SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:387–404
Published online 3 February 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.962

Automatic detection and replacement of syntactic constructs causing shift/reduce conflicts[‡]

Thomas R. Kramer[*, †, §]

National Institute of Standards and Technology, Gaithersburg, MD 20899, U.S.A.

SUMMARY

A method is described for repairing some shift/reduce conflicts caused by limited lookahead in LALR(1) parsers such as those built by bison. Also, six types of Extended BNF (EBNF) construct are identified that cause a shift/reduce conflict when a Yet Another Compiler Compiler (YACC) file translated directly from EBNF is processed by bison. For each type, a replacement EBNF construct is described representing the same grammar and causing no shift/reduce conflict when its YACC equivalent is processed by bison. Algorithms are given for identifying instances of each type and transforming them into their replacements. The algorithms are implemented in an automatic parser builder that builds parsers for subsets of the DMIS language. The parser builder reads an EBNF file and writes C++ classes, a YACC file, and a Lex file, which are then processed to build a parser. The parsers build a parse tree using the C++ classes. The EBNF for DMIS is written in natural terms so that natural C++ classes are generated. However, if translated directly into YACC, the natural EBNF leads to 22 shift/reduce conflicts that fall into the six types. The parser builder recognizes the six constructs and replaces them automatically before generating YACC. The YACC that is generated parses in terms of unnatural constructs while building a parse tree using natural C++ classes. The six types of construct may occur in any statement-based language that uses a minor separator, such as a comma; hence knowing how to recognize and replace them may be broadly useful. Published in 2010 by John Wiley & Sons, Ltd.

Received 11 July 2008; Revised 25 November 2009; Accepted 26 November 2009

KEY WORDS: bison; conflict; DMIS; EBNF; parse; shift/reduce; YACC

1. INTRODUCTION

This paper describes methods of detecting and repairing syntactic constructs in the formal definition of a language that, if translated directly into a parser builder's specification, will cause the parser builder to report problems when it processes the specification in an attempt to build a parser for the language. The problems reported by the parser builder will prevent it from building a surely error-free parser. The repairs that are performed enable the parser builder to build a parser sure to be error-free.

The language considered here to define languages is called Extended BNF (EBNF) and is an ISO standard [1]. Backus–Naur Form (BNF) is a well-known production language used for defining context-free grammars that describe formal languages. A BNF file is a list of productions. Each production declares that a production name on the left side is equivalent to zero to many definitions on the right side. Each definition is a list of expressions. An expression may be a constant or the

[*]Correspondence to: Thomas R. Kramer, National Institute of Standards and Technology, Gaithersburg, MD 20899, U.S.A.
[†]E-mail: kramer@cme.nist.gov
[‡]This article is a U.S. Government work and is in the public domain in the U.S.A.
[§]Guest Researcher.

name of a production. For EBNF, a third type of expression is added: optional. Every production in the file except the first is used on the right side of one or more other productions.

Two of the EBNF extensions to BNF are considered here: (1) single optional items and (2) multiple optional items. A single optional item is indicated by square brackets, e.g. [real], and represents zero or one of the expressions inside the brackets. A multiple optional item is indicated by a positive integer (N) followed by an asterisk and an optional item, e.g. 2*[direction], and represents zero to N of the expression inside the brackets.

EBNF uses a vertical bar to separate the alternative definitions in a production, a semicolon to end a production, and commas to separate the expressions in a definition. A partial specification of the location of a coordinate system using either

- two or three reals for X, Y, and (optionally) Z for the location of the origin plus zero, one, or two directions (for the Z axis and the X axis), or
- a translated copy of some other coordinate system

might be represented by the following EBNF production:

```
location = real, real, [real], 2*[direction]
                | coordSystem, vector
                ;
```

For the purposes of representing an EBNF file in a computer, it is useful to define the following classes: production, definition, expression, and optional. It is also useful to define list types for production, definition, and expression. The parser builder considered in this paper starts by reading in an EBNF file and storing it in a parse tree in terms of instances of those classes and lists. The parse tree is then modified and used as described below.

It is straightforward to remove the two extensions so as to convert EBNF to BNF. Wherever a single (i.e. 1*) or N* optional appears in a definition, the definition is replaced by $N+1$ copies of itself. In the first copy, the optional is entirely removed. In the second copy, the optional is replaced by one of the bracketed expression. In the third copy, the optional is replaced by two of the bracketed expression, etc.

The usual method of building a parser is to use two interacting modules: (1) a lexical analyzer, which reads characters and recognizes tokens (words, numbers, punctuation, etc.), and (2) a grammar analyzer that receives tokens from the lexical analyzer and recognizes phrases, sentences, paragraphs, etc.

Yet Another Compiler Compiler (YACC) [2, 3] is both a file specification for a grammar analyzer and the name of a system that will process such a file specification and generate C or C++ code for a grammar analyzer. An alternate YACC file processor named bison (not an acronym) [3, 4] is available. In this paper, YACC means the file type and bison means the processor. A typical YACC file consists primarily of grammar rules and associated actions. The grammar rules in a YACC file are equivalent to BNF. YACC also provides for the explicit declaration of tokens, which are equivalent to BNF constants.

Lex (not an acronym) [3] is both a file specification for a lexical analyzer and the name of a system that will process such a file specification and generate C or C++ code for a lexical analyzer. An alternate Lex file processor named flex (fast Lex) [3, 5] is used in the parser builder. In this paper, Lex means the file type and flex means the processor.

Bison builds a parser that uses a set of states and a stack of tokens (words, numbers, separators, operators, etc.). The parser works by doing the following repeatedly until the stack is reduced to a single token, which is the left side of the first YACC rule.

- Call the lexical analyzer to read a token from a file written in the language being parsed.
- If a sequence of tokens on the top of the stack matches the sequence of tokens on the right side of a YACC rule, and that rule is applicable in the current state, then remove the sequence from the stack, replace it with the token on the left side of the YACC rule, and go to another state. This is called reducing. The choice of the new state depends on the previous state and the rule that was matched. The token just returned by the lexical analyzer is called the lookahead token and is used in deciding whether to reduce.

- Put the new token on the stack and go to another state. The choice of new state depends on the current state and the token. This is called shifting.

