Designation: E 1460 – 92

An American National Standard

Standard Specification for Defining and Sharing Modular Health Knowledge Bases (Arden Syntax for Medical Logic Modules)[1]

This standard is issued under the fixed designation E 1460; the number immediately following the designation indicates the year of original adoption or, in the case of revision, the year of last revision. A number in parentheses indicates the year of last reapproval. A superscript epsilon (ε) indicates an editorial change since the last revision or reapproval.

1. Scope

1.1 This specification covers the sharing of computerized health knowledge bases among personnel, information systems, and institutions. The scope has been limited to those knowledge bases that can be represented as a set of discrete modules. Each module, referred to as a Medical Logic Module (MLM), contains sufficient knowledge to make a single decision. Contraindication alerts, management suggestions, data interpretations, treatment protocols, and diagnosis scores are examples of the health knowledge that can be represented using MLMs. Each MLM also contains management information to help maintain a knowledge base of MLMs and links to other sources of knowledge. Health personnel can create MLMs directly using this format, and the resulting MLMs can be used directly by an information system that conforms to this specification.

1.2 The major topics are found in the following sections.

[1] This specification is under the jurisdiction of ASTM Committee E-31 on Healthcare Informatics and is the direct responsibility of Subcommittee E31.15 on Medical Knowledge Representation.

Current edition approved Feb. 15, 1992. Published April 1992.

2. Referenced Documents

2.1 *ASTM Standards:*

E 1238 Specification for Transferring Clinical Laboratory Data Messages Between Independent Computer Systems[2]

E 1384 Guide for Content and Structure of the Computer-Based Patient Record[2]

2.2 *ISO Standards:*

ISO 8601 – 1988 Data Elements and Interchange Formats—Information Interchange (representation of dates and times)[3]

2.3 *ANSI Standards:*

ANSI X3.4 – 1986 Coded Character Sets—American National Standard Code for Information Interchange (7-bit ASCII)[4]

3. Terminology

3.1 *Definitions:*

3.1.1 *Medical Logic Module (MLM),* , *n*—an independent unit in a health knowledge base. Each MLM contains maintenance information, links to other sources of knowledge, and enough logic to make a single health decision.

3.2 *Definitions of Terms Specific to This Standard:*

3.2.1 *time,* , *n*—a point in absolute time. Also known as a timestamp, it includes both a date and a time-of-day.

3.2.2 *time-of-day,* , *n*—hours, minutes, seconds, and possibly, fractions of seconds past midnight.

3.2.3 *date,* , *n*—Gregorian year, month, and day.

3.2.4 *duration,* , *n*—a period of time (for example, **3 days**) that has no particular start or end point.

3.2.5 *institution,* , *n*—a health facility of any size that will provide automated decision support or quality assurance.

4. Significance and Use

4.1 Decision support systems have been used for health care successfully for many years, and several institutions have already assembled large knowledge bases. There are many conceptual similarities among these knowledge bases. Unfortunately, the syntax of each knowledge base is different. Since no one institution will ever define a complete health knowledge base, it will be necessary to share knowledge bases among institutions.

4.2 Many obstacles to sharing have been identified: disparate vocabularies, maintenance issues, regional differences, liability, royalties, syntactic differences, etc. This standard addresses one obstacle by defining a syntax for creating and sharing knowledge bases. In addition, the syntax facilitates addressing other obstacles by providing specific fields to enter maintenance information, assignment of clinical responsibility, links to the literature, and mappings between local vocabulary terms and terms in the knowledge base.

4.3 The range of health knowledge bases is large. This specification focuses on those knowledge bases that can be represented as a set of Medical Logic Modules (MLMs). Each MLM contains maintenance information, links to other sources of knowledge, and enough logic to make a single health decision. Knowledge bases that are composed of independent rules, formulae, or protocols are most amenable to being represented using MLMs.

4.4 This specification, which is an outcome of the Columbia-Presbyterian Medical Center 1989 Arden Homestead retreat on sharing health knowledge bases, is derived largely from HELP of LBF Hospital, Salt Lake City, UT (**1**)[5], and CARE, the language of the Regenstrief Medical Record System of the Regenstrief Institute for Health Care, Indianapolis, IN (**2**).

5. MLM Format

5.1 *File Format*—An MLM is a stream of text stored in an ASCII file (ANSI X3.4 – 1986). One or more MLMs may be placed in the same file. Within a file, an MLM begins with the marker **maintenance:** and ends with the marker **end:**.

[2] *Annual Book of ASTM Standards*, Vol 14.01.

[3] Available from ISO, 1 Rue de Varembe, Case Postale 56, CH 1211, Geneve, Switzerland.

[4] Available from American National Standards Institute, 11 W. 42nd St., 13th Floor, New York, NY 10036.

[5] The boldface numbers in parentheses refer to the list of references at the end of this standard.

5.2 *Character Set*—Within an MLM only the printable ASCII characters (ASCII 33 through and including 126), space (ASCII 32), carriage return (ASCII 13), line feed (ASCII 10), and horizontal tab (ASCII 9) may be used. The use of horizontal tab is discouraged because there is no agreement on how many spaces it represents. Other characters, like the bell and backspace, are not allowed within the MLM. (There is no limitation on the characters that may occur between MLMs within a file; for example, a form feed character may separate two MLMs even though it cannot occur within an MLM.)

5.3 *Line Break*—Lines are delimited by **line breaks**, which are any one of the following: a single carriage return, a single line feed, or a carriage return-line feed pair.

5.4 *White Space*—The space, carriage return, line feed, and horizontal tab are collectively referred to as **white space**.

5.5 *General Layout*—Annex A1 contains a complete description of MLMs expressed in BACKUS-NAUR Form. See Appendix X1 for MLM examples. (Planned editions and changes for future versions of this specification are listed in Appendix X2.) A typical MLM is arranged like this.

maintenance:
    slotname: slot-body;;
    slotname: slot-body;;
...
library:
    slotname: slot-body;;
...
knowledge:
    slotname: slot-body;;
...
end:

5.6 *Categories*—An MLM is composed of slots grouped into three categories: maintenance, library, and knowledge. A category is indicated by a category name followed immediately by a colon (that is, **maintenance:**, **library:**, and **knowledge:**). Category names need not be placed at the beginning of a line.

5.7 *Slots*—Within each category is a set of slots.

