LALR(1)解析器的再工程：剖析 YACC 和 CUP

张 昱　张 磊[1]

（中国科学技术大学 计算机科学技术系， 合肥 230027）

摘要　程序设计语言或领域规范语言在移植到 Java 环境时，需要考虑其前端分析器的构造。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，使得对复杂的文法规范的变换并不是一件容易的事。本文通过剖析 Yacc(BYacc 和 Bison)和 CUP，指出它们的不同之处，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

分类号　TP311.51, TP311.54, TP314

Reengineering LALR(1) Parsers: Anatomizing YACC and CUP

ZHANG Yu　ZHANG Lei　CHEN Yi-yun

（Department of Computer Science, University of Science & Technology of China, Hefei, 230027）

Abstract　When porting programming languages or domain specification languages to Java environment, their front-end parsers have to be reconstructed. Most of 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 focus all intention 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. In this paper, by means of anatomizing Yacc(including Berkeley Yacc and Bison) and CUP, the discrepancies among them are indicated. 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; Yacc(yet another compiler-compiler); CUP(constructor of useful parsers); ambiguity; conflict

随着Java跨平台优势的不断显露以及JVM软硬件技术的不断发展，越来越多研究开发人员着手将一些程序设计语言移植到Java环境[1]从而提供更多适合在Java环境中开发的语言；而领域应用系统也在逐步展开向Java环境的过渡。在这些系统中，一部分涉及语言前端分析的代码是利用Yacc[2](此处泛指最初Bell实验室推出的Yacc以及Berkeley Yacc[3]和Bison[4]等生成C代码解析器(parser)的Yacc的变种)自动生成的。这些解析器代码到Java移植的一种简单做法是使用类似的能生成Java源码的解析器的工具。由于Yacc是基于LALR(1)文法的，目前常见的生成Java源码的LALR(1)解析器的工具有BYacc/Java[5]、CUP(Constructor of Useful Parsers)[6]等。BYacc/Java是在BYacc v1.8 基础上修改的，增加了对输出Java源码解析器的支持；而CUP是用Java实现的。文献[7]系统地说明了Yacc家族在软件构造和再工程中的意义和方法。

在我们承担的Intel中国研究中心的项目——将SPEC CINT2000[8]中的 253.perlbmk移植到JVM中，使用CUP完成Perl解析器的构造。我们发现单纯地将Perl原先的Yacc文法文件perly.y按照CUP文法规范的要求变换成perly.cup，并不能保证CUP和Yacc生成的两个解析器对相同Perl程序段的解析产生相同的归约序列。也就是说，两个解析器的解析过程并不完全吻合。究其原因，主要是：1) 输入的文法本身具有二义性，需

[1]**张昱**(1972-)，女，副教授，主要研究领域为程序设计语言理论和实现技术、XML数据管理、软件体系结构，E-mail：yuzhang@ustc.edu.cn。**张磊**(1981-)，男，硕士研究生，主要研究领域为程序设计语言理论和实现技术。本项目受Intel中国研究中心(ICRC)基金的资助，项目名称为“Porting Perl to Java VM”和国家自然科学基金项目(编号 60173049)的资助。

要由Yacc和CUP按自己默认的方法消除二义；2) Perl的语法是上下文相关的，不能完全用LALR(1)文法表示，它通过在词法解析器中增加特殊的符号识别处理以及在文法的语义动作中加入对词法状态的判断和设置等方法，来补足其语法规则；3) Yacc和CUP生成的解析器的执行流程未必一致。

虽然总体上 Yacc 和 CUP 在消除文法二义上都是分两步进行：1) 按**优先级规则**消除部分移进/归约冲突(shift/reduce conflicts，简称 sr 冲突)；2) 再按**确定性规则**解决余下的冲突，即 a)对于 sr 冲突，优先移进，b) 对于归约/归约冲突(reduce/reduce conflicts，简称 rr 冲突)，选用列在文法最前面的语法规则进行归约。但是在实际操作 Perl 文法时，却有着不完全一致的冲突解决结果。于是我们不得不深入分析 Yacc 和 CUP 的源代码，探究这种不一致的根源以及如何修改文法消除这种不一致。我们同时也对常用的 BYacc 和 Bison 进行了对比分析。本文即是针对这些分析工作的总结。

1 AT&T Yacc、BYacc 和 Bison

最初的Yacc是由Bell实验室于上世纪 70 年代中期推出的，我们称其为AT&T Yacc。随后出现了许多变种，如Berkeley的Yacc(简称BYacc)和GNU的Bison。comp.compilers新闻组中的讨论认为这三者之间的差异主要表现在政治上，而不是功能上[9]。AT&T Yacc属于那些拥有Bell实验室Unix源码的人，它迄今仍有所有权归属问题。BYacc在用户遵循Berkeley的“*just don't sue us*”的许可下可以被自由地使用。而Bison源自BYacc，由GNU开发和维护，它增加了许多额外的特征。

```
%%
S: {}A ;
A: 'a' ;
%%
```

