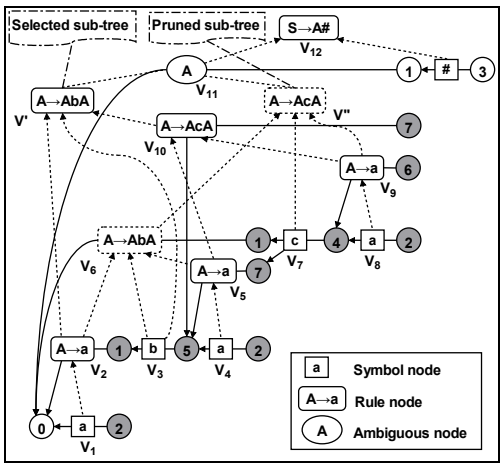


Fig.5 Another broadcast process of semantic values

图 5 另一种语义值传播方式

这种语法分析时基于分析森林剪枝的交互式语义处理和二义性化解方法,为用户解决语法分析冲突和计算文法符号的语义值提供了足够的灵活性.既有效控制了分析森林的规模,防止存储空间爆炸,也避免了语义动作延迟问题.

5 GLR 优化分析器的性能评价

VPGE(visual parser generation environment)是北京航空航天大学软件工程研究所历时两年自主研制的一个可视化分析器自动生成环境.其自动生成的 GLR 优化分析器采用了本文所述的 GLR 优化分析算法和语义处理方法.我们使用 VPGE 自动生成的 GLR 优化分析器和自由软件基金会的 Bison 自动生成的 LALR(1)分析器进行了一组性能对比实验.测试用例采用 Java 语言.一方面,Bison 不允许分析表中存在冲突,而 Java 文法恰好可以改写为能被 Bison 接受的 LALR(1)文法;另一方面,Java 文法的规模适中,具备了命令式程序设计语言的各种典型语法特征.我们使用改写后的 Java 文法,分别利用 VPGE 和 Bison 各生成了一个 Java1.4 版的分析器.它们只执行语法分析,不调用任何用户自定义的例程.两者均由 C 语言实现,采用相同的词法分析器.机器 CPU 为 AMD2000,内存 512 兆,操作系统为 Windows/XP,所用 Bison 为 1.28 版.Sun 公司发布的 Java1.4 文法共 280 条 BNF 产生式,Bison 报告 235 个语法分析冲突,317 个 LALR(1)自动机状态.改写后的 Java 文法共 350 条 BNF 产生式,448 个 LALR(1)自动机状态,Bison 报告 0 个冲突.可见,改写后的文法尽管消除了语法分析冲突,但是文法规模和 LALR(1)自动机的状态数都有所增加.

实验获得的数据见表 3,其中时间的统计单位是 CPU 的时钟周期(clock)数,所统计的语法分析时间中包含了词法分析时间,扣除了构造分析森林的时间.从表 3 可见,两个分析器都具有线性或近似线性的时间复杂

度.VPGE 生成的 GLR 优化分析器比 Bison 生成的 LALR(1)分析器的分析速度平均约慢 2 倍~5 倍,随着被分析代码行和单词数的逐渐增大,两者之比逐渐趋于稳定.根据文献[3]给出的 ASF+SDF 元环境生成的标准 GLR 分析器与 Bison1.28 版生成的 LALR(1)分析器的时间性能对比实验结果,前者的分析速度约比后者慢 10 倍~100 倍.这说明,GLR 优化分析器的性能明显优于 ASF+SDF 元环境生成的标准 GLR 分析器.

Table 3 Comparison of performances of VPGE and Bison generated parsers

表 3 VPGE 与 Bison 生成的分析器的时间性能对比

No.	No. of lines	No. of tokens	VPGE (clock)	Bison (clock)	VPGE/Bison
1	43	103	309346	426407	0.725
2	321	1 155	4 847 486	1 631 176	2.972
3	218	1 310	5 698 296	1 668 174	3.416
4	528	2 376	9 815 220	2 772 309	3.540
5	1 285	5 410	24 398 177	6 847 486	3.563
6	2 088	11 288	62 278 403	9 101 114	6.843
7	9 916	53 495	303 525 993	58 468 247	5.191
8	16 007	88 943	461 722 165	85 804 274	5.381
9	32 928	128 397	969 437 820	165 329 119	5.864

6 相关研究工作

国内对 GLR 分析算法的研究较少,主要集中在自然语言处理领域.文献[14]利用 GLR 分析器的控制结构计算有关的概率信息,用来选择自然语言句子的最佳分析结果.文献[15]描述了如何基于 GLR 算法解决自然语言句法层次上的超语法现象.尚未见到 GLR 算法应用于程序设计语言语法分析的相关工作.而语法分析器自动生成器方面的研究工作,国内则完全处于空白状态.鉴于语法分析和语法分析器自动生成技术在计算机科学领域的重要性,本节介绍和比较这一领域内国外的重要研究成果,以飨读者.