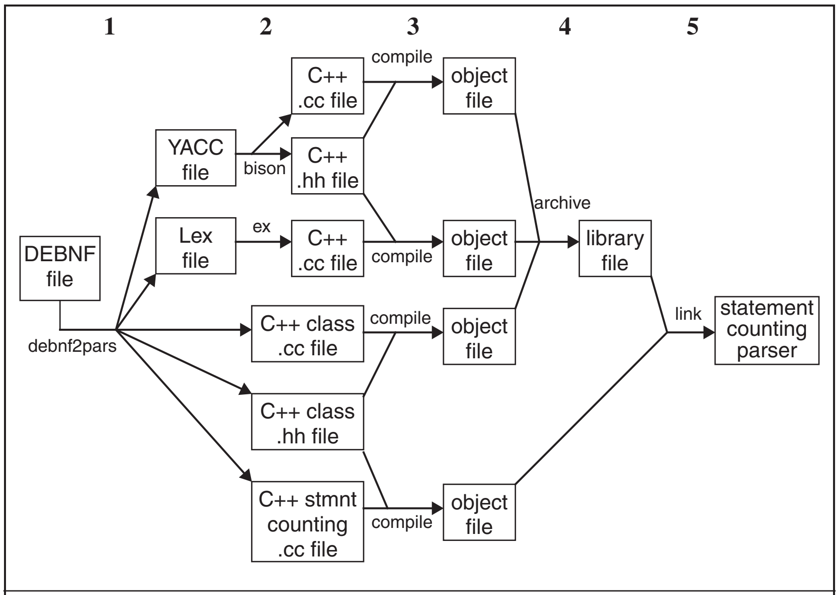
Bison may run into two common problems while it is running. First two different rules may match the same sequence of tokens. This is called a reduce/reduce conflict. Second, if one rule is matched so that reducing is appropriate, shifting instead of reducing might extend the sequence so that it matches the beginning of the sequence for some other rule. This is called a shift/reduce conflict. In either kind of conflict, it is not clear what decision bison should have on the parser make. Bison has rules for making up its mind, but its choice may be wrong.

An ideal automatic parser builder (excluding dealing with semantics) would read a formal representation of a language, build structures to represent a file written in the language, write access functions for the structures, and build a parser that, when it reads a file, will build a representation of the file in terms of the structures. The representation must be easy for an application programmer to use. The ideal can be approached by using EBNF to write a formal representation of the syntax of a language, automatically generating C++ classes from the EBNF, automatically generating Lex and YACC files from the EBNF, including actions in the YACC that build a parse tree in terms of the C++ classes, using flex to compile the Lex into C++, using bison to compile the YACC into C++, and using a C++ compiler to compile all the C++ code. Figure 1 shows the process graphically. The statement counting parser in Figure 1 is an application. Other applications using the parser would be built similarly, except that the source code would not be generated by debnf2pars.

There are several difficulties with this approach. This paper address one of them. The problem is: if 'natural' constructs are used in the EBNF, so that user-friendly C++ classes may be generated directly from the EBNF, if the natural EBNF constructs are translated directly into YACC, it is likely that shift/reduce conflicts will be reported when bison compiles the YACC file. Although shift/reduce conflicts will not prevent bison from generating code for a parser, they cannot be allowed to remain in the YACC because the parser may parse incorrectly. To deal with this automatically, the YACC generator needs to be able to recognize non-ambiguous EBNF constructs that will lead to shift/reduce conflicts, know alternate constructs that handle the same language but do not cause shift/reduce conflicts, and write YACC using the alternate constructs.

A system of the sort shown in Figure 1 has been built at the National Institute of Standards and Technology and incorporated in version 2.1.5 of the NIST Dimensional Measuring Interface Standard (DMIS) Test Suite. It may be downloaded from http://www.isd.mel.nist.gov/projects/metrology_interoperability/dmis_test_suite.htm. It is used for building parsers for subsets of a language named DMIS [6]. The YACC, Lex, and C++ class generator in the parser builder is called debnf2pars. Although debnf2pars contains a lot of code designed to deal with DMIS specifically, the methods discussed in this paper are applicable to handling any language that has certain characteristics similar to those of DMIS. These are: (1) the language is statement-based and includes an end of statement (EOS) symbol and (2) within statements, a comma or other symbol is used heavily as a secondary separator. In DMIS, a program is a list of statements, and commas are the primary cause of shift/reduce conflicts.

The 'natural' EBNF file for full DMIS has 1026 productions and contains 22 sets of interrelated productions that produce shift/reduce conflicts if translated directly into YACC. The sets fall into six types. For each type, a replacement type has been found. When each of the 22 sets of productions is replaced by a set of the appropriate replacement type and the resulting EBNF file is translated directly into YACC, a parser is produced that recognizes the same language but does not suffer from any shift/reduce conflicts. The replacements are unnatural in that they do not match the intuitively obvious way to describe the grammar. This paper describes the six types, how they are detected, and what their replacements are. Section 5 of this paper gives an overview of how the constructs are replaced and how the natural C++ classes are populated while parsing in terms of the unnatural replacements. Several thousand lines of complex but heavily documented C++ code are devoted to accomplishing those things. The interested reader is urged to download the code and read the in-line documentation.



1. debnf2pars reads the DEBNF file and writes:
    - a YACC code file for a DMIS parser,
    - a Lex code file for a DMIS lexical analyzer,
    - a C++ .cc code file for classes representing DMIS,
    - a C++ .hh header file for classes representing DMIS,
    - a C++ .cc code file for a system that counts DMIS statements.

2a. Bison reads the YACC file and writes two C++ files (.hh and .cc) for a parser.

2b. Flex reads the Lex file and writes a C++ .cc file for a lexical analyzer.

3a. A C++ compiler reads the .cc file produced by bison, the .hh file produced by bison, and the .hh file for C++ classes (arrow not shown on Figure 1), and writes an object file for a parser.

3b. The same compiler reads the .cc file produced by flex and the .hh file produced by bison and writes an object file for a lexical analyzer.

3c. The same compiler reads the .cc and .hh files for the C++ classes and writes an object file for the classes.

3d. The same compiler reads the .cc file for a system that counts DMIS statements and the .hh file for the C++ classes and writes an object file for the system.

