Multi-Dimensional & Hierarchical Toolkit User's Guide

Kevin C. O'Kane
kc.okane@gmail.com
http://www.cs.uni.edu/~okane
http://threadsafebooks.com/
Sept 29, 2015

Table of Contents

Index of Figures

1 Introduction

The Multi-Dimensional and Hierarchical Database Toolkit (MDH) is a Linux-based, open sourced, toolkit of portable libraries that support access to the Mumps multi-dimensional and hierarchical database and other services. The package is written in C and C++ and licensed under the GNU GPL/LGPL licenses.

The toolkit permits manipulation of very large, character string indexed, multi-dimensional, sparse matrices from C++ programs. The toolkit supports access to PostgreSQL and MySQL relational data base servers, the Perl Compatible Regular Expression Library, and the Glade GUI builder.

The toolkit makes Mumps data base and functions available as C++ classes and permits execution of Mumps scripts directly from C++ programs. The toolkit is provided with the Mumps distribution and is available if Mumps is installed. No further installation beyond the basic Mumps installation described above is required.

The class, function and macro libraries primarily operate on global arrays. Global arrays are undimensioned, string indexed, disk resident data structures whose size is limited only by available disk space. They can be viewed either as multi-dimensional sparse matrices or as tree structured hierarchies.

1.1 MDH Class Library Header File

To use the class libraries, add the following to the beginning of your C++ program:

```
#include <mumpsc/libmpscpp.h>
```

This statement inserts in the necessary header files for you C++ program. In addition to the MDH class libraries, the following standard systems headers will be included as well:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <string.h>
#include <math.h>
#include <stdlib.h>
```

2 MDH Data Types

The MDH is built upon two data classes. One is for global arrays (**global**)and the other is a string data type (**mstring**) which mimics that of Mumps strings.

2.1 Mstring Data Objects

The **mstring** class provides functionality similar to the basic typeless string data type in Mumps. Objects of **mstring** may contain text, integers and floating point values. Operations on **mstring** objects include addition, multiplication, subtraction, division, modulo, concatenation and so forth. Objects of type **mstring** are declared in the normal manner such as:

```
mstring mvar1,var2,var3;
```

They may be initialized with **int**, **long**, **float**, **double**, **char** * and **string** and **mstring** values such as:

```
mstring var1(10),var2(10.123),var3("test"),var4(stringVar);
```

Objects of type **mstring** may be assigned to most data types and most data types may be assigned to objects of type **mstring**.

Objects of type **mstring**, **string**, and null terminated character strings are the only legal indices for objects of class **global**.

2.1.1 Arithmetic Operations on Mstring Objects

When **mstring** objects contain numeric values, you may apply arithmetic operators directly to the **mstring** object or objects.

Both extended precision and basic hardware precision are available.

In hardware precision mode, floating point numbers are processed by the machine's arithmetic processing hardware. Floating point numbers are treated as 64-bit *double* values and integers are treated as signed 64-bit *long* integer values. Thus, integers may range from:

$-9,223,372,036,854,775,808\ (-2^{63}+1)$ to $9,223,372,036,854,775,807\ (2^{63}-1)$

Hardware floating point numbers utilize a one bit sign, an 11 bit exponent and a 52 bit fraction. This translates into approximately 16 decimal digits of precision in the range of $\pm \sim 10^{-323.3}$ to $\pm \sim 10^{308.3}$.

Extended precision is available through use of the GNU multiple precision arithmetic library[1] and the GNU MPFR library[2]. For integers, this means effectively unlimited precision. For floating point, the exponent is 64 bits and the fraction is user specified (default of value of 72 bits).

Hardware arithmetic will be selected during system build if (1) *configure* does not find the extended precision libraries or (2) the user specifies the configuration option:

--with-hardware-math.

If the extended precision libraries are found and the above option has *not* been specified, extended precision will be in effect.

If extended precision is used, the number of bits in the fraction of a floating point number can be set with:

--with-float-bits=value

where *value* is the number of bits. The default value is 72.

For extended precision floating point numbers, the number of digits of precision that may be printed is controlled by:

--with-float-digits=value

where *value* is the number of digits. The default is 20.

When printing an extended precision floating point number, the number of digits being printed should be consistent with the number of bits in the fraction. If the number of digits is too large, insignificant, random low-order digits may appear in the output.

2.2 Global Data Objects

Objects of class **global** provide access to the global array database. The class includes functions to create, delete (kill), and navigate global arrays.

In your C++ program, you must declare each global array that the program will use. Normally, these declarations will appear at the beginning of the program. A global declaration has the form:

```
global program_ref(database_name);
```

Where *program_ref* is the name by which the global array will be referred to in your program and *database_name* is the name of the actual global array in the file system. Both may be the same. The value for *database_name* may be expressed as a pointer to a character string constant.

1 http://www.mpfr.org/
2 http://gmplib.org/manual/index.html

For example, if your program uses a Mumps global array stored in the file system with the name *patient*, you might have the following C++ declaration in your program:

```
global patient("patient");
```

Once declared, a global array object may be used to access the contents of the global array database. For example, for the global array object *patient* declared above, the following reference might be made:

```
patient(ptid,test,date,time)=result;
```

where *ptid, test, data, result* and *time* are **mstring** or **char** * null terminated variables or constants.

Although objects of class **mstring** may be C++ arrays, objects of class **global** may not.

Objects of class **global** may *not* be initialized in declaration statements**.**

3 Operators Defined on Mstring & Global Objects

Objects of class **mstring** may appear as the operands of most C++ builtin operators by means of C++ operator overloading.

In the cases of binary operators, the other operand may be most other builtin data types as well as **global** and **mstring** objects.

Figure 1 contains the full list of C++ operators that have been overloaded for use with objects of types **mstring** and **global**. In these examples, assume the declarations:

