Mumps Language Tutorial

A Quick Overview of the Basics of the Mumps Language

by
Kevin C. O'Kane
Professor Emeritus
Department of Computer Science
University of Northern Iowa
Cedar Falls, IA 50614

kc.okane@gmail.com

Version 1.04
March 1, 2014

Mumps History

Beginning in 1966, Mumps (also referred to as ***M***), was developed by Neil Pappalardo and others in Dr. Octo Barnett's lab at the Massachusetts General Hospital on a PDP-7. It was later ported to a number of machines including the PDP-11 and VAX.

Mumps is a general purpose programming language that supports a novel, native, hierarchical database facility. The acronym stands for the **M**assachusetts General Hospital **U**tility **M**ulti-programming **S**ystem. It is widely used in financial and clinical applications and remains to this day 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.

Mumps Implementations

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

The dialects accepted by these systems vary. The examples in these slides will be drawn from GPL Mumps. You should consult the web site of the version you are using for further documentation. In the slides, non-standard extensions used by GPL Mumps are noted. These may not be present in other versions.

For full documentation and examples, see the web site noted and amazon.com (search for mumps language).

GPL Mumps Interpreter

The GPL Mumps Interpreter (and compiler), written in C/C++, are open source software under the GPL V2 License. The main version is for Linux but the software will run in Windows if Cygwin is installed.

The interpreter may be executed in interactive command line mode by typing, in a terminal window:

```
mumps
```

To exit, type *halt*. In this mode mumps commands may be entered and executed immediately. To execute a program contained in a file, type:

```
goto ^filename.mps
```

A file may be executed without starting the interpreter if you set its 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.

Variables

Mumps has *local* and *global* variables. Global variables are stored on disk and continue to exist when the program creating them terminates. Local variables are in memory and disappear when the program which created them ends.

A Mumps variable name must begin with a letter or percent sign (%) and may be followed by either letters, the percent sign, or numbers. Variable names are case sensitive. The underscore (_) and dollar sign ($) characters are not legal in variable names.

Global variable names are preceded by a circumflex (^).

The contents of all Mumps variables are stored as varying length character strings. The maximum string length permitted is determined during system configuration but this number is usually at least 4096.

Long variable names may have a negative impact on performance because they impact lookup time.

Mumps Variables

In Mumps there are no data declaration statements. Variables are created as needed.

Variables are created when a value is assigned for the first time by either a **set** or **read** command or if they appear as arguments to the **new** command.

Once created, variables normally persist until the program ends or they are destroyed by a **kill** command. Ordinarily, variables are known to all routines.

Mumps Variables

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

Variables receive values by means of the **set**, **merge** and **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 be numbers or strings or both. Strings must be quoted, numbers need not be quoted.

Variables

```
set %=123
set ^x1("ducks")=123        ; global array reference
set fido="123"              ; names are case sensitive
set Fido="dog"
set x("PI")=3.1414          ; 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 %%123
read _A
```

Mumps Strings

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

A double quote mark itself can be inserted in 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"
'now is the time'
```

Mumps Numbers

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 of accuracy and size. Consult documentation.

Mumps Strings & Numeric Expressions

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 numeric characters is interpreted as having the value of zero.

Numeric Interpretation of Strings