4. An archiver combines object files for parser, lexical analyzer, and C++ classes into a library.

5. A linker links (1) the object file for the system that counts DMIS statements and (2) the library file into an executable parser that counts DMIS statements.

Figure 1. Steps in building a statement counting parser.

2. PREVIOUS WORK

The idea of building a parser automatically from EBNF is not new. McCoy and Wetmore [7] built an EBNF to YACC converter in 1980. A system named ebnf2yacc [8] built by Dan Nuffer has been available from sourceforge, a free software web site, since 2001. Neither of those two systems defined EBNF classes or generated Lex. A BNF converter built by Pellauer *et al.* [9] includes some of the extensions of EBNF and has several features in common with the parser builder reported here. None of the systems that build a parser automatically from EBNF is reported as including a system for detecting and replacing constructs causing shift/reduce conflicts.

The issue of dealing with shift/reduce conflicts in the YACC processor (which builds an LALR(1) parser), and more generally in LALR(k) or other types of parsers, has been discussed in hundreds of journal articles. The most-discussed types of shift/reduce conflict are those that arise from: (1) arithmetic (or similar) operators used to build expressions, (2) IF–ELSEIF–ELSE constructs, (3) nested lists, and (4) limited lookahead. All four of these are discussed in Section 8 of [3]. The first three of these are ambiguities in the language described by the grammar. The arithmetic operator problem is handled in YACC by precedence and associativity declarations that were added to YACC to deal with this particular problem. The IF–ELSEIF–ELSE problem may be handled in at least four ways, ranging from using awkward productions to ignoring it since the YACC processor's default action is the correct one. The nested lists' problem may (or may not) require the language to be revised. The limited lookahead problem is discussed below.

Little work has been done on the more focused problem of detecting and repairing shift/reduce conflicts automatically. Passos *et al.* [10] studied methods of removing LALR(k) conflicts. They state that automatic conflict removal can be accomplished but report that building a system that will actually detect and remove conflicts is future work.

3. A METHOD FOR FIXING SHIFT/REDUCE CONFLICTS CAUSED BY LIMITED LOOKAHEAD

In the limited lookahead problem, there is no ambiguity in the language, but a parser built by bison only looks ahead by one token, and that token may not resolve an immediate ambiguity the parser faces. Recent versions of bison deal with this by offering the option of making two copies of the parser when a shift/reduce conflict is encountered and continuing parsing with both until one reports an error, at which point the one in error is killed. This method is not efficient.

Here is an example of a limited lookahead problem. In this example and all other examples in this paper, ISO standard EBNF is used. The (* empty *) below is an EBNF comment indicating a definition with an empty expression list.

```
goStatement = 'GO', optionalSpeed , '(' , placeList , ')' ;

optionalSpeed = (* empty *) | '(' , speed , ')' ;
```

The problem arises when the parser has found a 'GO' followed by a left parenthesis. The parser cannot tell if (1) it has just seen an empty optionalSpeed and is now looking at the left parenthesis that precedes a placeList, or (2) it is looking at the left parenthesis that precedes speed.

A method for trying to fix an EBNF construct that causes a shift/reduce conflict in an LALR(1) parser is the following:

- Determine what the lookahead token is when the conflict occurs. With bison, this may be done by studying bison's .output file.
- Determine if looking one more token ahead would provide enough information to resolve the conflict. If so, continue. If not, stop.
- Try to find alternative EBNF constructs that recognize the same grammar by one of the following methods. In all of the methods, try to find rephrasings in which there are no longer two alternatives when the token causing the problem is read.

  A. Combine two productions (i.e. flatten). Flattening is not the best choice, since the YACC description grows exponentially with the number of productions combined, but it may be the only choice.
  B. Move the lookahead token to the left, either (i) by combining it with the production that precedes it or (ii) by removing it from the left side of a production that includes it and adding it to the production containing the production that includes it.
  C. Find some other rearrangement of the EBNF. In the case of lists, switching from left recursion to right recursion is often one component of a successful rearrangement.

The method may be applied as follows to find two fixes for the shift/reduce problem presented above:

- First, note that a left parenthesis is causing the problem. Also, note that a left parenthesis occurs before both a speed and a placeList, so no matter in any event, the next thing after 'GO' is a left parenthesis. This suggests that some form of rearrangement may be successful.
- Second, observe that looking one more token ahead, the next token must be either a speed or a placeList, and in both cases there is enough information to resolve the conflict (a speed indicates a non-empty optionalSpeed, whereas a placeList indicates an empty one).
- Third (using method A), try flattening by combining the optionalSpeed production with the goStatement production. This may be done in the following way:

  ```
  goStatement = 'GO' , '(' , placeList ', )'
          | 'GO' , '(' , speed , ')' , '(' , placeList ', )' ;
  ```

  Now the parser does not have to decide between shifting and reducing when it sees the left parenthesis following 'GO'. It always must shift. When it sees the next item (placeList or speed), it will know which of the two definitions it is looking at.
- Third (using method B) rephrase the two productions by moving the left parenthesis out of the optionalSpeed production and into the goStatement. This requires additional fiddling with parentheses to be sure that all pairs of parentheses match properly.

  ```
  goStatement = 'GO' , '(' , optionalSpeed , placeList , ')' ;

  optionalSpeed = (* empty *) | speed , ')' , '(' ;
  ```

  Once again, the parser does not have to make a decision when it sees the left parenthesis after 'GO'. If the parser next sees a speed, it will know it is starting into a non-empty optionalSpeed, but if it next sees the beginning of a placeList, it will know it has seen an empty optionalSpeed and is starting into a placeList. The new definition of optionalSpeed, however, is weird, not natural.

The method (obviously) does not provide a fix if any of the steps fails. In this regard, it is analogous to integration by parts in mathematics. It is a methodology that is applicable only in limited circumstances, but it works some of the time and is valuable in those cases in which it works. Integration by parts requires the user to experiment with rearranging terms. The conflict fixing method also requires the user to experiment with rearranging terms.