```
mstring ms, msa[10];
global gb("test");
```

Unary Operators	Description	Examples
++ --	Suffix/postfix increment and decrement	ms++; gb("123")++;
[]	Array subscripting[3]	mstring msa[10]; msa[1] = "abc";
++ --	Prefix increment and decrement	++ms; ++gb("123");
+ -	Unary plus and minus	cout << +gb("123") << endl; cout << -ms << endl;
(*type*)	C-style explicit cast	ms = "123" int k = (int) ms("123");
*	Indirection (dereference)	global *p1 = &gb; (*p1)("111") = 10; mstring *p2 = msa; (*p2)[3] = "abc";
& (unary)	Address-of	mtstring *p1 = &ms;
new, new[]	Dynamic memory allocation	global *p3 = new global("xxx"); (*p3)("xxx") =2 2; mstring *p4 = new mstring; *p4=123;
delete, delete[]	Dynamic memory deallocation	delete p1;
Binary Operators[4]	Description	Examples
* / %	Multiplication, division, and remainder	ms = ms * 2; ms = gb("123") / ms;

3 Only with an **mstring** operand.

4 One operand, the first, may be of type **mstring** or **global** and the other may be of type **mstring**, **global**, **float**, **double**, **int**, **long**, **char***, or **string**.

		ms = gb("123") % 5;
+ -	Addition and subtraction	ms = ms + 2; ms = gb("123") - ms;
<< >>	stream insertion / extraction	cout << ms; cin >> gb("123");
< <=	For relational operators < and ≤ respectively[5]	if (ms <= gb("123")) ... if (ms < gb"abc")) ... if ("abc" < gb("123") ...
> >=	For relational operators > and ≥ respectively[5]	if (ms >= gb("123") ... if (ms > gb("abc")) ... if ("abc" > gb("123")) ...
== !=	For relational operators = and ≠ respectively[5]	if (ms == gb("123") ... if (ms != gb("abc")) ...
&&	Logical AND	if (ms && gb("123")) ...
||	Logical OR	if (ms || gb("123")) ...
Ternary Operator	Description	Examples
?:	Ternary conditional	ms ? ms : y
Assignment[6]	Description	Examples
=	Direct assignment	ms = 123 gb("123") = 1.3456 ms = "test"
+= -=	Compound assignment by sum and difference	ms=0; ms += 123 ms+="123"; gb("123")=0; gb("123") -= 10
*= /= %=	Compound assignment by product, quotient, and remainder	ms=0; ms *= 123 gb("123")=10; gb("123") /= 10 gb("123")=10; gb("123") %= 10
& (binary)	Concatenate. First operand must be of type **global** or **mstring**[7]. The second operand may be **string**, **mstring**, **global**, **char* int**, **long**, or **double.**	mstring i="aaa",j="bbb",k="ccc"; i=i&j&k; // i -> aaabbbccc

Figure 1 Operators Defined on **mstring** and **global**

3.1 Example Arithmetic Operations on global & mstring Objects

The operations of add, subtract, multiply, divide, pre/post increment and pre/post decrement are defined (overloaded) for **global** and **mstring** variables either together (in binary or the ternary operator) or in connection with other builtin data types. The contents of the **global** array node or **mstring** variable must be compatible with the dominant data type of the operation. If the contents not compatible with the operation (example, incrementing a string of text), the value of the **global** will be interpreted as zero. Examples:

Code Examples	Results
global gbl("gbl"); int i, j=10; string a = "10", b = "20", c = "30"; char aa[] = "10", bb[] = "20", cc[] = "30"; mstring aaa = "10", bbb = "20", ccc = "30";	

5 If one operand is a numeric type (**long**, **float** etc.), the **mstring** or **global** will be interpreted as a numeric value rather than as a string. If both operands are of type **global** or **mstring**, they will be compared as strings. If one operand is of type **global** or **mstring** and the other is of type **char*** or **string**, they will be compared as strings.

6 The left-hand-side must be of type **mstring** or **global** while the right-hand-side may be of types **mstring**, **global**, **float**, **double**, **int**, **long**, **char***, or **string**. When arithmetic assignment operators are used, right-hand-side **string**, **char***, and **global** operands will be converted to numeric following the default Mumps conversion rules.

7 Note: because the overloaded bitwise *and* operator (&) is of lower precedence than the bit shift operator <<, in output operations (such as when using *cout*), an expression involving the bitwise & operator must to be in parentheses.

gbl.Kill(); gbl(a,b,c) = 10; gbl(aa,bb,cc) = 20; gbl(aaa,bbb,ccc) = 30; i = gbl(a,b,c) + 20; cout << i << endl;	 50
i = 20 + gbl(a,b,c); cout << i << endl;	 50
i = gbl(a,b,c) / j; cout << i << endl;	 3
i = gbl(a,b,c) * 2; cout << i << endl;	 60
gbl(a,b,c) ++; cout << gbl(a,b,c) << endl;	 31
gbl(a,b,c) --; cout << gbl(a,b,c) << endl;	 30
i = ++ gbl(a,b,c); cout << i << " " << gbl(a,b,c) << endl;	 31 31
i = gbl(a,b,c) ++; cout << i << " " << gbl(a,b,c) << endl;	 31 32
gbl(a,b,c) += 10; cout << gbl(a,b,c) << endl;	 42
gbl(a,b,c) -= 10; cout << gbl(a,b,c) << endl;	32
gbl(a,b,c) *= 2; cout << gbl(a,b,c) << endl;	 64
gbl(a,b,c) /= 2; cout << gbl(a,b,c) << endl;	 32
aaa="aaa"; bbb="bbb"; ccc="ccc"; cout << (aaa&bbb&ccc) << endl;	 aaabbbccc
Figure 2 Code Examples	

4 Functions for Global and Mstring Objects

As is the case with Mumps functions, characters in strings are counted beginning with one, not zero. Thus, the substring beginning at position 3 through and including position 5 in the string "*abcdef*" is "*cde*".

If an object of type **mstring** contains a string that is to be used as a global array reference in connection with one of the functions below, the global array reference must be preceded by a circumflex character (^) as is the case in Mumps and, also, the indices must be constants. Example:

```
mstring x="^g(1)";
cout x.Qlength() << endl; // prints 1
```

Function Parameters

INT An expression involving **int, long, float, double, mstring** or **global** the result of which can be interpreted as an integer. Data of type **char*** may not be used.

STR An expression involving **int, long, float, double, mstring** or **global** the result of which can be interpreted as a string. Data of type **char*** may be used but not as part of an expression.

Function	Description
`int mstring::Ascii( [INT] )` `int global::Ascii( [INT] )`	Returns the decimal value of the first ASCII character in the invoking **global** or **mstring**. If an integer argument is given, it returns the decimal value of the character at the offset designated by the argument. **mstring** and **global** arguments will be interpreted as integers. `mstring s1="abcdef";` `s1.Ascii()  -> 97` `s1.Ascii(2) -> 98`
`void mstring::Assign(global)` `void mstring::Assign(mstring)` `void mstring::Assign(string)` `void mstring::Assign(char*)` `void mstring::Assign(int)` `void mstring::Assign(long)` `void mstring::Assign(double)`	Assign a value to the global array reference containg in the invoking **mstring**. Contents of invoking **mstring** must conform to Mumps global array naming conventions and all indices must be constants, global array references, or variables previously defined in the Mumps Interpreter symbol table (see: *SymPut()*). Items placed in the Mumps Interpreter symbol table are discarded when the program ends. This function throws a *MumpsGlobalException* in the event of error. `mstring x="^g(1,1)";` `global g("g");` `x.Assign("test test");` `cout << g(1,1) << endl; // -> test test` `SymPut("a","1"); // a put in symTab` `x="^g(a,a)";  // reference uses a` `x.Assign("abc");` `cout << g(1,1) << endl; // -> abc` `g(1)=1;` `x="^g(^g(1),^g(1))";` `x.Assign("xyz");` `cout << g(1,1) << endl; // -> xyz`
`double global::Avg()`	Returns the average of the values of data bearing nodes beneath the given global array reference. `global a("a");` `for (i=0; i<1000; i++)` `  for (j=1; j<10; j++)` `  a(i,j) = j;` `a("100").Avg() -> avg below node a("100")` `a().Avg()  -> average of all nodes`
`void global::Centroid(global B)`	A centroid vector B is calculated from the invoking two dimensional **global** array matrix. An element of the centroid vector is the average of the values of each for the corresponding column of the matrix. Any previous contents of the **global** array named to receive the centroid vector are lost. The invoking **global** array must contain at least two dimensions. `global A("A");`

	global B("B"); mstring i,j; for (i=0; i<10; i++) for (j=1; j<10; j++) A(i,j) = 5; A().Centroid(B()); mstring a=""; while (1) { a=B(a).Order(); if (a=="") break; cout << a << " --> " << B(a) << endl; } Yields: 1 --> 5 2 --> 5 3 --> 5 4 --> 5 5 --> 5 6 --> 5 7 --> 5 8 --> 5 9 --> 5
mstring mstring::Concat(char *) mstring mstring::Concat(global) mstring mstring::Concat(mstring) mstring mstring::Concat(string) mstring mstring::Concat(int) mstring mstring:Concat(long) mstring mstring::Concat(double) mstring global::Concat(string) mstring global::Concat(global) mstring global::Concat(char *) mstring global::Concat(mstring) mstring global::Concat(int) mstring global::Concat(long) mstring mstring::Concat(double)	Returns **mstring** consisting of the value from the invoking object concatenated with the value of the parameter mstring a="aaa",b="bbb",c; c=a.Concat(b); // c contains aaabbb
long global::Count()	Returns the number of data bearing nodes beneath the given global array reference. global a("a"); mstring i,j; for (i=1; i<11; i++) for (j=1; j<11; j++) a(i,j) = 5; a().Count() -> 100 a("5").Count() -> 10
void global::DocCorrelate(global B, mstring fcn, double threshold) void global::DocCorrelate(global B, char * fcn, double threshold)	DocCorrelate() builds a square *document-document* correlation matrix from the invoking **global** array *document-term matrix*. The name of the function to be used in calculating the *document-document* similarity is given by *fcn* and may be *Cosine*, *Jaccard*, *Dice*, or *Sim1*. The minimum correlation threshold is given in *threshold* which defaults to 0.80 if omitted. global A("A");

	global B("B"); long i,j; A("1","computer")=5; A("1","data")=2; A("1","program")=6; A("1","disk")=3; A("1","laptop")=7; A("1","monitor")=1; A("2","computer")=5; A("2","printer")=2; A("2","program")=6; A("2","memory")=3; A("2","laptop")=7; A("2","language")=1; A("3","computer")=5; A("3","printer")=2; A("3","disk")=6; A("3","memory")=3; A("3","laptop")=7; A("3","USB")=1; A().DocCorrelate(B(),"Cosine",.5); B.TreePrint(); Yields 1 2=0.887096774193548 3=0.741935483870968 2 1=0.887096774193548 3=0.701612903225806 3 1=0.741935483870968 2=0.701612903225806
mstring global::Extract([INT [,INT]]) mstring mstring::Extract([INT [,INT]])	Returns the substring of the invoking **global** or **mstring** beginning at the position designated by the 1st argument and ending at the position designated by the second argument, inclusive. If no second argument is given, the single character designated by the first argument is returned. If the second argument specifies a position beyond the end of the string, the remainder of the string including and following the character designated by the first argument is returned. global g1("g1"); g1("1")="abcdef"; g1("1").Extact(2) -> b g1("1").Extact(2,4) -> bcd g1("1").Extract(2,99) -> bcdef
mstring mstring::Eval()	Evaluates the Mumps expression in the invoking **mstring** object and returns the result in an **mstring**. If an error occurs, an *InterpreterException* is thrown. The invoking **mstring** object may contain a valid mumps

	expression. `mstring x="5*2";` `x.Eval()  -> 10` `global g("g");` `g("1","1")=22;` `x="^a(1,1)";` `x.Eval()  -> 22`