5.7.1 Each slot consists of a slot name, followed immediately by a colon (for example, **title:**), then followed by the slot body, and terminated with two adjacent semicolons (**;;**) which is referred to as **double semicolon**. The content of the slot body depends upon the slot, but it must not contain a double semicolon.

5.7.2 Each slot must be unique in the MLM, and categories and slots must follow the order in which they are listed in this standard. Some slots are required and others are optional.

5.8 *Slot Body Types*—These are the basic types of slot bodies:

5.8.1 *Textual Slots*—The textual slots contain arbitrary text (except for double semicolon, which ends the slot). As the MLM standard is augmented, slots that are currently considered to be textual may become coded or structured. An example of a textual slot is the title slot, which can contain arbitrary text.

5.8.2 *Textual List Slots*—Some slots contain textual lists. These are lists of arbitrary textual phrases separated by single semicolons (**;**). An example of a textual list slot is the keywords slot.

5.8.3 *Coded Slots*—Coded slots contain a simple coded entry like a number, a date, or a term from a predefined list. For example, the priority slot can only contain a number, and the validation slot can contain only the terms **production**, **research**, etc.

5.8.4 *Structured Slots*—Structured slots contain syntactically defined slot bodies. They are more complex than coded slots, and are further defined in Section 7. An example of this kind of slot is the logic slot.

5.9 *MLM Termination*—The end of the MLM is marked by the word **end** followed immediately by a colon (that is, **end:**).

5.10 *Case Insensitivity*—Category names, slot names, and the **end** terminator may be typed in uppercase (for example, **END**), lowercase (for example, **end**), or mixed case (for example, **eNd**).

6. Slot Descriptions

6.1 For each slot description, next to the slot name is an indication of whether the slot is textual, textual list, coded, or structured, and whether it is required or optional. Slots must appear in this order.

6.2 *Maintenance Category*—The maintenance category contains the slots that specify information unrelated to the health knowledge in the MLM. These slots are used for MLM knowledge base maintenance and change control.

6.2.1 *Title (textual, required)*—The title serves as a comment that describes briefly what the MLM does. For example,

title: Hepatitis B Surface Antigen in Pregnant Women;;

6.2.2 *Filename (coded, required)*—The MLM filename uniquely identifies an MLM within a single authoring institution. It is represented as a string of characters beginning with a letter and followed by letters, digits, and underscores (_). A filename may be 1 to 80 characters in length. Filenames are insensitive to case. The MLM filename is distinct from the name of the ASCII file which happens to hold one or more MLMs. For example,

filename: hepatitis_B_in_pregnancy;;

6.2.3 *Version (coded, required)*—The current version of the MLM is expressed as a fixed point number with two decimal places to the right of the decimal point. MLMs start at 1.00 and advance by .01 for small revisions and by 1 for large revisions. For example,

version: 1.00;;

6.2.4 *Institution (textual, required)*—The institution slot contains the name of the authoring institution. For example,

institution: Columbia University;;

6.2.5 *Author (textual list, required)*:

6.2.5.1 The author slot contains a list of the authors of the MLM, delimited by semicolons. The following format should be used: first name, middle name or initial, last name, comma, suffixes, and degrees.

6.2.5.2 An electronic mail address enclosed in parentheses may optionally follow each author's name. Internet addresses are assumed. Bitnet addresses should end in **.bitnet** and UUCP addresses should end in **.uucp**. For example,

author: John M. Smith, Jr., M.D. (jms@cuasdf.bitnet);;

6.2.6 *Specialist (textual, required)*—The domain specialist is the person in the institution responsible for validating and installing the MLM. This slot should always be present but

blank when transferring MLMs from one institution to another. It is the borrowing institution's responsibility to fill this slot and accept responsibility for the use of the MLM. The format is the same as for the author slot.

6.2.7 *Date (coded, required)*—The date of last revision of the MLM must be placed in this slot. Either a date or a time (that is, a point in absolute time composed of a date plus a time-of-day) can be used. The format is ISO extended format for **dates** and for **date-time combinations** with optional time zones (ISO 8601:1988 (E)). Dates are yyyy-mm-dd so that January 1, 1989 would be represented as 1989-01-01. Times are yyyy-mm-ddThh:mm:ss with optional fractional seconds and optional time zones. Thus, 1:30 pm on January 1, 1989 would be represented as 1989-01-01T13:30:00. For example,

date: 1989-01-01;;

6.2.8 *Validation (coded, required)*:

6.2.8.1 The validation slot specifies the validation status of the MLM. Use one of the following terms:

(a) production—approved for use in the clinical system,

(b) research—approved for use in a research study,

(c) testing—for debugging, default initial value, or

(d) expired—out of date, no longer in clinical use.

6.2.8.2 An example is:

validation: testing;;

6.2.8.3 —MLMs should never be shared with a validation status of **production**, since the domain specialist for the borrowing institution must set that validation status.

6.3 *Library Category*—The library category contains the slots pertinent to knowledge base maintenance that are related to the MLM's knowledge. These slots provide health personnel with predefined explanatory information and links to the health literature. They also facilitate searching through a knowledge base of MLMs.

6.3.1 *Purpose (textual, required)*—The purpose slot describes briefly why the MLM is being used. For example,

purpose: Screen for newborns who are at risk for developing hepatitis B;;

6.3.2 *Explanation (textual, required)*—The slot explains briefly in plain English how the MLM works. The explanation is shown to the health care provider when he or she asks why an MLM came to its decision. For example,

explanation: This woman has a positive hepatitis B surface antigen titer within the past year. Therefore her newborn is at risk for developing hepatitis B.;;

6.3.3 *Keywords (textual list, required)*—Keywords are descriptive words used for searching through modules. UMLS terms **(3)** are preferred but not mandatory. Terms are delimited by semicolons (commas are allowed within a keyword). For example,

keywords: hepatitis B; pregnancy;;

6.3.4 *Citations (textual, optional)*—Citations to the literature are entered in Vancouver style **(4)**. Citations may be numbered, serving as specific references. For example,

citations:
1. Steiner RW. Interpreting the fractional excretion of sodium. Am J Med 1984;77:699–702.
2. Goldman L, Cook EF, Brand DA, Lee TH, Rouan GW, Weisberg MC, et al. A computer protocol to predict myocardial infarction in emergency department patients with chest pain. N Engl J Med 1988;318(13):797–803.
;;

6.3.5 *Links (textual, optional)*—The links slot allows an institution to define links to other sources of information, such as an electronic textbook, teaching cases, or educational modules. The contents of this slot are institution-specific.