```
1+2      will be evaluated as 3
"ABC"+2  will be evaluated as 2
"1AB"+2  will be evaluated as 3
"AA1"+2  will be evaluated as 2
"1"+"2"  will be evaluated as 3
""       will be evaluated as 0
```

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 numeric 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 '

"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

Mumps Arrays

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

Global array names are prefixed by circumflex (^) and are stored on disk. They retain their values when a program terminates and can be accessed by other programs.

Local arrays are destroyed when the program creating them terminates and are not accessible to other programs unless the other program was invoked by the program which created the variable.

Arrays are not declared or pre-dimensioned.

A name used as an array name may also, at the same time, be used as the name of a scalar or a label.

Array elements are created by assignment (**set**) or appearance in a **read** statement.

The indices of an array are specified as a comma separated list of numbers or strings or both.

Mumps Arrays

Arrays 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 or character strings or a combination of both.

Arrays in Mumps may have multiple dimensions limited by the maximum line length (512 nominally).

Arrays may be viewed as either matrices or trees.

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

Data may be stored at any node along the path of a tree.

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

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

Mumps Arrays

Mumps arrays can be accessed directly if you know the indices.

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

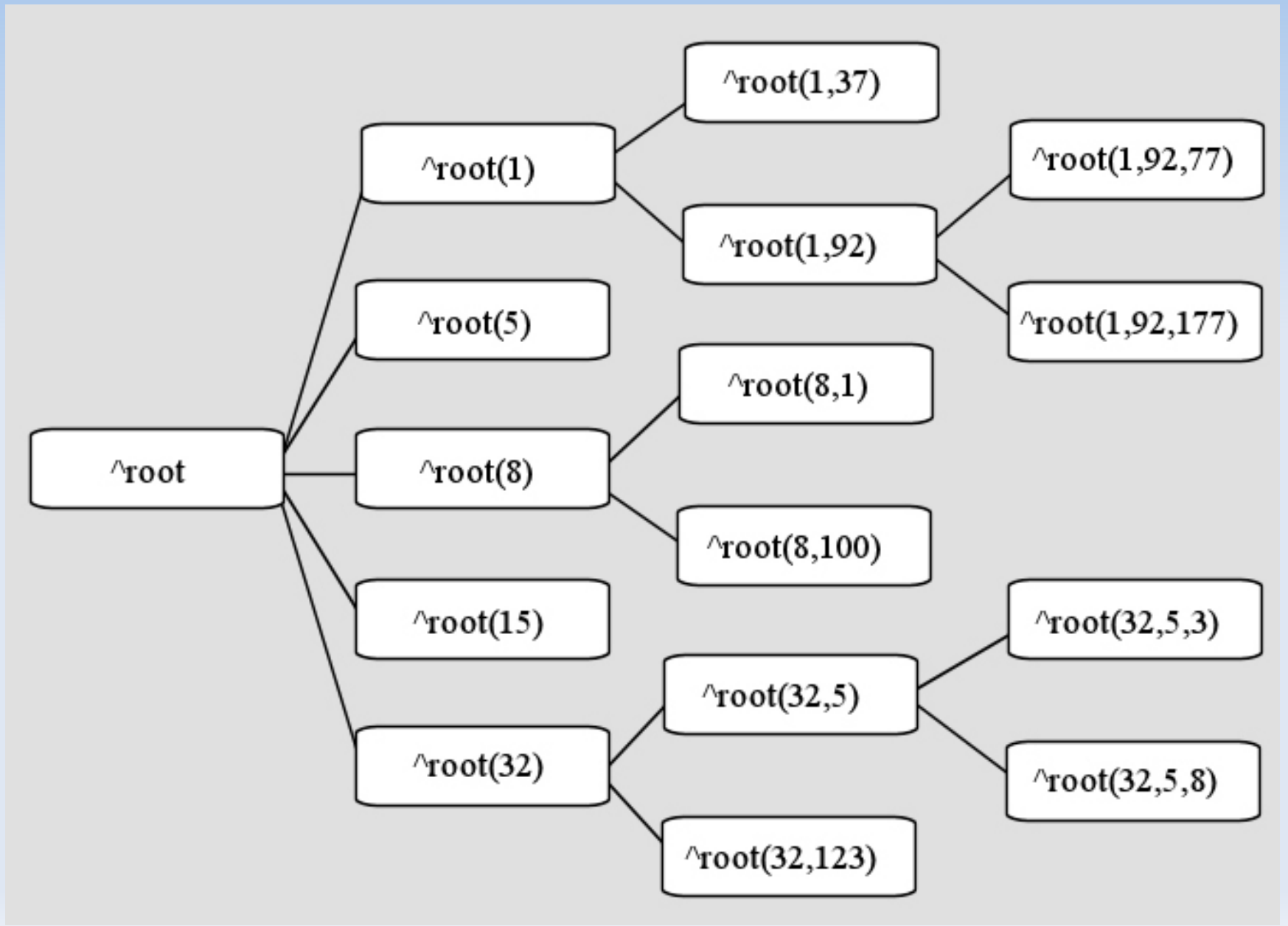
The first of these, **$data()**, indicates if a node exists, if it has data and if it has descendants.

The second, **$order()**, is used to navigate from one sibling node to the next (or prior) at a given level of a tree.

Local Arrays

