Users Manual for Version 2.1.5 of the NIST DMIS Test Suite (for DMIS 5.1)

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 7603
August 19, 2009

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

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 70NANB6H013.

Table of Contents

1 Introduction

1.1 Overview

This manual is a users manual for the NIST DMIS Test Suite, version 2.1.5. The test suite[1] is intended to serve two purposes:

- to help users and vendors use version 5.1 of DMIS (the Dimensional Measuring Interface Standard),
- 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.

The test suite and this manual were prepared at the National Institute of Standards and Technology (NIST). There is also a "System Builders Manual for Version 2.1.5 of the NIST DMIS Test Suite (for DMIS 5.1)"[2] The purpose of the system builders manual is to help system builders use software provided in the test suite for building systems that implement DMIS. The test suite, which includes both manuals, may be downloaded from

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

In addition, 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.

DMIS is the only international standard language (ISO 22093) for input files (programs) used for the control of dimensional measuring equipment, coordinate measuring machines in particular. It is also approved by the American National Standards Institute (ANSI). The most recent version of DMIS approved by the Dimensional Metrology Standards Consortium (DMSC), the organization that maintains DMIS, is DMIS 5.1. Copies of the standard on CD are available for purchase at http://www.dmisstandards.org/store.

Because DMIS is a very large language, and only subsets of it need to be implemented for many applications, subsets called conformance classes have been defined. To conform to a conformance class, a system using DMIS must fully implement the subset for that class. The DMSC has defined two "application profiles", one for Prismatic parts and one for Thin Walled parts. Seven addenda have also been defined. Each application profile and addendum may be implemented at three levels. Level 2 includes everything in level 1, plus additional items. Level 3 includes everything in level 2, plus additional items. The four conformance classes in the test suite are full DMIS and levels 1, 2, and 3 of the Prismatic application profile, which we will call prismatic1, prismatic2, and prismatic3.

EBNF (Extended Backus-Naur Form) is a standard formal language for defining the syntax of a language. Annex C of the DMIS standard is an EBNF definition of the syntax of the DMIS input language. The term DEBNF (short for DMIS EBNF) is used in this manual to mean the dialect of EBNF used in DMIS 5.1. Details are given in Section 6.

---

1. In the remainder of this manual "the test suite" means the NIST DMIS Test Suite, version 2.1.5.
2. In the remainder of this manual "the system builders manual" means the System Builders Manual for Version 2.1.5 of the NIST DMIS Test Suite (for DMIS 5.1).

As shown in Figure 1, the test suite has eight directories.

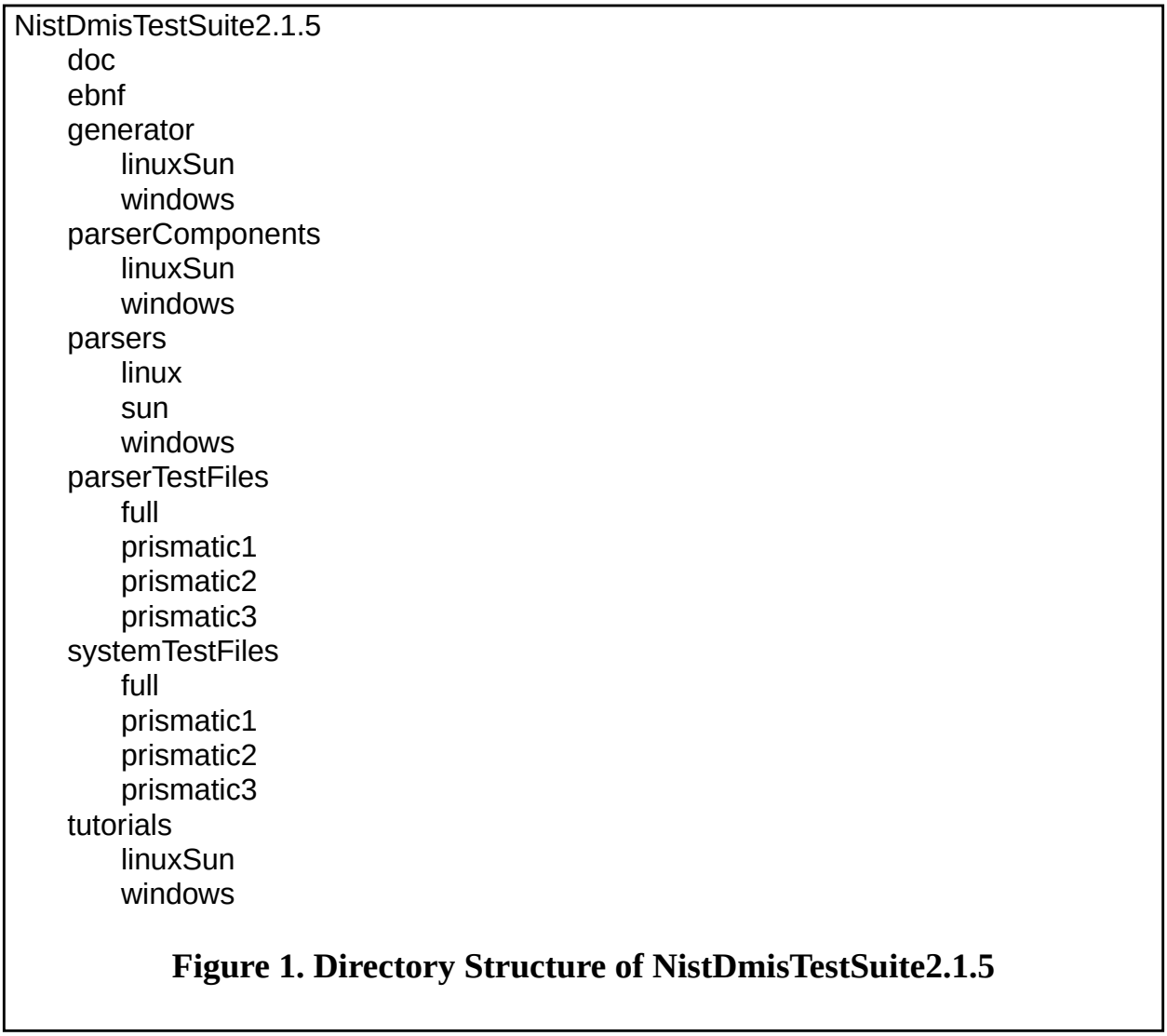


Figure 1. Directory Structure of NistDmisTestSuite2.1.5

Briefly, these contain the following.

The doc directory has this users manual, the system builders manual, and an Excel spreadsheet defining conformance classes.

The ebnf directory includes a DEBNF file for each of the four conformance classes.

The generator directory contains a system named debnf2pars for automatically generating all the code in the parserComponents directory. The input to debnf2pars is a DEBNF file.

The parserComponents directory has software from which parsers can be built for the four conformance classes and three operating systems. The software includes C++ classes for representing DMIS as well as parsers.

The parsers directory contains executable parsers for the four conformance classes and three

operating systems (Linux, SunOS, and Windows). It also contains scripts and data files for testing parsers. The parsers are made from the code in the parserComponents directory.

The parserTestFiles directory has a lot of DMIS input files for testing parsers. These are syntactically correct but do not necessarily make sense as programs.

The systemTestFiles directory has a modest number of DMIS input files that should be executable on commercial DMIS execution systems.

The tutorials directory contains:

- one tutorial (makeBound) showing how to use the C++ classes for building a single line of DMIS code,
- one tutorial (generate) showing how to use the C++ classes for building a DMIS input file generator, and
- one tutorial (analyze) showing how to use the parser and the C++ classes for building a DMIS input file consumer.

This manual describes how to use the test suite for various purposes.

The largest shortcomings of the test suite are:

- The current test suite tests only the compliance of DMIS input files to the requirements for input file syntax. There is no checking whether the input files describe useful, logical, or realizable measurement operations.
- There is currently no test utility for verifying the compliance of a DMIS execution system to the requirements for system behavior.
- Only four conformance classes are covered.
- For testing completeness of a DMIS generator, only counting the number of times each DMIS statement is used is implemented. For a robust completeness test, it would be necessary to implement counting the number of times every definition in the DEBNF file for a conformance class is used.
- The parsers do not attempt to load files referenced by INCLUD or EXTFIL, so the parsers may report errors that would not occur if such files were used as provided by those DMIS statements.
- The parsers check all files lines sequentially, so if a file uses flow of control statements (such as IF/ELSE/ENDIF or JUMPTO) that cause a file to be executed out of order or cause some statements not to be executed, the parsers may report warnings incorrectly or may fail to detect errors for which warnings should be given.
- The parsers do not evaluate variables, so labels that are created or referenced using the @variable method may cause the parser to report label warnings incorrectly.

1.2 Downloading and Installing the Test Suite

The test suite may be downloaded from:

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

Select and download the file NistDmisTestSuite2.1.5.zip.

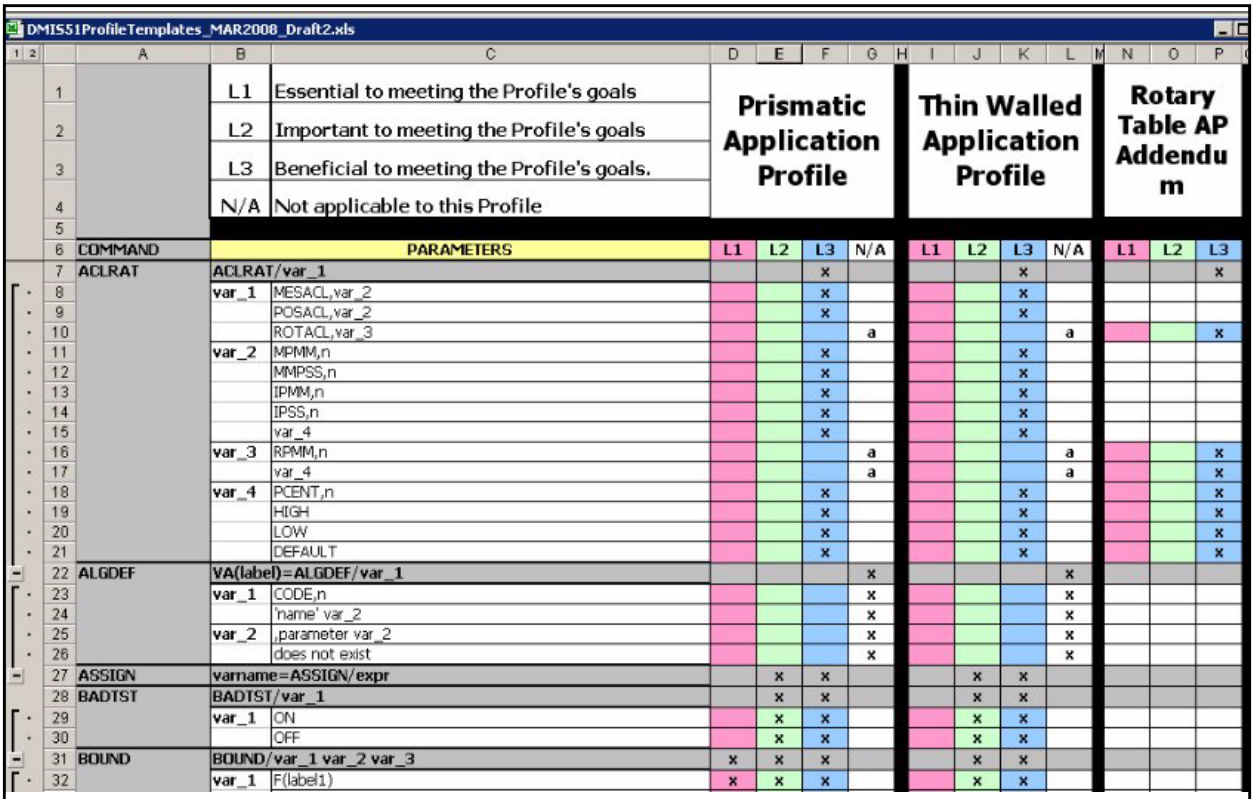
Unzip the file.

The top level directory that will be created is named NistDmisTestSuite2.1.5. The structure of this directory is shown in Figure 1 above.

