# Introduction to the Mumps Language

## A Quick Introduction to the Mumps Programming Language

Kevin C. O'Kane
Professor Emeritus
Department of Computer Science
University of Northern Iowa
Cedar Falls, IA 50614
kc.okane@gmail.com

A full text on this topic in both print and ebook formats is available on Amazon.com

Videos are available on youtube.com:

https://www.youtube.com/channel/UC5oHS9h-prWeBrBNzXm8rYA

Copies of the software used in this tutorial are are available at:

http://www.cs.uni.edu/~okane/
http://threadsafebooks.com/

November 4, 2017

# Mumps History 1

Mumps (***M****assachusetts General Hospital* ***U****tility* ***M****ulti-programming* ***S****ystem*) is a general purpose programming language environment that provides ACID (*Atomic, Consistent, Isolated, and Durable*) database access by means of program level subscripted arrays and variables. The Mumps database allows schema-less, key-value access to disk resident data organized as trees that may also be viewed as sparse multi-dimensional arrays.

Beginning in 1966, Mumps (also referred to as **M**), was developed by Neil Pappalardo and others in Octo Barnett's lab at the Massachusetts General Hospital (MGH) on a PDP-7, the same architecture on which Unix was being implemented at approximately the same time.

Initial experience with Mumps was very positive and it soon was ported to a number of other architectures including the PDP-11, the VAX, Data General, Prime, and, eventually, Intel x86 based systems, among others. It quickly became the basis for many early applications of computers to the health sciences.

# Mumps History 2

When Mumps was originally designed, there were very few general purpose database systems in existence. The origin of the term 'database' itself dates from this period. Such systems as existed, were mainly *ad hoc* application specific implementations that were neither portable nor extensible. The notion of a general purpose database design was just developing.

One of the first attempts to implement of a general purpose database system was GE/Honeywell's initial IDS - Integrated Data Store - which was developed in the mid-60s. Experience with this system lead to the subsequent creation of the CODASYL DBTG (Committee on Data Systems Languages - Data Base Task Group) whose Network Model database was proposed (1969).

The Network Model was very complex and was implemented, in varying degrees, by only a few vendors. All of these were mainframe based. Most notable of these were GE/Honeywell's IDS/2, Cullinet's Integrated Database Management System (IDMS), Univac's DMS-1100, and Digital Equipment Corporation's DEC-10 based DBMS32.

# Mumps History 3

At about the same time, IBM's IMS (Information Management System), was being developed in connection with the NASA Apollo program. It was first placed into service in 1968 running on IBM 360 mainframes. IMS, which is still in use today, is, like Mumps, a hierarchically structured database.

The table-based relational database model was initially proposed by E. F. Codd in 1970 but it wasn't until 1974 that IBM began to develop System R, the first system to utilize the model, as a research project. The first commercially available relational database system was released by Oracle in 1979.

# Mumps History 4

In late 1960s mini-computers, although expensive, were becoming more widely available but they were still mainly confined to dedicated, mostly laboratory, applications. The operating systems available on these systems were primitive and, for the most part, single user. On many, the user was the operating system, flipping switches to install boot code, and manually loading compilers, linkers and programs from paper or magnetic tape.

DEC's RSX-11, the first commercial multi-user system on the PDP-11, was introduced in 1972. RSTS/E, a time sharing system mainly used for BASIC language programming, was implemented in 1970. Other language support was limited to a small set of languages such as FORTRAN, BASIC and Assembly Language. Although Unix existed at this time, it was not available outside AT&T until 1984.

Thus, in 1966 when a DEC PDP-7 arrived at the Massachusetts General Hospital (MGH), there was very little in the way of software, operating system or database support available for it. So, as there were few other options available, people at MGH started from scratch and designed Mumps to be not only a multi-user operating system, but also a language, and a database, all in one.

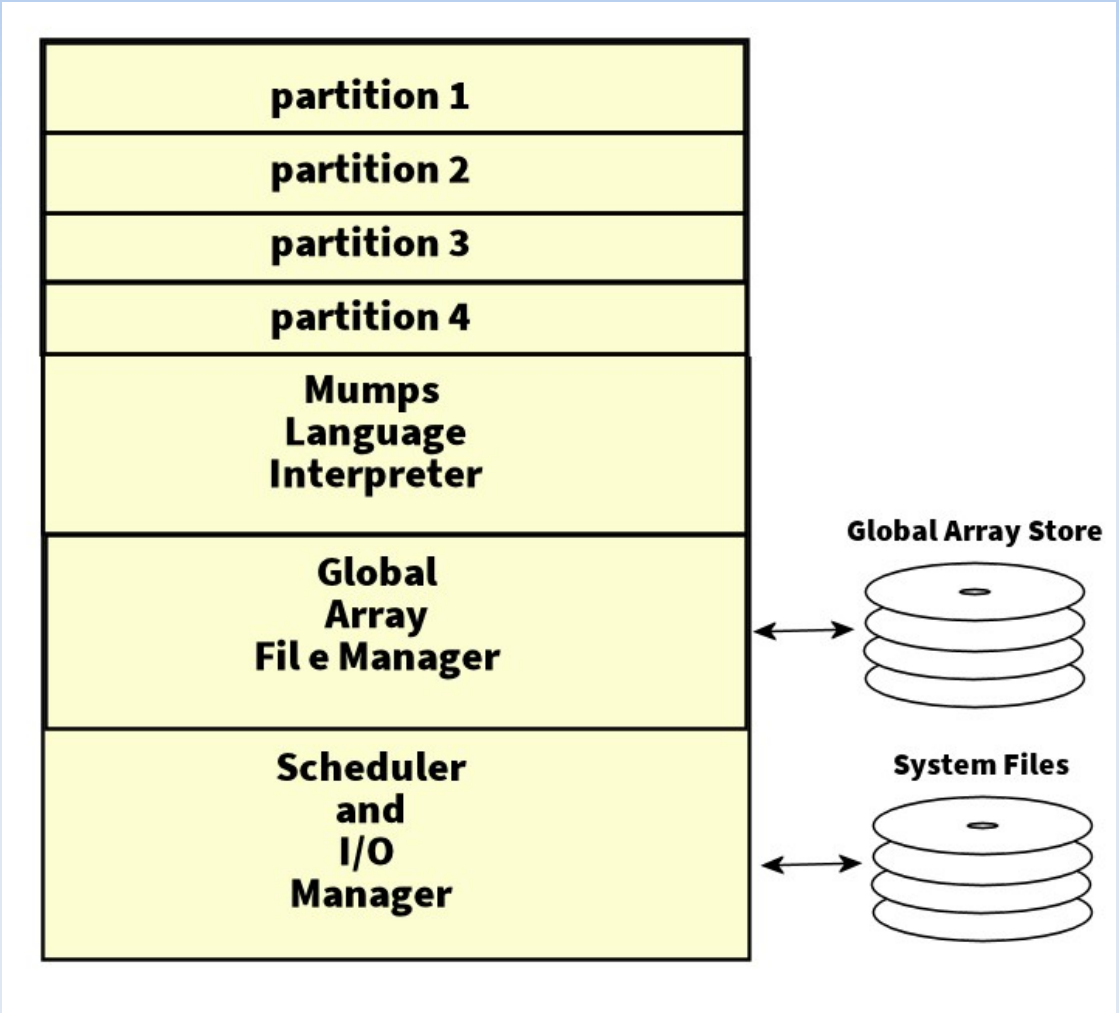
# Mumps History 5

For their database design, they selected a hierarchical model as this closely matched the tree structured nature of the medical record. To represent database trees in the language, they decided to use array references where each successive array index was part of a path description from the root of the array to both intermediate and terminal nodes. They called these disk resident structures *global arrays*.

While in those days, Mumps, out of necessity, was its own standalone operating system, this is not the case today where Mumps programs run in Unix, Linux, OS/X, and Windows based environments.

The early Mumps operating system divided the very limited memory available on early mini-computers into small partitions, each assigned to a separate user. The system allocated and managed the memory partitions and granted time-slices to each user partition usually in a round-robin protocol. The Mumps operating system itself provided the Mumps language interpreter (Mumps was not compiled), centralized I/O, and managed access to the hierarchical database through a centralized Global Array File Manager, as outlined in the following figure.

# Mumps History 6



# Mumps History 7

Memory on early mini-computers was limited, sometimes only a few thousand characters. Mumps programs were loaded into memory as source code rather than as compiled binary. This was done because it was determined that compiled Mumps programs would be far larger than the corresponding source code versions, especially if the source code employed size reducing abbreviations.

While interpreted programs ran slower, the time lost was more than made up by the time saved by not needing to page in and out large binary modules which, initially, was done from magnetic tape. Further, as is the case with most database applications, most program time is spent waiting for database access so the interpretation overhead factor was not really very large.

As an added benefit, small source code modules used less disk space which, when it became available, was very expensive.

# Mumps History 8

The legacy of small memory machines can still be seen to this day, as Mumps programmers tend to abbreviate their code, sometimes excessively, although the original reason for doing so is long past.

Because of its simplicity, low cost, and ease of use, Mumps quickly became popular and was used in many medical applications. COSTAR (COmputer-STored Ambulatory Record), for example, was a medical record, fiscal management and reporting system, developed in the mid-1970s for use in ambulatory care settings and it was widely used and adapted worldwide.

# Mumps History 9

Today, Mumps programs are employed extensively in financial and clinical applications. If you've been to a doctor, been seen at a hospital, or used an ATM machine, your data has probably been processed by a Mumps program.

Mumps programs are the basis of the U.S. Veterans Administration's computerized medical record system VistA (*Veterans Health Information Systems and Technology Architecture*), the largest of its kind in the world. VistA is a collection of 80 different software subsystems that support the largest medical records system in the United States. It supports the medical records of over 8 million veterans, is used by 180,000 medical staff at 163 hospitals, more than 800 clinics, and 135 nursing homes.

Mumps is used by many health care organizations including Allscripts, Epic, Coventry Healthcare, EMIS, Partners HealthCare (including Massachusetts General Hospital), MEDITECH, GE Healthcare (formerly IDX Systems and Centricity), Sunquest Information Systems, DASA, Quest Diagnostics, and Dynacare, among others.

# Mumps History 10

Some of the largest and most well known hospitals use Mumps based electronic health records systems. These include: Kaiser Permanente, Cleveland Clinic, Johns Hopkins Medicine Hospitals, UCLA Health, Texas Health Resources, Massachusetts General Hospital, Mount Sinai Health System in New York City and the Duke University Health System.

Among financial, institutions it is used by Ameritrade, the Bank of England and Barclays Bank, as well as others.

# Mumps History 11

Mumps has remained viable by providing:

- In addition to sequential and direct file access, Mumps also implements, as an integral part of the language, a hierarchical and multi-dimensional database paradigm. When viewed as trees, data nodes can addressed as path descriptions in a manner which is easy for a novice programmer to master in a relatively short time. Alternatively, the trees can be viewed as sparse n-dimensional matrices of effectively unlimited size.
- Mumps supports built-in string manipulation operators and functions that provide programmers with access to efficient methods to accomplish complex string manipulation and pattern matching operations.
- Mumps runs on inexpensive, commodity servers and is easily scaled as demand grows.
- Mumps can handle Big Data quantities of information that are beyond the capabilities of many RDBMS systems with very high levels of performance and throughput.
- Mumps is easily managed without the need for database administrators.
- Mumps databases are ACID (Atomicity, Consistency, Isolation, Durability) Compliant.

# Mumps Implementations

The implementations currently available are:

1. Intersystems (Caché)
   http://www.intersystems.com/ (called Caché®)
2. FIS (GT.M)
   http://www.fisglobal.com/products-technologyplatforms-gtm
3. MUMPS Database and Language by Ray Newman
   http://sourceforge.net/projects/mumps/
4. Open Mumps
   http://www.cs.uni.edu/~okane/

The dialects and extensions accepted by these implementations vary so you should consult the documentation of the version you are using for further details. The examples used here are drawn from GPL Mumps.