图 1 simple.y
Fig.1 simple.y

三者输入的文法格式是一样的。在文献[10]中详述了 LALR(1)分析算法并简介了 Yacc 工具。一般来说，它们生成的解析器在功能上看不出什么明显的差异。但是通过深入分析 BYacc v1.8 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。

首先以图 1 的 simple.y 作为 BYacc 和 Bison 的输入，则生成的两个解析器的主要信息对比如表 1。

表 1　BYacc 与 Bison 生成的解析器对比
Table 1 Comparison between the two parsers generated by BYacc and Bison

	BYacc 产生的解析器	**Bison 产生的解析器**
终结符	3 个：$end(0), 'a'(97), error(256)	3 个：$ (-1), 'a' (97), error (256)
非终结符	4 个：$accept, S, $$1, A	3 个：S, @1, A
规则	(0) $accept → S (1) $$1 → (2) S → $$1 A (3) A → 'a'	 (1) @1 → (2) S → @1 A (3) A → 'a'

状态
（括号内的编号为对应的规则编号，粗体字部分为该状态的核心项目集，非粗体字部分反映该状态将迁移到的状态）
	state 0： $accept : . S $end (0)
	$$1 : . (1)
	. reduce 1
	S goto 1
	$$1 goto 2
	state 1： $accept : S . $end (0)
	$end accept
	state 2： S : $$1 . A (2)
	'a' shift 3
	. error
	A goto 4
	state 3： A : 'a' . (3)
	. reduce 3
	state 4： S : $$1 A . (2)
	. reduce 2
		state 0：
		$default reduce using rule 1 (@1)
		S go to state 4
		@1 go to state 1
		state 1： S : @1 . A (2)
		'a' shift, and go to state 2
		A go to state 3
		state 2： A : 'a' . (3)
		$default reduce using rule 3 (A)
		state 3： S : @1 A . (2)
		$default reduce using rule 2 (S)
		state 4：
		$ go to state 5
		state 5：
		$ go to state 6
		state 6：
		$default accept

从表中可以看出：①BYacc 增加了一个非终结符$accept 和一个规则“$accept → S”；②两边对应的规则编号是一致的，BYacc 中新增的规则编号为 0；③为使 RHS(right-hand-side)中嵌入的动作都变换成只出现在 RHS 的右端(即用综合属性的计算模拟继承属性的计算)，Yacc 会加入产生ε的标记非终结符，使得每个嵌入动作由不同的标记非终结符代替，这些标记非终结符的名字在 BYacc 中以“$$”引导，而在 Bison 中则以“@”引导；④BYacc 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

析器的状态 0 为起始状态，接受状态编号则在最后；⑤由于 BYacc 将接受状态插在状态 1，因此总体上中间状态的编号在 BYacc 中要比 Bison 多 1，不过这也不是完全绝对的(原因见 3.2 节的 3))。

我们接着考察两工具对二义文法的处理差异，以 253.perlbmk 中的 perly.y 进行实验。BYacc 执行结果是“113 shift/reduce conflicts, 1 reduce/reduce conflict”，而 Bison 执行结果为“112 shift/reduce conflicts, 2 reduce/reduce conflicts”。对比分析所产生的冲突细节，只有一处不一样(图 2)。这种不一致出现的场合是很特殊的，它往往出现在某状态面临某输入时存在 3 种或 3 种以上可能的动作。以图 2 的 BYacc 处理为例，在状态 111 处，当面临‘[’时可能有三种动作：shift 170、按 114 规则归约和按 174 规则归约。这三个动作将引发三对冲突(两两组合)，但 BYacc 和 Bison 只从中选择两个：BYacc 认为是两个 sr 冲突，而 Bison 则认为是一个 sr 冲突和一个 rr 冲突。显然，这三个冲突最终是按执行移进来解决的。我们认为 BYacc 的这种选择要合理一些，毕竟两个归约动作最终都不会执行，因此根本不会有 rr 冲突。

BYacc的输出

```
State 111 contains 3 shift/reduce conflicts.
111: shift/reduce conflict (shift 169, reduce 114) on '{' resolved as shift 169
111: shift/reduce conflict (shift 170, reduce 114) on '[' resolved as shift 170
111: shift/reduce conflict (shift 170, reduce 174) on '[' resolved as shift 170
core of state 111 :
        term : scalar .    (114)
        term : scalar . '[' expr ']'    (117)
        term : scalar . '{' expr ';' '}'    (123)
        indirob : scalar .    (174)
```

