Maintainers Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)

Thomas R. Kramer (thomas.kramer@nist.gov, phone 301-975-3518)
John Horst (john.horst@nist.gov, phone 301-975-3430)

Intelligent Systems Division
National Institute of Standards and Technology
Technology Administration
U.S. Department of Commerce
Gaithersburg, Maryland 20899, USA

NISTIR 7720
October 25, 2010

Disclaimer

Acknowledgements

Funding for the work described in this paper was provided to Catholic University by the National Institute of Standards and Technology under grant Number 70NANB9H9131.

Table of Contents

1 Introduction

1.1 Overview

The National Institute of Standards and Technology (NIST) is supporting use of the Dimensional Measuring Interface Standard (DMIS) language. One form of this support is providing the NIST DMIS Test Suite (NDTS), which has been developed in the Intelligent Systems Division (ISD) of NIST.

This manual is a maintainers manual for the NIST DMIS Test Suite version 2.2.1. A maintainer is anyone who is considering modifying the test suite[1], particularly NIST personnel building a new version of the test suite.

The test suite is intended to serve two purposes:

- to help users and vendors use version 5.2 of DMIS,
- to provide utilities and test files for conducting conformance tests on
  - DMIS input files
  - computer systems that generate DMIS input files
  - computer systems that execute DMIS input files.

There are also a “Users Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)” and a “System Builders Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)”. Read those manuals carefully before starting on this manual. They contain a lot of important information that is not duplicated here. If you find terms in this manual you do not understand, search for them in the other two manuals.

1.2 How the Test Suite is Made Available

The primary mode of distributing the test suites has been to post them in the ISD web site as zip files. The web site is:

http://www.isd.mel.nist.gov/projects/metrology_interoperability/dmis_test_suite.htm

The following earlier versions of the test suite are in the site in addition to this one:

- DMIS Test Suite 1.01 for version 5.0
- DMIS Test Suite 1.1 for version 5.0
- DMIS Test Suite 2.0 for version 5.1
- DMIS Test Suite 2.1 for version 5.1
- DMIS Test Suite 2.1.1 for DMIS version 5.1
- DMIS Test Suite 2.1.4 for DMIS version 5.1
- DMIS Test Suite 2.1.5 for DMIS version 5.1

For this release and NDTS 2.1.5, since the test suite is very large (so that prospective users may want to look at the manuals before deciding whether to download it), the manuals may be downloaded separately from the same site.

In addition, NDTS 2.1.5 is available from http://code.google.com/p/dmis-test-suite/downloads/list.

It would be good to make it possible, in addition, to use the NDTS utilities on the web without

1. In the remainder of this manual “the test suite” means the NIST DMIS Test Suite, version 2.2.1.

downloading anything. The user would upload a file to test and it would be run through whichever utility the user wants (dmisParser, dmisConformanceChecker, dmisConformanceRecorder, or dmisConformanceTester). Utilities for other languages have been made available that way at NIST, and the idea of doing that for DMIS has been kicking around for several years.

Early versions of the NDTS were emailed occasionally. Versions 2.1.5 and 2.2.1 are too large to email.

1.3 Documentation

NDTS documentation includes:

- the three manuals (users, system builders, maintainers),
- the Excel spreadsheet defining conformance classes (last edited September 2010)
- in-line documentation of the code in ebnf, generator, and utilityComponents directories,
- one journal article[1].

1.4 Terminology

In this manual:

- bison = a system for generating C++ code from a YACC file (it's not an acronym)
- BNF = Backus-Naur Form
- DEBNF = DMIS EBNF
- DMIS = Dimensional Measuring Interface Standard
- DMSC = Dimensional Metrology Standards Consortium
- EBNF = Extended BNF
- flex = a system for generating C++ code from a Lex file (it's not an acronym)
- Lex = a language for encoding a lexical analyzer (it's not an acronym)
- NDTS = NIST DMIS Test Suite
- NIST = National Institute of Standards and Technology
- YACC = Yet Another Compiler Compiler, a language for encoding a parser
- The term "subclass" (rather than "child class" or anything else) is used to refer to a C++ class that is derived from another class.
- The term "rule", rather than "production" will be used for a YACC production. "Rule" is the usual term, and makes it easy to differentiate between EBNF productions and YACC productions.

1.5 Use of Fonts and a Warning

1.5.1 Fonts

Five formal languages are used in the test suite: DMIS, DEBNF, C++, YACC, and Lex. In addition, the test suite deals with files and commands. To help make it clear what sort of thing is being discussed, in this manual:

- DMIS code and keywords are shown in `this font`.
- File and directory names (including names of executable files) are shown in this font.
- C++ code (lines of code, class names, function names, etc.) is shown in *this font*.
- Commands typed in a command window or included in a Makefile are shown in ***this***

1. Kramer, T. R., "Automatic detection and replacement of syntactic constructs causing shift/reduce conflicts", Software, Practice and Experience, 40:387-404, 2010.

font.

- DEBNF code is shown in `this font`. So are `production`, `definition`, and `expression` when referring to DEBNF code.
- YACC code is shown in *this font*. So is *definition* when referring to YACC code.

Some terms are used in more than one context. “Production”, for example, is both a descriptive term used to talk about EBNF and the name of a C++ class. In these cases, the font used is intended to show the context.

1.5.2 Warning

A source of disorientation in maintaining the test suite is that three of the languages are used on two levels. YACC and Lex are used (1) to build a DEBNF parser and (2) to build a DMIS parser. C++ classes are defined both to represent EBNF and to represent DMIS. C++ is used both for the generator code and for the code written by the generator. You are sure to lose track periodically of which level you are working on. Sometimes you need to think simultaneously about two different levels using the same language. Hang in there.

1.6 Compilers

To build the test suite executables and libraries, you need a C++ compiler. Most Linux and Unix (such as Sun) systems come with a C++ compiler. You can download and use the Gnu C++ compiler for free from ftp://ftp.gnu.org/gnu/gcc or http://ftp.gnu.org/gnu/gcc. For Windows, you can download and use the Microsoft Visual C++ 2008 Express Edition for free from http://www.microsoft.com/express/vc.

You also need bison (a YACC compiler) and flex (a Lex compiler). Bison and flex are free software already installed on most Linux and unix (such as Sun) systems. For Linux and unix systems, they may be downloaded for free from ftp://ftp.gnu.org/gnu or http://ftp.gnu.org/gnu. For Windows systems they may be downloaded for free from http://gnuwin32.sourceforge.net or http://sourceforge.net/projects/gnuwin32.

2 The Big Ideas

The construction of the utilities in the test suite is based on several big ideas. These are used exclusively to deal with DMIS in the test suite, but they are applicable to any statement-based language whose syntax may be represented in EBNF.

Perhaps the biggest idea is that it is possible to generate large amounts of useful C++ code automatically. DMIS is a huge language. The EBNF file for DMIS is over 100 pages long. Over half a million lines of C++ source code (about 10,000 pages) are used in the test suite. That is too much for a small team. So, necessity has been the mother of invention.

The numbered items following are applications of this idea. Item 1 is not at all new. Item 2 has probably been done elsewhere. We believe items 3-6 are completely new.

1. A YACC-Lex parser for a target language may be generated automatically from an EBNF file describing the syntax of the language.

2. A set of C++ classes may be defined that represents EBNF and is rich enough to support automatic generation of both (1) C++ classes describing the target language described by an

EBNF file and (2) C++ source code for a variety of utilities for manipulating that language.

3. Methods of automatically analyzing the EBNF may be developed that recognize EBNF constructs that lead to shift/reduce conflicts in the automatically generated YACC file. Those constructs may be automatically modified to represent the same syntax but not produce conflicts in the YACC file.

4. The automatically generated parser may be built so that, as it parses a target language file, it builds a parse tree in terms of the automatically generated C++ classes. Moreover, the parse tree may be built in terms of classes generated from the unmodified EBNF even though the syntax recognition portions of the parser are built from modified EBNF.

5. A C++ data file may be built automatically describing the attributes and subclasses of the C++ classes representing the target language. This may be done in parallel with generating the classes. Copies of the C++ data file may be manually edited so that each copy represents the C++ code for a subset of the target language. Subsets of interest are those defining conformance modules of the language. Other C++ files with data on the C++ classes needed for building a conformance testing system may be also be generated automatically

6. Utilities that test conformance of target language files in various ways may be built semi-automatically in C++ using all the other automatically generated code. These utilities have a (relatively) small manually generated core that is included in a larger automatically generated C++ file.

3 In-line Documentation of Code

The manually generated source code in the test suite contains extensive in-line documentation. Most of the automatically generated source code does not.

3.1 Documentation in Automatically Generated Source Code

Much of the source code in the NDTS is generated automatically. Most of the automatically generated code has no in-line documentation.

One exception to this is dmis.hh, which has 14,770 lines of automatically generated documentation included in the 53,844 lines of the file. The C++ classes for DMIS are intended to serve two purposes. The first purpose is to support automatic generation of code for the utilities; for that, no documentation is required. The second purpose of the C++ classes is to be usable by DMIS system builders for building DMIS generators and consumers. It is necessary to include documentation in dmis.hh because that is only way a system builder can figure out what portion of DMIS each C++ class represents. Without it, the C++ classes would be unusable. The dmis.hh file starts with a page of documentation. After that, each class is documented. The automatically generated documentation for each class is of one of three types for (1) a parent class, (2) a class generated from a *production* with only one *definition*, or (3) a class generated from a *production* with two or more *definition*s. Examples of these are shown in Figure 1, Figure 2, and Figure 3, respectively.

```
/* dmisFreeStatement

This is a parent class.

*/
```

Figure 1. Automatically Generated Documentation - Parent Class

```
/* rotIncr

This is a class for the single definition of rotIncr.
It represents the following items:

INCR ',' rotOrient

*/
```

Figure 2. Automatically Generated Documentation - Single Definition

```
/* matDir_INNER

This is a derived class for one of the definitions of matDir.
It represents the following items:

INNER

*/
```

Figure 3. Automatically Generated Documentation - Multiple Definitions

3.2 Documentation of Hand-written Source Code

The in-line documentation of the hand-written code is very extensive.

3.2.1 C++ source code documentation.

Most of the .cc hand-written C++ source code files start with a 1-6 page description of what the file does. There is only one hand-written .hh file (generator/linuxSun/source/ebnf.hh) in the test suite, and it is about half documentation.

The generator/linuxSun/source/debnf2pars.y file, which has over 13,000 lines (250 pages), includes only about 150 lines of YACC. It is documented the same way as .cc files. The file is about half C++ code and half documentation.

To make it easy to find functions in a .cc file, in most cases they are declared in alphabetical order near the beginning of the file. Then they are defined in alphabetical order.

A uniform template has been used for documenting functions in hand-written C++ source code files. An example is shown in Figure 4. The first line has the name of the function. This is followed by a description of what the function returns and a list of functions that call the function.

Next is a description of what the function does. Depending on the complexity of what the function does, the description may be anywhere from one line to three pages.

Each argument of a function is given on a separate line, and a comment giving a description of the argument follows the argument.

To make it easy to know what local variables are used, the local variables of each function are declared at the beginning of the function, and each local variable is declared on a separate line. In some functions, each local variable is followed by a comment describing the variable.

The commenting in the .cc and .hh files has been done in such a way that it would be easy to modify it so it could be processed by the doxygen automatic documentation generation system. In particular, the documentation of the function arguments needs only an exclamation point to be added to each line.

```
/* findToken

Returned Value: int
If the text is the name of a token, this returns 1. Otherwise, it returns 0.

Called By:
   printCppClassPrinter
   printCppClassPrinterOpt1
   printCppClassPrinterOpt2
   printYaccProductions
   printYaccUnionAndTypes
   reviseSpelling
   selectProductions

If the token is found in the tokenNames, this also sets *n to the zero-based index of the position
at which the token was found in the sub-array of the tokenNames starting with the first letter of
the token.

*/

int findToken(    /* ARGUMENTS                             */
char * text,      /* text to look for                      */
int * n)          /* position at which found, if found     */

{
   char ** letterNames;
   int result;

…

}
```

Figure 4. Hand-written In-Line Documentation

3.2.2 EBNF documentation

The EBNF directory has two DEBNF files in it. One of the files (ebnf/dmisFull5.2.debnf) has no documentation in it and describes the same syntax as that given in the DMIS 5.2 spec.

The other file (ebnf/dmis.debnf) includes documentation and describes the syntax of files that have been produced by the preprocessor in the test suite, which is slightly different. Most of the documentation in this file (1) identifies the EBNF constructs that will produce shift/reduce errors in a YACC file that has the same structure, and (2) describes the alternate EBNF constructs that will be mirrored in the automatically generated YACC and do not produce conflicts. This file also contains hundreds of comments assigning names to classes and their attributes, but those are not considered to be documentation.

4 DEBNF C++ Classes and Parser

4.1 DEBNF C++ Classes

The generator/linuxSun/source/ebnfClasses.hh,.cc files contain C++ classes representing DEBNF. The classes support building a parse tree for DEBNF and doing data access on the tree (but not printing the DEBNF out again). In addition, one class, *stringCell*, is defined that is not needed for DEBNF but supports debnf2pars. The DEBNF classes include many attributes that support the functionality of debnf2pars but are not needed for dealing with the DEBNF parse tree.

The DEBNF classes include: *production*, *definition*, *expression*, and *optional*. For the first three of those, a list cell (*prodCell*, *defCell*, *expCell*) and a doubly linked list (*prodList*, *defList*, *expList*) are also defined. The lists cells each have *next*, *back*, and *data* attributes. Everything in the DEBNF classes is *public*, so data access functions are not needed. The list classes all have *pushBack*, *pushFront*, and *findLength* access functions. Each list type has a few other functions for special purposes (such as splicing in a sublist). The list classes allow complete control of the list structure and easy traversal in either direction.

4.1.1 Production class attributes

The *production* class has the following attributes:

```
defList * defs;          // the definitions of the production
bool endsInOptional;     // see below
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
```

FixTypeE is an enumeration defined in the *production* class whose allowed values are *fixListItemDeleted*, *fixListItemsInserted1*, *fixListItemsInserted2*, *fixNone*, *fixProdC*, and *fixProdCUser.*

The *endsInOptional* attribute has a default value of *false* and is set to *true* if and only if at least one *definition* of the *production* ends with an *optional* that starts with a comma.

4.1.2 Definition class attributes

The *definition* class has the following attributes:

```
char * className;        // class name for the definition
expList * expressions;   // expressions giving the definition
defList * newDefs;       // revised definitions to replace this
```

During processing by debnf2pars, the *newDefs* of each *definition* are set. Each *optional* in a *definition* leads to at least two *definition*s in the *newDefs*. Fixing shift/reduce conflicts also changes the *newDefs*. Printing rules in dmis.y is done primarily using the *newDefs*.

4.1.3 Expression class attributes

The *expression* class has the following attributes:

```
int theType;                 // see below
char * itemName;             // see below
char * attName;              // the name of the attribute that will represent the expression
optional * optValue;         // pointer to an optional; 0 unless theType is OPTIONAL
production * prodValue;      // pointer to a production; 0 unless theType is NONTERMINAL
```

TheType of an *expression* may be ENDLINE, KEYWORD, NONTERMINAL, ONECHAR, OPTIONAL, TERMINAL, TERMINALSTRING, or TWOCHAR. These have integer values #defined in debnf2parsYACC.cc.

The *itemName* of an *expression* is set to 0 if *theType* is OPTIONAL or TERMINALSTRING. The *itemName* is set for the following values of *theType*. For KEYWORD and NONTERMINAL, *itemName* is a string containing exactly the characters in the EBNF file. For ENDLINE, *itemName* is “ENDLINE”. For ONECHAR, *itemName* is a string containing the character between the apostrophes. For TERMINAL, *itemName* is a string containing the characters that were read converted to all upper case. For TWOCHAR, *itemName* is a string containing the two letters.

The *optValue* of an *expression* is 0 unless *theType* is OPTIONAL. In that case, *optValue* is set to point to an *optional*.

The *prodValue* of an *expression* is 0 unless *theType* is NONTERMINAL. In that case, *prodValue* is set to point to the *production* whose name is the *itemName*.

4.1.4 Optional class attributes

The *optional* class has the following attributes:

```
expList * expressions;   // the expressions in the optional
int digit;               // the multiplicity of the optional
```

4.2 DEBNF Parser

The DEBNF parser consists of a hand-written debnf2pars.lex file and about 150 lines of YACC in the debnf2pars.y file. The parser parses DEBNF into a parse tree using the DEBNF classes. In the current version of DEBNF, comments of the form `(*A=name*)` and `(*C=name*)` in the DEBNF file represent names for attributes and classes, respectively. If an attribute name for an `expression` that will become an attribute is not given in the DEBNF file, an attribute name will be assigned by the *findAttributeNames* function in debnf2pars.y. If a class name for a `definition` is not given in the DEBNF file, a class name will be assigned by the *findClassNames* function.

5 Test Suite Quality Control

5.1 Introduction

To be usable for conformance testing and be trusted by users, the NIST DMIS Test Suite must have very few bugs in it. A great deal of effort has been put into ensuring that there are few bugs. Whenever changes have been made in the test suite, tests have been performed to hunt for bugs. All bugs that have been found have been fixed. The test suite maintainers must continue to test and debug thoroughly when changes are made. If new functionality is introduced, new tests will need to be devised.

Five types of files are involved in maintaining the quality of the test suite:

- the text of DMIS 5.2
- the DEBNF file for DMIS
- the Excel Spreadsheet describing DMIS conformance classes
- test files that are DMIS 5.2 input files
- executable utilities and the C++ code from which they are built

There are no calibration artifacts maintained by other organizations that can be used by NIST to check the test suite. The five types of files are used to check each other. Every one of the five types of files may have errors in it. The last four types are currently being maintained only by NIST.

5.2 DMIS Input Test Files

Parser test files and system test files are described in sections 8 and 9 of the Users Manual. This section has additional information about the test files.

5.2.1 Parser test files vs. system test files

Parser test files are syntactically correct but may have semantic errors and may make no sense. System test files are syntactically and semantically correct so that it should be possible to execute them on commercial DMIS systems. Many of the system test files in the test suite do not produce any motion and therefore do nothing useful. The system test files that produce motion require using a physical part (or a simulation of one).

5.2.2 Parser test files

In Version 2.2.1, the parserTestFiles directory has test files only for full DMIS. In Version 2.1.5, that directory had test files for full DMIS plus the three levels of the prismatic AP.

In Version 2.2.1, however, parser test files may be produced automatically by the dmisTestFileReductor for any allowed set of conformance modules. The parser test files for full DMIS are used as input to the dmisTestFileReductor. To enable the dmisTestFileReductor to work, every non-comment line of every one of the 254 test files in the parserTestFiles/okIn directory was marked with a DMIS comment on the line above indicating the conformance modules needed to deal with the line. The marking was done by determining what conformance module the Excel spreadsheet requires for each item on the line and inserting the comment manually. For example, in the following two lines of DMIS code from featCone1.dmi, the comment `PM1 TW3` on the first line means that at least level 1 of the prismatic AP or level 3 of the thin-walled AP is needed in order for the second line to conform.

```
$$ PM1 TW3
F(c2)= FEAT/CONE,OUTER,CART,60,0,0,0,0,1,11.8
```

The conformance markings in the parser test files have been used for testing as described in Section 5.3.