`int global::Find(STR [,INT] )` `int mstring::Find(STR [,INT] )`	Searches the invoking string for the firts instance of the **STR** argument and, if **STR** is found, returns the character position of the character immediately following the instance of **STR**. If an **INT** argument is provided, the search begins at that character offset in the invoking string. Returns -1 is **STR** is not found. `mstring p="abcdefabcdef";` `p.Find("def")   -> 7` `p.Find("def",5) -> 13`
`mstring Horolog()`	Returns an **mstring** of the form "x,y" where *x* is the number of days since December 31, 1840 and *y* is the number of seconds since midnight.
`void global::IDF(double DocCount)`	The IDF() function calculates for the invoking **global** array vector the i*nverse document frequency* weight of each term. The vector indices should be words and have as stored values the number of documents in which each word occurs. The document count for each element will be replaced by the calculated IDF value. The IDF is calculated as: l*og2(DocCount/*W_n*)+1* where W_n is the number of documents in which a term appears (the document frequency). The value *DocCount* is the total number of documents present in the collection. `global a("a");` `a("now")=2;` `a("is")=5;` `a("the")=6;` `a("time")=3;` `a().IDF(4);` `a().TreePrint();` `Yields:` `is=0.678072` `now=2.000000` `the=0.415037` `time=1.415037`
`mstring global::Justify(INT [,INT] )` `mstring mstring::Justify(INT [,INT] )`	Right justifies the invoking object in an **mstring** field whose length is given by the first argument. If the second argument is present and a positive integer, the invoking object is right justified in a field whose length is given by the first argument with the number decimal places as specified by the second argument. The two argument form imposes a numeric interpretation upon the first argument. Rounding occurs in the two argument case.

	mstring p=123.456 p.Justify(10) -> 123.456 p.Justify(10,2) -> 123.46 p="abcdef"; p.Justify(p,10) -> abcdef
void global::Kill()	Kill (delete) the named **global** array node and all descendants. To kill and entire **global** array use: global gb("gb"); gb().Kill;
int global::Length([STR]) int mstring::Length([STR])	Returns the length of the invoking string. If an argument **STR** is given, the number returned is the number of invoking string segments divided by the argument. mstring p="abc & def"; p.Length() -> 9 p.Length("&") -> 2
double global::Max()	Returns the maximum numeric value of the data bearing nodes beneath the given reference. Non-numeric values are treated as zeros. global a("a"); mstring i,j; for (i=1; i<11; i++) for (j=1; j<11; j++) a(i,j) = rand()%1000; a().Max() -> 996 (results will vary) a("10").Max() -> 932
double global::Min()	Returns the minimum numeric value of the data bearing nodes beneath the given reference. Non-numeric values are treated as zeros. global a("a"); mstring i,j; for (i=1; i<11; i++) for (j=1; j<11; j++) a(i,j) = rand()%1000; a().Min() -> 11 (results will vary) a("10").Min() -->12
void global::Multiply(global, global)	The invoking global array matrix is multiplied by the first argument global array matrix and the result is placed in the second argument global array matrix. The number of columns of the invoking global array matrix must equal the number of rows of the first argument global array matrix. The resulting matrix (second argument) will have *n* rows and *m* columns where *n* is the number of rows of invoking global array matrix and *m* is the number of columns of the first argument global array matrix. The contents of the second argument, if any, will be deleted before the operation begins. The data stored at each node in the invoking matrix and the first argument matrix must be numeric. All calculations are performed in double precision arithmetic. Each input matrix must be two dimensional. The output matrix is also two dimensional.

	global d("d"); global e("e"); global f("f"); d("1","1")=2; d("1","2")=3; d("2","1")=1; d("2","2")=-1; d("3","2")=0; d("3","2")=4; e("1","1")=5; e("1","2")=-2; e("1","3")=4; e("1","4")=7; e("2","1")=-6; e("2","2")=1; e("2","3")=-3; e("2","4")=0; d().Multiply(e(),f()); f().TreePrint(); Yields: 1 1=-8 2=-1 3=-1 4=14 2 1=11 2=-3 3=7 4=7 3 1=-24 2=4 3=-12 4=0
mstring **global::**Name()	Returns an **mstring** containing of the **global** reference with all variables and expressions in the indices evaluated. global a("a"); mstring b="1",c="2",d="3"; a(b,c,d,c+d).Name() -> a("1","2","3","5")
int global::Pattern(STR) int mstring::Pattern(STR)	Evaluates the invoking string according to the pattern string **STR** (see Mumps documentation) and returns 0 (does not match) or 1 (does match). mstring p=12345; p.Pattern("5N" -> 1
mstring global::Piece(STR, INT [,INT]) mstring mstring::Piece(STR, INT [,INT])	Returns a substring of the invoking object delimited by the instances of the first **STR** argument. The **STR** delimiter divides the invoking object into pieces. The substring returned in the two argument case is the i^{th} substring of the invoking object there *i* is the value of the first **INT** argument. In the three argument form, the string returned begins at the i^{th} piece and ends at the j^{th} piece where *j* is the value of the second **INT** argument. If only one argument is given, *i* is assumed to be 1. mstring p="abc.def.ghi"; p.Piece(".") -> abc

	`p.Piece(".",2)  -> def` `p.Piece(".",2,3) -> def.ghi`
`int global::Qlength(mstring ref)` `int mstring::Qlength(char * ref)`	Returns the number of subscripts in the global array reference. **mstring** global array references must include the circumflex (^) character.[8] `global g("g");` `g(1,2,3,4,5).Qlength() -> 5` `mstring x="^g(1,2,3,4,5,6)";` `x.Qlength() -> 6`
`mstring mstring::Query()` `mstring global::Query()`	Returns an object of type **mstring** containing the next global array reference in the data base following the invoking global array reference or the empty string if there are none. The invoking object is either a global array reference or an **mstring** containing a string corresponding to a global array reference. **mstring** global array references must include the circumflex (^) character.[8] `mstring i,j;` `global g("g");` `for (i=1; i<10; i++)` `  for (j=1; j<10; j++)` `  g(i,j)=i+i;` `g().Query()  -> ^g("1","1")` `g(2).Query()  -> ^g("2","1")` `g(2,2).Query() -> ^g("2","3")` `i="^g()"` `i.Query() -> ^g("1","1")` `i=i.Query();` `i.Query() -> ^g("1","2")`
