A linguistic comparison of MUMPS and COBOL

by THOMAS MUNNECKE
Veterans Administration Hospital
Loma Linda, California

"I speak Spanish to God, Italian to women, French to men, and German to my horse." *Charles V of France*

COMPUTERS AND LANGUAGES

There are endless discussions in data processing circles about which computer language is best. Not surprisingly, the arguments generally boil down to each participant saying: "The language I know is best." These dogmatic beliefs often lead to vigorous debates among programmers who use different languages.

Languages have a deep relationship to the thought patterns of their users. If a programmer can't easily say something in a computer language, he is not inclined to think it, either. When programmers of the different languages meet, they are projecting their thought patterns into each others' language. Finding that the language does not express these thought patterns as richly as their native language, they often judge the other language as inferior.

Languages and their user communities tend to grow together. A user community cannot be expected to change its language unless it sees a new set of needs not met by its current language. Contrary to Charles V's fluency, it would be difficult to convince a German that he should learn Italian to speak to women. However, he could be convinced to learn the language of mathematics when he realizes that his spoken language is not sufficient for his mathematical needs.

MUMPS was created as a computer language in response to a new group of needs. MUMPS was designed to be a simple, small-computer-oriented language dedicated to a specific task. It turned against the trend toward disintegrative languages which grew out of early batch processing techniques. It pursued a new dimension of user/computer interaction. A dimension seldom seen, much less appreciated, by typical COBOL programmers.

Today, there is a crushing new group of needs which modern information systems must face. People costs to run computer systems far exceed computer costs. In order to recognize this linguistically, perhaps we should rename "computer systems" to "people systems." "Computer languages" should become "people languages." These terms more accurately reflect the needs of modern information systems. The computer is merely a medium by which an organization achieves its goals.

Viewing the computer as a medium is an effective way of reflecting these changing needs. The computer would then be in the same general category as books, magazines, television or telephone.

People might then view computer programmers turned computer scientists in the same light as a television cameraman who calls himself a television scientist or a printer who calls himself a book scientist.

This new group of needs can only be solved by the people who brought them about—the users of the information system. They cannot expect "computer scientists" to solve their problems any more than they can expect a printer to write a book for them.

The role of the computer must be to linguistically support the user in his own terms, adapting to his needs, and being as forgiving and friendly as possible. This, of course, is a flagrant violation of oldtime wisdom, where one could not waste computer time on such frills.

MUMPS takes this role of user adaptability seriously. The natural logic of the language encourages programmers to turn control over to users with simple, yet powerful commands. MUMPS users tend to show an almost reverent attitude to their systems.

Conversely, COBOL was designed in an era when computers were expensive, programmers were cheap, and only science fiction buffs dreamed of small computers. The natural logic of COBOL is oriented towards batch processing of fixed records. As people struggled to make COBOL adapt to the needs of on-line systems, they added features* such as data base management systems, data communications monitors, message formatting monitors, and the like. Rather than adding flexibility to COBOL, all of these disintegrative appendages have stifled the language.

MUMPS, perhaps, can best be understood by the appendages which it lacks. This lack of features is MUMPS' great-

* The term "feature" has sometimes been defined as a "design flaw which marketing" has noticed. For example, a computer manufacturer once announced a distributed processor which had the "security feature" that it could not be programmed locally. Only cross-compilations downloaded from the host processor were allowed. Cynics who felt that this feature was really a cover up for not having local software were vindicated some time later when the manufacturer announced a new feature—local programming ability.

est strength, not a weakness, as the old schools would have it. All of the following functions of these appendages are integrated into MUMPS' unique symbolic structure:

Assembler Language
Compiler
Data Base Management System
Sort/Merge Utilities
Job Control Language
Linkage Editor
Debugger
Data Communications Monitor
Core Image Dumps
Absolute Addresses
Type Definitions
Dimension Statements
Message Format Processor.

MUMPS is an entire data management system, built within a single linguistic framework. All of the utilities, control blocks, and organizational paraphernalia which are considered "features" of COBOL-based systems are simply not needed with MUMPS.

In order to facilitate comparison, IBM's data base management system, IMS 360 (Information Management System) has been selected as a typical add-on to COBOL for data management applications.