1.3 Documentation

The doc directory contains a copy of this users manual, a copy of the system builders manual, and an Excel spreadsheet defining the DMIS conformance classes. The spreadsheet is the same as the one in release 2.1.4 of the Test Suite. The doc directory does not contain a copy of the ISO standard for EBNF (ISO/IEC 14977), but that may be downloaded free of charge from ISO at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

1.3.1 How is the Excel spreadsheet defining conformance classes arranged?



DMIS51ProfileTemplates_MAR2008_Draft2.xls

1		L1	Essential to meeting the Profile's goals	Prismatic Application Profile					Thin Walled Application Profile				Rotary Table AP Addendum		
2		L2	Important to meeting the Profile's goals												
3		L3	Beneficial to meeting the Profile's goals.												
4		N/A	Not applicable to this Profile												
5															
6	COMMAND	PARAMETERS		L1	L2	L3	N/A	L1	L2	L3	N/A	L1	L2	L3
7	ACLRAT	ACLRAT/var_1				x				x				x
8		var_1	MESACL,var_2			x				x				
9			POSACL,var_2			x				x				
10			ROTACL,var_3				a				a			x
11		var_2	MPMM,n			x				x				
12			MMPSS,n			x				x				
13			IPMM,n			x				x				
14			IPSS,n			x				x				
15			var_4			x				x				
16		var_3	RPMM,n				a				a			x
17			var_4				a				a			x
18		var_4	PCENT,n			x				x				x
19			HIGH			x				x				x
20			LOW			x				x				x
21			DEFAULT			x				x				x
22	ALGDEF	VA(label)=ALGDEF/var_1					x				x			
23		var_1	CODE,n				x				x			
24			'name' var_2				x				x			
25		var_2	,parameter var_2				x				x			
26			does not exist				x				x			
27	ASSIGN	varname=ASSIGN/expr			x	x			x	x				
28	BADTST	BADTST/var_1			x	x			x	x				
29		var_1	ON		x	x			x	x				
30			OFF		x	x			x	x				
31	BOUND	BOUND/var_1 var_2 var_3		x	x	x			x	x				
32		var_1	F(label1)	x	x	x			x	x				

Figure 2. Excel SpreadSheet

The arrangement of the Excel spreadsheet defining conformance classes is largely self-evident, so it is only summarized here. Figure 2 shows the left side of the beginning of the spreadsheet.

The spreadsheet has a main row for each subsection of section 6 of DMIS 5.1. Each row is divided into subrows that follow the structure of the syntax described in the corresponding subsection of DMIS 5.1.

The spreadsheet has 11 main columns, the first five of which are showing on Figure 2. The first main column has the DMIS statement names. For each statement, the second column has a list of the parameters used by the statement, one subrow for each parameter. The rest of the columns also have these subrows.

Columns 3 and 4 are for the Prismatic Application Profile and the Thin Walled Application Profile. These two columns are each divided into four subcolumns. The first three subcolumns are

for the three levels at which the two application profiles may be implemented. An **x** in a subrow and subcolumn means the parameter in the subrow must be implemented at the level of the subcolumn. The meaning of fourth subcolumn is a little trickier. An **x** in the fourth subcolumn means the parameter does not have to be implemented in any application profile or addendum. An **a** in the fourth subcolumn means that the parameter must be implemented in at least one addendum (and in this case an **x** will be found in the same subrow for one or more of the addenda).

The last seven columns of the spreadsheet are for the addenda, and each one is divided into three subcolumns for the levels. An **x** in any of these columns means the parameter must be implemented. The addenda are:

- Rotary Table (the only one showing on Figure 2),
- Multi Carriage,
- Contact Scanning,
- In-Process Verification (IPV),
- Quality Information System (QIS),
- Measurement Uncertainty,
- Soft Gaging

1.4 Arrangement of this Manual

The remaining sections of this manual are in order of decreasing interest. Section 2 (“Parsers”), Section 3 (“Parser Test Files”), and Section 4 (“System Test Files”) are expected to be of interest to almost all test suite users. Section 5 (“Parser Components”) and Section 6 (“EBNF”) are expected to be of interest to fewer users. Very few users are expected to have interest in Section 7 (“Generator”). Appendix A is included for conformance testers. Appendix B describes how to use the Microsoft Visual C++ 2008 Express Edition to compile from source code. As mentioned earlier, the C++ classes and the tutorials are covered in the system builders manual.

The more esoteric sections start with a subsection, “Who should read this section”, describing the circumstances in which the section might be of interest.

1.5 Changes from Version 2.1.4

Major changes have been made from the NIST DMIS Test Suite Version 2.1.4 (the most recent earlier release). Version 2.1.5 does not change the way the test suite can be used for conformance testing; that is why only the last digit of the version number has been changed. The changes from version 2.1.4 are:

1.5.1 DEBNF

1.5.1.1 strange constructs

In Version 2.1.4, the DEBNF files contained strange constructs needed for avoiding conflicts if translated directly into YACC. In Version 2.1.5, the strange constructs have been eliminated so that the DEBNF is more natural. This is desirable because the C++ classes should be natural and are generated directly from the DEBNF.

1.5.1.2 callBlock

The DEBNF files in Version 2.1.4 did not define callBlock (because there is no such thing in DMIS). The DEBNF files in Version 2.1.5 do define callBlock (because callBlocks, which are

inserted by the preprocessor, allow DMIS MACROs to be parsed when they are called). If a DMIS input file is printed back out again from its parse tree, the callBlocks disappear.

1.5.2 Generator

1.5.2.1 C++ classes

Version 2.1.4 did not generate C++ classes. Version 2.1.5 does.

1.5.2.2 DEBNF conversion

While generating YACC, Version 2.1.5 converts natural DEBNF that makes YACC conflicts if translated directly into YACC into strange DEBNF constructs that do not make conflicts when translated directly into YACC. Version 2.1.4 did not have this conversion because the DEBNF already contained the necessary strange constructs.

1.5.2.3 better parsers generated

As described below, the parsers built by the Version 2.1.5 generator are better than those built by the Version 2.1.4 generator.

1.5.2.4 functionality divided

In Version 2.1.4, only two files were generated for parsing (one YACC file and one Lex file), and the function of counting DMIS statements was built into the YACC file. In Version 2.1.5, three files are generated for parsing (one YACC file, one Lex file, and one C++ file). The function of counting DMIS statements has been removed from the YACC file and put into the C++ file. Now, a parser that builds a parse tree but does not count DMIS statements is built from the YACC and Lex and saved in a library. The parser in the library is compiled together with the new C++ file to build an executable parser that counts DMIS statements. The parser in the library may be built into other applications that are DMIS consumers.

1.5.3 Parser Components

1.5.3.1 C++ classes

Version 2.1.5 includes C++ classes. Version 2.1.4 did not.

1.5.3.2 more kinds of file

As noted above, Version 2.1.5 has three files (YACC, Lex, and C++) where Version 2.1.4 had two (YACC and Lex).

1.5.4 Parsers

1.5.4.1 MACRO and CALL

The Version 2.1.5 parsers handle MACRO and CALL better than did the parsers in Version 2.1.4. MACROs are now parsed only when CALLed.

1.5.4.2 DMIS/ON,OFF

The Version 2.1.5 parsers handle DMIS/OFF and DMIS/ON better than did the Version 2.1.4 parsers.

1.5.4.3 parse tree

Each Version 2.1.5 parser builds a parse tree while parsing. The Version 2.1.4 parsers did not

build parse trees.

1.5.4.4 pretty printing

In Version 2.1.5, the C++ classes include functions that can pretty print a DMIS input file from its stored parse tree. As described in Section 2.3, when the executable parser is used to parse a single DMIS input file, the file is pretty printed back out again. Version 2.1.4 did not include pretty printing.

1.5.5 Tutorial Programs

Version 2.1.4 had no tutorial programs. Version 2.1.5 has three.

- makeBound: uses C++ classes to generate one line of DMIS code.
- generate: uses C++ classes to generate an entire DMIS input file.
- analyze: uses the parser utility and C++ classes in a DMIS consumer.

1.5.6 Documentation

Version 2.1.5 includes a DMIS system builders manual. Version 2.1.4 had no such manual.

1.5.7 Testing

1.5.7.1 cycle testing added

In a cycle test: (1) a DMIS input file is parsed then printed out again from the parse tree, (2) the original input file is reformatted, and (3) it is checked that the reformatted file and the reprinted file are identical. In Version 2.1.5, all of the scripts that test a set of DMIS files run a cycle test on every error-free test file and all such test files pass the cycle test. Version 2.1.4 had no cycle test.

1.5.7.2 reformatter

Version 2.1.5 includes a reformatter for running cycle tests. Version 2.1.4 had no reformatter.

2 Parsers

2.1 What is in the parsers directory?

The parsers directory contains executable parsers and files that test the parsers.

2.2 How is the parsers directory arranged?

The structure of the parsers directory is shown in Figure 3 below. There is a subdirectory for each of three operating systems: Linux, SunOS, and Windows XP. Each of these has a subdirectory for each of the four conformance classes. Each of those subdirectories contains:

- an executable parser (dmisFullParser, for example),
- a test file that runs a set of test files through the parser (testFullParser, for example),
- a file containing a list of parser test file names (runAllFull, for example),
- a file containing what the parser should print when it processes the preceding file (runAllFullOut, for example),
- a file containing a list of names of DMIS input files that should be executable on a DMIS execution system and do not produce motion (runAllFullSysNo, for example),
- a file containing what the parser should print when it processes the preceding file (runAllFullSysNoOut, for example).

Each subdirectory of the windows subdirectory also contains two auxiliary .bat files used by the

test...Parser file.

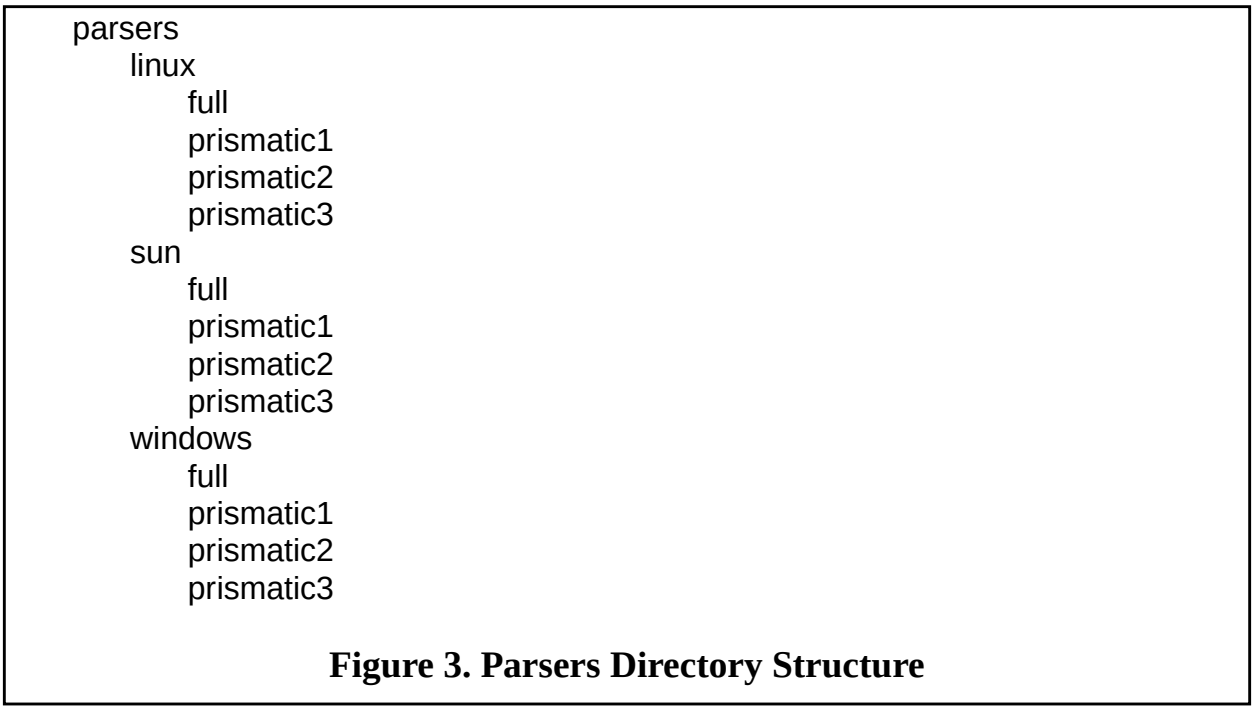


Figure 3. Parsers Directory Structure

2.3 How do I use a parser to parse a single DMIS input file?

For all three operating systems, you run a parser by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

In all cases, a parser takes one or two command arguments. The first argument is the path to the DMIS input file to parse. If there is a second argument, it is the name of the file into which the parser should write its messages. If there is no second argument, the parser writes its messages in the command window.

2.3.1 Linux and SunOS

Example 1. The following command should be given in the parsers/linux/full directory if you are using Linux or in the parsers/sun/full directory if you are using SunOS. The command parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the command window, and reprints the text of the input file in the command window. The reprinting is done using the parse tree built during parsing.