6.4 *Knowledge Category*—The knowledge category contains the slots that actually specify what the MLM does. These slots define the terms used in the MLM (data slot), the context in which the MLM should be evoked (evoke slot), the health condition to be tested (logic slot), and the action to take should the condition be true (action slot).

6.4.1 *Type (coded, required)*—The type slot specifies what slots are contained in the knowledge category. The only type that has been defined so far is **data-driven**, which implies that there are the following slots: data, priority, evoke, logic, action, and urgency. That is,

type: data-driven;;

6.4.2 *Data (structured, required)*—In the data slot, terms used locally in the MLM are mapped to entities within an institution. The actual phrasing of the mapping will depend upon the institution. The details of this slot are explained in Section 11.

6.4.3 *Priority (coded, optional)*—The priority is a number from 1 (low) to 99 (high) that specifies the relative order in which MLMs should be evoked should several of them simultaneously satisfy their evoke criteria. An institution may or may not choose to use a priority. The institution is responsible to maintain these numbers to avoid conflicts. A borrowing institution will need to adjust these numbers to suit its collection of MLMs. For example,

priority: 50;;

6.4.4 *Evoke (structured, required)*—The evoke slot contains the conditions under which the MLM becomes active. The details of this slot are explained in Section 13.

6.4.5 *Logic (structured, required)*—This slot contains the actual logic of the MLM. It generally tests some condition and then concludes **true** or **false**. The details of this slot are explained in Section 12.

6.4.6 *Action (structured, required)*—This slot contains the action produced when the logic slot concludes true. The details of this slot are explained in Section 9.17.1.

6.4.7 *Urgency (coded, optional)*—The urgency of the action or message is represented as a number from 1 (low) to 99 (high). Whereas the priority determines the order of execution of MLMs as they are evoked, the urgency determines the importance of the action of the MLM only if the MLM concludes true (that is, only if the MLM decides to carry out its action). For example,

urgency: 50;;

7. Structured Slot Syntax

7.1 *Tokens*—The structured slots consist of a stream of character strings known as syntactic elements or tokens. These tokens can be classified as follows:

7.1.1 *Reserved Words*—Reserved words are predefined tokens made of letters and digits. They are used to construct

statements, to represent operators, and to represent data constants. Some are not currently used, but are reserved for future use. The predefined synonyms of operators as well as the operators themselves are considered synonyms.

7.1.1.1 The existing reserved words are listed in Annex A2.

7.1.1.2 *The*—**The** is a special reserved word which is ignored wherever it is found in a structured slot. Its purpose is to improve the readability of the structured slots by permitting statements to be more like English.

7.1.1.3 *Case Insensitivity*—The syntax is insensitive to the case of reserved words. That is, reserved words may be typed in uppercase, lowercase, and mixed case. For example, **then** and **THEN** are the same word.

7.1.2 *Identifiers*—Identifiers are alphanumeric tokens. The first character of an identifier must be a letter, and the rest must be letters, digits, and underscores (_). Identifiers must be 1 to 80 characters in length. Reserved words are not considered identifiers; for example, **then** is a reserved word, not an identifier. Identifiers are used to represent variables which hold data.

7.1.2.1 *Case Insensitivity*—The syntax is insensitive to the case of identifiers. That is, identifiers may be typed in uppercase, lowercase, and mixed case. For example, **discharge_status** and **Discharge_Status** are the same identifier.

7.1.3 *Special Symbols*—The special symbols are predefined non-alphanumeric tokens. Special symbols are used for punctuation and to represent operators. They are listed in Annex A3.

7.1.4 *Number Constants*—Constant numbers contain one or more digits (**0** to **9**) and an optional decimal point (**.**). (In Specification E 1238E 1238, **.1** and **345.** are valid numbers.) A number constant may end with an exponent, represented by an **E** or **e**, followed by an optional sign and one or more digits. These are valid numbers:

0
345
0.1
34.5E34
0.1e-4

7.1.4.1 *Negative Numbers*—Negative numbers are created using the unary minus operator (**–**, see 9.9.4). The minus sign is not strictly a part of the number constant.

7.1.5 *Time Constants*—Time constants use the ISO extended format (with the **t** separator) for **date-time combinations** with optional fractional seconds (using **.** format) and with optional time zones (ISO 8601:1988 (E)). The basic format is: yyyy-mm-ddThh:mm:ss. Thus, 1:30 pm on January 1, 1989 would be represented as **1989-01-01T13:30:00**.

7.1.5.1 *Fractional Seconds*—Fractional seconds are represented by appending a decimal point (**.**) and one or more digits (for example, **1989-01-01T13:30:00.123**).

7.1.5.2 *Time Zones*—The local time zone is the default. ISO Coordinated Universal Time is represented by appending a **z** to the end (for example, **1989-01-01T13:30:00.123Z**). The local time zone can be explicitly stated by appending **+** or **–** hh:mm to indicate how many hours and minutes the local time is ahead or behind UTC. Thus EST time zone would use **1989-01-01T13:30:00-05:00**, which would be equivalent to **1989-01-01T18:30:00Z**.

7.1.6 *String Constants*—String constants begin and end with the quotation mark (“, which is ASCII 34). For example,

“this is a string”

7.1.6.1 *Internal Quotation Marks*—A quotation mark within a string is represented by using two adjacent quotation marks. For example,

“this string has one quotation mark:” ” ”

7.1.6.2 *Single Line Break*—Within a string, white space containing a single line break (that is, one of the carriage return-line feed combinations) is converted to a single space. For example,

“this is a string with
one space between ′with′ and ′one′”

7.1.6.3 *Multiple Line Breaks*—Within a string, white space containing a more than one line break is converted to a single line break.

“this is a string with

one line break between ′with′ and ′one′”

7.1.6.4 Strings must not contain double semicolon (**;;**).

7.1.7 *Term Constants*—Term constants begin and end with an apostrophe (’ which is ASCII 39), and they contain a valid MLM filename. For example,

′mlm_name′

7.1.8 *Mapping Clauses*—A mapping clause is a string of characters that begins with **{** and ends with **}**. The only requirement imposed on what is within the curly brackets is that unquoted curly brackets must match (that is, a compiler needs to be able to figure out where the enclosed portion ends), and they must not contain double semicolon (**;;**). Mapping clauses are used in the data slot to signify institution-specific definitions like database queries.