In addition, the first line of each of the files in okIn is a comment giving the minimum conformance modules needed to handle a syntactically correct reduced version of the file containing the DMIS item after which the file is named. For example, the first line of planid1.dmi is “`$$ PM1 TW1 QI2`” because QI2 is needed for PLANID, and either PM1 or TW1 is needed for DMISMN. As another example, the first line of funcsBool.dmi is “`$$ PM2 TW2`” because boolean functions require level 2 of either PM or TW.

5.2.3 System test files

In Version 2.2.1, the systemTestFiles directory is still divided into subdirectories for full DMIS and the three levels of the prismatic AP. All the files have been updated to DMIS 5.2.

5.2.4 Other test files

A large set of large test files was provided to NIST several years ago by an industrial partner. These were written for DMIS 5.0. In earlier releases of the test suite, they were used to test the utilities. They were manually updated to DMIS 5.1 and used to test version 2.1.5. They have not yet been updated to DMIS 5.2, so they are not yet usable for testing the utilities in NDTS 2.2.1.

5.3 Testing Conformance Information

5.3.1 Testing without the dmisTestFileReductor

To check that the dmisConformanceRecorder works properly and to test that the parser test files were marked correctly, a test was devised in which each of over 300 parser test files had its conformance information inserted without using the dmisConformanceRecorder. Then each file was run through the dmisConformanceRecorder, which replaced the conformance information, and it was checked that the file was unchanged.

For the files in the parserTestFiles/okIn directory, the conformance information was inserted automatically by a utility named insertConf that read the line markings, found the maximum level of each conformance module used in the line markings, and inserted conformance information on the DMISMN line of the file. Conformance information was inserted manually on the DMISMN lines of the files in parserTestFiles/annexAIn, but that was done without marking each line.

Then the checkLevels script was written which checks that the conformance information already in each file of okIn and annexAIn is the same as the conformance information produced by the dmisConformanceRecorder. Of course many errors were found in the manual markings and a few errors were found in the dmisConformanceRecorder. Corrections were made and the checkLevels script was run repeatedly until no differences were reported.

During further development, the checkLevels script has been run repeatedly as a regression test.

This should continue to be done.

In addition, a testFullTester script was written for testing the dmisConformanceTester. This is a weaker test because, although it checks that no error or warning messages are generated except the expected ones, it does not compare the conformance information that is generated with the expected conformance information. Since almost all the software for both the dmisConformanceTester and the dmisConformanceRecorder is in the same file (dmisConformanceTester.cc), a more stringent test does not seem needed.

5.3.2 Testing with the dmisTestFileReductor

The dmisTestFileReductor is described in Section 7 of the Users Manual. It was developed after all the testing described in Section 5.3.1 was completed. The way the dmisTestFileReductor works is described in Section 9 of this manual. The dmisTestFileReductor enabled additional extensive testing of the dmisConformanceRecorder and the parser test files as follows.

First a command of the following sort was executed from the utilities/linux/full directory

../bin/dmisTestFileReductor runOkFull outgoing PM3

That command runs all the files in the parserTestFile/okIn directory through the dmisTestFileReductor and, if the first line of a test file indicates that level 3 of the prismatic AP is high enough to warrant producing a reduced test file, writes a reduced test file of the same name in the utilities/linux/full/outgoing directory.

Then the checkRed script file is run from the utilities/linux/full directory. The checkRed script is the same as the checkLevels script described above with two changes:

- The checkRed script reads files from the outgoing directory only.
- The checkRed script checks whether a file exists before processing it.

In other words, if a file exists in the outgoing directory, a copy of it is made, the copy is run though the dmisConformanceRecorder, and it is checked that the file output from the dmisConformaceRecorder is identical to the input file. Thus, the conformance requirements of each file are derived in two completely independent ways, (1) by using the manually generated markings in the test files plus the dmisTestFileReductor and, (2) by running the dmisConformanceRecorder, and it is checked that the two methods get the same results.

Additional testing has included replacing the PM3 on the command line above with the following single AP levels or combinations of AP and addenda. The parentheses below were not used in the argument list; they are just to make the alternative sets arguments easy to read.

(PM2), (PM1), (TW3), (TW2), (TW1),

(PM3 RY3 MC3 CT3 IP3 QI3 MU3 SF3), (PM2 RY2 MC2 CT2 IP2 QI2 MU2 SF2),
(PM1 RY1 MC1 CT1 IP1 QI1 MU1 SF1),

(TW3 RY3 MC3 CT3 IP3 QI3 MU3 SF3), (TW2 RY2 MC2 CT2 IP2 QI2 MU2 SF2),
(TW1 RY1 MC1 CT1 IP1 QI1 MU1 SF1).

Between tests, all files in the outgoing directory were deleted.

Tests of this sort should be repeated if the test suite is modified.

5.4 Testing the C++ Classes and the Parser

As described in Section 3.2.1 of the Users Manual, the testFullParser (or testFullParser.bat) executable script processes 322 DMIS input files. Each DMIS file is parsed in to build a parse tree. TestFullParser checks that the messages produced by the parser are identical to the expected messages. If there are no errors in parsing the file, it is printed out again from the parse tree using the *printSelf* functions in dmis.cc. Then another version of the input file is made by reformatDmis (a utility built using lex), which also reads and reprints the DMIS file, but without using the parser or the C++ classes. ReformatDmis formats the files it prints the same way as the *printSelf* functions. The two reprinted versions are compared to check that they are identical.

Running testFullParser successfully demonstrates that:

- The dmisLex.cc, dmisYACC.cc, and dmis.cc files work correctly, and hence the generator, bison, and flex work correctly.
- The C++ classes for DMIS can represent all of DMIS (since the test files cover all of DMIS).

In addition, running testFullParser every time the software is rebuilt serves as a regression test.

6 Building Utility Components and Utilities

This section describes how the utilities and the source code for them are built in Linux. Building them for SunOS is identical except for the directory names. It is not necessary to regenerate source code for Sun since Linux and Sun use the same source code. Building source code for Windows could be done similarly, but has not been. In practice, for Windows, the source code is built by automatically editing the Linux source code lightly by changing slash (ASCII 47) to backslash (ASCII 92), .hh to .h, and .cc to .cpp for several files.

Most of the steps of building the source code are automated using the Makefile in utilityComponents/linuxSun. Executables and object files for Linux are stored in utilityComponents/linuxSun/binLinux and utilityComponents/linuxSun/ofilesLinux, respectively. Executables and object files for Sun are stored in utilityComponents/linuxSun/binSun and utilityComponents/linuxSun/ofilesSun.

6.1 Generating the Source Code

By source code we mean Lex (.lex) and YACC (.y) files as well as all C++ files (.cc and .hh). The procedure for generating source code is shown in Figure 5.

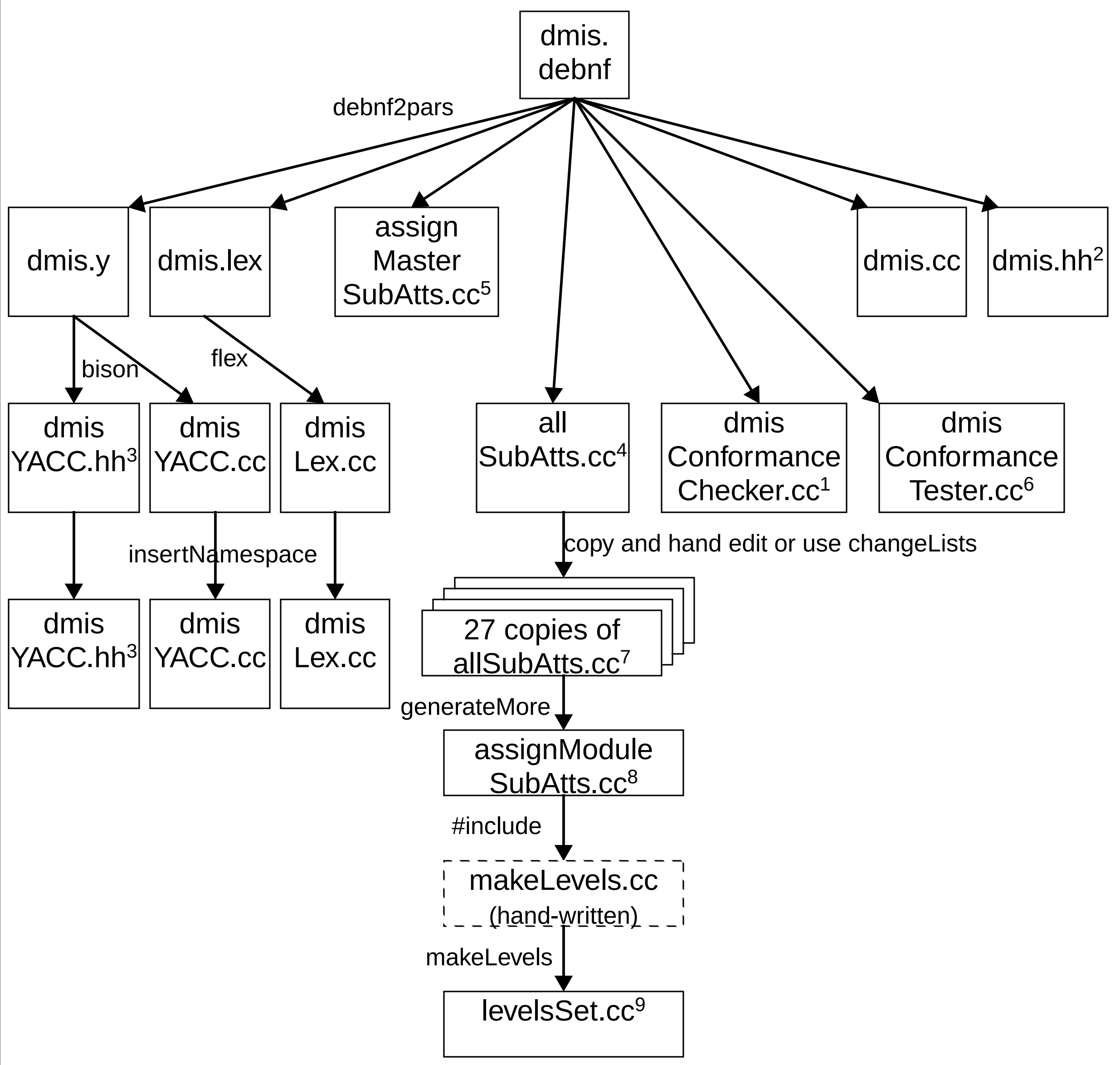


Figure 5. Automatic Generation of Source Code

1. dmisConformanceChecker.cc #includes dmisConformanceCheckerStart.cc, a hand-written file.
2. dmis.hh is #included in dmis.cc, dmisYACC.cc, dmisLex.cc, dmisConformanceChecker.cc (indirectly), dmisConformanceTester.cc (indirectly), and dmisParser.cc.
3. dmisYACC.hh is #included in dmisLex.cc.
4. allSubAtts.cc is #included in makeLevels.cc.
5. assignMasterSubAtts.cc is #included in dmisConformanceChecker.cc, dmisConformanceTester.cc, and makeLevels.cc.
6. dmisConformanceTester.cc #includes dmisConformanceCheckerStart.cc, a hand-written file.
7. The 27 copies of allSubAtts.cc (each with a different name) are read by generateMore and #included in makeLevels.cc, dmisConformanceChecker.cc (indirectly), and dmisConformanceTester.cc (indirectly).
8. assignModuleSubAtts.cc is #included in makeLevels.cc and dmisConformanceChecker.cc.
9. levelsSet.cc is #included in dmisConformanceTester.cc.

6.1.1 make linuxSource

The action begins by giving a

make linuxSource

command in the utilityComponents/linuxSun directory. This does the following:

- calls ***dos2unix…*** to copy dmis.debnf from the ebnf directory into utilityComponents/linuxSun and ensure it has unix style line endings (not shown on Figure 5).
- calls binLinux/debnf2pars. This generates the eight files shown coming directly from dmis.debnf in Figure 5. Details of how debnf2pars works are given in Section 13.
- moves seven of the eight files (excluding allSubAtts.cc) into the utilityComponents/linuxSun/source directory
- calls ***flex…***, which processes dmis.lex and generates dmisLex.cc (in the utilityComponents/linuxSun directory). This is shown at the upper left of Figure 5.
- calls ***bison…***, which processes dmis.y and generates dmisYACC.cc and dmisYACC.hh (in the utilityComponents/linuxSun directory). This is shown at the upper left of Figure 5.
- calls ***binLinux/insertNamespace***, which makes copies of dmisLex.cc, dmisYACC.cc and dmisYACC.hh, inserting namespace NDTS declarations while it copies, and puts the copies in the utilityComponents/linuxSun/source directory. This is shown at the middle left of Figure 5.
- deletes the copies of dmisLex.cc, dmisYACC.cc, dmisYACC.hh, and dmis.debnf that are in utilityComponents/linuxSun.

Namespace declarations (NDTS for dmis.hh and dmis.cc, NDTU for the four utilities and their helper files) are inserted directly in the .cc and .hh files generated by debnf2pars. See Section 3 of the System Builders Manual for a discussion of these two namespaces.

The dmis.hh file is #included directly or indirectly, in all the .cc files shown on Figure 5, plus dmisParser.cc

The dmisYACC.cc, dmisYACC.hh, dmisLex.cc, and dmis.cc files contain everything needed for parsing DMIS input files, building a parse tree, printing DMIS input files from a parse tree, and accessing everything in the parse tree.

The dmisConformanceChecker.cc file contains much of the source code for the dmisConformanceChecker utility. The dmisConformanceTester.cc file contains much of the source code for the dmisConformanceTester and the dmisConformanceRecorder utilities.

The allSubAtts.cc and assignMasterSubAtts.cc files contain C++ data describing the subclasses and attributes of the C++ classes representing DMIS. These two files as well as assignModuleSubAtts.cc and levelsSet.cc contain only array declarations and array assignments.

6.1.2 Editing allSubAtts.cc

The allSubAtts.cc file contains declarations of arrays that give either the subclasses of a class or the attributes of a class (none of the C++ classes for DMIS have both subclasses and attributes). Examples of both kinds of array declaration from that file are shown in Figure 6.

```
const char * aboveBelowSubs[] =
  {
    "aboveBelowSubs",
    "aboveBelow_ABOVE",
    "aboveBelow_BELOW",
    0
  };

const char * algdefStmAtts[] =
  {
    "algdefStmAtts",
    "a_vaLabel",
    "a_algdefMinor",
    0
  };
```

Figure 6. Sample AllSubAtts.cc Declarations

The first time the allSubAtts.cc file was generated, it was copied into the linuxSun/source directory. Then a copy was made in the linuxSun/source directory for each of the two application protocols (APs) and seven addenda of the Excel spreadsheet giving the DMIS conformance classes (a total of 9 copies). The names of the copies are all of the form <initials>3Lists.cc, where the initials stand for the AP or addendum and the 3 stands for level 3. For example p3Lists.cc is for level 3 of the prismatic AP.

The p3Lists.cc file was edited by adding one line (*const char ** p3Lists[1600] = {0};*) and inserting *p3* as the first two characters in the name of every array. For example the array names in Figure 6 were changed to *p3aboveBelowSubs* and *p3algdefStmAtts*. That much was easily done in emacs using search and replace. The other 8 files were edited in a similar manner.

Then the p3Lists.cc file was edited by commenting out sections of the file. To keep the file very regular (so that it is amenable to automatic editing and is easily compared with other files), only comment character lines of the form */* xx3* or **/* were inserted. The *xx3* just shown is either *p3* (indicating that the material being commented out is not in the prismatic AP at level 3 or in any addendum at any level) or the initials and level of the addendum (*rt3*, for example) that includes the commented out material – which is therefore not in prismatic level 3. Doing this editing required constant study of the Excel spreadsheet defining the conformance modules.

The p2Lists.cc file was created by making a copy of the p3Lists.cc file and then changing *p3* to *p2* in the array names and adding the comment character lines */* p2* or **/* to comment out subclasses and attributes that are in prismatic level 3 but not in prismatic level 2. The p1Lists.cc file was created by making a copy of the p2Lists.cc file and then doing analogous editing.

The tw3Lists.cc, tw2Lists.cc, and tw1Lists.cc files were created for the thin-walled AP by similar procedures. This completed preparing the 6 files needed for the APs.

The first addendum file, rt3Lists.cc, was edited by adding one line (*const char ** rt3Lists[1600] = {0};*) and inserting *rt3* as the first three characters in the name of every array. Then the comment character lines */* rt3* or **/* were inserted on separate lines to comment out everything except what is explicitly permitted in level 3 of the rotary table addendum. The rt2Lists.cc file was created by copying rt3Lists.cc, changing *rt3* to *rt2* at the beginning of array names, and inserting */* rt2* and **/* to comment out material included at level 3 of the rotary table addendum but not at level 2. The rt1Lists.cc file was created similarly from rt2Lists.cc. The other 18 files for the addenda were prepared analogously.

Editing the 27 copies of allSubAtts.cc took about 3 weeks. The file has 9123 lines. Editing adds lines, so the total number of lines in the copies approaches 270,000.

Since these files are precious and need to be modified if dmis.debnf changes, copies of them should be saved in a separate directory not part of the normal distribution of the NDTS. The utilityComponents/linuxSun/saveLists directory has been used for that purpose.

If the dmis.debnf file is edited for any reason, it is necessary to execute ***make linuxSource*** again. This will produce new source code files as described in Section 6.1.1. In particular, the linuxSun/allSubAtts.cc file will be new. There are 27 edited copies of that file, and all of them need to be changed when allSubAtts.cc changes. To deal with this, a copy of allSubAtts.cc is kept in linuxSun/source, the differences between the new and old copies are found using diff, and the changeLists.cc file is manually edited so that it will make those changes. The file includes *insertLines* and *deleteLines* functions. The calls to those functions are what is modified when the file is edited. When changeLists is compiled and executed, what it does is read the 27 xxLists.cc files in the linuxSun/saveLists directory, make the same changes in every file, and put the new copies in the linuxSun/source directory. After that, it is necessary to check whether the comment characters in each file are still appropriately placed and change them manually if not.

6.1.3 More automatic generation

The final step of automatic source code generation is to give the command

make moreLinuxSource

from the from the utilityComponents/linuxSun directory.

The first thing that ***make*** command does is to execute ***binLinux/generateMore***. GenerateMore reads the 27 xxLists.cc files and writes the assignModuleSubAtts.cc file.

Next the ***make*** command compiles the (hand-written) file linuxSun/source/makeLevels.cc and puts the executable into binLinux. MakeLevels.cc is only 541 lines long, but it #includes the hundreds of thousands of lines of code in the 27 xxLists.cc files, allSubAtts.cc, assignModuleSubAtts.cc, and assignMasterSubAtts.cc. The executable cannot be compiled until assignModuleSubAtts.cc has been generated.

Finally, the ***make*** command executes ***binLinux/makeLevels***, which generates the levelsSet.cc file.