The method has not yet been incorporated in a computer program. In the case of DMIS, the curious history relevant to shift/reduce conflicts was

- EBNF for DMIS was donated containing 21 strange looking constructs, none of which caused a shift/reduce conflict. A construct was judged to be strange looking if it included any of the following:
  - right recursion for a list,
  - a list for which no production defined a list item (i.e. the item was merged with the definitions in the production for the list),
  - a trailing comma in the definition of a production.
- Each strange looking construct was replaced by natural EBNF for the same grammar.
  - Lists were made left recursive.
  - A separate production was written for each list item that had been merged with a list.
  - Trailing commas were removed from productions that had them.
- Each replacement caused a shift/reduce conflict[¶].
- The 21 pairs of constructs were classified into five types.

[¶]This almost certainly implies that the donated EBNF file was reverse engineered from a YACC file.

- For each type, the essential features were recorded, and an algorithmic method was devised for changing the natural EBNF into the strange looking EBNF.
- By studying the EBNF conversions required for changing natural conflict-ridden constructs into strange looking conflict-free constructs, the method was discovered.
- Natural, correct EBNF was written for a complex DMIS statement that had been modeled incorrectly in the donated EBNF. The natural, correct EBNF caused a shift/reduce conflict.
- The method was used to find a strange looking construct that did not cause a conflict in this last case. It was realized that this case did not fall into one of the five previously discovered types, so a sixth type was defined.

It would be possible, of course, to work from YACC equivalent to EBNF while trying to identify constructs that will cause a shift/reduce conflicts. That would be more difficult because optional items, which are explicit in EBNF, are only implicit in YACC. It is necessary to compare two YACC definitions in order to tease out an optional item. Optional items are key to identifying lists, and they play a key role in the six types of construct that have been identified as causing shift/reduce conflicts.

Knowledge of the six types may be valuable because instances of them may occur in other grammars. The next section presents the six types, their essential features, and an algorithm for fixing each type.

4. SIX PROBLEMATIC CONSTRUCT TYPES AND THEIR REPLACEMENTS

A set of interrelated EBNF productions that will lead to a shift/reduce conflict if translated directly into YACC (after expanding optionals) is called a problematic construct. A problematic construct is given here by providing a number of properties that characterize the interrelationships.

The key feature of the first three problematic constructs is that a production having a definition that ends with an optional is used by another production. A key feature of the last three problematic constructs is nested lists. The last problematic construct has both key features.

All of the lists involved in the six problematic constructs with which this paper deals are comma-separated lists. For brevity, in the remainder of the paper, ‘list’ means comma-separated list.

Throughout this section, the term ‘the parser’ means ‘the parser that would be built by bison using the grammar rules just shown.’

In all the cases described below, the replacement construct recognizes the same sequences of expressions as the problematic construct but does not cause a shift/reduce conflict.

The explanations of why the shift/reduce conflicts occur in the problematic constructs and why they do not occur in the replacement constructs are given here in text. Readers familiar with bison may find it helpful to write YACC files corresponding to the EBNF (with optionals replaced by using two definitions), process them with bison using the $-v$ (verbose) option, and examine the .output files. The shift/reduce conflicts described here will appear explicitly in the .output files.

4.1. Useful EBNF conventions

The extensions to BNF in EBNF make it possible to write the same semantic statement in more than one way. In particular, there are two ways to say that a sequence of expressions is a single optional and at least three ways to define a non-empty, left-recursive list of arbitrary length. In addition, if a production has two or more definitions, it is permissible to split the production into multiple productions, each with the same name on the left side.

The two methods of writing a single optional are shown in Figure 2. The choices are either to use EBNF’s square bracket notation or write two definitions.

The three methods of writing a non-empty, left-recursive list of arbitrary length are shown in Figure 3. The choices are the following. First, use square bracket notation. Second, use curly braces notation (in EBNF, {X} means zero to many X*s*). Third, use two definitions instead of an optional.

```
geomMinor = gLabel , [C , OFFSET , C , rentVal] ;
geomMinor = gLabel | gLabel , C , OFFSET , C , rentVal ;
                 (the C , OFFSET , C , rentVal part is optional)
```

Figure 2. Two ways to represent a single optional.

```
itemList = [itemList , C ] , item ;
itemList = {item , C } , item ;
itemList = item | itemList , C , item
```

Figure 3. Three ways to represent a non-empty left-recursive list.

```
itemList = [itemList , C ] , item ;
item = 'THING1' , [C , 'THING2'] ;
                              (simplest form)
```

Figure 4. First problematic construct.

In order to make C++ code generation and the automatic analysis of an EBNF file easier, three conventions regarding the use of EBNF have been adopted. In addition, to make the printed representation of productions easy on human eyes, a literal comma is represented by C, not by ‘,’.

First, optionals are always represented using square brackets (the first format from Figure 2).

Second, lists are never empty, are always represented using square brackets (the first format from Figure 3), and are always left recursive. If a place in a production should contain a list that may be empty, the place is filled by a (non-empty) list enclosed in square brackets to indicate that the list is optional. To make it easy for humans to recognize that a production represents a list, all such productions have ‘List’ at the end of their names. In automatically recognizing that a production defines a list, however, it is the structure of the list representation that is recognized, not the suffix ‘List’ in the name.

Third, no two productions have the same name on the left side.

While these conventions make it easier to process EBNF automatically, the conventions are not essential. If they were not followed, it would be straightforward to write an EBNF preprocessor that would read an EBNF file, look for all forms of commas, optionals, lists, and productions with the same name, and write an EBNF file putting everything into the conventional form and describing the same target language.

4.2. First problematic construct

The first problematic construct has the following three properties. Its simplest form is shown in Figure 4.

1. There is a list (itemList).
2. An instance of itemList may appear in a target language file only just before the end of a statement.
3. The list item (item) used by itemList has at least one definition that has at least two expressions and ends with an expression that is an optional starting with a comma.

When the parser has seen a ‘THING1’ and has C as the lookahead token, the parser does not know whether the ‘THING1’ is an entire item and the C is separating one item from the next (in which case the parser should reduce) or whether only half an item has been seen and the C is to be followed by a ‘THING2’ (in which case the parser should shift).

itemList = ’THING1’ , [C , ’THING2’] , [C , itemList] ;

~~item = ’THING1’ , [C , ’THING2’] ;~~

(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion)

Figure 5. First replacement construct.

displyStm = ’DISPLY’ , ’/’ , displyOff , EOS
| ’DISPLY’ , ’/’ , displySpecList , EOS ;

displySpecList = [displySpecList , C] , displySpecItem ;

displySpecItem = device , C , ’DMIS’ , [C , vLabel]
| device , C , vLabel ;

---

displySpecList = device , C , ’DMIS’ , [C , vLabel] , [C , displySpecList]
| device , C , vLabel , [C , displySpecList] ;

~~displySpecItem = device , C , ’DMIS’ , [C , vLabel]~~
~~| device , C , vLabel ;~~

The original is above the line, productions changed in the replacement are shown below.
(*italic* indicates change, ~~strikethrough~~ indicates deletion)

Figure 6. DMIS example of first construct.