./dmisFullParser ../../../parserTestFiles/full/okInFull/units1.dmi

Example 2. The following command, given in the same directory as Example 1, parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the file u1.out, and reprints the text of the input file in the command window.

./dmisFullParser ../../../parserTestFiles/full/okInFull/units1.dmi u1.out

Example 3. The following command, given in the same directory as Example 1, parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the file u1.out, and reprints the text of the input file in the file u1.dmi.

./dmisFullParser ../../../parserTestFiles/full/okInFull/units1.dmi u1.out > u1.dmi

2.3.2 Windows

Example 1. The following command, given in the parsers\windows\full directory, parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the command window, and reprints the text of the input file in the command window. The reprinting is done using the parse tree built during parsing.

dmisFullParser.exe ..\..\..\parserTestFiles\full\okInFull\units1.dmi

Example 2. The following command, given in the parsers\windows\full directory, parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the file u1.out, and reprints the text of the input file in the command window.

dmisFullParser.exe ..\..\..\parserTestFiles\full\okInFull\units1.dmi u1.out

Example 3. The following command, given in the parsers\windows\full directory, parses the DMIS input file units1.dmi, writes the message “0 errors 0 warnings” in the file u1.out, and reprints the text of the input file in the file u1.dmi.

dmisFullParser.exe ..\..\..\parserTestFiles\full\okInFull\units1.dmi u1.out > u1.dmi

2.4 How can I use a single command to parse a whole set of DMIS input files?

There are two ways to run a set of DMIS input files through a parser, as described in Section 2.4.1 and Section 2.4.2. In both methods, you type a command in a command window.

2.4.1 First method for running a set of DMIS input files through a parser

In the first method:

- A new parser process is used for each file.
- The message output of the parser is compared with the expected message output.
- If no error or warning messages are given by the parser, the parsed file is printed back out again, the input file is reformatted so that it is in the format used by the parser’s printer, and a check is made that the file printed by the parser is identical to the reformatted file.
- The test stops at the first file for which there is a difference between the actual and expected message output or between the printed and reformatted files, if there is any input file for which that happens.

The name of each executable test file implies that the parser is being tested. This is correct when you have downloaded the test suite and are testing to see if it runs on your computer. However, once you are satisfied that the parser is working on your computer, you can make a copy of the executable test file and edit the names of the directories and the files, so that the copy can be used for testing whatever set of DMIS input files you want to test.

2.4.1.1 Linux and SunOS

In the parsers/linux and parsers/sun directories, an entire set of DMIS input files may be parsed with the help of a shell script.

To run all the parser test files in the Test Suite for a particular conformance class through a parser, get into the directory containing the parser for that conformance class and give a command that is the name of the executable parser test file, which is one of:

testFullParser
testPrismatic1Parser
testPrismatic2Parser
testPrismatic3Parser

The testFullParser script processes 312 DMIS input files. It takes 20 seconds or so to run on the Dell Precision 670 PC running Linux on which the Linux executables were built and about 24 seconds on a Sun Fire V215 running SunOS.

2.4.1.2 Windows

In the Windows XP operating system, an entire set of DMIS input files may be parsed with the help of a batch file.

To run all the parser test files in the Test Suite for a particular conformance class through a parser, get into the directory containing the parser for that conformance class and give a command that is the name of the executable parser test file, which is one of:

testFullParser.bat
testPrismatic1Parser.bat
testPrismatic2Parser.bat
testPrismatic3Parser.bat

The testFullParser.bat batch file processes 312 DMIS input files. It takes about 155 seconds to run on the Dell Dimension 8300 PC running Windows XP on which the executable was built.

2.4.2 Second method for running a set of DMIS input files through a parser

In the second method:

- Only one parser process runs (so the test is faster), and it parses all the files.
- The test stops if a test file cannot be found but does not stop if there is a parse error.
- No comparison is done.
- The parser counts the number of times each DMIS statement in the conformance class is used and reports that information after it parses all the files.
- The parser reports the percentage of DMIS statements in the conformance class that were used in at least one file.
- The parser reports the names of those DMIS statements in the conformance class that were not used in any of the files.

You can make a file listing the names of the DMIS input files you want to test and use this method to test them. The file names must include the path (either a relative path starting from the directory containing the parser or an absolute path).

2.4.2.1 Linux and SunOS

To run a set of DMIS input files through a parser for a particular conformance class, get into the directory containing the parser for that class and give a command that is the name of the parser followed by the name of a file containing the names of the DMIS files. The names of the DMIS files must end in .dmi. If another argument is given, it will be used as the name of a file, and the printing will go that file. If no additional argument is given, printing will go to the command window.

Example 1. The following command should be given in the parsers/linux/full directory if you

are using Linux or in the parsers/sun/full directory if you are using SunOS. The command runs all the DMIS input files in the parserTestFiles/full directory through the dmisFullParser. Output printing goes to the command window:

./dmisFullParser runAllFull

Example 2. To run the same test with output printing going to the file named results, give the following command. After the test runs, you can use "diff" to compare the results file with the runAllFullOut file, which contains what results should be.

./dmisFullParser runAllFull results

2.4.2.2 Windows

To run a set of DMIS input files through a parser for a particular conformance class, get into the directory containing the parser for that class and give a command that is the name of the parser followed by the name of a file containing the names of the DMIS files. The names of the DMIS files must end in .dmi. If another argument is given, it will be used as the name of a file, and the printing will go that file. If no additional argument is given, printing will go to the command window.

Example 1. The following command should be given in the parsers\windows\full directory. The command runs all the DMIS input files in the parserTestFiles\full directory through the dmisFullParser. Output printing goes to the command window:

dmisFullParser.exe runAllFull

Example 2. To run the same test with output printing going to the file named results, give the following command. After the test runs, you can compare the results file with the runAllFullOut file, which contains what results should be. In Windows, it does not matter whether or not you include the .exe suffix in the name of the command.

dmisFullParser runAllFull results

2.5 How do I modify a command that runs a set of test files so that it tests my parser?

This is possible only if your parser is an executable that takes arguments and can be run from a command window. If you are using Linux or SunOS you need to know how to write a script file; on Windows you need to know how to write a batch file.

2.5.1 Linux and SunOS

In Linux or SunOS, if you have a parser for full DMIS, for example, start by copying the testFullParser script file to testMyFullParser (or whatever name you prefer). Edit testMyFullParser by deleting either the "runOK" function or the "runOut" function defined near the beginning of the file and editing the other one so that:

- It calls your parser instead of dmisFullParser.
- It takes the arguments your parser takes.
- It performs the output test(s) you want.

Several other items may be need to be changed, particularly near the beginning where variables are defined and in the last section where files with syntactic errors are tested.

2.5.2 Windows

In Windows if you have a parser for full DMIS, for example, start by copying (using different

names for the new files if you prefer):

- testOneFull.bat to testMyOne.bat
- testOneFullOK.bat to testMyOneOK.bat
- testFullParser.bat to testMyParser.bat

Edit testMyOne.bat by changing the file so that:

- It calls your parser instead of dmisFullParser.
- It takes the arguments your parser takes.
- It performs the output test you want (the original calls fc and looks for an error condition).

Edit testMyOneOK.bat by changing the file so that:

- Five lines are deleted, the first of which is the line containing "reformat".
- The definition of OK on the first line refers to a file containing what your parser prints if it parses a file with no errors.
- It calls your parser instead of dmisFullParser.
- It takes the arguments your parser takes.
- It performs the output test you want (the original calls fc and looks for an error condition).

Edit testMyParser.bat by changing the file so that:

- The first six lines starting with "set" at the top are what you want them to be.
- The setting of TESTOK on line 8 refers to testMyOneOK.bat.
- The setting of TEST on line 9 refers to testMyOne.bat.
- It calls TESTOK to parse files expected to have no errors.
- It calls TEST to parse files expected to have errors.

2.6 What does the executable parser do when it parses a single DMIS input file?

If the first argument to the command to run the parser ends in .dmi, the parser assumes the file is a DMIS input file and does the following.

When the parser runs, it runs in two stages: preprocess and parse. Wherever there may be an error in the DMIS input file, an error or warning message is printed preceded by the line number (in the input file) of the line that caused the problem. In the parse phase, the text of the line that caused the problem is also printed, up to the point at which the problem occurred. See Section 2.9 for further details and examples.

2.6.1 Preprocess

The preprocessor reads the DMIS input file and writes the file PrEpRoCeSsDmIs. In general, one line of input becomes one line of output, except that comments and blank lines are deleted, continued lines are concatenated together (with the continuation signs removed), and CALL statements are modified as described below.

The length of every line of the input file is checked. If a line is more than 65536 characters long, the parser prints an error message and quits.

Each line of the input file is checked to be sure there is a carriage return followed by a line feed at the end. If one of those characters is found but the other is missing, an error message is printed, but the missing character is inserted where it belongs and processing continues.

If a comment line (one starting with $$) follows a continued line (one ending with $), an error message is printed, and processing continues.

The line number from the input file is inserted at the beginning of each output line. Where there are blank lines or continuation lines in the input, those line numbers do not appear in the output.

In order to check MACROs, when the preprocessor reads a DMIS MACRO, it reprints the MACRO in the preprocessed file and saves the text of the MACRO. At the point where the MACRO is CALLed, the text of the MACRO is inserted in the preprocessed file with the arguments to the CALL substituted for the MACRO arguments. This creates a callBlock starting with the CALL statement and ending with the ENDMAC statement copied in from the MACRO. For more details, see Section 6.6 and the in-line documentation in parserComponents/linuxSun/full/source/dmisFull.y of the functions isMacro, isCall, findMacroArgs, findCallArgs, insertCalledMacro, doCall, and doMacro.

2.6.2 Parse

In the parse phase, the parser reads and parses the PrEpRoCeSsDmIs file. While the parser parses, it builds a parse tree in terms of the C++ classes that represent DMIS. Details of parse tree structure are given in the system builders manual.

Error and warning messages in the parse phase are generated either by the parser built by bison or the lexical analyzer built by flex. Some of the error and warning messages from the lexical analyzer identify a very specific error (such as “variable reused”) but others are generic. Error messages from the parser built by bison tend to be generic and are often not intuitively clear.

When the parser parses the preprocessed file and reads a DMIS MACRO, the parser makes no attempt to parse it (the parser cannot parse it because the types of the arguments are not specified). But when the parser parses a CALL to the MACRO, it does parse it (in the CALL, the types of the things that replaced the MACRO arguments are known). If an error occurs while a CALLed MACRO is being parsed, the line number given is the line number of the CALL, not the line number of the MACRO.

At the end of the parse phase, a summary message of the form “N errors M warnings” is printed and the PrEpRoCeSsDmIs file is removed.

2.7 What does the executable parser do when it parses a set of DMIS input files?

If the first argument to the command to run the parser does not end in .dmi, the parser assumes the file contains a list of names of DMIS input files. The parser handles the files in the list in the order given and does the following for each.

First, the parser checks that the name ends in .dmi and checks that the file exists. If either of these checks fails, the parser prints an error message and quits.

Then the parser processes the file as described in Section 2.6, but in addition, it counts the number of times each type of DMIS statement (DMISMN, FEAT, CONST, etc.) is used.

After parsing all the DMIS input files listed, the parser prints the number of times each type of DMIS statement in the conformance class was used. It also prints the percentage of DMIS statements in the conformance class that were used. See Section 2.10 for an example.

2.8 For what purposes may the executable parsers be used?

The executable parsers may be used to determine if a file that is supposed to be a DMIS input file conforms to the DMIS 5.1 standard. This check should be useful in the following situations.

A user or conformance tester wants to determine if a DMIS input file generator generates files that conform to the DMIS 5.1 standard. The user employs the generator to generate a set of DMIS input files and then runs the input files through the parser. If an error occurs, the generator is not fully conformant. If a warning occurs, the generator might not be fully conformant. The number and type of errors and warnings that occur will give the user an idea of whether the generator is usable. If the input file generator is supposed to conform to one of the DMIS conformance classes, the user selects the parser for that conformance class.

A developer wants to check that the DMIS input file generator he or she is building is generating files that conform to DMIS 5.1. As development proceeds, the developer repeatedly generates files and runs them through the parser.

A user has a DMIS input file that runs on DMIS execution system X. System X becomes unavailable and the user would like to run the file on system Y, which is known to conform to DMIS 5.1. The user runs the file through the parser to determine if it conforms. If it does (and system Y is known to have adequate size, accuracy, probes, etc.), it will run on system Y. If it does not, it won't run on Y.