# Open Mumps Interpreter

The examples in this introduction were written and tested with the Open Mumps Interpreter, a free, open source distribution for Linux licensed under the GPL V2 License. The distribution is available at:

```
http://www.cs.uni.edu/~okane/
```

Once installed, the interpreter may be executed in interactive command line mode by typing:

```
mumps
```

To exit, type *halt*. Mumps commands may be entered and executed immediately. To execute a program contained in a file, to the interpreter type:

```
goto ^filename.mps
```

You may also, to a command window, type:

```
mumps filename.mps
```

Alternatively, in Linux, a file of Mumps code may be executed directly if you set the file's protections to *executable* and have on its first line:

```
#!/usr/bin/mumps
```

The program may now be executed by typing its name to the command prompt.

Additional documentation is available at the site referenced above.

# Interpreter Examples

```
okane@am6:~$ mumps

Mumps 17.26; Built: 11:21:15 Oct 22 2017
Float: double; DBMS: Native Stand Alone; Hardware math:
http://threadsafebooks.com/
Enter HALT to exit

> for i=1:1:10 write i," "
1 2 3 4 5 6 7 8 9 10
> halt


okane@am6:~$
```

```
okane@am6:~$ cat test.mps
#!/usr/bin/mumps                          Contents of test.mps

 for i=1:1:10 write i," "
 write !
okane@am6:~$ mumps test.mps               Example 1
1 2 3 4 5 6 7 8 9 10
okane@am6:~$ ls -l test.mps               File is executable
-rwxr--r-- 1 okane okane 53 Oct 23 11:14 test.mps
okane@am6:~$ ./test.mps                   Example 2
1 2 3 4 5 6 7 8 9 10
okane@am6:~$ mumps                        Example 3

Mumps 17.26; Built: 11:21:15 Oct 22 2017
Float: double; DBMS: Native Stand Alone; Hardware math: yes;
http://threadsafebooks.com/
Enter HALT to exit

> g ^test.mps
1 2 3 4 5 6 7 8 9 10

> h
```

The first image shows execution of Mumps code directly in the command line interpreter in a Linux terminal window. The command *mumps* invokes the interpreter and the user enters an iterative Mumps command and then exists the interpreter (*halt*).

The second image, also a Linux terminal window, shows in the first division (Contents of test.mps) a small Mumps program named *test.mps*.

The program can be executed by typing it's name as and argument to the name of the interpreter (*mumps test.mps*) as shown in Example 1.

The program can be executed by typing it's name if it's file is listed as executable and, consistent with Linux/Bash usage, on line one, the name of the interpreter to use is given (*/usr/bin/mumps*) (Example 2)

The program can also be executed from the Mumps CLI with a Mumps *goto* (*g*) command (Example 3)

# Mumps Syntax Warning

Mumps syntax will be discussed in detail below but it is important at this time to point out that standard Mumps code may *not* contain embedded blanks except within quoted strings.

In Mumps, a blank is a delimiter.

```
set var=3*x+y
set var = 3 * x + y ; blanks not allowed
```

# Variables 1

Mumps has two types of variables: *local* and *global*.

Global variables are stored on disk and continue to exist when the program that created them terminates.

Local variables are stored in memory and disappear when the program that created them terminates.

A Mumps variable name must begin with a letter or percent sign (%) and may be followed by letters, percent signs, or numbers.

Variable names are case sensitive. The underscore (_) and dollar sign ($) characters are not legal in variable names.

Global variable names are always preceded by a circumflex (^), local variables are not.

The contents of all Mumps variables are stored as varying length character strings. The maximum string length permitted is determined by the implementation but this number is usually at least 512 and often far larger (normally 4096 in Open Mumps).

# Variables 2

In Mumps there are no data declaration statements. Variables are created as needed when a value is assigned to a variable name for the first time.

Values may be assigned to variables by either a **set, merge** or **read** command.

Variables may also be created if they appear as arguments to the **new** command.

Once created, local variables normally persist until the program ends or they are destroyed by a **kill** command. Global variables persist until destroyed by a **kill** command.

In its original implementation, Mumps did not have a means to pass parameters to invoked routines. Consequently, to this day, variables are, ordinarily, known to all routines.

# Variables 3

Mumps variables are not typed. The basic data type is string although integer, floating point and logical (true/false) operations can be performed on string variables if their contents are appropriate.

The values in a string are, at a minimum, any ASCII character code between 32 to 127 (decimal) inclusive. Some implementations permit additional character codings for other languages.

In Open Mumps, some characters outside this range can be generated in **write** commands with the **$char()** function (discussed below).

Variables receive values by means of the **set**, **merge** or **read** commands.

Array references are formed by adding a parenthesized list of indices to the variable name such as:

*name("abc",2,3)*

Indices may evaluate to numbers or strings or both. Strings constants must be quoted, numeric constants need not be quoted.

# Example Variables

```
set %=123                    ; a scalar local variable
set ^x1("ducks")=123         ; ^ducks is a global array
set fido="123"               ; Local variable
set Fido="dog"               ; Names are case sensitive
set x("PI")=3.1414           ; x is a local array reference
set input_dat=123            ; underscore not permitted
set $x=123                   ; $ sign not permitted
set 1x=123                   ; must begin with a letter or %
read ^x(100)                 ; read value into global array element
read %%123                   ; read value into scalar
read _A                      ; underscore error
```

# String Constants

String constants are enclosed in double quote marks (").

A double quote mark itself can be inserted into a string by placing two immediately adjacent double quote marks ("") in the string.

The single quote mark (') is the *not* operator with no special meaning within quoted strings.

The C/C++/Java convention of preceding some special characters by a backslash does not apply in Mumps.

```
"The seas divide and many a tide"
"123.45" (means the same as 123.45)
"Bridget O'Shaunessey? You're not making that up?"
"""The time has come,"" the walrus said."
"\"the time has come" (mismatched quotes)
'now is the time'     (single quote means NOT)
```

# Numeric Constants

Numbers can be integers or floating point. Quotes are optional.

```
100
1.23
-123
-1.23
"3.1415"
```

Some implementations permit scientific notation. Each implementation has limits on accuracy and range.  Consult implementation documentation for details.

In Open Mumps, constants in scientific notation are a special case of strings and must be enclosed in quotes as strings and a numeric operator (such as unary +) is needed to impose a numeric interpretation on the contents:

```
> set i="123E4" set j="100E4"
> write i+j," ",+i," ",+j,!
2.23e+06 1.23e+06 1e+06
```

In GTM, however, quotes are not required

```
GTM>WRITE 8E6
8000000
GTM> WRITE 8E-6
.000008
```

# Mixed Strings & Numeric Constants

Mumps  has some peculiar ways of handling strings when they participate in numeric calculations.

If a string begins with a number but ends with trailing non-numeric characters and it is used as an operand in an arithmetic operation, only the leading numeric portion will participate in the operation. The trailing non-numeric portion will be ignored.

A string not beginning with a numeric character is interpreted numerically as having the value of zero.

# Numeric Interpretation of Strings

```
1+2          is evaluated as 3
"ABC"+2      is evaluated as 2
"1AB"+2      is evaluated as 3
"AA1"+2      is evaluated as 2
"1"+"2"      is evaluated as 3
""           is evaluated as 0
+"-12AB"     is evaluated as -12
+"123.45e4"  is evaluated as 1.2345e+06
```

# Logical Values

Logical values in Mumps are special cases of strings.

A numeric value of zero, any string beginning with a non-numeric character, or a string of length zero is interpreted as *false*.

Any *numeric* string value other than zero is interpreted as *true*.

Logical expressions yield either the digit zero (for *false*) or one (for *true*).

The result of any expression can be used as a logical operand.

# Logical Expressions

Logical expressions yield either zero (for *false*) or one (for *true*). The result of any numeric expression can be used as a logical operand.

The *not* operator is the single quote (')

| | | | |
|---|---|---|---|
| `1` | `true` | `'1` | `false` |
| `0` | `false` | `'0` | `true` |
| `""` | `false` | `'""` | `true` |
| `"A"` | `false` | `'"A"` | `true` |
| `"99"` | `true` | `'"99"` | `false` |
| `"1A"` | `true` | `'"1A"` | `false` |
| `"000"` | `false` | `'"000"` | `true` |
| `"-000"` | `false` | `'"-000"` | `true` |
| `"+000"` | `false` | `'"+000"` | `true` |
| `"0001"` | `true` | `'"0001"` | `false` |

# Arrays 1

Arrays in Mumps come in two varieties: local and global.

Global array names are always prefixed by a circumflex (^) and are stored on disk. They retain their values when a program terminates and, once set, can be accessed by other programs executing at the same time. They can only be deleted by the **kill** command.

Local arrays are destroyed when the program creating them terminates or when they are the subject of a **kill** command. Local arrays are not accessible to other programs unless the other programs are invoked by the program that created them.

Arrays (both global and local) are not declared or pre-dimensioned.

Arrays (both global and local) elements are created by **set**, **merge** or **read** statements when referenced for the first time.

The indices of an array (both global and local) are given by a comma separated list of numbers, or strings, or both, surrounded by parentheses.

# Arrays 2

Arrays (both local and global) are sparse. That is, if you create an element of an array, let us say element 10, it does not mean that Mumps has created any other elements. In other words, it does not imply that there exist elements 1 through 9. You must explicitly create these it you want them.

Array indices may be positive or negative numbers, character strings, or a combination of both.

Arrays in Mumps may have multiple dimensions limited by the maximum line length (at least 512 characters and generally much longer).

Arrays may be viewed as either matrices or trees.

When viewed as trees, each successive index is part of a path description from the root to an internal or leaf node.

Data may be stored (or not stored) at any node along the path of a tree.

Global array names are prefixed with the circumflex character (^) and local arrays are not.

Local arrays are destroyed when the program ends while global arrays, being disk resident, persist.

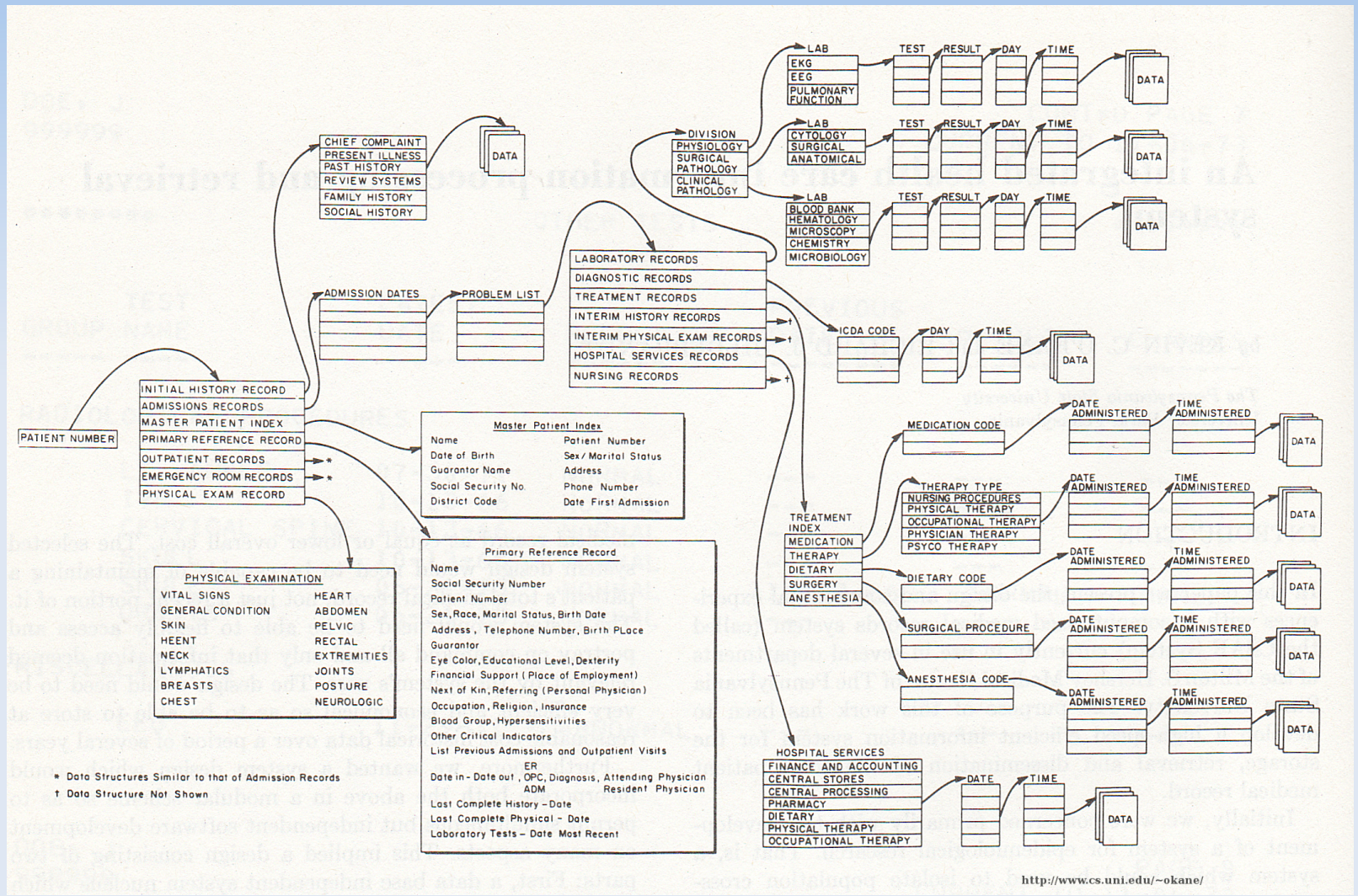
# Arrays 3

```
set a(1,2,3)="text value"          ; local array
set ^b(1,2,3)="text value"         ; global array

set a("text string")=100           ; local array
set ^b("text string")=100          ; global array

set i="testing" set a(i)=1001      ; local array
set i="testing" set ^b(i)=1001     ; global array

set a("Iowa","Black Hawk County","Cedar Falls")="UNI"
set ^b("Iowa","Black Hawk County","Cedar Falls")="UNI"

set a("Iowa","Black Hawk County",Waterloo")="John Deere"
set ^b("Iowa","Black Hawk County",Waterloo")="John Deere"

set a[1][2][3]=123 ; brackets not used for array refs
set a(1, 2, 3)=123 ; no embedded blanks
set a[1,2,3]=123       ; brackets again
```