7.1.9 *Comments*—A comment is a string of characters that begins with **/*** and ends with ***/**. They must not contain double semicolon (**;;**). Comments are used to document how the slot works, but they are ignored logically (like **the**).

7.1.10 *White Space*—Any string of spaces, carriage returns, line feeds, and horizontal tabs is known as white space. White space is used to separate other syntactic elements and to format the slot for easier reading. White space is required between any two tokens that may begin or end with letters, digits, or underscores (for example, **if done**). They are also required between two string constants. They are optional between other tokens (for example, **3 + 4** versus **3 + 4**).

7.2 *Organization*—The tokens are organized into the following constructs:

7.2.1 *Statements*—A structured slot is composed of a set of statements. Each statement specifies a logical constraint or an action to be performed. In general, statements are carried out sequentially in the order that they appear. These are examples of statements (each is preceded by a comment that tells what it does):

```
/* this assigns 0 to variable “var1” */
        let var1 be 0;
/* this causes the MLM named “hyperkalemia″ to be
executed */
        call′ hyperkalemia’;
/* this concludes“ true” if the potassium is greater than 5 */
```

```
if potassium >5.0 then
        conclude true;
endif;
```

7.2.1.1 *Statement Termination*—All statements except the last statement in a slot must end with a semicolon (**;**). For the last statement in a slot, the terminating semicolon is optional; whether or not it has a semicolon, the last statement will be followed by a double semicolon (**;;**) marking the end of the slot. For example, the logic slot could contain:

```
logic:
    last_potas : = last potas_list;
    if last_potas > 5.0 then
        conclude true;
    endif;
    ;;
```

7.2.1.2 The syntax of the statements depends upon the individual slot. For a detailed description of the allowable statement types in each structured slot, see Sections 10, 11, 12, and 13.

7.2.2 *Expressions*—Statements are composed of reserved words, special symbols, and expressions. An expression represents a data value, which may belong to any one of the types defined in Section 8. Expressions may contain any of the following:

7.2.2.1 *Constant*—The data value may be represented explicitly using a constant like the number **3**, the time **1991-03-23T00:00:00**, etc. See 7.1. These are valid expressions:

null
true
345.4
“this is a string”
1991-05-01T23:12:23

7.2.2.2 *Variable*—An identifier (see 7.1.2) within an expression signifies a variable (see 7.2.3). These are valid expressions:

var1
this_is_a_variable
a

7.2.2.3 *Operator and Arguments*—An expression may contain an operator and one or more sub-expressions known as arguments. For example, in **3 + 4**, **+** is an operator and **3** and **4** are arguments. The result of such an expression is a new data value, which is **7** in this example. Expressions may be nested so that an expression may be an argument in another expression. These are valid expressions:

4 * cosine 5
var1 = 7 and var2 = 15
(4 + 3) * 7

7.2.2.4 For details on operators, precedence, associativity, and parentheses, see 9.1.

7.2.3 *Variables*—A variable is a temporary holding area for a data value. Variables are not declared explicitly, but are declared implicitly when they are first used. A variable is assigned a data value using an assignment statement (see 10.2.1). When it is later used in an expression, it represents the value that was assigned to it. For example, **var1** is a valid variable name. If the variable is used before it is assigned a value, then its value is **null**.

7.2.3.1 *Scope*—The scope of a variable is the entire MLM, not an individual slot. MLMs cannot read variables from other MLMs directly; thus, variables used in an MLM are not available to MLMs that are called (see 10.2.4).

7.2.3.2 *Special Variables*—Some variables, like event variables, MLM variables, message variables, and destination variables, are special. They can only be used in particular constructs, and not in general expressions. These variables use special assignment statments in the data slot as defined in Section 11 (these special assignment statements are equivalent to declarations for the special variables).

8. Data Types

8.1 The basic function of an MLM is to retrieve patient data, manipulate the data, come to some decision, and possibly perform an action. Data may come from one of several sources: a direct query to the patient database, a constant in the MLM, or the result of an operation on other data. The data classified into several data types.

8.2 *Null*—Null is a special data type that signifies uncertainty. Such uncertainty may be the result of a lack of information in the patient database or an explicit **null** value in the database. Null results from an error in execution, like a type mismatch or division by zero. Null may be specified explicitly within a slot using the word **null** (that is, the null constant). The following expressions result in null (each is preceded by a comment):

```
/* explicit null */
        null
/* division by zero */
        3/0
/* addition of Boolean “true” is illegal */
        true + 3
```

8.3 *Boolean*—The Boolean data type includes the two truth values: true and false. The word **true** signifies Boolean true and the word **false** signifies Boolean false.

8.3.1 The logical operators use tri-state logic by using **null** to signify the third state, uncertainty. For example, **true or null** is true because although **null** is uncertain, a disjunction that includes **true** is always true regardless of the other arguments. But **false or null** is null because **false** in a disjunction adds no information. See 9.4.1 for a full truth table.

8.4 *Number*—There is a single number type, so there is no distinction between integer and real numbers. Number constants (for example, **3.4E-12**) are defined in 7.1.4.

8.5 *Time*—The time data type refers to points in absolute time; it is also referred to as **timestamp** in other systems. Both date and time-of-day must be specified. Times back to the year 1800 must be supported. Time constants (for example, **1990-07-12T00:00:00**) are defined in 7.1.5.

8.5.1 *Granularity*—The granularity of time is always infinitesimal (not discrete seconds). Times stored in patient databases will have varying granularities. When a time is read by the MLM, it is always truncated to the beginning of the granule interval. For example, if the time-of-day is recorded only to the minute, then zero seconds are assumed; if only the date is known, then the time-of-day is assumed to be midnight.

8.5.2 *Midnight*—Midnight (that is, **T00:00:00** in the time-of-day fields) is the beginning of the day to come (not the end of day that just ended).

8.5.3 *Now*—The word **now** is a time constant that signifies the time of execution of the MLM. **Now** is constant through the execution of the MLM; that is, if **now** is used more than once, it will have the exact same value within the same MLM.