A user has a DMIS input file and set of CMMs that implement different DMIS conformance classes. The user wants to know which of the CMMs can execute the input file. The user runs the file through a parser for each CMM's conformance class. If an error occurs in the parser for a class, the file will not run on a CMM that implements that class.

The (non-executable) parsers in library files may be built into DMIS systems as described in the system builders manual.

2.9 What do the parser's error and warning messages mean?

If a parser runs into an error during the preprocessing stage, it will print an error message and fix the problem if it can. If the input file or preprocessing output file cannot be opened or if the parser reads a line more than 65536 characters long, it will print an error message and quit. Every error message that can be sent during preprocessing is easy to understand. If there is a preprocessing error, the line number of the line causing the error will be printed, but the line will not be printed. For example the following preprocessor error message means that line 3 is the last line of the file and the endline character is missing from that line.

```
3: Error - no endline on last line of file
```

If a parser runs into an apparent error while parsing the preprocessed file, it will print two lines: (1) the line number from the input file followed by either an error message or a warning message, and (2) the text of the line on which the error occurred, but only up to the point where an error was detected. For example, the following two lines mean that an error was found on line 7, and the error is that a MACRO has two arguments that are the same. The actual line 7 may be longer, but the parser stopped parsing the line when it ran into the error at the second x.

```
3: argument reused
M(featdef)=MACRO/x,x
```

The error message above was written by a human. The error messages written by a human are usually understandable with a little study. Often it will help to refer to the text of the DMIS 5.1 standard.

One type of error that may be puzzling even though a human wrote the message occurs when a DMIS major or minor word is used that is not in the conformance class of the parser. In this case, the parser will not recognize the word as a DMIS word, and will try to make sense of it some other way. For example, ROTAB is a DMIS word not in the Prismatic Level 2 conformance class. If the dmisPrismatic2Parser reads a file with a ROTAB statement in it, it will assume that ROTAB is the name of a variable to which a value is about to be assigned, check its list of DECLared variables, find that ROTAB has not been DECLared, and emit the following message.

```
4: Error - undefined variable
ROTAB
```

Many error messages are machine-authored, not written by a human. The machine-authored error messages are generated automatically on the fly by the parser and may be difficult to understand. They all start with “syntax error”. Here is a machine-authored example.

```
4: syntax error, unexpected RPAREN, expecting C
y = ASSIGN/MN(x)
```

In this example, the parser has been reading the ASSIGN statement and has just read the right parenthesis (known to the parser as RPAREN) following x. The parser knows that the MN function must have at least two arguments and the arguments are separated by commas (known to the parser as C). The parser was expecting to see a comma after x, but it saw a right parenthesis instead, so it stopped parsing the line and printed an error message.

The parser stops trying to parse a line as soon as it finds an error on the line, but then it tries to continue parsing starting with the next line. Usually this is successful, but sometimes the parser will become confused and start giving off incorrect error messages. This is most likely to happen if an error occurs on the first line of a block (such as in a MEAS or GOTARG statement). To deal with this problem, the error that caused the initial error message should be fixed in the DMIS input file, and the file should be run through the parser again.

Warning messages usually indicate errors. The parser can be tricked into thinking there is a problem for multiply DECLared variables, multiply defined labels, and references to undefined labels even if there is no problem. When it detects a problem of any of those types, since it cannot be sure there is an error, the parser emits a warning message rather than an error message. The percentage of cases in which a warning is not actually an error is very low, however.

2.10 What does the parser’s summary report mean?

As described in Section 2.7, the parser for any conformance class may be run by giving it the name of a file containing a list of names of DMIS input files to parse. If this is done, after parsing all the files, the parser prints a report giving (1) the names of the files that were parsed and the errors and warnings for each file, (2) the number of times each DMIS statement in the conformance class was used, (3) the percentage of DMIS statements in the conformance class that were used in any of the files that were parsed, and (4) the names of DMIS statements in the conformance class that were not used in any of the files that were parsed. Items 2 to 4 constitute a summary report.

Example 1 - Suppose a user using a Windows PC gives the following command from the parsers\windows\full directory:

dmisFullParser.exe runAllFull results

The parser will take a few seconds to run, and it will print a file 2422 lines long named “results”. The results file will have over 2000 lines giving the names of each DMIS input file parsed and the warnings and errors for each input file. After that, the results file will have the following summary report (where many lines have been omitted from the list of DMIS statement uses). The meaning of the summary report should be obvious. There are no names of DMIS statements that were not used at the end of the results file because all of the statements were used.

```
Total statement uses for all files
aclrat 11
algdef 4
assign 223
badtst 4
bound 4
calibMaster 1
calibRtab 3
calibSens 12
...
vform 30
windef 5
wkplan 5
wrist 22
write 10
xtern 5
xtract 6
100.0% of the commands in the conformance class were used.
```

Example 2 - Suppose a user using a Windows PC gives the following command from the parsers\windows\full directory:

dmisFullParser.exe ..\prismatic3\runAllPrismatic3 results

This is telling the full DMIS parser to parse the test files for the Prismatic Level 3 parser. There are lots of statements in full DMIS that are not in the Prismatic Level 3 conformance class. Again, the parser will take a few seconds to run. This time it will print a file 1757 lines long named “results”. This time, the file will have the following summary report (where many lines have been omitted from the list of DMIS statement uses). The summary report indicates that 67.5% of the statements in full DMIS were used, and there were a lot of statements in full DMIS that were not used.

```
Total statement uses for all files
aclrat 8
assign 212
badtst 4
bound 4
calibMaster 1
calibSens 5
```

```
call 18
...
vform 12
wkplan 5
wrist 7
write 10
xtern 5
xtract 6
67.5% of the commands in the conformance class were used.
The following commands were not used:
algdef          calibRtab       clmpid          clmpsn
cnfrmrul        crgdef          crmode          croscl
crslct          cutcom          czone           czslct
dmehw           dmeid           dmeswi          dmeswv
dmisOff         dmisOn          fildef          fixtid
fixtsn          gecomp          geom            group
litdef          lotid           mfgdev          operid
pameas          partid          partrv          partsn
path            planid          prevop          procid
qisdef          rapid           recallRotaryTablerefmnt
rotab           rotdef          rotset          saveRotaryTable
scnmod          snsgrp          tooldf          uncertalg
uncertset        windef
```

3 Parser Test Files

Note: In general, parser test files will cause errors in DMIS execution systems. Use files from the systemsTestFiles directory (see Section 4) if you want to test a DMIS execution system.

3.1 What are the parser test files like?

The files in the parserTestFiles directory conform to the syntax rules implicit in the EBNF for DMIS plus additional rules about variables and labels.

The syntax rules implicit in the EBNF include the syntax described in the “Input Formats” descriptions in section 6 of DMIS 5.1, plus the following:

- Integer values (not real values) must be used where DMIS requires integer values.

The additional rules stated in the text of DMIS 5.1 to which the parser test files conform are:

- All variables must be explicitly declared with DECL or implicitly declared by being parameters of a MACRO.
- A variable of a given name may not be declared twice with DECL or used twice as a parameter of the same MACRO.
- Wherever a statement or an expression requires a variable to be of a given type, the variable used must be of the correct type.
- Blocks of a given type may contain only those types of statements that are allowed in blocks of that type.
- All labels must be defined before they are referenced.

- A label of a given type may be defined only once, except for feature labels, since DMIS allows feature labels to be redefined.

Most of the files in the parserTestFiles directory are **not** suitable as test files for DMIS execution systems (systems that carry out the statements in a DMIS input file) because the files:

- may contain semantic errors, such as having a negative number where a positive number is required,
- may contain nonsense such as attempting to measure points on a feature that are not near the feature,
- may crash the equipment.

3.2 What do the “dmi” and “out” filename suffixes mean?

Most of the files in the parserTestFiles directory are DMIS input files and have the suffix .dmi. These are files for testing DMIS parsers.

The OK.out file contains two lines: “0 errors” and “0 warnings”. All the other files in the parserTestFiles directory that have the suffix .out are files containing the error and warning messages that the NIST DMIS parser prints when it parses a file with the same base name and a .dmi suffix. The files in this directory that have the suffix .out are **not** DMIS output files as defined in DMIS 5.1.

3.3 What is in the parserTestFiles directory?

As shown in Figure 4, the parserTestFiles directory has four subdirectories: full, prismatic1, prismatic2, and prismatic3. Each of them has five or six subdirectories, as described below. In the prismatic subdirectories:

- The subdirectory names have a P1, P2, or P3 suffix.
- The subdirectories contain fewer files than the corresponding subdirectories of the full directory.
- Many of the files are shorter than the corresponding files in the full directory.

If a test file in one directory has the same base name as a test file in another directory but with p1, p2 or p3 added (aclrat1.dmi and aclrat1p3.dmi, for example), the file with the p1, p2 or p3 near the end will be the same as the other file, except that items not required in the conformance class will have been removed.

1. The annexA… directories contain the examples from Annex A of DMIS 5.1. In most cases these have been modified by adding DMISMN and ENDFIL.

2. The okIn… directories contain one or more syntactically correct parser test files for each subsection of section 6 of DMIS 5.1 that must be implemented in the conformance class. The name of most of these files corresponds to the name of a DMIS statement. These directories also contain syntactically correct programs testing expressions and the preprocessor of the NIST DMIS parser.

3. The errorIn… directories contain parser test files with errors in them. The file names of all of these files include the string Error.

4. The errorOut… directories contain .out files corresponding to the files in the errorIn… directories. Each file contains the error and warning messages printed by the NIST DMIS parser when it parses the corresponding file from the corresponding errorIn… directory.

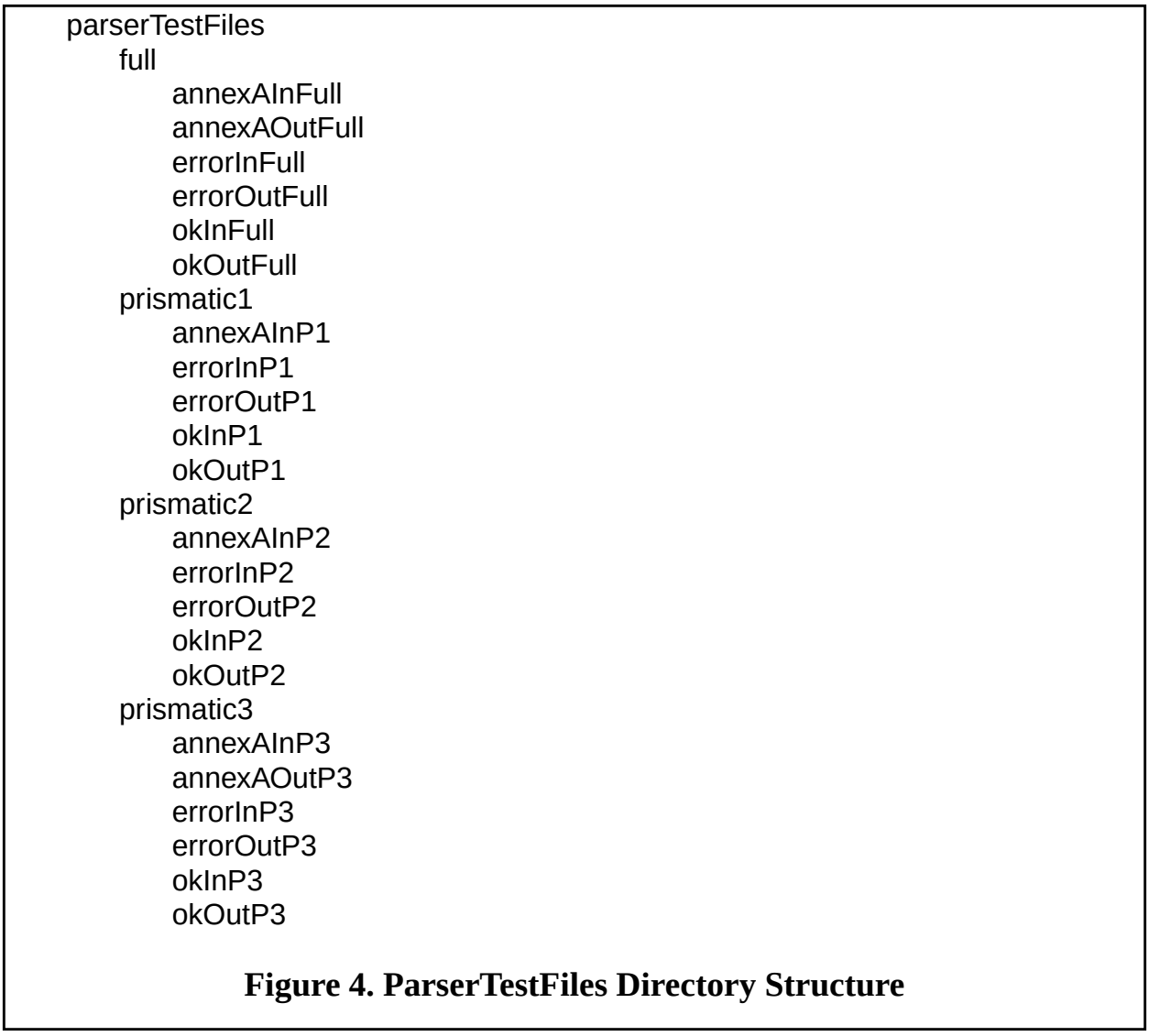


Figure 4. ParserTestFiles Directory Structure

3.4 For what purposes may the parser test files be used?

The parser test files may be used by developers building DMIS parsers to check that their parsers are working correctly. A method of automating this testing is described in Section 2.5.