This elegant simplicity of MUMPS is generally not appreciated by newcomers examining the MUMPS syntax. For example, the up-arrow is the only indicator in the MUMPS language that the program is working with a data-base record instead of a local array variable. A reader who was expecting a long list of data-base manager subroutine calls will be disappointed. He should not, however, condemn the language for successfully eliminating those appendages of archaic languages.

INTRODUCTION TO MUMPS[1]

MUMPS, the Massachusetts General Hospital Multi-Programming System, is a high-level interpretive programming language and data-management system. It is particularly suited for interactive applications which require a large, shared dynamic data-base and the efficient manipulation of textual data.

The development of MUMPS began at the Laboratory of Computer Science, Massachusetts General Hospital, Boston, Massachusetts, in 1966. Previous research in medical information systems had encountered frustration and dissatisfaction with the current state of the technology. The experiments during 1960 to 1965 with assembly language systems were usually unsuccessful due to long development time and excessive turnaround times for even the most simple program modifications. For these reasons, the staff of the Laboratory of Computer Science set out to design an efficient time-sharing system for clinical data management.[2,3] The characteristics of this system were patterned after JOSS, a high-level, interpretive language developed at the RAND Corporation in 1964. Experience was also drawn from descendants of JOSS such as TELCOMP and STRING-COMP developed by Bolt, Beranek and Newman, and FILECOMP specified by General Electric for the MEDINET system.

The goal of the MUMPS system was to combine a simple yet powerful high-level language with an easy-to-use database handling system. The MUMPS Language was designed to be easy to learn, with simple methods of program creation, modification and debugging. Language capabilities for the handling of variable length character strings and multi-terminal I/O were also required. A sparse hierarchical data-base system was developed as an integral part of MUMPS. The hierarchical structure was determined to be the most appropriate method of handling the complex of demographic data, diagnosis, laboratory results and other data required for clinical data management. A data-base handler was designed to facilitate access to the hierarchical structure from the MUMPS language. Basic features were developed to implement symbolic update and retrieval functions. The design emphasized the support of a dynamic data-base, subject to frequent updates interspersed with on-line queries. The MUMPS programmer could freely design both the content and structure of data to best fit his application. Finally, the system was implemented with a compact, time-sharing executive to make efficient use of all resources in a mini-computer environment.

The appropriateness of these design goals is now well documented. The MUMPS system has found a significant place in both medical and non-medical environments. Since the original implementation at the Laboratory of Computer Science, at least seven other dialects and variations of MUMPS have been developed. In order to rejoin these divergent views and promote program interchange, the U.S. Department of Health, Education and Welfare, and the National Bureau of Standards initiated development of Standard MUMPS. This standard specification, along with other MUMPS documents for language teaching and translation was accepted by the American National Standards Institute in 1977.[4]

MUMPS is more than a programming language. It is a linguistically integrated data management system combining with a single syntax what other operating systems might call: (a) an application programming language; (b) a job control language; (c) a linkage editor; (d) a data-base management system; and (e) a data communications monitor.

The MUMPS storage hierarchy

MUMPS goes beyond traditional data management by allowing records (nodes) to be interconnected hierarchically. Thus, a node can be the child of another node and the parent of any number of nodes. Networks are not allowed: each node can have only one immediate parent (though there can be any number of generations in the structure), and no node can have a parent as a descendant. It is also possible to have nodes which serve only as connectors, i.e., which contain no data. These are often called *pointers* though they should not be confused with the traditional use of this term. The set of subscripts or keys which define a node is actually the path from the top of the hierarchy to the node. Thus, there is a single key associated with the descent from a parent to a child. MUMPS provides capabilities for addressing a node, for the purpose of storage or retrieval by the complete sequence of keys, or by a portion of the sequence. When a portion is used, the missing keys are supplied by the system based on the most recent reference to the database. It is also possible to test whether or not an arbitrary node exists and whether or not it has any data. Also, there is a means of sequencing among a set of siblings, i.e., given an arbitrary position within the hierarchy of finding the next numerically higher single key. This technique cannot be used to cross from one set of siblings to another. In general, there are capabilities for moving from parent to child and from sibling to sibling, but the only way to move back up is to begin again from the top.