Bison的输出

```
State 112 contains 2 shift/reduce conflicts and 1 reduce/reduce conflict.
112: shift/reduce conflict(shift 171,reduce 114) on '{' resolved as shift 171
112: shift/reduce conflict(shift 172,reduce 114) on '[' resolved as shift 172
112: reduce/reduce conflict(reduce 114,reduce 174) on '[' resolved as reduce 114
core of state 112 :
      term   :   scalar .     (114)
      term   :   scalar . '[' expr ']'     (117)
      term   :   scalar . '{' expr ';' '}'     (123)
      indirob   :   scalar .     (174)
```

图 2　BYacc 和 Bison 对 perly.y 冲突处理上的不同

Fig.2 Difference between BYacc and Bison on resolving conflicts in perly.y

2　perly.y 改写成 perly.cup 中存在的问题

253.perlbmk 是利用 BYacc v1.8 来分析 perly.y 生成解析器代码 perly.c 的。我们选择 CUP 实现 Perl 解析器到 Java 的移植，因此首先需要将 perly.y 改写成 perly.cup。

2.1　.y 文件与.cup 文件的不同

.y文件

```
%{定义部分%}
%%
规则
%%
用户代码
```

.cup文件

```
package 和 import 部分
用户代码
定义部分
规则
```

图 3　.y 文件与.cup 文件的结构

Fig.3 Structures of .y and .cup file

1) 文件结构

图 3 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**包括语法规则定义以及归约后要执行的语义动作(程序代码)。默认的开始符是第一个规则的 LHS。**用户代码**是任何合法的程序代码，在.cup 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。

在.cup 中允许出现以下几种用户代码块：

1) action code{: :}　　系统将在 parser 文件中生成单独的、非 public 的类 action 来包含该块中的代码。
2) parser code{: :}　　与 action code 非常类似，不过{: :}中的内容将被纳入 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。如果词法分析器返回的是一个 Symbol，则 scan with 中的代码返回的也应该是一个 Symbol。

2) 符号

在.y 文件中允许 RHS 中的终结符直接用其对应的串表示，如‘{’；而在.cup 中则不允许，必须为每一个终结符引入一个名字，在 RHS 中使用这些名字代表相应的终结符。

.y 文件中的每个符号都有值，符号的类型可以不一样。可以在语义动作中用$$引用 LHS，用$1、$2 等自左至右依次引用 RHS 中的符号。在缺省情况下，BYacc 会把$1 的值传给$$。

在.cup 中用 RESULT 引用 LHS，用户可以为 RHS 中的符号引入标号，如 exp:e1 表示为符号 exp 引入 e1 标号，这样就可以通过标号来引用相应的 RHS 符号。需要指出的是，如果希望 LHS 有值，必须在语义动作中显式地将 RHS 符号的值按所需的要求计算并赋值给 RESULT。

3) 运行

运行 bison –d –y perly.y 或 byacc –d perly.y，将默认地输出 y.tab.c 和 y.tab.h，y.tab.c 中的 yyparse( )负责扫描输入、移进、归约；运行 java java_cup.Main < perly.y 将默认地输出 parser.java 和 sym.java，parser 对象中的 parse( )方法负责完成对输入串的扫描、移进和归约。

4) 一个简单的例子

图 4 为一个简单的表达式解析器的.y 和.cup 文件定义，其中词法分析部分分别使用 Flex 和 JFlex 工具。从该图中，可以看到更多的关于.y 文件和.cup 文件的规范差异。

表达式的词法描述—flex文件

```
%{ #include “y.tab.h” %}
%%
[0-9]+ {    yylval.ival = atoi(yytext);
            return NUMBER; }
[ \t]    /* ignore whitespace */
\n       return 0; /* logical EOF */
“+”     return PLUS;
“-”      return MINUS;
…
%%
```

表达式的词法描述—jflex文件

```
import java_cup.runtime.Symbol;
%%
%cup
%%
[0-9]+ { return new Symbol
   ( sym.NUMBER, new Integer
   (yytext())); }
[ \t] { /* ignore whitespace */ }
\n { return new Symbol
   (sym.EOF); /* logical EOF */ }
“+” { return new Symbol
   (sym.PLUS); }
“-” { return new Symbol
   (sym.MINUS); }
…
```

表达式的语法描述—.y文件

```
%union {
   int ival;
   char *sval;
}
%token PLUS MINUS EQUALS
%token <sval> NAME
%token <ival> NUMBER
%type <ival> expression
%%
statement : NAME EQUALS expression { }
          | expression { printf (“ = %d\n”, $1); }
          ;
expression : expression PLUS NUMBER
             { $$ = $1 + $3; }
          | expression MINUS NUMBER
             { $$ = $1 - $3; }
          | NUMBER { $$ = $1; }
          ;
extern FILE *yyin;
int yyerror (char *s)
{
   fprintf (stderr, “%s\n”, s);
}
int main ()
{
   if (yyin == NULL) yyin = stdin;
   while (!feof(yyin)) yyparse();
}
```

