The first version of this standard was developed under the auspices of the American Society for Testing and Materials (ASTM) and published in April 1992 as ASTM E1460-92. This new and revised version, developed and published under Health Level Seven, Inc (HL7), is in most cases backward compatible and is based on the previous ASTM version. The authors of the standard, and the members of the HL7 committee administering the standard, wish to acknowledge and thank ASTM for their contributions to the Arden Syntax standard.

Arden Syntax for Medical Logic Systems

TABLE OF CONTENTS

1 SCOPE

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.

2 REFERENCED DOCUMENTS

2.1 ASTM Standards[1]:

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

E 1384 Guide for Content and Structure of an Automated Primary Record of Care

2.2 ISO Standards[2]:

ISO 8601 - 1988 Data Elements and Interchange Formats-Information Interchange (representation of dates and times)

ISO 8979 - 1986 Latin-1 Coded Character Set

2.3 ANSI Standards[3]:

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

ANSI/ISO 9899 Programming Language C

ANSI/ISO/IEC 9075 Information technology -- Database languages -- SQL

[1] Annual Book of ASTM Standards, Vol 14.01.

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

[3] Available from American National Standards Institute, 1430 Broadway, New York, NY 10018.

2.4 Health Level Seven Standards[4]:

HL7 Version 2.3

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 Descriptions 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.

3.2.6 event, n

a clinically meaningful change in state. This is often, but not always, reflected by a change in the clinical database. For example, ordering a medication is an event that could update the clinical database; when the stop time of the medication order is passed, the stopping of the medication would be an event, even though there might not be any change to the database.

3.3 Notation Used in This Standard

Throughout this standard, the location for optional elements is noted by placing the optional elements inside square brackets (**[ ]**). This is not to be confused with the element operator **[ ]** (see Section 9.12.18). Thus, **Is [Not] Equal** means that **Is Equal** and **Is Not Equal** are both valid constructs. The two most common optional elements are **not** and **of**

[4] Available from Health Level Seven, Inc., 3300 Washtenaw Ave, Suite 227, Ann Arbor, MI 48104.

4 SIGNIFICANCE AND USE

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.

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.

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.

This specification, which is an outcome of the Columbia-Presbyterian Medical Center 1989 Arden Homestead retreat on sharing health knowledge bases, was derived largely from HELP of LDS 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) [international users may extend this by using ISO 8859/1 ("Latin-1"), but a conforming implementation need only implement X3.4]. 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:**. MLMs may be separated by white space, as defined in Section 7.1.10 and/or comments as defined in Section 7.1.9.

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), horizontal tab (ASCII 9), vertical tab (ASCII 11), and form feed (ASCII 12) may be used. The use of horizontal tab is discouraged because there is no agreement on how many spaces it represents. Other characters, such as the bell and backspace, are not allowed within the MLM. Inside a string constant (Section 7.1.6) or comment (Section 7.1.9), these character set restrictions are lifted.

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

5.4 White Space

The space, carriage return, line feed, horizontal tab, vertical tab, and form feed are collectively referred to as white space. See also Section 7.1.10.

5.5 General Layout

Annex A1 contains a context-free grammar (formal description) of Arden Syntax MLMs expressed in Backus-Naur Form **(3)**. See Appendix X1 for MLM examples. 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:**).White space may precede the category name and follow the colon, but no white space is allowed between the category name and the colon. Categories must appear in the order they appear in this standard.

5.7 Slots

Within each category is a set of slots.

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. White space may precede the slot name and follow the colon, but no white space is allowed between the slot name and the colon. The content of the slot body depends upon the slot, but it must not contain a double semicolon, except inside comments (Section 7.1.9), string constants (Section 7.1.6), and mapping clauses (Section 7.1.8).

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

A textual slot contains 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. For required textual slots, the text may be empty.

5.8.2 Textual List Slots

Some slots contain textual lists. These are lists of arbitrary textual phrases, optionally separated by single semicolons (**;**). An example of a textual list slot is the keywords slot. The list may be empty. It may not contain a double semicolon (which ends the 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:**). White space may precede the terminator and follow the colon but no white space is allowed between the terminator and the colon.

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**). See also Sections 7.1.1.2 and 7.1.2.1.

6 SLOT DESCRIPTIONS

Next to each 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 the order they appear in this specification.

6.1 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. The maintenance category also contains information about the version of the Arden Syntax that is being used.

6.1.1 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;;
```

6.1.2 Mlmname (coded, required)

The mlmname 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 (_). An mlmname may be 1 to 80 characters in length. Mlmnames are insensitive to case. The mlmname is distinct from the name of the ASCII file, which happens to hold one or more MLMs. For example,

```
mlmname: hepatitis_B_in_pregnancy;;
```

While mlmname is preferred as the name of this slot, filename is also permitted for backward compatibility.

6.1.3 Arden Syntax version (coded, required)

The Arden Syntax version informs the compiler which version of the standard has been used to write the MLM. If this slot is missing, the MLM is assumed to be written with the ASTM E1460-1992 standard (which didn't include this slot). Otherwise, the slot is of the following form (the string "Version 2"" may use upper or lower case letters):

```
arden: Version 2;;
```

This slot is required for version 2 of the syntax, but is optional for backward compatibility. That is, if it is missing, the assumed version is version 1.

6.1.4 Version (textual, required)

The current version of the MLM is arbitrary text, up to 80 characters in length, as is convenient for the institution's version control system (such as SCCS or RCS). It is suggested that versions start at 1.00 and advance by .01 for small revisions and by 1 for large revisions. The exact form of the version information is institution-specific, but must allow determining which MLM is the most recent (see Section 11.2.3). For example,

```
version: 1.00;;
```

6.1.5 Institution (textual, required)

The institution slot contains the name of the authoring institution, up to 80 characters in length. For example,

```
institution: Columbia University;;
```

6.1.6 Author (textual list, required):

The author slot is free-form text. It should contain 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, comma, and degrees.

An electronic mail address enclosed in parentheses may optionally follow each author's name. Internet addresses are assumed. For example,

```
author: John M. Smith, Jr., M.D. (jms@camis.columbia.edu);;
```

6.1.7 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. For example,

```
specialist: Jane Doe, Ph.D.;;
```

or

```
specialist: ;;
```

6.1.8 Date (coded, required)

The date of last revision of the MLM must be placed in this slot. Either a date or a date-time (that is, a point in absolute time composed of a date plus a time-of-day) can be used. The format for dates and for date-time combinations is ISO extended format (with the **T** or **t** separator) with optional time zones (ISO 8601:1988 (E)). Dates are **yyyy-mm-dd** so that January 2, 1989 would be represented as 1989-01-02. The earliest date-time Arden has to support is January 1, 1800 (1800-01-01T00:00:00Z). Times are **yyyy-mm-ddThh:mm:ss** with optional fractional seconds and optional time zones. Thus, 1:30 p.m. on January 2, 1989 UTC would be represented as 1989-01-02T13:30:00Z. For example,

```
date: 1989-01-02;;
```

6.1.9 Validation (coded, required):

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 (when an MLM is written, this should be the initial value), or

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

An example is:

```
validation: testing;;
```

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

6.2 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.

6.2.1 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;;
```

6.2.2 Explanation (textual, required)

The slot explains briefly in plain English how the MLM works. The explanation can be 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.;;
```

6.2.3 Keywords (textual list, required)

Keywords are descriptive words used for searching through modules. UMLS terms **(4)** are preferred but not mandatory. Terms are delimited by semicolons (commas are allowed within a keyword). For example,

```
keywords: hepatitis B; pregnancy;;
```

6.2.4 Citations (structured / textual, optional)

There are two supported formats for the citations slot. The first is a textual format with no implied structure. The textual format is provided for backward compatibility and is a deprecated form. The second is a structured format described later in this section. Citations to the literature should be entered in Vancouver style **(5)**. Citations must be numbered, serving as specific references. The individual citations may also be assigned a type. The type should follow the number and specify the function of the citation for the particular MLM. Citation types are:

a) **Support --** citations which support, verify, or validate the algorithm in the logic slot;

b) **Refute --** citations which refute or offer alternatives to the algorithm in the logic slot;

For example,

```
citations:
1. SUPPORT  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.
;;
```

6.2.5 Links (structured / textual, optional)

There are two supported formats for the links slot. The first is a textual format with no implied structure. The textual format is provided for backward compatibility and is a deprecated form. The second is a structured format described later in this section. 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 individual links are delimited by semicolons. The contents of the links are institution-specific. Links to sites on intranets or the internet should be prefixed by the term "URL"" (Uniform Resource Locator) and the title of the document and link text should follow the defined standards for representing protocols and data sources (e.g. "Document Title", 'FILE://link.html'; "Second Document", 'http://www.nlm.nih.gov/'). Electronic material can also be entered in the **citations** slot above. The preferred form for structured links is:
link type, space (ASCII 32), link description (Arden Syntax string), comma, link text (Arden Syntax term). The only required element is the link text.

For example:

```
links:
    OTHER_LINK 'CTIM .34.56.78';
    MESH 'agranulocytosis/ci and sulfamethoxazole/ae';
    URL "NLM Web Page", 'http://www.nlm.nih.gov/';
    URL "Visible Human Project",
       'http://www.nlm.nih.gov/research/visible/visible_human.html';
    URL "DOS HTML File", 'file://doslinx.htm';
    URL "UNIX HTML File", 'file://UnixLinx.html/';
;;
```

Each institution should test for expired links when receiving shared MLMs.

6.3 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 condition to be tested (logic slot), and the action to take should the condition be true (action slot).

6.3.1 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. For backward compatibility with the 1992 standard, the type **data-driven** (with a dash "**-**" separating the words) is also permitted. That is,

```
type: data_driven;;
```

or

```
type: data-driven;;
```

6.3.2 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.

6.3.3 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 satisfy their evoke criteria simultaneously. An institution may choose whether or not to use a priority. The institution is responsible for maintaining these numbers to avoid conflicts. A borrowing institution will need to adjust these numbers to suit its collection of MLMs. If the priority slot is omitted, a default value of 50 is used. For example,

```
priority: 90;;
```

6.3.4 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.

6.3.5 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 10.

6.3.6 Action (structured, required)

6.3.7 This slot contains the action produced when the logic slot concludes **true**. The details of this slot are explained in Section **12**. Urgency (coded, optional)

The urgency of the action or message is represented as a number from 1 (low) to 99 (high), or by a variable representing a number from 1 to 99. 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). If the urgency slot is omitted, or the variable representing urgency is null or outside the range 1 to 99, a default urgency of 50 is used. For example,

```
urgency: 90;;
urgency: urg_var;;
```

7 STRUCTURED SLOT SYNTAX

7.1 Tokens

The structured slots consist of a stream of character strings known as lexical 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.

The existing reserved words are listed in Annex A2.

7.1.1.1 The

The is a special reserved word which is ignored wherever it is found in a structured slot (that is, it is treated exactly the same as white space). Its purpose is to improve the readability of the structured slots by permitting statements to be more like English.

7.1.1.2 Case Insensitivity

With the exception of the **format with …** format specification, 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. See Sections 5.10 and 9.8.2 and Annex A5.

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. It is an error for an identifier to be longer than 80 characters. 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. See Sections 5.10 and 7.1.1.2.

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 (**.**). (As in Specification E 1238 and HL7 2.3, **.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
.3
3.
3e10
```

7.1.4.1 Negative Numbers

Negative numbers are created using the unary minus operator (**-**, see Section 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** or **t** separator) for date-time combinations with optional fractional seconds (using **.** format) and with optional time zones (see Section 6.1.8).

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 (UTC) 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 the EST (Eastern Standard Time, North America) time zone would use **1989-01-01T13:30:00-05:00**, which would be equivalent to **1989-01-01T18:30:00Z**.

7.1.5.3 Constructing times

The + operator can be used to construct a time from durations. Here is an example of constructing a time: **1800-01-01 + (1993-1800)years + (5-1)months + (17-1)days** produces the value **1993-05-17**.

7.1.6 String Constants

String constants begin and end with the quotation mark (", which is ASCII 34). For example,

```
"this is a string".
```

There is no limit on the length of strings.

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 (see Section 5.3) 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 more than one line break is converted to a single line break.

```
 "this is a string with



one line break between 'with' and 'one'"
```

7.1.7 Term Constants

Term constants begin and end with an apostrophe (' which is ASCII 39), and they contain a valid mlmname. For example,

```
'mlm_name'
```

7.1.8 Mapping Clauses

A mapping clause is a string of characters that begins with **{** and ends with **}** (ASCII 123 and 125, respectively). Mapping clauses are used in the data slot to signify institution-specific definitions such as database queries. The only requirement imposed on what is within the curly brackets is that curly brackets are not allowed within mapping clauses. The definition of comments and quotes inside mapping clauses is not specified by this standard; it is recommended that they be the same as those given in this standard. The Arden Syntax conventions for variable names, such as case insensitivity or the treatment of **the** as white space, need not be observed in a mapping clause. A **<mapping>** may (in an implementation-defined manner), within the curly brackets, use Arden variables; but it cannot set any Arden variables (Arden variables can only set by the **<var>**(s) on the left side of the assignment operator). Because of this, an MLM may require some modification before it can be processed at another institution, even if the other institution's compiler is set to skip over read mappings.

It is strongly recommended that MLM authors include comments to all the mapping clauses used in an MLM, so MLM recipients understand the intention of the mapping clause definition when sharing MLMs. Identifiers from the UMLS Metathesaurus could aid in identifying and describing the concepts in the comments. Authors should also put all literals and constants in the data slot, with explanation, to allow MLM recipients to more easily customize MLMs.

7.1.9 Comments

A comment is a string of characters that begins with **/*** and ends with ***/**. Comments are used to document how the slot works, but they are ignored logically (like **the** and other white space). Comments do not nest (e.g., **/* A comment /* */** is a single comment). A comment need not be preceded or followed by white space. Thus, **x/**/y** is the same as **x y**.

A comment may also be specified by the characters // through line break (see Section 5.3). When // is encountered, everything else on the line is ignored, including ***/**.

7.1.10 White Space

Any string of spaces, carriage returns, line feeds, horizontal tabs, vertical tabs, form feeds, and comments 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**). See also Sections 5.4 and 7.1.1.1.

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 for the last statement in a slot must end with a semicolon (**;**). Thus, the semicolon acts as a statement separator. If the last statement of a slot has a terminating semicolon, there must be at least one white space between it and the double semicolon that terminates the slot (**;;;** is illegal but **;/**/;;** is legal). For example, the logic slot could contain:

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

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. 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 Section 7.1.2) within an expression signifies a variable (see Section 7.2.3). These are valid variables:

```
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
```

For details on operators, precedence, associativity, and parentheses, see Section 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 Section 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 Section 10.2.4). Non-Arden variables may be referenced and set within mapping statements, as restricted by the special rules for the individual mapping statements (for example, Section 11.2.3); in mapping statements, Arden variables may be referenced but not set. It is institution-defined how conflicts between Arden and non-Arden variable names are resolved.

7.2.3.2 Special Variables

Some variables, such as 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 statements in the data slot as defined in Section 11 (these special assignment statements are equivalent to declarations for the special variables). Special variables can be converted to strings and passed as arguments. The only valid operators on special variables are **is [not] equal** (Section 9.6.1), = (Section 9.5.1), and <> (Section 9.5.2).

8 DATA TYPES

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 various sources, such as a direct query to the patient database, a constant in the MLM, or the result of an operation on other data.

Data items may be kept in an ordered collection, called a list (ordered by position in the list, not by primary time). Lists are described further in Section 8.8.

The data are classified into several data types.