This scheme of node interconnection is very useful because many real world systems can be represented by a hierarchy or something simpler (e.g., a hierarchy includes a simple indexed organization as a subset). Thus, a complex entity may be represented as being constructed of subentities, each of which is in turn sub-divided. In addition, there is frequently a need for network structures. These relationships can be created by the applications programmer. For example, although MUMPS does not itself provide inverted files, MUMPS programs construct such files by storing logical references to other nodes. For example, in an inventory control system a parts file might have part-number as a key. A file of suppliers could logically point into the parts number file by means of the part numbers.

People often ask of MUMPS systems: "Why does it not support other languages?" "Why does it not compile its code?" These are a sampling of hundreds of similar questions as to why MUMPS does not support the features of the language and operating systems to which they are accustomed. These questions are of the variety: "When did you stop beating your wife?"

MUMPS' only retort to these questions is: Why are all these features needed? Are the 12 access methods used by a typical COBOL/IMS system a strength or a weakness? Is the linguistic distintegration characteristic of almost all "modern" operating systems solvable by adding more fragmentary features? Is the increasingly complex combine of pre-compiled object modules, control blocks, and linkage editors really suited to today's online environment?

Standards

A standard is only as strong as its weakest linguistic link. A standard which ignores major linguistic structures such as terminal communication, data-bases, and the like necessarily will force users into an integration crunch and a reversion back to disintegrative designs.

The American National Standards Institute MUMPS standard is a linguistically strong standard. Programs and data-bases written in standard MUMPS are assured of portability because MUMPS requires no linguistic support from all of the peripheral languages common to older operating systems.

Oddly enough, this linguistic independence is often the most vocally criticized aspect of MUMPS. People reviewing MUMPS, after noting its lack of features, go on to criticize its "incompatibility" with current systems. One reviewer's "incompatibility" is another's "linguistic purity." The fact that MUMPS' designers refused to disintegrate their linguistic realms to archaic operating system features should not be held against them.

Access methods

The list below compares the access methods used by MUMPS and COBOL using IBM's Information Management System (IMS).

COBOL/IMS	MUMPS
QSAM	GLOBALS
BSAM	
QISAM	
BISAM	
BPAM	
VSAM	
BTAM	
HSAM	
HISAM	
HIDAM	
HDAM	
OSAM	

While this incomplete list of 12 access methods may look impressive, it raises the question: Are all these really necessary? To the MUMPS programmer, of course, the answer is no. Even the term "access method" is foreign to him.

Pointers

Today's disk storage technology requires internal pointers for efficient data management. The following list shows the difference between COBOL/IMS and MUMPS.

COBOL/IMS	MUMPS
Physical Parent	$NEXT
Physical Child	$DATA
Hierarchical Forward	
Hierarchical Backward	
Physical Twin Backward	
Logical Twin Forward	
Logical Twin Backward	
Physical Child Last	

The differences between these two columns run far deeper than is apparent. While the MUMPS program is able to dynamically structure its search, based on the content of the data it finds as it works its way through the data-base, the COBOL/IMS pointer system is rigidly defined in pre-compiled control blocks which may be changed with only the greatest caution.

MUMPS' richness in content-oriented data structuring is possibly best illustrated by the fact that MUMPS can represent a data value of "nothing." Whereas COBOL would require a bogus value, for example 999 or spaces, MUMPS simply recognizes a null data value. This is roughly equivalent to the discovery of zero in algebra.

Languages

People often say "We are a COBOL shop." However, due to the inherently weak nature of COBOL and the languages which have sprouted up around it, *many* languages are used within the COBOL environment.

COBOL/IMS LANGUAGES

Language	Pages of Documentation
COBOL	300
Data Language/1	100
Job Control Language	200
Linkage Editor	100
Message Format Services	100
System Definition MACROS	300
Assembler Language	200
MACRO Assembler	100
Program Specification Blocks	150
Data-Base Definition Blocks	150
Total Pages	1700

Each of the languages listed above has its own linguistic domain, reference language, and documentation. This has a profound effect on the overall operation of the computer support staff. Specialists have arisen, spreading the responsibilities of a given system over a multitude of people, such as systems programmers, data-base administrators, network administrators, and control block librarians, in addition to the traditional programmers and systems analysts. Listings of code in each of these languages is thus spread out over each of these specialists, and typically jealously guarded. Thus, a programmer searching for the cause of an error may spend a good portion of his time searching for specialists and/or their listings.