# Array Examples

```
set a=”1ST FLEET”
set b=”BOSTON”
set c=”FLAG”
set ^ship(a,b,c)="CONSTITUTION"
set ^captain(^ship(a,b,c))="JONES"
set ^home(^captain(^ship(a,b,c)))="PORTSMOUTH"
write ^ship(a,b,c) → CONSTITUTION
write ^captain("CONSTITUTION") → JONES
write ^home("JONES") → PORTSMOUTH
write ^home(^captain("CONSTITUTION")) → PORTSMOUTH
write ^home(^captain(^ship(a,b,c))) → PORTSMOUTH
```

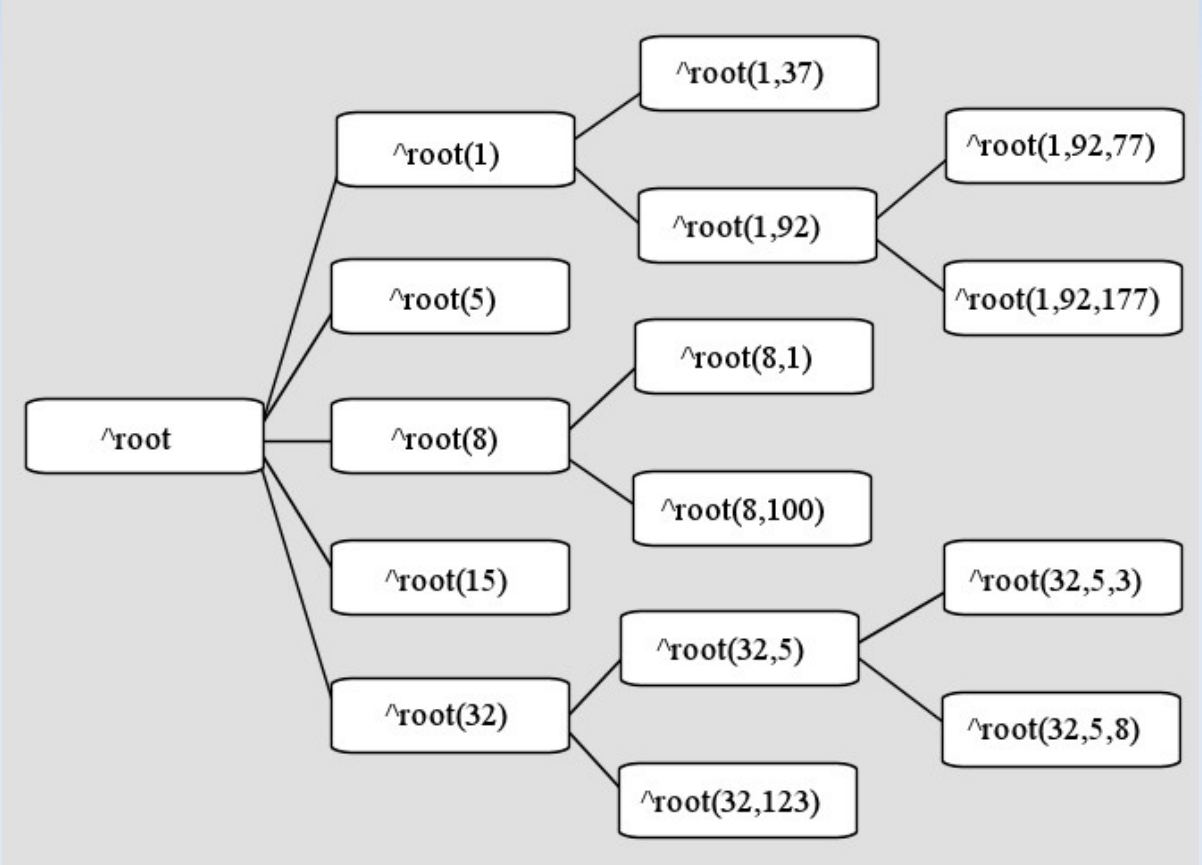
# Hierarchical Data




Mumps was originally written to manage medical records which are often viewed as hierarchically organized. For that reason, the designers needed a convenient means to store data in a tree structure and developed the notion of global arrays as a result.

# Arrays As Trees 1

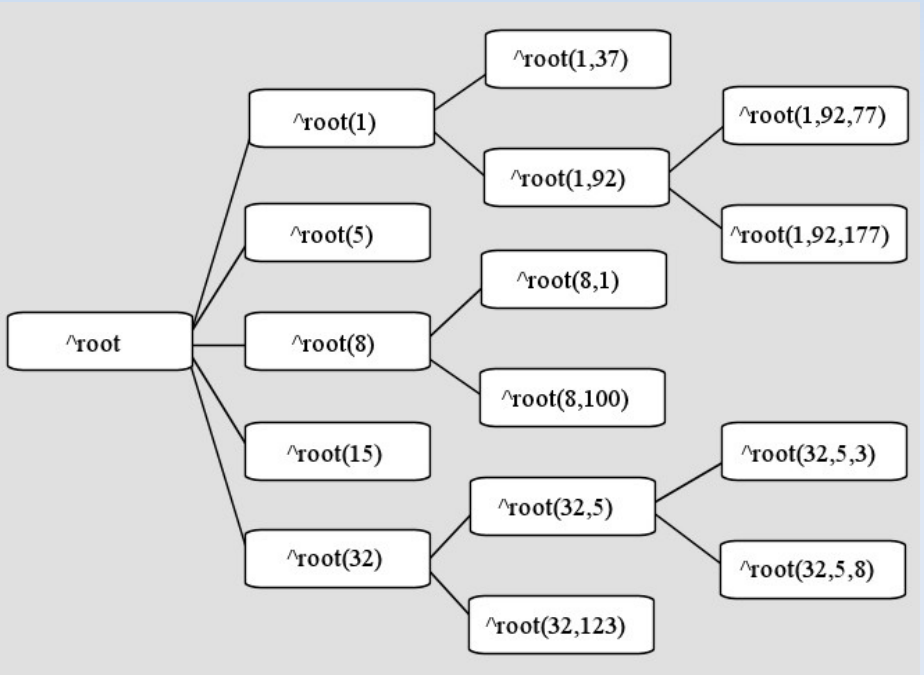
In Mumps, both global and local arrays can be viewed as trees. When viewed as a tree, each successive index in an array reference is interpreted as part of a path description from the root of the array to a node. Nodes along the path may optionally contain data. In the diagram below, the array is named *root*. Numeric indices have been used for simplicity but string indices are also legal.



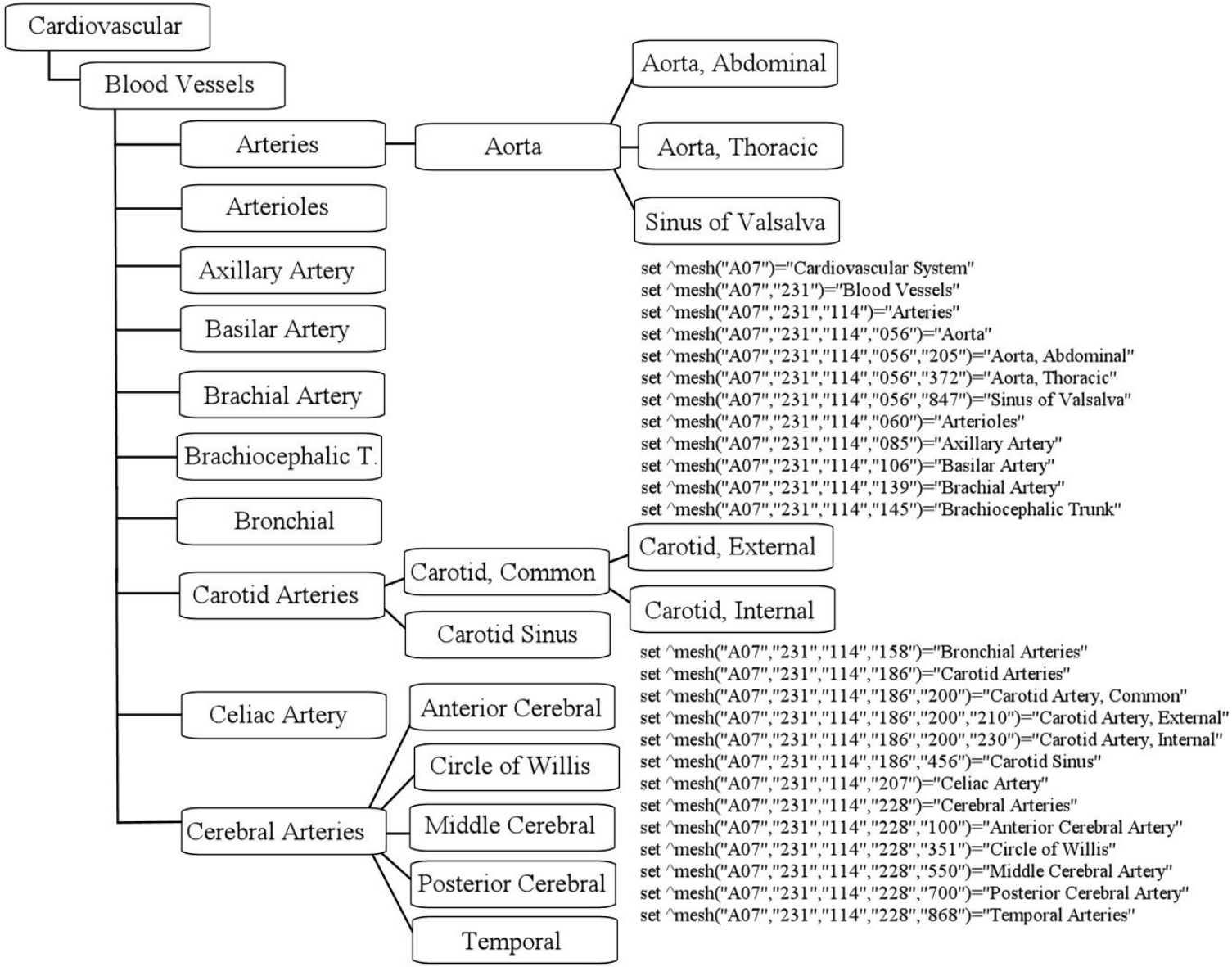
# Arrays as Trees 2

One way data may be inserted into the tree by assignment statements. Not all nodes need have data.

```
set ^root(1,37)=1
set ^root(1,92,77)=2
set ^root(1,92,177)=3
set ^root(5)=4
set ^root(8,1)=5
set ^root(8,100)=6
set ^root(15)=7
set ^root(32,5)=8
set ^root(32,5,3)=9
set ^root(32,5,8)=10
set ^root(32,123)=11
```