8.1 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, such as 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 + 3
```

8.2 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.

The logical operators use tri-state logic by using **null** to signify the third state, uncertainty. For example, **true or null** is true. Although **null** is uncertain, a disjunction that includes **true** is always true regardless of the other arguments. However, **false or null** is null because **false** in a disjunction adds no information. See Section 9.4 for full truth tables.

8.3 Number

There is a single number type, so there is no distinction between integer and floating point numbers. Number constants (for example, **3.4E-12**) are defined in Section 7.1.4. Internally, all arithmetic is done in floating point. For example, **1/2** evaluates to **0.5**.

8.4 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 and times before 1800-01-01 are not valid. Time constants (for example, **1990-07-12T00:00:00**) are defined in Section 7.1.5.

8.4.1 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 he 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.

8.4.2 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).

8.4.3 Now

The word **now** is a time constant that signifies the time when the MLM started execution. **Now** is constant through the execution of the MLM; that is, if **now** is used more than once, it will have the same value within the same MLM. **Now** inside a nested MLM may be different from the **now** of the calling MLM.

8.4.4 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.

8.4.5 Triggertime

If the MLM is triggered directly by an event or another MLM, the **triggertime** is the same as the **eventtime**. If the MLM is triggered by a delayed trigger (see Section 13.3.2) or a delayed MLM call (see Section 12.2.4), the **triggertime** is the **eventtime** plus the delay time. Using **triggertime**, an MLM can trigger another MLM as if the second MLM were directly triggered by the event. The following inequality is guaranteed within a single MLM: **eventtime** ≤ **triggertime** ≤ **now**.

8.5 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 Section 9.11). For example, **1 day**, **45 seconds**, and **3.2 months** are durations.

8.5.1 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. For both types of durations, the duration amount may be a floating point value.

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 Section 9.8.

8.5.2 Time and Duration Arithmetic

Operations among times and durations are carried out as follows:

8.5.2.1 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**.

8.5.2.2 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,

1800-01-01T00:00:00) 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**.

8.5.2.3 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 computed through interpolation (the integer part of the months are added; then the fractional part is used on the next month for addition and on the previous month for subtraction). For example, **1991-01-31T00:00:00 + 1.1 months** results in **1991-02-28T00:00:00 + (0.1 * 2629746 seconds)** or **1991-03-03T01:02:54.6.** Explanation:

1991-01-31T00:00:00 + 1 month = 1991-02-28T00:00:00

and

0.1 Months * 2629746 seconds / month [from 8.5.2.4] = 262974.6 seconds

262974.6 seconds / (60 seconds / minute) / (1440 minutes /day) = 3.0436875 days

0.0436875 days * 1440 minutes / day = 62.91 minutes

= 1 hour, 2 minutes, 54.6 seconds.

therefore

0.1 months = 3 days 1 hours 2 minutes 54.6 seconds

thus

1991-01-31T00:00:00 + 1.1 months = 1991-02-28T00:00:00 + 3 days 1 hour 2 minutes 54.6 seconds

= 1991-03-03T01:02:54.6

Contrary to addition and subtraction on numbers, addition and subtraction of durations is not invertible. For example:

```
1993-01-31 + 1 month = 1993-02-28
1993-02-28 - 1 month  = 1993-01-28 (3 days earlier)
```

The order of operations is important: **(d+1 month)+1 day** may have a different value than **d+(1 month+1 day)**.

Other examples:

```
1991-01-31T00:00:00 - 2.1 months = 1990-11-27T00:00:00
1991-01-31T00:00:00 - 1.1 months = 1990-12-27T21:36:00
1991-04-30T00:00:00 - 0.1 months = 1991-04-27T00:00:00
```

8.5.2.4 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**.

8.6 String

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

```
"this is a string constant"
```

8.7 Term

Terms are currently used only to represent mlmnames within a structured slot and the link text portion of a structured link record. Mlmnames are used only in a **call** statement (see Section 10.2.4). In the future they will be used for controlled vocabulary terms. Term constants are defined in Section 7.1.7. For example,

```
'mlm_name2'
'http://www.nlm.nih.gov/'
```

8.8 List

A list is an ordered set of elements, each of which may be null, Boolean, event, destination, message, term, 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: **()**. White space is allowed within an empty list's parentheses. Other lists are created by using list operators like the comma (**,**) to build lists from single items (see Section 9.2). For the output format of lists (including single element lists), see Section 9.8. For example, these are valid lists:

```
4, 3, 5
3, true, 5, null
,1
()
```

8.9 Query Results

The result of a database query has a time value in addition to its data value.

Queries in the data slot retrieve data from the patient database or from other databases (for example, a controlled vocabulary database or a financial database). The result of a query is assigned to a variable for use in the other slots.

8.9.1 Primary Time

Every item in the patient database is assumed to have some primary time (also called time of occurrence) 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. If there is no medically relevant time for a data item, its primary time value should be equivalent to the **eventtime** (the time when the information was correct).

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 numeric value) and a time value (for example, when the specimen was drawn).

8.9.2 Retrieval Order

The result of a query is by default sorted in chronological order by the primary time of the result. The query may specify a different sort order.

8.9.3 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 latest potas > 5.0 then
conclude true;
endif;
```

8.9.4 Time Function Operator

By using the **time** operator (see Section 9.17), one can set or retrieve the primary time associated with a variable or list element. The time retrieve function is describe in Section 9.17.1. Setting primary times is discussed in the second paragraph of Section 9.17.1. For example, one could use this statement to check the primary time of the most recent potassium measurement:

```
if time of latest potas is within the past 3 days then
conclude true;
endif ;
```

The **eventtime** is not necessarily the primary time of the evoking event. 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:

```
 <num:type> := <num:type> op <num:type>
```

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

9.1.2.1

Each **num** is one of the following:

1—the operator requires a single element

k, **m**, or **n**—the operator normally takes a single element but a list with 0, 1, or more elements may be used as described below. If the same letter appears more than once in a data type constraint, then the arguments so indicated must have the same number of elements; otherwise the operation results in **null**. Refer to Section 9.1.3.4 regarding the replication of the single element **n** times.

9.1.2.2

Each **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)

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

<num:type>(s) to the right of the **:=** indicates the data type(s) of the argument(s). 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

<num: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.

9.1.3 List Handling

Except as otherwise stated, lists are treated as follows.

9.1.3.1

When an operator has a template of the form **<n:type> := op <n:type>** or **<n:type> := <n:type> op**, the scalar operator is applied to each element of the list, producing a list with the same number of elements (if the list is empty, the resulting list is also empty). For example, **-(3,4,5)** results in **-3, -4, -5**.

Unary operators that act this way are:

```
not …
… is present
… is not present
… is null
… is not null
… is Boolean
… is not Boolean
… is number
… is not number
… is time
… is not time
… is duration
… is not duration
… is string
… is not string
+ …
- …
… ago
… year
… years
… month
… months
… day
… days
… hour
… hours
… minute
… minutes
… second
… seconds
time [of] …
arccos [of] …
arcsin [of] …
arctan [of] …
cos [of] …
cosine [of] …
sin [of] …
sine [of] …
tan [of] …
tangent [of] …
exp [of] …
truncate [of] …
floor [of] …
ceiling [of] …
log [of] …
log10 [of] …
abs [of] …
year [of] …
month [of] …
```

```
day [of] …
hour [of] …
minute [of] …
second [of] …
sort data …
sort time …
reverse …
```

9.1.3.2

When an operator has a template of the form **<1:type> := op <n:type>** or **<1:type> := <n:type> op**, the operator is applied to the entire list, producing a single element. For example, **max(3,4,5)** results in **5**.

Unary operators that act this way are:

```
count [of] …
exist [of] …
avg [of] …
average [of] …
median [of] …
sum  [of] …
stddev [of] …
variance [of] …
any [of] …
all [of] …
no [of] …
min [of] …
minimum [of] …
max [of] …
maximum [of] …
last [of] …
first [of] …
earliest [of] …
latest [of] …
string [of] …
… is list
… is not list
index min [of] …
index minimum [of] …
index max [of] …
index maximum [of] …
index earliest [of] …
index latest [of] …
```

9.1.3.3

When an operator has a template of the form **<m:type> := op <n:type>** or **<m:type> := <n:type> op**, the operator is applied to the entire list, producing another list. For example, **increase(11,15,13,12)** results in **(4, -2, -1)**.

Unary operators that act this way are:

```
slope [of] …
increase [of] …
decrease [of] …
percent increase [of] …
% increase [of] …
percent decrease [of] …
% decrease [of] …
interval [of] …
extract characters [of] …
```

9.1.3.4

When an operator has a template of the form **<n:type> := <n:type> op <n:type>**, the scalar operator is applied pair-wise to the elements of the lists, producing a list with the same number of elements (if the list is empty, the resulting list is also empty). For example, **(1,2)+(3,4)** results in (**4,6)** and **()+()** results in **()**.

If one of the operands is a single element and the other operand has n elements, the single element is replicated n times. For example, **1+(3,4)** is equivalent to **(1,1)+(3,4)** and results in (**4,5)**.

If the number of elements in the two arguments differ and one argument is not a single element, the result is **null**.

Binary operators that act this way are:

```
… or …
… and …
… = …
… eq …
… is …
… <> …
… ne …
… is not equal …
… < …
… lt …
… is less than …
… is not greater than or equal …
… <= …
… le …
… is less than or equal …
… is not greater than …
… > …
… gt …
… is greater than …
… is not less than or equal …
… >= …
… ge …
… is greater than or equal …
… is not less than …
… is within past …
… is not within past …
… is within same day as
… is not within same day as …
… is before …
… is not before …
… is after …
… is not after …
… matches pattern …
… occur equal …
… occur within past …
… occur not within past …
… occur within same day as …
… occur not within same day as …
… occur before …
… occur not before …
… occur after …
… occur not after …
… + …
… - …
… * …
… / …
… ** …
… before …
… after …
… round …
```

The following operators are of the form **<n:type> := <m:type> op <m:type>**; they replicate the arguments if necessary but may return a list with a different number of elements:

```
… where …
```

9.1.3.5

When an operator has a template of the form **<n:type> := <n:type> op$_1$ <n:type> op$_2$ <n:type>**, the scalar operator is applied triple-wise to each element of the lists, producing a list with the same number of elements (if the list is empty, the resulting list is also empty). For example, **(1,2) is within (0,2) to (3,4)** results in (**true,true)**.

If one of the operands is a single element and the other operands have n elements, the single element is replicated n times. If two of the operands are a single element and the other operand has n elements, the single elements are replicated n times. For example, **(1,2) is within 2 to (3,4)** is equivalent to **(1,2) is within (2,2) to (3,4)** and results in **(false,true)**.

If the number of elements in any pair of arguments differ and one argument is not a single element, the result is **null**.

Ternary operators that act this way are:

```
… is within … to …
… is not within … to …
… is within … preceding …
… is not within … preceding …
… is within … following …
… is not within … following …
… is within … surrounding …
… is not within … surrounding …
… occur within … to …
… occur not within … to …
… occur within … preceding …
… occur not within … preceding …
… occur within … following …
… occur not within … following …
… occur within … surrounding …
… occur not within … surrounding …
```

9.1.3.6

When an operator has a template of the form **<n:type> := op$_1$ <1:type> op$_2$ <m:type>**, the operator is applied to the entire second argument, producing a new list. The first argument must be a single element (if not, the result of the operator is **null**). For example, **min 2 from (5,3,4)** results in (**3, 4)**.

Binary operators that act this way are:

```
min … from …
minimum … from …
max …  from …
maximum … from …
last … from …
first … from …
latest … from …
earliest … from …
index min … from …
index minimum … from …
index max …  from …
index maximum … from …
index latest … from …
index earliest … from …
```

9.1.3.7

When an operator has a template of the form **<n:type> := op$_1$ <n:type> op$_2$ <m:type>**, the operator is applied to the entire second argument, producing a new list. The first argument is typically a single element. For example, **1 is in (0,3)** results in **false** and **(1,2,3) is in (0,3)** results in (**false,false,true)**.

Binary operators that act this way are:

```
nearest … from …
… is in …
… is not in …
index nearest … from …
```

9.1.3.8

When an operator has a template of the form **<n:type> := <k:type> op <m:type>**, the operator is applied to the entire two lists, producing a new list. For example, **1,(3,4)** results in (**1,3,4)**.

Binary operators that act this way are:

```
… , …
… merge …
… || …
… seqto …
```

9.1.4 Primary Time Handling

Queries attach primary times to their results (see Sections 8.9.1). 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

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

Binary and ternary operators maintain primary times if all operands have primary times and all of the primary times are equal. If any operand is missing a primary time or if the primary times are not all equal, the primary time is lost.

Example (primary times are the same, the primary time is kept):

Data Values: 6 := 2 * 3;

Time Values: (Jan 1) (Jan 1) (Jan 1);

Example (primary times are different, then primary time is lost):

Data Values: 42 := 6 * 7;

Time Values: (null) (Feb 1) (Jan 1);

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**. Parentheses can always be used to override operator precedence.

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. The associativity of each operator is shown in Annex **A4**. 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 (however, **(2**3)**4** and **2**(3**4)** are both legal).

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. Primary times are maintained according to Section 9.1.4.

9.2.1 **,** (binary, left associative)

Binary **,** (list concatenation) appends two lists. Primary times are maintained in 9.2.1. Its usage is:

```
<n:any-type> := <k:any-type> , <m:any-type>
(4,2) := 4, 2
(4,"a",null) := (4,"a") , null
```

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. Its usage is (where **(3)** means a list with 3 as its only element):

```
<1:any-type> := , <1: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 (as defined in 9.2.4). All elements of both lists must have primary times; otherwise **null** is returned (the construct **`x where time of it is present`** can be used to select only elements of **`x`** that have primary times). The primary times are maintained. **Merge** is typically used to put together the results of two separate queries. The expression **x merge y** is equivalent to **sort time (x,y)**. 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 data values **1,3** and time values **1991-01-01T00:00:00**, **1991-01-03T00:00:00**):

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

9.2.4 Sort Operators (unary, non-associative)

The sort operators (**sort data** and **sort time**) reorder a list based on element keys, which are either the element values (**sort data**) or the primary times (**sort time**). Direction of sorting is always ascending. For a descending sort, **reverse** can be used.

When sorting by primary times, if any of the elements do not have primary times, the result is **null.** (The sort argument can always be qualified by **where time of it is present**, if this is not desired behavior.) Elements with the same key will be kept in the same order as they appear in the argument. If any pair of element key cannot not be compared because of type clashes, **sort** returns **null** (that is, when sorting by data, any null value (or non-comparable value) results in **null**; when sorting by time, any null primary time results in **null**). Its usage is (assuming that **data1** has a data value of **30,10,20** with time values **1991-01-01T00:00:00**, **1991-02-01T00:00:00**, **1991-01-03T00:00:00**):

```
<n:any-type> := SORT DATA <n:any-type>
<n:any-type> := SORT TIME <n:any-type>
 (10, 20, 30) := SORT DATA data1
(30, 20, 10) := REVERSE SORT DATA data1
null := SORT DATA (3,1,2,null)
null := SORT DATA (3,"abc")
() := SORT TIME ()
(1, 2, 3, 3) := SORT DATA (1,3,2,3)
(30, 20, 10) := SORT TIME data1
```

9.3 Where Operator

The **where** operator does not follow the default list handling or the default time 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. The **where** operator maintains the primary time(s) of the operand(s) to the left of **WHERE**. The primary time(s) of the operand(s) to the right of **WHERE** are dropped. Its usage is:

```
<n:any-type> := <m:any-type> WHERE <m:any-type>
(10,30) := (10,20,30,40) WHERE (true,false,true,3)
```

Example

7.38 := (7.34, 7.38, 7.4) WHERE time of it is within 20 minutes after time of VentChange

(1/1 16:20) (1/1 18:01) (1/1 16:20) (Jan 1 02:06) (Jan 1 16:12)

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 (the rules in Section 9.1.3.4 are used to replicate a single-element argument if necessary). 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)
```

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.