6.2 Building the Library

The library file (dmis.a for Linux and Sun, dmis.lib for Windows) is linked in to all the utilities. The library file combines dmisYACC.cc, dmisLex.cc, dmis.cc, and dmis.hh. Hence, the library includes:

- the *parseDmis* function that runs the parser and builds a parse tree,
- the *printTree* function that prints a DMIS input file from a parse tree,
- the constructors and destructors for the C++ classes for DMIS, and
- all the access functions for putting information into the C++ classes or getting it out.

6.2.1 Linux

The libLinux/dmis.a archive file is built by executing

```
make libLinux/dmis.a
```

from the utilityComponents/linuxSun directory. The ***make*** command calls the compiler to generate the object files dmisYACC.o, dmisLex.o, and dmis.o and saves them in the utilityComponents/linuxSun/ofilesLinux directory. Then it uses **ar** to archive them in dmis.a.

6.2.2 Sun

The libSun/dmis.a archive file is built by executing

```
make libSun/dmis.a
```

from the utilityComponents/linuxSun directory. The ***make*** command calls the compiler to generate the object files dmisYACC.o, dmisLex.o, and dmis.o and saves them in the utilityComponents/linuxSun/ofilesSun directory. Then it uses **ar** to archive them in dmis.a.

6.2.3 Windows

The utilityComponents\windows\dmisClasses\Release\dmis.lib file was built using the Microsoft Visual C++ 2008 Express Edition. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A). The “project name” for the library is dmisClasses. A difference from the instructions in Appendix A is that the “Application type” “Static library” should be selected rather than “Console Application”.

6.3 Building the dmisParser

The dmisParser is built from three pieces: dmis.a (or dmis.lib), dmisParser.cc, and dmisParserDriver.cc. Almost all the work done by the executable is done by the *parseDmis* function from dmis.a. The dmisParserDriver.cc file is only 8 lines long, and all it does it call the *runParser* function from dmisParser.cc, which is a hand-written166-line file. All that function does is run *parseDmis* once if the given file is a DMIS input file, or many times if the given file is a list of the names of DMIS input files.

6.3.1 Linux

The utilities/linux/bin/dmisParser executable file is built by getting into the utilityComponents/linuxSun directory and executing

```
make ../../utilities/linux/bin/dmisParser
```

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesLinux/dmisParser.o and utilityComponents/linuxSun/ofilesLinux/dmisParserDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/linux/bin/dmisParser.

6.3.2 Sun

The utilities/sun/bin/dmisParser executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/sun/bin/dmisParser

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesSun/dmisParser.o and utilityComponents/linuxSun/ofilesSun/dmisParserDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/sun/bin/dmisParser.

6.3.3 Windows

The utilities\windows\bin\dmisParser.exe executable file was built in the utilityComponents\windows\dmisParser\Release directory using the Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).

6.4 Building the dmisConformanceChecker

The dmisConformanceChecker is built from dmis.a, the 27 xxLists.cc files, and six other files in utilityComponents/linuxSun/source:

- dmisConformanceCheckerDriver.cc
- dmisConformanceChecker.cc
- dmisConformanceCheckerStart.cc
- dmis.hh
- assignModuleSubAtts.cc
- assignMasterSubAtts.cc

dmisConformanceChecker.cc #includes dmisConformanceCheckerStart.cc, assignMasterSubAtts.cc, and assignModuleSubAtts.cc.

dmisConformanceCheckerStart.cc #includes the 27 xxLists.cc files and dmis.hh.

6.4.1 Linux

The utilities/linux/bin/dmisConformanceChecker executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/linux/bin/dmisConformanceChecker

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesLinux/dmisConformanceChecker.o and utilityComponents/linuxSun/ofilesLinux/dmisConformanceCheckerDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/linux/bin/dmisConformanceChecker.

6.4.2 Sun

The utilities/sun/bin/dmisConformanceChecker executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/sun/bin/dmisConformanceChecker

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesSun/dmisConformanceChecker.o and utilityComponents/linuxSun/ofilesSun/dmisConformanceCheckerDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/sun/bin/dmisConformanceChecker.

6.4.3 Windows

The utilities\windows\bin\dmisConformanceChecker.exe executable file was built in the utilityComponents\windows\dmisConformanceChecker\Release directory using the

Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).

6.5 Building the dmisConformanceRecorder

The dmisConformanceRecorder is built from dmis.a, the 27 xxLists.cc files, and six other files in utilityComponents/linuxSun/source:

- dmisConformanceRecorderDriver.cc
- dmisConformanceTester.cc
- dmisConformanceTesterStart.cc
- dmis.hh
- assignModuleSubAtts.cc
- assignMasterSubAtts.cc

dmisConformanceTester.cc #includes dmisConformanceTesterStart.cc, assignMasterSubAtts.cc, and assignModuleSubAtts.cc.

dmisConformanceTesterStart.cc #includes the 27 xxLists.cc files and dmis.hh.

6.5.1 Linux

The utilities/linux/bin/dmisConformanceRecorder executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/linux/bin/dmisConformanceRecorder

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesLinux/dmisConformanceTester.o and utilityComponents/linuxSun/ofilesLinux/dmisConformanceRecorderDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/linux/bin/dmisConformanceRecorder.

6.5.2 Sun

The utilities/sun/bin/dmisConformanceRecorder executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/sun/bin/dmisConformanceRecorder

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesSun/dmisConformanceTester.o and utilityComponents/linuxSun/ofilesSun/dmisConformanceRecorderDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/sun/bin/dmisConformanceRecorder.

6.5.3 Windows

The utilities\windows\bin\dmisConformanceRecorder.exe executable file was built in the utilityComponents\windows\dmisConformanceRecorder\Release directory using the Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).

6.6 Building the dmisConformanceTester

The dmisConformanceTester is built from dmis.a, the 27 xxLists.cc files, and six other files in utilityComponents/linuxSun/source:

- dmisConformanceTesterDriver.cc
- dmisConformanceTester.cc

- dmisConformanceTesterStart.cc
- dmis.hh
- assignModuleSubAtts.cc
- assignMasterSubAtts.cc

dmisConformanceTester.cc #includes dmisConformanceTesterStart.cc, assignMasterSubAtts.cc, and assignModuleSubAtts.cc.

dmisConformanceTesterStart.cc #includes the 27 xxLists.cc files and dmis.hh.

6.6.1 Linux

The utilities/linux/bin/dmisConformanceTester executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/linux/bin/dmisConformanceTester

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesLinux/dmisConformanceTester.o and utilityComponents/linuxSun/ofilesLinux/dmisConformanceTesterDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/linux/bin/dmisConformanceTester.

6.6.2 Sun

The utilities/sun/bin/dmisConformanceTester executable file is built by getting into the utilityComponents/linuxSun directory and executing

make ../../utilities/sun/bin/dmisConformanceTester

The ***make*** command calls the compiler to generate utilityComponents/linuxSun/ofilesSun/dmisConformanceTester.o and utilityComponents/linuxSun/ofilesSun/dmisConformanceTesterDriver.o. Then it links those files with dmis.a and puts the executable file in utilities/sun/bin/dmisConformanceTester.

6.6.3 Windows

The utilities\windows\bin\dmisConformanceTester.exe executable file was built in the utilityComponents\windows\dmisConformanceTester\Release directory using the Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).

6.7 Editing the C++ Code in the UtilityComponents Directory

The 27 xxLists.cc files can be edited by inserting or removing comment characters (carefully, so that the syntax is still correct C++). This would redefine the conformance modules. As long as generateMore is rerun after the editing is done (as described in Section 6.1) and everything is recompiled, the utilities should behave as intended.

Parts of the following manually written C++ files in the utilityComponents directory can be edited with some confidence that the results will be as intended. Wherever that code works with automatically generated code, it will be important to understand what is going on before making changes.

- changeLists.cc
- dmisConformanceCheckerDriver.cc
- dmisConformanceCheckerStart.cc
- dmisConformanceRecorderDriver.cc

- dmisConformanceTesterDriver.cc
- dmisConformanceTesterStart.cc
- insertNamespace.cc
- makeLevels.cc

Because there are a lot of dependencies in the code, editing any other code in the utilityComponents directory is likely to produce undesirable results. In particular, the C++ code generated by flex and bison that does lexical analysis and parsing (dmisLex.cc, dmisYACC.cc, and dmisYACC.hh) has a lot of giant switch statements and arrays of numbers and names. Hand-editing that part of the code is effectively impossible and should not be attempted.

7 How the dmisConformanceChecker Works

7.1 Overview

The dmisConformanceChecker takes as arguments the name of a file plus the names of zero to eight conformance modules. The number of arguments (*argc*) and the array of arguments (*argv*) are passed on to the *checkDmis* function.

In *checkDmis*, if any names of conformance modules are given as arguments, the *masterSubAtts* array of arrays for evaluating conformance is set up by calling *prepareLists* as described in Section 7.2, and the names of DMIS statements allowed by the conformance modules are found by calling *findStatements*.

If the file name provided as an argument (call the file argFile) ends in .dmi, *checkDmis* calls the *checkOneFile* function to process the file. If the name of argFile does not end in .dmi, argFile should be a list of the names of DMIS input files. In this case, *checkDmis* calls *checkManyFiles*, and *checkManyFiles* calls the *checkOneFile* function to process each of the files in the list. In both cases, the number of conformance modules is passed to *checkOneFile* as an argument.

The *checkOneFile* function:

- calls *parseDmis* to parse the file, which always builds a list of the DMIS statements found in the file and builds a parse tree if there are no errors.
- reports the number of errors and warnings found by the parser. If there are no errors or warnings, the file conforms to full DMIS. If there are warnings but no errors, the file probably does not conform to full DMIS, but it might.
- calls *analyzeItems* to go through the list of DMIS statements and, for each DMIS statement, add one to the DMIS statement counter for that kind of statement. There is a counter for each kind of DMIS statement.
- if the number of conformance modules is non-zero, calls *check_inputFile* to make a conformance check as described in Section 7.3.
- reports the number of conformance errors.

Back in the *checkDmis* function, If argFile is a list of names of DMIS input files, after all the files have been parsed, a report on the uses of statements is made as described in Section 7.3. If argFile is a .dmi file, no report on statement uses is made.

7.2 Arrays for Conformance Checking

The central actors in the methods used to check the DMIS input file against the named modules are sets of corresponding arrays of strings. All elements of a set of arrays are for checking the same C++ class and have the same first string (which is the name of the class followed by either *Subs* or *Atts*). The first string is used to find the arrays belonging to the set. The remaining elements in each array give the names of either (1) allowed subclasses of the class or (2) allowed attributes of the class. Each of the 27 files described in Section 6.1.2 representing the conformance modules defines zero or one member of each set of corresponding arrays. For example, one set of arrays is those whose first element is *"aclratMinorSubs"*. The p2Lists.cc file defines one member of that set, an array named *p2aclratMinorSubs*, which is *{"aclratMinorSubs", "aclratMeas", "aclratPos", "aclratHedRot", "aclratHedMeas", 0}*. The rt3Lists.cc file defines another member of the set, the array *rt3aclratMinorSubs*, which is *{"aclratMinorSubs", "aclratRot", 0}*. The rt2Lists.cc file does not define a member of the set. The arrays are terminated with zeros so that it is not necessary to keep track of the length of the arrays.

To prepare to test a DMIS file against a set of conformance modules, the members of each set of matching arrays from the xxLists.cc files representing those modules are combined in an array whose name is the first string from the corresponding modules.

For example, if the PM2 (an alias for p2) and RY3 (an alias for rt3) conformance modules are selected, an array named *aclratMinorSubs* is created, which is *{"aclratMinorSubs", "aclratMeas", "aclratPos", "aclratHedRot", "aclratHedMeas", "aclratRot", 0}*. This array is declared in dmisConformanceChecker.cc as an empty array big enough to contain all possible names of subclasses of the *alcratMinor* class, plus the name of the array at the front and a 0 at the end. When *prepareLists* runs with PM2 and RY3 as arguments, it populates the empty *aclratMinorSubs* array by inserting in it all the strings in *p2aclratMinorSubs* and *rt3aclratMinorSubs*. The same sort of procedure is used for all other matching sets of arrays.

Matching the members of each set of corresponding arrays is facilitated by putting them into large arrays in alphabetical order. That is what the *assignMasterSubAtts* function (defined in assignMasterSubAtts.cc) and the *assignModuleSubAtts* function (defined in assignModuleSubAtts.cc) do. The *prepareLists* function calls those two functions. Calling *assignMasterSubAtts* in the dmisConformanceChecker sets up the array (*masterSubAtts*) containing all the almost empty arrays (since they are defined with only a name and terminating zero in dmisConformanceChecker.cc). Calling *assignModuleSubAtts* sets up 27 fully populated arrays of arrays (*p1Lists*, *p2Lists*, … , *unc3Lists*) representing the conformance modules. Then *prepareLists* selects the 1 to 8 arrays of arrays to be combined (from the 27) and calls *combineLists* to combine them as described above.

The procedures described above are summarized conceptually in Table 1, which diagrams the situation when two conformance modules (p2 and rt3, for example) are combined. The first two columns represent the *p2Lists* and *rt3Lists* arrays. The third column represents the *masterSubAtts* array after *prepareLists* is finished executing. In each row, the first string in each array is the same, and the array in the third column contains all the entries from the arrays in the first two columns. The strings in the arrays are not the actual strings that would be used.

Table 1. Combining Conformance Modules

p2Lists	rt3Lists	masterSubAtts
p2a = {“a”, “a1”, “a2”, 0}		a = {“a”, “a1”, a2”, 0}
		b = {“b”, 0}
p2c = {“c”, “c1”, “c3”, 0}	rt3c = {“c”, “c2”, 0}	c = {“c”, “c1”, “c3”, “c2”, 0}
	rt3d = {“d”, “d1”, 0}	d = {“d”, “d1”, 0}
p2e = {“e”, “e1”, “e2”, 0}	rt3e = {“e”, “e1”, 0}	e = {“e”, “e1”, “e2”, 0}
		f = {“f”, 0}

It is an oddity of the software that the sets of corresponding arrays of strings (the rows of the table), which are important, are never built or named, while the alphabetically ordered arrays of arrays (the columns of the table) are built, named, and processed, even though they exist only to make it easy to match corresponding arrays.

After the *masterSubAtts* array is built, it is used in the *findStatements* function to build the *theStatements* array which contains the names of all C++ classes that represent DMIS statements and are included in the conformance class described by *masterSubAtts*. See the documentation of the *findStatements* function in dmisConformanceCheckerStart.cc for details. The *theStatements* array is used in the *reportSummary* function.

7.3 Checking Conformance

The *check_inputFile* function is called to start the conformance checking. That function is the root of a tree of *check_xxx* functions which traverse the parse tree and check whether the subclasses and attributes present in the parse tree are allowed in the conformance class. The dmisConformanceChecker.cc file contains 1632 automatically generated *check_xxx* functions – one for every C++ class in dmis.hh that has subclasses or attributes. There are two types of *check_xxx* functions, one for classes with attributes and one for classes with subclasses. Classes which have neither attributes nor subclasses do not need a checking function.

As an example of how the *check_xxx* functions work, the *check_aclratMinor function* is shown in Figure 7. The function will be called if an instance of the *aclratMinor* class (*a_aclratMinor*) has been found in the parse tree. An instance of *aclratMinor* must be an instance of one of 7 subclasses (*aclratMeas*, *aclratPos*, *aclratRot*, *aclratScan*, *aclratHedRot*, *aclratHedMeas*, *aclratHedScan*). The function checks for each of the seven subclasses until the correct one is found. When the correct subclass is found, the function checks whether the name of that subclass is included in the *aclratMinorSubs* array. If not, *warnSub* is called, which adds one to *numErrors*, prints a warning message, and prints *a_aclratMinor*. If that subclass is allowed and the subclass has either attributes or subclasses, the checking function for that subclass is called.

The *check_xxx* functions for classes with attributes work the same way, except they use a *get_* function to test each attribute and use *warnAtt*, rather than *warnSub* if there is an error. Also, if the value of an attribute is a list, a *for* loop is used to call the checking functions for all elements of the list. As an example, the *check_boundFeat* function is shown in Figure 8.

There are two slightly odd things about the *check_xxx* functions for classes with attributes. First, checking for a non-zero return value from *get_* functions works both for pointers and for booleans; that feels strange but is convenient. It is also the reason why almost all attributes are either pointers or booleans. Second, the names of all attributes for which the *get_* function returns a non-zero value are checked against the array of strings for the class regardless of whether or not an attribute is optional. For required attributes, this is a waste of time since the name must be present in the array of strings and the *get_* function must return non-zero (otherwise, the parser would have signalled an error). The extra checks have not been eliminated for two reasons. First, they serve as a check on the correctness of the dmisConformanceChecker; if a required attribute is signalled as not being allowed in a conformance class, there is a bug somewhere. Second, eliminating the redundancy would require making debnf2pars more complex.

7.4 Reporting the Results

If argFile is a list of .dmi files and at least one conformance module is used as an argument, then the *reportSummary* function is called by *checkDmis* to report the results. The report written by *reportSummary* has separate sections for (1) the number of times each DMIS statement in the conformance class that was used at least once was used, (2) the names of DMIS statements in the conformance class that were not used, and (3) the names of DMIS statements not in the conformance class that were used. It is necessary to know which statements are in the conformance class in order to produce the report. *ReportSummary* uses the *theStatements* array to determine that.

If argFile is a list of .dmi files but no conformance module is used as an argument, implying that checking should be done against full DMIS, then the *reportSummaryFull* function is called by *checkDmis* to report the results. There are no DMIS statements not in full DMIS, so *reportSummaryFull* reports only (1) the number of times each DMIS statement that was used at least once was used, (2) the names of DMIS statements that were not used.

Both *ReportSummary* and *reportSummaryFull* are generated automatically. They are found at the end of the dmisConformanceChecker.cc file.

```
void check_aclratMinor(
   aclratMinor * a_aclratMinor)
{
  if (isA(a_aclratMinor, aclratMeas))
    {
      if (!findString("aclratMeas", aclratMinorSubs))
        {warnSub("aclratMeas", "aclratMinor", a_aclratMinor);}
      else
        check_aclratMeas(dynamic_cast<aclratMeas *>(a_aclratMinor));
      return;
    }
  if (isA(a_aclratMinor, aclratPos))
    {
      if (!findString("aclratPos", aclratMinorSubs))
        {warnSub("aclratPos", "aclratMinor", a_aclratMinor);}
      else
        check_aclratPos(dynamic_cast<aclratPos *>(a_aclratMinor));
      return;
    }
  if (isA(a_aclratMinor, aclratRot))
      …
  if (isA(a_aclratMinor, aclratScan))
      …
  if (isA(a_aclratMinor, aclratHedRot))
      …
  if (isA(a_aclratMinor, aclratHedMeas))
      …
  if (isA(a_aclratMinor, aclratHedScan))
      …
}
```

Figure 7. check_aclratMinor Function

```
void check_boundFeat(
   boundFeat * a_boundFeat
{
   if (a_boundFeat->get_fLabel())
      {
         if (!findString("a_fLabel", boundFeatAtts))
            {warnAtt("a_fLabel", "boundFeat", a_boundFeat);}
         else
            check_fLabel(a_boundFeat->get_fLabel());
      }
   if (a_boundFeat->get_featureList())
      {
         if (!findString("a_featureList", boundFeatAtts))
            {warnAtt("a_featureList", "boundFeat", a_boundFeat);}
         else
            {
               std::list<featureLabel *> * theList;
               std::list<featureLabel *>::iterator iter;
               theList = a_boundFeat->get_featureList();
               for (iter = theList->begin(); iter != theList->end(); iter++)
                  check_featureLabel(*iter);
            }
      }
}
```

Figure 8. check_boundFeat Function

8 How the dmisConformanceTester and dmisConformanceRecorder Work

Descriptions of what the dmisConformanceRecorder and the dmisConformanceTester do and how to use them are given in Sections 5 and 6 of the Users Manual. Read that first.