The parser test files have been used at NIST for:

- debugging the NIST DMIS parsers,
- checking that the EBNF files defining the four conformance classes are correct.

See Section 2.3 and Section 2.4 for descriptions of how to use test files to test parsers.

4 System Test Files

4.1 What is in the systemTestFiles directory?

The systemTestFiles directory contains DMIS input files that may be safely run on a DMIS

execution system and should run without error.

As shown in Figure 5, the systemTestFiles directory has four subdirectories: full, prismatic1, prismatic2, and prismatic3. Each of them has two subdirectories, one for programs that produce no motion, and one for programs that produce motion.

Each okInNoMotion… subdirectory contains a subset of the files in the corresponding okIn… directory under the parserTestFiles directory (possibly modified). The subset consists of files which do not move the axes and do not make semantic errors.

The okInMotion… subdirectories contains files that move the sensor. These files contain instructions about how to use themselves, including suggestions for editing. The instructions should be followed. The files in these directories should only be run by a person who is knowledgeable about using the machine on which they are to be run.

The full/okInMotionFull subdirectory contains the files freeMotion.dmi, gohome.dmi, IMTS1.dmi, and simple1.dmi. The first two of these move the sensor without attempting to inspect anything. Each of the last two inspects a specific machined part, which must be available in order to execute the file.

The prismatic3/okInMotionP3 subdirectory contains the files freeMotionp3.dmi, gohome.dmi, IMTS1.dmi, and simple1.dmi.

The prismatic2/okInMotionP2 subdirectory contains the files freeMotionp2.dmi, IMTS1p2.dmi, and simple1p2.dmi.

The prismatic1/okInMotionP1 subdirectory contains only the file simple1p1.dmi.

As with the parser test files, a p1, p2, or p3 before .dmi means the file has been modified to be suitable for the prismatic1, prismatic2, or prismatic3 conformance class.

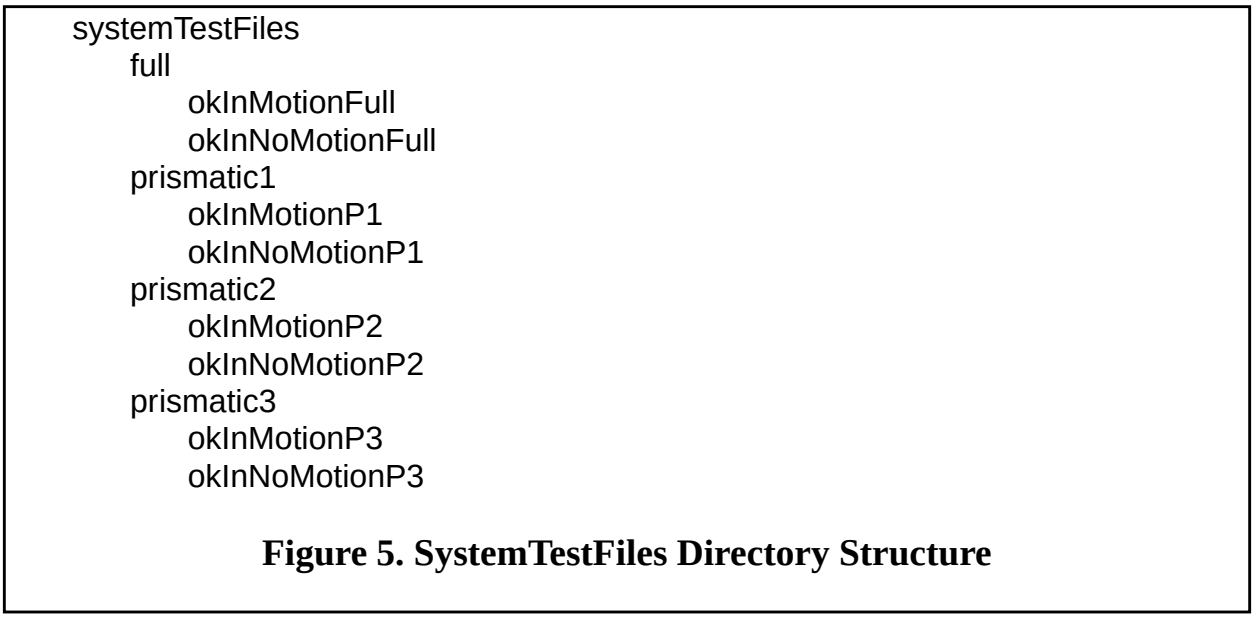


Figure 5. SystemTestFiles Directory Structure

4.2 For what purposes may system test files be used?

System test files may be used by users, conformance testers, or developers to determine if a DMIS execution system can parse and execute DMIS input files correctly. Although many files are

provided, they do not cover all of the functionality of a DMIS execution system. Moreover, the DMIS output files that should be generated when these input files are executed are not included in the test suite.

5 Parser Components

5.1 Who should read this section?

Read this section if you want to:

- understand how the test suite parsers were built,
- make a parser for a computer architecture not included in the test suite,
- build a parser starting from a DEBNF file not included in the test suite.

If you want to use any of the parser components in a DMIS generator or executor, see the system builders manual.

5.2 How is the parserComponents directory arranged?

As shown in Figure 6 and Figure 7, the parserComponents directory is divided at the top level by operating system. Linux and Sun are lumped together at this level since they use identical C++, YACC, and Lex files; Windows is separate. At the next level down, the directory is divided by conformance class: full, prismatic1, prismatic2, and prismatic3; in addition, this level has bin directories for the debnf2pars executables.

In Windows, each of the full, prismatic1, prismatic2, and prismatic3 directories has a source directory and two other directories, one whose name ends in “Classes” and one whose name ends in “Parser”. The ones ending in “Classes” contain the classes and also the YACC and Lex. The ones ending in “Parser” contain the statement counting parser. These directories were built by Visual C++ 2008 and are arranged as Visual C++ pleases.

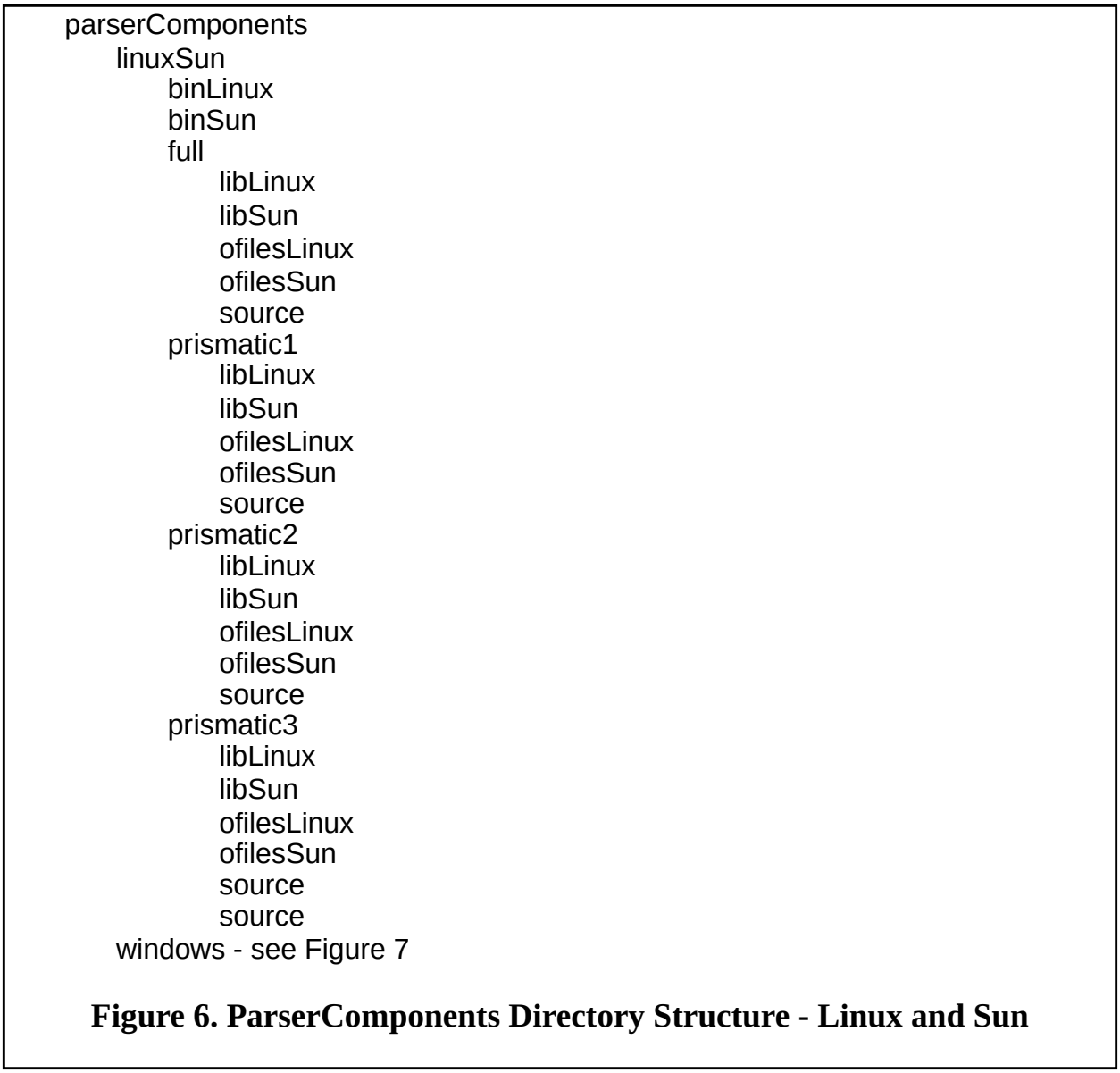


Figure 6. ParserComponents Directory Structure - Linux and Sun

parserComponents
- linuxSun - see Figure 6
- windows
  - bin
  - full
    - dmisFullClasses
      - Debug
      - dmisFull
    - dmisFullParser
      - Debug
      - dmisFullParser
    - source
  - prismatic1
    - dmisPrismatic1Classes
      - Debug
      - dmisPrismatic1
    - dmisPrismatic1Parser
      - Debug
      - dmisPrismatic1Parser
    - source
  - prismatic2
    - dmisPrismatic2Classes
      - Debug
      - dmisPrismatic2
    - dmisPrismatic2Parser
      - Debug
      - dmisPrismatic2Parser
    - source
  - prismatic3
    - dmisPrismatic3Classes
      - Debug
      - dmisPrismatic3
    - dmisPrismatic3Parser
      - Debug
      - dmisPrismatic3Parser
    - source

Figure 7. ParserComponents Directory Structure - Windows

5.3 What types of files are included in the parserComponents directory?

The parserComponents directory includes:

- executable debnf2pars utilities for generating C++, YACC, and Lex from DEBNF,
- object files that were built from C++ files,
- C++ files that were built
  - from a YACC file by bison, or
  - from a Lex file by flex, or

- from a DEBNF file by debnf2pars.

• YACC and Lex files that were built from DEBNF files by debnf2pars,
• library files (.a for Linux and Sun, .lib for Windows).
• for Linux and Sun, Makefiles for processing the other components,
• for Windows, auxiliary files generated and used by Visual C++.

We do not attempt to explain YACC, Lex, Makefiles, or C++ here. A good explanation of YACC and Lex is provided in the book “lex & yacc” written by John R. Levine, Tony Mason, and Doug Brown and published by O’Reilly. The bison and flex manuals are also quite readable. They are available for downloading free of charge from http://www.gnu.org/software/bison and http://flex.sourceforge.net, respectively. Good explanations of Makefiles and C++ are widely available.