表达式的语法描述—.cup文件

```
parser code
{:
   public static void main (String argv[]) throws
   Exception
   {
      new parser(new Yylex(System.in)).parse();
   }
:}
terminal PLUS, MINUS, EQUALS;
terminal String NAME;
terminal Integer NUMBER;
non terminal statement;
non terminal Integer expression;
statement ::= NAME EQUALS expression:e
              | expression:e
              {: System.out.println(“ = ” + e); :}
              ;
expression ::= expression:e PLUS NUMBER:n
              {: RESULT = new Integer(e.intValue()
                 + n.intValue()); :}
              | expression:e MINUS NUMBER:n
              {: RESULT = new Integer(e.intValue()
                 - n.intValue()); :}
              | NUMBER:n
              {: RESULT = n; :}
              ;
```

图 4　一个例子：表达式分析器的文法定义

Fig.4 An example: specifications of an expression parser

2.2　Perl 文法移植中的问题

为简便起见，以下称 CUP 生成的解析器为 CUP Parser，BYacc 生成的解析器为 Yacc Parser。按常规的变换方法将 perly.y 改写成 perly.cup 后，利用 CUP 工具，就可以得到 CUP Parser。在利用它解析 Perl 程序时，发现许多与 Yacc Parser 解析不一致的地方。

两个 Parser 最大的不同是读下一个记号(token)的时机不一致。CUP Parser 总是在移进/归约前先读下一个记号；而 Yacc Parser 在大多数情况下先执行缺省归约，然后再读入下一记号，接着进行移进/归约处理。由于这种不一致，使得相当多的 Perl 程序用 Yacc Parser 解析正常，而用 CUP Parser 却解析异常或失败。比如以下 4 个典型的例子。

例 1．PL_expect 词法状态的不一致问题

Perl 编译器中用 PL_expect 指示在当前上下文中期待的下一记号的类型。在 Perl 文法的语义动作中多

次出现对 PL_expect 值的设置，如：

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

CUP Parser 会在读入 line 后的下一记号后按规则 12 归约；而 Yacc Parser 则先按规则 12 归约，设置 PL_expect，然后再读入 line 后的记号。由于 PL_expect 在 CUP Parser 中设置滞后，故影响词法的正确分析，导致解析异常。

```
package :   PACKAGE WORD ';'
            { package($2); }                    （规则 62）
        |   PACKAGE ';'
            { package(Nullop); }                （规则 63）
        ;
use     :   USE startsub
            { CvSPECIAL_on(PL_compcv);
              /* It's a BEGIN {} */ }           （规则 64, ε产生式）
            WORD WORD listexpr ';'
            { utilize($1, $2, $4, $5, $6); }    （规则 65）
        ;
```

图 5 **use** 与 **package** 的语法规则

Fig.5 Grammar rules of **use** and **package**

例 2．package 和 use 的问题

图 5 为 use 与 package 的语法定义。其中 package( )会将当前符号表的位置及名称进行现场保护，然后使得当前新建的包对应的符号表成为当前符号表，即进入到包的作用域。当 CUP Parser 解析如下 Perl 程序时：

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

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，因此还没有进入到包 example 的作用域，从而将“show”错认为是主程序中的子例程。

对 use 的使用也存在类似的问题，因为执行 utilize( )时会查找、装载、解析 use 指定的包文件。

```
block   :   '{' remember lineseq '}'
            { if (PL_copline > (line_t)$1)
                  PL_copline = $1;
              $$ = block_end($2, $3); }     (规则 3)
        ;
remember:   /* NULL */   /* start a full lexical scope */
            { $$ = block_start(TRUE); }     (规则 4)
        ;
```

图 6 **block** 的语法规则

Fig.6 Grammar rules of **block**

例 3　block/mblock 的现场保护与恢复的时机问题

mblock 的语法结构与 block 类似，这里只以 block 结构(图 6)为例来说明。规则 4 为ε 产生式，其归约后执行 block_start( )完成块的启动，包括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。对于 CUP Parser，块的现场保护和现场恢复始终是滞后的。这使得在按规则 3 或 4 归约时，已经读入的下一个记号不是在正确的作用域环境上进行分析的。

例 4　连续地定义和调用同一子例程问题

当用 sub 定义一个子例程后，接着调用该子例程，在 CUP Parser 中解析会出错（在符号表中找不到该子例程）。例如：

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

这是因为 CUP Parser 读入第二个“try”后才按

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

归约，其语义动作 newSUB( )会将子例程 try 插入符号表，这就造成了读第二个“try”时在符号表中找不到相应条目，从而使得分析出错。

CUP Parser 和 Yacc Parser 的另一个主要的不同是两者对 sr 冲突的解决并不完全一致。如：

```
term    :   scalar  %prec '('
            { $$ = $1; }                        (规则 114)
        |   scalar '{' expr ';' '}'   %prec '('
            { $$ = newBINOP(OP_HELEM, 0,
                  oopsHV($1), jmaybe($3));
              PL_expect = XOPERATOR; } (规则 123)
        ;
```