The code for both the dmisConformanceTester and the dmisConformanceRecorder is almost all in the dmisConformanceTesterStart.cc and dmisConformanceTester.cc files in the utilityComponents/linuxSun/source directory. There are separate drivers for the two utilities, dmisConformanceTesterDriver.cc and dmisConformanceRecorderDriver.cc, in the same directory. The driver files are only 8 lines long. The dmisConformanceTesterDriver.cc file calls the *testDmis* function defined in dmisConformanceTesterStart.cc., while the dmisConformanceRecorderDriver.cc file calls the *recordConformance* function defined in the same file. Both start by calling *parseDmis*, and both stop without doing anything more if there are parse errors.

The heart of both the dmisConformanceTester and the dmisConformanceRecorder is the *check_inputFile* function. That function makes extensive use of the *levels struct*s defined in the levelsSet.cc file.

8.1 LevelsSet.cc

The levelsSet.cc file contains 4231 *levels struct*s, one for each subclass or attribute named in each array of masterSubAtts.cc as fully populated by allSubAtts.cc. Each of the 9 entries of a *levels struct* is an integer representing the level required in a conformance module in order for the named subclass or attribute to be in conformance. The first two entries are for the prismatic (p or PM) and thin-walled (tw or TW) APs, and they are alternatives. The rest of the entries in a *levels struct* are for the seven addenda in the following order:

- rotary table (rt and RT)
- multi carriage (mc and MC)
- contact scanning (cs and CT)
- in-process verification (ipv and IP)
- quality information systems (qis and QI)
- measurement uncertainty (unc and MU)
- soft gaging (sga and SF).

For example, the first line of levelsSet.cc is

levels aboveBelowSubs_aboveBelow_ABOVE = {3,3,0,0,0,0,0,0,0};

That means that level 3 of the prismatic AP or level 3 of the thin-walled AP is needed in order for a conforming DMIS input file to have an instance of the *aboveBelow_ABOVE* subclass named in the *aboveBelowSubs* array. The zeros on the line above mean that the subclass is not in any of the addenda.

If there are non-zero entries for both an AP and an addendum, that means that both are required (which happens in many cases). There is one exception to that rule: *intFuncPtdataAtts_a_faLabel* is allowed at level 2 of CT and at level 3 of PM and TW. In this case, CT,2 is an alternative to PM,3 or TW,3. Which to use is handled by hard-coding a decision-making method. See the documentation of the *levels struct* and the *resetCurrentLevels* function in the dmisConformanceTesterStart.cc file for details.

There are a few cases in which the qis and ipv addenda are both non-zero. In these cases, qis and ipv are alternatives, not both required. In these cases, negative numbers are used in levelsSet.cc. The methods of making decisions about what to require are described in Section 6 of the Users Manual. The *setLevsArray* function makes the decisions. See the documentation of that function in the dmisConformanceTesterStart.cc file for details.

If the checking of subclasses and attributes were done strictly according to the Excel spreadsheet for conformance classes, it would often be redundant. For example, *macroBlock*, *macroStm*, and *endmacStm* are all allowed at level 2 of both APs, but none is allowed at level 1 of either AP. Hence, going strictly according to the spreadsheet, there would be a 2 in p and tw for both *dmisBlockSubs_macroBlock* and *macroBlockAtts_a_macroStm*. If a MACRO statement is found in an error-free DMIS file, a macroBlock must also occur, so both the block and the statement would require level 2.

Two types of problem arose that have been fixed by not making all the entries in levelsSet.cc be strictly according to the Excel spreadsheet. The redundancies described above enabled this to be done without compromising the functionality of either the dmisConformanceTester or the dmisConformanceRecorder. The levelsSet.cc file is generated automatically by the makeLevels system utility (documented only in the makeLevels.cc source code). To readjust the entries in levelsSet.cc, changes were made in the p1Lists.cc and tw1Lists.cc files which are #included in levelsSet.cc.

The first type of problem arose because, in the dmisConformanceTester, if a conformance error is found, *printSelf* is called for the offending item. This meant, for example, that if a *macroBlock* were found at level 1, the entire block would be printed and appear to be non-conforming, even though only the *macroStm* line and the *endmacStm* line are actually the offenders. That would be confusing to the user. To avoid this, *macroBlock* and every other type of block are allowed at level 1 of both the prismatic AP and the thin-walled AP. Since at least level 1 of either of those APs is required for every conforming DMIS program, no block is ever out of conformance. Since the statements that start and end a block (*macroStm* and *endmacStm* in the case of a *macroBlock*) still require the higher level, no out-of-conformance situations will slip through.

The second type of problem arose because in some cases, a class is required by both an AP and an addendum, even though each subclass of the class is allowed in only one AP or addendum. If a non-zero entry appears for both the AP and the addendum in the *levels struct* for such a class, it will appear that both the AP and the addendum are required. For example, the *valueStm_realVar* subclass of *valueStm* is the parent of 11 subclasses. One of these is *valueRt* (for the rotary table addendum). This would normally lead to *valueStmSubs_valueStm_realVar* being set to *{2,2,1,0,0,0,0,0,0}*, which would make it appear that level 1 of rt is always needed in order to use *valueStm_realVar*. This has been avoided by using *{1,1,0,0,0,0,0,0,0}* instead. The testing is not compromised because the subclasses of *valueStm_realVar* are all set at the correct levels.

8.2 Check_inputFile

The *check_inputFile* function is the root of a tree of *check_xxx* functions which traverse the parse tree. The dmisConformanceTester.cc file contains 1632 automatically generated *check_xxx* functions – one for every C++ class in dmis.hh that has subclasses or attributes. There are two types of *check_xxx* functions, one for classes with attributes and one for classes with subclasses. Classes which have neither attributes nor subclasses do not need a checking function. That much is the same as for the dmisConformanceChecker. The *check_xxx* functions all take a *log* argument, which is a flag indicating whether the function is working for the dmisConformanceTester (*log* set to 1) or the dmisConformanceRecorder (*log* set to 0). The *check_xxx* functions have two kinds of functionality.

First, each top-level block of code in each *check_xxx* function calls *adjustLevels* if an instance of a subclass or attribute is found. *AdjustLevels* updates two global variables: *currentLevels* and *levelForcers*. *CurrentLevels* is a *levels struct* that keeps track of the minimum level of each conformance module that is required so that the portion of the parse tree that has been traversed is in conformance. *LevelForcers* is an array of strings. Each entry in *levelForcers* is the name of the class or class-and-attribute that first forced the corresponding entry of *currentLevels* to have its current value. For example, suppose *currentLevels* is *{2,1,0,0,0,0,0,0,0}* and an instance of a *featElongcylStm* is found. *AdjustLevels* will compare the entries in the *levels struct* for *dmisFreeStatementSubs_featElongcylStm*, which is *{2,3,0,0,0,0,0,0,0}*, to the entries in *currentLevels*. Since the 3 for tw in *{2,3,0,0,0,0,0,0,0}* is greater than the 1 for tw in *currentLevels*, *currentLevels* will be changed to *{2,3,0,0,0,0,0,0,0}*, and *levelForcers[1]* will be set to *"featElongcylStm"*. See the documentation of *adjustLevels* in dmisConformanceTesterStart.cc for more details.

Second, if the *check_xxx* function is working for the dmisConformanceTester (*log* set to 1), the user will have specified a conformance class to check against. In this case, the kind of conformance check performed by the dmisConformanceChecker and described in Section 7.3 will be done.

After the parse tree has been traversed, so that *currentLevels* and *levelForcers* have been set, the *testDmis* and *recordConformance* functions behave differently, as described in the next two sections.

8.3 dmisConformanceTester

In the dmisConformanceTester, it is the *checkOneFile* function (called by the *testDmis* function) that calls *parseDmis* and *check_inputFile*. After those functions have finished their work, *checkOneFile* calls *resetCurrentLevels* to deal with the *intFuncPtdataAtts_a_faLabel* problem described in Section 8.1. Then *checkOneFile* calls *setLevsArray*, which prints a conformance statement based on the values in *currentLevels* into a buffer, dealing with user preferences and the possibly negative entries for ipv and qis while it does so. *CheckOneFile* then prints the buffer and calls *showLevelForcers* to print the level forcer for each conformance module that is required.

8.4 dmisConformanceRecorder

In *recordConformance*, after the parse tree has been traversed as described in Section 8.2, *resetCurrentLevels* and *setLevsArray* are called as described in Section 8.3. Finally *recordConformance* calls *insertConformanceStatement*, which makes a backup copy of the DMIS input file, performs several checks, and inserts a conformance statement in the file.

9 How the dmisTestFileReductor Works

A description of what the dmisTestFileReductor does and how to use it is given in Section 7 of the Users Manual. Read that first.

The dmisTestFileReductor is less complex than the other utilities. It does not use the DMIS parser or the C++ classes for DMIS.

9.1 Main function

The *main* function checks for the following user generated errors and exits if any occur.

- Duplicate APs or Addendums are used in the arguments.
- The third argument is not PM1, PM2, PM3, TW1, TW2, or TW3.
- The fourth and succeeding arguments are not the name of a level of an addendum.
- There are fewer than 4 arguments.
- The incoming file cannot be opened.

If the incoming file ends in .dmi, it is assumed to be a DMIS input file, and it is processed by *processDmisFile*. Otherwise, the incoming file is assumed to contain a list of the names of DMIS input files, and each file in the list is processed by *processDmisFile*.

9.2 ProcessDmisFile function

The *processDmisFile* function examines the incoming file, decides whether there should be an outgoing file, and prints the outgoing file if so. The *fileRequirements* variable used by *processDmisFile* is an initially empty character array that becomes filled with information about the minimum levels of AP and addenda needed to execute the file.

ProcessDmisFile checks the incoming file's first line to see if it meets the requirements of the *conformanceModules*. If a file's first line fails to meet the requirements of the *conformanceModules*, no outgoing file is written.

If an outgoing file is to be written:

- The *fileRequirements* is updated by calling *compareReq*.
- Then *printInConformed* is called to print the new file.

9.3 CompareReq function

The *compareReq* function reads through the file and keeps track of the maximum level required by the lines of the file for each of the 9 conformance modules plus the IPQI pseudo conformance module.

After reading the file, *compareReq* prints the requirements into the *fileRequirements* array.

For IP, QI, and IPQI, if *ipqi* is not 0 after the file has been processed:

- If *ip* is not 0, the larger of the *ipqi* and *ip* values is used for IP.
- If both *ip* and *qi* are 0, the value of *ipqi* is used for IP.
- Otherwise (*ip* is 0 and *qi* is not 0), the larger of the *ipqi* and *qi* values is used for QI.

9.4 PrintInConformed function

The *printInConformed* function reads through the incoming file again. As it reads, it checks the requirements for each DMIS code line against the requirements given in the command to run the dmisTestFileReductor. If the requirements are met, the comment line giving the requirements and the DMIS code line that follows are both printed in the outgoing file.

When the DMISMN line of the file is encountered, the *fileRequirements* are transcribed (with commas added) to the end of the DMISMN line, replacing the requirements given in the incoming file.

10 More on EBNF

First read the section on EBNF in the Users Manual.

10.1 DmisStatement

An important convention that has been used in building dmis.debnf is that every `production` whose first `definition` ends with *#* (the character for ENDLINE), is considered to be a *dmisStatement* when it has been parsed into a *production*. There is one exception to this rule, and it is hard-coded. Namely, *noParseStatement* is not a *dmisStatement*.

DmisStatements are the things whose uses are counted in the dmisConformanceChecker. The *prepareStatementNames* function of debnf2pars builds a list of *dmisStatement*s by checking for an ENDLINE at the end of the first *definition* of each *production*. Then when classes are being printed, every class whose name is on the *statementNames* list is declared to be a subclass of *dmisStatement*. In order to support this method of finding *dmisStatements*, if the first *definition* of a *production* in dmis.debnf ends with *#*, then all *definition*s must end with *#*. It is up to the maintainer of dmis.debnf to see that this convention is followed.

In order to avoid difficulties in constructing the dmis.debnf, a few things that should be *production*s that end with *#* are not even defined; rather their subtypes are defined and end with *#*. If they existed, these undefined *production*s would be named *calibStm*, *recallStm*, and *saveStm*, correspondng to `CALIB`, `RECALL`, and `SAVE` in DMIS.

In addition, because there are so many subtypes of `FEAT` and `TOL` in DMIS and it is desirable to count the subtypes separately, EBNF *production*s for `FEAT` and `TOL` are not defined, but their subtypes are defined and end with *#*.

It would be desirable to have classes for `CALIB`, `FEAT`, `RECALL`, `SAVE`, and `TOL` in order to make the API for the classes easier to use. This could be done by hard-coding in debnf2pars.y, but has not been done.

10.2 Naming Conventions

Several naming conventions have been used in dmis.debnf. It is up to the maintainer to see that they are followed. These include the ones described at the beginning of dmis.debnf plus the following:

- The names of all *production*s for DMIS statements and only those *production*s end with *Stm*.
- The names of all *production*s that are lists and only those *production*s end with *List*.

The structure of EBNF names for DMIS statements and lists is not used in debnf2pars. Other naming conventions are detected in debnf2pars, such as spelling tokens in all upper case letters and starting names of primitive types with an upper case letter followed by a lower case letter.

10.3 Token Spelling Structure Conventions

Wherever possible, the spelling of a token name in dmis.debnf is identical to the spelling of the name in DMIS and dmis.debnf does not include a *production* giving the spelling. The EBNF standard, however, does not allow digits or underscores in *production* names. Some DMIS token names contain digits and/or underscores. To deal with this, different names (not containing digits or underscores) are used in the DEBNF file. However, since DMIS input files must use the DMIS spelling, a method is needed for telling the generator what spellings to look for when it is reading a DMIS file. This is done by putting the spellings of these irregular token names at the beginning of the DEBNF file.

It is required in debnf2pars that explicit token spellings be given in terms of *expression*s that are either of the form *'x'* or of the form *('X'|'x')*. For example, 2RC is a DMIS token. In the DEBNF file, the token is called *TWORC* and the line giving its spelling is
TWORC = '2', ('R'|'r'), ('C'|'c').

Where the form *('X'|'x')* is used, the first letter must be upper case, the second letter must be lower case, and both must be the same letter. In the *'x'* form, the x may be any printable character as far as debnf2pars is concerned, but in practice, it will never be a letter.

10.4 Multiple Optionals

Three distinctly different ways[1] to represent a multiple optional are available in EBNF. For example, zero to three origin specifications (a comma followed by an origin) may be written any of the following ways.

```
[c, orig], [c, orig], [c, orig]
3*[c, orig]
[c, orig, [c, orig, [c, orig]]]
```

The first representation is not used in the test suite because it does not simplify solving the following problem.

When C++ is generated for a class for a *definition* with a multiple optional, there will be a separate attribute of the class for each possible occurrence of the optional. In the example, the three attributes might be named *orig1*, *orig2*, and *orig3*. Now when a DMIS file is parsed and there are fewer than the maximum number of occurrences of the optional, it is necessary to decide which of the attributes are non-zero. In the example, if there are two origin specifications in the DMIS file, then there are three ways in which the non-zero attributes may be chosen: (*orig1*, *orig2*), (*orig1*, *orig3*), or (*orig2*, *orig3*). To make life easy for the application programmer, who does not want to have to test for all possibilities, the parser should choose one of them. The obvious correct choice is to use the first N attributes when there are N non-zero values. That is what the test suite implements. In the example, that is (*orig1*, *orig2*).

The second representation is the most compact and does lend itself to solving the problem. With the second representation, however, there is no easy way to assign meaningful names to the attributes. An automatic name assignment method is easy to program but makes poor names. In an earlier version of the test suite, before manual name assignment was implemented, the second representation was used exclusively in dmis.debnf. Code for dealing with multiple optionals represented that way was written and is still included in debnf2pars. That code might be removed to reduce the complexity of debnf2pars.

The third representation both lends itself to solving the problem and allows manual name assignment, so in this version of the test suite, debnf2pars includes code for dealing with that representation, and that representation is used exclusively in dmis.debnf.

11 More on C++ Classes for DMIS

Read section 2 of the System Builders Manual before reading this section. That section describes the C++ classes for DMIS. This section deals with important abstractions concerning the classes.

The automatically generated C++ classes for DMIS are more regular and more verbose than one would write manually. The regularity is a tremendous advantage for implementing automatic generation of conformance checking code (or any other type of code). The regularity also reduces the variety of types of code one must write when using the C++ classes, which is another advantage. The verbosity is not a drawback for the automatic generation of code. The generator is screamingly fast. If the verbosity increases the time it take to generate the code from 10 seconds

1. If the multiplicity of the optional is three or more, they could be mixed to make even more ways to do it.

to 20, it is not a problem. All the applications that have been built using the C++ classes are also screamingly fast, so the verbosity does not seem to slow down applications significantly.

The classes form a hierarchy that is several levels deep. For example, the threads through the inheritance hierarchy with *intConst* at the bottom and *cppBase* at the top (i.e. *intConst* and all its ancestors) is shown in Figure 9.

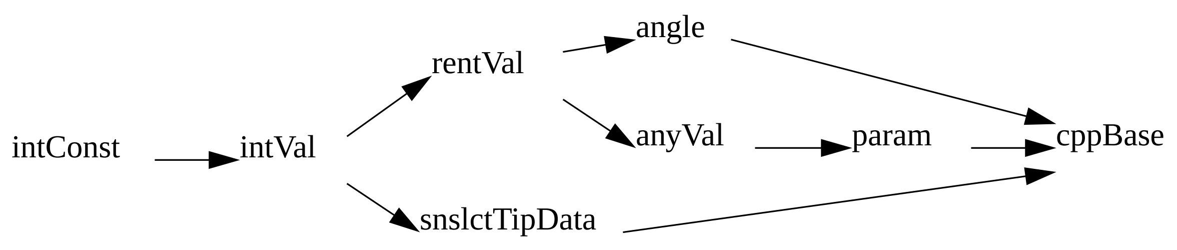


Figure 9. Ancestors of intConst

Many classes are parent classes and many classes have attributes, but no parent class has attributes. In the code, the suffix *atts* is used with classes that have attributes, and the suffix *subs* is used with classes that have subclasses.

There are frequent instances of multiple inheritance in the C++ classes. The cases of multiple inheritance arise from two sources. First, the classes are designed to make it easy to count instances of DMIS statements (*dmisStatement* class). Hence, many classes have both *dmisStatement* and *dmisFreeStatement* as parents. Declaring that a class is a subclass of *dmisStatement* is hard-coded in debnf2pars.y. Second, some classes may be used for more than one purpose. An integer variable (*intVar* class), for example, can be used as an integer value (*intVal* class), as a variable in a `READ` or `WRITE` command (*rwVar* class), or in a `PROMPT` command (*promptVar* class), so that *intVar* has *intVal*, *rwVar*, and *promptVar* as parents.