8.5.4 *Eventtime*—One way that MLMs are evoked is by a triggering event. For example, the storage of a serum potassium in the patient database is an event that might evoke an MLM. The word **eventtime** is a time constant that signifies the time that the evoking event occurred (for example, the time that the database was updated). The **eventtime** is useful because MLMs may be evoked after a time delay; using **eventtime**, the MLM can query for what has occurred since the evoking event.

8.6 *Duration*—The duration data type signifies an interval of time that is not anchored to any particular point in absolute time. There are no duration constants. Instead one builds durations using the duration operators (see 9.11). For example, **1 day**, **45 seconds**, and **3 months** are durations.

8.6.1 *Sub-types*—The duration data type has two sub-types: months and seconds. The reason for the division is that the number of seconds in a month or in a year depends on the starting date. Durations of months and years are expressed as months. Durations of seconds, minutes, hours, days, and weeks are expressed as seconds. There are no complex durations; the sub-type must be either months or seconds, but not both.

8.6.2 The printing of a duration (that is, its string version) is independent of its internal representation. The health care provider who reads the result of an MLM may not realize that there are two sub-types of durations. How durations are printed is location-specific. For example, the string version of **6E + 08 seconds** might be **19.01 years**. See 9.8.

8.6.3 *Time and Duration Arithmetic*—Operations among times and durations are carried out as follows:

8.6.3.1 *Time – Time*—The subtraction of two times always results in a seconds duration. For example, **1990-03-01T00:00:00 – 1990-02-01T00:00:00** results in **2419200 seconds**.

8.6.3.2 *Time and Seconds*—The addition or subtraction of a time and a seconds duration results in a time. The arithmetic is straightforward: the time is expressed as the number of seconds since some anchor point (for example, **1600-03-01T00:00:00**) and the number of seconds is added to or subtracted from the time. For example, **1990-02-01T00:00:00 + 2419201 seconds** results in **1990-03-01T00:00:01**.

8.6.3.3 *Time and Months*—The addition or subtraction of a time and a months duration results in a time. The time is expressed in date and time-of-day format (for example, **1991-01-31T00:00:00**). Months are then added to or subtracted from the year and month components of the date (that is, **1991-01** in the example). If the resulting time is invalid due to the number of days in the new month, then the days are truncated to the last valid day of the month. For example, **1991-01-31T00:00:00 + 1 month** results in **1991-02-28T00:00:00**. If the month has a fractional component (for example, **1.1 months**) then integer months are used (that is, **1 month** and **2 months** in the example) and the result is gotten through interpolation. For example, **1991-01-31T00:00:00 + 1.1 months** results in **1991-03-03T02:24:00**.

8.6.3.4 *Months and Seconds*—Operations between months and seconds are done by first converting the months arguments to seconds using this conversion constant: 2629746 seconds/month (the average number of seconds in a month in the Gregorian calendar). For example, **1 month / 1 second** results in **2629746**.

8.7 *String*—Strings are streams of characters of variable length. String constants are defined in 7.1.6. For example,

“this is a string constant”

8.8 *Term*—Terms are currently used only to represent MLM filenames within a structured slot. They are used only in a **call** statement (see 10.2.4). In the future they will be used for controlled vocabulary terms. Term constants are defined in 7.1.7. For example,

′mlm_filename2′

8.9 *List*—A list is an ordered set of elements, each of which is null, Boolean, number, time, duration, or string. There are no nested lists; that is, a list cannot be the element of another list. Lists may be heterogeneous; that is, the elements in a list may be of different types. There is one list constant, the empty list, which is signified by using a pair of empty parentheses: (). Other lists are created by using list operators like **,** to build lists from single items (see 9.2). For example, these are valid lists:

4, 3, 5

3, true, 5, null

,1

()

8.10 *Query Results*—The result of a database query has a time value in addition to its data value.

8.10.1 Queries in the data slot retrieve data from the patient database. The result of a query is assigned to a variable for use in the other slots. The result may be a single item or a list.

8.10.2 *Primary Time*, Every item in the patient database is assumed to have some primary time (also called time of occurrence) associated with it. This time is defined as the medically relevant time for that query. For different entities, the primary time might signify different times. The primary time of a blood test might be the time it was drawn from the patient (or the closest to that time), whereas the primary time of a medication order might be the time the order was placed.

8.10.3 Implicit in every query to the patient database is a request for the primary time of the data. For example, when one retrieves a list of serum potassiums, one actually retrieves a list of pairs. Each pair contains a data value (the serum potassium) and a time value (when it was drawn).

8.10.4 *Retrieval Order*—The result of a query is always sorted in chronological order by the primary time of the result.

8.10.5 *Data Value*—If a variable has been assigned the result of a query, then the use of the variable always refers to the data value. For example, if **potas** is a variable that has been assigned a list of serum potassiums, then one could use this statement to check the value of the most recent potassium measurement:

if last potas > 5.0 then
conclude true;
endif;

8.10.6 *Time Function Operator*—By using the **time** operator (see 9.17), one can get to the primary time value. For example, one could use this statement to check the primary time of the most recent potassium measurement:

if time of (last potas) is within the past 3 days then
conclude true;
endif;

8.10.7 The **eventtime** is not necessarily the primary time. For example, if the storage of a serum potassium evokes an MLM, then the **eventtime** is the time that the result was stored in the database, but the primary time of the result is the time that it was drawn from the patient.

9. Operator Descriptions

9.1 *General Properties*—Operators are used in expressions to manipulate data. They accept one or more arguments (data values) and they produce a result (a new data value). The following properties apply to the operator definitions in this section.

9.1.1 *Number of Arguments*—Operators may have one, two, or three arguments. Some operators have two forms: one with one argument and one with two arguments. Operators are described as follows:

unary operator—one argument
binary operator—two arguments
ternary operator—three arguments

9.1.2 *Data Type Constraints*—Most operators work on only a subset of all the data types. Every operator description includes a type constraint that shows the position and allowable types of all of its arguments. Its general format is like this:

<type> : = <type> op <type>

9.1.2.1 In this constraint, **op** is the operator being described.

9.1.2.2 **<type>** is one of the following:

<null>—null data type
<Boolean>—Boolean data type
<number>—number data type
<time>—time data type
<duration>—duration data type
<string>—string data type
<item>—not used in expressions, only in "call" statements (see 10.2.4)
<list>—treated specially
<any-type>—null, Boolean, number, time, duration, or string
<non-null>—Boolean, number, time, duration, or string
<ordered>—number, time, duration, or string

9.1.2.3 **<type>** to the right of the **: =** indicates the data type of the arguments. If the operator is applied to an argument with a type outside of its defined set, then **null** results. For example, ** is not defined for the time data type so **3**1991-03-24T00:00:00** results in **null**. For most operators, **null** is not in the defined set, so **null** is returned when **null** is an argument. For example, **null** is not defined for **+** so **3 + null** results in **null**.