Where can be used to filter out invalid data. For example, if a query returns either numeric values or text comments, the following can be used to select elements from the query that have proper numeric values:

```
queryResult where they are number
```

Similarly, if a query returns some values without primary times, the following can be used to select elements from the query that have proper primary times:

```
queryResult where time of it is present
```

In this example, the unary operator **time** is applied to the queryResult (which is what the value of "**it**" is), resulting in a list of times (for those results that have a primary time) and nulls (for those results that do not have a primary time). The unary operator **is present** is then applied to that list, give a list of Booleans: true where there is a primary time and false where there is no primary time. Finally, the **where** operator is used to remove those values that do not have primary times.

9.3.1.1 It

The word **it** and 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**. An implementation of the Arden Syntax may choose to flag use of **it** outside a **where** expression as an error at compile time.

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:

```
<n:Boolean> := <n:any-type> OR <n:any-type>
true := true OR false
false := false OR false
true := true OR null
null := false OR null
null := false OR 3.4
(true, true) := (true, false) OR (false, true)
() := () OR ()
```

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:

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

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

	AND	TRUE	FALSE	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:

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

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. Primary times are not used in determining equality; the primary time of the result is determined by the rules in Section 9.1.4. Its usage is:

```
<n:Boolean> := <n:non-null> = <n:non-null>
false := 1 = 2
(null,true,false) := (1,2,"a") = (null,2,3)
null := (3/0) = (3/0)
null := 5 = ()
null := (1,2,3) = ()
```

```
() := null = ()
() := () = ()
null := 5 = null
(null,null,null) := (1,2,3) = null
null := null = null
(true,true,false) := (1,2,3) = (1,2,4)
```

Use **is present** or **exists** instead of = to test whether an argument is equal to **null**. See Sections 9.6.15 and 9.12.3.

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:

```
<n:Boolean> := <n:non-null> <> <n:non-null>
true := 1 <> 2(null,false,true) := (1,2,"a") <> (null,2,3)
null := (3/0) <> (3/0)
```

9.5.3 **<** (binary, non-associative)

The < operator has three synonyms: **lt**, **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:

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

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:

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

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:

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

```
null := "aaa" > 1
```

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:

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

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 (using the definition of **`not`**, see Section 9.4.3). For example, these are valid:

```
surgery_time WAS BEFORE discharge_time
surgery_time IS NOT AFTER discharge_time
```

9.6.1 Is [not] Equal (binary, non-associative)

See Section 9.5.1.

9.6.2 Is [not] Less Than (binary, non-associative)

See Section 9.5.3.

9.6.3 Is [not] Greater Than (binary, non-associative)

See Section 9.5.5.

9.6.4 Is [not] Less Than or Equal (binary, non-associative)

See Section 9.5.4.

9.6.5 Is [not] Greater Than or Equal (binary, non-associative)

See Section 9.5.6.

9.6.6 Is [not] 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:

```
<n:Boolean> := <n:ordered> IS WITHIN <n:ordered> TO <n: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 [not] 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:

```
<n:Boolean> := <n:time> IS WITHIN <n:duration> PRECEDING <n:time>
true := 1990-03-08T00:00:00 IS WITHIN 3 days PRECEDING 1990-03-10T00:00:00
```

9.6.8 Is [not] 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:

```
<n:Boolean> := <n:time> IS WITHIN <n:duration> FOLLOWING <n:time>
false := 1990-03-08T00:00:00 IS WITHIN 3 days FOLLOWING 1990-03-10T00:00:00
```

9.6.9 Is [not] 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:

```
<n:Boolean> := <n:time> IS WITHIN <n:duration> SURROUNDING <n:time>
true := 1990-03-08T00:00:00 IS WITHIN 3 days SURROUNDING 1990-03-10T00:00:00
```

9.6.10 Is [not] 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):

```
<n:Boolean> := <n:time> IS WITHIN PAST <n:duration>
true := 1990-03-08T00:00:00 IS WITHIN PAST 3 days
```

9.6.11 Is [not] 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:

```
<n:Boolean> := <n:time> IS WITHIN SAME DAY AS <n:time>
true := 1990-03-08T11:11:11 IS WITHIN SAME DAY AS 1990-03-08T01:01:01
```

9.6.12 Is [not] 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:

```
<n:Boolean> := <n:time> IS BEFORE <n: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 [not] 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:

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

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

9.6.14 Is [not] 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. Primary times are retained only if they match (that is, the = operator is used for determining membership, except that **null** will match). Its usage is:

```
<n:Boolean> := <n:any-type> IS IN <m:any-type>
false := 2 IS IN (4,5,6)
(false,true) := (3,4) IS IN (4,5,6)
true := null is in (1/0,2)
```

9.6.15 Is [not] 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**. This operator is used to test whether an argument is **null** since **arg=null** always results in **null** regardless of **arg**. Its usage is:

```
<n:Boolean> := <n:any-type> IS PRESENT
true := 3 IS PRESENT
false := null IS PRESENT
(true,false) := (3,null) IS PRESENT
(false,true) := (3,null) IS NULL
```

9.6.16 Is [not] Null (unary, non-associative)

See Section 9.6.15.

9.6.17 Is [not] 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:

```
<n:Boolean> := <n:any-type> IS BOOLEAN
true := false IS BOOLEAN
true := 3 IS NOT BOOLEAN
(false,true,false) := (null,false,3) IS BOOLEAN
```

9.6.18 Is [not] 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:

```
<n:Boolean> := <n:any-type> IS NUMBER
true := 3 IS NUMBER
false := null IS NUMBER
```

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 [not] 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:

```
<n:Boolean> := <n:any-type> IS STRING
true := "asdf" IS STRING
false := null IS STRING
```

9.6.20 Is [not] 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:

```
<n:Boolean> := <n:any-type> IS TIME
true := 1991-03-12T00:00:00 IS TIME
false := null IS TIME
```

9.6.21 Is [not] 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:

```
<n:Boolean> := <n:any-type> IS DURATION
true := (3 days) IS DURATION
false := null IS DURATION
```

9.6.22 Is [not] List (unary, non-associative)

The **is list** operator returns **true** if the argument is a list. Otherwise it returns **false**. **Is list** never returns **null**. Its usage is:

```
<1:Boolean> := <n:any-type> IS LIST
true := (3, 2, 1) IS LIST
False := 5 IS LIST
false := null IS LIST
```

The **is list** operator does not follow the default list handling because it does not operate on each item in the argument, but rather operates on the argument as a whole. Thus it never returns a list. Notice the difference:

```
true := (3, 2, "asdf") IS LIST
(true, true, false) := (3, 2, "asdf") IS NUMBER
```

9.7 Occur Comparison Operators:

9.7.1 General Properties:

The following comparison operators are analogous to the **is** comparison operators in Section 9.6. They use the word **occur** instead of **is**. The word **occur** can be replaced with **occurs** or **occurred**. An optional **not** may follow the **occur**, negating the result (using the definition of **not**, see Section 9.4.3).

The effect is that rather than using the left argument directly, the primary time of the left argument is used instead (that is, the **time** of the left argument is used; see Section 9.17). The following pairs are equivalent expressions:

```
time of var IS NOT BEFORE 1990-03-05T11:11:11
```

```
var OCCURRED NOT BEFORE 1990-03-05T11:11:11

time of surgery IS WITHIN THE PAST 3 days
surgery OCCURRED WITHIN THE PAST 3 days

time(a) IS WITHIN 1990-03-05T11:11:11 TO time(b)
a OCCURRED WITHIN 1990-03-05T11:11:11 TO time(b)
```

In the following operator examples, **data** is the result of a query; its primary time is 1990-03-05T11:11:11; and **now** is 1990-03-06T00:00:00.

9.7.2 Occur [not] Equal (binary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR EQUAL <n:time>
false := data OCCURED EQUAL 1990-03-01T00:00:00
```

9.7.3 Occur [not] Within ... To (ternary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN <n:time> TO <n:time>
true := data OCCURED WITHIN 1990-03-01T00:00:00 TO 1990-03-11T00:00:00
```

9.7.4 Occur [not] Within ... Preceding (ternary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN <n:duration> PRECEDING <n:time>
false := data OCCURRED WITHIN 3 days PRECEDING 1990-03-10T00:00:00
```

9.7.5 Occur [not] Within ... Following (ternary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN <n:duration> FOLLOWING <n:time>
false := data OCCURRED WITHIN 3 days FOLLOWING 1990-03-10T00:00:00
```

9.7.6 Occur [not] Within . . . Surrounding (ternary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN <n:duration> SURROUNDING <n:time>
false := data OCCURED WITHIN 3 days SURROUNDING 1990-03-10T00:00:00
```

9.7.7 Occur [not] Within Past (binary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN PAST <n:duration>
true := data OCCURED WITHIN PAST 3 days
```

9.7.8 Occur [not] Within Same Day As (binary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR WITHIN SAME DAY AS <n:time>
false := data OCCURED WITHIN SAME DAY AS 1990-03-08T01:01:01
```

9.7.9 Occur [not] Before (binary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR BEFORE <n:time>
true := data OCCURRED BEFORE 1990-03-08T01:01:01
```

9.7.10 Occur [not] After (binary, non-associative):

```
<n:Boolean> := <n:any-type> OCCUR AFTER <n:time>
false := data OCCURED AFTER 1990-03-08T01:01:01
```

9.8 String Operators

The string operators do not follow the default list handling or the default primary time handling.

9.8.1 || (binary, left associative)

The || operator (string concatenation) converts its arguments to strings and then concatenates those strings together. The null data type is converted to the string **null** and then appended to the other argument. Thus || never returns **null**. Lists are converted to strings and then appended to the other argument; the list is enclosed in parentheses and the elements are separated by **,** with no separating blanks. The string representation of Booleans, numbers, times, and durations is location-specific to allow for the use of the native language. The **formatted with** operators **%z** operator is used to convert values to strings (see Section 9.8.2). The **string** operator is a generalization of the || operator (see Section 9.8.3), except that the **string** operator does not do anything special for lists. The primary times of its arguments are lost. Its usage is:

```
<1:string> := <m:any-type> || <n:any-type>
"null3" := null || 3
"45" := 4 || 5
"4.7four" := 4.7 || "four"
"true" := true || ""
"3 days left" := 3 days || " left"
"on 1990-03-15T13:45:01" := "on " || 1990-03-15T13:45:01
"list=(1,2,3)" := "list=" || (1,2,3)
```

9.8.2 Formatted with (binary, left-associative)

The **formatted with** operator allows a formatting string to be used for additional control over how data items are output. The formatting string is similar to the ANSI C language printf control string, with additional ability to format an Arden time. The primary times of its arguments are lost.

```
<string> := <data> formatted with <format_string>

"01::02::03" := (1,2,3) formatted with "%2.2d::%2.2d::%2.2d"

"The result was 10.61 mg"
:= 10.60528 formatted with "The result was %.2f mg"

"The date was Jan 10 1998"
:= 1998-01-10T17:25:00 formatted with "The date was %.2t"

"The year was 1998"
:= 1998-01-10T17:25:00 formatted with "The year was %.0t"

/* longer example */
a := "ten";
b := "twenty";
c := "thirty";
f := "%s, %s, %s or more";
"ten, twenty, thirty or more" := (a, b, c) formatted with f;
```

If **data** is a single item, it serves as the single parameter for format string substitution. If **data** is a list, the list is not formatted as a list. Instead, it is assumed to be a list of parameters for format string substitution. Parameters are substituted into the **format string** as described below, which becomes the result of the operation.

A format string consists of a literal string and typically contains 1 or more format specifications.

A format specification, which consists of optional and required fields, has the following form:

```
%[flags][width][.precision]type
```

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, %s). If a percent sign is followed by a character that has no meaning as a format field, the character is not revised. For example, to print a percent-sign character, use %%.

Note that to retain compatibility with C language functions, several formatting type specifiers have been retained that will probably not be useful to the Arden MLM author. The most likely format specification types an MLM author will use are:

```
%c     (for outputting special characters)
%s     (string width control)
%d     (integer formatting)
%t     (time formatting)
%e     (floating point number formatting with exponent)
%f     (floating point number formatting without exponent)
%g     (floating point number formatting using %e or %f)
```

A complete description of supported types within the format specification can be found in Annex A5.

9.8.3 String ... (unary, right associative)

The **string** operator expects a string or list of strings as its argument. It returns a single string made by concatenating all the elements, as the || operator (see Section 9.8.1). If the argument is an empty list, the result is the empty string (""). The element operator (Section 9.12.18) can be used to select certain items from the list. The primary times of its arguments are lost. Its usage is:

```
<1:string> := STRING <m:any-type>
"abc" := STRING ("a","b","c")
"abc" := STRING ("a","bc")
"" := STRING ()
"edcba" := STRING REVERSE EXTRACT CHARACTERS "abcde"
```

9.8.4 Matches pattern (binary, non-associative)

The effect of this operator is similar to the LIKE operator in SQL (ISO / IEC 9075). **Matches pattern** is used to determine whether or not a particular string matches a pattern. This operator expects two string arguments. The first argument is a string to be matched, and the second is the pattern used for matching. **Matches pattern** returns a Boolean value: true if the pattern of the second argument matches the first argument and false if it does not. The first argument also may be a list of strings, in which case the result is a list of Boolean values, each corresponding to the match between one string and the pattern of the second argument. If the arguments are not strings, null is returned. Matching is case-insensitive. The primary time of the arguments are lost.

The pattern of the second argument may be any legal string character. In addition, two wild-card characters may be used. The underscore (_) will match exactly any one character. The percent sign (**%**) will match 0 to arbitrarily many characters. In order to match one of the literal wild-card character, precede it with an escape (\) character.

```
<n : Boolean> := <k : string> MATCHES PATTERN <m : string>
<n : list of Boolean> := <k : list of strings> MATCHES PATTERN <m : string>
true := "fatal heart attack" MATCHES PATTERN "%heart%";
false := "fatal heart attack" MATCHES PATTERN "heart";
true := "abnormal values" MATCHES PATTERN "%value_";
false := "fatal pneumonia" MATCHES PATTERN "%pulmonary%";
(true, false) := ("stunned myocardium", "myocardial infarction") MATCHES
    PATTERN
"%myocardium";
true := "5%" MATCHES PATTERN "_\%";
```

9.9 Arithmetic Operators

The behavior of time and duration data types is explained in Section 8.5.2.

