第31卷 第15期 计 算 机 工 程 2005年8月
Vol.31 № 15 Computer Engineering August 2005
·基金项目论文· 文章编号：1000—3428(2005)15—0050—03 文献标识码：A 中图分类号：TP311.51

# LALR(1)解析器的再工程：YACC和CUP剖析

张 昱，张 磊
（中国科学技术大学计算机科学技术系，合肥 230027）

**摘 要**：现有的前端分析多数是通过编写相应的可能含有二义的LALR (1)文法规范，利用YACC或其变种自动生成的。在这些解析器的Java再工程中，可以用CUP去替代YACC，这样再工程的焦点转移到对文法规范的变换。由于YACC及CUP在二义的解决、解析器的构造等有细微的差别，因此对复杂的文法规范的变换并不是一件容易的事。该文通过剖析YACC和CUP，指出了它们的不同之处，并总结出有二义的YACC文法到CUP文法变换的基本原则和几个变换法则。

**关键词**：LALR(1)；解析器；YACC; CUP；二义性；冲突

# Reengineering LALR(1) Parsers: YACC and CUP Anatomizing

ZHANG Yu, ZHANG Lei
（Department of Computer Science and Technology, University of Science & Technology of China, Hefei 230027）

【**Abstract**】Most of the existing parsers are auto-generated by YACC or its variants via writing corresponding and possibly ambiguous LALR(1) syntax specifications. It is simple to reengineer these parsers to Java with CUP instead of YACC, which focuses all intentions on translating the syntax specifications. Since there are some subtle differences between YACC and CUP in recolving conflicts and constructing parsers, etc., it is not easy to translate a complex syntax specification.By means of anatomizing YACC and CUP, this paper indicates the discrepancies among them. Moreover, one basic fundamental and several transform principles are summarized, which can be used to translate any YACC syntax specification into CUP one.

【**Key words**】LALR(1); Parser; Yet another compiler-compiler (YACC); Constructor of useful parsers (CUP); Ambiguity; Conflict

随着Java跨平台优势的不断显露及JVM软硬件技术的不断发展，越来越多的人着手将程序设计语言的编译运行系统或领域应用系统移植到Java环境。这些系统中有一部分包含有用YACC[0]工具自动生成的C代码，对它们的Java再工程可以用类似YACC的能生成Java代码的工具，如CUP[0]；这样，移植的主要工作将转为对YACC输入文法文件(.y文件)的改写，该文法是LALR(1)的。

我们用上述方法完成了Perl解析器的移植，在移植中，发现单纯地按照CUP文法规范将perly.y变换成CUP输入文法文件perly.cup并不能保证二者生成的解析器对相同的Perl脚本产生相同的归约序列。究其原因，主要是：

(1)输入的文法本身具有二义性，需要YACC和CUP按默认的方法消除二义。总体上YACC和CUP都是分两步来消除文法二义的：先按优先级规则消除部分移进/归约冲突(sr冲突)；再按确定性规则解决余下的冲突，即对于sr冲突，优先移进；对于归约/归约冲突(rr冲突)，选用列在前面的文法规则进行归约。

(2) Perl的语法不能完全用LALR(1)文法表示，需要在词法分析器中增加特殊的符号识别处理并在文法中嵌入用来判断或设置词法状态的语义动作等来补足其文法。

(3) YACC和CUP生成的解析器的执行流程未必一致。

本文将结合Perl文法的移植讨论YACC和CUP生成的解析器不一致的根源以及如何消除不一致。

## 1 生成C代码的YACC及其变种

最初的YACC是由Bell实验室于20世纪70年代中期推出的，称为AT&T YACC。随后出现了许多变种，如Berkeley的YACC(简称BYACC)和GNU的Bison等。

三者的输入文法规范是一样的。一般来说，它们生成的解析器在功能上没有什么明显的差异。但是通过深入分析BYACC v1.9和Bison v1.35，它们在生成的LALR(1)分析表和冲突的处理上还是略有差别的：(1)BYACC比Bison多加了一个非终结符$accept和一个编号为0的规则“$accept → S”；(2)为使规则右端(RHS)内嵌的语义动作都转换到只出现在RHS的最右端，即用综合属性的计算模拟继承属性的计算，YACC会自动加入产生$\varepsilon$的标记非终结符，这些标记非终结符的前缀名在BYACC中是“$$”，而在Bison中则是“@”；(3)BYACC生成的解析器中状态0和1分别固定为起始状态和接受状态，而Bison生成的解析器中状态0为起始状态，接受状态编号则在最后；(4)当文法中存在移进动作和多个归约动作相互冲突时，BYACC和Bison均只报出这些冲突的一部分，二者在所报信息的选取上不完全一样。