```
set a(1,2,3)="text value"
set a("text string")=100
set i="testing" set a(i)=1001
set a("Iowa","Black Hawk County","Cedar Falls")="UNI"
set a("Iowa","Black Hawk County",Waterloo")="John Deere"
set a[1][2][3]=123
set a(1, 2, 3)=123
set a[1,2,3]=123
```

Arrays as Trees



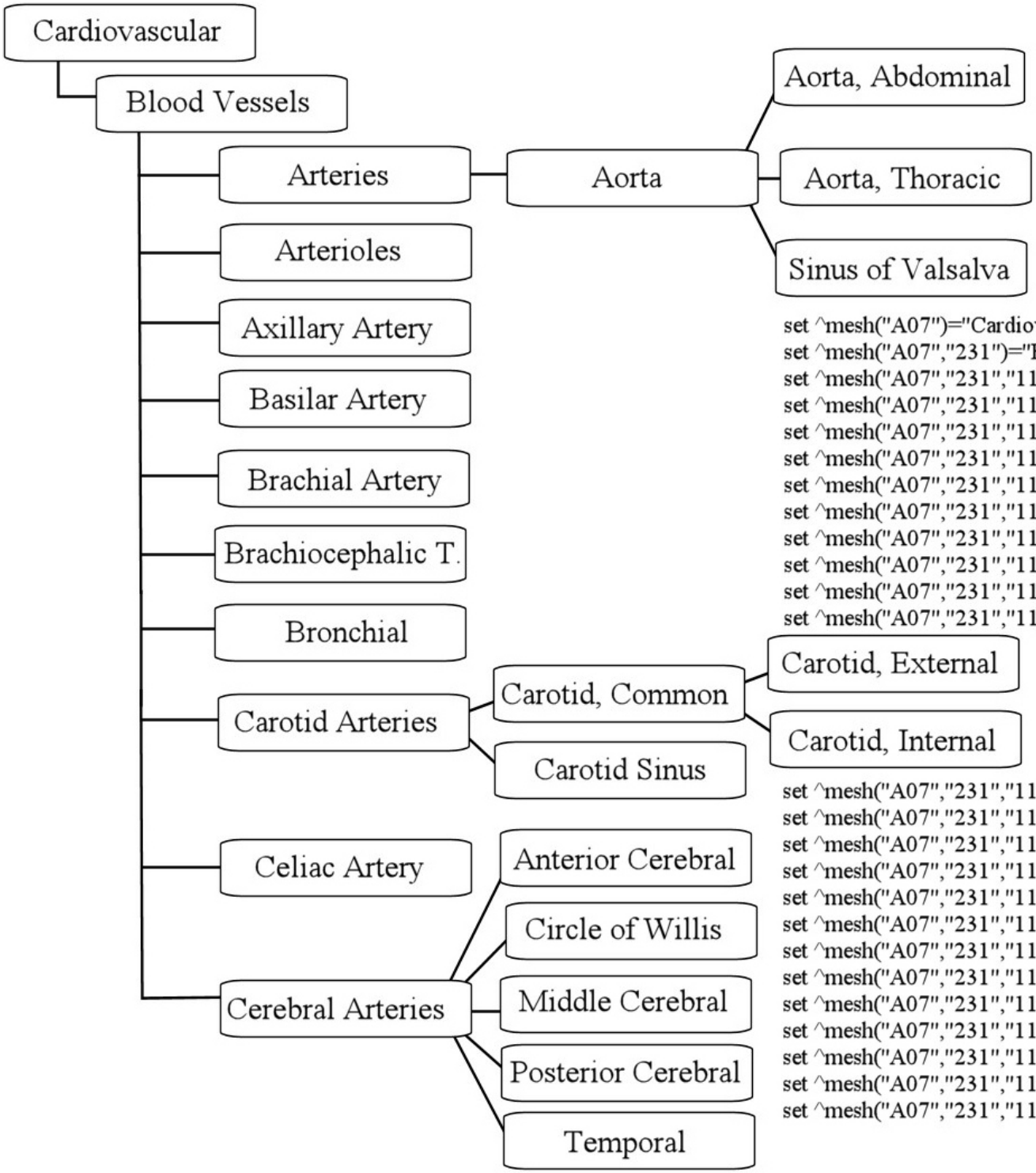
Creating Global Arrays

```
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
```

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)=""
```