`mstring mstring::Qsubscript(int)` `mstring global::Qsubscript(int)`	Returns the subscript of a global array reference designated by the argument. **mstring** global array references must include the circumflex (^) character.[8] `global g("g");` `g(9,8,7).Qsubscript(3) -> 7` `mstring x="^g(9,8,7)";` `x.Qsubscript(3) -> 7`
`bool global::ReadLine()` `bool global::ReadLine(FILE *)` `bool global::ReadLine(istream &)` `bool mstring::ReadLine()` `bool mstring::ReadLine(FILE *)` `bool mstring::ReadLine(istream &)`	Reads the next input line into the invoking objsect. If no argument is given *stdin* is used. Otherwise, the inout file is determined by the argument.
`int sw(mstring s, mstring t, [int show_aligns=0, int show_mat=0, int gap=-1, int mismatch=-1, int match=2])` `int sw(string s, string t, [int show_aligns=0, int show_mat=0, int`	Calculate the *Smith-Waterman* Alignment between strings *s* and *t*. Result returned is the highest alignment score achieved. Parameters other than the first two are optional. If only some of the optional parameters are supplied, only trailing parameters may be omitted, as per C/C++ rules.

8 See example in Figure 5 on page 43.

gap=-1, int mismatch=-1, int match=2]) int sw(char *s, char *t, [int show_aligns=0, int show_mat=0, int gap=-1, int mismatch=-1, int match=2])	If you compare very long strings (>100,000 character), you may exceed stack space. This can be increased under Linux with the command: `ulimit -s unlimited` Other options are: `ulimit -a` and `ulimit -aH` to show limits. If *show_aligns* is zero, no printout of alternative alignments is produced (default). If *show_aligns* is not zero, a summary of the alternative alignments will be printed. If *show_mat* is zero, intermediate matrices will not be printed (default). The parameters *gap*, *mismatch* and *match* are the gap and mismatch penalties (normally negative integers) and the match reward (a positive integer). If insufficient memory is available, a *segmentation* violation will be raised. **The first character of each sequence string MUST be blank.** In the printed output, a colon represents a match, a hyphen represents a stretch of the associated string and a blank indicates mismatch. `char s[]=" now is the time for all good men to come to the aid of the party";` `char t[]=" time for good men";` `int i=sw(s,t,1,0,-1,-1,3);` `cout << "Score: " << i << endl;` `Results in:` `12 time- for all good-- men 32` `:::: ::::: ::::: ::::` `1 time for -- good men 22` `score=48`
int SQL_Command(mstring) int SQL_Command(string) int SQL_Command(char *)	Passes the string argument to the SQL database server. See Mumps *sql* command for a description of the argument. The results are written to a file named *mumps.tmp* where columns are <tab> separated.
int SQL_Connect(char *) int SQL_Connect(string) int SQL_Connect(mstring)	Establishes connection with the database server (see Mumps command *sql/d* for a description of the arguments).
int SQL_Disconnect();	Disconnects from the database server.
int SQL_Format() int SQL_Format(mstring) int SQL_Format(string) int SQL_Format(char *)	Formats the Mumps **global** array database on the SQL server (see Mumps *sql/d* for a description of the arguments). If no argument is given, system defaults are used.
mstring SQL_Message()	Returns most recent SQL database server returned message or the empty string if there is none.

`bool SQL_Msql()`	Returns *true* if the **global** arrays are being stored in a MySQL database server.
`bool SQL_Native()`	Returns *true* if the **global** arrays are being stored in a native database.
`bool SQL_Open()`	Returns *true* if there is a connection to the database server, *false* otherwise.
`bool SQL_Postgres()`	Returns *true* if the **global** arrays are being stored in a PostgreSQL database server.
`mstring SQL_Table()` `mstring SQL_Table(mstring, [int] )` `mstring SQL_Table(string, [int] )` `mstring SQL_Table(char *, [int] )`	Returns an **mstring** containing name of the current **global** array table (default: *mumps*), followed by a comma, followed by the maximum number of columns permitted in the table (default is 10). If arguments are provided, they set the name of the table and the maximum number of columns in the table (maximum of 10). If the second argument is omitted, it defaults to 10.
`double global::Sum()`	The global array nodes beneath the invoking referenced global array are summed. Non-numeric quantities are treated as zero. `global a("a");` `mstring i, j;` `for (i = 1; i < 11; i++)` `  for (j = 1; j < 11; j++)` `   a(i, j) = 5;` `cout << a().Sum() << endl;   // -> 500` `cout << a("5").Sum() << endl; // -> 50`
`mstring SymGet(T1 name)`	Retrieves the value of the variable whose name is contained in *name* from the Mumps Interpreter symbol table. Throws *MumpsSymbolTableException* if the variable is not found. The data type **T1** may be **global**, **mstring** or **char***. See also: *SymPut().* `SymPut("k","100");` `cout << SymGet("k") << endl; // -> 100`
`bool SymPut(T1 name, T1 value)`	Insert into the Mumps Interpreter symbol table a variable whose name is contained in *name* with the value contained in *value*. The data type **T1** and **T2** may be any combination of **global**, **char*** or **mstring**. Returns *true* if successful, *false* otherwise. Variables in the Mumps Interpreter symbol table may be accessed by expressions passed to the function *mstring::Eval()* or *mstring::Assign()*. See also: *SymGet()*. `mstring i="3*k";` `SymPut("k","100");` `cout << i.Eval() << endl; // -> 300`
`void global::TermCorrelate(global B)`	*TermCorrelate()* builds a square *term-term* correlation matrix in **global** array B from the invoking **global** array document-term matrix. `global A("A");` `global B("B");` `int main() {` `  long i,j;` `  A("1","computer")=5;` `  A("1","data")=2;` `  A("1","program")=6;`