图 7 **term** 的两个语法规则

Fig.7 Two grammar rules of **term**

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(图 7)，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

[*]这里的规则均使用perly.y中的表示形式。

步归约成 scalar 后，Yacc Parser 会移进‘{’，最终按规则 123 将“$XXX{123}”归约成 term；而 CUP Parser 则按规则 114 将 scalar 归约成 term，从而导致后面的解析不能正常完成。造成这种差异的原因是：在面临这种冲突时，Yacc Parser 的优先级规则对此冲突没有起作用，于是按确定性规则优先移进；而 CUP Parser 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，所以将栈顶句柄归约成 term。

3 剖析 BYacc 与 CUP

本节深入分析 BYacc 与 CUP 源码，总结它们的不同之处，解释节 2 中提到的差异现象产生的根源。由于 BYacc 是用 C 实现的，而 CUP 是用 Java 实现的，因编程语言的不同使得两者必然有很多的不同，本文不强调这种不同。

3.1 主要数据结构对比

与 BYacc 类似，CUP 也为输入的文法自动引入一个非终结符$start 和一个规则$start→startsymbol，只是该规则的编号紧接在开始符规则对应的编号之后。同样，CUP 也会自动引入标记非终结符将嵌入的动作全部转换成出现在 RHS 的右端，这些标记非终结符的名字是以“NT$”引导的。

1）状态的表示

CUP 中用 lalr_state 类收集管理一个状态的信息，包括：_index(状态号)、_items(状态的项目集)、_transitions(可能进入的下一状态)、_next_index(下一状态的个数)、_all(所有状态的 hash 表)、_all_kernels(所有核心项目集的 hash 表)等。BYacc 中用 struct core 表示状态，其中有域：number(状态号)、next(下一个状态)、link(当前状态可能转向的状态)、accessing_symbol(访问此状态的符号)、nitems(状态中包含的核心规则数)、items[](核心项目集)。

两者的信息基本一致，主要的区别是 BYacc 只保存核心项目集，而非核心项目则在用到时由计算而得，在文法较复杂时这将会大大地节省空间开销；而 CUP 在_items 中记录了所有的核心项目和非核心项目。

2）LALR 分析表的表示

CUP分别用parse_action_table类和parse_reduce_table类表示动作表和转移表，两者的核心属性都是数组，下标即对应状态号。parse_action_table中数组的每一项包含针对每一终结符的动作，parse_reduce_table中数组的每一项包含针对每一非终结符的下一状态。它们建成的表与标准的LALR分析表[10]基本一致。

BYacc 没有将终结符和非终结符分开处理，而是按移进和归约分成 struct shifts 和 struct reductions 两个结构。

CUP 采用二维数组存储动作表和转移表，BYacc 则采用链表存储。通常情况下，动作表的大部分动作是出错处理，由于 BYacc 只记录了表中有移进、归约动作的部分，所以节省了一定的存储空间。

3.2 主要处理流程对比

表 2 CUP 和 BYacc 的主要处理流程

Table2 Main flow of CUP and BYacc

BYacc	CUP
读入.y 文件,建立符号表和规则表：set_signals( ); getargs( ); open_files( ); reader( );	读入.cup 文件,建立符号表和规则表：parse_grammar_spec( )
根据规则建立 lr0 状态表：lr0( ) 根据 lr0 状态表建立 lalr 表：lalr( ) 分析解决冲突：make_parser( )	建立 lalr 表：build_parser( ) 1) 初始化 2) 建立 lr0 状态表：lalr_state.build_machine( ) 3) 建立表,分析解决冲突：lalr_state.build_table_entries( )
输出文件(verbose( )输出 xxx.output 文件，output( )输出 xxx.c 文件)	输出 xxx.java 文件：emit_parser( )

表 2 给出了 CUP 和 BYacc 的主要处理流程。整体上看，两者的流程基本上是一致的，但是局部处理稍有不同，表现在：

1) 对输入文件的解析方法不同：CUP 用自身生成的 LALR 解析器来解析，见 parser.java；BYacc 则用手

工编写的解析器，见 reader( );

2) CUP 是先建立 lr0 状态表，再进行动作表的分析，见 build_parser( )；BYacc 是在建立 lr0 表的同时对动作表进行了初步地分析，见 generate_states( )。

3) 两者建 lr0 状态表的过程基本相同：a) 确定第一个项目(也是核心项); b) 根据项目集建立状态(BYacc 要先计算闭包); c) 计算下一个可能状态的项目集，重复这一过程可得所有的状态。

   在 c)步中需要记录没有算过后继状态的状态，CUP 和 BYacc 采用了不同的数据结构，CUP 用 Stack work_stack 存储，BYacc 用队列链表 this_state 存储，这导致两者产生的状态的状态编号是一一对应的，但却不完全相同。