9.1.2.4 **<type>** to the left of the **: =** indicates the data type of the result. Unless stated otherwise, the operators can also return **null** regardless of the stated usual result type.

9.1.3 *List Handling*—Except as otherwise stated, lists are treated as follows:

9.1.3.1 When a list is the argument to an operator, that operator is applied to each element in the list one at a time, and the result of the operation is a list of the same length as the operator. For example, **abs(-3, -4, -5)**, which returns the absolute value, results in **3, 4, 5**.

9.1.3.2 When an operator has more than one argument and two or more are lists, then all the lists must have the same number of elements, and the operation is performed between corresponding elements in each list. The result is a list of the same length. If the lists do not have the same number of elements, then a single **null** is the result. For example, **(3,4,5) + (1,2,2)** results in **4, 6, 7** but **(1,2) + (3,4,5)** results in **null**.

9.1.3.3 When an operator has more than one argument and at least one is a list and at least one is a single item (that is, not a list) then the operation is performed as follows. The operation is performed between each element on the list and the single item, and the result is a list. For example, **(3,4,5) + 1** results in **4, 5, 6**.

9.1.4 *Primary Time Handling*—The queries attached a primary time to their result (see 8.10). Some operators maintain those primary times and others lose them. Except as otherwise stated, primary times are treated as follows:

9.1.4.1 Unary operators maintain primary times. In this example, **result1** still has primary times attached if **data1** is the result of a query:

result1 : = sin(data1);

9.1.4.2 Binary and ternary operators maintain primary times except if arguments are derived from different queries. This is necessary because even if the arguments are lists of the same length, it is not clear whose primary time to attach to the result. Thus, if **data1** and **data2** are the results of different queries, then in this example, **result1** has no primary times attached:

result1 : = data1 + data2;

9.1.5 *Operator Precedence*—Expressions are nested structures which may contain more than one operator and several arguments. The order in which operators are executed is decided by using an operator property called precedence. Operators groups into several precedence groups. Operators of higher precedence are performed before operators of lower precedence. For example, the expression **3 + 4*5** (three plus four times five) is executed as follows: since * has higher precedence than + , it is performed first so that **4*5** results in **20**; then + is performed so that **3 + 20** results in **23**.

9.1.5.1 *Precedence Table*—The operators are shown grouped by precedence in Annex A4.

9.1.6 *Associativity*—When an expression contains more than one operator within the same precedence group, the operators' associativity property decides the order of execution. There are three types of associativity:

9.1.6.1 *Left*—Left associative operators are executed from left to right. For example, **3–4–5** has two subtractions (**–**). Since they are the same operator, they must be in the same

precedence group. Since **–** is left associative, **3–4** is performed first resulting in **(–1)**; then **(–1)–5** is performed, resulting in **(–6)**.

9.1.6.2 *Right*—Right associative operators are executed from right to left. For example, **average sum 3** has two operators in the same precedence group. Since they are right associative, **sum 3** is performed first resulting in **3**; then **average 3** is performed, resulting in **3**.

9.1.6.3 *Non-Associative*—Non-associative operators cannot have more than one operator from the same precedence group in the same expression unless parentheses are used. Thus the expression **2**3**4** is illegal since ** (the exponentiation operator) is non-associative.

9.1.6.4 The associativity of each operator is shown in Annex A4.

9.1.7 *Parentheses*—One can use parentheses to force a different order of execution. Expressions within parentheses are always performed before ones outside of parentheses. For example, the expression **(3 + 4)*5** is executed as follows: **3 + 4** is within parentheses, so it is performed first regardless of precedence, resulting in **7**; then * is performed so that **7*5** results in **35**. Similarly, **(2**3)**4** is a legal expression which results in **4096**.

9.2 *List Operators*—The list operators do not follow the default list handling, or the default primary time handling.

9.2.1 **,** *(binary, left associative)*—Binary **,** (list concatenation) appends two lists, appends a single item to a list, or creates a list from two single items. Binary **,** always loses the associated primary times of arguments that are the results of queries; use the **merge** operator instead. Its usage is:

<any-type> : = <any-type>, <any-type>
(4,2) : = 4, 2
(4,"a",2) : = (4,"a") , 2

9.2.2 **,** *(unary, non-associative)*—Unary **,** turns a single element into a list of length one. It does nothing if the argument is already a list. Unary **,** maintains the associated primary times if its argument is the result of a query. Its usage is (where **(,3)** means a list with **3** as its only element):

<any-type> : = , <any-type>
(,3) : = , 3

9.2.3 *Merge (Binary, Left Associative)*—The **merge** operator appends two lists, appends a single item to a list, or creates a list from two single items. It then sorts the result in chronological order based on the primary times of the elements. Both arguments must be the result of a query; otherwise **null** is returned. The primary times are maintained. **Merge** is used to put together the results of two separate queries. Its usage is (assuming that **data1** has a data value of **2** and a time of **1991-01-02T00:00:00**, and that **data2** has these data values **1,3** and these time values **1991-01-01T00:00:00**, **1991-01-03T00:00:00**):

<any-type> : = <any-type> MERGE <any-type>
(1, 2, 3) : = data1 MERGE data2
null : = (4,3) MERGE (2,1)

9.3 *Where Operator*, The **where** operator does not follow the default list handling.

9.3.1 *Where (binary, non-associative)*—The **where** operator performs the equivalent of a relational **select ... where ...** on its left argument. In general, the left argument is a list, often the result of a query to the database. The right argument is usually of type Boolean (although this is not required), and must be the same length as the left argument. The result is a list that contains only those elements of the left argument where the corresponding element in the right argument is Boolean **true**. If the right argument is anything else, including **false**, **null**, or any other type, then the element in the left argument is dropped. Its usage is:

<any-type> : = <any-type> WHERE <any-type>
(1,3) : = (1,2,3,4) WHERE (true,false,true,3)

9.3.1.1 **Where** handles mixed single items and lists in a manner analogous to the other binary operators. If the right argument to **where** is a single item, then if it is **true**, the entire left argument is kept (whether or not it is a list); if it is not **true**, then the empty list is returned. If only the left argument is a single item, then the result is a list with as many of the single items as there are elements equal to **true** in the right argument. If the two arguments are lists of different length, then a single **null** results. For example,

1 : = 1 WHERE true
(1,2,3) : = (1,2,3) WHERE true
(1,1) : = 1 WHERE (true,false,true)
null : = (1,2,3,4) WHERE (true,false,true)

9.3.1.2 **Where** is generally used to select certain items from a list. The list is used as the left argument, and some comparison operator is applied to the list in the right argument. For example, **potassium list where potassium_list > 5.0** would select from the list those values that are greater than 5.