```
 A("1","disk")=3;
 A("1","laptop")=7;
 A("1","monitor")=1;

 A("2","computer")=5;
 A("2","printer")=2;
 A("2","program")=6;
 A("2","memory")=3;
 A("2","laptop")=7;
 A("2","language")=1;

 A("3","computer")=5;
 A("3","printer")=2;
 A("3","disk")=6;
 A("3","memory")=3;
 A("3","laptop")=7;
 A("3","USB")=1;

 A.TermCorrelate(B);

 mstring a;
 mstring b;

 a="";

 while (1) {
  a=B(a).Order();
  if (a=="") break;
  cout << a << endl;
  b="";
  while (1) {
   b=B(a,b).Order();
   if (b=="") break;
   cout <<"    " << b << "(" << B(a,b)
        << ")" << endl;
   }
  }
 return 0;
 }

Yields:

    USB
    computer(1)
    disk(1)
    laptop(1)
    memory(1)
    printer(1)
computer
    USB(1)
    data(1)
    disk(2)
    language(1)
    laptop(3)
    memory(2)
    monitor(1)
    printer(2)
    program(2)
data
    computer(1)
```

```
    disk(1)
    laptop(1)
    monitor(1)
    program(1)
disk
    USB(1)
    computer(2)
    data(1)
    laptop(2)
    memory(1)
    monitor(1)
    printer(1)
    program(1)
language
    computer(1)
    laptop(1)
    memory(1)
    printer(1)
    program(1)
laptop
    USB(1)
    computer(3)
    data(1)
    disk(2)
    language(1)
    memory(2)
    monitor(1)
    printer(2)
    program(2)
memory
    USB(1)
    computer(2)
    disk(1)
    language(1)
    laptop(2)
    printer(2)
    program(1)
monitor
    computer(1)
    data(1)
    disk(1)
    laptop(1)
    program(1)
printer
    USB(1)
    computer(2)
    disk(1)
    language(1)
    laptop(2)
    memory(2)
    program(1)
program
    computer(2)
    data(1)
    disk(1)
    language(1)
    laptop(2)
    memory(1)
    monitor(1)
    printer(1)
```

`void global::Transpose(global)`	The invoking two dimensional matrix **global** object is transposed and the result is placed in two dimensional **global** array object given as the argument. Any prior contents of the output array out are deleted before the operation commences. `global d("d");` `global f("f");` `d("1","1")=2;` `d("1","2")=3;` `d("2","1")=4;` `d("2","2")=0;` `d().Transpose(f());` `f.TreePrint();` `Results:` `1` `  1=2` `  2=4` `2` `  1=3` `  2=0`
`void global::TreePrint([int, [ char ] ] )`	Prints the invoking global array as a tree. If a the first **int** argument is given, it is the number of spaces to indent each level (default is 1 if not specified). If the second argument is given, it is the character used to indent (default is blank character). See example in *global::Multiply()* above.
`bool ZSeek(FILE *file, mstring offset)` `bool ZSeek(FILE *file, global offset)` `bool ZTell(FILE *file)`	These functions are used in connection with direct access files opened with **FILE** pointers (see: *fopen()*). They are compatible with 64 bit file systems. *ZSeek()* positions the file designated by *file* to the offset specified in *offset*, a positive integer contained in a variable of type **mstring** or **global**. *ZTell()* places the current file offset in the file designated by *file* to the integer value in the **mstring** or **global** variable represented given by *offset*. Both functions return *true* if successful. Ordinarily, file offsets will be obtained by *ZTell()* and these will be stored in a data base. These values will be subsequently used by *ZSeek()* to reposition the file to the point it was at when the *ZTell()* was performed. After re-positioning, the next input or output operation on the file will occur at the point designated by *offset*. All offsets are positive integers relative to the start of the file.

Figure 3 Functions Defined on **mstring** and **global**

Some Function Examples	Results
`char gname[]="doc";` `global doc(gname);` `doc("1")="abcdef";` `mstring ppp = "abcdef";`	

```
mstring aaa;

cout << ppp.Ascii() << endl;                          97
cout << doc("1").Ascii() << endl;                     97
cout << ppp.Ascii(1) << endl;                         97
cout << doc("1").Ascii(1) << endl;                    97

cout << ppp.Length() << endl;                         6
cout << doc("1").Length() << endl;                    6

ppp="aaa & bbb";
aaa="&";

cout << ppp.Length("&") << endl;                      2
cout << ppp.Length("*") << endl;                      1
cout << ppp.Length(aaa) << endl;                      2

doc("1")="&";
cout << ppp.Length(doc("1")) << endl;                 2

string strng="&";
cout << ppp.Length(strng) << endl;                    2

ppp = "123abc456abc";
doc("1")="123abc456abc";
doc("9")="abc";
cout << ppp.Find("abc") << endl;                      7
cout << doc("1").Find("abc") << endl;                 7
cout << ppp.Find("abc",5) << endl;                    13
cout << doc("1").Find("abc",5) << endl;               13
cout << doc("1").Find(doc("9"),5) << endl;            13
strng="abc";
cout << ppp.Find(strng,5) << endl;                    13

cout << Horolog() << endl;                            63815,68346

doc("1").ReadLine();                                  abcdef [input]
cout << "readline global " <<doc("1") << endl;        readline global abcdef

ppp.ReadLine();                                       abcdef [input]
cout << "readline mstring " <<ppp << endl;            readline mstring abcdef

ppp="123";
doc("1")=ppp;
strng="3N";

cout << ppp.Pattern("3N") << endl;                    1

doc("9")="3N";
cout << ppp.Pattern(doc("9")) << endl;                1
cout << doc("1").Pattern("3N") << endl;               1

doc("1")="3N";
cout << ppp.Pattern(doc("1")) << endl;                1

cout << doc("1").Justify(10,2) << endl;                     3.00
cout << doc("1").Justify(10) << endl;                         3N
cout << ppp.Justify(10,2) << endl;                       123.00
cout << ppp.Justify(10) << endl;                            123
```

`cout << doc("1").Data() << endl;`	`1`
`doc("2","3")=123;`	`11`
`cout << doc("2").Data() << endl;`	
`ppp="abcdef";`	
`mstring off="2";`	
`cout << ppp.Extract(2,3) << endl;`	`bc`
`cout << ppp.Extract(off,off+1) << endl;`	`bc`
`cout << ppp.Extract(2) << endl;`	`b`
`cout << ppp.Extract() << endl;`	`a`
`doc("1")=ppp;`	
`cout << doc("1").Extract(2,3) << endl;`	`bc`
`cout << doc("1").Extract(2) << endl;`	`b`
`cout << doc("1").Extract() << endl;`	`a`
`ppp=-123.45678;`	
`cout << ppp.Fnumber("P","2") << endl;`	`(123.46)`
`cout << ppp.Fnumber("P") << endl;`	`(123.457)`
`doc("1")=-123.45678;`	
`cout << doc("1").Fnumber("P","2") << endl;`	`(123.46)`
`cout << doc("1").Fnumber("P") << endl;`	`(123.45678)`
`ppp="abc.def.ghi";`	
`cout << ppp.Piece(".",2) << endl;`	`def`
`cout << ppp.Piece(".",2,3) << endl;`	`def.ghi`
`strng=".";`	
`cout << ppp.Piece(strng,2,3) << endl;`	`def.ghi`
`doc("9")=strng;`	
`cout << ppp.Piece(doc("9"),2,3) << endl;`	`def.ghi`
`doc("1")=".";`	
`cout << ppp.Piece(doc("1"),2) << endl;`	`def`
`cout << ppp.Piece(doc("1"),2,3) << endl;`	`def.ghi`
`long d=1;`	
`float e=1.0;`	
`int f=1;`	
`doc("9")="abcdef";`	
`cout << doc("9").Ascii(e) << endl;`	`97`
`cout << doc("9").Ascii(f) << endl;`	`97`
`cout << doc("9").Ascii(d+1) << endl;`	`98`
`cout << doc("9").Ascii(e+1) << endl;`	`98`
`cout << doc("9").Ascii(f+1) << endl;`	`98`
`off=1;`	
`cout << doc("9").Ascii(off+d) << endl;`	`98`
`cout << doc("9").Ascii(off+e) << endl;`	`98`
`cout << doc("9").Ascii(off+f) << endl;`	`98`
`mstring g=1;`	
`cout << doc("9").Ascii(off+g) << endl;`	`98`
`cout << doc("9").Ascii(off+g) << endl;`	`98`
`cout << doc("9").Ascii(off+g) << endl;`	`98`

Figure 4 Function Examples

Assume that the following entries have been made into the global array data base:

```
 set ^mesh("A01")="Body Regions"
 set ^mesh("A01","047")="Abdomen"
 set ^mesh("A01","047","025")="Abdominal Cavity"
 set ^mesh("A01","047","025","600")="Peritoneum"
 set ^mesh("A01","047","025","600","225")="Douglas' Pouch"
 set ^mesh("A01","047","025","600","451")="Mesentery"
 set ^mesh("A01","047","025","600","451","535")="Mesocolon"
 set ^mesh("A01","047","025","600","573")="Omentum"
 set ^mesh("A01","047","025","600","678")="Peritoneal Cavity"
 set ^mesh("A01","047","025","750")="Retroperitoneal Space"
 set ^mesh("A01","047","050")="Abdominal Wall"
 set ^mesh("A01","047","365")="Groin"
 set ^mesh("A01","047","412")="Inguinal Canal"
 set ^mesh("A01","047","849")="Umbilicus"
 set ^mesh("A01","176")="Back"
 set ^mesh("A01","176","519")="Lumbosacral Region"
 set ^mesh("A01","176","780")="Sacrococcygeal Region"
 set ^mesh("A01","236")="Breast"
 set ^mesh("A01","236","500")="Nipples"
 set ^mesh("A01","378")="Extremities"
 set ^mesh("A01","378","100")="Amputation Stumps"
 set ^mesh("A01","378","610")="Lower Extremity"
 set ^mesh("A01","378","610","100")="Buttocks"
 set ^mesh("A01","378","610","250")="Foot"
 set ^mesh("A01","378","610","250","149")="Ankle"
 set ^mesh("A01","378","610","250","300")="Forefoot, Human"
 set ^mesh("A01","378","610","250","300","480")="Metatarsus"
 set ^mesh("A01","378","610","250","300","792")="Toes"
 set ^mesh("A01","378","610","250","300","792","380")="Hallux"
 set ^mesh("A01","378","610","250","510")="Heel"
 set ^mesh("A01","378","610","400")="Hip"
 set ^mesh("A01","378","610","450")="Knee"
 set ^mesh("A01","378","610","500")="Leg"
 set ^mesh("A01","378","610","750")="Thigh"
 set ^mesh("A01","378","800")="Upper Extremity"
 set ^mesh("A01","378","800","075")="Arm"
 set ^mesh("A01","378","800","090")="Axilla"
 set ^mesh("A01","378","800","420")="Elbow"
 set ^mesh("A01","378","800","585")="Forearm"
 set ^mesh("A01","378","800","667")="Hand"
 set ^mesh("A01","378","800","667","430")="Fingers"
 set ^mesh("A01","378","800","667","430","705")="Thumb"
 set ^mesh("A01","378","800","667","715")="Wrist"
 set ^mesh("A01","378","800","750")="Shoulder"
```

```
global mesh("mesh");
mstring x;
int i,j;

x = "^mesh()"; // initial global array reference - beginning of array
x = x.Query(); // find first real reference

while (1) {
 if (x == "") break; // nothing to print
```

```
 i = x.Qlength(); // how many subscripts
 for (j=0; j<i; j++) cout << " "; // indent by number of subscripts
 cout << x.Qsubscript(i) << " " << x.Eval() << endl; // show index & value
 x = x.Query(); // get next
 }
```

```
The above code yields:

  047 Abdomen
   025 Abdominal Cavity
    600 Peritoneum
     225 Douglas' Pouch
     451 Mesentery
      535 Mesocolon
     573 Omentum
     678 Peritoneal Cavity
    750 Retroperitoneal Space
   050 Abdominal Wall
   365 Groin
   412 Inguinal Canal
   849 Umbilicus
  176 Back
   519 Lumbosacral Region
   780 Sacrococcygeal Region
  236 Breast
   500 Nipples
  378 Extremities
   100 Amputation Stumps
   610 Lower Extremity
    100 Buttocks
    250 Foot
     149 Ankle
     300 Forefoot, Human
      480 Metatarsus
      792 Toes
       380 Hallux
     510 Heel
    400 Hip
    450 Knee
    500 Leg
    750 Thigh
   800 Upper Extremity
    075 Arm
    090 Axilla
    420 Elbow
    585 Forearm
    667 Hand
     430 Fingers
      705 Thumb
     715 Wrist
    750 Shoulder
```

Figure 5 Query(), Qsububscript() and Qlength() Example

5 Examples

```
#include <fstream>
#include <mumpsc/libmpscpp.h>

global doc("doc");
global idf("idf");
```

```
#!/usr/bin/mumps
# weight.mps December 26, 2011
```

```
global indx("index");

int main() {

 FILE *u1;

 ofstream u2 ("document-term-matrix-
              weighted.txt", ios::out);
 assert (u2 != 0);

 mstring d,tt,w,null;
 double x,idfmin=6.0;

 null="";

 indx().Kill();

 for (d=doc(null).Order(); d != null; d =
      doc(d).Order() ) {
  u2 << "doc=" << d << "    ";

  for (w = doc(d,null).Order(); w != null;
       w = doc(d,w).Order()) {
   if (idf(w) < idfmin) {
    doc(d,w).Kill();
    }
   else {
    x = idf(w)*doc(d,w);
    doc(d,w)=x;
    indx(w,d)=x;
    u2 << w << "(" << x << ") ";
    }
   }
  u2 << endl << endl;
  }

 u2.close();

 ofstream u3
  ("term-document-matrix-weighted.txt",
    ios::out);
 assert (u3 != 0);

 for (w=indx(null).Order(); w != null;
      w=indx(w).Order()) {
  u3 << w << "   ";
  for (d=indx(w,null).Order(); d != null;
       d=indx(w,d).Order()) {
   u3 << d << "(" << indx(w,d) << ") ";
   }
  u3 << endl << endl;
  }

 u3.close();

 return 0;

 }
```

```
 open 2:"document-term-matrix-
        weighted.txt,new"



 idfmin=6.0;



 kill ^index

 for d=$order(^doc(d)) do

 . use 2 write !,"doc=",d,?15

 . for w=$order(^doc(d,w)) do

 .. if ^idf<w<idfmin kill ^doc(d,w)


.. else do
 ... set x=^idf(w)*^doc(d,w)
 ... set ^doc(d,w)=x
 ... set ^index(w,d)=x
 ... write w,"(",x,") "
 . write !




close 2

 open 2:"term-document-matrix-
        weighted.txt,new"
 use 2


for w=$order(^index(w)) do

 . write w,?26
 . for d=$order(^index(w,d)) do

 .. write d,"(",^index(w,d),") "