4) CUP 和 BYacc 冲突解决的实现细节有所不同，主要表现在用优先级规则对 sr 冲突的解决上：

   a) BYacc(见 remove_conflicts( ))：如果 lookahead 记号和栈顶符号都有优先级，选择优先级高的；如果优先级相同，则看结合性：右结合选择移进，左结合选择归约，无结合两者皆不选。如果两者不全有优先级，则优先移进(与确定性规则处理 sr 冲突一致)。

   b) CUP(见 fix_with_precedence( ))：如果栈顶符号有优先级，则比较它和 lookahead 的优先级，大于则移进，小于则归约，等于再看结合性：左结合则归约，右结合则移进，无结合则作特殊标记。如果栈顶符号没有优先级，lookahead 有优先级，则移进。如果两者皆没有优先级，则用确定性规则解决。

   由此，总结 BYacc 和 CUP 在按优先级规则解决 sr 冲突的不同点：**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 2.2 节中例 5 问题产生的原因。

3.3 输出文件的对比

表 3 列出 CUP 和 BYacc 输出文件中的文法信息记录。表 4 说明了它们在循环解析中的解析流程。

表 3 CUP 和 BYacc 输出文件中的文法信息记录

Table3 Grammar info registered in CUP and BYacc output files

BYacc		CUP
yylhs[nvars]:	记录每个产生式的 LHS	symbol符号类（属性包括：符号的id、最左字符和最右字符的位置、符号的值、相关的状态值）
yylen[nvars]:	记录每个产生式的 RHS 的符号个数	stack:栈（栈中元素为 symbol 类）
yydefred[nstates]:	记录每个状态的缺省归约式(0 表示无)	_production_table[][]: 产生式表，共 2 列。行下标表示规则编号，第 0 列为规则的 LHS 标号，第 1 列为 RHS 的符号数.
yydgoto[nvars]:	记录每个非终结符的缺省转移	_action_table[][]: 动作表,每一行对应一个状态,由<符号,下一个动作>组成,按符号标号的升序排列.下一个动作是移进(用正数表示),归约(用负数表示)或错误(用零表示)
yysindex[nstates]:	移进动作的索引	
yyrindex[nstates]:	归约动作的索引	
yygindex[nvars]:	转移动作的索引	
yytable[]:	记录上述所有动作	
yycheck[]:	记录对应上述动作的符号	_reduce_table[][]: 转移表,每一行对应一个状态,由<符号,下一个状态>组成
yyss[]:	状态栈	上述 3 个表是压缩成 String[ ]类型存储的，emit 类中的 do_table_as_string()方法和 lr_parser 类中的 unpackFromStrings()方法分别完成了压缩和解压的操作
yyvs[]:	数据栈	
yyssp:	状态栈的栈顶指针	
yyvsp:	数据栈的栈顶指针	
yylval:	当前记号的值	
yyval:	用于暂时保存归约后 LHS 的值	

表 4 CUP 和 BYacc 输出文件中的解析流程

Table4 Parsing flow in CUP and BYacc output files

BYacc	CUP
i) 如果栈顶状态有缺省归约，则作归约； ii) 如果下一个记号为空，则读入下一个记号；若无下一记号，检查栈中信息决定是接受还是出错，并退出； iii) 根据下一个记号和栈顶状态查表，看是否可以移进，可以则作移进，并将下一个记号设为空，转向 i) iv) 根据下一个记号和栈顶状态查表，看是否可以归约,可以则作归约，转向 i) v) 错误处理，若是异常则恢复并转向 i)，否则解析失败并退出	i) 初始化 ii) 读入下一个记号，若无下一记号，检查栈中信息决定是接受还是出错，并退出； iii) 根据下一个记号和栈顶状态查表,作移进、归约或出错处理 iv) 如果分析未结束,继续执行 ii)

BYacc Parser 与 CUP Parser 的主要不同体现在缺省归约及其处理上。

BYacc 生成的动作表与标准的 LALR 动作表略有不同，其中某些状态包含了一些缺省归约的动作(如表 1 状态栏中的“.reduce 1”等)。当分析到此状态时，解析器将不试着去检查归约上的合法 lookahead，在读入下一个记号之前作缺省归约。解析器可能会在最终声明错误之前执行多个归约，这样出错信息失去了其信息价值，但分析表却变得十分简单。通过分析 BYacc 的源代码，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；和②该状态的所有归约动作有相同的归约式。

在CUP中也可以有类似BYacc中的缺省归约，通过在命令行里加-compact_red选项，它将最常用的归约式设成缺省归约式，其目的是节省空间。同样它也是把错误动作作为缺省的归约动作，使得错误被延迟发现。但在生成的解析器代码中并没有对缺省归约的特别处理，所以这个选项仅仅是为了节省存储空间。