## 2 YACC与CUP在输入文法规范上的差异

图1示意了.y和.cup文件的结构。其中，定义部分声明符号及其类型、终结符的优先级和结合性。规则由带语义动作(程序代码)的语法规则定义组成。默认的开始符是第1个规则的左部(LHS)。用户代码是任何合法的程序代码，在.cup

**基金项目**：国家自然科学基金资助项目(60173049)；Intel中国研究中心(ICRC)基金资助项目

**作者简介**：张 昱(1972—)，女，副教授，主研方向：程序设计语言理论和实现技术，XML数据管理，软件体系结构；张 磊，硕士生

**定稿日期**：2004-06-20 **E-mail**：yuzhang@ustc.edu.cn

中它们被包在{: :}块中，在.y 中则无须特别的括号，这些代码将直接被复制到最终的解析器代码中。

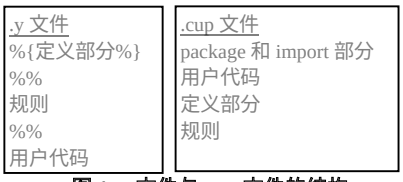


**图 1 .y 文件与.cup 文件的结构**

在.cup 中允许出现以下几种用户代码块：

（1) action code{: :}。 CUP 将在产生的 parser 文件中生成非 public 类 action 来包含该块中的代码。

（2) parser code{: :}。 CUP 将块中的内容加入到 parser 文件中的 parser 类中。

（3) init code{: :}。该块中的代码是在解析器请求第一个记号之前执行的。一般是用来初始化词法分析器以及语义动作所需的各种表及其它数据结构。

（4) scan code{: :}。该块中的代码指出解析器怎样取得下一个记号，它返回的值类型应该与词法分析器返回的一致。

在符号表示上，.y 文件允许在规则的 RHS 中直接出现终结符的串值，如‘{’；而在.cup 中则只能使用终结符的名字。.y 文件中的每个符号都有值，一般$$表示 LHS 的值，$1、$2 等自左至右依次表示 RHS 的符号值，YACC 默认地会将$1 的值传给$$。在.cup 中用 RESULT 引用 LHS 的值，用户需要为 RHS 中的符号引入标号来引用相应的值，如 exp:e1 中 e1 为符号 exp 的标号；如果希望 LHS 有值，必须在语义动作中显式地给出对 RESULT 的赋值。

## 3 perly.y 到 perly.cup 移植中存在的问题

按常规的变换方法将 perly.y 改写成 perly.cup 后，利用 CUP 工具就可以生成解析器 CParser。在用它解析 Perl 程序时，会发现一些与 YACC 解析器 YParser 解析不一致的地方。

两个 Parser 最大的不同是读下一记号的时机不同。CParser 总是在每次移进/归约前先读下一记号；而 YParser 在多数情况下先执行缺省归约，再读入下一记号，接着进行移进/归约处理。由此导致许多 Perl 程序用 YParser 解析正常，而用 CParser 却解析异常或失败。下面列举几个典型的例子。

**例 1** PL_expect 词法状态不一致问题。PL_expect 是存储当前解析所期待的下一记号类型的全局量，语义动作中多次出现对该变量的设置，如：

```
line : label ‘;’ { …   PL_expect = XSTATE; }          (规则 12)
```

CParser 会在读入 line 后的下一记号后按规则 12 归约并设置 PL_expect；而 YParser 则先按规则 12 归约并设置 PL_expect，然后再读入 line 后的记号。由于对 PL_expect 的设置在 CParser 中滞后，因此影响词法的正确分析，导致解析异常。

**例 2** package 的问题。package 结构的部分定义为

```
package : PACKAGE WORD ';'{package($2);}          (规则 62)
```

调用 package( )函数会现场保护当前符号表的位置及名称，将该新建包的符号表设为当前符号表。当 CParser 解析

```
package example;
sub show{}
```

时，会出现不可恢复的语法错误。因为它在读入“sub show”时，还没有对“package example;”进行归约，从而将“show”错认为是主程序中的子例程。

**例 3** block 的现场保护与恢复的时机问题。

```
block   :       '{' remember lineseq '}'
        { if (PL_copline > (line_t)$1) PL_copline = $1;
          $$ = block_end($2, $3); };                  (规则 3)
remember: { $$ = block_start(TRUE); };                (规则 4)
```

上面为块结构定义的相关规则，规则 4 是$\varepsilon$产生式，归约后执行 block_start( )完成原作用域的现场保护、新作用域及词法状态的设置等；按规则 3 归约后执行 block_end( )进行作用域的现场恢复。CParser 中块的现场保护和恢复始终是滞后的。这使得在按规则 3 或 4 归约时，已经读入的下一记号没有在正确作用域上被分析。