9.9.1 **+** (binary, left associative)

Binary **+** (addition) adds the left and right arguments. It can perform simple addition, add two durations, or increment a time by a duration. Underflow or overflow results in **null**. Its usage is:

```
<n:number> := <n:number> + <n:number>
   6 := 4 + 2
   () := 5 + ()
   null := (1,2,3) + ()
   () := null + ()
   null := 5 + null
   (null,null,null) := (1,2,3) + null
   null := null + null
<n:duration> := <n:duration> + <n:duration>
   3 days := 1 day + 2 days
<n:time> := <n:time> + <n:duration>
   1990-03-15T00:00:00 := 1990-03-13T00:00:00 + 2 days
   1993-05-17T00:00:00 := 0000-00-00 + 1993 years + 5 months + 17 days
<n:time> := <n:duration> + <n:time>
   1990-03-15T00:00:00 := 2 days + 1990-03-13T00:00:00
```

9.9.2 **+** (unary, non-associative)

Unary **+** has no effect on its argument if it is of a valid type. Its usage is:

```
<n:number> := + <n:number>
   2 := + 2
   null := + "asdf"
<n:duration> := + <n:duration>
   2 days := + 2 days
```

9.9.3 **-** (binary, left associative)

Binary **-** (subtraction) subtracts the right argument from the left. It can perform numeric subtraction, subtract two durations, decrement a time by a duration, or find the duration between two times. Underflow or overflow results in **null**. In writing expressions, care must be taken that the subtraction operator is not confused with the "**-**"" in time constant (Section 7.1.5). Any ambiguity is resolved in favor of time constants. Its usage is:

```
<n:number> := <n:number> - <n:number>
   4 := 6 - 2
<n:duration> := <n:duration> - <n:duration>
   1 day := 3 days - 2 days
<n:time> := <n:time> - <n:duration>
   1990-03-13T00:00:00 := 1990-03-15T00:00:00 - 2 days
<n:duration> := <n:time> - <n:time>
   2 days := 1990-03-15T00:00:00 - 1990-03-13T00:00:00
```

9.9.4 **-** (unary, non-associative)

Unary **-** is used for arithmetic negation; this is how one makes negative number constants. Underflow or overflow results in **null**. One cannot put two arithmetic operators together, so the following expression is illegal: **3 + -4**. Instead one must use one of these: **3 + (-4)**, **3 - 4**, or **-4 + 3**. Its usage is:

```
<n:number> := - <n:number>
   (-2) := - 2
<n:duration> := - <n:duration>
   (-2) days := - (2 days)
```

9.9.5 ***** (binary, left associative)

The ***** operator (multiplication) multiplies the left and right arguments. Underflow or overflow results in **null**. It can perform numeric multiplication or multiply a duration by a number. Its usage is:

```
<n:number> := <n:number> * <n:number>
   8 := 4 * 2
<n:duration> := <n:number> * <n:duration>
   6 days := 3 * 2 days
<n:duration> := <n:duration> * <n:number>
   6 days := 2 days * 3
```

9.9.6 **/** (binary, left associative)

The / operator (division) divides the left argument by the right one. It can perform numeric division, divide a duration by a number, or find the ratio between two durations. **Null** results from division by zero, underflow, or overflow. Duration unit conversion can be done with the / operator (e.g., **… / 1 year** turns any duration into years). Its usage is:

```
<n:number> := <n:number> / <n:number>
   4 := 8 / 2
<n:duration> := <n:duration> / <n:number>
   2 days := 6 days / 3
<n:number> := <n:duration> / <n:duration>
   120 := 2 minutes / 1 second
```

```
      36 := 3 years / 1 month
```

9.9.7 ** (binary, non-associative)

The ****** operator (exponentiation) raises the left argument to the power of the right argument. Its usage is:

```
   <n:number> := <n:number> ** <n:number>
      9 := 3 ** 2
```

9.10 Temporal Operators

The behavior of time and duration data types is explained in Section 8.5.2.

9.10.1 After (binary, non-associative)

The **after** operator is equivalent to addition between a duration and a time. Its usage is:

```
   <n:time> := <n:duration> AFTER <n:time>
      1990-03-15T00:00:00 := 2 days AFTER 1990-03-13T00:00:00
```

9.10.2 Before (binary, non-associative)

The **before** operator is equivalent to the subtraction of a duration from a time. Its usage is:

```
   <n:time> := <n:duration> BEFORE <n:time>
      1990-03-11T00:00:00 := 2 days BEFORE 1990-03-13T00:00:00
```

9.10.3 Ago (unary, non-associative)

The **ago** operator subtracts a duration from **now**, resulting in a time. Its usage is (assuming that **now** is 1990-04-19T00:03:15):

```
   <n:time> := <n:duration> AGO
      1990-04-17T00:03:15 := 2 days AGO
```

9.11 Duration Operators

The behavior of the duration data type is explained in Section 8.5.2. Because the precedence of the temporal operators is lower than that of the duration operators, **3 hours before 3 days ago** is parsed as **(3 hours) before ((3 days) ago)**, and it would return what time it was three days and three hours before the current time.

9.11.1 Year (unary, non-associative)

The **year** operator has one synonym: **years**. It creates a months duration from a number: one year is 12 months. Its usage is:

```
   <n:duration> := <n:number> YEAR
      24 months := 2 YEAR
```

9.11.2 Extract year (unary, right-associative)

The **extract year** operator extracts the year from a time. Its usage is:

```
   <n:number> := EXTRACT YEAR <n:time>
      1990 := EXTRACT YEAR 1990-01-03T14:23:17.3
      null := EXTRACT YEAR (1 YEAR)
```

9.11.3 Month (unary, non-associative)

The **month** operator has one synonym: **months**. It creates a months duration from a number. Its usage is:

```
<n:duration> := <n:number> MONTH
```

9.11.4 Extract month (unary, right-associative)

The **extract month** operator extracts the month from a time. Its usage is:

```
<n:number> := EXTRACT MONTH <n:time>
   1 := EXTRACT MONTH 1990-01-03T14:23:17.3
   null := EXTRACT MONTH 1
```

9.11.5 Week (unary, non-associative)

The **week** operator has one synonym: **weeks**. It creates a seconds duration from a number: one week is 604800 seconds. Its usage is:

```
<n:duration> := <n:number> WEEK
```

9.11.6 Day (unary, non-associative)

The **day** operator has one synonym: **days**. It creates a seconds duration from a number: one day is 86400 seconds. Its usage is:

```
<n:duration> := <n:number> DAY
```

9.11.7 Extract day (unary, right-associative)

The **extract day** operator extracts the day from a time. Its usage is:

```
<n:number> := EXTRACT DAY <n:time>
   3 := EXTRACT DAY 1990-01-03T14:23:17.3
   null := EXTRACT DAY "this is not a time"
```

9.11.8 Hour (unary, non-associative)

The **hour** operator has one synonym: **hours**. It creates a seconds duration from a number: one hour is 3600 seconds. Its usage is:

```
<n:duration> := <n:number> HOUR
```

9.11.9 Extract hour (unary, right-associative)

The **extract hour** operator extracts the hour from a time. Its usage is:

```
<n:number> := EXTRACT HOUR <n:time>
   14 := EXTRACT HOUR 1990-01-03T14:23:17.3
   null := EXTRACT HOUR (1 HOUR)
```

9.11.10 Minute (unary, non-associative)

The **minute** operator has one synonym: **minutes**. It creates a seconds duration from a number: one minute is 60 seconds. Its usage is:

```
<n:duration> := <n:number> MINUTE
```

9.11.11 Extract minute (unary, right-associative)

The **extract minute** operator extracts the minute from a time. Its usage is:

```
<n:number> := EXTRACT MINUTE <n:time>
   23 := EXTRACT MINUTE 1990-01-03T14:23:17.3
   0 := EXTRACT MINUTE 1990-01-03
   null := EXTRACT MINUTE 0000-00-00
```

9.11.12 Second (unary, non-associative)

The **second** operator has one synonym: **seconds**. It creates a seconds duration from a number. Its usage is:

```
<n:duration> := <n:number> SECOND
```

9.11.13 Extract second (unary, right-associative)

The **extract second** operator extracts the second from a time. Its usage is:

```
<n:number> := EXTRACT SECOND <n:time>
   17.3 := EXTRACT SECOND 1990-01-03T14:23:17.3
   null := EXTRACT SECOND (1 second)
```

9.12 Aggregation Operators:

9.12.1 General Properties:

The aggregation operators do not follow the default list handling, or the default primary time handling. They perform aggregation on a list. That is, they take a list as an argument (they are all unary) and return a single item as a result. Unless otherwise noted, if all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). An argument that is a single item is treated as a list of length one.

Each of the operators may be followed by the word **of**. Parentheses are not required. For example, these are all the same:

```
SUM a_list
SUM OF a_list
SUM(a_list)
SUM OF(a_list)
```

Multiple aggregation and transformation operators (for example, see Section 9.14) may be placed in an expression without parentheses; for example:

```
AVERAGE OF LAST 3 FROM a_list
```

9.12.2 Count (unary, right associative)

The **count** operator returns the number of items (including null items) in a list. **Count** never returns **null**. The result loses the primary time. Its usage is:

```
<1:number> := COUNT <n:any-type>
   4 := COUNT (12,13,14,null)
   1 := COUNT "asdf"
   0 := COUNT ()
   1 := COUNT null
```

9.12.3 Exist (unary, right associative)

The **exist** operator has one synonym: **exists**. It returns **true** if there is at least one non-null item in a list of any type. **Exist** never returns **null**. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:Boolean> := EXIST <n:any-type>
   true := EXIST (12,13,14)
   false := EXIST null
   false := EXIST ()
   true := EXIST ("plugh",null)
```

9.12.4 Average (unary, right associative)

The **average** operator has one synonym: **avg**. It calculates the average of a number, time, or duration list. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:number> := AVERAGE <n:number>
   14 := AVERAGE (12,13,17)
   3 := AVERAGE 3
null := AVERAGE ()
   <1:time> := AVERAGE <n:time>
1990-03-11T03:10:00 := AVERAGE (1990-03-10T03:10:00, 1990-03-12T03:10:00)
   <1:duration> := AVERAGE <n:duration>
   3 days := AVERAGE (2 days, 3 days, 4 days)
```

9.12.5 Median (unary, right associative)

The **median** operator calculates the median value of a number, time, or duration list. The list is first sorted. If there is an odd number of items, it selects the middle value. If there is an even number of items, it averages the middle two values. If there is a tie, then it selects the latest of those elements that have a primary time. If a single element is selected or if the two selected elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:number> := MEDIAN <n:number>
   13 := MEDIAN (12,17,13)
   3 := MEDIAN 3
   null := MEDIAN ()
<1:time> := MEDIAN <n:time>
   1990-03-11T03:10:00 = MEDIAN (1990-03-10T03:10:00, 1990-03-11T03:10:00,
    1990-03-28T03:10:00)
<1:duration> := MEDIAN <n:duration>
   3 days := MEDIAN (1 hour, 3 days, 4 years)
```

9.12.6 Sum (unary, right associative)

The **sum** operator calculates the sum of a number or duration list. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:number> := SUM <n:number>
   39 := SUM (12,13,14)
   3 := SUM 3
   0 := SUM ()
```

```
<1:duration> := SUM <n:duration>
   7 days := SUM (1 day, 6 days)
```

9.12.7 stddev (unary, right associative)

The **stddev** operator returns the sample standard deviation of a numeric list. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:number> := STDDEV <n:number>
   1.58113883 := STDDEV (12,13,14,15,16)
   null := STDDEV 3
   null := STDDEV ()
```

9.12.8 Variance (unary, right associative)

The **variance** operator returns the sample variance of a numeric list. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:number> := VARIANCE <n:number>
   2.5 := VARIANCE (12,13,14,15,16)
   null := VARIANCE 3
   null := VARIANCE ()
```

9.12.9 Minimum (unary, right associative)

The **minimum** operator has one synonym: **min**. It returns the smallest value in a homogeneous list of an ordered type (that is, all numbers, all times, all durations, or all strings), using the <= operator (see Section 9.5.4). If there is a tie, it selects the element with the latest primary time. The primary time of the selected argument is maintained. Its usage is:

```
<1:ordered> := MINIMUM <n:ordered>
   12 := MINIMUM (12,13,14)
   3 := MIN 3
   null := MINIMUM ()
   null := MINIMUM (1,"abc")
```

9.12.10 Maximum (unary, right associative)

The **maximum** operator has one synonym: **max**. It returns the largest value in a homogeneous list of an ordered type, using the >= operator (see Section 9.5.6). If there is a tie, it selects the element with the latest primary time. The primary time of the selected argument is maintained. Its usage is:

```
<1:ordered> := MAXIMUM <n:ordered>
   14 := MAXIMUM (12,13,14)
   3 := MAXIMUM 3
   null := MAXIMUM ()
   null := MAXIMUM (1,"abc")
```

9.12.11 Last (unary, right associative)

The **last** operator returns the value at the end of a list, regardless of type. If the list is empty, **null** is returned. The expression **last x** is equivalent to **x[count x]**. **Last** on the result of a time-sorted query will return the most recent value. The primary time of the selected argument is maintained. Note that **last** is different than **last** specified in Arden Syntax version E 1460-92. That operator is now called **latest** (see Section 9.12.16). Its usage is:

```
<1:any-type> := LAST <n:any-type>
   14 := LAST (12,13,14)
   3 := LAST 3
   null := LAST ()
```

9.12.12 First (unary, right associative)

The **first** operator returns the value at the beginning of a list. If the list is empty, **null** is returned. The expression **`first x`** is equivalent to **`x[1]`**. **First** on the result of a time-sorted query will return the earliest value. The primary time of the selected argument is maintained. Note that **first** is different than **first** specified in Arden Syntax version E 1460-92. That operator is now called **earliest** (see Section 9.12.17). Its usage is:

```
<1:any-type> := FIRST <n:any-type>
   12 := FIRST (12,13,14)
   3 := FIRST 3
   null := FIRST ()
```

9.12.13 Any (unary, right associative)

The **any** operator returns **true** if any of the items in a list is **true**. It returns **false** if they are all **false**. Otherwise it returns **null**. The special case of a list with zero members, results in false. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:Boolean> := ANY <n:any-type>
   true := ANY (true,false,false)
   false := ANY false
   false := ANY ()
   null := ANY (3, 5, "red")
   false := ANY (false, false)
   null := ANY (false, null)
```

9.12.14 All (unary, right associative)

The **all** operator returns **true** if all of the items in a list are **true**. It returns **false** if any of the items is **false**. Otherwise it returns **null**. The special case of a list with zero members, results in true. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:Boolean> := ALL <n:any-type>
   false := ALL (true,false,false)
   false := ALL false
   true := ALL ()
   null := ALL (3, 5, "red")
   null := ALL (true, null)
```

9.12.15 No (unary, right associative)

The **no** operator returns **true** if all of the items in a list are **false**. It returns **false** if any of the items is **true**. Otherwise it returns **null**. The special case of a list with zero members, results in true. If all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time is lost). Its usage is:

```
<1:Boolean> := NO <n:any-type>
   false := NO (true,false,false)
   true := NO false
   true := NO ()
   null := NO (3, 5, "red")
   null := NO (false, null)
```

9.12.16 Latest (unary, right associative)

The **latest** operator returns the value with the latest primary time in a list. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If the list is empty, **null** is returned. The primary time of the selected argument is maintained. Its usage is:

```
<1:any-type> := LATEST <n:any-type>
null := LATEST ()
```

"penicillin" := LATEST ("penicillin", "ibuprofen", "pseudoephedrine HCL");

(T16:40)   (T16:40)   (T14:05)   (T14:04)

9.12.17 Earliest (unary, right associative)

The **earliest** operator returns the value with the earliest primary time in a list. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If the list is empty, **null** is returned. The primary time of the argument is maintained. Its usage is:

```
<1:any-type> := EARLIEST <n:any-type>
null := EARLIEST ()
```

"pseudoephedrine HCL" := EARLIEST ("penicillin", "ibuprofen", "pseudoephedrine HCL");

(T14:04)   (T16:40)   (T14:05)   (T14:04)

9.12.18 Element (binary)

The **element** (**[…]**) operator is used to select one or more elements from a list, based on ordinal position starting at 1 for the first element. The arguments to "index" are a list expression (to the left of the **[…]**) and a list of integers (inside the **[…]**). The element operator maintains the primary times of the selected arguments, even if the data have different primary times. The use of the element operator is as follows:

```
<n:any-type> := <k:any-type>[n:index]20 := (10,20,30,40)[2]
() := (10,20)[()]
(null,20) := (10,20)[1.5,2]
(10,30,50) := (10,20,30,40,50)[1,3,5]
(10,30,50) := (10,20,30,40,50)[1,(3,5)]
(10,20,30) := (10,20,30,40,50)[1 seqto 3]
```

9.12.19 Extract characters ... (unary, right associative)

The **extract characters** operator expects a string as its argument. It returns a list of the single characters in the string. If the argument has more than one element, the elements are first concatenated, as for the **||** operator (see Section 9.8.1). If the argument is an empty list, the result is the empty list **()**. The **string** operator (Section 9.8.3) can be used to put the list back together; and the index operator (Section 9.12.18) can be used to select certain items from the list. The primary times of its arguments are lost. Its usage is:

```
<n:string> := EXTRACT CHARACTERS <m:any-type>
("a","b","c") := EXTRACT CHARACTERS "abc"
("a","b","c") := EXTRACT CHARACTERS ("ab","c")
() := EXTRACT CHARACTERS ()
() := EXTRACT CHARACTERS ""
"edcba" := STRING REVERSE EXTRACT CHARACTERS "abcde"
```

9.12.20 Seqto (binary, non-associative)

The **seqto** operator generates a list of integers in ascending order. Both arguments must be single integers; otherwise null is returned. If the first argument is greater than the second argument, the result is the empty list. The primary times are lost. Its usage is:

```
<n:number> := <1:number> SEQTO <1:number>
   (2,3,4) := 2 SEQTO 4
   () := 4 SEQTO 2
   null := 4.5 SEQTO 2
   (2) := 2 SEQTO 2
   (-3,-2,-1) := -3 SEQTO -1
   (2,4,6,8) := 2 * (1 SEQTO 4)
   null := (1.5 seqto 5)
```

9.12.21 Reverse (unary, right-associative)

The **reverse** operator generates a new list with the elements in the reverse order. The primary times of its arguments are maintained. Its usage is:

```
<n:any-type> := REVERSE <n:any-type>
   (3,2,1) := reverse (1,2,3)
   (6,5,4,3,2,1) := reverse (1 seqto 6)
   () := reverse ()
```

9.12.22 Index Extraction Aggregation operators

These operators behave similarly to their non-index extracting counterparts with the exception that they return the value of the index of the element that matches the specified criteria rather than the value of the element. These operators do not maintain primary times.