5.4 What are the steps in building a parser from parser components?

Note first that the parsers (and their components) for the four conformance classes and three computer architectures are already completely built. It is not necessary to rebuild anything as described here. This description is provided in case you want to modify something or you are curious about how the test suite parsers were built.

The five steps in building an executable parser from components are shown in Figure 8.

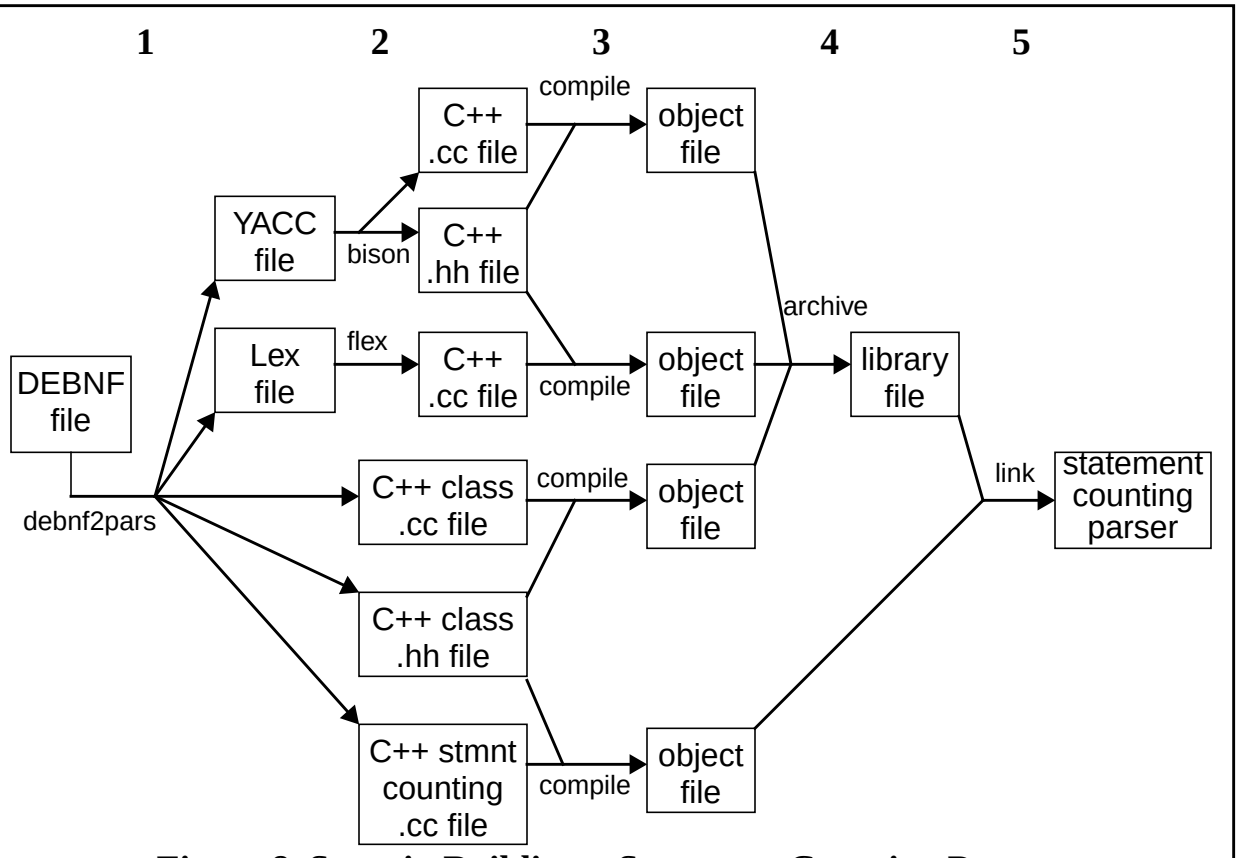


Figure 8. Steps in Building a Statement Counting Parser

As shown in Figure 8, building a parser starting from a DEBNF file includes the following steps.

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

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

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

3a. A C++ compiler reads the .cc file produced by bison, the .hh file produced by bison, and the .hh file for C++ classes (arrow not shown on Figure 8), and writes an object file for a parser.

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

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

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

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

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

In Windows, the file name suffixes are .cpp and .h, not .cc and .hh.

In all cases, the first step, running debnf2pars, requires three non-obvious things:
   • The DEBNF file name must have the suffix .debnf (e.g., dmisFull.debnf).
   • The DEBNF file must be in the directory from which debnf2pars is invoked.
   • In the invocation of debnf2pars, the .debnf suffix must be omitted. For example:

debnf2pars dmisFull

5.4.1 Rebuilding in Linux and SunOS

Everything in is already built, so you do not need to rebuild anything unless either (1) the executables do not run on your system, or (2) you want to start with your own DEBNF file.

If you want to recompile, the commands required for each step are given for Linux and SunOS in the Makefiles found in the full, prismatic1, prismatic2, and prismatic3 subdirectories of the parserComponents/linuxSun directory. Edit the Makefiles so that LINCOMPILE and LINLINK (or SUNCOMPILE and SUNLINK) are set to point to your C++ compiler. Then run make commands.

If you are starting with your own DEBNF file, the best way to begin is by creating a new subdirectory of parserComponents/linuxSun, putting your DEBNF file into the new subdirectory, copying one of the Makefiles into the new subdirectory, and modifying the

Makefile.

5.4.2 Rebuilding in Windows

Everything in is already built, so you do not need to rebuild anything unless either (1) the executables do not run on your system, or (2) you want to start with your own DEBNF file.

For Windows, all the steps above after C++ code has been generated may be accomplished using the Microsoft Visual C++ 2008 Express Edition, which may be downloaded from http://www.microsoft.com/express/vc and used with no charge. This compiler must be run using its graphical user interface.

To rebuild an already-built executable or library:

- Start Visual C++.
- From the File menu, select Open.
- In the Open Project popup window that appears, use the browser to choose the project you want. Projects have a ".sln" suffix (dmisFullParser.sln, for example). Then press the Open button. The popup will disappear.
- From the Build menu, select Rebuild Solution.
- Select Save All from the File menu, then select Exit from the File menu.

Rebuild a library before rebuilding an executable that uses the library.

Instructions for compiling the tutorials in Windows starting from source code are given in Appendix B. If all you want do is change existing source code and then recompile, the instructions above should work.

5.5 How can I build a parser for a computer architecture not included in the test suite?

A parser for a computer architecture not included in the test suite can easily be built for any of the four conformance classes covered by the test suite. The method is to start with the C++ files dmisFull.cc, dmisFull.hh, dmisFullLex.cc, dmisFullYACC.cc, dmisFullYACC.hh, and dmisFullParser.cc (or similarly named files for the other conformance classes) and compile them into an executable using a C++ compiler. The files to compile may be found in the appropriate source subdirectory in the linuxSun directory. That C++ code is suitable with no changes for use in other unix-like operating systems.

If you want to build a parser for a computer architecture not included in the test suite from a DEBNF file not included in the test suite, first rebuild the YACC and Lex Generator as described in Section 7.3. Then use the generator to produce YACC and Lex files as described in Section 5.10. Then generate C++ from the YACC and Lex files as described in Section 5.10.

5.6 What compilers do I need in order to build parsers?

To build a parser starting with existing C++ code as described in Section 5.4, all you need is 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.

To build a parser starting from a DEBNF file as described in Section 5.10, bison (a YACC compiler) and flex (a Lex compiler) must be installed. 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.

5.7 What are the C++ classes that represent DMIS like?

The system builders manual explains this.

5.8 How can I use the parser components to help build a DMIS generation system?

The system builders manual explains how to do this. The “generate” tutorial provides an example.

5.9 How can I use the parser components to help build a DMIS execution system?

The system builders manual explains how to do this. The “analyze” tutorial provides an example.

5.10 How can I build a parser from my own DEBNF file?

You can modify one of the four DEBNF files included in the test suite (to define a conformance class or represent what your system can do, for example). The best way to do this is to comment out the parts that are not to be included. After modifying the DEBNF file, use the debnf2pars executable for your computer to produce YACC and Lex files from the DEBNF file. Debnf2pars may be run using a command of the form.

debnf2pars dmisFull

This command will produce the files dmisFullParser.cc, dmisFull.cc, dmisFull.hh, dmisFull.y, and dmisFull.lex. Substitute the name of your DEBNF file for dmisFull. See details at the end of Section 5.4. Then generate C++ files using bison and flex as described below and compile the C++.

Bison may be run using a command of the form:

bison -d -l -o dmisFullYACC.cc dmisFull.y

This command will produce dmisFullYACC.hh as well as dmisFullYACC.cc. Substitute the name of your YACC file for dmisFull.y and the name of the C++ file you want for dmisFullYACC.cc.

Flex may be run using a command of the form:

flex -L -t dmisFull.lex > dmisFullLex.cc

Substitute the name of your Lex file for dmisFull.lex and the name of the C++ file you want for dmisFullLex.cc.

It is likely that when bison is run you will get messages saying there are useless non-terminals and useless rules. It is OK to ignore these, but it is better to edit your DEBNF file further by commenting out the unused parts or by replacing a production with a simpler one, and then rerun debnf2pars and rerun bison and flex.

It is also possible to write a DEBNF file from scratch and process it as just described, but it is likely that bison will report conflicts. These warnings should not be ignored. It takes some skill to modify the DEBNF to eliminate them. Conflicts have already been dealt with in the DEBNF files included in the test suite, and commenting out parts of those files will not produce conflicts.

5.11 Can I build a modified parser by editing the C++ code and recompiling?

The C++ code generated by flex and bison that does lexical analysis and parsing (dmisFullLex.cc, dmisFullYACC.cc, and dmisFullYACC.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. Many auxiliary functions are included that could be hand edited, but it would be better to edit them in the YACC (dmisFull.y) or Lex (dmisFull.lex) files where they also appear and then re-run flex or bison to regenerate the C++ code. That way your changes will survive if the C++ is regenerated.

The C++ code that represents DMIS (dmisFull.cc and dmisFull.hh) is easy to read and may be edited by hand. However, this code is used in dmisFull.y, so it may be necessary to change that also when those files are edited. If you want to try hand editing that code, read about it in the system builders manual first.

6 EBNF

6.1 Who should read this section?

Read this section if you:

- are building a DMIS parser by modifying a DEBNF file,
- want to deal with a DEBNF file for some other reason,
- are curious about DEBNF.

6.2 What is EBNF?

EBNF (Extended Backus-Naur Form) is an international standard language for describing the syntax of formal languages. EBNF is ISO standard 14977. A copy of the final standard may be downloaded free of charge from ISO at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html. The standard gives a good intuitive description of the EBNF language.

6.3 Why use EBNF?

It is a good idea to use a formal syntax language such as EBNF to describe a computer-readable language such as DMIS because a formal language allows the syntax of the target language (DMIS) to be specified completely and unambiguously. It is difficult to give a complete and unambiguous description of syntax any other way. It is a good idea to use a *standard* formal syntax language because being a standard ensures the language has been carefully developed and is publicly available to all.

6.4 What is DEBNF?

DEBNF (DMIS EBNF) is a dialect of EBNF used in the DMIS standard to define the syntax of DMIS. All of the semantics (meanings) of DEBNF are consistent with those of EBNF, but DEBNF uses some conventions that provide a shorthand allowing it to provide additional meanings that could be expressed in EBNF but would require many more pages.

DEBNF uses some, but not all, of the extensions to BNF provided by EBNF.

To be consistent with the DMIS standard, DEBNF is used in the NIST DMIS test suite.

6.5 What is DEBNF Syntax?

6.5.1 Overview

A DEBNF file is a list of productions. Each production sets a production name to be equivalent to a list of definitions. Each definition is a list of expressions. An expression may be (among other things) the name of a production, a token, a single character, or an optional. A production name is also called a non-terminal symbol.

For example, the DEBNF production for the DMISMN statement is:

```
dismnStm = DMISMN , '/' , stringConst , c , versionTag , # ;
```

In this example:

- The name of the production is **dismnStm**.
- **=** is an assignment symbol equating the name on the left to the definition on the right.
- There is only one definition, and it has six expressions.
- **DMISMN** is a token.
- **'/'** is a slash.
- **c** is a the name of a production that represents a comma.
- **stringConst** and **versionTag** are the names of other productions.
- **#** is an end-of-line symbol.
- The expressions are separated by commas, and the production is ended by a semicolon.