4 二义文法移植到 CUP 的解决方案

鉴于 Yacc 与 CUP 之间的区别，如何能将一个二义的.y 文法顺利地移植成.cup 文法，使得基于它们生成的 Yacc Parser 与 CUP Parser 在解析时具有相同的解析效果？本节总结我们在实践中被验证是有效的、有代表性的变换法则。

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，从而便于 CUP Parser 与 Yacc Parser 之间的对应调试。

变换法则 1：对于形如$A\rightarrow X_1\ X_2\ \ldots X_{n-1}\ X_n\ act$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文法时，可以变换为如下两条规则：

$$A\rightarrow X_1\ X_2\ \ldots X_{n-1}\ N_{new}\ X_n\ act'$$

$$N_{new}\rightarrow seg$$

其中N_{new}为新引入的非终结符，act'为act中除去seg的部分。

```
line ::=        label:la expectstat SEMICOLONFLAG
                {:……PL_expect = XSTATE;   :}     (规则 12)
        | ……
        ;
expectstat ::=  {: PL_expect = XSTATE;   :};     (规则 179)
```

图 8　PL_expect 问题的解决

Fig.8 Resolution of PL_expect issue

这里对解析上下文的影响包括 2.2 节中提到的对词法状态的修改、作用域的变换、包文件的转载解析等等。

2.2 节的例 1 就属于这种情况，我们引入非终结符 *expectstat*，在将ε归约成 *expectstat* 后，会将 PL_expect 设置成 XSTATE，同时对规则 12 进行修改(图 8)。

变换法则 2：对于形如$A\rightarrow X_1\ X_2\ \ldots X_{n-1}\ X_n\ act$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文法时，可以按以下两种方法之一提供的规则进行变换：

方法 1　$A\rightarrow X_1\ X_2\ \ldots X_{n-1}\ act\ X_n$

方法 2　$A\rightarrow A_{body}\ X_n$

$$A_{body}\rightarrow X_1\ X_2\ \ldots X_{n-1}\ act$$

对于方法 1 中的规则，不论是 Yacc 还是 CUP 都会自动地在该规则前增加一条ε产生式 $M\rightarrow act$，其中 M 为引入的标记非终结符。这样就使得原先的一条规则转换成连续编号的两条规则，从而使得后续规则的编号均要加 1。这不符合前面提出的基本原则。对于方法 2，可以让其中的第 1 条规则占据原规则在文法中的位置，而将第 2 条规则加到文法的最后。这样，不会影响文法中其它规则的编号，有利于解析器的对比调试。

```
变换方法一:
packages ::= PACKAGE WORD:wd
             {:Other.package(wd); :}     (规则 62,ε产生式)
             SEMICOLONFLAG               (规则 63)
       |     PACKAGE
             {:Other.package(null);:}    (规则 62,ε产生式)
             SEMICOLONFLAG               (规则 65)
       ;
变换方法二:
packages::=  pkgbody SEMICOLONFLAG;      (规则 62)
pkgbody ::=  PACKAGE WORD:wd
             {: Other.package(wd);:}     (规则 181)
       |     PACKAGE
             {: Other.package(null); :}  (规则 182)
       ;
```

图 9　**package** 问题的解决

Fig.9 Resolution of **package** issue

2.2 节的例 2 就属于变换法则 2 所说的情况。图 9 示意了

package 语法规则在两种方法下的变换结果。例 3 中 block 的现场恢复滞后也可以通过此变换法则解决。

变换法则 3：对于形如$A \rightarrow X_1\ X_2\ \ldots X_{n-1}\ X_n\ act$的规则，其中$X_n$是非终结符，$X_n \Rightarrow^+ \alpha\ Y$ (符号“$\Rightarrow^+$”表示“一步或多步推导”，α是符号串，Y表示一个终结符，它可以是若干终结符的或)。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：

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

$A_{body} \rightarrow X_1\ X_2\ \ldots X_{n-1}\ X_{nbody}\ act$

$X_{nbody} \Rightarrow^* \alpha$　(符号“$\Rightarrow^*$”表示“零步或多步推导”)

$A_{lasttoken} \rightarrow Y$

2.2 节的例 4 即是这种情况。规则 53 尾部的语义动作 newSUB( )引用$5，即 subbody 的值，而在 perly.y 中，subbody 定义为：

```
subbody  :    block     { $$ = $1; }                          (规则 60)
         |    ';'    { $$ = Nullop; PL_expect = XSTATE; } ;   (规则 61)
```

其中 block(图 6)推导出的句子以‘}’结尾，依据变换法则 2，它在移植到 CUP 时，可按方法 2 变换成：

```
block     ::=  blockbody:bb RIGHTBRACKET {: RESULT = bb; :};      (规则 3)
blockbody::=   LEFTBRACKET:lt remember:rm lineseq:lq {: …… :};    (规则 183)
```