9.12.22.1 Index Latest (unary, right associative)

The **index latest** operator returns the index of the element with the latest primary time in a list. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If the list is empty, **null** is returned. Its usage is:

```
<1:any-type> := INDEX LATEST <n:any-type>
   null := INDEX LATEST ()
```

```
1 := INDEX LATEST ("penicillin", "ibuprofen", "psuedophedrine HCL");
                              (T16:40)     (T14:05)       (T14:04)
```

9.12.22.2 Index Earliest (unary, right associative)

The **index earliest** operator returns the index of the element with the earliest primary time in a list. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If the list is empty, **null** is returned. Its usage is:

```
<1:any-type> := EARLIEST <n:any-type>
   null := EARLIEST ()

3 := INDEX LATEST ("penicillin", "ibuprofen", "psuedophedrine HCL");
                              (T16:40)     (T14:05)       (T14:04)
```

9.12.22.3 Index Minimum (unary, right associative)

The **index minimum** operator has one synonym: **index min**. It returns the index of the element with the smallest value in a homogeneous list of an ordered type (that is, all numbers, all times, all durations, or all strings), using the <= operator (see Section 9.5.4). If there is a tie, it selects the element with the latest primary time. Its usage is:

```
<1:ordered> := INDEX MINIMUM <n:ordered>
   1 := INDEX MINIMUM (12,13,14)
   3 := INDEX MIN 3
   null := INDEX MINIMUM ()
   null := INDEX MINIMUM (1,"abc")
```

9.12.22.4 Index Maximum (unary, right associative)

The **maximum** operator has one synonym: **max**. It returns the largest value in a homogeneous list of an ordered type, using the >= operator (see Section 9.5.6). If there is a tie, it selects the element with the latest primary time. Its usage is:

```
<1:ordered> := INDEX MAXIMUM <n:ordered>
   3 := INDEX MAXIMUM (12,13,14)
   3 := INDEX MAX 3
   null := INDEX MAXIMUM ()
   null := INDEX MAXIMUM (1,"abc")
```

9.12.22.5

There are no index extraction equivalents for last and first as INDEX FIRST would always return 1 and INDEX LAST is equivalent to the COUNT operator.

9.13 Query Aggregation Operators:

9.13.1 General Properties:

The query aggregation operators do not follow the default list handling, or the default primary time handling. They perform aggregation on a list. That is, they take a list as one argument and return a single item as a result. If the list argument is a single item, then it is treated as a list of length one. Unless otherwise specified, if all the elements of the list have the same primary time, the result maintains that primary time (otherwise the primary time lost).

The unary query aggregation operators (that is, those that do not include the **from** word) may optionally be followed by **of**.

9.13.2 Nearest ... From (binary, right associative)

The **nearest ... from** operator expects a time as its first argument and a list as its second argument. It selects the item from the list whose time of occurrence is nearest the specified time. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). In the case of a tie, the element with the smallest index is used. The primary times of the argument are maintained. Assume that **data** is a list that is the result of a query with these values: **12, 13, 14**; **data** has these primary times:1990-03-15T15:00:00, 1990-03-16T15:00:00, 1990-03-17T15:00:00; and now is 1990-03-18T16:00:00. The usage of the **nearest ... from** operator is:

```
<n:any-type> := NEAREST <n:time> FROM <m:any-type>
   13 := NEAREST (2 days ago) FROM data
   null := NEAREST (2 days ago) FROM (3,4)
   null := NEAREST (2 days ago) FROM ()
```

9.13.3 Index Nearest ... From (binary, right associative)

The **index nearest ... from** operator functions exactly as the **nearest … from operator** (Section 9.13.2), except that it returns the index of the element rather than the element itself. **Index nearest … from** does not maintain primary time. Assume that **data** is a list that is the result of a query with these values: **12, 13, 14**; **data** has these primary times:1990-03-15T15:00:00, 1990-03-16T15:00:00, 1990-03-17T15:00:00; and now is 1990-03-18T16:00:00. The usage of the **index nearest ... from** operator is:

```
<n:number> := INDEX NEAREST <n:time> FROM <m:any-type>
   2 := INDEX NEAREST (2 days ago) FROM data
   null := INDEX NEAREST (2 days ago) FROM (3,4)
```

9.13.4 Slope (unary, right associative)

The **slope** operator performs a regression and returns the slope for the result of a query assuming the y axis contains the values and the x axis contains the times. The result is expressed as units per day, but is considered to be a number. **Null** results if the argument has fewer than two items. If all the elements of the list have the same primary time, the result is **null**. If one or more of the primary times is non-existent, the result is **null**. The result of the slope operator does not have a primary time. Its usage is (assuming the same **data** as above):

```
<1:number> := SLOPE <n:number>
   1 := SLOPE data
   null := SLOPE (3,4)
```

9.14 Transformation Operators:

9.14.1 General Properties:

The transformation operators do not follow the default list handling, or the default primary time handling. They transform a list, producing another list. If the list argument is a single item, then it is treated as a list of length one. The result is always a list even if there is only one item (except if there is an error, in which case the result is **null**).

Operators that are unary (that is, that do not include the **from** word) may optionally be followed by **of**.

9.14.2 Minimum ... From (binary, right associative)

The **minimum ... from** operator has one synonym: **min ... from**. It expects a number (call it N) as its first argument and a homogeneous list of an ordered type as its second argument. It returns a list with the N smallest items from the argument list, in the same order that they are in the second argument, and with any duplicates preserved. The result is **null** if N is not a non-negative integer. If there are not enough items in the argument list, then as many as possible are returned. If there is a tie, then it selects the latest of those elements that have a primary time. The primary times of the argument are maintained. Its usage is:

```
<n:ordered> := MINIMUM <1:number> FROM <m:ordered>
   (11,12) := MINIMUM 2 FROM (11,14,13,12)
   (,3) := MINIMUM 2 FROM 3
   null := MINIMUM 2 FROM (3, "asdf")
   () := MINIMUM 2 FROM ()
   () := MINIMUM 0 FROM (2,3)
   (1,2,2) := MINIMUM 3 FROM (3,5,1,2,4,2)
```

9.14.3 Maximum ... From (binary, right associative)

The **maximum ... from** operator has one synonym: **max ... from**. It expects a number (call it N) as its first argument and a homogeneous list of an ordered type as its second argument. It returns a list with the N largest items from the argument list, in the same order that they are in the second argument, and with any duplicates preserved. The result is **null** if N is not a non-negative integer. If there are not enough items in the argument list, then as many as possible are returned. If there is a tie, then it selects the latest of those elements that have a primary time. The primary times of the argument are maintained. Its usage is:

```
<n:ordered> := MAXIMUM <1:number> FROM <m:ordered>
   (14,13) := MAXIMUM 2 FROM (11,14,13,12)
   (,3) := MAXIMUM 2 FROM 3
   null := MAXIMUM 2 FROM (3, "asdf")
   () := MAXIMUM 2 FROM ()
   () := MAXIMUM 0 FROM (1,2,3)
   (5,4,4) := MAXIMUM 3 FROM (1,5,2,4,1,4)
```

9.14.4 First ... From (binary, right associative)

The **first ... from** operator expects a number (call it N) as its first argument and a list as its second argument. It returns a list with the first N items from the argument list. The result is **null** if N is not a non-negative integer. If the list is the result of a time-sorted query, then the returned items are the earliest in time. If there are not enough items in the argument list, then as many as possible are returned. This means that **first 1 from x** differs from **first x** if **x** is empty; the former returns **()** and the latter returns **null**. The primary times of the argument are maintained. Its usage is:

```
<n:any-type> := FIRST <1:number> FROM <m:any-type>
   (11,14) := FIRST 2 FROM (11,14,13,12)
   (3) := FIRST 2 FROM 3
   (null,1) := FIRST 2 FROM (null,1,2,null)
   () := FIRST 2 FROM ()
```

9.14.5 Last ... From (binary, right associative)

The **last ... from** operator expects a number (call it N) as its first argument and a list as its second argument. It returns a list with the last N items from the argument list. The result is **null** if N is not a non-negative integer. If the list is the result of a time-sorted query, then the returned items are the latest in time. If there are not enough items in the argument list, then as many as possible are returned. This means that **last 1**

from x differs from **last x** if **x** is empty; the former returns **()** and the latter returns **null**. The primary times of the argument are maintained. Its usage is:

```
<n:any-type> := LAST <1:number> FROM <m:any-type>
   (13,12) := LAST 2 FROM (11,14,13,12)
   (3) := LAST 2 FROM 3
   (2,null) := LAST 2 FROM (null,1,2,null)
   () := LAST 2 FROM ()
```

9.14.6 Increase (unary, right associative)

The **increase** operator returns a list of the differences between successive items in a homogeneous numeric, time, or duration list. There is one fewer item in the result than in the argument; if the argument is an empty list, then **null** is returned. The primary time of the second item in each successive pair is kept. Its usage is:

```
<n:number> := INCREASE <m:number>
   (4,-2,-1) := INCREASE (11,15,13,12)
   () := INCREASE 3
   null := INCREASE ()
<n: duration> := INCREASE <m:time>
   (1 day) := INCREASE (1990-03-01,1990-03-02)
<n:duration> := INCREASE <m:duration>
   (1 day) := INCREASE (1 day, 2 days)
```

9.14.7 Decrease (unary, right associative)

The **decrease** operator returns a list of the negative differences between successive items in a homogeneous numeric, time, or duration list. There is one fewer item in the result than in the argument; if the argument is an empty list, then **null** is returned. **Decrease** is the additive inverse of **increase**. The primary time of the second item in each successive pair is kept. Its usage is:

```
<n:number> := DECREASE <m:number>
   (-4,2,1) := DECREASE (11,15,13,12)
   () := DECREASE 3
   null := DECREASE ()
<n: duration> := DECREASE <m:time>
   ((-1) day) := DECREASE (1990-03-01,1990-03-02)
<n:duration> := DECREASE <m:duration>
   ((-1) day) := DECREASE (1 day, 2 days)
```

9.14.8 % Increase (unary, right associative)

The **% increase** operator has one synonym: **percent increase**. It returns a list of the percent increase between items in successive pairs in a homogeneous number or duration list (the denominator is the first item in each pair; if it is zero, then **null** is returned). The primary time of the second item in each successive pair is kept. Its usage is:

```
<n:number> := % INCREASE <m:number>
   (36.3636,-13.3333) := % INCREASE (11,15,13)
   () := % INCREASE 3
   null := % INCREASE ()
```

```
<n:number> := % INCREASE <m:duration>
   (100) := % INCREASE (1 day, 2 days)
```

9.14.9 % Decrease (unary, right associative)

The **% decrease** operator has one synonym: **percent decrease**. It returns a list of the percent decrease between items in successive pairs in a homogeneous number or duration list (the denominator is the first item in each pair, if it is zero, then **null** is returned). The primary time of the second item in each successive pair is kept. Its usage is:

```
<n:number> := % DECREASE <m:number>
   (-36.3636,13.3333) := % DECREASE (11,15,13)
   () := % DECREASE 3
   null := % DECREASE ()
<n:number> := % DECREASE <m:duration>
   (-100) := % DECREASE (1 day, 2 days)
```

9.14.10 Earliest ... From (binary, right associative)

The **earliest ... from** operator expects a number (call it N) as its first argument and a list as its second argument. It returns a list with the earliest N items from the argument list, in the order they appear in the argument list. The result is **null** if N is not a non-negative integer. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If there are not enough items in the argument list, then as many as possible are returned. This means that **earliest 1 from x** differs from **earliest x** if **x** is empty; the former returns **()** and the latter returns **null**. The primary times of the argument are maintained. Its usage is:

```
<n:any-type> := EARLIEST <1:number> FROM <m:any-type>
   () := EARLIEST 2 FROM ()
   <
```

9.14.11 Latest ... From (binary, right associative)

The **latest ... from** operator expects a number (call it N) as its first argument and a list as its second argument. It returns a list with the latest N items from the argument list, in the order they appear in the argument list. The result is **null** if N is not a non-negative integer. If any of the elements do not have primary times, the result is **null** (the argument can always be qualified by **where time of it is present**, if this is not desired behavior). If there are not enough items in the argument list, then as many as possible are returned. This means that **latest 1 from x** differs from **latest x** if **x** is empty; the former returns **()** and the latter returns **null**. The primary times of the argument are maintained. Its usage is:

```
<n:any-type> := LATEST <1:number> FROM <m:any-type>
   () := LATEST 2 FROM ()
```

9.14.12 Index Extraction Transformation Operators

These operators behave similarly to their non-index extracting counterparts with the exception that they return the value of the index of the element that matches the specified criteria rather than the element itself. These operators do not maintain primary times.