# Tree Structured Data




```
set ^mesh("A07")="Cardiovascular System"
set ^mesh("A07","231")="Blood Vessels"
set ^mesh("A07","231","114")="Arteries"
set ^mesh("A07","231","114","056")="Aorta"
set ^mesh("A07","231","114","056","205")="Aorta, Abdominal"
set ^mesh("A07","231","114","056","372")="Aorta, Thoracic"
set ^mesh("A07","231","114","056","847")="Sinus of Valsalva"
set ^mesh("A07","231","114","060")="Arterioles"
set ^mesh("A07","231","114","085")="Axillary Artery"
set ^mesh("A07","231","114","106")="Basilar Artery"
set ^mesh("A07","231","114","139")="Brachial Artery"
set ^mesh("A07","231","114","145")="Brachiocephalic Trunk"
set ^mesh("A07","231","114","158")="Bronchial Arteries"
set ^mesh("A07","231","114","186")="Carotid Arteries"
set ^mesh("A07","231","114","186","200")="Carotid Artery, Common"
set ^mesh("A07","231","114","186","200","210")="Carotid Artery, External"
set ^mesh("A07","231","114","186","200","230")="Carotid Artery, Internal"
set ^mesh("A07","231","114","186","456")="Carotid Sinus"
set ^mesh("A07","231","114","207")="Celiac Artery"
set ^mesh("A07","231","114","228")="Cerebral Arteries"
set ^mesh("A07","231","114","228","100")="Anterior Cerebral Artery"
set ^mesh("A07","231","114","228","351")="Circle of Willis"
set ^mesh("A07","231","114","228","550")="Middle Cerebral Artery"
set ^mesh("A07","231","114","228","700")="Posterior Cerebral Artery"
set ^mesh("A07","231","114","228","868")="Temporal Arteries"
```

# String Indices

```
set ^lab(1234,”hct”,”05/10/2008”,38)=””
set ^lab(1234,”hct”,”05/12/2008”,42)=””
set ^lab(1234,”hct”,”05/15/2008”,35)=””
set ^lab(1234,”hct”,”05/19/2008”,41)=””
```

Mumps permits both numeric and string indices.

Sometimes the indices themselves are the data and nothing is actually stored at the node ("" is the empty string). That is the case in the above where the last index values are the actual test results. Using the functions **$data()** and **$order()** it is easy to navigate through nodes at any level of a tree and retrieve the values of the indices.

In the code above, the Hematocrit (*hct*) results for patient 1234 are stored for several dates. The actual *hct* results for each observation are the last index value.

# Access to Mumps Arrays 1

Mumps array nodes can be accessed directly if you know all the indices.

Alternatively, you can navigate through an array tree by means of the **$data()** and **$order()** builtin functions.

The first of these, **$data()**, tells you if a node exists, if it has data, and if it has descendants.

The second, **$order()**, is used to navigate, at a given level of a tree, from one sibling node to the next node with an alphabetically higher (or lower) index value.

# Access to Mumps Arrays 2

For example, given:

```
set ^a(1)=100,^a(2)=200,^a(3)=300,^a(2,1)=210,^a(4,1)=410

$data(^a(1))   is 1  (node exists, has no descendant)
$data(^a(1,1)) is 0  (node does not exist)
$data(^a(2))   is 11 (node exists, has data, has descendant)
$data(^a(4))   is 10 (node exists, has no data, has descendant)

$order(^a(""))   is 1  (first index at level 1)
$order(^a(1))    is 2  (next higher index)
$order(^a(2,"")) is 1  (first index at level 2 beneath index 2)
$order(^a(4))    is "" (no more indices at level 1)
```

# Global Array Examples 1

Building a conventional three dimensional global array ^*mat1*:

```
for i=1:1:100 do ; store values only at leaf nodes
. for j=1:1:100 do
.. for k=1:1:100 do
... set ^mat1(i,j,k)=0
```

The **do** without an argument causes the block following the command to be executed. Blocks have leading decimal points to indicate their level of nesting.

The matrix ^*mat1()* in this example is similar to a three dimensional matrix in a language such as C or Fortran. There are 1,000,000 cells each initialized with the value zero. The global array may be though of as 100 planes of matrices each 100 by 100.

Access to each element of the matrix requires three indices ( ^*mat1(i,j,k)* ).

# Global Array Examples 2

```
for i=1:1:100 do     ; store values at all node levels
. set ^mat(i)=i
. for j=1:1:100 do
.. set ^mat(i,j)=j
.. for k=1:1:100 do
... set ^mat1(i,j,k)=k
```

This version of the matrix is best thought of as a tree of depth three. At the first level under the root ^*mat*, there are 100 nodes at each of which is stored a value from 1 to 100, inclusive. For each node at level 1, there are 100 descendant nodes at level two each containing a value from 1 to 100. Finally, for each node at level two, there are 100 descendant nodes at level 3 likewise containing values between 1 and 100.

In total, there are 100 level one nodes, 10,000 level two nodes, and 1,000,000 level three nodes. The tree contains 1,010,100 values in total (100 + 10,000 + 1,000,000).

Access to a node requires one, two or three indices depending on the the level of the tree at which the node sought is stored.

# Global Array Examples 3

```
for i=10:10:100 do  ; sparse matrix - elements missing
. for j=10:10:100 do
.. for k=10:10:100 do
... set ^mat1(i,j,k)=0
```

Note: *for i=10:10:100* means iterate with *i* beginning at 10 and incrementing by 10 up to and including 100. Thus, *i* will have the values 10, 20 ... 100.

In this version of the matrix, the array is also a tree but, unlike the other examples, many elements do not exist. At level one, there are only ten nodes (10, 20, 30, ... 90, 100). Each level one node has ten descendants and each level two node likewise has ten descendants.

In total, the tree has 10,110 (10 + 100 + 10,000) nodes.

For example, the node *^a(15,15,15)* does not exist. You may, however, create it with something of the form:

```
set ^mat1(15,15,15)=15
```

Now the tree has 10,111 nodes.

# Mumps Commands

A Mumps program consists of a sequence of commands. Most commands have arguments that are to be executed although some commands do not have arguments.

In most cases, more than one command may appear on a line.

Some examples common commands that have arguments are:

- **set** (assignment)
- **for** (loop control)
- **if** (conditional execution)
- **read** (input)
- **write** (output)

On the other hand, the **halt** command does not have an argument (if it does, it becomes the **hang** command).

Each Mumps command begins with a keyword that may, in most cases, be abbreviated. Most abbreviations are a single letter. Excessively abbreviated Mumps code is compact but difficult to read.

The complete list is given below.

# Postconditionals 1

Mumps commands may optionally be followed by what is known as a *postconditional.*

Postconditionals are truth-valued expressions that immediately follow a command. There are no spaces between a command and a postconditional however, a postconditional is delimited from the command word (or command abbreviation) by a colon.

If the postconditional expression is true, the command and its arguments are executed. If the expression is false, the command and all of its arguments are skipped and execution advances to the next command.

The following is an example of a post-conditional applied to the **set** command:

```
set:a=b i=2
```

The **set** command argument (assign 2 to variable *i*) will be executed only if *a equals b.* Some commands permit individual arguments to be postconditionalized.

# Postconditionals 2

One use of postconditionals of Mumps is for loop control (**for**). The scope of a **for** command is the current line only (although the argumentless **do** and blocks can effectively extend this).

A **for** loop with current line scope creates a problem if a loop needs to terminate for a condition not established in the loop control expression. If you attempt to use an **if** command to remedy the problem, because the scope of an **if** command is also the remainder of the line on which it appears, a similar problem exists.

For example, assume you have a global array named ^*a* which has an unknown number of nodes. Assume that each node is indexed sequentially beginning with 1 (1, 2 , 3 , ...).

If you attempt to print out the values in the nodes with the following:

```
for i=1:1 write ^a(i),!
```

You will encounter an error when you eventually attempt to access a node of the array that does not exist.

# Postconditionals 3

If you add an **if** command:

```
for i=1:1 if '$data(^a(i)) quit  write ^a(i),!
```

you still have a problem (single quote is the *NOT* operator in Mumps).

The expression *'$data(^a(i))* is TRUE if the node *^a(i)* does NOT exist and false otherwise.

The intent is to **quit** the loop when there are no more nodes in *^a* (note the two blanks after the **quit** command - since it has no arguments, two blanks are required if there is another command on the line).

However, the **if** command has scope of the remainder of the line on which it occurs.

Thus the **write** will *not* execute if a node *does* exist because the expression in the **if** will be *false* *($data(^a(i))* is *true* which is then made *false* by the not operator) and the remainder of the line is skipped.

Likewise, if the node does *not* exist (*$data(^a(i))* is *false* but becomes *true* because of the not operator), the loop will terminate due to execution of the **quit** command.

Thus, nothing at all will be printed.

# Postconditionals 4

An **else** command will not help since it will still be on the same line as the **if** and will be ignored if the **if** expression is false **(else** also has single line scope):

```
for i=1:1 if '$data(^a(i)) quit  else  write ^a(i),!
```

Note the two blanks after the **else**.

If ^*a(i)* exists, the remainder of the line is skipped (including the **else**). If ^*a(i)* does not exist, the loop is terminated.

Hence the postconditional was invented.

```
for i=1:1 quit:'$data(^a(i))  write ^a(i),!
```

The **quit** will execute only when the postconditional expression is true which occurs when there are no more nodes of the array. Otherwise, the **quit** is not executed.

Problem solved!

# Operator Precedence

Expressions in Mumps are evaluated strictly *left-to right without precedence*. If you want a different order of evaluation, you *must* use parentheses.

This is true for all Mumps expressions in all Mumps commands. It is a common source of error, especially in **if** commands with compound predicates.

For example, *a<10&b>20* really means *(((a<10)&b)>20)* when you probably wanted *(a<10)&(b>20)* or, equivalently, *a<10&(b>20)*.

# Basic Operators 1

```
Assignment:             =
Unary Arithmetic:       + -
Binary Arithmetic       + addition
                        - subtraction
                        * multiplication
                        / full division
                        \ integer division
                        # modulo
                        ** exponentiation
Arithmetic Relational
                        > greater than
                        < less than
                        '> not greater / less than or equal
                        '< not less / greater than or equal
String Binary           _ concatenate
```

# Basic Operators 2

```
String relational operators

      =    equal
      [    contains - left operand contains right
      ]    follows  - left operand alphabetically follows
                      right operand
      ?    pattern
      ]]   Sorts after

      '=   not equal
      '[   not contains
      ']   not follows
      '?   not pattern
      ']]  not sorts after
```

# Pattern Match Operator

A for the entire upper and lower case alphabet.
C for the 33 control characters.
E for any of the 128 ASCII characters.
L for the 26 lower case letters.
N for the numerics
P for the 33 punctuation characters.
U for the 26 upper case characters.
A literal string.

The letters are preceded by a repetition count. A dot means any number. Consult documentation for more detail.

```
set A="123-45-6789"
if A?3N1"-"2N1"-"4N write "OK"  ; writes OK
if A'?3N1"-"2N1"-"4N write "OK" ; writes nothing

set A="JONES, J. L."
if A?.A1",".A write "OK"  ; writes OK
if A'?.A1",".A write "OK" ; writes nothing
```

# Logical Operators

```
Logical operators:   & and
                     ! or
                     ' not

   1&1       yields 1
   2&1       yields 1
   1&0       yields 0
   1&(0<1)   yields 1
   1!1       yields 1
   1!0       yields 1
   0!0       yields 0
   2!0       yields 1
   '0        yields 1
   '1        yields 0
   '99       yields 0    ; any non-zero value is true
   '""       yields 1    ; strings are false except if they
                         ; have a leading non-zero numeric
```

# Indirection Operator

The indirection operator (@) causes the string in the expression to its right to be executed and the result replaces the indirect expression.

```
set a="2+2"
write @a,!                  ; writes 4

kill ^x
set ^x(1)=99
set ^x(5)=999
set v="^x(y)"
set y=1
set x=$order(@v)            ; equivalent to ^x(1)
write x,!                   ; writes next index of ^x(1): 5
set v1="^x"
set x=$order(@(v1_"("_y_")")) ;
write x,!                     ; writes 5
```

# Commands 1

| | |
|---|---|
| break | Suspends execution or exits a block (non-standard extension) |
| close | release an I/O device |
| database | set global array database (non-standard extension) |
| do | execute a program, section of code or block |
| else | conditional execution based on $test |
| for | iterative execution of a line or block |
| goto | transfer of control to a label or program |
| halt | terminate execution |
| hang | delay execution for a specified period of time |
| html | write line to web server (non-standard extension) |

# Commands 2

| | |
|---|---|
| if | conditional execution of remainder of line |
| job | Create an independent process |
| lock | Exclusive access/release named resource |
| kill | delete a local or global variable |
| merge | copy arrays |
| new | create new copies of local variables |
| open | obtain ownership of a device |
| quit | end a for loop or exit a block |
| read | read from a device |
| set | assign a value to a global or local variable |

# Commands 3

| | |
|---|---|
| shell | execute a command shell (non-standard extension) |
| sql | execute an SQL statement (non-standard extension) |
| tcommit | commit a transaction |
| trestart | roll back / restart a transaction |
| trollback | Roll back a transaction |
| tstart | Begin a transaction |
| use | select which device to read/write |
| view | Implementation defined |
| write | write to device |
| xecute | dynamically execute strings |
| z... | implementation defined - all begin with the letter z |

# Syntax Rules 1

A line may begin with a label. If so, the label must begin in column one.