Note: sometimes the indices themselves are the data and nothing ("") is actually stored at the node. That is the case here where the last index is the test result.




```
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"
```

Global Array Examples

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

for i=0:1:100 do   ; store values at all node levels
. set ^mat(i)=i
. for j=0:1:100 do
.. set ^mat(i,j)=j
.. for k=0:1:100 do
... set ^mat1(i,j,k)=k

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

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
```

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 in any Mumps expression in any Mumps command and 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).*

Postconditionals

A post-conditional is an expression immediately following a command. If the expression is true, the command is executed. If the expression is false, the command is skipped and execution advances to the next command which may be on the same line or the next.

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

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

The **set** command is executed only if *a equals b.*

Postconditionals

Postconditionals are used to exit a single line loop where an **if** command would not work (the **for** means loop with *i* beginning at 1, incrementing by 1 and terminating when *i* greater than 100):

```
for i=1:1:100 quit:'$data(^a(i))  write ^a(i)
```

```
for i=1:1:100 if '$data(^a(i)) quit else  write ^a(i),!
```

The **if** command will skip the entire remainder of the line if the expression is false. The **else** command is never executed! Nothing is ever written.

Why?

if *$data(^a(i))* is true (data exists), the remainder of the line is not executed. If false, the **quit** is executed.

In the first example, if the postconditional is false, execution continues on the same line. If true, the loop terminates.

Operators

```
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
```

Operators

String relational operators

```
=    equal
[    contains - left operand contains right
]    follows  - left operand follows right
?    pattern
'?   not pattern
'=   not equal
'[   not contains
']   not follows
]]   Sorts after
']]  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&(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 value to its right to be executed.

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

Commands

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

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

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

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

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

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 no argument and is not the final command on a line, there must be at least two blanks after it or after its post-conditional, if present.

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 one blank before the next command word.

A semi-colon causes the remainder of the line to be interpreted as a comment. The semi-colon may be in column one or anywhere a command word is permitted.

Non-Standard Line Syntax Rules

GPL 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. If the last command on a line takes no argument, there must be at least two blanks after the command and its post-conditional, if present, and any double-slash (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 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 halt
                           ; postconditional
```

Blocks

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

Blocks are entered by the argumentless form of the **do** command (thus requiring two blanks between the **do** and the next command, if any). The lines following a **do** command belong to the **do** if they contain an incremented level of dots. The block ends when the number of dots declines to an earlier level.

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

Blocks

```
set a=1
if a=1 do
. write "a is 1",!
. set a=a*3
else  do
. write "a is not 1",!
. set a=a*4
write "a is ",a,!
```

writes *a is 1* and *a is 3*

Blocks and $Test

$test is a system variable which indicates if certain operations succeeded (true) or failed (false). An **if** command sets **$test**. The value in **$test** determines if an **else** command will execute (it does if **$test** is false):

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

Writes: 0 10
        1 10.
```

$test is restored when exiting a deeper block.

Quit

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

- A current single line scope **for** command to terminate, or
- A subroutine to return, or
- A block to be exited with execution resuming on the line containing the invoking **do** command.

Quit

```
; read and write until no more input ($test is 0).
  set f=0
  for  do  quit:f=1  ; this quit, when executed, terminates the loop
  . read a
  . if '$test set f=1 quit ; this quit returns to the do
  . write a,!

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

; non-standard use of break (GPL Mumps)
  for  do
  . read a
  . if '$test break ; exits the loop and the block
  . write a,!
```

Quit

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

; inner loop quits if an element of array b has the value 99 but
; 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

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

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

- A current single line scope **for** command to terminate, or
- A subroutine to return, or
- A block to be exited with execution resuming on the line containing the invoking **do** command.

Break

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

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

Close Command

Closes and disconnects one or more I/O units. May be implementation defined. All data is written to output files and buffers are released. No further I/O may take place to closed unit numbers until a successful new **open** command has been issued on the unit number.

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

Do Command

Executes a dependent block, or a labeled block of code either local or remote.