9.14.12.1 Index Minimum ... From (binary, right associative)

The **index minimum ... from** operator has one synonym: **index min ... from**. It expects a number (call it N) as its first argument and a homogeneous list of an ordered type as its second argument. It returns a list with the indices of the N smallest items from the argument list, in the same order that they are in the second argument, and with any duplicates preserved. The result is **null** if N is not a non-negative integer. If there are not enough items in the argument list, then as many indices as possible are returned. If there is a tie, then it selects the latest of those elements that have a primary time. The primary times of the argument are not maintained. Its usage is:

```
<n:number> := INDEX MINIMUM <1:number> FROM <m:ordered>
   (1,4) := INDEX MINIMUM 2 FROM (11,14,13,12)
   (3,4,6) := INDEX MINIMUM 3 FROM (3,5,1,2,4,2)
   null := INDEX MIN 2 FROM (3, "asdf")
   (,1) := INDEX MINIMUM 2 FROM 3
   () := INDEX MINIMUM 0 FROM (2,3)
```

9.14.12.2 Index Maximum ... From (binary, right associative)

The **index maximum ... from** operator has one synonym: **index max ... from**. It expects a number (call it N) as its first argument and a homogeneous list of an ordered type as its second argument. It returns a list with the indices of the N largest items from the argument list, in the same order that they are in the second argument, and with any duplicates preserved. The result is **null** if N is not a non-negative integer. If there are not enough items in the argument list, then as many indices as possible are returned. If there is a tie, then it selects the latest of those elements that have a primary time. The primary times of the argument are not maintained. Its usage is:

```
<n:number> := INDEX MAXIMUM <1:number> FROM <m:ordered>
   (2,3) := INDEX MAXIMUM 2 FROM (11,14,13,12)
   (2,3,5) := INDEX MAXIMUM 3 FROM (3,5,1,2,4,2)
   null := INDEX MAX 2 FROM (3, "asdf")
   (,1) := INDEX MAXIMUM 2 FROM 3
   () := INDEX MAXIMUM 0 FROM (2,3)
```

9.14.12.3 First… From; Last… From

There are no INDEX extraction operator parallels for **First … From** and **Last … From** as these can be generated using either the seqto operator (for **First … From**) or the seqto and count operators (for **Last … From**). Thus if these functions are needed, use the following:

Index First x From y : 1 seqto x

Index Last x From y : (count(y)-x) seqto count(y)

9.15 Query Transformation Operator:

9.15.1 General Properties:

The query transformation operator does not follow the default list handling, or the default primary time handling. It transforms a list, producing another list. If the list argument is a single item, then it is treated as a list of length one. The result is always a list even if there is only one item (except if there is an error, in which case the result is **null**).

The query transformation operator can only be applied to the result of a query, since it requires that a time be associated with each item in the argument list. **Null** is returned if it is used on other data.

The query transformation operator may optionally be followed by **of**.

9.15.2 Interval (unary, right associative)

The **interval** operator returns the difference between the primary times of succeeding items in a list. It is analogous to **increase**. The primary times of the argument are lost. Its usage is (assuming that **data** is the result of a query with these primary times: **1990-03-15T15:00:00, 1990-03-16T15:00:00, 1990-03-18T21:00:00**):

```
<n:duration> := INTERVAL <m:any-type>
   (1 day, 2.25 days) := INTERVAL data
   null := INTERVAL (3,4)
```

9.16 Numeric Function Operators

The numeric function operators are all unary functions that work with numbers. When an illegal operation is attempted (for example, **log 0**) then **null** is returned.

9.16.1 Arccos (unary, right associative)

The **arccos** operator calculates the arc-cosine (expressed in radians) of its argument. Its usage is:

```
<n:number> := ARCCOS <n:number>
   0 := ARCCOS 1
```

9.16.2 Arcsin (unary, right associative)

The **arcsin** operator calculates the arc-sine (expressed in radians) of its argument. Its usage is:

```
<n:number> := ARCSIN <n:number>
   0 := ARCSIN 0
```

9.16.3 Arctan (unary, right associative)

The **arctan** operator calculates the arc-tangent (expressed in radians) of its argument. Its usage is:

```
<n:number> := ARCTAN <n:number>
   0 := ARCTAN 0
```

9.16.4 Cosine (unary, right associative)

The **cosine** operator has one synonym: **cos**. It calculates the cosine of its argument (expressed in radians). Its usage is:

```
<n:number> := COSINE <n:number>
   1 := COSINE 0
```

9.16.5 Sine (unary, right associative)

The **sine** operator has one synonym: **sin**. It calculates the sine of its argument (expressed in radians). Its usage is:

```
<n:number> := SINE <n:number>
   0 := SINE 0
```

9.16.6 Tangent (unary, right associative)

The **tangent** operator has one synonym: **tan**. It calculates the tangent of its argument (expressed in radians). Its usage is:

```
<n:number> := TANGENT <n:number>
   0 := TANGENT 0
```

9.16.7 Exp (unary, right associative)

The **exp** operator raises mathematical e to the power of its argument. Its usage is:

```
<n:number> := EXP <n:number>
   1 := EXP 0
```

9.16.8 Log (unary, right associative)

The **log** operator returns the natural logarithm of its argument. Its usage is:

```
<n:number> := LOG <n:number>
   0 := LOG 1
```

9.16.9 Log10 (unary, right associative)

The **log10** operator returns the base 10 logarithm of its argument. Its usage is:

```
<n:number> := LOG10 <n:number>
   1 := LOG10 10
```

9.16.10 Int (unary, right associative)

The **int** operator returns the largest integer less than or equal to its argument (truncates towards negative infinity). It is synonymous with **floor** (Section 9.16.11). Its usage is:

```
<n:number> := INT <n:number>
   -2 := INT -1.5
   -2 := INT -2.0
   1 := INT (1.5)
```

9.16.11 Floor (unary, right associative)

The **floor** operator is synonymous with **int**. It returns the largest integer less than or equal to its argument (truncates towards negative infinity).

9.16.12 Ceiling (unary, right associative)

The **ceiling** operator returns the smallest integer greater than or equal to its argument (truncates towards positive infinity). Its usage is:

```
<n:number> := CEILING <n:number>
   -1 := CEILING -1.5
   -1 := CEILING -1.0
   2 := CEILING 1.5
```

9.16.13 Truncate (unary, right associative)

The **truncate** operator removes any fractional part of a number (truncates towards zero). Its usage is:

```
<n:number> := TRUNCATE <n:number>
   -1 := TRUNCATE -1.5
   -1 := TRUNCATE -1.0
   1 := TRUNCATE 1.5
```

9.16.14 Round (binary, right associative)

The **round** operator rounds a number to an integer.

For positive numbers: If the fractional portion of the operand is greater than or equal to 0.5, the operator rounds to the next highest integer. Fractional portions less than 0.5 round to the next lowest integer.

For negative numbers: If the absolute value of the fractional portion of the operand is greater than or equal 0.5, the operator rounds to the next lower negative integer. Fractional portions with absolute values less than 0.5 round to the next highest integer.

Its usage is:

```
<n:number> := ROUND <n:number>
   1 := ROUND 0.5
   3 :=  ROUND 3.4
   4 :=  ROUND 3.5
   -3 := ROUND -3.5
   -3 := ROUND -3.4
   -4 := ROUND -3.7
```

9.16.15 Abs (unary, right associative)

The **abs** operator returns absolute value of its argument. Its usage is:

```
<n:number> := ABS <n:number>
   1.5 := ABS (-1.5)
```

9.16.16 sqrt (unary, right associative)

The **sqrt** operator returns the square root of its argument. Because imaginary numbers are not supported, the square root of a negative number results in **null**. Its usage is:

```
<n:number> := SQRT <n:number>
   2 := SQRT 4
   null := SQRT(-1)
```

9.16.17 As number (unary, non-associative)

The **as number** operator attempts to convert a string or Boolean to a number. If conversion to a number is possible, the number is returned, otherwise **null** is returned. The primary time of the argument is preserved. The usual use for this will be to convert a string which contains a valid number representation i.e. "123" into the represented number. If the string does not contain a valid number then the result will be null. Boolean values are translated at follows: Boolean TRUE is represented at 1 and Boolean FALSE is represented at 0.

```
<n:number> := <n:numeric string> AS NUMBER;
   5 := "5" AS NUMBER;
   null := "xyz" AS NUMBER;
<n:number> := <n:Boolean> AS NUMBER;
   1 := True AS NUMBER;
   0 := False AS NUMBER;
<n:number> := <n:number> AS NUMBER;
   6 := 6 AS NUMBER;
    (7, 8, 230, 4100, null, null, 1, 0, null, null, null) := ("7", 8,
    "2.3E+2", 4.1E+3, "ABC", Null, True, False, 1997-10-31T00:00:00, now, 3
    days) AS NUMBER;
   ():= () AS NUMBER;
```

9.17 Time Function Operator

The time function operator does not follow the default primary time handling.

9.17.1 Time (unary, right associative)

The **time** operator returns the primary time (that is, time of occurrence) of the result of a value derived from a query (see Section 8.9). **Null** is returned if it is used on data that has no primary time. The result of **time** preserves the primary time of its argument; so **time time x** is equivalent to **time x**. Its usage is (assuming that **data** is the result of a query with one element whose primary time is: **1990-03-15T15:00:00**):

```
<n:time> := TIME [OF] <n:any-type>
   1990-03-15T15:00:00 := TIME OF data
   1990-03-15T15:00:00 := TIME TIME data
   null := TIME (3,4)
```

The inverse of the **time** operator (to set the primary time of a value) can be achieved by using **time** on the left side of an assignment statement. For example:

```
TIME [OF] <n:any-type> := <n:time>;
   TIME data1 := time data2;
```

10 LOGIC SLOT

10.1 Purpose

The logic slot uses data about the patient obtained from the data slot, manipulates the data, tests some condition, and decides whether to execute the action slot. It is in this slot that most of the actual health logic is obtained.

10.2 Logic Slot Statements

The logic slot is composed of a set of statements.

10.2.1 Assignment Statement

The assignment statement places the value of an expression into a variable. There are two equivalent versions:

```
<identifier> := <expr> ;
   LET <identifier> BE <expr> ;
```

<identifier> is an identifier; it represents the name of the variable. **<expr>** is a valid expression as defined in Section 7.2.2.

Any reference to the identifier that occurs after the assignment statement will return the value that was assigned from the expression (even if it is in another structured slot; for example, the action slot). A subsequent assignment to the same variable will overwrite the value. If a variable is referred to before its first assignment, **null** is returned (it is poor programming practice to depend on this).

The following variables cannot be re-assigned in the logic slot after they have been assigned in the data slot: **event** (Section 11.2.2), **mlm** (Section 11.2.3), and **interface** (Section 11.2.12). Once defined in the data slot, they should not change.

After executing these statements, the value of variable **var2** is **5**:

```
var1 := 1;
var1 := 3;
var2 := var1 + 2;
```

10.2.2 If-Then Statement

The if-then statement permits conditional execution based upon the value of an expression. It tests whether the expression (<**expr**>) is equal to a single Boolean **true**. If it is, then a block of statements (<**block**>) is executed. (A block of statements is simply a collection of valid statements possibly including other if-then statements; thus the if-then statement is a nested structure.) If the expression is a list, or if it is any single item other than **true**, then the block of statements is not executed. The flow of control then continues with subsequent statements. The if-then statement has several forms:

10.2.2.1 Simple If-Then Statement

This form executes **<block1>** if **<expr1>** is **true**:

```
IF <expr1> THEN
   <block1>
ENDIF;
```

10.2.2.2 If-Then-Else Statement

This form executes **<block1>** if **<expr1>** is **true**; otherwise it executes **<block2>**:

```
IF <expr1> THEN
   <block1>
ELSE
   <block2>
ENDIF;
```

10.2.2.3 If-Then-Elseif Statement

This form sequentially tests each of the expressions **<expr1>** to **<exprN>** (there may be any number of them). When it finds one that is **true**, its associated block is executed. Once one block is executed, no other expressions are tested, and no other blocks are executed. If none of the expressions is true, then **<blockE>** is executed. The **else <blockE>** portion is optional. Its form is:

```
IF <expr1> THEN
   <block1>
ELSEIF <expr2> THEN
   <block2>
ELSEIF <expr3> THEN
   <block3>
...
ELSEIF <exprN> THEN
   <blockN>
ELSE
   <blockE>
ENDIF;
```

10.2.2.4 Treatment of Null

It is important to emphasize that non-**true** is different from **false**. That is, the **else** portion of the if-then-else statement is executed whether the expression is **false**, or **null**, or anything other than **true**. Thus these two if-then statements, which appear to be the same, produce different results when **var1** is **null**.

```
IF var1 THEN
   var2 := 0;
ELSE
   var2 := 45;
ENDIF;

IF not(var1) THEN
   var2 := 45;
ELSE
   var2 := 0;
ENDIF;
```

To avoid the **null** problem, it is safer to test for existence first, then test for **true**.

```
IF var1 is Boolean THEN
IF var1 THEN
   var2 := "var1 is true";
ELSE
   var2 := "var1 is false";
ENDIF;
ELSE
   var2 := "var1 is null or some other type";
ENDIF;
```

10.2.2.5 Treatment of Lists

Lists are always non-true; therefore using an expression that contains a list will always produce the same negative result. Instead, one of the Boolean aggregation operators should be used: **any**, **all**, or **no** (see Sections 9.12.13, 9.12.14, and 9.12.15). For example, to execute a statement if any of the elements in **Bool_list** is true, use:

```
IF any(Bool_list) THEN
   var2 := 0;
ENDIF;
```

10.2.3 Conclude Statement

The conclude statement ends execution in the logic slot. If the expression (<**expr**$_1$>) in the conclude statement is a single **true** then the action slot is executed immediately. Otherwise the whole MLM terminates immediately. No further execution in the logic slot occurs regardless of the expression. There may be more than one conclude statement in the logic slot, but only one will be executed in a single run of the MLM. Its form is:

```
CONCLUDE <expr>;
```

The cautions for the if-then statement about **null** and list (in Section 10.2.2) also hold for the conclude statement.

If no conclude statement is executed, then the logic slot terminates after it executes its last statement, and the action slot is not executed. In effect, the default is **conclude false**.

These are valid conclude statements:

```
CONCLUDE false;
CONCLUDE potas > 5.0;
```

10.2.4 Call Statement

The call statement permits nesting of MLMs. Given an MLM filename, the MLM can be called directly with optional parameters and return zero or more results. Given an event definition, all the MLMs that are normally evoked by that event can be called; the called MLMs can be given optional parameters and optionally return results. Given an interface definition, the foreign function can be called directly with optional parameters and return zero or more results. There are two basic forms, CALL and CALL…WITH, (the pairs represent equivalent versions):

```
<var> := CALL <name>;
   LET <var> BE CALL <name>;

<var> := CALL <name> WITH <expr>;
   LET <var> BE CALL <name> WITH <expr>;

(<var>, <var>, …) := CALL <name> WITH <expr>;
   LET (<var>, <var>, …) BE CALL <name> WITH <expr>;

<var> := CALL <name> WITH <expr>, …, <expr>;
   LET <var> BE CALL <name> WITH <expr>, …, <expr>;

(<var>, <var>, …) := CALL <name> WITH <expr>, …, <expr>;
   LET (<var>, <var>, …) BE CALL <name> WITH <expr>, …, <expr>;
```

10.2.4.1 Commas

Because arguments to a call are separated by commas (see **argument**, Section **11.2.4**), and comma is also an operator (list construction, see Section 9.2.1), there is an apparent ambiguity. This ambiguity is resolved in favor of comma as a parameter separator. Any argument expression containing the comma operator or another operator of the same or lower precedence must be enclosed in parentheses. For example,

This call passes three arguments:

```
x := CALL xxx with (a,b),(c merge d),e+f;
```

This call passes two arguments:

```
y := CALL yyy WITH expr1, expr2;
```

This call appears similar to the one above, but it only passes one argument :

```
z := CALL zzz WITH (expr3, expr4);
```

10.2.4.2 <name>

<name> is an identifier that must represent either a valid MLM variable as defined by the MLM statement in the data slot (see Section 11.2.3), a valid event variable as defined by the event statement in the data slot (see Section 11.2.2), or a valid interface variable as defined by the interface statement in the data slot (see Section 11.2.12).

10.2.4.3 <expr>

<expr>s are optional parameters, which may be of any type, including list and null. Primary times associated with the parameter are maintained.

10.2.4.4 <var>

<var> is an identifier that represents the local variable that will be assigned the result.