If column one has a semi-colon (;), the line is a comment. If a semi-colon appears in a position where a command word could appear, the remainder of the line is a comment.

After a label there must be at least one blank or a <*tab*> character before the first command.

If there is no label, column one must be a blank or a <*tab*> character followed by some number of blanks, possibly zero, before the first command.

After most command words or abbreviations there may be an optional post-conditional. No blanks or <*tab*> characters are permitted between the command word and the post-conditional.

If a command has an argument, there must be at least one blank after the command word and its post-conditional, if present, and the argument.

# Syntax Rules 2

Expressions (both in arguments and post-conditionals) may not contain embedded blanks except within double-quoted strings.

If a command has no argument and it is the final command on a line, it is followed by the new line character.

If a command has an argument and it is the final command on a line, its last argument is followed by a new line character.

If a command has an argument and it is not the last command on a line, it is followed by at least one blank before the next command word.

If a command has no argument, there must be two blanks after the command word if there is another command on the line. If it is the last command on a line, it is followed by the new line character.

# Open Mumps Syntax Extensions

In Open Mumps:

If a line begins with a pound-sign (#) or two forward slashes (//), the remainder of the line is taken to be a comment (non-standard extension).

If a line begins with a plus-sign (+), the remainder of the line is interpreted to be an in-line C/C++ statement (non-standard compiler extension).

After the last argument on a line and at least one blank (two if the command has no arguments), a double-slash (//) causes the remainder of the line to be interpreted as a comment  (non-standard extension).

# Line Syntax Examples

```
label set a=123
      set a=123
      set a=123 set b=345
      set:i=j a=123

; standard comment
  set a=123 ; standard comment
# non-standard comment
  set a=123 // non-standard comment
+ printf("hello world\n");  // non-standard C/C++ embed

set a=123              ; only labels, ;, #, or // in col 1
 label set a=123       ; label must be in col 1
 set a = 123           ; no blanks allowed in arguments
 halt:a=b set a=123    ; Halt needs 2 blanks after
                       ; the postconditional
```

# Blocks 1

Originally, all Mumps commands only had line scope. That is, no command extended beyond the line on which it appeared. In later years, however, a block structure facility was added to the language.

This resulted in the introduction of the argumentless **do** command. Originally, all **do** commands had arguments consisting of either a label, offset, file name or combination of these, that addressed a block of code to be invoked as a subroutine.

An argumentless **do** command also invokes a block of code. The block invoked consists of the lines immediately following the line containing the **do** command if they are at a line level one greater than the line level of the **do**. The block ends when the line level declines to the line level of the invoking **do** or lower.

Mumps lines are normally at line level one. A higher line level is achieved by preceding the code on a line with one or more periods. A line with one period is at line level two, one with two periods is at level three and so on.

Execution of a Mumps program normally proceeds from a level one line to the next except as **if**, **else**, and **goto** commands may alter flow. If execution encounters a line at a level greater than the current level, the line is skipped unless it is entered by means of an argumentless **do** command.

# Blocks 2

For example:

```
1)  set a=1
2)  if a=1 do
3)  . write "a is 1",!      ; block dependent on do
4)  write "hello",!
```

The **do** on line 2, at line level one, if executed, invokes the one line block on line 3 which is at line level two. If the **do** is not executed, the block consisting of line 3 is skipped and line 4 is executed after line 2.

Code at line levels greater than one should only be entered by means of an argumentless **do** command. The **goto** command should not be used to enter or exit blocks of code with line levels greater than one.

# Blocks 3

```
1)  set a=1
2)  if a=1 do
3)  . write "a is 1",!
4)  . set a=a*3
5)  else  do
6)  . write "a is not 1",!
7)  . set a=a*4
8)  write "a is ",a,!
```

Because *a* on line 2 has a value of 1, lines 3 & 4 will be executed and lines 6 & 7 will not be executed. Line 8 is executed in either case.

# Blocks 4

```
1)  if a'=0 do
2)  . write "a is 1",!
3)  . set a=a*3
4)  . if a>10 do
5)  .. write "a is greater than 10",!
6)  .. set a=a/2
7)  . set a=a+a
8)  write "a is ",a,!
```

The block beginning on line 2 is entered if variable *a* is not zero. Otherwise, execution skips to line 8

The block beginning at line 5 is entered if variable *a* is greater than 10. Otherwise line 5 & 6 are skipped and execution resumes at line 7.

# Blocks and $Test 1

**$test** is a builtin system variable that indicates if certain operations succeeded (1) or failed (0).

For example, if a **read** command fails to read data (end of file, for example), **$test** will be 0 after the **read** command, otherwise, 1.

The **open** command sets **$test** to be 1 if a file is successfully opened, 0 otherwise. The **if** is another command sets **$test** based on the result of its predicate.

Argumentless **if** and **else** commands where execution of the command is determined by the current value in **$test** (the **else** command is always argumentless).

An oddity in Mumps is that the **else** command is thus not necessarily connected to a preceding **if** command. The **else** command, in fact, is standalone. If the value in **$test** is false, the remainder of the line on which the **else** appears is executed. If the value is true, execution skips to the next line. No **if** is required.

# Blocks and $Test 2

For example:

```
read x
else  write "end of file",! halt
```

The **else** executes based on the value of **$test** set as the result of the **read** command. If the **read** fails **($test** becomes 0), the code on the line following the **else** will execute and the program will halt. If the **read** succeeds **($test** becomes 1), the program will not halt.

However, in practice, an **else** command is often used with a preceding **if** command where **$test** is set by the **if**'s predicate.

In the case of an argumentless **if** command, the value of **$test** determines if the remainder of the line is executed.

Both the argumentless **if** and **else** require at least two blanks following the command word or abbreviation.

# Blocks and $Test 3

The value of **$test** is restored upon exit from a block:

```
1)  set a=1,b=2
2)  if a=1 do                   ; $test becomes true
3)  . set a=0
4)  . if b=3 do                 ; $test becomes false
5)  .. set b=0                  ; not executed
6)  . else  do                  ; executed
7)  .. set b=10                 ; executed
8)  . write $test," ",b,!       ; $test is false
9)  write $test," ",b,!         ; $test restored to true
```

In the above, line 2 sets **$test** to be 1 (the predicate is true). Lines 3 & 4 are executed. Line 4 sets **$test** to be 0 (the predicate is false). Line 5 is not executed.

Line 6 is executed and, since **$test** is false, the **do** is executed and line 7 is executed. Line 8 is executed with **$test** as 0 and *b* as 10.

Line 9 is executed. **$test** is restored to the value it had in line 2 (1). The value of *b* is 10.

# Quit 1

A **quit** command in standard Mumps causes:

1) A single-line scope **for** command to terminate, or
2) A subroutine or function invoked by an entry reference to return, or
3) A code block to be exited.

A **quit** command without arguments requires at least two blanks after it if there are more commands on the line. This case only occurs when a **quit** has a postconditional. If a **quit** does not have a postconditional, the line it is on is terminated unconditionally.

# Quit 2

Using **quit** to terminate a **for** command:

```
 for i=1:1 quit:i>100  write i," ",i*i,!
```

When *i* becomes 101, the loop will terminate.

```
; read and write until no more input ($test becomes 0).
  set f=0
  for  read a quit:'$test  write a,! ; quit, when executed, ends the loop
```

The **for** command without arguments (note the two blanks following it) loops forever. When there is no more data to be read (end of file), **$test** becomes 0 and the **quit** executes and the **for** terminates. If the **read** succeeds, **$test** is 1 and the value read is written and the loop iterates.

# Quit 3

Using **quit** to return form a legacy subroutine

Mumps originally used the **do** command to invoke a local or remote block of code which, when completed, would return the the line containing the invoking **do** command. This was similar to the early BASIC **gosub** statement. The original Mumps **do** command did not allow arguments to be passed.

The argument for a **do** command was either a label, a label with offset, a file name, or a combination of all three (some systems used different syntax):

```
do lab1              ; execute beginning at label lab1
do lab1+3            ; execute beginning at the 3rd line following lab1
do ^file.mps         ; execute the contents of file.mps
do lab1^file.mps     ; execute contents of file.mps beginning at lab1
do lab1+3^file.mps   ; execute file.mps beginning at 3rd line from lab1
```

In each case, when the code block thus invoked ended or encountered a **quit** command, return was made to the invoking **do** command. Note: if the invoking **do** command had additional arguments, they would now be executed in sequence.

```
; using quit as a return from a subroutine
  do top
...
top   set page=page+1
      write #,?70,"Page ",page,!
      quit  ; return to invoking do command
```

# Quit 4

```
; non-standard use of break instead of quit in Open Mumps

  for  do
  . read a
  . if '$test break ; exits the loop and the block
  . write a,!
```

# Quit 5

```
; loop while elements of array a exist
 for i=1:1 quit:'$data(a(i))  write a(i),!

; display all nodes at level one
 set i="" for  set i=$order(^a(i)) quit:i=""  write ^a(i),!

; nested inner loop quits if an element of array b has the value 99
; outer loop continues to next value of i.
  set x=0
  for i=1:1:10 for j=1:1:10 if b(i,j)=99 set x=x+1 quit

; outer loop terminates when f becomes 1 in the block
  set f=0
  for i=1:1:10 do  quit:f=1
  . if a(i)=99 set f=1

; The last line of the block is not executed when i>50
  set f=1
  for i=1:1:100 do
  . set a(i)=i
  . set b(i)=i*2
  . if i>50 quit
  . set c(i)=i*i
```

# Quit 6

Later versions of Mumps permitted functions and subroutines that could pass parameters and return, in the case of functions, values. Thus, the **quit** command can also have an argument:

```
; returning a value from a function

   set i=$$aaa(2)
   write i,!          ; writes 4
   halt

 aaa(x)   set x=x*x
          quit x
```

# Break

Originally, **break** was used as an aid in debugging. See documentation for your system to see if it is implemented.

In the Open Mumps dialect, a **break** command is used to terminate a block (non-standard). Execution continues at the first statement following the block.

```
; non-standard use of break (Open Mumps)

  for  do
  . read a
  . if '$test break ; exits the loop and the block
  . write a,!
```

# Close Command

The **close** command closes and disconnects one or more I/O units. May be implementation defined. All buffers are written to output files as needed and released.

No further I/O may take place to closed unit numbers until a successful **open** command has been issued on the unit.

The syntax of the arguments to this command varies depending on implementation.

```
close 1,2 ; closes units 1 and 2
```

# Do Command

The **do** command executes a dependent or labeled block of code, either in the current program or a file on disk.

```
    if a=b do     ; executes the dependent block that follows
    . set x=1
    . write x
```

---

```
        do abc    ; executes the code block beginning at abc
    ... intervening code ...
abc     set x=1
        write x
        quit      ; returns to invoking do
```

---

```
   do ^abc.mps   ; invokes the code block in file abc.mps
```

---

```
   do abc(123)   ; invokes code at label abc passing an argument
```

# Else Command

The **else** command executes the remainder of the current line if **$test** is false (0).