The first problematic construct is repaired as follows. The end result is shown in Figure 5.

1. Switch itemList from left recursion to right recursion.
2. Replace item in the production for itemList with its definition.
3. If item does not appear in the right side of any other production, remove the production for item from the list of productions.

With the replacement construct, when the parser has seen a ‘THING1’ and has C as the lookahead token, there is no item to look for. Moreover, the only thing that ends the itemList is EOS; hence, as long as the parser keeps getting acceptable tokens from the lexer, the parser just keeps shifting, until it gets EOS, at which point it reduces as many times as there are ‘THING1’s on the stack, ending up with a single itemList.

Determining from the EBNF grammar that an instance of itemList may appear in a target language file only just before the end of a statement is non-trivial. Note first that by ‘instance of itemList’ we mean to include all of the items that occur sequentially, not just some of them. Thus, in the analysis of the grammar, we exclude the appearance of itemList in its own definition. Second, it may be that itemList does not appear in the grammar just before EOS, but does appear only at the end of a definition of a production that is used elsewhere only before EOS. To make a firm determination, a recursive alwaysFollowedByNewline function is used, which returns a boolean and takes as its argument a pointer, prod, to a production to test. The function looks at all definitions of all productions that use prod. If the name of prod appears at the end of a definition of some production, user, the function calls itself recursively with user as its argument. If prod appears as the next-to-last expression in a definition, the function checks that the last expression is EOS; if the check fails, false is returned. If prod appears before the next-to-last place in a definition, false is returned (since EOS is last everywhere it is used). The top-level function call returns false if any branch of the tree of recursive function calls returns false. It returns true if all branches return true.

An example of a DMIS instance of the first construct is shown in Figure 6 along with productions changed in its replacement. There are five other instances of this construct in DMIS. The itemList in this example is displySpecList, and the item is displySpec. The production for displyOff is omitted from the figure since it is not a part of the problem.

outerItem = innerItem , C , 'THING1' ;

innerItem = 'THING2' , [C , 'THING3'] ;

(simplest form)

Figure 7. Second problematic construct.

outerItem = innerItemC , 'THING1' ;

~~innerItem = 'THING2' , [C , 'THING3'] ;~~

innerItemC = 'THING2' , [C , 'THING3'] , C ;

(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion, **bold** indicates new)

Figure 8. Second replacement construct.

4.3. Second problematic construct

The second problematic construct has the following four properties. Its simplest form is shown in Figure 7.

1. There is a production (outerItem) that is not a list.
2. An instance of outerItem may appear in a target language file only just before the end of a statement. This is checked by the alwaysFollowedByNewline function described earlier.
3. outerItem has a definition that uses another production (innerItem) followed by a comma and at least one other expression.
4. The innerItem has at least one definition that has at least two expressions and ends with an expression that is an optional starting with a comma.

When the parser has seen a 'THING2' and has C as the lookahead token, the parser does not know whether the 'THING2' is an entire innerItem and the C is separating the innerItem from 'THING1' (in which case the parser should reduce) or whether only half an innerItem has been seen and the C is to be followed by a 'THING3' (in which case the parser should shift).

The second problematic construct is repaired as follows. The end result is shown in Figure 8.

1. Define innerItemC to be the same as innerItem but with a trailing comma.
2. In each place in a definition of outerItem in which innerItem,C appears, replace innerItem,C with innerItemC.
3. If innerItem no longer appears in the right side of any production, remove the production for innerItem from the list of productions.

With the replacement construct, when the parser has seen a 'THING2' and has C as the lookahead token, it shifts the C and then gets the next lookahead token, which must be either 'THING1' or 'THING3'. If the lookahead token is 'THING1', the parser reduces 'THING2',C to innerItemC. If the lookahead token is 'THING3', the parser shifts the 'THING3'.

An example of a DMIS instance of the second construct is shown in Figure 9 along with the productions changed in its replacement. DMIS contains eight other instances of this construct. The productions for datsetMcs and datsetMatrix are omitted from the figure since they are not part of the problem. The outerItem in this example is datsetDats, and the innerItem is datsetSpec. In order to determine that an instance of datsetDats appears only before EOS in a DMIS file, it is necessary to observe that datsetDats appears only at the end of a definition of datsetMinor, and datsetMinor appears only before EOS.

4.4. Third problematic construct

The third problematic construct has the following five properties. Its simplest form is shown in Figure 10.

1. There is a production (outerItem) that is not a list.
2. In every definition that uses outerItem, outerItem is followed by a comma.

datsetStm = dLabel , '=' , 'DATSET' , '/' , datsetMinor , EOS ;

datsetMinor = datsetMcs
    | datsetDats
    | datsetMatrix ;

datsetDats = 2*[datsetSpec , C] , datsetSpec ;

datsetSpec = datLabel , C , orig , 2*[C , orig]
    | datLabel , C , dir , 3*[C , orig] ;

---

datsetDats = 2[datsetSpecC] , datsetSpec ;*

datsetSpecC = datLabel , C , orig , 2*[C , orig] , C
    | datLabel , C , dir , 3*[C , orig] , C

The original is above the line, productions changed in the replacement are shown below.
(*italic* indicates change, **bold** indicates new)

Figure 9. DMIS example of second construct.

outestItem = outerItem , C , 'THING1' ;

outerItem = 'THING2' , C , innerItem ;

innerItem = 'THING3' , [C , 'THING4'] ;

(simplest form)

Figure 10. Third problematic construct.

outestItem = outerItem , 'THING1' ;

outerItem = 'THING2' , C , innerItemC ;

~~innerItem = 'THING3' , [C , 'THING4'] ;~~

innerItemC = 'THING3' , [C , 'THING4'] , C ;

(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion, **bold** indicates new)

Figure 11. Third replacement construct.