Some object-oriented theorists might object that “can be used as” and “is a subclass of” are different concepts, and Figure 9 is actually showing a “can be used as” hierarchy. Some object-oriented languages, EXPRESS for example, have a construct for “can be used as”. In EXPRESS, it’s the select type. Making a distinction between “can be used as” and “is a subclass of” would be required if any of the ancestors of a class had attributes. In Figure 9, for example, if *param* had an attribute such as *name*, it would be inherited by *intConst*, which has no use for a *name*. Then, a different modeling method would need to be used indicating that, although *intConst* is not a subclass of *param*, an instance of it can be used wherever an instance of a *param* is required. Since no ancestor of any C++ class for DMIS built by the test suite has any attributes, this is never a problem. As long as there are no attributes, there really is no conceptual distinction between “can be used as” and “is a subclass of”.

There are no enumerations in the C++ classes for DMIS. Instead, where a human would normally program an enumeration with N values, debnf2pars programs a parent class with N subclasses. For example, the scope of a variable, which might be represented by an enumeration *declScope* with values *COMMON*, *LOCAL*, and *GLOBAL*, is represented instead by the class *declScope* with subclasses *declScope_COMMON*, *declScope_GLOBAL*, and *declScope_LOCAL*. The subclasses have neither attributes nor subclasses of their own. In manual programming, testing the type of a class takes no more code than testing the value of an enumeration constant. In size of executable, enumerations would almost certainly be smaller, but the C++ class system fits in less than 10 Mb of an executable, which is not large relative to the size of modern computer memories. In terms of speed, if there is a penalty at all, it cannot be much, since the executables built on the C++ class system are so fast.

12 Building Code Generators

12.1 Pros and Cons of Code Generators

Using a code generator has several pros and a few cons.

This section compares a programmer building and using a code generator with the same programmer manually generating code with the same functionality. In the case of large application such as the utilities in the test suite, a realistic alternative to using a code generator would be to have a team of programmers rather than a single programmer. Keeping the team coordinated would require a substantial portion of programmer time and would give using a code generator relatively more advantage.

12.1.1 Pros:

An upper-end personal computer can write code roughly 100,000,000 times as fast as a human. This means an entire lifetime of work by a human (50 years at 2000 hours per year) can be done in less than a minute by a computer.

Once a computer knows how to write a particular kind of code (i.e. the generator code includes debugged functions that write the code), the computer will write any number of functions of the same kind without making any errors. The computer will not get bored doing that. Humans make a lot of errors and get bored, leading to taking more time and making more errors.

Once the code formatting rules are programmed into the generator, they will be followed entirely consistently. Humans are unable to be completely consistent. Since compilers do not care about formatting, human-made inconsistencies persist in finished code, ready to confuse whoever is maintaining the code.

Using a computer to generate code provides the human programmer with more time for perfecting code or developing new types of code.

Automatically generated code is data-driven. In the test suite, the data is the DEBNF file that is input to the generators. When the input data changes, if a generator has been built, thousands of lines of code can be updated in a few seconds.

12.1.2 Cons:

Where there is a lot of variety in the input to a generator, it is often necessary to hard-code the generator to write specific code for specific input. This takes more time than writing the code directly.

Because the usefulness of a generator is maximized when many functions or classes of the same kind are generated, the programmer building the generator will strive to design the output of the generator to have little variety. This may lead to generating code that is more verbose and/or less efficient than code that a human would write.

12.2 Development Technique

To develop any code that generates code, one useful technique is to write samples of the code that should be generated from some specific inputs. If possible, compile, test and debug the hand-written code. When the generator is ready to be tested, feed it the same inputs. The inputs should be diverse enough to cause the generator to generate all the varieties of code it can generate. Then check that the generated code is identical to the hand-written code. In Linux or Unix, this may be done using diff. Of course the hand-written and automatically generated versions of the code are never identical the first time they are compared. Modify the generator (or the hand-written code), regenerate the code, and compare again until there are no differences.

The technique just described has been used throughout the development of the generators in the test suite.

12.3 Generating Code Automatically

There are two distinct approaches to generating code automatically in a C++ program, either:

- generate code text directly with print statements, or
- define structures that represent code, write functions that print code from the structures, build the structures in your program, and call the print functions to print code text.

Which approach to use depends on the situation. If structures are required for other reasons or there are many instances of each kind of structure, it may be best to use the structures approach. If only one function of a given type is to be generated, it will probably be best to print code text directly. It is often convenient to mix the two approaches.

- In the test suite, structures representing EBNF are required for parsing and manipulating EBNF, so the dmis.y, dmis.hh, and dmis.cc files for full DMIS are printed primarily from structures.
- The dmisConformanceChecker.cc and dmisConformanceTester.cc files in the test suite print code directly.
- The dmis.hh and dmis.cc files define *printSelf* functions that generate code from structures. The generate tutorial in the System Builders Manual shows how to write a DMIS program by building structures and then calling the structure printer.

When text is printed directly, any variability in what is printed results from using arguments to the functions that do the printing.

Regardless of how printing is done, if code is being printed that a human might want to read, the code should be pretty-printed. That is, line length should be controlled, code sections should be arranged appropriately and consistently, and lines should be indented appropriately and consistently. All of the code produced by the test suite is pretty printed. Controlling line length often requires first printing to a buffer and then counting characters while printing to a file from the buffer. Controlling line length also requires avoiding constructing long names. Except for controlling line length, the code needed to pretty print code is brief and easy to write. The maximum line length the test suite tries to maintain is 80 characters. Keeping under that limit is largely successful but fails in some cases where there are long names.

12.4 Printing Directly

In C++, there are two very different methods that may be used for printing: the C printf function (and its variants) or the << notation. The printf method is horribly ugly to look at and awkward to use. The << method is less ugly to look at but has massive hidden ugliness in the overloading of << that is required. It is easier to exert fine control using the printf method. The printf method is used exclusively in the test suite.

There are so many different things one might want to print, there is no easy way to implement the functionality that is required. That is the reason there is no civilized set of printing functions. Just learn to live with that.

When the code printers in the test suite have long sections of hard-coded material to print, that is done by using a single fprintf function call with an argument that is a long string split over several lines. This is a very useful technique. Here is a short example:

```
fprintf(yaccFile,
"char                    warningMessage[256];\n"
"extern FILE *              yyin;\n"
"extern int                yylex();\n");
```

In a few cases, the test suite inserts large sections of documentation in automatically generated code. This is done using the technique just described.

12.5 Combining Manually Written and Automatically Generated Code

Where a program has both one-of-a-kind functions and many-of-a-kind functions, and there is no need to intersperse the two types, it may be simpler to write the fixed code in a file than to have a generator regurgitate it. This is done for the dmisConformanceTester. The automatically generated dmisConformanceTester.cc file defines over a thousand *check_xxx* functions and #includes the hand-written dmisConformanceTesterStart.cc file, which has 16 one-of-a-kind functions. Similarly, the automatically generated dmisConformanceChecker.cc file #includes the hand-written dmisConformanceCheckerStart.cc file.

12.6 An Alternative - Generate and Execute

An alternative to generating source code automatically, compiling it, and executing it is to generate and execute the code in the same process. This may be done three ways. First, if you are working with a language that includes a real-time compiler, you don't need to change much. Just put a compile step between the generate and execute steps. Second, if you are working with an interpretable language such as Lisp, you don't need to compile, so just go directly from generation to execution. In Lisp, you can either generate an executable structure and then execute it, or you can write out a file and then load it back in and execute it. Third, you can write extremely generic data driven code with a lot of function pointers.

Although these approaches might make maintaining several hundred thousand lines of code unnecessary, for dealing with DMIS, those alternative approaches seem inferior for several reasons. First, the automatically generated code in the test suite is quasi-static. That is, unless DMIS changes, the conformance classes change, or a bug is fixed, there is no reason to change the code. It is a waste of time and effort to regenerate the code. The code will run more slowly if has to be regenerated and then executed. Second, it may be necessary to do some hand-editing of automatically generated code (as in the 27 xxLists.cc files) before it is ready to use. Third, it makes development harder because there is no chance to debug between generation and execution. Fourth, if either Lisp backquote notation or extremely generic code is used, it becomes nearly impossible by studying the code to see what the code is going to do when it executes.

13 How debnf2pars Works

13.1 Introduction

The debnf2pars code generator is very complex. This section only gives an overview of how debnf2pars works and focuses on some special features. The debnf2pars.y file includes about 125 pages of in-line documentation. To get a deep understanding of debnf2pars, first read this section, and then study the in-line documentation.

One capability built into debnf2pars is the ability to deal with DEBNF constructs that can cause shift/reduce conflicts in dmis.y. The journal article cited in Section 1.3 describes the constructs that cause conflicts, why the conflicts arise, and what the replacement constructs are. The article does not explain how a parser is built that parses in terms of the replacement constructs while building a parse tree using C++ classes derived from the original constructs. An explanation of how that is done in one case is given in Section 13.9.1. See the in-line documentation in debnf2pars.y for additional explanations.

As shown in Figure 5, debnf2pars reads a file written in DEBNF and writes eight files:

- a YACC file (dmis.y) for a parser of the language described in the DEBNF file,
- a Lex file (dmis.lex) for the lexical analyzer used by the parser,
- a C++ header file (dmis.hh) defining classes for representing DMIS,
- a C++ code file (dmis.cc) implementing the functions and methods declared in the header file,
- a C++ code file (dmisConformanceChecker.cc) containing most of the functions required for the dmisConformanceChecker,
- a C++ code file (dmisConformanceTester.cc) containing most of the functions required for the dmisConformanceRecorder and the dmisConformanceTester,
- a C++ code file (allSubAtts.cc) defining arrays that give either the subclasses or the attributes of every class that has either subclasses or attributes,
- a C++ code file (assignMasterSubAtts.cc) that assigns values by name to the entries of the *masterSubAtts* array.

13.2 Data Structures Used in debnf2pars

Most of the data structures used in debnf2pars are instances of the C++ classes for EBNF described in Section 4.1.

In addition, the *attCell* class and the *classData* class defined in debnf2pars.y are used to hold data about the C++ classes for DMIS. An *attCell* has the data for one attribute of a class and a *next* pointer so that lists of *attCell*s can be constructed (even though no *attList* class is defined). A *classData* has the data for one class. The *classDatas* array, a global variable in debnf2pars.y, is an alphabetically ordered array of pointers to *classData* instances. Ordering is done by class name. The dmisConformanceChecker.cc and dmisConformanceTester.cc files are printed using the *classDatas* array.

The debnf2pars.y file declares and uses the global variables:

- *productions* (a *prodList* of all the *production*s in the DEBNF file)
- *statementNames* (a *stringCell*, the head of list)
- *tokenNames* (an array of strings giving the names of tokens, alphabetically arranged)
- *tokenLexes* (an array of strings giving the spellings of some tokens, alphabetically arranged)
- *terminalNames* (an array of strings giving the names of terminal symbols).

The debnf2pars.y file also declares and uses five global variables, each of which is a unique *expression*. These are used to simplify testing for equality. *CommaExp* is used during parsing EBNF and in many other activities. The rest are used only for fixing shift/reduce conflicts and printing dmis.y.

- *commaExp* (represents a comma)
- *nullExp* (represents a null pointer)
- *trueExp* (represents boolean true)
- *falseExp* (represents boolean false)
- *equalSignExp* (represents an equal sign)

13.3 The Main Function

The *main* function in debnf2pars.y goes through the following steps.

- Initialize the *classDatas*, *tokenNames*, and *tokenLexes* arrays with zeros.
- Call *yyparse* to parse in the DEBNF file dmis.debnf, causing the *productions* list to be built and populating the *tokenNames* and *terminalNames* arrays. For each *production* a list of *definition*s is built, and it is determined if the *production* represents a list.
- Call *prepareStatementNames* to go through the *productions* and make a list, *statementNames*, of the names of those that are DMIS statements.
- Call *reviseSpelling* to set up the *tokenLexes* array that will be used to generate correct spellings when dmis.lex is written. The function first makes *tokenLexes* be a copy of *tokenNames*. Then it goes through the *productions*, and if a *production* gives a spelling for a token name, the name is changed in the *tokenLexes*.
- Call *productions.findUsedIn* to set the *usedIn* list of each *production*, P. The *usedIn* list of a *production* P is a list of other *production*s that use P. To set the *usedIn* list, it is necessary to go through every *expression* of every *definition* of every *production*. In the process of doing that, a second job is done. Namely, in every *expression* representing P, the *prodValue* attribute of the *expression* is set to point to P.
- Call *addData* to find the class names for every *definition* of every *production* and to determine which *production*s will have classes that are parent classes or subclasses of classes for other *production*s.
- Call *selectProductions* to select those *production*s for which classes should be printed.
- Call *recordClasses* to put pointers to classData instances into the *classDatas* array.
- Call *printCppClasses* to print dmis.hh and dmis.cc.
- Call *printConfAllSubAtts* to print allSubAtts.cc.
- Call *printConfAssignMaster* to print assignMasterSubAtts.cc.
- Call *printConfChecker* to print dmisConformanceChecker.cc.
- Call *printConfTester* to print dmisConformanceTester.cc.

- Call *fixConflicts1* to modify the list of *productions* by detecting and fixing constructs of one simple type that will cause shift/reduce conflicts if translated directly into YACC.
- Call *fixConflicts2* to modify the list of *productions* by detecting and fixing more complex constructs that will cause shift/reduce conflicts if translated directly into YACC.
- Call *printYacc* to print dmis.y.
- Call *printLex* to print dmis.lex.

13.4 DEBNF Parser

The DEBNF parser in the debnf2pars.y file (not the DMIS parser built by debnf2pars) consists of a hand-written debnf2pars.lex file and about 150 lines of YACC. The parser parses DEBNF into a parse tree using the DEBNF classes. In the current version of DEBNF, comments of the form *`(*A=name*)`* and *`(*C=name*)`* in the DEBNF file are parsed into names for attributes and classes, respectively. If an attribute name for an *`expression`* that will become an attribute is not given in the DEBNF file, an attribute name will be assigned by the *findAttributeNames* function in debnf2pars.y. If a class name for a *`definition`* is not given in the DEBNF file, a class name will be assigned by the *findClassNames* function.

13.5 Generating dmis.hh and dmis.cc

Read Section 2 of the System Builders manual and Section 11 of this manual before continuing this section.

Printing the dmis.hh and dmis.cc files is done by *printCppClasses* and 20 *printCppXxx* functions (plus their subordinates) in a hierarchy below that. Most of these functions are straightforward and easy to understand. Seven of them print in both files. The hierarchy of functions headed by *printCppClasses* is shown in Figure 10. Each level of indentation in the figure indicates that the less indented function above calls the more indented function below. On the figure, subordinates of functions not named *print<Something>* are omitted, and if a *print<Something>* function appears more than once, the hierarchy under it is only shown the first time.

The remainder of this section describes the *printCppClasses* function and discusses issues of printing dmis.hh and dmis.cc. For details, see the in-line documentation of the *printCppXxx* functions in debnf2pars.y.

- *printCppClasses*
  - *printCppBaseClass*
    - *printCppClassStart*
  - *printCppDocumentation*
  - *printCppNames*
  - *printCppProductionClasses*
    - *findClassData*
    - *makeClassDataSubs*
    - *printCppClassDerived*
      - *findAttributeNames*
      - *flattenOpts*
      - *makeClassDataAtts*
      - *printCppClassConstructors*
        - *printCppClassConstructorArgs*
      - *printCppClassData*
      - *printCppClassDestructor*
      - *printCppClassDoc* (calls itself recursively)
      - *printCppClassMethods*
      - *printCppClassPrinter*
        - *printCppClassPrinterListNo*
        - *printCppClassPrinterListYes*
        - *printCppClassPrinterOpt*
          - *printCppClassPrinterOpt1* (calls *printCppClassPrinterOpt*)
            - *printCppClassPrinterListNo*
            - *printCppClassPrinterListYes*
          - *printCppClassPrinterOpt2*
      - *printCppClassStart*
    - *printCppClassParent*
      - *printCppClassDestructor*
      - *printCppClassStart*
    - *printCppClassTop* (subordinates identical to those of *printCppClassDerived*)
      - *findAttributeNames*
      - *flattenOpts*
      - *makeClassDataAtts*
      - *printCppClassConstructors*
      - *printCppClassData*
      - *printCppClassDestructor*
      - *printCppClassDoc*
      - *printCppClassMethods*
      - *printCppClassPrinter*
      - *printCppClassStart*
  - *printStarLine*

Figure 10. Hierarchy of Functions Headed by printCppClasses

13.5.1 printCppClasses

At the head of the hierarchy, *printCppClasses* does the following:

- Open dmis.hh and dmis.cc for writing and print #includes and namespace declarations.
- Call *printCppDocumentation* to print (entirely hard coded) documentation in dmis.hh.
- Call *printCppNames* to print the declaration of all classes in dmis.hh.
- Call *printCppBaseClass* to print the definition and implementation of the base class in dmis.hh and dmis.cc.
- Go through the *productions* list repeatedly, keeping track of which ones have already been used for printing classes. Each time a not-yet-used *production* is found all of whose supertype classes have already been printed, call *printCppProductionClasses* to print one or more C++ classes to represent that *production*. Class definitions for them are printed in dmis.hh and class implementations in dmis.cc.
- Close the namespace declarations and close the files.

It would be nice simply to print the classes in alphabetical order, but C++ compilers complain when a class is made a subclass of a class that has not yet been defined. Hence the procedure of looping repeatedly through the *productions* is used.

If a *production* has more than one *definition*, a parent class is printed for the *production* and a class is printed for each *definition*. A parent class is very minimal. It has only a constructor that does nothing, a destructor that does nothing, and a *printSelf* function with no definition. Because a class is printed for every *definition*, it is important for keeping the number of classes down that optionals be represented in the DEBNF file using square bracket notation, not by writing multiple `definition`s.

13.5.2 Flattening Optionals

For several purposes, the foremost of which is printing classes, it is useful to pretend temporarily that the *optional* items in a *definition* (with one type of exception) are not optional.

For example, if the original EBNF `definition` is:

```
ifStm, [calibSens], [elseStm, [calibSens]], endifStm
```

then the `definition` with optionals appearing to be required is:

```
ifStm, calibSens, elseStm, calibSens, endifStm
```

The process of making *optional*s appear to be required has been called flattening *optional*s. The function that does the work is called *flattenOpts*. Other debnf2pars activities that use *flattenOpts* are printing dmis.y (see Figure 18) and fixing shift/reduce conflicts (see Figure 14 and Figure 15).

The one exception made by *flattenOpts* is that an *optional* consisting entirely of constant terms and containing at least one keyword (`[EXTERN, c, DMIS, c]`, for example) is left unchanged. The exception is made because a single boolean attribute (*has_EXTERN* in the example) will be used to represent the entire *optional* in the C++ class derived from the *definition*. Keeping the *optional* unchanged makes it easy to treat it as a single item.