9.3.1.3 *It*—The word **it** and its synonym **they** are used in conjunction with **where**. To simplify **where** expressions, **it** may be used in the right argument to represent the entire left argument. For example, **potassium_list where they > 5.0** would select those values from the list that are greater than 5. **It** is most useful when the left argument is a complex expression; for example, **(potassium_list + sodium_list/3) where it > 5.0** would assign the entire expression in parentheses to **it**. If there are nested **where** expressions, **it** refers to the left argument of the innermost **where**. If **it** is used outside of a where expression, then it has a value of **null**.

9.3.1.4 **Where** can also be used to count how many items in a list are Boolean **true** by using this construct:

count(Boolean_list where it = true)

9.3.1.5 **Where** can calculate a weighted sum using this construct (assume that risk1 to risk3 are Boolean flags):

sum((0.45, 0.25, 0.30) where (risk1, risk2, risk3))

9.4 *Logical Operators*:

9.4.1 *Or (binary, left associative)*—The **or** operator performs the logical disjunction of its two arguments. If either argument is **true** (even if the other is not Boolean), the result is **true**. If both arguments are **false**, the result is **false**. Otherwise the result is **null**. Its usage is:

<Boolean> : = <any-type> OR <any-type>
true : = true OR false
false : = false OR false
true : = true OR null
null : = false OR null
null : = false OR 3.4

9.4.1.1 Its truth table is given here. **Other** means any of these data types: null, number, time, duration, or string.

	OR	TRUE	FALSE	other (Right argument)
(Left argument)	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NULL
	other	TRUE	NULL	NULL

9.4.2 *And (binary, left associative)*—The **and** operator performs the logical conjunction of its two arguments. If either argument is **false** (even if the other is not Boolean), the result is **false**. If both arguments are **true**, the result is **true**. Otherwise the result is **null**. Its usage is:

<Boolean> : = <any-type> AND <any-type>
false : = true AND false
null : = true AND null
false : = false AND null

9.4.2.1 Its truth table is given here. **Other** means any of these data types: null, number, time, duration, or string.

	AND	TRUE	TRUE	other (Right argument)
(Left argument)	TRUE	TRUE	FALSE	NULL
	FALSE	FALSE	FALSE	FALSE
	other	NULL	FALSE	NULL

9.4.3 *Not (unary, non-associative)*—The **not** operator performs the logical negation of its argument. Thus **true** becomes **false**, **false** becomes **true**, and anything else becomes **null**. Its usage is:

<Boolean>: = NOT< any-type>
true: = NOT false
null : = NOT null

9.4.3.1 Its truth table is given here. **Other** means any of these data types: null, number, time, duration, or string.

NOT	TRUE	FALSE	other
	FALSE	TRUE	NULL

9.5 *Simple Comparison Operators*:

9.5.1 = *(binary, non-associative)*—The = operator has two synonyms: **eq** and **is equal**. It checks for equality, returning **true** or **false**. If the arguments are of different types, **false** is returned. If an argument is **null**, then **null** is always returned. Its usage is:

<Boolean> : = <non-null> = <non-null>
false : = 1 = 2
(null,true) : = (1,2) = (null,2)

9.5.1.1 *Testing Null*—Use **is present** instead of = to test whether an argument is equal to **null**. See 9.6.15.

9.5.2 **<>** *(binary, non-associative)*—The **<>** operator has two synonyms: **ne** and **is not equal**. It checks for inequality, returning **true** or **false**. If the arguments are of different types, **true** is returned. If an argument is **null**, then **null** is returned. Its usage is:

<Boolean> : = <non-null>< >< non-null>
true : = 1< > 2

9.5.3 < *(binary, non-associative)*—The < operator has three synonyms: **It**, **is less than**, and **is not greater than or equal**. It is used on ordered types; if the types do not match, **null** is returned. Its usage is:

<Boolean> : = <ordered>< <ordered>
true : = 1< 2
true : = 1990-03-02T00:00:00< 1990-03-10T00:00:00
true : = 2 days < 1 year
true : = ”aaa”< “aab”

9.5.4 < = *(binary, non-associative)*—The < = operator has three synonyms: **le**, **is less than or equal**, and **is not greater than**. It is used on ordered types; if the types do not match, **null** is returned. Its usage is:

<Boolean> : = <ordered> < = <ordered>
true : = 1 < = 2
true : = 1990-03-02T00:00:00< = 1990-03-10T00:00:00
true : = 2 days < = 1 year
true : = ”aaa” < = ”aab”

9.5.5 > *(binary, non-associative)*—The > operator has three synonyms: **gt**, **is greater than**, and **is not less than or equal**. It is used on ordered types; if the types do not match, **null** is returned. Its usage is:

<Boolean> : = <ordered> > <ordered>
false : = 1 > 2
false : = 1990-03-02T00:00:00 > 1990-03-10T00:00:00
false : = 2 days > 1 year
false : = ”aaa” > “aab”

9.5.6 > = *(binary, non-associative)*—The > = operator has three synonyms: **ge**, **is greater than or equal**, and **is not less than**. It is used on ordered types; if the types do not match, **null** is returned. Its usage is:

<Boolean> : = <ordered> > = <ordered>
false : = 1 > = 2
false : = 1990-03-02T00:00:00 > = 1990-03-10T00:00:00
false : = 2 days > = 1 year
false : = ”aaa” >= ”aab”

9.6 *Is Comparison Operators*—The following comparison operators include the word **is**, which can be replaced with **are**, **was**, or **were**. An optional **not** may follow the **is**, negating the result. For example, these are valid:

surgery_time WAS BEFORE discharge_time
surgery_time IS NOT AFTER discharge_time

9.6.1 *Is Equal (binary, non-associative)*—See 9.5.1.