10.2.4.5 MLM Call

If **<name>** is an MLM variable, then when the call statement is executed, the main MLM (that is, the one issuing the call) is interrupted, and the named MLM is called. If the called MLM has argument statement(s) in its data slot (see Section 11.2.4), then the values of the **<expr>**s are assigned. If a called MLM's argument statement has more variables (parameters) than sent by the call statement, then **null** is assigned to the extra variable(s). If the call statement passes more variables (parameters) than the called MLM is expecting, the additional parameters are silently dropped. The called MLM is executed, and when it terminates, execution of the main MLM resumes. If the called MLM concludes true and there is a return statement in the called MLM's action slot (see Section 12.2.2), then the value of its expression is assigned to **<var>**. If the return statement has more values than the calling MLM can accept, then the extra return values are silently dropped. If the return statement has fewer values than the calling MLM is expecting, then the extra return values are **null**. If there is no return statement, or if the called MLM concludes false, then **null** is assigned to **<var>**. Examples:

```
var1 := CALL my_mlm1 WITH param1, param2;


(var2, var3, var4) := CALL my_mlm2 WITH param1, param2;
```

10.2.4.6 Event Call

If **<name>** is an event variable, then execution is similar. The main MLM is interrupted, and all the MLMs whose evoke slots refer to the named event are executed (see Section 13). They each receive the parameters if there are any via their argument statement(s). The results of all called MLM's return statements are concatenated together into a list; called MLMs with no return statement and called MLMs that return a single **null** are not included in the result. The order of the returned values is implementation dependent. The result is assigned to **<var>**, and execution continues. **<var>** will always be a list, even if it has one item. Example:

```
var1 := CALL my_event WITH param1, param2;
```

10.2.4.7 Interface Call

If **<name>** is an INTERFACE variable, then when the call statement is executed, the MLM (that is, the one issuing the call) is interrupted, and the named INTERFACE is called. If the called INTERFACE functions accept variables (parameters), then the values of the **<expr>**s are assigned. If a called INTERFACE's function expects more variables (parameters) than sent by the call statement, then **null** is assigned to the extra variable(s). The called function is executed, and when it finishes, execution of the MLM resumes. If the called function returns one or more values, then the values are assigned to the **<var>s**. If the function returns more values than the calling MLM can accept, then the extra return values are silently dropped. If the INTERFACE function returns fewer values than the calling MLM is expecting, then the extra values are **null**. If the function does not return any values, then **null** is assigned to **<var>**. Examples:

```
var1  := CALL my_interface_function1 WITH param1, param2;


(var1, var2, var3) := CALL my_interface_function2 WITH param1, param2;
```

10.2.4.8 Example: Valid MLM Statement and Call Statement

Here is a valid call statement:

```
/* Define find_allergies MLM */
find_allergies := MLM 'find_allergies' from institution "ABC Hospital";
/* Lists two medications and their allergens */
med_orders:= ("PEN-G", "aspirin");
med_allergens:= ("penicillin", "aspirin");
/* Lists three patient allergies and their reactions */
patient_allergies:= ("milk", "codeine", "penicillin" );
patient_reactions:= ("hives", NULL, "anaphylaxis");
/* Passes 4 arguments and receives 3 lists as values */
(meds, allergens, reactions):= call find_allergies with med_orders,
                                     med_allergens,
                                     patient_allergies,
                                     patient_reactions;
```

10.2.4.9 Example: Valid Interface Statement and Call Statement

Here is a valid interface statement:

```
/* Define find_allergies external function*/
find_allergies := INTERFACE
    {\\RuleServer\AllergyRules\my_institution\find_allergies.exe};
/* Lists two medications and their allergens */
med_orders:= ("PEN-G", "aspirin");
med_allergens:= ("penicillin", "aspirin");
/* Lists three patient allergies and their reactions */
patient_allergies:= ("milk", "codeine", "penicillin" );
patient_reactions:= ("hives", NULL, "anaphylaxis");
/* Passes 4 arguments and receives 3 lists as values */
(meds, allergens, reactions):= call find_allergies with med_orders,
                                     med_allergens,
                                     patient_allergies,
                                     patient_reactions;
```

10.2.5 While Loop

A simple form of looping is provided by the **while** loop. Its form is:

```
WHILE <expr> DO
   <block>
ENDDO;
```

The **while** loop tests whether an expression (**<expr>)** is equal to a single Boolean **true** (similar to the conditional execution introduced in the **if ... then** syntax **-** see Section 10.2.2). If it is, the block of statements (**<block>**) is executed repeatedly until <**expr**> is not **true**. If **<expr>** is not **true**, the block is not executed.

Authors should take care when using **while** loops in MLMs, since it is possible to create infinite loops. It is the author's responsibility, not the compiler, to avoid infinite looping.

Here is an example:

```
/* Initialize variables */
a_list:= ();
```

```
m_list:= ();
r_list:= ();
num:= 1;
/* Checks each allergen in the medications to determine if the patient is
    allergic to it */
while num <= (count med_allergen) do
allergen:= last(first num from med_allergens);
allergy_found:= (patient_allergies = allergen);
reaction:= patient_reactions where allergy_found;
medication:= med_orders where (med_allergens = allergen);
/* Adds the allergen, medication, and reaction to variables that will */
/* be returned to the calling MLM */
If any allergy_found then
a_list:= a_list, allergen;
m_list:= m_list, medication;
r_list:= r_list, reaction;
endif;
/* Increments the counter that is used to stop the while-loop */
num:= num + 1 ;
enddo;
```

10.2.6 For Loop

Another form of looping is provided by the **for** loop. Its form is:

```
FOR <identifier> in <expr> DO
   <block>
ENDDO;
```

The **<expr>** will usually be a list generator. If **<expr>** is empty or null, the block is not executed. Otherwise, the block is executed with the **<identifier>** taking on consecutive elements in **<expr>**. The **<identifier>** cannot be assigned to inside the **<block>** (the compiler must produce a compilation error if this is attempted). After the **enddo**, the **<identifier>** becomes undefined and its value should not be used. A compiler may flag this as an error.

Here is an example:

```
/* Initialize variables */
a_list:= ();
m_list:= ();
r_list:= ();
/* Checks each allergen in the medications to determine if the patient is
    allergic to it */
for allergen in med_allergens do
   allergy_found:= (patient_allergies = allergen);
   reaction:= patient_reactions where allergy_found;
   medication:= med_orders where (med_allergens = allergen);
   /* Adds the allergen, medication, and reaction to variables that will */
   /* be returned to the calling MLM */
   If any allergy_found then
       a_list:= a_list, allergen;
```

```
            m_list:= m_list, medication;
            r_list:= r_list, reaction;
        endif;
    enddo;
```

Here is an example using a set number of iterations:

```
    for i in (1 seqto 10) do
       …
    enddo;
```

10.3 Logic Slot Usage

The general approach in the logic slot is to use the operators and expressions to manipulate the patient data obtained in the data slot in order to test for some condition in the patient. Once sufficient data, positive or negative, has been amassed the **conclude** statement is executed. If there is no **conclude** statement in the logic slot, then it will never conclude **true**, and the action slot will never be executed. Some logic slots are simple (for example, test whether the serum potassium is greater than 5.0), and some are complex (for example, calculate a diagnosis score).

11 DATA SLOT

11.1 Purpose

The purpose of the data slot is to define local variables used in the rest of the MLM. The goal is to isolate institution-specific portions to one slot. Within the data slot, the institution-specific portions are placed in mapping clauses (see Section 7.1.8) so that the institution-specific part does not interfere with the MLM syntax.

11.2 Data Slot Statements:

The following variables cannot be re-assigned in the logic slot after they have been assigned in the data slot: **event** (Section 11.2.2), **mlm** (Section 11.2.3), and **interface** (Section 11.2.12). Once defined in the data slot, they should not change.

11.2.1 Read Statement

The main source of data is the patient database. Each institution will need to do its own queries; databases may be hierarchical, relational, object oriented, etc. The vocabulary used to represent entities in the database will vary from institution to institution. (No attempt was made to select a standard vocabulary in this version of this specification.) The read statement is designed to isolate those parts of a database query that are specific to an institution from those parts that are universal.

There is no restriction that a read statement must derive its input from the patient database. A read statement might access a medical dictionary, for example; or it might interactively request information from somebody (and, if the compiler does on-demand optimization, the interaction might happen only if needed). How this is done is implementation defined.

11.2.1.1

The database query itself is divided into three parts: the aggregation or transformation operator, the time constraint, and the rest of the query. For backward compatibility, parentheses may be placed around the **<mapping> WHERE <constraint>** part. The general form of the read statement is (there are two equivalent versions):

```
<var> := READ <aggregation> <mapping> WHERE <constraint>;
   LET <var> BE READ <aggregation> <mapping> WHERE <constraint>;
```

11.2.1.2 Definitions

<var> is a variable that is assigned that result of the query.

<aggregation> is an aggregation operator (see Section 9.12) or a transformation operator (see Section 9.14), which is applied after the query constraints. If **<aggregation>** is omitted, then all the data that satisfy the constraints are returned. Only the following aggregation and transformation operators are permitted:

```
exist
sum
average
avg
minimum
min
maximum
max
last
first
earliest
latest
minimum ... from
min ... from
max ... from
maximum ... from
last ... from
first ... from
earliest ... from
latest ... from
```

In the default sort ordering, first and last are equivalent to earliest and latest.

<constraint> is any occur comparison operator (see Section 9.7) with **it** (or **they**) as the left argument. In this case **it** refers to the body of the query. The comparison operator specifies the time constraints for the query. If **<constraint>** is omitted, then there are no constraints on time. Examples of valid constraints are:

```
they occurred within the past 3 days
it occurred before the time of surgery
```

<mapping> is a valid mapping clause (see Section 7.1.8), which contains the institution-specific part of the query enclosed in curly brackets. It contains any vocabulary terms and any query syntax that is necessary in the institution to perform a query, except that the aggregation and time constraints are missing. **<mapping>** is required.

11.2.1.3 Examples

These are valid read statements (the portions within curly brackets are arbitrary):

```
var1 := READ {select potassium from results where specimen = `serum`};
var1 := READ latest {select potassium from results};
```

```
LET var1 BE READ {select potassium from results} WHERE it occurred within the
    past 1 week;
```

```
var1 := READ earliest 3 from {select potassium from results} WHERE it
    occurred within the past 1 week;
```

11.2.1.4 Effect

The effect of the read statement is to execute a query, mapping the data in the patient database to a variable that can be used elsewhere in the MLM. The execution of the read statement will be institution-specific. The time constraints must be added to whatever other constraints are within the mapping clause, and the aggregation or transformation operator must also be added to complete the query.

11.2.1.5 Result Type

The result of a query includes the primary time for each item that is returned (see Section 8.9). If **<aggregation>** is an aggregation operator, then the query returns a single item. If **<aggregation>** is a transformation operator or it is absent, then the query returns a list. Thus even if the query requests an entity that is usually singular, such as the birthdate of the patient, a list is assumed unless an aggregation operator is applied (but the list might contain only a single value, in which case it would be indistinguishable from a scalar). The reason for this is that a patient database may have multiple values for a birthdate; it may be that the latest one is assumed to be correct. For example,

```
birthdate := READ latest {select birthdate from demographics};
```

11.2.1.6 Multiple Variables

A query may return more than one result at a time. This is useful for batteries of tests in order to keep the corresponding tests within one blood sample coordinated. The two versions are equivalent (the parentheses around the where are optional):

```
(<var>, <var>, ...) := READ <aggregation> <mapping> WHERE <constraint> ;
```

```
LET (<var>, <var>, ...) BE READ <aggregation> (<mapping> WHERE <constraint>);
```

This is the only situation where a "list of lists" is allowed. The where constraint (if any) is applied separately to each of the resulting lists. Queries must always return the same number of elements, with the same primary times.

11.2.1.7

There may be one or more **<var>** within the parentheses. **<aggregation>**, **<constraint>**, and **<mapping>** are defined as above. The fact that multiple entities are being queried at once is represented in the institution-specific part, **<mapping>**. The **<aggregation>** and **<constraint>** are performed separately on the individual variables; it is institution-defined whether the **<mapping>** returns all the values with matching primary times. For example,

```
/* in this example three anion gaps are calculated */
```

```
(Na,Cl,HCO3) := read last 3 from {select sodium, chloride, bicarb from
    electro};
```

```
anion_gap := Na - (Cl + HCO3) ;
```

11.2.1.8

The order in which read mappings are evaluated is undefined, except that an implementation must guarantee that a read mapping is evaluated before the first time that its value is needed. An implementation may optimize code to avoid executing a read mapping, even if the read mapping has side effects.

11.2.2 Event Statement

The event statement assigns an institution-specific event definition to a variable. An event can be an insertion or update in the patient database, or any other medically relevant occurrence. The variable is currently used in the evoke slot (see Section 13), as part of the call statement to call other MLMs (see Section 10.2.4), and as a Boolean value in a **logic** or **action** slot. There are two equivalent versions:

```
<var> := EVENT <mapping>;
LET <var> BE EVENT <mapping>;
```

11.2.2.1 Definitions

<var> is a variable that represents the event to be defined. It can only be used in the evoke slot or as part of a call statement.

<mapping> is a valid mapping clause (see Section 7.1.8) which contains the institution-specific event definition. How the event is defined and used is up to the institution.

The variable that represents the event can be treated like a Boolean in the **logic** or **action** slots.

The **time** operator (see Section 9.17) can be applied to an event variable. It yields the clinically relevant time of the event. This may be different from the **eventtime** variable, which refers to the time that the event was recorded in the database (see Section 8.4.4).

The order in which event mappings are evaluated is undefined, except that an implementation must guarantee that an event mapping is evaluated before the first time that its value is needed.

11.2.2.2 Example:

```
event1 := EVENT {storage of serum potassium};
```

11.2.3 MLM statement

The MLM statement assigns a valid mlmname to a variable. That variable is currently used only as part of the call statement to call another MLM, as defined in Section 10.2.4. There are two basic forms (the pairs represent equivalent versions):

```
<var> := MLM <term>;
LET <var> BE MLM <term>;
<var> := MLM <term> FROM INSTITUTION <string>;
LET <var> BE MLM <term> FROM INSTITUTION <string>;
```

11.2.3.1 Definitions

<var> is a variable that represents the MLM to be called. It can only be used as part of a call statement.

<term> is a valid constant term as defined in Section 7.1.7. It is the mlmname of the MLM to be called. **mlm_self** (case insensitive) is a special constant that represents the name of the current MLM.

<string> is a valid constant string as defined in Section 7.1.6. If specified, it is the institution name found in the institution slot of the MLM to be called.

If the institution is specified, then a unique MLM is found using the institution name, the mlmname, and the latest version number. If the institution is not specified, then a unique MLM is found using the same institution as the main (calling) MLM, the mlmname, the MLM's validation, and the latest version number. Although the exact form of the version is institution-specific, within an institution it is possible to determine the latest version of an MLM (see Section 6.1.4).

11.2.3.2 Examples:

```
mlm1 := MLM 'mlm_to_be_called';
mlm2 := MLM 'diagnosis_score' FROM INSTITUTION "LDS Hospital";
```

11.2.4 Argument Statement

The argument statement is used by an MLM that is called by another MLM, as defined in Section 10.2.4. If the main MLM passes parameters to the called MLM, then the called MLM retrieves the parameters via the argument statement. The argument statements access the corresponding passed arguments. Thus, the first variable refers to the first passed argument, the second variable to the second argument, etc. If the number of variables is greater than the number of arguments passed from the CALL, **null** is assigned to the extra left-hand-side variable(s). If the MLM is evoked instead of called, all the arguments are treated as **null** (just like any other uninitialized variable). There are two basic forms (the pairs represent equivalent version). One receives a single parameter, and the other receives multiple parameters:

```
<var> := ARGUMENT;
LET <var> BE ARGUMENT;

(<var1>,<var2>,…) := ARGUMENT;
LET (<var1>,<var2>,…) BE ARGUMENT;
```

<var> is a variable that is assigned whatever expression followed **with** in the main MLM's call statement. If there was no such expression, or if the MLM was not called by another MLM, then **null** is assigned.