13.5.3 Printing printSelf

The *printSelf* functions in dmis.cc print DMIS code, and they are moderately complex. Hence printing the *printSelf* functions in debnf2pars.y is complex. It is done by the hierarchy of *printCppClassPrinterXxx* functions shown in Figure 10. Since *optional*s may be nested, *printCppClassPrinterOpt1* may call its superior, *printCppClassPrinterOpt*.

13.6 Generating dmisConformanceChecker.cc

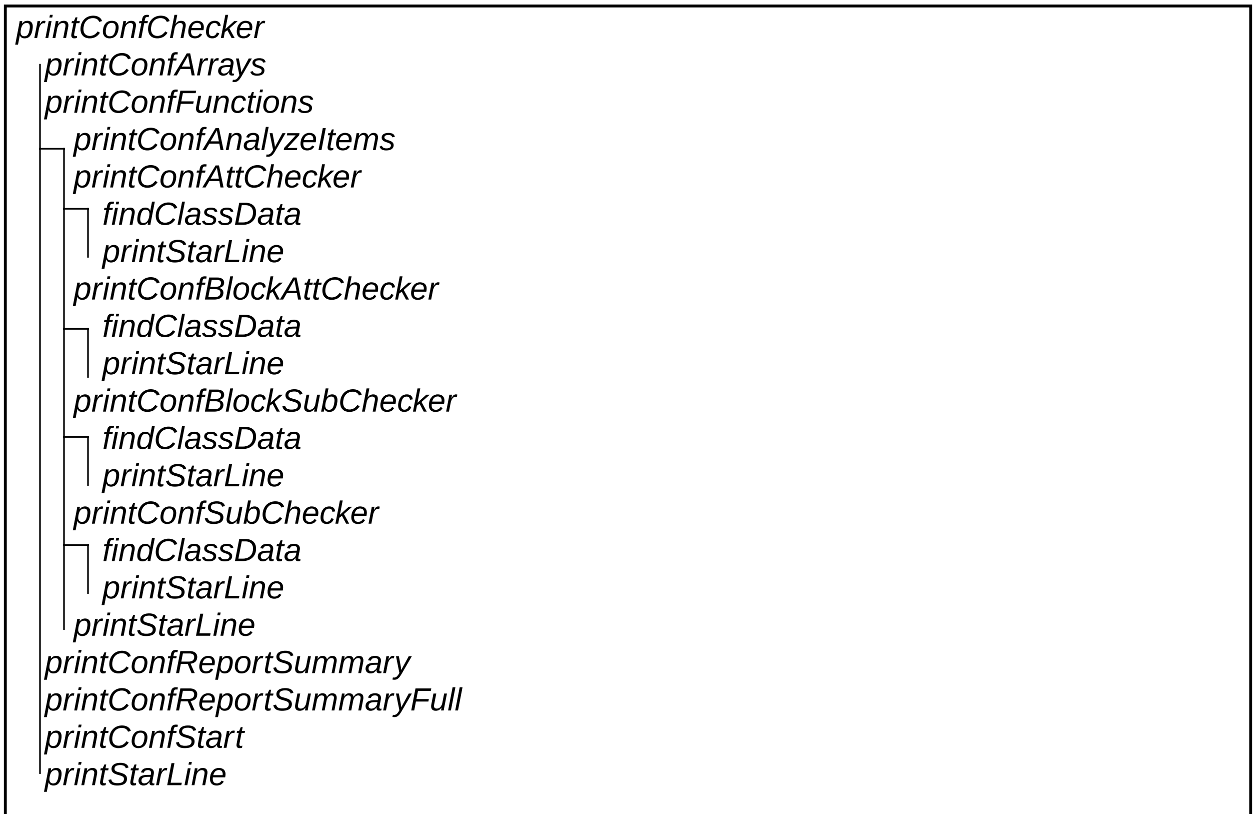
The dmisConformanceChecker.cc file is generated by the hierarchy of functions headed by *printConfChecker* shown in Figure 11.

PrintConfArrays prints an array declaration for each entry in *classDatas* that has either subclasses or attributes. For example:

```
const char * evalStmAtts[3] = {"evalStmAtts", 0};
const char * extensMinorSubs[4] = {"extensMinorSubs", 0};
```

PrintConfFunctions prints the functions that test conformance. There are four types of them for: blocks with attributes, blocks with subclasses, statements with attributes, and statements with subclasses. Each type has its own print function subordinate to *printConfFunctions*.

The *printConfAnalyzeItems* subordinate of *printConfFunctions* prints the *analyzeItems* function that counts uses of *dmisStatement*s.



```
printConfChecker
  printConfArrays
  printConfFunctions
    printConfAnalyzeItems
    printConfAttChecker
      findClassData
      printStarLine
    printConfBlockAttChecker
      findClassData
      printStarLine
    printConfBlockSubChecker
      findClassData
      printStarLine
    printConfSubChecker
      findClassData
      printStarLine
    printStarLine
  printConfReportSummary
  printConfReportSummaryFull
  printConfStart
  printStarLine
```

Figure 11. Hierarchy of Functions Headed by printConfChecker

While the *printConfChecker* function and its subordinates have the ugliness that characterizes code printers, and understanding the placement of characters such as backslashes requires some study, there are no deep or complex issues associated with printing the dmisConformanceChecker.cc file or the dmisConformanceTester.cc file, which is next.

13.7 Generating dmisConformanceTester.cc

The dmisConformanceTester.cc file is generated by the hierarchy of functions headed by *printConfTester* shown in Figure 12. The functions subordinate to *printConfTester* are very similar to those subordinate to *printConfChecker*.

```
printConfTester
  printConfArrays
  printTestFunctions
    printTestAttChecker
      findClassData
      printStarLine
    printTestBlockAttChecker
      findClassData
      printStarLine
    printTestBlockSubChecker
      findClassData
      printStarLine
    printTestSubChecker
      findClassData
      printStarLine
  printTestStart
  printStarLine
```

Figure 12. Hierarchy of Functions Headed by printConfTester

13.8 Generating allSubAtts.cc and MasterSubAtts.cc

The allSubAtts.cc file is printed in a straightforward manner by *printConfAllSubAtts*, which goes through the *classDatas* array in alphabetical order. For each entry in *classDatas*, the function prints either an array naming the subclasses of a class or an array naming the attributes of a class.

The masterSubAtts.cc file is printed in a straightforward manner by *printConfAssignMaster*, which goes through the *classDatas* array in alphabetical order while incrementing a counter. For each entry in *classDatas*, the function prints either

masterSubAtts[<count>] = <name>Subs; or

masterSubAtts[<count>] = <name>Atts;

according to whether the *classData* indicates subclasses or attributes. The *<name>* is the name given in the *classData*, and *<count>* is the current value of the counter.

13.9 Modifying the C++ Class Instances for EBNF to Avoid Shift/Reduce Conflicts

The paper “Automatic detection…” cited in Section 1.3 describes the EBNF constructs that cause conflicts, why the conflicts occur, and what the constructs are that replace the problematic constructs. It does not describe how the replacements are made; this section gets into that.

The *main* function calls both *fixConflicts1* and *fixConflicts2*. These functions and their subordinates detect problematic constructs and do most of the work of replacing them. In some cases, constructs modified by *fixConflicts1* will be further modified by *fixConflicts2*.

```
fixConflicts1
  alwaysFollowedByComma
    alwaysFollowedByCommaInExps (also calls itself)
      expIsComma
      expIsProd
  removeCommaInExps (also calls itself)
    expIsComma
    expIsProd
```

Figure 13. Hierarchy of Functions Headed by fixConflicts1

```
fixConflicts2
  fixConflictList
    expIsComma
    fixConflictListNested
      alwaysFollowedByComma
      alwaysFollowedByNewline (also calls itself)
        expIsNewline
        expIsProd
      expIsComma
      fixConflictListNested1
        insertItemAndCommaInList
      fixConflictListNested2
        insertItemInList
          findProdSet
            replaceOptsInDefinitions
              checkDuplicates
                expListsSame
              replaceOptInDefinition
                replaceOptMulti
        removeCommaInExps
      fixConflictListOptional
      flattenOpts
    fixConflictListOptional
      insertItemInList
  fixConflictsUsers (see Figure 15)
```

Figure 14. Hierarchy of Functions Headed by fixConflicts2

```
fixConflictsUsers (from Figure 14)
  fixConflictOther
    alwaysFollowedByNewline
    expIsComma
    expIsNewline
    expIsProd
    fixConflictOther1
      makeNewDefs
      makeProdC
        findProdSet (see Figure 14)
        makeNewDefs
        prepareDefsForPlain
          findOptType
          flattenOpts (also calls itself)
            findOptType
      prepareDefsForPlain
      replaceProdC
        expIsComma
    fixConflictOther2
      expIsComma
      expIsProd
      makeNewDefs
      makeProdC
      prepareDefsForPlain
      replaceProdC
    replaceOptsInDefinitions (see Figure 14)
```

Figure 15. Hierarchy of Functions Headed by fixConflictsUsers

The hierarchy of functions headed by *fixConflicts1* is shown in Figure 13. The hierarchy of functions headed by *fixConflicts2* is shown in Figure 14 and Figure 15. The figures do not include member functions of EBNF classes, which are used profusely.

As may be gathered from the figures, *fixConflicts2* is substantially more complex than *fixConflicts1*. The nature of the functions subordinate to the two fixers, however, is similar.

13.9.1 An example

The remainder of this section is devoted to a full explanation of one example of a shift/reduce conflict, how it is detected, and how it is fixed. The example involves `production`s used in the DMIS `PAMEAS` statement. It is handled by *fixConflicts1*. It has been simplified by removing additional *`definition`*s that are not part of the problem. The EBNF *`production`*s involved from dmis.debnf that form a problematic construct are shown in Figure 16. The changed *`production`*s (as they would be if they were printed) are shown in Figure 17.

```
pameasDetail =
        SCNVEL, c, fedratLinear, [c, PITCH, c, realVal]
       | PITCH, c, realVal
       ;
pameasVar2ListItem =
         pameasDetail, c, pLabel
       ;
```

Figure 16. EBNF Construct Before Fixing

This is an example of the following problem type, which is fixable.

There is a *production* P, such that (1) P has at least one *definition* with an optional at the end starting with a comma and (2) the name of P is followed by a comma, *c*, wherever it appears in a *definition*.

In the example, P is *pameasDetail*. The optional at the end of a *definition* is *[c, PITCH, c, realVal]*. The only place *pameasDetail* is used is in the one *definition* of *pameasVar2ListItem*, and in that place it is followed by a comma, *c*. Therefore, *pameasDetail* is always followed by a comma.

The problem is best understood by playing the role of parser. As parser, you get tokens (words, punctuation, numbers, etc.) from a lexer. Each time you get a token, you need to decide whether to shift (add the token to a list of tokens that form the beginning of a *definition*) or reduce (change a sequence of tokens that form a complete *definition* of a *production* to the name of the *production*). You can look one token ahead (at the “lookahead token”). You remember what you are doing based on what you have already processed.

The example crops up when you are reading a *pameasVar2ListItem*. You get to the point where you have seen the sequence *SCNVEL, c, fedratLinear* and have *c* as the lookahead token. You know you are working on the first *definition* of a *pameasDetail*, but you cannot tell whether the *c* is (1) the first *c* in the optional part of *pameasDetail* or (2) the *c* in the *definition* of *pameasVar2ListItem*. In the first case, there are more terms in the *pameasDetail*, so you should shift the *c*. In the second case, you have seen a whole *pameasDetail*, so you should reduce the sequence to a *pameasDetail*.

The general rule for fixing a conflict of this type is to change the EBNF by (1) inserting a comma after every *definition* of P, and (2) removing the comma that follows the name of P everywhere that P is used. For the example, that changes the EBNF as shown in Figure 17. Now when you see the *c* following *fedratLinear*, it has to be part of the *pameasDetail*. You cannot tell whether it is the *c* at the beginning of the optional or the *c* at the end of the *definition*, but it does not matter, because you must shift it in either case. If the next token received from the lexer is *PITCH*, you will know that you are in the optional and will shift. If the next token is *pLabel*, you will know that you have seen an entire *pameasDetail* that did not include an optional and you will reduce.

```
pameasDetail =
       SCNVEL, c, fedratLinear, [c, PITCH, c, realVal], c
      | PITCH, c, realVal, c
      ;
pameasVar2ListItem =
        pameasDetail, pLabel
      ;
```

Figure 17. EBNF Construct After Fixing

The chain of events in debnf2pars that does the work starts while the EBNF is being parsed. Each time an EBNF `production` is parsed, an instance P of the C++ *production* class is built, and then the *findEndsInOptional* function is called to examine the last *expression* in each of the *definition*s of P. If that *expression* is an *optional* and the first *expression* in the *optional* is a comma, then the *endsInOptional* attribute of P is set to *true*. Otherwise, the *endsInOptional* attribute of P is set to *false*. For `pameasDetail`, *endsInOptional* is set to *true*.

After the EBNF file has been parsed, *main* calls the *findUsedIn* function of the *prodList* named *productions*. That function calls the *findUsedIn* function of every *production*. The *findUsedIn* function of a *production* P looks through every *expression* of every *definition* of every other *production* Q. If the *itemName* of an *expression* in Q is the name of P, and Q is not already at the end of the *usedIn* list of P, a pointer to Q is added to the end of the *usedIn* list of P. As a result of this process, the *usedIn* list of the *production* for `pameasDetail` ends up with one element, a pointer to the *production* for `pameasVar2ListItem`.

Next, when *main* calls *fixConflicts1*, the function looks at every *production* P in order to determine if P is involved in the kind of problematic construct described above. Eventually, f*ixConflicts1* gets to `pameasDetail`. Since the *endsInOptional* attribute of `pameasDetail` is set to *true*, the first requirement for being a problem is satisfied. *FixConflicts1* therefore calls *alwaysFollowedByComma* to determine whether `pameasDetail` is always followed by a comma, the second requirement. To determine whether a *production* P is always followed by a comma, *alwaysFollowedByComma* (with the help of *alwaysFollowedByCommaInExps*) hunts through every *expression* of every *definition* of every *production* in the *usedIn* list of P (including digging into *optional*s) looking for P. If every time P is found it is followed by a comma, then *alwaysFollowedByComma* returns *true*. For `pameasDetail`, the only member of *usedIn* is `pameasVar2ListItem`, which has only one *definition*, and in that *definition* `pameasDetail` is followed by a comma the one time it appears. Hence, *alwaysFollowedByComma* returns *true* for `pameasDetail`.

Now *fixConflicts1* knows that it has found a problematic construct it knows how to fix, so it sets about fixing it. To fix the problem, two steps are needed.

First, *fixConflicts1* goes through all the *definition*s of all the *production*s in the *usedIn* list of the problem *production* P and calls *removeCommaInExps* for the *expression* list of each *definition*. Each time *removeCommaInExps* finds an *expression* in the list whose *prodValue* is P, it checks that the next *expression* is a comma and then calls the *removeCell* function of the *expList* class to remove the comma *expression* from the list of *expression*s. Each time *removeCommaInExps* finds an *expression* in the list that has an *optValue*, it calls itself recursively to act on the *expression*s in the *optional* to which the *optValue* points. For `pameasDetail`, the result is that the comma after `pameasDetail` in the one *definition* of `pameasVar2ListItem` is removed.

Second (and finally), *fixConflicts1* goes through the *definitions* of the problem *production* P (`pameasDetail`) and adds a comma at the end of the *expression*s list of each *definition*.

This example is the simplest possible real example of how *fixConflicts1* works. *FixConflicts2* is more complex. However, the example illustrates several recurring themes. First, much of the work is done by functions that execute during and immediately after parsing and fill in values for the attributes of *production*s and *expression*s. If those attributes did not exist, all the work that is currently done to fill them in would have to be done more than once – in *fixConflicts1*, *fixConflicts2*, and other functions. Second, fixing one type of problem is done by looking at all the *productions*, seeing if any *production* P has the problem, and immediately fixing the problem for P. Third, fixing the problem is usually done by some sort of list surgery (adding an *expression* to an *expList* or removing one from it, removing a *definition*, adding a *definition*, etc.). Most of the list surgery functions are member functions of the EBNF classes; a few are defined in debnf2pars.y.

One important difference between *fixConflicts1* and *fixConflicts2* is that *fixConflicts2* sets the *fixType* of *production*s it modifies but *fixConflicts1* does not. The *fixType* is used in the process of printing dmis.y. Changes made by *fixConflicts1* do not need any special handling during that process.

All of the functions that participate in fixing conflicts are heavily documented, so it should be possible to understand what is going on in throughout the conflict fixing process.

13.10 Generating dmis.y

The first two subsections of this section discuss the overarching issues of building dmis.y. The remaining subsections discuss specific issues and how they are handled.

13.10.1 Parser Functionality

The dmis.y file defines the parser that will be built by the bison processor and the C++ compiler. The functionality required of the parser, therefore, must be built into dmis.y. The following functionalities are desirable in the parser:

- Be able to parse all error-free DMIS input files.
- Be able to continue through errors.
- Be able to handle `CALL`s to `MACRO`s defined in the file.
- Wherever there is a parse error, describe it and print the section of DMIS code that caused it with the line number from the input file.
- While parsing, build a parse tree in terms of C++ classes for DMIS. Make the parse tree available for use after parsing is complete.
- Wherever an undefined label is used or a label is defined more than once that should not be, print an error message and the section of code containing the label.
- Wherever an undefined variable is used or a variable is declared more than once, print an error message and the section of code containing the variable.

Most of those desirable functionalities are built into dmis.y by debnf2pars. The ideal is not quite reached. First, some errors will confuse the parser so that it will be unable to continue. Second, if there is an error, the parser will not make the parse tree available for use. Third, because the parser parses the DMIS file sequentially, but the file might not execute sequentially (because of flow of control statements such as `JUMPTO` or `IF`), the parser cannot be sure that its judgement of undefined and multiply defined labels and variables is accurate, so it issues only warning messages (not error messages) in most of those cases.

Each kind of functionality that is implemented must be embodied in dmis.y. This makes dmis.y more complex, requiring debnf2pars to be more complex. In some cases, implementing one kind of functionality interferes with implementing another.

The primary example of this is the combination of having an error-free parser and building a parse tree in terms of C++ classes. In order to have the C++ classes form a usable application programming interface, the C++ classes are built from “natural” EBNF. If the rules in dmis.y are built directly from the natural EBNF, however, they will lead to many shift/reduce conflicts, leading to a parser that will not always parse correctly. To deal with this, before writing YACC rules, debnf2pars changes its internal representation of the EBNF *`production`*s so that the YACC rules built directly from them recognizes the same grammar but does not have conflicts. This is complex. After that, there is the even more complex task of getting debnf2pars to build parse trees in terms of the C++ classes while parsing in terms of the modified EBNF.

13.10.2 General approach

The general approach to building dmis.y in debnf2pars is:

- Parse the DEBNF file and build a parse tree in terms of C++ structures for EBNF (defined in ebnf.hh),
- Convert all C++ structures for extensions to BNF (the E part of EBNF) to C++ structures for BNF.
- Modify the BNF structures to avoid shift/reduce conflicts.
- Print YACC from the modified BNF C++ structures.

The convert step above is possible because the extensions to BNF which are used in DEBNF can all be replaced by more verbose but equivalent BNF statements. The extensions to be removed are the *optional*s. debnf2pars replaces most of the *optional*s in the parse tree with their BNF equivalents by using multiple *definition*s, so that the parse tree becomes a BNF parse tree.