Error detection and correction

Try as we may, errors will occur. MUMPS, being an interactive system, will display the error with an explanation directly in the MUMPS language. The programmer may examine variables, modify the program, and resume processing directly from the keyboard, all using MUMPS language. Thus, all communication is accomplished in one language, at a terminal, as soon as the error occurs. This is not the case with the COBOL/IMS environment. An error begins with a hexadecimal dump (typically, 25–50 pages long), which the programmer sees hours or days after the error occurs. He must thread these numbers through a maze of languages, specialists, and listings. Following is a highly polished translation of one error which might occur in a COBOL/IMS environment:

COBOL/IMS error detection:

"An Assembler Language *Dump* shows that the *COBOL Return Code* of the *PCB* of the *PSB* defined in the *PARM* of the *JCL EXEC* card (or specified in the *IMS SYSTEM*) generated by the *PSBGEN* utility contains a *PCB MACRO* with *SENSEG MACRO* parameter *PROCOPT* incompatible with *DL/1* call *FUNCTION* parameter."

This particular error traversed six languages, on four different listings, requiring working knowledge of 1000 pages of documentation. Furthermore, the error was presented to the programmer after the fact, perhaps after irreversible conditions had changed things.

With the functions of the data-base management system, job control language, control blocks, utilities, assemblers, compilers, and communications monitors stripped away, one arrives at a comparison of the entire COBOL language and a portion of MUMPS.

CASE COMPARISON-PAYROLL PROGRAM

The author once translated a COBOL program into MUMPS. The COBOL program was part of a payroll system which received a batch of time and attendance records and computed the gross and net pays, various leave balances, and the like. The MUMPS version replaced the batch system with on-line data entry and validation, with immediate computations. Thus, the MUMPS version had more work to do.

ITEM	COBOL	MUMPS	PERCENT
Lines of Code	3600	300	8
"IF" Statements	460	89	19
"GOTO" Statements	650	43	6
Total Program Size	120K	9K	8

The MUMPS version required approximately 8% of the number of lines of code, 19% of the number of conditional checks (even with the added validity checks), 6% of the program branches, and 8% of the run-time memory.

Execution time on a $100,000 MUMPS minicomputer was approximately twice as long as the several million dollar COBOL/IBM 370/158 computer. Exact programming times were hard to estimate, but three weeks were spent on the MUMPS version, while the original COBOL version took an estimated six to nine months.

CASE COMPARISON—MESSAGE DISPLAY

To illustrate these differences, a portion of a COBOL program was selected which writes out the message: "Affidavit

XXX processed, Precinct is YYY." The MUMPS version of this simple message display program is: WRITE !, "AFFIDAVIT ", AFFNO," PROCESSED. PRECINCT IS ", PREC

The COBOL version illustrated below requires the message to be formatted in the DATA division. (Note that the 'W' is actually a special character, which must be multipunched on a keypunch. Thus, the keypunch, printer, and computer all have different understandings of the same character):

```
Data division:
03 FILLER   PICTURE X(10) VALUE 'AFFIDAVIT'.
03 MSG-AFF-NO   PICTURE X(7) VALUE SPACES.
03 FILLER   PICTURE X(10) VALUE 'PROCESSED'.
03 FILLER   PICTURE X(13) VALUE 'PRECINCT
                IS'.
03 MSG-PREC-NO   PICTURE X(5) VALUE ''.
03 FOB   PICTURE X VALUE 'W'.
```

Then the message must be transmitted in the procedure division.

```
Procedure division:
000-ENTRY.
  MOVE H-PREC-NO TO MSG-PREC-NO.
  MOVE H-AFF-NO TO MSG-AFF-NO.
  MOVE CMPL-MSG TO OUT-SEG-1.
TERM-TRAN.
  CALL 'TELECALL' USING DECB-ADDR,
    TPTRNSMT, TP-SW.
RETURN-TO-VRNEWAFF.
```

The MUMPS version used 6% of the lines of code, and 9.6% of the number of characters. The COBOL version made six explicit declarations of the lengths of fields involved. MUMPS made none.