**$test** is builtin system variable which is set by several commands to indicate if they were successful. No preceding **if** command is required. Two blanks must follow the command.

```
else  write "error",! halt    ; executed if $test is false

else  do
. write "error",!
. halt
```

# For Command 1

The **for** command loops. It can be iterative with the basic format:

```
for variable=start:increment:limit

for i=1:1:10 write i,!     ; writes 1,2,...9,10
for i=10:-1:0 write i,!    ; writes 10,9,...2,1,0
for i=1:2:10 write i,!     ; writes 1,3,5,...9
for i=1:1 write i,!        ; no upper limit - endless
```

# For Command 2

**For** commands can be nested:

```
 for i=1:1:10 write !,i,": " for j=1:1:5 write j," "
```

```
output:

1: 1 2 3 4 5
2: 1 2 3 4 5
3: 1 2 3 4 5
.
.
.
10: 1 2 3 4 5
```

# For Command 3

A comma list of values may also be used:

```
for i=1,3,22,99 write i,!    ; 1,3,22,99
```

Both modes may be mixed:

```
for i=3,22,99:1:110 write i,!  ; 3,22,99,100,...110
for i=3,22,99:1 write i,!      ; 3,22,99,100,...
```

With no arguments, the command becomes loop forever (two blanks required after **for**):

```
set i=1
for  write i,! set i=1+1 quit:i>5   // 1,2,3,4,5
```

# For with Quit 1

**Quit** terminates a **for** loop. Note the two blanks after **for** and **do**

```
set i=1
for  do  quit:i>5
. write i,!
. set i=i+1
```

writes 1 through 5

# For with Quit 2

The **quit** causes a block to terminate and control to be returned to the invoking line of code:

```
 for i=1:1:10 do
 . write i
 . if i>5 write ! quit
 . write " ",i*i,!
```

output:

```
1 1
2 4
3 9
4 16
5 25
6
7
8
9
10
```

# Nested For with Quit

```
  for i=1:1:10 do
  . write i,": "
  . for j=1:1 do  quit:j>5
  .. write j," "
  . write !
```

output:

```
1: 1 2 3 4 5 6
2: 1 2 3 4 5 6
3: 1 2 3 4 5 6
.
.
.
8: 1 2 3 4 5 6
9: 1 2 3 4 5 6
10: 1 2 3 4 5 6
```

# Goto Command

Transfer of control to a local or remote label. Return is not made.

```
goto abc              ; go to label abc in current routine
goto abc^xyz.mps ; go to label abc in file xyz.mps

goto abc:i=10,xyz:i=11  ; argument level postconditionals
```

Note: the arguments of both the **do** and **goto** commands may be individually postconditionalized as seen above. The commands themselves may be postconditionalized as well.

# Halt Command

The **halt** command terminates a program.

```
halt
```

The program terminates. **Halt** takes no arguments but may be postconditionalized.

Two blanks are required after the command if there is another command on the line (meaningful only if the **halt** is postconditionalized).

# Hang Command

Pause the program for a fixed number of seconds. Both **halt** and **hang** have the same abbreviation (**h**) but the **hang** has an argument and the **halt** does not. Both may be postconditionalized.

```
hang 10  ; pause for 10 seconds
```

# If Command

```
Command                                  Output
set i=1,j=2,k=3
if i=1 write "yes",!                    ; yes
if i<j write "yes",!                    ; yes
if i<j,k>j write "yes",!                ; yes
if i<j&k>j write "yes",!                ; does not write
if i<j&(k>j) write "yes",!              ; yes
if i write "yes",!                      ; yes
if 'i write "yes",!                     ; does not write
if '(i=0) write "yes",!                 ; yes
if i=0!(j=2) write "yes",!              ; yes
if a>b open 1:"file,old" else  write "error",! halt
                                        ; the else clause never
                                        ; executes
if  write "hello world",!               ; executes if $test is 1
else  write "goodbye world",!           ; executes if $test is 0
```

# If and Else Commands

The **if** command executes the remainder of the line it appears on if it is followed by an expression and if the expression is true (evaluates as non-zero).

```
if a>b open 1:"file,old"
```

**If** sets **$test.** If the final predicate of the **if** is true, **$test** is 1, 0 otherwise. (**if** commands may have more than one predicate).

An **if** with no arguments executes the remainder of the line if **$test** is true. An **if** with no arguments must be followed by two blanks.

The **else** command is not directly related to the **if** command. An **else** command executes the remainder of the line if **$test** is false (0). An **else** requires no preceding **if** command. An **else** command following an **if** command on the same line will not normally execute unless an intervening command between the **if** and **else** changes **$test** to false. The **else** command never has argumements and is always followed by two blanks.

# Job Command

Creates an concurrently executing independent process. Implementation defined.

# Kill Command

The **kill** command deletes local and global variables and array elements.

```
kill i,j,k       ; removes i, j and k from the local
                 ; symbol table
kill (i,j,k)     ; removes all variables except i, j and k
kill a(1,2)      ; deletes node a(1,2) and any descendants
                 ; of a(1,2)
kill ^a          ; deletes the entire global array ^a
kill ^a(1,2)     ; deletes ^a(1,2) and any descendants
                 ; of ^a(1,2)
```

# Lock Command

The **lock** command marks for exclusive access a global array node and its descendants.

```
lock ^a(1,2)    ; requests ^a(1,2) and descendants
                ; for exclusive access
```

Lock may have a timeout which, if the lock is not granted, will terminate the command and report failure/success in **$test**.

Implementations vary. **Lock** is a voluntary signaling mechanism and does not necessarily prevent access. Consult documentation. See also: transaction processing.

# Merge Command

The **merge** command copies one array and its descendants to another.

```
merge ^a(1,2)=^b  ; global array ^b and its
                  ; descendants are copied
                  ; as descendants of ^a(1,2)
```

# New Command

The **new** command creates a new copy of one or more variables pushing any previous copies onto the stack. The previous copies, if any, will be restored when the block containing the **new** command ends.

```
   if a=b do
   . new a,b          ; variables local to this block
   . set a=10,b=20
   . write a,b,!

; the previous values and a and b, if any, are restored.
; the versions of a and b from the block are deleted
```

# Open, Use and Unit Numbers

The format of the **open** command is implementation dependent. In Open Mumps, unit numbers are used. Unit 5 is always open and considered to be the console and is always open for both input and output. In Linux terms, unit 5 is *stdin* and *stdout.* Other unit numbers are available for assignment by the **open** command.

In Open Mumps, the format of the argument to an **open** command is the unit number followed by a colon followed by a string. The string must contain a file name followed by a comma followed by one of the keywords *old, new,* or *append*.

*old* means that the file exists and is being opened for input.

*new* means that the file is to be created and is being opened for output.

*append* means that the file is being opened for output and new data will be appended to existing data.

**Open** arguments vary widely depending on implementation.

# Open Mumps Open & Use Example

```
open 1:"aaa.dat,old" ; existing file
if '$test write "aaa.dat not found",! halt

open 2:"bbb.dat,new" ; new means create (or re-create)
if '$test write "error writing bbb.dat",! halt

write "copying ...",!

for  do
. use 1          ; switch to unit 1
. read rec       ; read from unit 1
. if '$test break
. use 2          ; switch to unit 2
. write rec,!    ; write to unit 2
close 1,2           ; close the open files
use 5               ; revert to console i/o
write "done",!
```

# Open with Variables

```
set in="aaa.dat,old"
set out="bbb.dat,new"
open 1:in
if '$test write "error on ",in,! halt
open 2:out
if '$test write "error on ",out,! halt
write "copying ...",!
for  do
. use 1
. read rec
. if '$test break
. use 2
. write rec,!
close 1,2
use 5
write "done",!
```

# I/O Format Codes

The **read** and **write** commands have basic format controls for output intended to be displayed or printed. These codes are embedded among command arguments. While they are mainly used with the **write** command, the **read** command permits a a written prompt.

! - new line (!! means two new lines, *etc*.)

# - new page

?x - advance to column "x" (newline generated if needed).

# Read Command

The **read** command reads an entire line into the variable. The command may include a prompt which is written to the device. Reading takes place from the current I/O unit (see **$io**). Variables are created if they do not exist.

```
read a                  ; read a line into a
read a,^b(1),c          ; read 3 lines
read !,"Name:",x        ; write prompt then read into x
                        ; prompts: constant strings, !, ?
read *a                 ; read ASCII numeric code of char typed
read a#10               ; read maximum of 10 characters
read a:5                ; read with a 5 second timeout
                        ; $test will indicate if anything was read
```

# Set Command

The **set** command is the basic assignment command. Each argument consists of a variable or function reference on the left hand side of the assignment operator (=) and an expression to the right. The right hand expression is evaluated and assigned to the variable or passed to the function on the left.

```
set a=10,b=20,c=30,x="abc.def.ghi"
set $piece(x,".",2)="000"
```

In the second example, the second 'piece' of *x* changes from *def* to *000*.

Expressions of the form:

```
set a=b=c=10
```

are not permitted in Mumps.

# Database Transaction Commands

Some versions of Mumps permit database transaction commands. The implementation of these varies so the following commands may or may not be implemented. Check your implementation's documentation for details.

```
TCommit
TREstart
TROllback
TSTART
```

In Open Mumps these are not implemented. However, when used with a SQL backend store (MySQL or PostgreSQL), the full range of SQL transaction controls are available.

# Use Command

The **use** command selects the I/O unit to be used by the next **read** or **write** command. This unit remains in effect for subsequent I/O activity until changed by another **use** command.

Implementation of this command will be vendor specific.

At any given time, one I/O unit is in effect. All **read** and **write** operations default to the current unit until explicitly changed.

```
use 2     ; unit 2 must be open
```

# View Command

The **view** command is vendor defined. It is often used for debugging or similar activities. It is not implemented in Open Mumps.

# Write Command

The **write** command writes text lines to the current I/O unit.

The format codes **!**, # and **?***exp* may be used to skip lines (!), skip to the top of a page (#), or indent to a specific column *(?exp*), respectively.

```
write "hello world",!
set i="hello",j="world" write i," ",j,!
set i="hello",j="world" write i,!,j,!
write 1,?10,2,?20,3,?30,4,!!
```

# Xecute Command

The **xecute** command is used to dynamically execute strings as though they were code.

```
set a="set b=10+456 write b"
xecute a                          ; 466 is written

set a="set c=""1+1"" write @c"

xecute a                          ; 2 is written

set b="a"
xecute @b                         ; 2 is written

for  read x xecute x              ; read and xecute input
```

# Z... Commands

**Z c**ommands are commands that begin with the letter *Z* and are implementation defined. They have no standard meaning.

# Navigating Arrays 1

Global (and local) arrays are navigated by means of the **$data()** and **$order()** functions.

The **$data()** function determines if a node exists, whether it has data assigned to it, and if it has descendants.

The **$order()** permits you to move from one sibling node to another at a given level of an array tree (global or local).

The function **$data()** returns a 0 if the array reference passed as a parameter does not exist. It returns 1 if the node exists but has no descendants, 10 if it exists, has no data but has descendants, and 11 if it exists, has data and has descendants.

The **$order()** function returns the next higher (or lower) value of the last index in the array reference passed to the function.

# Navigating Arrays 2

Function **$order(),** by default, returns indices in ascending collating sequence order unless a second argument of **-1** is given. In this case, the indices are presented in descending  collating sequence order.

**$order(“”)** returns the first value (or last value when the second argument is -1) of the last index of the array reference passed to it.

**$order()** returns an empty string when there are no more nodes at this level of an array tree.

# Navigating Arrays 3

```
kill ^a                      ; all prior values deleted
for i=1:1:9 set ^a(i)=1      ; initialize

write $data(^a(1))           ; writes 1
write $order(^a(""))         ; writes 1
write $order(^a(1))          ; writes 2
write $order(^a(9))          ; writes the empty string (nothing)

set i=5
for j=1:1:5 set ^a(i,j)=j ; initialize at level 2

write $data(^a(5))           ; writes 11
write $data(^a(5,1))         ; writes 1
write $data(^a(5,15))        ; writes 0
write $order(^a(5,""))       ; writes 1
write $order(^a(5,2))        ; writes 3

set ^a(10)=10
write $order(^a(1))          ; writes 10
write $order(^a(10))         ; writes 2

set ^a(11,1)=11
write $data(^a(11))          ; writes 10
write $data(^a(11,1))        ; writes 1
```

# Navigating Arrays 4

The following writes 1 through 5 (see data initializations on previous slide)

```
set j=""
for  set j=$order(^a(5,j)) quit:j="" write j,!
```

The following writes one row per line:

```
set i=""
for  do
. set i=$order(^a(i))
. if i="" break
. write "row ",i," "
. if $data(^a(i))>1 set j="" do
.. set j=$order(^a(i,j))
.. if j="" break
.. write j," " ; elements of the row on the same line
. write ! ; end of row: write new line
```

# Indirection 1

Indirection is indicated by means of the unary indirection operator (**@**) which causes the string expression to its right to be executed as a code expression.