**例 4** 连续地定义和调用同一子例程问题。当在一个子例程的定义后紧接着调用它，CParser 会报在符号表中找不到该子例程的错误。如下面程序段：

```
sub try ($$) {; }
try 1,  13 %  4 ==  1;
```

这是因为 CParser 读入第 2 个“try”后才按

```
subrout:SUB startsub subname proto subbody
        { newSUB($2, $3, $4, $5); };                  (规则 53)
```

归约，执行 newSUB( )时才将 try 插入符号表。

CParser 和 YParser 的另一个主要的不同是二者对 sr 冲突的解决并不完全一致。如：

**例 5** sr 冲突的解决差异。CUP 和 YACC 在分析 Perl 文法后都指出在面临‘{’时，有 sr 冲突，涉及的规则是 114 和 123：

```
term    : scalar   %prec '(' { $$ = $1; }             (规则 114)
        | scalar '{' expr ';' '}'      %prec '(' {…}  (规则 123)
```

按确定性规则该冲突的解决是执行移进。在解析语句“$XXX{123} = 123; ”时，当$XXX 经多步归约成 scalar 后，YParser 会移进‘{’，最终按规则 123 将“$XXX{123}”归约成 term；而 CParser 则直接按规则 114 归约。这种差异是因为在面临该冲突时，YParser 是按确定性规则优先移进；CParser 则按优先级规则解决冲突，由于栈顶符号 scalar 的优先级与'(' 相同，是最高的，因此将栈顶归约成 term。

## 4 剖析 BYACC 与 CUP

与 BYACC 类似，CUP 也为输入文法添加一个非终结符 $start 和一个编号紧接在开始符规则之后的规则 $start→startsymbol。同样，CUP 也自动引入前缀名为“NT$”的标记非终结符将内嵌在规则 RHS 中的动作转换成仅出现在 RHS 的最右端。

CUP 用 lalr_state 类收集管理状态，BYACC 用 struct core 存储状态。二者信息基本一致，主要区别是 BYACC 只保存核心项目集，而非核心项目则在用到时再计算，这将大大节省空间开销；而 CUP 却记录了所有的核心项目和非核心项目。

CUP 中动作表和转移表由 parse_action_table 和 parse_reduce_table 类管理，内部使用二维数组存储表，它们与标准的 LALR 分析表结构 0 基本一致。BYacc 则按移进和归约分成 struct shifts 和 struct reductions 两个结构来管理，用链表实现动作表和转移表。通常动作表中大部分是出错处理，由于 BYACC 只记录了动作表中有移进、归约的部分，因此节省了一定的存储空间。CUP 和 BYACC 在对输入文法的处理流程上略有不同，表现在：

(1)CUP 用自身生成的 LALR 解析器来解析输入的文法文件；BYACC 则是用手工编写的解析器；

(2)CUP 先建立 lr0 状态表，再进行动作表的分析；BYACC 在建立 lr0 表的同时对动作表进行分析；

(3)二者建 lr0 状态表的过程基本相同，但是中间使用的数据结构不同，CUP 用栈而 BYACC 用链队列，这导致二者产生的状态的状态编号是一一对应的，但编号却不完全一致。

(4)CUP 和 BYACC 冲突解决的不一致主要集中在按优先级规则对 sr 冲突的解决上：在栈顶符号和 lookahead 中只有一个有优先级的情况下，BYACC 优先移进；而 CUP 则优先选择有优先级的一方。这解释了上节例 5 现象产生的原因。

从 YParser 和 CParser 的处理流程看，主要不同表现在缺省归约及其处理上。BYACC 生成的动作表中某些状态包含了一些缺省归约，当分析到这些状态时，解析器将不检查归约上的合法 lookahead，而在读下一记号前直接作缺省归约。这样，解析器可能会使得一些错误被延迟发现，但分析表却变得十分简单。通过分析 BYACC 源代码，某状态有缺省归约的条件是：(1)该状态中没有可执行的移进动作；(2)该状态的所有归约动作有相同的归约式。

在 CUP 中也有类似 BYACC 的缺省归约，通过在命令行里加-compact_red 选项可将最常用的归约式设成缺省归约式，旨在节省空间；但在生成的解析器代码中并没有对缺省归约的特别处理。

## 5 二义文法移植到 CUP 的附加处理

**基本原则** 将新增的语法规则放置在文法中原有语法规则之后。

遵循该基本原则可以避免因增添语法规则而导致原有语法规则的编号发生变化。

**变换法则 1** 对于形如 $A \to X_1\ X_2\ \cdots X_{n-1}\ X_n$ act 的规则，其中 $X_n$ 是终结符，act 是语义动作，若 act 中含有影响解析上下文的片段 seg，seg 独立于 act 中的其它代码且不含对 RHS 符号值的引用，则在将该规则移植到 CUP 时，可变换为如下两条规则：