3. Every definition of outerItem has another production (innerItem) at the end.
4. If innerItem is used elsewhere in a definition of outerItem, it is followed by a comma.
5. The innerItem has at least one definition that has at least two expressions and ends with an expression that is an optional starting with a comma.

When the parser has seen a 'THING3' and has C as the lookahead token, the parser does not know whether the 'THING3' is an entire innerItem, so that an entire outerItem has been seen and the C is separating the outerItem from 'THING1' (in which case the parser should reduce) or whether only half an innerItem has been seen and the C is to be followed by a 'THING4' (in which case the parser should shift).

The third problematic construct is repaired as follows. The end result is shown in Figure 11.

1. Define innerItemC to be the same as innerItem but with a trailing comma.
2. In each place in a definition of outerItem in which innerItem,C appears, replace innerItem,C with innerItemC.
3. In every definition that uses outerItem, delete the C that follows outerItem.
4. If innerItem no longer appears in the right side of any production, remove the production for innerItem from the list of productions.

With the replacement construct, when the parser has seen a 'THING3' and has C as the lookahead token, it shifts the C and then gets the next lookahead token, which must be either

outerList = [outerList , C] , outerItem ;
outerItem = ’THING’ , C , innerList ;
innerList = [innerList , C] , innerItem ;
(simplest form)

Figure 12. Fourth problematic construct.

outerList = ’THING’ , C , innerList , [C , outerList] ;
~~outerItem = ’THING’ , C , innerList ;~~
innerList = [innerList , C] , innerItem ;
(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion)

Figure 13. Fourth replacement construct.

‘THING1’ or ‘THING4’. If the lookahead token is ‘THING1’, the parser reduces ‘THING3’,C to innerItemC. If the lookahead token is ‘THING4’, the parser shifts the ‘THING4’.

4.5. Fourth problematic construct

The fourth problematic construct has the following three properties. Its simplest form is shown in Figure 12.

1. There is a list (outerList).
2. An instance of outerList may appear in a target language file only just before the end of a statement. This is checked by the alwaysFollowedByNewline function described earlier.
3. The item (outerItem) used to define outerList has a definition with at least three expressions, the last of which is a list (innerList) and the next-to-last of which is a comma.

When the parser has seen an innerItem and has C as the lookahead token, the parser does not know whether it has seen an entire outerItem and the C is separating one outerItem from the next (in which case the parser should reduce) or whether only part of an outerItem has been seen and the C is separating one innerItem from the next (in which case the parser should shift).

The fourth problematic construct is repaired as follows. The end result is shown in Figure 13.

1. Switch outerList from left recursion to right recursion.
2. Replace outerItem in the production for outerList with its definition.
3. If outerItem does not appear in the right side of any other production, remove the production for outerItem from the list of productions.

Note that innerList is not changed.

This is the same structural change as in the first problematic construct. Thus, if there is a production *P* defining a list item, and *P* has one definition with an optional at the end and another definition that has a list at the end, a single structural change will fix both problems.

With the replacement production, when the parser has seen an innerItem and has C as the lookahead token, there is no outerItem to look for; the C is shifted onto the stack. As long as innerItems continue to be found following commas, the parser reduces them into an innerList. When the parser sees a ‘THING’ following a comma, it shifts it onto the stack. The end of the stack becomes a series of subsequences of the form ‘THING’,C,innerList,C. This stops only when innerItem is followed by EOS, at which point the parser reduces as many times as there are ‘THING’s on the stack and ends up with a single outerList.

An example of a DMIS instance of the fourth construct is shown in Figure 14 along with the productions changed in its replacement. DMIS contains three other instances of this construct. The productions for swLabel, stringVal, and angle are omitted from the figure since they are not outputSensorWristList = [outputSensorWristList , C] , outputSensorWristItem ;

outputSensorWristItem = swLabel , C , outputSensorWristAngleList ;

outputSensorWristAngleList =
    [outputSensorWristAngleList , C] , outputSensorWristAngle ;

outputSensorWristAngle = stringVal , C , angle ;

---

*outputSensorWristList = swLabel , C , outputSensorWristAngleList ,
    [C , outputSensorWristList] ;*

~~outputSensorWristItem = swLabel , C , outputSensorWristAngleList ;~~

The original is above the line, productions changed in the replacement are shown below.
(*italic* indicates change, ~~strikethrough~~ indicates deletion)

Figure 14. DMIS example of fourth construct.

someProduction = outerList , C , anotherItem ;

outerList = [outerList , C] , outerItem ;

outerItem = 'THING' , C , innerList ;

innerList = [innerList , C] , innerItem ;

(simplest form)

Figure 15. Fifth problematic construct.

part of the problem. The outerList in this example is outputSensorWristList, the outerItem is outputSensorWristItem, and the innerList is outputSensorWristAngleList.

4.6. Fifth problematic construct

The problem in the fifth problematic construct is the same as in the fourth problematic construct with one further twist: the outer list is not at the end of a statement. Rather, in each definition in which the outer list appears, it is followed by a comma and then other expressions. This is fixed similarly (by using right recursion on the outer list and not defining an item for the outer list) but with two things added to deal with the added twist: (1) the replacement outer list has the comma outside the optional recursion rather than inside, and (2) in each definition in which the outer list appears, the comma following the outer list is removed.

In detail, the fifth problematic construct has the following three properties. Its simplest form is shown in Figure 15.

1. There is a list (outerList).
2. In each definition in which outerList appears, it is followed by a comma.
3. The item (outerItem) used to define outerList has a definition with at least three expressions, the last of which is a list (innerList) and the next-to-last of which is a comma.

The parser's quandary in this case is identical to its problem with the fourth problematic construct.

The fifth problematic construct is repaired as follows. The end result is shown in Figure 16.