Indirection permits strings created by your program, read from a file, or loaded from a database can be interpretively evaluated and executed at runtime as expressions.

Note: The **xecute** command permits entire commands and lines of code to be executed. The @ operator applies to expressions.

# Indirection 2

```
set i=2,x="2+i"
write @x,!                  ; 4 is written
set a=123
set b="a"
write @b,!                  ; 123 is written
set c="b"
write @@c,!                 ; 123 is written
set d="@@c+@@c"
write @d,!                  ; 246 is written
write @"a+a",!              ; 246 is written
set @("^a("_a_")")=789      ; equiv to ^a(a)=789
write ^a(123),!             ; 789 is written
read x write @x             ; xecute the input expr as code
set a="^m1.mps" do @a       ; routine m1.mps is executed
set a="b=123" set @a        ; 123 is assigned to variable b
```

# Subroutines 1

Originally, subroutines were ordinary local blocks of code in the current routine or in files of source code on disk. These were (and still can be) invoked by a **do** command whose argument is a label indicating the first line of the code block, or the name of a file, or some combination of these.

With this form of subroutine invocation, there are no parameters or return values. However, the full symbol table is accessible to such a subroutine and any changes to a variable made in an invoked block are available upon return. This form of subroutine call is similar to the early BASIC GOSUB statement.

Later versions of Mumps permitted parameters and, for functions, return values. Both *call by value* and *call by name* are supported.

These later changes to Mumps also permit the programmer to create variables local to the subroutine (using the **new** command) which are deleted upon exit.

In most cases, the full symbol table of variables, still remains accessible to a subroutine.

In all cases, all global variables are available to all routines.

# Subroutines 2

```
;       original style of Mumps subroutine invocation

        set i=100
        write i,!   ; writes 100
        do block1   ; invoke local code block
        write i,!   ; writes 200
        halt

block1  set i=i+i
        quit        ; returns to invocation point
```

# Subroutines 3

Invoking a subroutine original style:

```
do lab1                 ; call local code block
do ^file1.mps           ; call file containing program
do lab2^file1.mps       ; call file, entry point lab2
```

Invoking a subroutine with parameters (*call by value*):

```
do lab2(a,b,c)              ; call local label with params
do ^file2.mps(a,b,c)        ; call file program with params
do lab2^file2.mps(a,b,c) ; call file with params, at entry point
                            ; lab2
```

If you pass parameters, they are *call by name* unless you precede their names with a dot:

```
do lab3(.a,.b,.c)          ; call local call by name
do ^file3.mps(.a,.b,.c) ; call file call by name
```

# Subroutines 4

This subroutine creates a variable that is not destroyed on exit. The variable is accessible after return is made.

```
 do two
 write "expect 99 1 ->  ",x," ",$data(x),!
 halt

two
    set x=99
    quit
```

# Subroutines 5

Similar to the original style, but this subroutine uses the **new** command to create new copy of *x* which is deleted upon exit from the routine. The variable *x* is not available after return is made.

```
 set y=99
 do one
 write "expect 99 0  -> ",y," ",$data(x),!
 halt

one new x
    set x=100
    write "expect 99 100 -> ",y," ",x,!
    quit
```

# Subroutines 6

A *Call by value* example. Parameter variable *d* only exists in the subroutine. The variable and any changes to it are lost on exit. The variable *d* does not exist after return is made.

```
 set x=101
 do three(x)
 write "expect 0 ->  ",$d(d),!  ; d only exists in the subroutine
 write “expect 101 -> “,x,!     ; x is unchanged

three(d)
    write "expect 101 -> ",d,!
    set d=d+1
    quit
```

# Subroutines 7

A *Call by name* example. Modification of *z* in the subroutine changes *x* in the caller. Note the *.x* in the call. This signifies *call by name.*

```
 kill
 set x=33
 do four(.x)
 write "expect 44 ->  ",x,!

four(z)
    write "expect 33 -> ",z,!
    set z=44
    quit
```

# Subroutines 8

Using the **new** command. Subroutine *one* creates *x* and subroutine *two* uses it. Changes to *x* in *two* are seen upon return to *one.* Variable *x* is destroyed upon return from subroutine *one*.

```
 set y=99
 do one
 write "expect 99 0  -> ",y," ",$data(x),!

one new x
    set x=100
    write "expect 99 100 -> ",y," ",x,!
    do two
    write "expect 99 99 -> ",y," ",x,!
    quit

two set x=99
    quit
```

# Functions

A function with returns a value. The calling code variable *i* is not changed by the subroutine (call by value). The variable *i* in the function is a temporary copy.

```
 set i=100
 set x=$$sub(i)
 write x," ",i,!  ; writes 500 100
 halt

sub(i)
 set i=i*5
 quit i
```

# Builtin Functions & Variables

Mumps has many builtin functions, called intrinsic functions, and system variables, called intrinsic variables. These handle string manipulation, tree navigation and so on.

Each function and system variable begins with a dollar sign. Some system variables are read-only while others can be set.

While most functions appear in expressions only and yield a result, some functions may appear on the left hand side of an assignment operator or in **read** statements.

# Intrinsic Special Variables 1

| | |
|---|---|
| $Device | Status of current device |
| $ECode | List of error codes |
| $EStack | Number of stack levels |
| $ETrap | Code to execute on error |
| $Horolog | days,seconds time stamp |
| $Io | Current IO unit |
| $Job | Current process ID |
| $Key | Read command control code |
| $Principal | Principal IO device |
| $Quit | Indicates how current process invoked. |
| $STack | Current process stack level |
| $Storage | Amount of memory available |
| Uppercase characters indicate abbreviations | |

# Intrinsic Special Variables 2

| | |
|---|---|
| $SYstem | System ID |
| $Test | Result of prior operation |
| $TLevel | Number transactions in process |
| $TRestart | Number of restarts on current transaction |
| $X | Position of horizontal cursor |
| $Y | Position of vertical cursor |
| $Z... | Implementer defined |
| Uppercase characters indicate abbreviations | |

# Intrinsic Functions 1

| | |
|---|---|
| $Ascii() | ASCII numeric code of a character |
| $Char() | ASCII character from numeric code |
| $Data() | Determines variable's definition |
| $Extract() | Extract a substring[1] |
| $Find() | Find a substring |
| $FNumber() | Format a number |
| $Get() | Get default or actual value |
| $Justify() | Format a number or string |
| $Length() | Determine string length |
| $NAme() | Evaluate array reference |
| $Order() | Find next or previous node |
| $Piece() | Extract substring based on pattern[1] |

Uppercase characters indicate abbreviations.

1. Function may appear on LHS of assignment or in a **read** command

# Intrinsic Functions 2

| | |
|---|---|
| $QLength() | Number of subscripts in an array reference |
| $QSubscript() | Value of specified subscript |
| $Query() | Next array reference |
| $Random() | Random number |
| $REverse() | String in reverse order |
| $Select() | Value of first true argument |
| $STack() | Stack information |
| $Text() | String containing a line of code |
| $TRanslate() | Translate characters in a string |
| $View() | Implementation defined |
| $Z...() | Implementation defined |
| | |

Uppercase characters indicate abbreviations.

# $Ascii()

**$Ascii(arg[,pos])**

**$Ascii()** returns the numeric equivalent of the character argument. If a second argument is given, it is the position of the character in the first argument.

```
$A("A")           yields 65 - the ASCII code for A
$A("Boston")      yields 66 - the ASCII code for B
$A("Boston",2)    yields 98 - the ASCII code for o
```

# $Char()

**$Char(nbr[, ...])**

**$Char()** returns a string of characters corresponding to the ASCII codes given as arguments.

```
$C(65)          yields A - the ASCII equivalent of 65
$C(65,66,67)    yields ABC
$C(65,-1,66)    yields AB - invalid codes are ignored
```

# $Data()

**$Data(var)**

**$Data()** returns an integer which indicates whether the argument exists, has data, and descendants. The value returned is 0 if *var* is undefined, 1 if *var* is defined and has no associated array descendants; 10 if *var* is defined but has no associated value (but does have descendants); and 11 is *var* is defined and has descendants. The argument *var* may be either a local or global variable.

```
set A(1,11)="foo"
set A(1,11,21)="bar"
$data(A(1))          ; yields 10
$data(A(1,11))       ; yields 11
$data(A(1,11,21))    ; yields  1
$data(A(1,11,22)     ; yields  0
```

# $Extract()

**$Extract(e1,i2[,i3])**

**$Extract()** returns a substring of the first argument. The substring begins at the position noted by the second operand. Position counting begins at one.

If the third operand is omitted, the substring consists only of the i2'th character of e1. If the third argument is present, the substring begins at position i2 and ends at position i3.

If only e1 is given, the function returns the first character of the string e1.

If i3 specifies a position beyond the end of e1, the substring ends at the end of e1.

```
$extract("ABC",2) YIELDS "B"
$extract("ABCDEF",3,5) YIELDS "CDE"
```

# $Find()

**$Find(e1,e2[,i3])**

**$Find()** searches the first argument for an occurrence of the second argument. If one is found, the integer returned is one greater than the end position of the second argument in the first argument.

If i3 is specified, the search begins at position i3 in argument e1. Position counting begins with one.

If the second argument is not found, the value returned is 0.

```
$find("ABC","B") YIELDS 3
$find("ABCABC","A",3) YIELDS 5
```

# $FNumber()

**$FNumber(a,b[,c])**

**$Fnumber()** is a function used to format numbers using codes based on local currency flags. See your documentation for details.

```
$FN(100,"P")      yields 100
$FN(-100,"P")     yields (100)
$FN(-100,"T")     yields 100-
$FN(10000,",2")   yields 10,000.00
$FN(100,"+")      yields +100
```

# $Get()

**$Get(var[,default])**

**$get()** returns the current value of a variable, or a default value, if the variable is undefined. If a default value is not specified, the empty string is used.

```
kill x
$get(x,"?")   yields ?
set x=123
$get(x,"?")   yields 123
kill x
$get(x) yields “”
```

# $Justify()

**$Justify(str,fld[,dec]))**

**$Justify()** returns a string in which the first argument is right justified in a field whose length is given by the second argument.

In the three argument form, the first argument is right justified in a field whose length is given by the second argument rounded to *dec* decimal places.

The three argument form imposes a numeric interpretation upon the first argument. If the field length is too small, it will be extended.

```
$justify(39,7)        yields "     39"
$justify("test",7)    yields "   test"
$justify(39,7,1)      yields "   39.0"
$justify(“test”,7,2) yields “   0.00”
$justify(123.45,3)    yields “123.45”
```

# $Length()

**$Length(exp[,str])**

**$Length()** returns the length of the string (in the 1 argument form) or the number of pieces in the string delimited by the second argument.

```
set x="1234 x 5678 x 9999"
$length(x)        yields 18
$length(x,"x")    yields 3 (number parts)
```

# $NAme()

**$NAme(arrayVar[,int])**

**$Name()** Returns a string containing an evaluated representation of all or part of the array variable reference. It does not check to see if the array variable exists. If a second argument is given, it specifies the portion of the representation to return: if zero, the name of the array, otherwise the indices 1 through the value of the integer. If no second argument is provided or if it exceeds the number of indices, the entire representation is returned.

```
set x=10,y=20,z=30,a="abc(x,y,z)"
$na(abc(x,y,z))          yields abc("10","20","30")
$na(abc(x,y,z),1)        yields abc("10")
$na(abc(x,y,z),2)        yields abc("10","20")
$na(abc(10,20,25+5))     yields abc("10","20","30")
$na(@a)                  yields abc("10","20","30")
```

abc() need not exist

# $Order()

**$Order(vn[,-1])**

The **$Order()** function returns the next ascending or descending index at a given level of an array reference. The function traverses an array from one sibling node to the next in key ascending or descending order. The result returned is the next value of the last index of the global or local array given as the first argument to **$Order()**.