(1) $A \to X_1\ X_2\ \cdots X_{n-1}\ N_{new}\ X_n\ act'$

(2) $N_{new} \to seg$

其中 $N_{new}$ 为新引入的非终结符，act'为 act 中除去 seg 的部分。

这里“影响解析上下文的片段”是指诸如对词法状态的修改、作用域的变换、包文件的装载解析等。例 1 中的问题可用该法则解决。

**变换法则 2** 对于形如 $A \to X_1\ X_2\ \cdots X_{n-1}\ X_n\ act$ 的规则，其中 $X_n$ 是终结符，*act* 是语义动作，若 *act* 影响解析上下文且不含有对 $X_n$ 值的引用，则在将该规则移植到 CUP 时，可以按以下两种方法之一提供的规则进行变换：

(1) $A \to X_1\ X_2\ \cdots X_{n-1}\ act\ X_n$

(2) $A \to A_{body}\ X_n$

$A_{body} \to X_1\ X_2\ \cdots X_{n-1}\ act$

其中方法(1)会影响文法中其后续规则的编号。上述的例 2 可以用该变换法则解决。

**变换法则 3** 对于形如 $A \to X_1\ X_2\ \cdots X_{n-1}\ X_n\ act$ 的规则，其中 $X_n$ 是非终结符，$X_n \Rightarrow^+ \alpha\ Y$ (“$\Rightarrow^+$”表示“一步或多步推导”，$\alpha$是符号串，$Y$ 为一个终结符)。若 *act* 影响解析上下文，且含有对 $X_n$ 值的引用，但 $X_n$ 值独立于 $Y$ 值，则在将该规则移植到 CUP 文法时，可以按如下规则变换：

$A \to A_{body}\ A_{lasttoken}$

$A_{body} \to X_1\ X_2\ \cdots X_{n-1}\ X_{nbody}\ act$

$X_{nbody} \Rightarrow^* \alpha$ (“$\Rightarrow^*$”表示“0 步或多步推导”)

$A_{lasttoken} \to Y$

例 4 即可用该法则处理。

**变换法则 4** 对于形如 $A \to X_1\ X_2\ \alpha$的产生式，其中 $X_1$ 是终结符，$X_2$ 是产生$\varepsilon$的非终结符，它附加的语义动作为 *act*，$\alpha$是任意非空符号串。若 *act* 中含有影响解析上下文的片段，而词法分析器在读入 $X_1$ 后的下一个记号时也可能会对该解析上下文进行修改，为保证 CParser 处理与 YParser 一致，可以在 CUP 文法中的 parser code 代码块中定义一个恢复词法分析器对该解析上下文设置的方法，并且在 *act* 的尾部追加对该方法的调用。

该法则可以解决例 3 中的问题。在 block_start( )中有对 PL_hints 全局量的设置，若当前被解析块在‘{’之后是 warn 或 die 时，词法分析器在识别出 warn 或 die 后也会执行对 PL_hints 的设置。为确保词法分析器对 PL_hints 的设置能保持到执行 block_start()之后，可以在 perly.cup 中的 parser code 代码块中增添 protected 方法 restoreHints()，它完成词法分析器中对 PL_hints 的设置；再在语义动作 block_start()后增加对 parser.restoreHints()的调用。

**变换法则 5** 假设文法中存在语法规则 $A \to \alpha\beta$ 和 $B \to \alpha$ %prec $t$ ，其中$\alpha$、$\beta$是任意非空的符号串，%prec $t$ 使得符号串$\alpha$具有与终结符 $t$ 相同的优先级。再假设项目 $A \to \alpha \cdot \beta$和 $B \to \alpha \cdot$ 属于项目集 $I_i$，它们在面临某 lookahead 时存在 sr 冲突，若该 lookahead 无优先级，为使在 CParser 中仍按移进解决此冲突，则可将 $B \to \alpha$ %prec $t$ 修改为

$B \to C$ %prec $t$

$C \to \alpha$

该法则用于解决 YACC 和 CUP 在处理 sr 冲突中的不一致。通过变换使原先 sr 冲突中的归约式 $B \to \alpha$ %prec $t$ 变为 $C \to \alpha$，此时 $C \to \alpha$没有优先级，因此将按确定性原则解决冲突，即优先移进。

## 6 结束语

本文的工作成果已经应用于“Perl 到 JVM 的移植”项目中，产生的 CParser 具有与原先 YParser 同样的解析功能和效果。由于 Perl 语言本身复杂性，使得本研究成果能适用于许多其它程序设计语言或领域规范语言解析器的 Java 再工程。

### 参考文献


1 Johnson S C. YACC—Yet Another Compiler-compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975

2 Hudson S E. CUP Parser Generator for Java (v0.10k). http://www. cs. princeton.edu/~appel/modern/java/CUP/, 1999-09

3 Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools. Addison Wesley, 1986