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

    do abc        ; executes code block beginning at label abc
    ...
abc    set x=1
       write x
       quit       ; returns to invoking do
-----------------------------------------------

   do ^abc.mps    ; invokes code block in file abc.mps
-----------------------------------------------

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

Else Command

The remainder of the line is executed if **$test** is false (0). **$**test is a system variable which is set by several commands to indicate success. 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

The **for** command can be iterative with the general 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

The for command 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

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 *do 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

Note: two blanks after **for** and **do**

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

writes 1 through 6

For with Quit

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

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

Halt Command

Terminate a program.

```
halt
```

Any code remaining on the line or in the program is not executed.

Hang Command

Pause the program for a fixed number of seconds.

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

If Command

```
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

Keyword **if** followed by an expression. If expression is true, remainder of line executed. If false, next line executed.

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

if sets **$test.** If the expression is true, **$test** is 1, 0 otherwise.

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 **$test** is false (0). An **else** requires no preceding **if** command. An **else** command following an **if** command on the same line will not normally execute unless an intervening command between the **if** and **else** changed **$test** to false.

Job Command

Creates an independent process. Implementation defined.

Kill Command

Kills (deletes) local and global variables.

```
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

Locks for exclusive access a global array node and ist 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. Consult documentation. See also transaction processing.

Merge Command

Copies on 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

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

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

; the previous values and a and b are restored.
```

Open, Use and Unit Numbers

Format of open command implementation dependent. In GPL Mumps unit 5 is always open for both input and output. Unit 5 is stdin and stdout

```
open 1:"aaa.dat,old" ; old means file exists
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 input/output
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",!
```

Input/Output Control Codes

The **write** command has the following basic format controls:

! - 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. It may include a prompt. 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 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 assignment statement.

```
set a=10,b=20,c=30
```

Transaction Commands

The following commands may or may not be implemented. They are intended to make transaction processing possible. Check implementation documentation.

- TCommit
- TREstart
- TROllback
- TSTART

Use Command

Select an I/O unit. Implementations may vary. At any given time, one I/O unit is in effect. All **read** and **write** operations default to that unit. You can select a different unit with the **use** command:

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

View Command

Implementation defined

Write command

```
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

Execute strings as 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              ; xecute input
```

Z... Commands

Implementation defined.

Navigating Arrays

Global (and local) arrays are navigated by means of the **$data()** and **$order()** functions. The first of these determines if a node exists, if it has data and if it has descendants. The second permits you to move from one sibling to another at a given level of a global array tree.

The function **$data()** returns a 0 if the array reference passed as a parameter to it 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.

Navigating Arrays

$order(), returns the next ascending (or descending) value of the last index of the global array reference passed to it as an argument.

By default, indices are presented 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 of -1 is given) if the value of the last index of the array reference passed to it is the empty string. It returns an empty string when there are no more values (nodes).

Navigating Arrays

```
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 (cont'd)

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

Indirection is one of the more powerful and also dangerous features of the language. With indirection, strings created by your program, read from a file, or loaded from a database can be interpretively evaluated and executed at runtime.

Indirection occurs at two levels. One is by means of the unary indirection operator (@) which causes the string expression to its right to be executed as a code expression. The other form is the **xecute** command which executes its string expression argument as command level text.

Indirection

```
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

Originally, subroutines were ordinary local blocks of code in the current routine or files of code. They are invoked by the **do** command. There were no parameters or return values. The full symbol table of variables is accessible any changes to a variable in a subroutine block would be effective upon return. This was similar to the early BASIC GOSUB implementation.

Later versions of Mumps added parameters and return values as well as call be name (subroutine can alter the calling routine's variable) and call be value (subroutine cannot alter calling program's variables). The later changes also permitted the programmer to create variables local to the subroutine (**new** command) which would be deleted upon exit. However, in most cases, the full symbol table of variables, is accessible to a subroutine.

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

Subroutines

```
do lab1                  ; local label code block
do ^file1.mps            ; file containing program

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

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

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

Subroutines

```
;       original subroutine style of invocation

        set i=100
        write i,!   ; writes 100
        do block1
        write i,!   ; writes 200
        halt

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

Subroutines

```
; subroutine creates a variable which is not
; destroyed on exit

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


two
    set x=99
    quit
```

Subroutines

```
; similar to original style but subroutine creates
; a new copy of x which is deleted upon return.