而依据变换法则 1，规则 61 将改写为 subbody ::= expectstat SEMICOLONFLAG，它推导出句子‘;’。这样，再依据变换法则 3，规则 53 将改写为

```
subrout   ::=  subrouthead:sr sublasttoken{: RESULT = sr; :};
```

另外增加两条规则：

```
subrouthead   ::=  SUB startsub:ssb subname:sn proto:pt subbody:sbd
                   {: RESULT = RESULT = OP.newSUB(ssb.intValue(), sn, pt, sbd); :};   (规则 185)
sublasttoken  ::=  RIGHTBRACKET                                  (规则 186)
                   |  SEMICOLONFLAG                              (规则 187)
```

变换法则 4：对于形如$A \rightarrow X_1\ X_2\ \alpha$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CUP Parser处理与Yacc Parser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。

这一变换法则可以解决 2.2 节例 3 中 block 现场保护时的问题：block_start( )中有 PL_hints &= ~HINT_BLOCK_SCOPEPL_hints，若当前解析的块在‘{’之后是 warn 或 die，词法分析器在识别出 warn 或 die 时，会执行 PL_hints |= HINT_BLOCK_SCOPEPL_hints。在 CUP Parser 中，由于对 warn 或 die 的词法分析在 block_start( )之前进行，因此使得词法分析器对 PL_hints 的设置不能象 Yacc Parser 那样在 block_start( )之后起作用。为此依据变换法则 4，在 perly.cup 中嵌入如图 10 的 parser code 代码块，它将对应于生成的解析器类中的一个 protected 方法 restoreHints( )；另外在文法的语义动作 block_start( )后，再增加对 parser.restoreHints()的调用。

```
parser code{:
  void restoreHints()
  {
    if ( ( cur_token.sym==Sym.FUNC ||
           cur_token.sym==Sym.LSTOP) &&
         cur_token.value instanceof Integer)
    {
      int optype=((Integer)cur_token.value).intValue();
      if (optype==OPCode.OP_WARN ||
          optype==OPCode.OP_DIE )
          PerlGlobal.PL_hints |=
             OPCode.HINT_BLOCK_SCOPE);
    }
  }
:};
```

图 10 restoreHints( )的定义

Fig.10 Definition of restoreHints( )

变换法则 5：假设文法中存在语法规则$A \rightarrow \alpha\beta$和$B \rightarrow \alpha$ %prec $\boldsymbol{t}$ ，其中α、β是任意非空的符号串，%prec $\boldsymbol{t}$使得符号串α具有与终结符$\boldsymbol{t}$相同的优先级。再假设项目$A \rightarrow \alpha \cdot \beta$和$B \rightarrow \alpha \cdot$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CUP Parser中仍按移进解决此冲突，则可将$B \rightarrow \alpha$ %prec $\boldsymbol{t}$修改为：

$B \rightarrow C$ %prec $\boldsymbol{t}$

$C \rightarrow \alpha$

变换法则 5 是就 Yacc 和 CUP 在处理 sr 冲突不完全一致所提出的变换方法。通过变换，将原先 sr 冲突中的归约式 $B \rightarrow \alpha$ %prec ***t*** 变为 $C \rightarrow \alpha$，此时 $C \rightarrow \alpha$没有优先级，因此将按确定性原则解决冲突，即优先移进。2.2 节的例 5 就可以采用这种方法来解决。

5 结束语

本文的工作是基于笔者一年多来在项目“将 253.perlbmk 到 JVM 的移植”中的开发测试实践以及对 LALR 解析器生成工具的深入分析。利用文中第 4 节总结的变换法则对 perly.cup 修改，已使我们得到的 CUP Parser 具有与原先 253.perlbmk 同样的解析功能和效果。由于 Perl 语言本身的复杂性，使得这里总结的变换法则将适用于很多其它程序设计语言或领域规范语言的解析器的 Java 再工程。

参考文献

[1] http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[2] S.C.Johnson, *YACC—yet another compiler-compiler*, Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[3] *Berkeley Yacc*(*v1.9*), 1993. Available at ftp://ftp.cs.berkeley.edu/ucb/4bsd/

[4] http://www.gnu.org/software/bison/bison.html

[5] Bob Jamison, *BYACC/Java* (*v1.1*), 2001. Available at http://troi.lincom-asg.com/~rjamison/byacc/

[6] Scott E. Hudson, *CUP Parser Generator for Java* (*v0.10k*), Sep 1999. Available at http://www.cs.princeton.edu/~appel/modern/java/CUP/

[7] John Aycock, *Why Bison is Becoming Extinct*, ACM Crossroads student magazine issue about tools/tutorials, Mid Summer 2001-7.5, Available at http://www.acm.org/crossroads/xrds7-5/bison.html.

[8] http://www.spec.org/cpu2000

[9] http://compilers.iecc.com/comparch/article/98-06-039

[10] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1986.
影印版：编译原理技术与工具，北京:人民邮电出版社，2002.2.