The default traversal is in key ascending order by default except if the optional second argument is present and evaluates to "-1" in which case the traversal is in descending key order.

If the second argument is present and has a value of "1", the traversal will be in ascending key order which is the default. In Open Mumps, numeric indices are retrieved in ASCII collating sequence order. Other systems may retrieve subscripts in numeric order. Check your documentation.

# $Order() examples

```
for i=1:1:9 s ^a(i)=i
set ^b(1)=1
set ^b(2)=-1
write "expect (next higher) 1 ",$order(^a("")),!
write "expect (next lower) 9 ",$order(^a(""),-1),!
write "expect 1 ",$order(^a(""),^b(1)),!
write "expect 9 ",$order(^a(""),^b(2)),!
set i=0,j=1
write "expect 1 ",$order(^a(""),j),!
write "expect 9 ",$order(^a(""),-j),!
write "expect 1 ",$order(^a(""),i+j),!
write "expect 9 ",$order(^a(""),i-j),!

set i=""
write "expect 1 2 3 ... 9",!
for  set i=$order(^a(i)) quit:i=”” write i,!

set i=""
write "expect 9 8 7 ... 1",!
for  set i=$order(^a(i),-1) quit:i="" write i,!
```

# $Piece()

**$Piece(str,pat[,i3[,i4]])**

The **$Piece()** function returns a substring of the first argument delimited by the instances of the second argument.

The substring returned in the three argument case is that substring of the first argument that lies between the i3 minus one and the i3 occurrence of the second argument.

In the four argument form, the string returned is that substring of the first argument delimited by the i3 minus one instance of the second argument and the i4 instance of the second argument. If only two arguments are given, i3 is assumed to be 1.

The function may appear on the left hand side of an assignment operator in which case the substring addressed is replaced by the result of the right hand side of the assignment operator.

```
$piece("A.BX.Y",".",2)     yields "BX"
$piece("A.BX.Y",".",1)     yields "A"
$piece("A.BX.Y",".")       yields "A"
$piece("A.BX.Y",".",2,3)   yields "BX.Y"

set x="abc.def.ghi"
set $piece(x,".",2)="xxx" causes x to be "abc.xxx.ghi"
```

# $QLength()

**$QLength(string)**

**$QLength()** returns the number of subscripts contained in the array reference in the string argument

```
set x="a(1,2,3)"
write $qlength(x)
write $qlength("^a(i,j)"),!
write $qlength("a"),!
```

writes 3, 2 and 0

# $QSubscript(string, int)

**$QSubscript(string,int)**

The **$QSubscript()** function returns a portion of the array reference given by the first string. If the second integer argument is -1, the environment is returned (if defined in your implementation), if 0, the name of the global array is returned. In Open Mumps, subscripts of arrays are evaluated before being returned.

For values greater than 0, the value returned is that of the associated subscript.

If a value exceeds the number of indices, an empty string is returned.

# $QSubscript() Examples

```
set i=10,j=20,k=30
set x="^a(i,j,k)"
write $qsubscript(x,0),!
write $qsubscript(x,1),!
write $qsubscript(x,2),!
write $qsubscript("^a(i,j,k)",3),!
```

writes ^a, 10, 20, and 30 respectively.

Note that in Open Mumps, unlike other versions, the indices of the array are evaluated.

# $QUery()

**$QUery(string)**

The **$QUery()** function returns the next array element in the array space denoted by the string argument.

The argument to **$query(string)** is a global or local array reference (not a string like the other **$q**... functions). The value returned is a string containing the next ascending entry in the array space or, if there are no more, the empty string.

# $QUery() Examples

```
set a(1,2,3)=99
set a(1,2,4)=98
set a(1,2,5)=97

set x="a"
set x=$query(@x)
write "expect a(1,2,3) ",x,!

set x=$query(@x)
write "expect a(1,2,4) ",x,!

set x=$query(@x)
write "expect a(1,2,5) ",x,!

write “expect a(1,2,3) “,$query(a(1)),!
```

# $Random()

**$Random(int)**

**$Random()** returns a random number between zero and one less than the integer argument.

```
$random(10)        yields a random number between 0
                   and 9
```

# $Reverse()

**$REverse(str)**

**$Reverse()** returns the string passed as the argument in reverse order.

```
$reverse("abc")   yields cba
```

# $Select()

**$Select(texp1:exp1[,...])**

**$Select()** evaluates each truth valued expression (texp1, ...) and, if true, returns the result of the corresponding expression following the colon (:). Evaluation terminates at the first true expression.

```
set x=10
$select(x=9:"A",x=10:"B",x=11:"C",1:””) yields B

set x=22
$select(x=9:"A",x=10:"B",x=11:"C",1:””) yields “”
```

# $Stack()

**$STack(intexp1[,...])**

**$Stack()** returns information concerning the Mumps stack environment based on the numeric codes supplied. Consult your documentation for details as the relate to your implementation. Not implemented in Open Mumps.

# $Text()

**$Test(entryRef)**

**$Test()** returns a string from the routine at the location given by the entry reference (label, offset, and/or routine).

Assume the program code:

```
L1      set a=10
        set b=20
        set c=30
; line of comment

$text(L1)       yields "L1 set a=10"
$text(L1+1)     yields "   set b=20"
$text(4)        yields "; line of comment"
```

# $TRanslate()

**$TRanslate(exp1[,exp2[,exp3)**

Returns exp1 after dropping or substituting characters. If the second and third operands are omitted, the original string is returned. Characters from the first operand are selected if they occur in the second and (1) are replaced by the character from the third operand which positionally corresponds to the second operand or, (2) dropped if there is no corresponding third operand character (third operand is shorter than second).

```
set x="arma virumque cano"

$tr(x,"a")      yields "rm virumque cno"
$tr(x,"a","A") yields "ArmA virumque cAno"
```

# $View()

**$View()**

**$View()** is implementation defined.

# $Z....()

**$Z...()**

**$Z...()** functions are added by the implementor and are, thus, implementation defined. See your documentation.

# Programming Example A.1

We have a file of text each very lone line of which is represents an abstract from a medical journal.

The first token (tokens are delimited by a blank) on each line is a number that refers back to the original journal article. This is followed by the article number (*1,2,3...*). There then follow an arbitrary number of word tokens each separated from one another by a blank.

The words are the text of the abstract of the original journal article modified as follows:

1) All words are in lower case and punctuation, except for hyphens, has been removed.
2) Common words (*and, or, the ...*) have been removed.
3) The resulting lane may be very long if the original abstract was long.

There is a Mumps global array named ^*Title* indexed by document number that contains the original article's title.

# Programming Example A.2

Example line from the file:

*5745 5 platelet affinity serotonin increased alcoholics former alcoholics biological marker dependence? kinetics serotonin platelet uptake were studied alcoholics former alcoholics see whether differences found between alcohol-preferring non-preferring rats could reproduced man three groups patients were studied dependent alcoholics admission treatment dependent alcoholics after days treatment former dependent alcoholics abstinent years controls were non-alcoholics matched age sex km serotonin uptake platelets lower patients from three groups compared controls this phenomenon could congenital induced the previous excessive intake alcohol believe that this increased platelet affinity serotonin the absence cirrhosis the liver or depression could a marker alcohol dependence enabling therapeutic effort be focussed these patients*

This is article 5. The contents of ^*Title(5)* are:

*Platelet affinity for serotonin is increased in alcoholics and former alcoholics: a biological marker for dependence?*

# Programming Example A.3

*Objective:* Write a Mumps program that will permit Boolean searching of words in the file.

For example, if we want to find all the articles containing both *seratonin* and *cirrhosis*, we would enter the following to our Mumps program:

*serotonin & cirrhosis*

The program will scan the file and locate those abstracts containing both words and respond by printing the abstract number and title of the corresponding article.

# Programming Example A.4

For example:

```
$ ./slides.boolean.mps

Enter query terms serotonin & cirrhosis

Mumps expression to be evaluated on the data set: $f(line,"serotonin")&$f(line,"cirrhosis")

5        Platelet affinity for serotonin is increased in alcoholics and former alcoholics

10000 documents searched
```

(long titles truncated)

# Programming Example A.5

Another Example:

```
$ ./slides.boolean.mps

Enter query terms serotonin & platelet

Mumps expression to be evaluated on the data set: $f(line,"serotonin")&$f(line,"platelet")

5       Platelet affinity for serotonin is increased in alcoholics and former alcoholics
269     Cooperative mediation by serotonin S and thromboxane A prostaglandin H receptor
2724    The effect of ketanserin on blood pressure and platelets during cardiopulmonary
3804    Hypersensitivity of phospholipase C in platelets of spontaneously hypertensive r
8399    Cerebral vasoconstrictor responses to serotonin after dietary treatment of ather

10000 documents searched
```

(long titles truncated)

# Programming Example A.6

The program:

```
 1) #!/usr/bin/mumpsRO
 2) # Copyright 2016 Kevin C. O'Kane
 3) # boolean.mps February 14, 2014
 4) # assumes that ^titles(docnbr) exists
 5)
 6)  read "Enter query terms ",query
 7)
 8)  set query=$zlower(query)
 9)  set exp=""
10)
11) for i=1:1 do
12) . set w=$piece(query," ",i)
13) . if w="" break
14) . if $find("()",w) set exp=exp_w continue
15) . if w="|" set exp=exp_"!" continue
16) . if w="~" set exp=exp_"'" continue
17) . if w="&" set exp=exp_"&" continue
18) . set exp=exp_"$f(line,""_w_"")"
19)
20) write !,"Mumps expression to be evaluated on the data set: ",exp,!!
21)
```

# Programming Example A.7

```
22) set $noerr=1  // turns off error messages
23) set line="  " set i=@exp  // test trial of the expression
24) if $noerr<0 write "Expression error number ",-$noerror,!  got to again
25)
26) set M=10000
27)
28) set file="osu.converted,old"
29)
30) open 1:file
31) if '$test write "file error",! halt
32)
33) set i=0
34)
35) for j=1:1:M do
36) . use 1
37) . read line
38) . if '$test break
39) . if @exp do
40) .. set off=$piece(line," ",1)
41) .. set docnbr=$piece(line," ",2)
42) .. use 5
43) .. write docnbr,?10,$e(^title(docnbr),1,80),!
44)
45) use 5
46) write !,M," documents searched",!!
47) halt
```

# Programming Example A.8

The details:

The program initially reads in a query in the form of a logical expression involving words, parentheses and symbols separated by blanks. For example:

```
word1 & word1
( word1 & word2 ) | word3
( word1 | word2 ) & ( word4 | word5 )
( word1 & word2 ) & ~ word3
```

where & means *and*, | means *or* and ~ means *not.* Thus, the last expression above would retrieve titles of those documents that contain both *word1* and *word2* but not *word3*.

The program initially converts all query text to lowercase using a built in function. It then extracts each blank delimited token and builds a Mumps logical expression that corresponds to the input. In place of words, the program substitutes an expression of the form:

```
$f(line,"word1")
```

The end result is an executable Mumps expression (*exp*). for example, the 3rd expression above would translate to:

```
($f(line,"word1")!$f(line,"word2"))&($f(line,"word4")!$f(line,"word5"))
```

# Programming Example A.9

The expression in *exp* is then tested for syntax (a special feature of the Open Mumps Interpreter):

```
set $noerr=1  // turns off error messages
set line="  " set i=@exp  // test trial of the expression
if $noerr<0 write "Expression error number ",-$noerror,!  got to again
```

If the expression parses, the program proceeds to read (up to a limit) lines from the file of abstracts and apply the expression to each line. The builtin variable *$noerr* is less than zero if the expression contains a syntax error.

# Programming Example A.10

If the expression parses, the program proceeds to read (up to a limit) lines from the file of abstracts and apply the expression (*exp*) to each line:

```
for j=1:1:M do
 . use 1
 . read line
 . if '$test break
 . if @exp do
 .. set off=$piece(line," ",1)
 .. set docnbr=$piece(line," ",2)
 .. use 5
 .. write docnbr,?10,$e(^title(docnbr),1,80),!
```

If the expression is *true*, that is, the words are found (or not found) by *$find()* and the *and / or / not* logic is *true*, the title corresponding to the abstract number is fetched and printed.