Comparisons:

ITEM	COBOL	MUMPS
Lines of Code	16	1
Characters	500	48
Number of Bindings	6	0

DATA INDEPENDENCE

Someone from the disintegrative school might defend it at this point by saying: "But what about data independence?** The traditional file structures provide for data-independent programs, through well-defined linkages."

If one examines the situation carefully, one sees that data independence is a mythical construct of the disintegrative school—data dependence is merely being transferred to yet another language or languages. This process is somewhat akin to a doctor "curing" a patient by erasing his symptoms from the medical record.

** Data independence[5] is defined to be the "immunity of applications to change in storage and access strategy."

Mr. C. J. Date[5] discussed the data independence of IMS. If one examines the darker side of IMS's data independence (i.e., its data dependence on control blocks), things are slightly different. For example, in order to add a single byte to a key field in a COBOL/IMS system, the following procedures must be followed:

1) Change the data-base definition block
2) Change the program specification block
3) Regenerate the accumulated control block
4) Change any message output format, message input format, device input format, or device output format block referencing the field.
5) Change the data division of each COBOL program which references the field. Furthermore, the procedure sections of each program must be scanned for move statements which overtly or covertly reference the field. Once associated fields have been identified, they too must be scanned for reference to yet other fields. COBOL can covertly reference fields through redefining, corresponding moves, assembler language subroutines, or parameter passing.
6) Unload the data-bases under the old data definition.
7) Reload the data-bases under the new data definition with a program which shifts the data to its new format. Depending on the size of the data-base, the unload/reload process can take from a few minutes to several days. The data-bases are not accessible to terminals during a major portion of this time.
8) Since the changes are so pervasive, prudence dictates that the control blocks, application programs, and databases be tested on a duplicate "test" system. Thus, all of the above steps must be carefully sequenced through twice.

Thus, the COBOL/IMS concept of data independence can trigger off a complicated sequence of control block changes, recompilations, job control language changes, and significant data-base down time for the simple process of adding a single byte to a field. The operations staff must use several macro languages, COBOL, job control languages, linkage editor, and manual procedures to accomplish this task.

Many attempts to correct this complicated sequence have been made, including adding *another* linguistic entity such as a master data dictionary. However, this can only serve to further disintegrate linguistic control. As a new language is added it imposes its own (weak) data definition structure, reference language, source language control, etc.

One is tempted to ask, "Why is adding a single byte to a field such a major undertaking for such a sophisticated computer system? Why is every linguistic domain so dependent on exact field length specifications?" The answer is that the disintegrative approaches are built up from compiled logic which uses absolute addresses (or absolute offsets). The languages lose control of the data at the moment of compilation.

MUMPS, on the other hand, makes no such linguistic distinctions. Fields are treated dynamically according to whatever data are found in them. Data-base structures grow and shrink according to whatever data is stored in them. Reorganization is seldom necessary due to internal techniques of balanced multiway trees.[6] All data references are symbolic; if a field does not exist in a particular instance, it takes no space. There are no data definitions, procedure-scanning, absolute addresses, REDEFINES, assembler language subroutines, control blocks, or data set definitions to worry about—they simply do not exist. The only changes a MUMPS programmer may need to make to add a byte to a field are:

1.) If an explicit length reference is made to the field, it will have to be changed to the new length. For example, IF $LENGTH (INPUT) > 6 WRITE "TOO LONG" would have to have the "6" changed to a "7".
2.) If the field is printed on a pre-printed form, the output routine may have to be changed.