YACC rules are equivalent to BNF *`production`*s, so the rules in the YACC file would be relatively straightforward to print from the BNF parse tree if there were no actions following the rules. However, there are actions after all rules. Since the actions build a parse tree in terms of C++ classes generated from the unmodified EBNF *`production`*s while the rules are for the modified *`production`*s, generating the YACC file is very complex. The steps of the general approach become somewhat interleaved, not fully sequential.

The function call hierarchy in debnf2pars.y headed by *printYacc* is shown in Figure 18 and Figure 19. Functions belonging to classes are omitted from the hierarchy, and if a function appears more than once, its subordinates are shown only the first time it appears.

```
printYacc
  printYaccMiddle
    printYaccUnionAndTypes
      findToken
  printYaccProductions
    printYaccFirstProduction
      makeNewDefs
      prepareDefsForPlain
        prepareDefForPlain
          flattenOpts
      printYaccFirstAction
        printYaccActionItem
      printYaccRule
        printYaccExpression
    printYaccProduction
      printYaccDatMinor
        printYaccExpression
      printYaccForFixList1
        prepareDefsForPlain
        printYaccActionItem
        printYaccRule
      printYaccForFixList2
        printYaccForFixDef1
          prepareDefsForPlain
          printYaccActionItem
          printYaccRule
        printYaccForFixDef2
          prepareDefsForPlain
          printYaccActionItem
          printYaccRule
      printYaccForListDefault
      printYaccForNewDefs
        findProd
        printYaccRule
        printYaccAction
      printYaccForPlain
      printYaccForSupertype
      printYaccLabelDefinition
      prodIsLabel
    printYaccProductionsStart
      findProd
  printYaccStart (see Figure 19)
```

Figure 18. Hierarchy of Functions Headed by printYaccProductions

- *printYaccStart*
  - *printStarLine*
  - *printYaccDoCall*
  - *printYaccDoLabel*
  - *printYaccDoMacro*
  - *printYaccFindCallArgs*
  - *printYaccFindMacro*
  - *printYaccFindMacroArgs*
  - *printYaccGetStatement*
  - *printYaccGlobals*
  - *printYaccHandleLabel*
  - *printYaccIncDefs*
  - *printYaccInsertCalledMacro*
  - *printYaccIsCall*
  - *printYaccIsEndmac*
  - *printYaccIsMacro*
  - *printYaccMacroClass*
  - *printYaccParseDmis*
  - *printYaccPreprocess*
  - *printYaccResetParser*
  - *printYaccStringClasses*
  - *printYaccWarn*
  - *printYaccYyerror*

Figure 19. Hierarchy of Functions Headed by printYaccStart

In addition to generating a lot of YACC code, debnf2pars writes C++ code in dmis.y for 5 classes, 16 auxiliary functions, and 23 global variable declarations. All of that, except for one declaration, is hard-coded[1] in the subordinates of *printYaccStart* shown in Figure 19.

The classes are:
- macroList
- macroListCell
- macroRep
- stringList
- stringListCell

The functions are:
- *doCall* (returns int)
- *doLabel* (returns nothing)
- *doMacro* (returns nothing)

1. It might be a good idea to move the hard-coded material from dmis.y into a dmisStart.cc file and have dmis.y #include dmisStart.cc (as has been done with dmisConformanceTester.cc and dmisConformanceChecker.cc). Then all the functions that print the hard-coded material could be removed from debnf2pars.y.

- *findCallArgs* (returns int)
- *findMacro* (returns macroRep *)
- *findMacroArgs* (returns int)
- *getStatement* (returns bool)
- *handleLabel* (returns nothing)
- *insertCalledMacro* (returns nothing)
- *isCall* (returns int)
- *isMacro* (returns int)
- *parseDmis* (returns nothing)
- *preprocess* (returns nothing)
- *resetParser* (returns nothing)
- *warn* (returns nothing)
- *yyerror* (returns int)

The functions that deal with `MACRO`s and `CALL`s are all subordinate to the *preprocess* function, which is described in Section 3.4.1 of the Users Manual.

While most YACC rules are printed into dmis.y from *productions* without taking special actions based on the *production* name or *fixType*, there are dozens of special cases. The amount of code in debnf2pars.y for handling the special cases is large, probably larger than the code for non-special cases. There are enough special cases that some general approaches to types of special case have been developed. One of these is described in Section 13.10.7.

13.10.3 Printing YACC Rules

You won’t understand this section unless you are familiar with YACC. Refer to “lex & yacc” as needed.

The standard terminology for talking about YACC is deficient because it talks about actions corresponding to rules. This is inappropriate because a rule includes everything starting with the name on the left hand side and ending with a semicolon and may include many *definition*/action pairs. In most of what is written about YACC, “rule” sometimes means the entire rule, and sometimes means only the *definition* part of the rule. In the following discussion, to avoid confusion, rule means the whole thing, and *definition* means a sequence of tokens to be recognized.

The principal job of debnf2pars in printing the dmis.y file is to write it so that the parser built from it parses correctly and builds a parse tree as it parses. Most of the rules dmis.y are devoted to that task. Whenever a *definition*, D, is matched by the parser, a value which is an instance, K, of a C++ class, k, is assigned to the left side of the rule, and a pointer to K is passed to some other rule closer to the root of the tree. K is built by calling the constructor for k. The arguments to the k constructor are selected from the values for the components of D.

13.10.3.1 Writing basic rules

Consider, for example, the rule shown in Figure 20. The *definition* in the rule (second line) is all YACC. The action in the rule (third line) is all C++ except for the terms starting with *$*, which are YACC. The *$$* is the value for *intFuncIndx* that will be passed up to some other rule. The *$3* stands for the value of the third *expression* in the *definition*, which is a *stringVal*. The *$5* stands for the value of the fifth *expression* in the *definition*, which is another *stringVal*. Thus, this rule is recognizing the sequence of 6 tokens in the *definition* and building an instance of the *intFuncIndx* class from them. Note that the constructor does not need any information from the other terms in the *definition*. They are the same in every case of a call to `INDX` in a DMIS file, so there is no point in recording them[1].

```
intFuncIndx :
        INDX LPAREN stringVal C stringVal RPAREN
          { $$ = new intFuncIndx($3, $5); }
       ;
```

Figure 20. YACC Rule for intFuncIndx

The name of the constructor to use in the action (*intFuncIndx*) is the value of the *className* attribute of the *production*, which has been assigned previously.

The arguments to the constructor must also be found. All of the class constructors take as arguments the values of the attributes of the class in the order in which they occur in the class. The attributes of a class represent the non-constants in the EBNF *definition* from which the class was generated, in the order in which they appear in the EBNF (after optionals have been flattened). The *definition* in the YACC rule is the same as that EBNF, except for format. As a result of those facts, in the example of Figure 20, it is easy to determine what the arguments to the constructor should be. Just count *expression*s in the *definition*, and, if the nth *expression* is not a constant, use *$n* as an argument.

There are no functions in debnf2pars.y that implement this default case. The behavior just described is what the functions described in the next section do when there are no *optional*s in a *production*.

13.10.3.2 Writing rules for productions containing optionals

The descriptions in this section are complex because the code is (necessarily) complex. You might want to look at the examples in before tackling the text.

Unlike EBNF, YACC does not have special notation for an optional. To represent a simple optional, it is necessary to write two YACC *definition*s, one with the optional, and one without it. In the action following the *definition* in which the optional is not used, the constructor still requires an argument value for the optional, and that value must be 0 (a null pointer). Each top-level optional in an EBNF *definition* doubles the number of YACC *definition*s required to represent the same thing. For nested optionals the factor is 1.5 rather than 2 since the inner optional has no effect unless the outer optional is used. Dmis.y has rules with as many as 48 *definition*s where the EBNF has one *definition* with six optionals.

In order to print YACC rules, a data representation is needed that will support the following three activities:

- generating multiple YACC *definition*s to replace EBNF *definition*s with optionals,
- printing YACC *definition*s,

1. To reprint DMIS, the functions that do the printing must know about the constants. In the test suite, the printSelf functions know about them. In some other systems that generate parsers automatically, the print functions do not know about the constants, so the constants must be recorded in the C++ classes, making the classes unwieldy.

- printing YACC actions.

Since the EBNF `definition`s are represented in C++ as lists of *expression*s (*expList*s), the obvious way to handle the problem is to make modified copies of the *expList*s. The modified *expList*s must support:

- representing *true* and *false* for *optional*s containing only constants,
- representing null pointers for other *optional*s,
- generating correct values of N for the $N arguments to class constructors.

That method has been implemented. When the *fixType* of the *production* is *fixTypeNone*, it is implemented by *printYaccForPlain* and four subordinates: *makeNewDefs, prepareDefsForPlain*, *printYaccRule*, and *printYaccAction*. *PrintYaccForPlain* prints the YACC rule for one *production*.

PrintYaccForPlain first calls *makeNewDefs,* which makes the *newDefs* of each *definition* of the *production*. It initializes the *newDefs* list of a *definition* D by putting a copy of D at the beginning of the list.

PrintYaccForPlain then goes through the *definition*s of the *production*. For each *definition*, D, it calls *prepareDefsForPlain*, which acts on the newDefs of D.

PrepareDefsForPlain goes through the *definition*s in the newDefs of D (which may grow). It keeps calling *prepareDefForPlain* as long as *prepareDefForPlain* makes any changes to the *newDefs* list. After that, *prepareDefsForPlain* goes on to the next *definition*. *PrepareDefForPlain* acts only on the first *optional* it finds in the *expressions* of the *definition*. Since a *definition* may contain several *optional*s, it may be necessary to call *prepareDefForPlain* several times before no *optional*s remain in the *definition* on which it is acting. See the in-line documentation of *prepareDefForPlain* for details.

PrepareDefForPlain deals with an *optional* by making a copy of the *definition* containing the *optional* and inserting the copy in the *newDefs* list immediately after the original. In the original, it removes the *optional* completely and substitutes either a single *falseExp* (if the *expressions* of the *optional* are all constants) or one or more *nullExps* (otherwise). In the copy, it replaces the *optional* with the *expressions* of the *optional*. If the *expressions* of the *optional* are all constants, *trueExp* is inserted immediately before the *expressions*.

For each *definition* d in the newDefs of D, *printYaccForPlain* does the following:

- To help with checking labels as described in Section 13.10.7, possibly change some *expression*s in d.
- Call *printYaccRule* to print the YACC *definition* corresponding to d.
- If any *expression*s were changed to help with label checking, change them back to they way they were.
- Call *printYaccAction* to print the action for d.

PrintYaccRule goes through the *expression*s of d. It calls *printYaccExpression* to print each of them except for any occurrences of *nullExp*, *trueExp*, and *falseExp*. It does not print *nullExp* and *falseExp* because they represent *expression*s that do not appear in the rule. It does not print *trueExp* because *trueExp* refers to the constant *expression*s that follow it that used to be part of an *optional*, which will be printed.

PrintYaccAction also goes through the *expressions* of d. It uses a counter to count positions in the *expressions*. *PrintYaccAction* sets the position counter to 1, prints the beginning of a call to a class constructor, and then for each *expression* in d calls *printYaccActionItem* to possibly print an argument of the constructor and possibly update the position counter. For each *expression* E in d:

- If E is *nullExp*, 0 is printed in the constructor arguments representing a null pointer argument. The position counter is not incremented because nothing was printed for *nullExp* in the YACC *definition*.
- If E is *falseExp*, *false* is printed in the constructor arguments representing a false boolean argument. The position counter is not incremented because nothing was printed for *falseExp* in the YACC *definition*.
- If E is *trueExp*, *true* is printed in the constructor arguments representing a true boolean argument. The position counter is not incremented because nothing was printed for *trueExp* in the YACC *definition*.
- If E is a TERMINAL or NONTERMINAL (a non-constant item) $N is printed in the constructor arguments, where the value of N is the current value of the position counter, and then the counter is incremented.
- If E is anything else, it is a constant, so no constructor argument is printed, but the position counter is incremented because the constants were printed in the YACC *definition*.

The steps in the progression from EBNF *production* to YACC rule for `rmeasSpecVecbldOrient` are shown in Figure 21. This is the simplest possible example in which an *optional* containing only constants is used[1].

In Figure 21, the single *definition* in the EBNF *production* at the top is processed to give the two EBNF *definition*s in the middle, which are the *newDefs* of the *definition* at the top. This is done in *printYaccForPlain* which first calls *makeNewDefs* then calls *prepareDefsForPlain*, which calls *prepareDefForPlain* three times.

The *definition*s in the middle of Figure 21 are then processed to give the YACC rule at the bottom.

- The second line of YACC is printed from the upper middle *definition* by *printYaccRule*.
- The third line of YACC is printed from the upper middle *definition* by *printYaccAction*.
- The fourth line of YACC is printed from the lower middle *definition* by *printYaccRule*.
- The fifth line of YACC is printed from the lower middle *definition* by *printYaccAction*.

1. Of course debnf2pars is acting on the C++ representation of the EBNF `production`, but there is no print representation of that. The font for C++ is used in the discussion.

```
rmeasSpecVecbldOrient =
        rmeasSpecVecbld, [c, ORIENT]
      ;
```

```
        rmeasSpecVecbld, falseExp
        rmeasSpecVecbld, trueExp, c, ORIENT
```

```
rmeasSpecVecbldOrient :
          rmeasSpecVecbld
            { $$ = new rmeasSpecVecbldOrient($1, false); }
        | rmeasSpecVecbld C ORIENT
            { $$ = new rmeasSpecVecbldOrient($1, true); }
        ;
```

Figure 21. EBNF Production to YACC Rule for rmeasSpecVecbldOrient

An example involving nested *optional*s that are not all constants is shown in Figure 22. The steps in the progression from EBNF *production* to YACC rule for `promptIntEnd` are shown in the figure. The first *definition* in the EBNF *production* at the top is processed to give the first three EBNF *definition*s in the middle (which are the *newDefs* of the first *definition* at the top). The second *definition* at the top is processed to give the fourth *definition* in the middle (which is the *newDefs* of the second *definition* at the top). The same functions are involved as in Figure 21, but here *prepareDefsForPlain* is called twice (once for each *definition* at the top), and *prepareDefForPlain* is called six times (five to make the first three *definitions* in the middle, and once for the fourth *definition*).

The eight lines in the middle of the YACC rule are built by processing the four *definition*s in the middle. For each *definition*, first *printYaccRule* is called and then *printYaccAction* is called.

```
promptIntEnd =
        stringVal, [c, intVal, [c, intVal]]
      | promptItemList
      ;
```

```
        stringVal, nullExp, nullExp
        stringVal, c, intVal, nullExp
        stringVal, c, intVal, c, intVal

        promptItemList
```

```
promptIntEnd :
          stringVal
            { $$ = new promptIntEnd_stringVal($1, 0, 0); }
        | stringVal C intVal
            { $$ = new promptIntEnd_stringVal($1, $3, 0); }
        | stringVal C intVal C intVal
            { $$ = new promptIntEnd_stringVal($1, $3, $5); }
        | promptItemList
            { $$ = new promptIntEnd_promptItemList($1); }
        ;
```

Figure 22. EBNF Production to YACC Rule for promptIntEnd

13.10.3.3 Writing rules for productions that are lists in default form

If the *isList* attribute of a *production* is *true* and its *fixType* is *fixNone*, then it is a list in default form, and *printYaccProduction* calls *printYaccForListDefault* to print the rule. The default form of the EBNF for a comma-separated list is shown at the top of Figure 23. The YACC rule that would be printed by *printYaccForListDefault* is shown at the bottom of the figure. In the first *definition* and action of the rule, the *thing* is the first item on the list, so a new list of *thing* is made, and the *thing* is put at the end of the list (which is also the front, since there is only one element). In the second *definition* and action of the rule, the beginning of the list already exists and *thing* comes after that, so the *thing* is inserted at the end of the list.

For a list that is not comma-separated, everything is the same except that the `, c` in the EBNF at the top is not there, and the *C* in the second *definition* of the rule is not there.

```
thingList =
    [thingList, c], thing
      ;
```

```
thingList :
        thing
           { $$ = new std::list<thing *>;
             $$->push_back($1); }
        | thingList C thing
           { $$ = $1;
             $$->push_back($3); }
        ;
```

Figure 23. EBNF Production to YACC Rule for List

13.10.3.4 Writing rules for productions in which shift/reduce conflicts are fixed

If a *production* has had a shift/reduce conflict that is not the simplest type, the value of its *fixType* attribute will be one of those shown in the first column of Table 2, and the rule for it will be printed by the function in the second column. The third column gives an example of a *production* that has the given *fixType*. The fourth column gives the number of *production*s in dmis.debnf with that *fixType*. In the case of *fixListItemDeleted*, the *production* is not printed at all.

If a *production* has any of the last four *fixType*s in the first column, it will have been modified before printing YACC begins. In order to print the YACC rule's actions in terms of the C++ classes derived from the unmodified *production*s while the rule's *definition*s are printed from the modified *production*s, the functions in the second column are necessarily complex. They are all heavily documented in debnf2pars.y. The documentation includes examples.

Table 2. Printing Productions with Shift/Reduce Conflicts

fixType	top printing function	example production	productions with fixType
fixListItemDeleted	NA	displySpecItem	16
fixListItemsInserted1	printYaccForFixList1	snslctWristAngleList	1
fixListItemsInserted2	printYaccForFixList2	displySpecList	16
fixProdC	printYaccForNewDefs	datsetSpecC	5
fixProdCUser	printYaccForNewDefs	datsetDats	13

Here we describe only *printYaccForFixList2*, its subordinate *printYaccForFixDef2*, and its precursor *insertItemInList*. The other three functions listed in Table 2 are of the same general nature. Read the in-line documentation to get the details.

PrintYaccProduction will call *printYaccForFixList2* to print the YACC rule for a *production* if the *fixType* of the *production* is *fixListItemsInserted2*. That *fixType* is assigned only in *insertItemInList*, which may have been called by any of three conflict-fixing functions.

InsertItemInList takes three arguments:

- *listItem* - a pointer to a *production* for the list item
- *theList* - a pointer to a *production* that is a list of the *listItem*
- *commaInside* - a boolean that is
  true if there should be a comma inside the recursive use of the list and
  false if there should be a comma outside the recursive use of the list.

InsertItemInList makes new *definition*s for the (only) *definition* of a list. The new *definition*s are right recursive with the comma inside the recursion if *commaInside* is *true* and outside (before) the recursion if *commaInside* is *false*, i.e. each new *definition* follows one of the following patterns:

- `aListItem, [c, theList]`
- `aListItem, c, [theList]`

Call the old *definition* of the list *oldDef* and the first new *definition* of the list *neoDef*. Call *oldDef->newDefs newDefs*.