 . write !


 close 2
```

Figure 6 Document Weighting

6 Perl Compatible Regular Expression Library License

Programs written with the MDH may call upon the Perl Compatible Regular Expression Library. In some cases, this library is distributed with the Mumps Compiler. The PCRE Library is not covered by the GNU GPL/LGPL Licenses but, rather, by the license shownn below. The following is the PCRE license:

```
PCRE LICENCE
------------
PCRE is a library of functions to support regular expressions whose syntax
and semantics are as close as possible to those of the Perl 5 language.
Written by: Philip Hazel
University of Cambridge Computing Service,
Cambridge, England. Phone: +44 1223 334714.
Copyright (c) 1997-2001 University of Cambridge
Permission is granted to anyone to use this software for any purpose on any
computer system, and to redistribute it freely, subject to the following
restrictions:
1. This software is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
2. The origin of this software must not be misrepresented, either by
   explicit claim or by omission. In practice, this means that if you use
   PCRE in software which you distribute to others, commercially or
   otherwise, you must put a sentence like this
     Regular expression support is provided by the PCRE library package,
     which is open source software, written by Philip Hazel, and copyright
     by the University of Cambridge, England.
   somewhere reasonably visible in your documentation and in any relevant
   files or online help data or similar. A reference to the ftp site for
   the source, that is, to
     ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/
   should also be given in the documentation.
3. Altered versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
4. If PCRE is embedded in any software that is released under the GNU
   General Purpose Licence (GPL), or Lesser General Purpose Licence (LGPL),
   then the terms of that licence shall supersede any condition above with
   which it is incompatible.
The documentation for PCRE, supplied in the "doc" directory, is distributed
under the same terms as the software itself.
End
```

7 Using Perl Regular Expressions

Author: Matthew Lockner

In addition to Mumps 95 pattern matching using the '?' operator, it is also possible to perform pattern matching against Perl regular expressions via the perlmatch function. Support for this functionality is provided by the Perl-Compatible Regular Expressions library (PCRE), which supports a majority of the functionality found in Perl's regular expression engine.

The perlmatch function works in a somewhat similar fashion to the '?' operator. It is provided with a subject string and a Perl pattern against which to match the subject. The result of the function is boolean and may be used in boolean expression contexts such as the "If" statement.

Some subtleties that differ significantly from Mumps pattern matching should be noted:

A Mumps match expects that the pattern will match against the entire subject string, in that successful matching implies that no characters are left unmatched even if the pattern matched against an initial segment of the subject string. Using perlmatch, it is sufficient that the entire Perl pattern matches an initial segment of the subject string to return a successful match.

The perlmatch function has the side effect of creating variables in the local symbol table to hold backreferences, the equivalent concept of $1, $2, $3, ... in Perl. Up to nine backreferences are currently supported, and can be accessed through the same naming scheme as Perl ($1 through $9). These variables remain defined up to a subsequent call to perlmatch , at which point they are replaced by the backreferences captured from that invocation. Undefined backreferences are cleared between invocations; that is, if a match operation captured five backreferences, then $6 through $9 will contain the null string.

Examples

This program asks the user to input a telephone number. If the data entered looks like a valid telephone number, it extracts and prints the area code portion using a backreference; otherwise, it prints a failure message and exits.

```
Zmain
Write "Please enter a telephone number:",!
Read phonenum
If $$^perlmatch(phonenum,"^(1-)?(\(?\d{3}\)?)?(-| )?\d{3}-?\d{4}$") Do
. Write "+++ This looks like a phone number.",!
. Write "The area code is: ",$2,!
Else  Do
. Write "--- This didn't look like a phone number.",!
Halt
```

The output of several sample runs of the program follows:

Please enter a telephone number:
1-123-555-4567
+++ This looks like a phone number.
The area code is: 123
Please enter a telephone number:
(123)-555-1234
+++ This looks like a phone number.
The area code is: (123)
Please enter a telephone number:
(123) 555-0987
+++ This looks like a phone number.
The area code is: (123)

As in Perl, sections of the regular expression contained in parentheses define what is contained in the backreferences following a match operation. The backreference variables are named in a left-to-right order with respect to the expression, meaning that $1 is assigned the portion matched against the leftmost parenthesized section of the regular expression, with further references assigned names in increasing order. For a much more in-depth treatment of the subject of Perl regular expressions, refer to the *perlre* manpage distributed with the Perl language (also widely available online).

8 Mumps 95 Pattern Matching

Author: Matthew Lockner

Mumps 95 compliant pattern matching (the '?' operator) is implemented in this compiler as given by the following grammar:

```
pattern        ::= {pattern_atom}
pattern_atom   ::= count pattern_element
count          ::= int | '.' | '.' int
                 | int '.' | int '.' int
```

```
pattern_element ::= pattern_code {pattern_code} | string | alternation
pattern_code    ::= 'A' | 'C' | 'E' | 'L' | 'N' | 'P' | 'U'
alternation     ::= '(' pattern_atom {',' pattern_atom} ')'
```

The largest difference between the current and previous standard is the introduction of the alternation construct, an extension that works as in other popular regular expressions implementations. It allows for one of many possible pattern fragments to match a given portion of subject text.

A string literal must be quoted. Also note that alternations are only allowed to contain pattern atoms and not full patterns; while this is a possible shortcoming, it is in accordance with the standard. It is a trivial matter to extend alternations to the ability to contain full patterns, and this may be implemented upon sufficient demand.

Pattern matching is supported by the Perl-Compatible Regular Expressions library (PCRE). Mumps patterns are translated via a recursive-descent parser in the Mumps library into a form consistent with Perl regular expressions, where PCRE then does the actual work of matching. Internally, much of this translation is simple character-level transliteration (substituting '|' for the comma in alternation lists, for example). Pattern code sequences are supported using the POSIX character classes supported in PCRE and are mostly intuitive, with the possible exception of 'E', which is substituted with [[:print][:cntrl:]]. Currently, this construct should cover the ASCII 7-bit character set (lower ASCII).

Due to the heavy string-handling requirements of the pattern translation process, this module uses a separate set of string-handling functions built on top of the C standard string functions, using no dynamic memory allocation and fixed-length buffers for all operations whose length is given by the constant STR_MAX in *sysparms.h*. If an operation overflows during the execution of a Mumps compiled binary, a diagnostic is output to *stderr* and the program terminates. If such termination occurs too frequently, simply increase the value of STR_MAX.

Alphabetical Index