1. Switch outerList from left recursion to right recursion.
2. Replace outerItem in the production for outerList with its definition.
3. Define outerListC by copying the revised definition of outerList and changing outerList to outerListC in two places.
4. Modify outerListC by moving the comma out of the recursion (so that an instance of outerListC will always have a comma at the end).
5. In each definition in which outerList appears, replace outerList,C with outerListC.

someProduction = outerListC , anotherItem ;

~~outerList = [outerList , C] , outerItem ;~~

outerListC = 'THING' , C , innerList , C , [outerListC] ;

~~outerItem = 'THING' , C , innerList ;~~

innerList = [innerList , C] , innerItem ;

(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion, **bold** indicates new)

Figure 16. Fifth replacement construct.

```
outerList = [outerList , C] , outerItem ;
outerItem = 'THING1' , C , innerList ;
innerList = [innerList , C] , innerItem ;
innerItem = 'THING2' , [C , 'THING3'] ;
```

(simplest form)

Figure 17. Sixth problematic construct.

6. Remove the definition of outerList from the list of productions.
7. If outerItem does not appear in the right side of any other production, remove the production for outerItem from the list of productions.

The reason this works is the same as the reason the fourth replacement construct works, except that where EOS served to terminate an outerList in the fourth replacement construct, anotherItem serves to terminate outerListC in this case. In this case, the troublemaking comma is handled by including it in the production; hence, the parser does not have to make any decision when it sees a comma; the comma is always shifted.

4.7. *Sixth problematic construct*

The sixth problematic construct is like the first and fourth problematic constructs combined. An outer list contains an inner list, and the inner list item ends with an optional expression starting with a comma. In detail, the sixth problematic construct has the following four properties. Its simplest form is shown in Figure 17.

1. There is a list (outerList).
2. An instance of outerList may appear in a target language file only just before the end of a statement. This is checked by the alwaysFollowedByNewline function described earlier.
3. The list item (outerItem) used to define outerList has a definition with at least three expressions, the last of which is a list (innerList), and the next-to-last of which is a comma.
4. The list item (innerItem) used by the innerList has at least one definition that has at least two expressions and ends with an expression that is an optional starting with a comma.

Bison reports two shift/reduce conflicts. These occur exactly where and how they would be expected based on the analysis of the first and fourth problematic constructs.

The approach to fixing this includes adding a trailing comma to the inner list. However, since an instance of the outer list must not end with a comma and adding a trailing comma to the inner list causes the outer list to end with a comma, an additional repair is needed to fix that. The sixth problematic construct is repaired as follows. The end result is shown in Figure 18.

1. Switch outerList from left recursion to right recursion.
2. Replace outerItem in the production for outerList with its definition.

outerList ='THING1' , C , [innerListC] , innerItem | 'THING1' , C , innerListC , outerList ;

~~outerItem = 'THING1' , C , innerList ;~~

innerListC = [innerListC] , 'THING2' , C , ['THING3' , C] ;

innerItem = 'THING2' , [C , 'THING3'] ;

~~innerList = [innerList , C] , innerItem ;~~

(simplest form, *italic* indicates change, ~~strikethrough~~ indicates deletion, **bold** indicates new)

Figure 18. Sixth replacement construct.

3. Define innerListC by starting with the production for innerList, replacing innerItem with its definition, removing the comma from the recursion, and changing the end of the definition so that an instance of innerListC will always have a comma at the end.
4. In the definition of outerList, remove the C from [C, outerList] since innerListC now provides a comma.
5. Expand the optional in the definition of outerList, so that there are two definitions, one ending in outerList and one ending in innerListC.
6. In the definition of outerList that ends with innerListC, replace innerListC with [innerListC], innerItem. This fixes the problem that innerListC now ends with a comma.
7. If outerItem does not appear in the right side of any other production, remove the production for outerItem from the list of productions.
8. If InnerList does not appear in the right side of any other production, remove the production for InnerList from the list of productions.

5. IMPLEMENTATION DETAILS

The source code for the debn2pars generator is in the 12 000-line file debnf2pars.y. Only 140 lines of that is YACC code. The rest is roughly half C++ code and half documentation of the code. In addition, as mentioned earlier, classes have been defined and implemented for representing EBNF. That is another 1700 lines of code, also about half documentation. In total, there are over 100 pages of in-line documentation. The NIST DMIS Test Suite 2.1.5, which contains all the source code, is publicly available for download and reuse. The Test Suite contains a 42-page users manual and an 18-page system builders manual. This section attempts only to describe the top-level functioning of debnf2pars and to give a broad brush description of aspects of the code relevant to identifying and replacing problematic constructs.

A class is defined for each of the elements of EBNF (production, definition, expression, and optional). To support the complex operations performed in debnf2pars, each element (in addition to its obvious attributes) has specialized attributes to support certain operations. The attributes of the four elements are

- production
  - defList * defs; // the definitions of the production
  - bool endsInOptional; // true = the last expression in at least one definition is optional
  - fixTypeE fixType; // the kind of fix that has been or should be applied
  - int isList; // 0 = not a list, 1 = list no commas, 2 = list with commas
  - bool isSupertype; // true = this production is a supertype of other productions
  - char * lhs; // the name of the production (lhs = left-hand side)
  - prodList subtypeOf; // list of productions of which this is a subtype
  - prodList usedIn; // list of productions in which the name of this appears
  - bool wasPrinted; // true = classes for this production have been printed

- definition
  - char * className; // name of class to instantiate in action
  - expList * expressions; // expressions giving the definition
  - defList * newDefs; // revised definitions to replace this
- expression
  - int theType; // one of eight enumerated types
  - char * itemName; // name in the expression (only some types have names)
  - char * attName; // name for the class attribute that will contain an instance
  - optional * optValue; // optional this points to (set only if theType is OPTIONAL)
  - production * prodValue; // production this points to (set if theType is NONTERMINAL)
- optional
  - expList * expressions; // the expression in the optional
  - int digit; // the multiplicity of the optional

List cell classes and list classes are defined for EBNF production, definition, and expression. A great deal of analysis is required to identify problematic constructs, and then a great deal of list surgery is required to fix the problems. Standard C++ list classes do not allow enough fine control for the list surgery; hence, list classes were defined from scratch. For each of the three list classes, a doubly linked list cell class is defined with three pointer attributes: next, back, and data. Then the list class itself is defined. All three list classes have attributes named first and last, which are pointers to list cells. When a list is empty, first and last point to NULL. All three list classes have findLength, pushBack, insertAfterCell, and removeCell functions. Those list classes that need them have other functions, including dup (duplicate), pushFront, spliceInAfterCell, and spliceInFirst. The last two of these insert one list of expressions in another list of expressions. In addition, all attributes of all EBNF classes are public; hence, access functions are not needed in debnf2pars. Moreover, debnf2pars does additional kinds of specialized list surgery (such as removing the last two elements of a list) for which member functions of the EBNF classes are not defined.