 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

```
; call be value example
; parameter variable d only exists in subroutine three
; any changes to d are lost on exit

 do three(101)
 write "expect 0 ->  ",$d(d),!  ; d only exists in the subroutine

three(d)
    write "expect 101 -> ",d,!
    quit
```

Subroutines

```
; call by name example
; modification of z in the subroutine changes x
; in the caller

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

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

Subroutines

```
; using new command
; subroutine one creates x and subroutine two uses it.
; it 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
```

Subroutines

```
; subroutine as a function with return value
; i is not changed in the subroutine

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

sub(i)

 set i=i*5
 quit i
```

Functions

Mumps has many builtin functions and system 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

$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

$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

$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

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

Uppercase characters indicate abbreviations.

Functions - $Ascii

$Ascii(arg)

```
$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
```

Functions - $Char

$Char(nbr)

```
$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
```

Functions - $Data

$Data(var)

$data returns an integer which indicates whether the variable argument is defined. The value returned is 0 if vn is undefined, 1 if vn is defined and has no associated array descendants; 10 if vn is defined but has no associated value (but does have descendants); and 11 is vn is defined and has descendants. The argument vn 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(e1,i2) or $Extract(e1,i2,i3)

$Extract(e1,i2) $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(e1,e2) or $Find(e1,e2,i3)

$Find(e1,e2) $Find(e1,e2,i3)

$Find() searches the first argument for an occurrence of the second argument.

If one is found, the value 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 1.

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])

```
$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
```

Based on local currency flags.

$Get ()

$Get(var)

Gets current value of a variable or a default value if undefined.

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

$Justify(e1,i2) or $Justify(e1,i2,i3)

$Justify() right justifies the first argument in a string field whose length is given by the second argument.

In the two operand form, the first argument is interpreted as a string.

In the three argument form, the first argument is right justified in a field whose length is given by the second argument with i3 decimal places.

The three argument form imposes a numeric interpretation upon the first argument.

```
$justify(39,3)      YIELDS " 39"
$justify("TEST",7) YIELDS " TEST"
$justify(39,4,1)    YIELDS "39.0"
```

$Length()

$Length(exp)

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

$NAme()

$NAme()

Give a string with all or part of an array filled in.

```
set x=10,y=20,z=30
$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")
```

abc() need not exist

Functions - $Order()

$Order(vn[,d])

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

The default traversal is in key ascending order 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. In GPL Mumps, numeric indices are retrieved in ASCII collating sequence order. Other systems may retrieve subscripts in numeric order. Check 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  do  quit:i=""
. set i=$order(^a(i),1)
. if i="" quit
. write i,!

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

$Piece(e1,e2,i3) or $Piece(e1,e2,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 i3th minus one and i3th occurrence of the second argument.

In the four argument form, the string returned is that substring of the first argument delimited by the i3th minus one instance of the second argument and the i4th instance of the second argument.

If only two arguments are given, i3 is assumed to be 1.

```
$piece("A.BX.Y",".",2)     yields "BX"
$piece("A.BX.Y",".",1)     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(e1)

$QLength() returns the number of subscripts in the variable name.

```
set i=1,j=2,k=3
set b(1)=99
write $qlength(^a(i,j,k)),!
write $qlength(^a(b(1),2)),!
write $qlength(^a),!
```

writes 3, 2 and 0

$QSubscript(e1,e2)

The **$QSubscript()** function returns a portion of the array reference given by e1. If the second argument is -1, the environment is returned (if defined), if 0, the name of the global array is returned.

For values greater than 0. the value of the associated subscript.

If a value exceeds the number of indices, an empty string is returned. Note: the variables or values of the subscripts must be valid.

$QSubscript() Examples

```
set i=1,j=2,k=3,m="k"
write $qsubscript(^a(i,j,k),-1),!
write $qsubscript(^a(i,j,k),0),!
write $qsubscript(^a(i,j,k),1),!
write $qsubscript(^a(i,j,k),2),!
write $qsubscript(^a(i,j,@m),3),!

writes ^a, 1, 2, and 3 respectively.
```

$QUery(e1)

The **$QUery()** function returns the next array element in the array space.

The first argument to **$query()** is a string representation of a global or local array. The value returned is the next ascending entry in the array space.

An empty string is returned when there are no more array references to return.

$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,!
```

$Random()

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

$Reverse()

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

$Select()

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

At first true expression, value after the colon returned.

$Text()

Assume 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()

```
set x="arma virumque cano"

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

$View

Implementation defined

$Z....

Implementation defined