MISCELLANEOUS OBSERVATIONS OF COBOL AND MUMPS

1. To a MUMPS programmer, COBOL appears to be a linguistic flatland in which only the simplest data structures may be expressed. Problems which he dismisses with a simple statement in MUMPS would be pages of code in COBOL.
2. COBOL's rigid structure is its most prominent characteristic. MUMPS is known for its flexible data and program structures. For example, if a COBOL program encounters a 5 digit number to be printed in a 4 digit field, it will change the data to meet the format. MUMPS would rather print the right data in the wrong format than print the wrong data in the right format. In all of the structure/content design tradeoffs, MUMPS stresses content, while COBOL stresses structure.
3. COBOL makes a very strict distinction between "program" and "data." These distinctions are not necessarily made in MUMPS. A MUMPS program could execute a data-base, or a program could be treated as data. This allows MUMPS to be used as an implementation language for higher level languages or systems. It also allows for all of the MUMPS operating system utilities to be written in MUMPS itself, rather than resorting to assembler languages, linkage editors and the like.
4. COBOL is usually compiled, whereas MUMPS is usually interpreted. The COBOL language disappears at execution time. It assumes that the programmer has accounted for all eventualities *before* the program was compiled. MUMPS, on the other hand, is free to make use of the interpreter during the execution of the program.
5. MUMPS takes the "small is beautiful" approach to computing. Originally designed for minicomputers, it exploits the dedicated nature of small computers. It makes heavy use of the cheapest resource (central processor time), and minimizes the most expensive resource (people time). MUMPS systems generally grow by adding more systems, rather than larger ones. COBOL, on the other hand, grew up in the "bigger is better" school. Manufacturers stressed the "economies of scale" of large computers, saying that a larger computer would work more cheaply per unit of work. These economies have clearly turned around with today's microelectronic technology.

   COBOL users and large scale computer manufacturers, fearing loss of control, have often responded by scaling problems up to the point where they can be solved only by large-scale computing equipment. MUMPS users, on the other hand, tend to scale problems down to smaller and smaller computers.
6. The author has a theory that the response time of an interactive computer system increases exponentially with the cost of a computer. This is due to the fact that a computer must be idling along at 30-50% capacity in order to handle unexpected interactive loads. Thus, the cost of good response time is proportional to the cost of "wasting" computer time in reserve for the unpredictable needs of an online system. The owner of a $500 Radio Shack computer does not hesitate to "waste" computer time to serve his needs, but his techniques would bring shudders to the manager of a large scale computer.
7. There is a controversy in the data processing field titled "Superprogrammers versus Mongolian Hordes." MUMPS supports the "superprogrammer" philosophy. Individuals, or small teams of MUMPS programmers, are capable of producing what large teams of COBOL programmers can do. Few "superprogrammers" are content to remain COBOL programmers. Their talents are frustrated by COBOL's awkwardness, inflexibility, and slow development cycles. Good COBOL programmers tend to be promoted to higher paying positions in the COBOL organizational hierarchy, a clear case of the Peter Principle. In contrast, MUMPS programmers can draw higher salaries due to their higher productivity, and happily remain MUMPS programmers.
8. COBOL programmers tend to exhibit great concern about computer efficiency with a corresponding lack of concern about the efficiency of the users of the system. I have labelled this characteristic "cyclephobia"—an irrational fear of wasting computer cycles. Cyclephobes tend to see problems in light of the primitive operations expressable in COBOL. MUMPS programmers, on the other hand, have a much healthier attitude toward computer/user efficiency tradeoffs. This is partly because they use an inherently more friendly computer—the small computer, and partly because MUMPS naturally directs the programmer to "friendly," responsive computer interactions.

REFERENCES

1. Munnecke, T. H., R. F. Walters, J. Bowie, C. B. Lazarus and D. A. Blidger, "Mumps: Characteristics and Comparison with Other Programming Systems," *Medical Informatics*, Vol. 2, No. 3, pp. 173–196, 1977.

2. Greenes, R. A., A. N. Papalardo, C. W. Marble and G. O. Barnett, "Design and Implementation of a Clinical Data Management System," *Computers in Biomedical Research* **2**, pp. 469–485, 1969.
3. Bowie, J. and G. O. Barnett, "MUMPS—An Economical and Efficient Time-Sharing System for Information Management," *Computer Programs in Biomedicine*, 6, pp. 11–22, 1976.
4. American National Standards Institute, Inc., *American National Standard MUMPS Language Standard,* ANSI 11.1-1977, 1977.
5. Date, C. J., *An Introduction to Database Systems,* Addison Wesley, Reading, Massachusetts, 1975.
6. Knuth, P. E., *The Art of Computer Programming, Vol. 3. Sorting and Searching,* Addison-Wesley, Reading, Massachusetts 1973.