InsertItemInList:

- creates *neoDef* by copying *oldDef* then removing the list item and reassembling what is left so the *neoDef* is in one of two forms: `[c, theList]` or `c, [theList]`
- creates the *newDefs*
- inserts *neoDef* as the first member of the *newDefs*
- puts as many copies of *neoDef* into *newDefs* as there are *definition*s of *listItem*.
- copies the *expressions* from each *definition* in *listItem->defs* onto the beginning of the corresponding *definition* in the *newDefs*
- sets the *className* of each *definition* in the *newDefs* to the *className* of the corresponding *definition* in *listItem->defs*. This is needed so that the correct class of list item can be instantiated in the actions.
- sets the *fixType* of *theList* to *fixListItemsInserted2*
- makes a *prodList* called *prodSet* of all the *production*s used in *newDefs*
- changes the *usedIn* list of each *production* in *prodSet* by adding *theList*
- removes *theList* from *listItem->usedIn*
- if *listItem->usedIn* is then empty: (1) sets *fixType* of *listItem* to *fixListItemDeleted* and (2) removes *listItem* from the *usedIn* list of each *production* in *prodSet*.

Figure 24 shows, as an example, the action of *insertItemInList* on *displySpecList*.

Before:

```
 displySpecList =
        [displySpecList, c], displySpecItem
      ;

   displySpecItem =
        device, c, DMIS, [c, vLabel]
      | device, c, vLabel
      ;
```

After

```
   displySpecList =
       device, c, DMIS, [c, vLabel], [c, displySpecList]
      | device, c, vLabel, [c, displySpecList]
      ;
```

Figure 24. Example of Effect of InsertItemInList

Note that when *insertItemInList* is done, the *newDefs* of the one *definition* of *theList* have been set.

PrintYaccForFixList2 goes through the *newDefs* and, for each of them, calls either *printYaccForFixDef2* (15 of the 16 times it is called) or *printYaccForFixDef1* (1 of the 16 times). Only *printYaccForFixDef2* is described here.

The most important argument to *printYaccForFixDef2* is one *definition* D of a list (D is any of the *newDefs* mentioned above). *PrintYaccForFixDef2* makes the *newDefs* for D by calling *prepareDefsForPlain*. Then it goes through the *newDefs* of D and prints a YACC *definition* (by calling *printYaccRule*) and a YACC action for each one (call it d). The actions differ according to whether d has the recursive list (which will be at the end of the *expressions* of d in that case) or not.

What *printYaccForFixDef2* prints for the first *definition* in the “After” *production* shown in Figure 24 is shown in Figure 25. Since that *definition* has two *optional*s, four *definition*/action pairs are printed.

So that *printYaccForFixDef2* knows what type of list to make, the class name of the *production* that is the list item (*displySpecItem* in Figure 25) is passed to it as an argument. The class name of D (*displySpecItem_1* in Figure 25) is also passed in so that the function will know what kind of item to make and add to the list.

```
 device, c, DMIS
   { $$ = new std::list<displySpecItem *>;
     $$->push_front(new displySpecItem_1($1, 0)); }
| device C DMIS C displySpecList
   { $$ = $5;
     $$->push_front(new displySpecItem_1($1, 0)); }
| device C DMIS C vLabel
   { $$ = new std::list<displySpecItem *>;
     $$->push_front(new displySpecItem_1($1, $5)); }
| device C DMIS C vLabel C displySpecList
   { $$ = $7;
     $$->push_front(new displySpecItem_1($1, $5)); }
```

Figure 25. Example of What printYaccForFixDef2 Prints

13.10.4 Reporting errors

Section 3.6 of the Users Manual describes what the parser's error and warning messages mean; read that first.

Error reporting in dmis.y is handled using the built-in facilities provided by YACC. If a YACC parser receives an input token from the lexer that does not match any rule:

- It calls the *yyerror* function.
- It discards any partially parsed rules until it returns to a state in which it could shift the special "error" symbol.
- It resumes parsing, starting by shifting an "error" token. YACC rules are included in dmis.y as described in Section 13.10.5 that match the "error" token.

The *yyerror* function defined in dmis.y reports the lexer message in *lexMessage* buffer if there is one and reports the error message generated automatically by the parser if not. Then it prints the line of text on which the error occurred, up to the point at which the error occurred. The text is found in the *lineText* buffer where the lexer has stored it.

The lexer and the parser collaborate in implementing error reporting. Many errors are detected by the lexer defined in dmis.lex. Whenever the lexer detects an error, it puts an error message in the *lexMessage* buffer and returns BAD. When the parser gets the BAD token, it finds that no YACC rule uses it and goes through the steps in the bullets above.

The only role of debnf2pars in this is to print the hard-coded text in dmis.y that makes it happen.

13.10.5 Continuing through errors

See the "lex & yacc" book regarding *error* and *yyerrok* in YACC.

To continue through errors, the parser looks for any parser error followed by the end of a statement. If that is found, the parser is reset by the code shown in Figure 26, which consists of a YACC *definition* using the special YACC symbol *error* and an action in C++ code. The action adds one to *numErrors*, resets global variables *aLabelFound* and *setLabelType* to what they need to be at the beginning of a line, and uses *yyerrok*. After that action, the parser is ready to continue at the beginning of the next line of DMIS code.

The code shown in the figure is inserted by debnf2pars at the end of the YACC code for the following YACC rules: *dmisFirstStatement*, *dmisFreeStatement* (which covers 179 types of DMIS statement), *endfilStm*, *endgoStm*, *endmesStm*, *endselStm*, *endsimreqtStm*, and *endxtnStm*. It would be good if that code were inserted at the end of the rule for every DMIS statement, but for the following rules, inserting the code shown in the figure produces a reduce/reduce conflict with *dmisFreeStatement*, so the code is not inserted: *caseStm*, *dftcasStm*, *endcasStm*, *doStm*, *enddoStm*, *ifStm*, *endifStm*, *endmacStm*, and *extfilStm*[1].

```
error ENDLINE
{
   numErrors++;
   yyerrok;
   aLabelFound = 1;
   setLabelType = 0;
}
```

Figure 26. Error Handling Code in YACC

13.10.6 Checking labels

The parser defined in dmis.y checks for undefined and multiply defined labels in DMIS input files.

In DMIS, most labels are defined by a DMIS statement that sets a label equal to something. The label being defined is on the left side of an equal sign, and its definition is on the right side. The items on the right side may include labels, and these must have been defined previously. The label on the left side is not defined until the end of the right side is reached. Except for feature labels, a label may not be defined more than once in a DMIS input file. Labels are used in many DMIS statements that do not define labels, and any label used in those statements must already have been defined. There is one exception to this. That is in the DMIS `EXIST` function which tests whether a label is defined. It is not a parse error if a label being tested in an `EXIST` function is not defined. This exception is dealt with using the *handleLabel* function in dmis.y.

The *doLabel* function and the YACC actions in dmis.y that deal with labels use the global variables *aLabel*, *aLabelFound*, *aLabelType*, *setLabel*, and *setLabelType*. The name of a label that needs to be remembered is placed in *aLabel* and the type of the label is placed in *aLabelType*.

The *doLabel* function is called by the actions for all rules that read labels. The action that calls *doLabel* places the type of the label in the *labelType* argument of *doLabel* and the name of the

1. It should be possible to eliminate the reduce/reduce conflict problem by inserting the code after every statement that is a *dmisFreeStatement* rather than at the end of *dmisFreeStatement*. However, there are nearly 200 kinds of *dmisFreeStatement*, so that would be a big pain.

label in the *labelName* argument. At the time *doLabel* is called, if *aLabelFound* is set to 0, that indicates a label error has already occurred in the DMIS statement being parsed. In this case *doLabel* does nothing so that the existing error will be remembered.

If there is no previous label error in a statement, then *doLabel* sets *aLabelType* to the *labelType* argument and sets *aLabel* to the *labelName* argument. Also, *aLabelFound* is reset to 1 if a label of the given *labelType* and *labelName* has already been recorded and to 0 if not. *ALabelFound* being set to 0 does not indicate an error at this point.

If an equal sign is encountered after a label, the value of *aLabelFound* should be 0, since the equal sign indicates that the label is being defined. If the value is 0, *setLabel* is set to *aLabel*, *setLabelType* is set to *aLabelType*, and *aLabelFound* is set to 1. If the value is not 0 and the definition is not for a feature, a warning message is issued indicating the label may be multiply defined.

When the end of line is reached, if *setLabelType* is not 0, that indicates a label has been defined, so the *setLabel* is recorded, and *setLabelType* is set back to 0. Also, if *aLabelFound* is 0, a warning message is issued indicating an undefined label may have been used, and *aLabelFound* is reset to 1.

Some labels in DMIS are not defined using an equal sign but by some other construction. This includes labels defined by `CONST/SGAGE`, `CONST/SPART`, `DATDEF`, or `DATTRGDEF`. For those statements, special actions for label checking are triggered by a comma rather than an equal sign.

13.10.7 A YACC technique used for label checking

The label checking described in the previous section requires that global variables be set and reset in the process of parsing part of a DMIS statement represented by a single YACC rule (call the rule R). It is possible to do this using mid-rule actions, but that is cumbersome. A better method used in debnf2pars is to print a new rule in dmis.y that is an alias for the *expression* at which an action should occur, attach the action to the new rule, and substitute the alias for the *expression* in R at which the action should be taken.

For example, the expected YACC rule for the DMIS `OPERID` statement would be:
opLabel EQUALS OPERID SLASH stringVal ENDLINE
but a label checking action should be taken immediately after the *EQUALS* is parsed, and another label checking action should be taken after *ENDLINE*.

To handle the situation, two YACC rules are printed that are not in the EBNF: *equalSign* as an alias for *EQUALS* and *endOfLine* for *ENDLINE*. Then the YACC rule for the `OPERID` statement is written using the aliases:
opLabel equalSign OPERID SLASH stringVal endOfLine

The rules for *equalSign* and *endOfLine* are followed by actions that issue warning messages if there is a problem and then reset the global variables used in label checking.

Using aliases this way does not confuse the YACC processor. The YACC processor would signal a reduce/reduce conflict if there were two aliases for the same thing (such as *EQUALS : '='* and *equalSign : '='*), but since the aliases are in series (*EQUALS : '='* and *equalSign : EQUALS*), not in parallel, no conflict arises.

The following rules, all of which are used for label checking, are printed by debnf2pars at the beginning of the YACC rules in dmis.y.

- *endOfLine* (an alias for *ENDLINE*)
- *equalSign* (an alias for *EQUALS*)
- *defCheckComma* (an alias for a comma)
- *undefCheckComma* (an alias for a comma)
- *existLParen* (an alias for a left parenthesis)
- *existRParen* (an alias for a right parenthesis)

See the documentation of *printYaccProductionsStart* in debnf2pars.y for more details.

13.11 Generating dmis.lex

Much of the printing of dmis.lex is hard coded. That is, most of the text that ends up in dmis.lex may be found as strings in debnf2pars.y. The top level *printLex* function in debnf2pars.y:

- calls *printLexStart*
- calls *printLexMiddle*
- calls *printLexToken* for every entry in the *tokenNames* array
- calls *printLexEnd*

Several issues lead to complexity in dmis.lex. Descriptions of the issues and how they are handled in the printing of dmis.lex follow. The handling of several of the issues is done by using states in dmis.lex. In total, 12 states are used. They are printed into dmis.lex near the end of *printLexMiddle*. Line continuations and comments in DMIS files will be removed by the preprocessor, so dmis.lex does not have to handle them.

13.11.1 Using the pre-processed DMIS file

The lexer built from dmis.lex reads the pre-processed DMIS file, not the original. The pre-processed file has a line number at the beginning of each line. The line numbers need to be read (and stored in *lineNo*). That is handled in dmis.lex by code printed by *printLexEnd*. The preprocessor also handles `MACRO` and `CALL` specially; see Section 13.11.4.

13.11.2 Strings in DMIS

Strings in DMIS start and end with a single quote character, but if there are two single quote characters together inside a string, that represents a single quote that is part of the string. The INSTRING state is used in dmis.lex for handling strings. The code for reading strings is printed by *printLexEnd*.

13.11.3 Communication between dmis.y and dmis.lex

There is a fair amount of interplay between the parser built from dmis.y and the lexer built from dmis.lex. This is implemented by 8 shared variables (*resetLex*, *lexMessage*, *lexWarning*, *lineText*, *lineNo*, *getCallArgs*, *inDecl*, *macroIsReal*). These variables are set, tested, and used in dmis.y and dmis.lex[1]. They are declared as extern in dmis.lex. Printing those declarations is done near the beginning of *printLexMiddle*.

1. One of them, macroIsReal, seems to have been made obsolete as a variable in dmis.y and dmis.lex (it is set in both but neither uses it). The name macroIsReal, however, has migrated to dmis.cc where it is still needed, so getting rid of its obsolete uses must be done carefully.

13.11.4 MACROs

When the lexer built from dmis.lex encounters a `MACRO` statement in a preprocessed DMIS input file, it treats the lines of the `MACRO` as strings until it hits the `ENDMAC` line that ends the `MACRO`. So that the lines of the `MACRO` can be used without case confusion when the `MACRO` is `CALL`ed, they are converted to upper case (except in quoted strings) and saved. The preprocessor has inserted a line number before each line, and that also needs to be handled. `MACRO`, `ENDMAC`, and the lines in between are handled in dmis.lex by code printed by *printLexEnd*. Three states are involved: MACROIN, MACROLINE, and MACROLINENUM.

13.11.5 Scope of variables

A DMIS `MACRO` may include `DECL` statements that declare variables. The preprocessor puts the text of a `MACRO` into every `CALL` to the `MACRO` with the arguments to the `MACRO` replaced by the arguments to the `CALL`. The lexer built from dmis.lex treats the lines of `CALL` blocks as normal DMIS. For the purposes of checking for multiply declared and undeclared variables, it is necessary to keep track of the scope of variables inside of `CALL`s. Since a `CALL` block may include another `CALL`, a stack of scopes must be maintained. The code for this in dmis.lex is written by *printLexMiddle*. It includes defining three classes (*varAndType*, *scope*, and *scopeList*) and using a *scopeStack* global variable.

13.11.6 Label names

Dealing with labels in dmis.lex is a big problem. There is a 2-page discussion of this in the documentation of *printLexToken*. Three states are involved in handling labels in dmis.lex: READLABEL, READ2, and AT2. The text for handling labels is printed by *printLexEnd*.

13.11.7 DMIS/OFF

When a `DMIS/OFF` command is read by the lexer built from dmis.lex, all the lines following that in the original DMIS input file are uninterpretable strings, up to a `DMIS/ON` command. The strings need to be saved so they can be reprinted. However, the preprocessor has inserted a line number before each line which is not part of the string, and that needs to be handled. `DMIS/OFF` and `DMIS/ON` and the lines in between are handled by code printed by *printLexEnd*. Three states are involved: DMISOFFIN, DMISOFFLINE, and DMISOFFLINENUM.

13.11.8 Arguments to CALL

The arguments to a `CALL` statement (which have already been inserted in the text of the `CALL`ed `MACRO` by the preprocessor) are treated as a string in the lexer built from dmis.lex. To enable this, the CALLARGS state is used in dmis.lex. The text using CALLARGS is printed by *printLexEnd*.

14 Building the Generators

Source code for the generator is in the generator directory. The structure of that directory is shown in Figure 27. There is no binLinux or binSun because the executables that would go in those directories are built directly in the subdirectories with those names in the utilityComponents/linuxSun directory.

```
generator
   linuxSun
      ofilesLinux
      ofilesSun
      source
   windows
      debnf2pars
         debnf2pars
         Debug
         Release
      generateMore
         Debug
         generateMore
         Release
      source
```

Figure 27. Generator Directory Structure

14.1 Building debnf2pars

The debnf2pars executable is built starting with the following hand-written files found in generator/linuxSun/source or generator\windows\source:

- debnf2pars.y
- debnf2pars.lex
- ebnfClasses.hh (.h for windows)
- ebnfClasses.cc (.cpp for windows)

The first two are not C++ files, but C++ files are created from them. This may be done the same way in Linux and Sun. Windows is the same except for using backslashes instead of slashes and working from a Windows directory.

To build debnf2parsYACC.cc and debnf2parsYACC.hh from debnf2pars.y, get into the generator/linuxSun directory and give the command:

bison -d -l -o source/debnf2parsYACC.cc source/debnf2pars.y

To build debnf2parsLex.cc from debnf2pars.lex, get into the generator/linuxSun directory and give the command:

flex -L -t source/debnf2pars.lex > source/debnf2parsLex.cc

14.1.1 Linux

The utilityComponents/linuxSun/binLinux/debnf2pars executable file is built by getting into the generator/linuxSun directory and executing

make ../../utilityComponents/linuxSun/binLinux/debnf2pars

If debnf2parsYACC.cc (and debnf2parsYACC.hh) or debnf2parsLex.cc does not yet exist, the ***make*** command starts by calling bison and/or flex as described above. Once those files exist, the ***make*** command calls the compiler to:

- compile debnf2parsYACC.cc into debnf2parsYACC.o,
- compile debnf2parsLex.cc into debnf2parsLex.o,
- compile ebnfClasses.cc into ebnfClasses.o,

Those files are put into the generator/linuxSun/ofilesLinux directory.

Then the ***make*** command calls the linker to link the three object files into the debnf2pars executable and puts it in the utilityComponents/linuxSun/binLinux directory.

14.1.2 Sun

The utilityComponents/linuxSun/binSun/debnf2pars executable file is built by getting into the generator/linuxSun directory and executing

make ../../utilityComponents/linuxSun/binSun/debnf2pars

If debnf2parsYACC.cc (and debnf2parsYACC.hh) or debnf2parsLex.cc does not yet exist, the ***make*** command starts by calling bison and/or flex as described above. Once those files exist, the ***make*** command calls the compiler to:

- compile debnf2parsYACC.cc into debnf2parsYACC.o,
- compile debnf2parsLex.cc into debnf2parsLex.o,
- compile ebnfClasses.cc into ebnfClasses.o,

Those files are put into the generator/linuxSun/ofilesSun directory.

Then the ***make*** command calls the linker to link the three object files into the debnf2pars executable and puts it in the utilityComponents/linuxSun/binSun directory.

14.1.3 Windows

The utilityComponents\windows\bin\debnf2pars.exe executable file was built in the generator\windows\debnf2pars\Release directory using the Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).

14.2 Building generateMore

The generateMore executable is built starting with generator/linuxSun/source/generateMore.cc or generator\windows\source\generateMore.cpp.

14.2.1 Linux

The utilityComponents/linuxSun/binLinux/generateMore executable file is built by getting into the generator/linuxSun directory and executing

make ../../utilityComponents/linuxSun/binLinux/generateMore

The ***make*** command calls the compiler to compile generateMore.cc into generator/linuxSun/ofilesLinux/generateMore.o, and then calls the linker to link the object file into the generateMore executable and puts it in the utilityComponents/linuxSun/binLinux directory.

14.2.2 Sun

The utilityComponents/linuxSun/binSun/generateMore executable file is built by getting into the generator/linuxSun directory and executing

make ../../utilityComponents/linuxSun/binSun/generateMore

The ***make*** command calls the compiler to compile generateMore.cc into generator/linuxSun/ofilesSun/generateMore.o, and then calls the linker to link the object file into the generateMore executable and puts it in the utilityComponents/linuxSun/binSun directory.

14.2.3 Windows

The utilityComponents\windows\bin\generateMore.exe executable file was built in the generator\windows\generateMore\Release directory using the Microsoft Visual C++ 2008 Express Edition and then copied to that file. For instructions on compiling in Windows, see the System Builders Manual (Section 1.4.3 and Appendix A).