The production means that a **dismnStm** is equivalent to the token **DMISMN** followed by a slash followed by a **stringConst** followed by a comma followed by a **versionTag** and an end-of-line.

The order in which productions are given in a DEBNF file is irrelevant except that the top-level production (**inputFile** for DMIS) must be given first. The order in which the alternate definitions of a production is given is also irrelevant. The ordering of the expressions in a definition, however, is significant. It is OK if a production has no definitions, and it is OK if there are no expressions in a definition.

6.5.2 Rules of Standard EBNF

The following are the elements of standard EBNF used in DEBNF. Standard EBNF includes additional elements that are not needed (and, hence, not used) in DEBNF.

1. An EBNF file is a list of productions in which every production name except the first one is used in defining some other production.

2. A production consists of a non-terminal symbol followed by an equal sign, followed by a (possibly empty) list of alternative definitions, followed by a semicolon.

3. A definition is a list of expressions separated by commas.

4. A vertical bar **|** is used between the definitions of a production. For example, the following means **strVar6** may be either the token **LONG** or the token **SHORT**:

```
strVar6 = LONG | SHORT ;
```

5. An expression is a symbol name (token or non-terminal), a single character, a group, a constant, or an optional.

6. Symbol names start with a letter and include only letters and digits.

7. A single character must be preceded and followed by an apostrophe, e.g. **'/'** .

8. A group is two alternatives enclosed in parentheses (other types of groups are allowed in full EBNF). The alternatives are separated by a vertical bar. For example, the spelling of the token AND is defined as follows in the DEBNF file for full DMIS.

```
AND = '.' , ('A'|'a') , ('N'|'n') , ('D'|'d') , '.' ;
```

This means the spelling may be any of **.AND.** , **.ANd.** , **.AnD.** , **.And.** , **.aND.** , **.aNd.** , **.anD.** , **.and.** .

9. A constant is any string of printable characters or space surrounded by apostrophes, for example **'**'** or **'Not defined here'**.

10. A simple optional expression in a production is set off using square brackets.
For example, **a** , **[b** , **c]** , **d** means that either **a,d** or **a,b,c,d** is allowed.
Simple optional expressions (and multiple optional expressions, which follow) may be nested.
For example, **a , [b ,[c]] , d** means **a,d** or **a,b,d** or **a,b,c,d** is allowed.

11. A multiple optional expression in a production is set off using square brackets preceded by a digit (full EBNF allows any positive integer) and an asterisk. The digit gives the upper limit on the number of repetitions. For example, **a , 2*[b , c] , d** means **a,d** or **a,b,c,d** or **a,b,c,b,c,d** is allowed.

12. White space (spaces, tabs, and newlines) may be used anywhere except inside symbol names and constants and has no meaning. Thus, a single definition may extend across several lines. Spaces (but not tabs or newlines) may be used inside a constant, where they are part of the constant.

13. Comments are indicated by being enclosed with **(*** at the beginning and ***)** at the end. Multiple comments on the same line are allowed and may occur in the middle of definitions. For example, the following is allowed: **a** , **b** , **(* comment1 *) c** , **(* comment2 *) d**. Comments that extend across several lines are also allowed, but comments may not be nested. Comments have no formal meaning. They are treated like white space.

6.5.3 Conventions of DEBNF

DEBNF uses the following conventions in addition to the rules for standard EBNF.

1. Token - A word in upper case letters is a token, e.g. **DMISMN**. In a DMIS input file, the word must appear using the same characters as in the DEBNF file, with the following exceptions. First, in the DMIS file, either lower case or upper case letters may be used (and mixed, e.g. **DmisMn**). Second, since the EBNF standard requires that a symbol name start with a letter and include only letters and digits, DMIS tokens that start with a dot, digit, or minus sign or contain an underscore have been spelled differently. The new spellings are given as productions near the beginning of the DEBNF file. If a token appears on the left side of a production, the right side gives its spelling.

2. The name of a production other than a production spelling a token must start with a lower case letter (digits are also allowed, but underscores are not allowed), except as provided in the next paragraph.

3. The name of a data type that is not meaningfully defined by a production must start with an

upper case letter and contain a least one lower case letter, e.g. **StringVarname**. The characters to be used for these data types in a DMIS input file are specified in section 5 of DMIS 5.1. In order to make the DEBNF file be legitimate EBNF, these data types are given dummy definitions in the DEBNF file.

4. A comma is represented by a production whose name is **c**. This is to avoid having a mix of literal commas and separator commas, which is very hard to read. For example, the definition of DMISMN given earlier could also be written as follows, but the eye stumbles at the literal comma.

```
dmismnStm = DMISMN , '/' , stringConst , ',' , versionTag , # ;
```

5. In the DEBNF representation of DMIS, the end of a line of a DMIS input file is indicated by the **#** character. This is the only DEBNF convention that violates the rules of EBNF. In a DMIS input file, the end of a line is indicated by a carriage return followed by a line feed, i.e. ASCII 13 followed by ASCII 10.

6. DEBNF does not require any specific syntax or naming convention for lists. Lists that are not optional could be represented in at least three ways. It is not obvious from the right side of a production that a list is being defined. Therefore, to make DEBNF files easier to comprehend, the DEBNF files in the test suite use two conventions. First, the name of a list always ends in “List”. Second, the definition of a simple list (repetitions of a single type of item) usually has the following form.

```
itemList = [itemList , c] , item ;
```

6.6 What DEBNF files are available?

The ebnf directory contains four DEBNF files: one each for full DMIS, prismatic1, prismatic2, prismatic3. No addenda are included.

Annex C of DMIS 5.1 contains an 83-page DEBNF file for all of DMIS. The dmisFull.debnf file in the ebnf directory differs from Annex C of DMIS 5.1 by having blocks defined, which makes it several pages longer. Also, the callBlock in the dmisFull.debnf file in the ebnf directory does not exist in DMIS. As described in Section 2.6.1, the preprocessor creates a callBlock from a call statement. The DMIS file printer in the dmisFull.cc file knows that only the first line of a call block should be printed, so the call block disappears (as it should) if a DMIS file is printed back out again.

6.7 How is a DEBNF file for a conformance class generated?

In theory, a DEBNF file for a conformance class is constructed by starting with the DEBNF file for all of DMIS and commenting out those items that are not included in the conformance class on the Excel spreadsheet defining conformance classes. This almost works in practice, but in some cases one DEBNF statement must be commented out and replaced by a new one. For example, **datumLabel** is defined to be either **dLabel** or **daLabel**. In the prismatic conformance class, in some places where full DMIS uses **datumLabel**, both **dLabel** and **daLabel** are allowed, but in other places only **daLabel** is allowed. Where only **daLabel** is allowed, it is necessary to comment out **datumLabel** and replace it with **daLabel**.

7 Generator

7.1 Who should read this section?

Read this section if you:

- want to build a parser for a DEBNF file not included in the test suite for a computer architecture not included in the test suite (so you need a generator that will run on that architecture),
- want to modify the generator and you are familiar with YACC and Lex,
- are curious about how the generator works.

To use the generator to build a parser, you do not need to know how the generator works, only how to use it. That is described in Section 5.

Warning: the generator is built using YACC and it writes YACC. Also, it creates a parse tree when it parses EBNF and it builds parsers that build parse trees. Keeping the two levels (generator and generated) separate in your mind is a lot like simultaneously patting your head and rubbing your stomach.

7.2 What is the generator?

The generator is software that will read a file written in DEBNF and will automatically write five files:

- a YACC file for a parser of the language described in the DEBNF file,
- a Lex file for the lexical analyzer used by the parser,
- a C++ code file for a system that runs the parser and counts DMIS statement uses.
- a C++ header file defining classes for representing DMIS,
- a C++ code file implementing the functions and methods declared in the header file.

The executable that does the work is named debnf2pars. A parser built by debnf2pars builds a parse tree in terms of the C++ classes as it parses. Further details about the C++ classes are given in the system builders manual.

Source code for the generator is in the generator directory. The structure of that directory is shown in Figure 9. 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 parserComponents/linuxSun directory.

Figure 9. Generator Directory Structure

7.3 How can I build a generator for a computer architecture not in the test suite?

To build a generator for a computer architecture not included in the test suite, use a C++ compiler for that architecture to compile and link the following files found in the generator/source directory:

debnf2parsLex.cc
debnf2parsYACC.cc
debnf2parsYACC.hh
ebnfClasses.cc
ebnfClasses.hh

7.4 How does the generator work?

The generator:

- parses in the DEBNF file (causing a parse tree and two arrays of tokens to be built),
- adds a lot of data to the parse tree that is needed in subsequent steps,
- prints two files for the C++ classes needed to represent DMIS,
- modifies the parse tree by finding productions which, if transcribed directly into YACC, would cause a conflict in bison and replacing them with productions which look strange, recognize the same grammar, and will not cause conflicts in bison.
- prints a YACC file for a parser,
- prints a Lex file for the parser, and
- prints a C++ file for the system that uses the parser and counts DMIS statements.

The DEBNF parser in the generator was built using YACC and Lex. DEBNF is fairly small language. Only about three of the 200 or so pages of the YACC file for the generator, debnf2pars.y, are actually YACC for parsing DEBNF and parse tree building. The rest of debnf2pars.y is C++ code for adding data to the parse tree, modifying the parse tree, and writing C++, YACC, and Lex files.

The structure of a parse tree for the part of EBNF covered by DEBNF is defined in the ebnfClasses.hh file. Doubly linked lists are used in the parse tree. Manipulation functions for the lists are defined in ebnfClasses.cc. BNF is a subset of EBNF, so the EBNF parse tree structure also works for BNF. The semantics of DEBNF are identical to those of a subset of EBNF, so an EBNF parse tree works for DEBNF.

The extensions to BNF which are used in DEBNF can all be replaced by more verbose but equivalent BNF statements. These extensions are various forms of optional production. The generator replaces all the optional productions in the parse tree with their BNF equivalents, so that the parse tree becomes a BNF parse tree.

YACC productions are equivalent to BNF productions, 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. Since the actions build a parse tree in terms of C++ classes generated from the unmodified EBNF productions while the rules are for the modified productions, generating the YACC file is very complex.

The generator is not purely data driven. Several things specific to DMIS are hard-coded (or semi-hard-coded) as strings in the generator which are inserted as code in the YACC and Lex files that are generated. The Lex file, in particular, contains quite a few arcane constructions for dealing with the peculiarities of DMIS.

In addition to generating a lot of YACC code, the generator writes a lot of C++ code for auxiliary functions in the YACC file it writes.

The source code for the generator is heavily commented. The comments provide many more details about how the generator works. Start with the in-line documentation of the main function.

Appendix A Using the Test Suite in Conformance Testing

DMIS conformance testing is desirable so that:

- DMIS users can be assured that DMIS generation systems and DMIS execution systems conform to the DMIS standard.
- Vendors of DMIS generation systems and DMIS execution systems can have formal recognition that their systems conform to the DMIS standard.

A full discussion of DMIS conformance testing would be longer than this manual. This appendix presents only how the current test suite may be used to do some types of conformance testing.

A.1 Testing DMIS Generation Systems

If a DMIS generation system is claimed to implement one of the four conformance classes covered by the test suite, the test suite may be used to determine (to a limited extent) whether the system writes files that conform to that class.

To do this, the conformance tester should obtain a set of DMIS input files generated by the system and run the files (as described in Section 2.4.2) through the parser for that conformance class included in the test suite. Parser output should be sent to a file. The output file from the parser may be examined to determine:

- if there are errors in any of the DMIS input files,
- how many times each DMIS statement in the conformance class was used.
- what percentage of the DMIS statements in the conformance class were used,
- the names of the DMIS statements that were not used.

The test described above is not ideal because:

- It does not determine which alternatives for a given DMIS statement are implemented. That is, it is only a coarse-grained coverage test, not a fine-grained test.
- It does not determine whether any of the input files do what the person who used the system to generate the files wanted done.
- If the files are obtained from the system vendor, the conformance tester will have to take it on faith that the files were generated by the system and may be deceived.
- If the files are not obtained from the system vendor, the conformance tester will have to get access to the system, learn how to use the system, and use it to generate files. This is enormously time consuming.
- It is not clear that requiring all DMIS statements in the conformance class to be used is a sensible requirement. For example, if GOTARG is never used, that does not imply that any functionality is missing from the system, since the same functionality may be obtained by a series of GOTOs. The important point is that the generation system should be able to generate input files that do everything that can be done by a system that implements all the statements in the conformance class, and the test does not determine whether the system does that.

A.2 Testing DMIS Execution Systems

If a DMIS execution system is claimed to implement one of the four conformance classes covered by the test suite, the test suite may be used to determine (to a limited extent) whether the system executes files that conform to that class.