9.6.2 *Is Less Than (binary, non-associative)*—See 9.5.3.

9.6.3 *Is Greater Than (binary, non-associative)*—See 9.5.5.

9.6.4 *Is Less Than or Equal (binary, non-associative)*—See 9.5.4.

9.6.5 *Is Greater Than or Equal (binary, non-associative)*—See 9.5.6.

9.6.6 *Is Within ... To (ternary, non-associative)*—The **is within ... to** operator checks whether the first argument is within the range specified by the second and third arguments; the range is inclusive. It is used on ordered types; if the types do not match, **null** is returned. Its usage is:

<Boolean> : = <ordered> IS WITHIN< ordered> TO <ordered>
true : = 3 IS WITHIN 2 TO 5
true : = 1990-03-10T00:00:00 IS WITHIN 1990-03-05T00:00:00 TO 1990-03-15T00:00:00
true : = 3 days IS WITHIN 2 days TO 5 months
true : = ”ccc” IS WITHIN “a” TO “d”

9.6.7 *Is Within ... Preceding (ternary, non-associative)*—The **is within ... preceding** operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument minus the second to the third). Its usage is:

<Boolean> : = <time> IS WITHIN <duration> PRECEDING <time>

true : = 1990-03-08T00:00:00 IS WITHIN 3 days PRECEDING 1990-03-10T00:00:00

9.6.8 *Is Within ... Following (ternary, non-associative)*—The **is within ... following** operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument to the third plus the second). Its usage is:

<Boolean> : = <time> IS WITHIN <duration> FOLLOWING <time>

false : = 1990-03-08T00:00:00 IS WITHIN 3 days FOLLOWING 1990-03-10T00:00:00

9.6.9 *Is Within ... Surrounding (ternary, non-associative)*—The **is within ... surrounding** operator checks whether the left argument is within the inclusive time period defined by the second two arguments (from the third argument minus the second to the third plus the second). Its usage is:

<Boolean> : = <time> IS WITHIN< duration> SURROUNDING <time>

true : = 1990-03-08T00:00:00 IS WITHIN 3 days SURROUNDING 1990-03-10T00:00:00

9.6.10 *Is Within Past (binary, non-associative)*—The **is within past** checks whether the left argument is within the time period defined by the right argument (**now** minus the right argument to **now**). Its usage is (assuming **now** is 1990-03-09T00:00:00):

<Boolean> : = <time> IS WITHIN PAST< duration>

true : = 1990-03-08T00:00:00 IS WITHIN PAST 3 days

9.6.11 *Is Within Same Day As (binary, non-associative)*—The **is within same day as** operator checks whether the left argument is on the same day as the second argument. Its usage is:

<Boolean> : = <time> IS WITHIN SAME DAY AS <time>

true : = 1990-03-08T11.11.11 IS WITHIN SAME DAY AS 1990-03-08T01.01.01

9.6.12 *Is Before (binary, non-associative)*—The **is before** operator checks whether the left argument is before the second argument; it is not inclusive. Its usage is:

<Boolean> : = <time> IS BEFORE< time>

false : = 1990-03-08T00:00:00 IS BEFORE 1990-03-07T00:00:00

false : = 1990-03-08T00:00:00 IS BEFORE 1990-03-08T00:00:00

9.6.13 *Is After (binary, non-associative)*—The **is after** operator checks whether the left argument is after the second argument; it is not inclusive. Its usage is:

<Boolean> : = <time> IS AFTER <time>

true : = 1990-03-08T00:00:00 IS AFTER 1990-03-07T00:00:00

9.6.14 *Is In (binary, non-associative)*—The **is in** operator does not follow the default list handling. It checks for membership of the left argument in the right argument, which is usually a list. If the left argument is a list, then a list results; if the left argument is a single item, then a single item results. If the right argument is a single item, then it is treated as a list of length one. Its usage is:

<Boolean> : = <any-type> IS IN <any-type>

false : = 2 IS IN (4,5,6)

(false,true) : = (3,4) IS IN (4,5,6)

9.6.15 *Is Present (unary, non-associative)*—The **is present** operator has one synonym: **is not null**. (Similarly, **is not present** has one synonym: **is null**.) It returns **true** if the argument is not **null**, and it returns **false** if the argument is **null**. **Is present** never returns **null** itself. This operator is used to test whether an argument is **null** since **arg = null** always results in **null** regardless of **arg**. Its usage is:

<Boolean> : = <any-type> IS PRESENT

true : = 3 IS PRESENT

false : = null IS PRESENT

9.6.16 *Is Null (unary, non-associative)*—See 9.6.15.

9.6.17 *Is Boolean (unary, non-associative)*—The **is boolean** operator returns **true** if the argument's data type is Boolean. Otherwise it returns **false**. **Is boolean** never returns **null**. Its usage is:

<Boolean> : = <any-type> IS BOOLEAN

true : = false IS BOOLEAN

true : = 3 IS NOT BOOLEAN

false : = null IS BOOLEAN

9.6.18 *Is Number (unary, non-associative)*—The **is number** operator returns **true** if the argument's data type is number. Otherwise it returns **false**. **Is number** never returns **null**. Its usage is:

<Boolean> : = <any-type> IS NUMBER

true : = 3 IS NUMBER

false : = null IS NUMBER

9.6.18.1 The **is number** is useful for ensuring that a list is all numbers before an aggregation operator is applied. This avoids returning **null**. For example,

sum(serum_K where it IS NUMBER)

9.6.19 *Is String (unary, non-associative)*—The **is string** operator returns **true** if the argument's data type is string. Otherwise it returns **false**. **Is string** never returns **null**. Its usage is:

<Boolean> : = <any-type> IS STRING

true : = "asdf" IS STRING

false : = null IS STRING

9.6.20 *Is Time (unary, non-associative)*—The **is time** operator returns **true** if the argument's data type is time. Otherwise it returns **false**. **Is time** never returns **null**. Its usage is:

<Boolean> : = <any-type> IS TIME

true : = 1991-03-12T00:00:00 IS TIME

false : = null IS TIME

9.6.21 *Is Duration (unary, non-associative)*—The **is duration** operator returns **true** if the argument's data type is duration. Otherwise it returns **false**. **Is duration** never returns **null**. Its usage is:

<Boolean> : = <any-type> IS DURATION