Rekers JG 是使用 GLR 算法实现编程语言分析器生成系统(ASF+SDF 元环境)的第 1 人,在其博士论文[7]中给出了一个完整的 GLR 算法伪代码描述,纠正了早期 GLR 算法[1]存在的错误,使 GLR 分析算法的处理范围真正扩大到任意上下文无关文法[11].ASF+SDF 元环境(ASF+SDF meta environment)之所以被称为元环境,是因为它并非是应用某种特定计算机语言的软件开发环境,而是设计和实现计算机语言和语言原型的软件开发环境.ASF+SDF 元环境采用对文法没有限制的 GLR 分析算法,以使语言的设计者能够以更自然的结构定义语言的语法.它生成的 GLR 分析器为输入串构造分析森林,语法分析时不调用语义动作,运行速度较慢.

Bison 是自由软件基金会(GNU)开发的与 UNIX 系统工具 YACC 兼容的 LALR(1)分析器自动生成器.在经历了十几年的发展和演化后,已经成为 LR 分析器自动生成器事实上的工业标准.VPGE 的研制工作从 2002 年开始,到 2004 年初形成了第 1 个版本.我们还获悉,2004 年正式发布的 Bison 1.875 版[16]在原有设计框架之下也增加了对 GLR 分析算法的支持.经过仔细比较后发现,为了保持与以往版本的兼容,Bison 的 GLR 实现不构造分析树和分析森林,而是与以往的 LALR(1)版本一样,在分析过程中调用用户定义的语义动作.Bison 生成的 GLR 分析器存在语义动作延迟问题,并且将如何解决因语义动作延迟而产生的副作用留给用户自行处理.它要求用户自行判断哪些文法符号存在二义性,在编写文法输入规约时事先为二义性文法符号定义语义值合并例程.分析器在遇到全局冲突时调用这个例程以选择一个冲突分支,否则分析器便终止执行.事实上,文法符号的二义性在形式语言中是不可判定问题,如何判断哪些文法符号具有二义性,以及如何编写语义值合并例程完全需要凭借用户的经验和反复尝试.目前我们正在研读 Bison1.875 版的源代码,做进一步的比较和分析.

Aycock J 等人提出了一种构造 GLR 分析器的方法,可以明显改善 GLR 分析器的性能[17].不足之处在于,算法的性能没有与当今流行的分析器自动生成工具比较.此外,他们提出的算法对所能处理的文法施加了限制,不允许文法存在左递归和隐含的右递归.文献[3]报告了一个与 Bison 兼容的“高效、实用的”GLR 分析器生成器的性能评价结果,声称自动生成的 GLR 分析器在分析无二义性的输入串时分析速度只比 Bison 慢 15%.GLR 算法的时间复杂度在理论上高于线性的 LR 分析算法.另据我们的观察,Bison 生成的 LALR(1)分析器的源代码和所用的算法均采用了大量优化技术.为了加快分析速度,它的源代码甚至使用了非结构化程序设计和某些编译器优化选项,例如在代码中几乎无限制地使用 goto 语句,大量应用位运算技术.该文所述的工作与 VPGE 的研制工

作大致在同一时间开始,所用算法的实际性能还有待于进一步验证和比较.

绝大多数程序设计语言分析器自动生成系统基于 LALR(k)或 LL(k)确定性分析方法(k 代表分析过程中动态向前展望的符号数,典型的 k 取值为 1).ANTLR 和 BtYACC 是其中的例外.ANTLR[18](another tool for language recognition)是一个基于谓词扩展的 LL(k)自顶向下分析方法($0<k<\infty$)的语法分析程序生成系统,正在获得越来越多的实际应用.BtYACC[19](Backtracking YACC,带回溯的 YACC)是与 YACC 兼容的一个变体,借助于回溯机制处理输入串的局部二义性.它们采用的分析算法与 GLR 算法不属于同类算法,对所能接受的文法仍有一定的限制,不能处理任意上下文无关文法.有关回溯分析算法和 GLR 算法的异同以及谓词扩展 LL(k)分析算法的详细内容,可分别参考文献[11,19].