To do this, the conformance tester should run the no-motion system test files in the test suite for that conformance class through the execution system. All of them should execute without error or warning. The conformance tester might also run the system test files that do produce motion. They also should run without error or warning. To run the programs that inspect parts, the conformance tester will have to have the parts.

The test described above is not ideal because:

- The systems test files in the test suite do not fully cover the requirements of the conformance class.
- The conformance tester will have learn how to set up and operate the system or get someone else to do it.
- The test suite does not include DMIS output files corresponding to the input files, so it will be difficult to judge whether the output files produced by the system are correct.

A.3 Checking Characterization Files

Section 5.5 of the DMIS 5.1 standard describes characterization files. A characterization file may be used by a vendor of a DMIS system to specify which statements in the DMIS vocabulary are supported by the system. The first part of the input section of a characterization file is to be written in DEBNF. If that part of the input section (everything between “CHFIL1” and “ENDCH1”) is placed in a separate file (call it F), a conformance tester may check F by running it through the parser generator in the test suite.

If F is processed without error by debnf2pars (the YACC-Lex generator portion of the parser generator), that means that the DEBNF syntax used in F is correct. If the YACC file generated from F is processed without error or warning by bison, that means that F does not have any references to undefined non-terminals or any unreferenced non-terminals. If F was prepared according to the instructions in the DMIS 5.1 standard by commenting out items that are not implemented in the system, bison should emit no warnings about reduce/reduce or shift/reduce conflicts in the YACC file. If debnf2pars, bison, and flex do not report any errors, then the parser builder should be able to build a parser for DMIS input files that conform to the characterization file. That parser may be used as described in Appendix A.1 or Appendix A.2 to test files that are generated by the system (if it is a generator) or executed by the system (if it is an executor). This will show whether a DMIS generation system writes what is claimed in F (with the limitations described in Appendix A.1) or whether a DMIS execution system reads what is claimed in F (with the limitations described in Appendix A.2).

If the C++ code produced in the parser generator by bison and flex will not compile, but no error or warning was reported by debnf2pars, bison, or flex, that probably indicates a bug in debnf2pars, not an error in F. There are no known bugs in debnf2pars, however.

Appendix B Compiling Source Code in Windows

This appendix gives instructions for compiling source code for Windows using the Microsoft Visual C++ 2008 Express Edition. If you are using some other version of Visual C++, these exact instructions are not likely to work, but they may be helpful hints.

B.1 dmisFull.lib

These are instructions for making dmisFull.lib. This already exists. Unless it does not run on your machine, it does not need to be remade.

The easiest way to remake dmisFull.lib is to follow the instructions in Section 5.4.2 using the project file "dmisFullClasses.sln" in the parserComponents\windows\full\dmisFullClasses directory. If that does not work, then follow the instructions given here.

Instructions almost identical to these can be used in the prismatic directories. Just substitute "Prismatic1", "Prismatic2", or "Prismatic3" for "Full".

These instructions assume that a subdirectory of the parserComponents\windows\full directory named dmisFullClasses does not yet exist. If you want to try these instructions, first delete or rename the dmisFullClasses subdirectory.

1. Start Visual C++. If it is already running, shut it down and restart it.
2. From the File menu, select New and then Project. This brings up a popup with two large boxes on top, and three long thin boxes on the bottom, with a check box after the last one.
3. In the top left (Project types) box, select Win32.
4. In the top right (Templates) box, select Win32 Console Application.
5. In the bottom boxes put:

   Name - dmisFull

   Location - <NDTS>\parserComponents\windows\full
   where <NDTS> is the full path to the test suite, for example:
   R:\proj\dmis\kramer\NistDmisTestSuite2.1.5

   Solution Name - dmisFullClasses

   Create directory for solution - leave checked

   Then press OK.
6. In the popup that appears, press Next (not Finish).
7. This brings up a popup labeled Application Settings.

   Under Application Type, select Static library

   Under Additional Options, first uncheck Precompiled header, then press Finish. This puts control back into the main Visual C++ window.
8. To get the project to use the source code, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window. It may be necessary to select Add Existing Item multiple times, since only one or two items at a time can be added.

From the <NDTS>\parserComponents\windows\full\source directory, select the following source code files, and then press Add:

dmisFullLex.cpp
dmisFullYACC.cpp
dmisFull.cpp
dmisFullYACC.h
dmisFull.h

Visual C++ will appear to put the files in a location shown in the Solution Explorer hierarchy window on the left of the main window. This is a project hierarchy, not a directory hierarchy (although it looks like a directory hierarchy). If the source code is put in the wrong place, it can be dragged up or down the hierarchy into the right place. Header files go in the fake HeaderFiles directory, and .cpp files go in the fake SourceFiles directory.

9. To make a library, select Build Solution from the Build menu.

   The library will appear in the file

   <NDTS>\parserComponents\windows\full\dmisFullClasses\Debug\dmisFull.lib

   A second Debug directory will also appear in the following location, but the library is not in it (it has the object code).

   <NDTS>\parserComponents\windows\full\dmisFullClasses\dmisFull\Debug.

   Visual C++ does not like the sprintf function and issues a lot of warnings about it. These can be ignored.

10. Select Save All from the File menu, then select Exit from the File menu.

B.2 dmisFullParser

These are instructions for making the dmisFullParser executable. This already exists. Unless it does not run on your machine, it does not need to be remade.

The easiest way to remake dmisFullParser is to follow the instructions in Section 5.4.2 using the project file “dmisFullParser.sln” in the parserComponents\windows\full\dmisFullParser directory. If that does not work, then follow the instructions given here.

Instructions almost identical to these can be used in the prismatic directories. Just substitute “Prismatic1”, “Prismatic2”, or “Prismatic3” for “Full”.

These instructions assume that a subdirectory of the parserComponents\windows\full directory named dmisFullParser does not yet exist. If you want to try these instructions, first delete or rename the dmisFullParser subdirectory.

1. Start Visual C++. If it is already running, shut it down and restart it.
2. From the File menu, select New and then Project. This brings up a popup with two large boxes on top, and three long thin boxes on the bottom, with a check box after the last one.
3. In the top left (Project types) box, select Win32.
4. In the top right (Templates) box, select Win32 Console Application.

5. In the bottom boxes:

   Name- dmisFullParser

   Location - <NDTS>\parserComponents\windows\full
   where <NDTS> is the full path to the test suite, for example:
   R:\proj\dmis\kramer\NistDmisTestSuite2.1.5

   Solution Name - dmisFullParser

   Create directory for solution - leave checked

   Then press OK.

6. In the popup that appears, press Next.

7. This brings up a popup labeled Application Settings.

   Under Application Type, select Console Application.

   Under Additional Options, first uncheck Precompiled Header, then check Empty Project.

   Then press Finish. This puts control back into the main Visual C++ window.

8. To get the project to use the source code, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window. It may be necessary to select Add Existing Item twice, since only one item at a time can be added.

   From the <NDTS>\parserComponents\windows\full\source directory, select the following source code files, and then press Add:

   dmisFullParser.cpp
   dmisFull.h

   Visual C++ will appear to put the files in a location shown in the Solution Explorer hierarchy window on the left of the main window. This is a project hierarchy, not a directory hierarchy (although it looks like a directory hierarchy). If the source code is put in the wrong place, it can be dragged up or down the hierarchy into the right place. Header files go in the fake HeaderFiles directory, and .cpp files go in the fake SourceFiles directory.

9. To get the project to use the dmisFull library, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window.

   From the <NDTS>\parserComponents\windows\full\dmisFullClasses\Debug directory, select dmisFull.lib.

   When you add dmisFull.lib, Visual C++ will display a popup window asking if you want to create a rule for making dmisFull.lib.

   Press the No button.

   In the hierarchy window on the left of the main window, dmisFull.lib goes directly into dmisFullParser, not in any fake directory.

   10. To make the executable dmisFullParser, select Build Solution from the Build menu. The executable will appear in the file

   <NDTS>\parserComponents\windows\full\dmisFullParser\Debug\dmisFullParser.exe

You may want to copy it to:

<NDTS>\parsers\windows\full

11. Select Save All from the File menu, then select Exit from the File menu.

B.3 debnf2pars

These are instructions for making the debnf2pars executable. This already exists. Unless it does not run on your machine, it does not need to be remade.

The easiest way to remake debnf2pars is to follow the instructions in Section 5.4.2 using the project file "debnf2pars.sln" in the generator\windows\debnf2pars directory. If that does not work, then follow the instructions given here.

These instructions assume that a subdirectory of the generator\windows directory named debnf2pars does not yet exist. If you want to try these instructions, first delete or rename the debnf2pars subdirectory.

1. Start Visual C++. If it is already running, shut it down and restart it.
2. From the File menu, select New and then Project. This brings up a popup with two large boxes on top, and three long thin boxes on the bottom, with a check box after the last one.
3. In the top left (Project types) box, select Win32.
4. In the top right (Templates) box, select Win32 Console Application.
5. In the bottom boxes:

   Name - debnf2pars

   Location - <NDTS>\generator\windows
   where <NDTS> is the full path to the test suite, for example:
   R:\proj\dmis\kramer\NistDmisTestSuite2.1.5

   Solution Name - debnf2pars

   Create directory for solution - leave checked

   Then press OK.
6. In the popup that appears, press Next.
7. This brings up a popup labeled Application Settings.

   Under Application Type, select Console Application.

   Under Additional Options, first uncheck Precompiled Header, then check Empty Project. Then press Finish.

   This puts control back into the main Visual C++ window.
8. To get the project to use the source code, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window. It may be necessary to select Add Existing Item multiple times, since only one or two items at a time can be added.

   From the <NDTS>\generator\windows\source directory, select the following source code files, and then press Add:

debnf2parsLex.cpp
debnf2parsYACC.cpp
debnf2parsYACC.h
ebnfClasses.cpp
ebnfClasses.h

Visual C++ will appear to put the files in a location shown in the Solution Explorer window on the left of the main window. This is a project hierarchy, not a directory hierarchy (although it looks like a directory hierarchy). If the source code is put in the wrong place, it can be dragged up or down the hierarchy into the right place. Header files go in the fake HeaderFiles directory, and .cpp files go in the fake SourceFiles directory.

9. To make the executable, select Build Solution from the Build menu.

   The executable will be in

   <NDTS>\generator\windows\debnf2pars\Debug\debnf2pars.exe

B.4 reformatDmis

These are instructions for making the reformatDmis executable. This already exists. Unless it does not run on your machine, it does not need to be remade.

The easiest way to remake reformatDmis is to follow the instructions in Section 5.4.2 using the project file “reformatDmis.sln” in the parsers\windows\reform\reformatDmis directory. Then copy the executable into parsers\windows. If that does not work, then follow the instructions given here.

These instructions assume that a subdirectory of the parsers\windows directory named reform does not yet exist. If you want to try these instructions, first delete or rename the reform subdirectory.

1. Start Visual C++. If it is already running, shut it down and restart it.
2. From the File menu, select New and then Project. This brings up a popup with two large boxes on top, and three long thin boxes on the bottom, with a check box after the last one.
3. In the top left (Project types) box, select Win32.
4. In the top right (Templates) box, select Win32 Console Application.
5. In the bottom boxes:

   Name - reformatDmis

   Location - <NDTS>\parsers\windows\reform
   where <NDTS> is the full path to the test suite, for example:
   R:\proj\dmis\kramer\NistDmisTestSuite2.1.5

   Solution Name - reformatDmis

   Create directory for solution - leave checked

   Then press OK.

6. In the popup that appears, press Next.

7. This brings up a popup labeled Application Settings.

   Under Application Type, select Console Application.

   Under Additional Options, first uncheck Precompiled Header, then check Empty Project. Then press Finish.

   This puts control back into the main Visual C++ window.

8. To get the project to use the source code, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window.

   From the <NDTS>\parsers\windows\reform directory, select the following source code file, and then press Add:

   reformatDmis.cpp

   Visual C++ will appear to put the file in a location shown in the Solution Explorer window on the left of the main window. This is a project hierarchy, not a directory hierarchy (although it looks like a directory hierarchy). If the source code is put in the wrong place, it can be dragged up or down the hierarchy into the right place. Header files go in the fake HeaderFiles directory, and .cpp files go in the fake SourceFiles directory.

9. To make the executable, select Build Solution from the Build menu.

   The executable will be in
   <NDTS>\parsers\windows\reform\reformatDmis\Debug\reformatDmis.exe

10. Copy reformatDmis.exe into <NDTS>\parsers\windows