At the top level of processing, debnf2pars does the following:

- The EBNF file identified in the command argument is read in. This builds a parse tree to represent the EBNF in terms of the classes. The trunk of the parse tree is simply a list of productions. Parsing the file also causes a list of token names to be built and sets the isList attribute of every production.
- The prepareStatementNames function goes through the list of productions and makes a list of those that are DMIS statements.
- The reviseSpelling function first makes the list of token lexes to be identical to the list of token names. Then it goes though the list of productions, and if a production gives a spelling for a token name, the name is changed in the token lexes list.
- The findUsedIn function of the prodList class goes through the list of productions. For each production P, it goes through the production list and, if P appears in the expressions of any definition of a production Q, adds Q to the usedIn list of P. While it is looking through the expressions of the definitions of Q, it makes P be the value of the prodValue attribute of any expression whose itemName is the lhs of P.
- The addData function goes through the list of productions. For each production P, it sets the className attribute of each definition and, if every definition of P has only one expression, and that expression is a production that is not a list, it sets the isSupertype attribute of P to true and adds P to the subtypeOf list of all the productions in the expressions of P's definitions.
- The printCppClasses function is called to print the .hh and .cc files defining and implementing the classes to represent the subset of DMIS given in the EBNF file.
- The fixConflicts function is called to find and fix problematic constructs as described in this paper. It works by going through the productions. If a production P is a comma-separated

list that has its fixType set to fixTypeNone, it calls fixConflictList to check further and make a fix if necessary. Otherwise, if P has an optional at the end starting with a comma, it calls fixConflictUsers to check further and make a fix if necessary. There are seven other 'fixConflict...' functions subordinate to fixConflictList and fixConflictUsers that may be called (and they have many subordinate functions that may be called). If a fix is made to *P*, the fixType of *P* is set to the type of fix that was made. Also, newDefs may be set for the definitions of *P* or other productions participating in the problematic construct.
- The printYacc function is called to print the YACC file for a parser. There are 42 other 'PrintYacc...' functions subordinate to printYacc, many of which specialize in printing the YACC for a production that has a given fixType. One of the immediate subordinates of printYacc is printYaccProductions, which goes through the list of productions and prints the YACC for each one.
- The printLex function is called to print the Lex for a parser. It has eight 'printLex...' subordinates. PrintLex has no involvement in fixing shift/reduce conflicts.
- The printParser function (with seven 'printParser...' subordinates) is called to print a C++ code file for an executable parser that uses the YACC parser (by calling yyparse) and then traverses the parse tree and counts DMIS statement uses. PrintParser has no involvement in fixing shift/reduce conflicts.

Except for the functions near the top of the 'fixConflict...' function hierarchy, which are easy to understand, the functions in that hierarchy include a lot of complex, low-level code that creates list cells, destroys list cells, redirects pointers to list cells, and so forth. The functions that deal with printing YACC for productions whose fixType is not set to fixTypeNone are similarly complex. The only way to get a good understanding of those complex functions is by reading the in-line documentation.

6. CONCLUSION

A new method for fixing shift/reduce conflicts in YACC files has been presented. It seems likely that, using this approach, additional examples of repairable shift/reduce conflicts may be found in the existing statement-based languages that use a minor separator such as a comma.

Six examples of types of EBNF constructs that cause shift/reduce conflicts have also been presented, with explicit instructions for how to change the constructs to eliminate the conflicts. The YACC equivalents of these examples are trivial to derive; hence, the examples provide six new instances of YACC problems and how to fix them.

The method and the examples should be useful to people and computers who are building YACC parsers (or other LALR(1) parsers) and have encountered shift/reduce conflicts. They may also be useful to those people who are maintaining parser builder tools.

Developing algorithms for automatically finding and fixing EBNF constructs that will lead to shift/reduce conflicts if translated directly into YACC has enabled building an automatic parser builder that will build natural C++ classes and build parsers that will build a parse tree using the natural C++ classes while parsing in terms of unnatural YACC constructs.

REFERENCES

1. ISO/IEC 14977. Information technology—Syntactic metalanguage—Extended BNF. *ISO/IEC 14977*, ISO, Geneva, Switzerland, 1996(E). Available on the internet at: http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html [2008].
2. Johnson SC. YACC—Yet Another Compiler–Compiler. *Computing Science Technical Report 32*, Bell Laboratories, Murray Hill, NJ, 1975. Available on the internet at: http://www.tom-yam.or.jp/2238/ref/yacc.pdf [2008].
3. Levine JR, Mason T, Brown D. *Lex & Yacc*. O'Reilly: U.S.A., 1995.
4. Donnelly C, Stallman R. Bison, the Yacc-compatible parser generator, version 2.3, 2006. Available on the internet at: http://www.gnu.org/software/bison/manual [2008].

5. Paxon V, Estes WL, Millaway J. Flex, a fast scanner generator, version 2.5.31. Regents of the University of California, 2003. Available on the internet at: http//www-inst.eecs.berkely.edu/~cs164/sp08/docs/flex/flex.pdf [2008].
6. Dimensional Metrology Standards Consortium. Dimensional Measuring Interface Standard 5.1, *ANSI/DMIS 105.1-2007*, Part 1, 2007.
7. McCoy EE, Wetmore T III. A program for the conversion of productions in an extended Backus–Naur-Form to an equivalent Backus–Naur-Form. Naval Postgraduate School, Monterey, CA, 1980. Available from NTIS.
8. Nuffer D. *ebnf2yacc*. Available on the internet at: http://sourceforge.net/projects/ebnf2yacc [2008].
9. Pellauer M, Forsberg M, Ranta A. BNF converter multiligual front-end generation from labelled BNF grammars. *Technical Report No. 2004-09*, Chalmers University of Technology and Goteborg University, 2004. Available on the internet at: http://www.cs.chalmers.se/~markus/tech2004.pdf [2008].
10. Passos LT, Bigonha MAS, Bigonha RS. A methodology for removing LALR(k) conflicts. *Journal of Universal Computer Science* 2007; **13**(6):737–752.