7 结 论

本文剖析了 GLR 分析算法,提出在程序设计语言分析器自动生成系统中正确、高效地实现 GLR 算法的实用途径.为了提高基本算法的性能,引入多层次的优化策略,加快了 GLR 基本算法的执行速度.随着计算机硬件处理速度和内存容量的飞速提高,算法的性能问题将得到进一步的缓解和改善.

本文还提出了一种语法分析时基于分析森林剪枝的交互式语义处理和二义性化解策略,用于 GLR 分析器的自动生成,避免了 GLR 分析器可能存在的语义动作延迟问题.

在不改变文法的可读性和易理解性的前提下,简单改写语言参考手册中发布的标准文法可以减少一部分语法分析冲突.难以解决的冲突可以适当保留.然后再根据分析表中是否存在冲突,选择使用 LR 分析引擎或 GLR 优化分析引擎.这是应用本文介绍的技术和工具构造程序设计语言分析器时我们所推荐的方法.

致谢 在此,向对本文的工作给予支持和建议的何新贵院士、刘超教授、高仲仪教授和审稿人表示感谢.

References:

[1] Tomita M. Efficient Parsing for Natural Language. Boston: Kluwer, 1985.

[2] Aycock JD. Practical Earley parsing and the SPARK toolkit [Ph.D. Thesis]. Victoria: University of Victoria, 2001.

[3] McPeak S, Necula G. Elkhound: A GLR parser generator. In: Proc. of the 13th Int'l Conf. on Compiler Construction. Barcelona: Spring-Verlag, 2004. 51−56.

[4] Gao ZY, Jin MZ. The Theory and Construction of Compilers. Beijing: Beijing University of Aeronautics and Astronautics Press, 1990. 100−127 (in Chinese).

[5] Johnson SC. YACC-Yet another compiler-compiler. Technical Report, Rep. CSTR 32, Murray Hill: Bell Laboratories, 1974.

[6] Donnelly C. Bison: The YACC-compatible parser generator. 1992. http://www.cs.utah.edu/dept/old/texinfo/bison/bison_toc.html

[7] Rekers JG. Parser generation for interactive environments [Ph.D. Thesis]. Amsterdam: University of Amsterdam, 1992.

[8] Irwin W, Churcher N. A generated parser of C++. 2001. http://citeseer.nj.nec.com/irwin01generated.html

[9] Corp R, Monica S. RACK: A parser generator for AI languages. In: Proc. of the 2nd Int'l IEEE Conf. on Tools for Artificial Intelligence. Herndon: IEEE Publisher, 1990. 430−435.

[10] Robert NM, Michael AA, Kfoury, AJ. An introduction to formal language theory. New York: Springer-Verlag, 1988. 61−75.

[11] Brand MVD, Sellink A, Verhoed C. Current parsing techniques in software renovation considered harmful. In: Proc. of the 6th Int'l Workshop on Program Comprehension. Ischia: IEEE Computer Society,1998. 108−117.

[12] Billot S, Lang B. The structure of shared forest in ambiguous parsing. In: Proc. of the 27th Annual Meeting of the Association for Computational Linguistics (ACL-89). 1989. 143−151.

[13] Parr TJ. Obtaining practical variants of LL(k) and LR(k) for $k>1$ by splitting the atomic k-tuple [Ph.D. Thesis]. West Lafayette: Purdue University, 1993.

[14] Zhu SH, Zhou M, Liu X, Huang CN. An efficient stochastic context-free parsing algorithm. Journal of Software, 1998,9(8):59−87 (in Chinese with English abstract).

[15] Weng FL, Zhou B, Wu LD. Process the phenomena of extra grammaticality in NL parsing. Journal of Chinese Information Processing, 1994,8(3):1−13 (in Chinese with English abstract).

[16] Donnelly C, Stallmen R. The Bison Manual: Using the YACC-Compatible Parser Generator, for Bison Version 1.875. Boston: GNU Press, 2004.

[17] Aycock J, Horspool N, Janousek J, Melichar B. Even faster generalized LR parsing. Acta Informatica, 2001,37(9):633−651.

[18] Parr TJ, Quong RW. ANTLR: A predicated-LL(*k*) parser generator. Software—Practice and Experience, 1995,25(7):789−810.

[19] Yacc BB. Siber System Corporation. 2004. http://www.siber.com//btyacc

附中文参考文献:

[4] 高仲仪,金茂忠.编译原理与编译程序构造.北京:北京航空航天大学出版社,1990.

[14] 朱胜火,周明,刘昕,黄昌宁.一种有效的概率上下文无关文法分析算法.软件学报,1998,9(8):59−87.

[15] 翁富良,周斌,吴立德.NL 句法分析中超语法符合现象的处理.中文信息学报,1994,8(3):1−13.

◇◇