11.2.4.1 Example:

In the calling MLM:

```
var1 := CALL my_mlm WITH param1, (item1, item2);
```

In the called MLM, named "**my_mlm**":

```
(arg1, list1) := ARGUMENT;
```

11.2.5 Message Statement

The message statement assigns an institution-specific message (for example, an alert) to a variable. It allows an institution to write coded messages in the patient database (see Section 12.2.1). There are two equivalent versions:

```
<var> := MESSAGE <mapping>;
LET <var> BE MESSAGE <mapping>;
```

<var> is a variable that represents the message to be defined. It can only be used in a write statement.

<mapping> is a valid mapping clause (see Section 7.1.8), which contains the message definition. How the message is defined and used is up to the institution.

11.2.5.1 Example:

```
message1 := MESSAGE {pneumonia~23 45 65};
```

11.2.6 Destination Statement

The destination statement assigns an institution-specific destination to a variable. It allows one to write a message to an institution-specific destination (see Section 12.2.1). There are two equivalent versions:

```
<var> := DESTINATION <mapping>;
LET <var> BE DESTINATION <mapping>;
```

<var> is a variable that represents the destination to be defined. It can only be used in a write statement.

<mapping> is a valid mapping clause (see Section 7.1.8) that represents an institution-specific destination. How the destination is defined and used is up to the institution.

11.2.6.1 Example

In this example, the destination is an electronic mail address:

```
destination1 := DESTINATION {email: user@cuasdf.bitnet};
destination2 := DESTINATION { attending_physician(Pt_id) };
destination3 := DESTINATION { "primary physician email" };
```

11.2.7 Assignment Statement

The **assignment** statement, defined in Section 10.2.1, is also permitted in the data slot.

11.2.8 If-Then Statement

The **if-then** statement, defined in Section 10.2.2, is also permitted in the data slot.

11.2.9 Call Statement

The **call** statement, defined in Section 10.2.4, is also permitted in the data slot.

11.2.10 While Loop

The **while** loop, defined in Section 10.2.5, is also permitted in the data slot.

11.2.11 For Loop

The **for** loop, defined in Section 10.2.6, is also permitted in the data slot.

11.2.12 Interface Statement

The **interface** statement assigns an institution-specific foreign function interface definition to a variable. The interface statement permits specification of a foreign function, i.e., a function written in another programming language. Sometimes medical logic requires information not directly available from the database (via **read** statements). It may be desirable to call operating system functions or libraries obtained from other vendors. A foreign function, when specified, can then be called with the call statement (see Section 10.2.4). Curly braces (**{}**) are used to specify the foreign function. The specification within the curly braces is implementation specific. There are two equivalent versions:

```
<var> := INTERFACE <mapping>;
LET <var> BE INTERFACE <mapping>;
```

<var> is a variable that represents the interface to be defined. It can only be used as part of a call statement.

<mapping> is a valid mapping clause (see Section 7.1.8) which contains the institution-specific event definition. How the function interface is defined and used is up to the institution.

11.2.12.1 Example:

The implementation within the {}-braces shows that a string (**char***) will be returned when the third-party API (**ThirdPartyAPI**) is used to call the drug-drug interaction function (**DrugDrugInteraction**). The function expects that two medication strings (**char*,char***) will be passed.

```
data:
   /* Declares the third-party drug-drug interaction function */
   func_drugint := INTERFACE {char* ThirdPartyAPI :
                              DrugDrugInteraction (char*, char*)};
logic:
   /* Calls the drug-drug interaction function */
   alert_text := call func_drugint with "terfenadine", "erythromycin";
```

11.3 Data Slot Usage:

The data slot is used to map institution-specific entities to variables used locally in the MLM. Keeping the mappings in one slot facilitates modifying an MLM for use in another institution.

Although the data slot can perform assignment statements and if-then statements like the logic slot, it is recommended that most of the logic be left in the logic slot. For example, it would be possible to write an MLM with all its mappings and health logic in the data slot, leaving only a simple conclude statement in the logic slot; but this defeats the purpose of separating the data slot and the logic slot. Assignment statements and if-then statements should be used in the data slot only where necessary to support database queries (for example, to calculate a time constraint or to handle details of database semantics, such as handling missing data).

12 ACTION SLOT

12.1 Purpose

Once the MLM has concluded that the condition specified in the logic slot holds true, the action slot is executed, performing whatever actions are appropriate to the condition. Typical actions include sending a message to a health care provider, adding an interpretation to the patient record, returning a result to a calling MLM, and evoking other MLMs. Good programming practice is for an MLM's action slot to contain only return statements, or to contain only call and write statements. If an MLM is called from an action slot (see Section 12.2.4) or evoked by an external event (see Section 13), the only effect of a return statement is to terminate execution of the action slot.

12.2 Action Slot Statements:

12.2.1 Write Statement

The write statement is the main statement in the action slot. It sends a text or coded message (for example, an alert) to a destination. It has several forms:

```
WRITE <expr>;
WRITE <expr> AT <destination>;
WRITE <message>;
WRITE <message> AT <destination>;
```

<expr> is any valid expression, which usually contains text to be read by the health care provider or variables defined in the logic slot.

<destination> is a destination variable as defined in Section 11.2.6. The format and implementation of the destination is institution-specific. Typical destinations include the patient record, a printer, databases, and electronic mail addresses. When the destination is omitted, the message is sent to the default destination. This is generally the health care provider or the patient record, but the implementation is institution-specific.

<message> is a message variable as defined in Section 11.2.5. The message variable permits institutions to write institution-specific coded MLM messages to databases that will not accommodate the **<expr>** form.

The effect of the write statement is to send the specified message either to the default destination (which is usually a health care provider or the patient record) or the destination that is specified.

Within a single MLM, the effect of grouping write statements is unspecified, and depends on the implementation of the syntax.

If an MLM is called by another MLM's action block (see Section 12.2.4), its write statements are output as a separate group from the calling MLM's. However, the order of the groupings is unspecified and depends on the implementation of the syntax.

12.2.1.1 Examples<expr>

In these examples, **serum_pot** is a variable assigned in the logic slot, **email_dest** is a destination variable defined in the data slot, and **a_message** is a message variable defined in the data slot.

```
WRITE "the patient's potassium is " || serum_pot;
WRITE "this is an email alert" AT email_dest;
WRITE a_message;
```

12.2.1.2 Examples<message>

An institution can store coded messages without using the message variable. For example, the following message could be stored not as a free text string but as a unique code that symbolizes the message along with a single field that holds the serum potassium value, which is variable:

```
WRITE "the patient's potassium is " || serum_pot;


WRITE CK0023 || serum_pot;
```

CK0023 would be the institution-specific code representing "**the patient's potassium is**".

The message must be explicitly assigned to the institution-specific code before the code is used in a write statement. Generally, this assignment should take place in the data slot.

12.2.2 Return Statement

The return statement is used in MLMs that are called by other MLMs. It returns a result back to the calling MLM; the result is assigned to the variable in the call statement (see Section 10.2.4). One or more results can be returned by the MLM. Its form is:

```
RETURN <expr>;
RETURN <expr>, ... , <expr>;
```

<expr> is any valid expression, which may be a single item or a list. Primary times are maintained.

When a return statement is executed, no further statements in the MLM are executed.

12.2.2.1 Examples:

```
RETURN (diagnosis_score,diagnosis_name);
RETURN diagnosis_score, diagnosis_name;
```

The first example returns one expression, which is a list. The second example returns two expressions.

12.2.3 If-then Statement

The if-then statement, defined in Section 10.2.2, is also permitted in the action slot.

12.2.4 Call Statement

The call statement in the action slot permits an MLM to call other MLMs conditionally based upon the conclusion in the logic slot. It is similar to the call statement in the logic slot defined in Section 10.2.4; the arguments can be accessed with the argument statement in Section 11.2.4. Given an mlmname, the MLM can be called directly with an optional delay. Given an event definition, all the MLMs that are normally evoked by that event can be called with an optional delay. If the call statement is used to evoke an event, any arguments are ignored. Its forms are:

```
CALL <name>;
CALL <name> DELAY <duration>;
CALL <name> WITH <expr>;
CALL <name> WITH <expr> DELAY <duration>;

CALL <name> WITH <expr>, ..., <expr>;
CALL <name> WITH <expr>, ..., <expr> DELAY <duration>;
```

<name> is an identifier that must represent either a valid MLM variable as defined by an MLM statement in the data slot (see Section 11.2.3), or a valid event variable as defined by an event statement in the data slot (see Section 11.2.2).

<duration> is a valid expression whose value is a duration.

12.2.4.1 Operation

If **<name>** is an MLM variable, then when the main MLM terminates, the named MLM is called. If **<name>** is an event variable, then all the MLMs whose evoke slots refer to the named event are executed (see Section 13). If a delay is present, then the execution of the called MLMs is delayed by the specified duration. Whereas the call statement in the logic slot is synchronous, the call statement in the action slot is asynchronous. The order of execution of called MLMs is implementation dependent.

12.2.4.2 Example

(where **mlmx** has been assigned a suitable value in the data slot, say by **mlmx := MLM 'my_mlm'**):

```
CALL mlmx DELAY 3 days ;
```

12.2.5 WHILE Loop

The **while** loop, defined in Section 10.2.5, is also permitted in the action slot

12.2.6 FOR Loop

The **for** loop, defined in Section 10.2.6, is also permitted in the action slot.

12.3 Action Slot Usage

The action slot is usually simple, containing a single message to be written or a single value to be returned to a calling MLM. Multiple actions can be performed by listing several action statements. The slot can be made more complex by using its if-then statement to select among alternative actions. While this is useful, it is recommended that the amount of health logic in the action slot be kept to a minimum.

13 EVOKE SLOT

13.1 Purpose

The evoke slot defines how an MLM may be triggered. An MLM may be triggered by any of the following:

13.1.1 Occurrence of Some Event

For example, on the storage of a serum potassium value in the patient database, in order to check for values that are far out of range.

13.1.2 A Time Delay After an Event

For example, five days after ordering gentamicin for a patient, in order to check renal function.

13.1.3 Periodically After an Event

For example, every five days after ordering gentamicin for a patient, in order to check renal function over a period of time.

13.2 Events

Events are distinct from data. An event may be an update or insertion in the patient database, a medically relevant occurrence, or an institution-defined occurrence. Examples include the storage of a serum potassium level, the ordering of a medication, the transferring of a patient to a new bed, and the recording of a new address for a patient.

13.2.1 Event Properties

The main attribute of an event is the time that it occurred, which must be an instant in time. Events have no values. Note the distinction between events and data. Data have values and have primary times, which are the times that are medically most relevant. For example, a serum potassium result may have a value of 5.0 and a primary time that is the time that it was drawn from the patient. But the **storage of serum potassium** event has no value, and its time is the time that the potassium was stored in the patient database.

13.2.2 Time of Events

The **time** operator (see Section 9.17) applied to an event results in the time that the event occurred. For example, **time of storage_of_potassium** returns the time that the potassium was stored. This value might be different from the time of the corresponding data value that is retrieved by a read mapping (the data value typically uses a clinically relevant time, which would often be different from the time of storing the data). **Eventtime** (see Section 8.4.4) is the time of the event that evoked the MLM.

13.2.3 Declaration of Events

Events are declared in the data slot as defined in Section 11.2.2.

13.3 Evoke Slot Statements:

13.3.1 Simple Trigger Statement

A simple trigger statement specifies an event or a set of events. When any of the events occurs, the MLM is triggered. Its form is:

```
<event-expr>
```

<event-expr> is an expression that contains only event variables as defined in Section 11.2.2, the **or** operator (see Section 9.4.1), the **any** operator (see Section 9.12.13), and parentheses. The keyword **`call`** may also be present, to indicate that the MLM may be called by another MLM.

13.3.1.1 Operation

Although events do not have values, they are used in this statement as if they were syntactically Boolean. Thus one ends up with a statement like this: **event1 OR event2 OR event3**. The MLM is triggered whenever an event occurs and any of the evoke statements evaluate to **true**. If more than one event occurs, the MLM may be triggered. No additional trigger criteria must be satisfied for the MLM to be evoked.

13.3.1.2 Examples

In the following examples, all the variables are event variables defined in the data slot.

```
penicillin_storage
penicillin_storage OR cephalosporin_storage
ANY OF (penicillin_storage,cephalosporin_storage,aminoglycoside_storage)


data:
   penicillin_storage := event {store penicillin order}
   cephalosporin_storage := event {store cephalosporin order}
;;
evoke:
   penicillin_storage OR
   cephalosporin_storage;;
```

13.3.2 Delayed Trigger Statement

A delayed trigger statement permits the MLM to be triggered some time after an event occurs. Its basic form is shown first, followed by a more specific example of the basic form:

```
<time-expr>
<duration-expr> AFTER TIME <event>
```

<time-expr> is an expression that contains only times expressed as time constants (see Section 7.1.5) or as the **time** operator (see Section 9.17) applied to event variables (see Section 11.2.2); durations expressed as duration operators (see Section 9.11) applied to number constants (see Section 7.1.4); and the **after** operator (see Section 9.10.1).

<duration-expr> is a duration constant formed by using a number constant with a duration operator.

<event> is an event variable.

13.3.2.1 Operation

The MLM is triggered at the time specified in the delayed trigger statement. This is usually some specified duration after the occurrence of an event. This statement can also be used to trigger an MLM once on a particular date using a time constant.

13.3.2.2 Examples

In the following examples, all variables are event variables:

```
3 days after time of penicillin_storage
1992-01-01T00:00:00
```

13.3.3 Periodic Trigger Statement

A periodic trigger statement permits the MLM to be triggered at specified time intervals after an event occurs. The cycles may continue for a specified duration, and they may be terminated by a Boolean condition. It has two forms:

```
EVERY <duration-expr> FOR <duration-expr> STARTING <time-expr>
EVERY <duration-expr> FOR <duration-expr> STARTING <time-expr> UNTIL
    <Boolean-expr>
```

<duration-expr> is a duration constant formed by using a number constant (see Section 7.1.4) with a duration operator (see Section 9.11).

<time-expr> is a time expressed as a time constant (see Section 7.1.5) or as the **time** operator (see Section 9.17) applied to an event variable (see Section 11.2.2).

<Boolean-expr> is any valid expression. It is usually a Boolean expression that becomes **true** when the MLM triggering should stop.

13.3.3.1 Operation

The MLM is first triggered at the time specified after the **starting** word. It is then triggered repeatedly in cycles of length equal to the duration specified after the **every** word. These cycles continue for the duration specified after the **for** word. The **for** duration is inclusive, so **every 1 day for 1 day starting 3 days after time of event1** would trigger the MLM twice: at three days and at four days after the event.

13.3.3.2 Until

If there is an **until** clause, then it is evaluated as soon as the MLM is triggered; the clause may contain references to the patient database unrelated to the event. If it is **true** then the MLM exits immediately, and no further triggering occurs. Otherwise, the MLM is executed, and it is triggered again after the **every** duration (assuming the **for** duration has not run out).

13.3.3.3 Examples

In the following examples, variables beginning with **event** are event variables:

```
every 1 day for 14 days starting 1992-01-01T00:00:00
every 1 day for 14 days starting time of event1
every 2 hours for 1 day starting 5 hours after the time of event2
every 1 week for 1 month starting 3 days after the time of event3 until
    last(serum_potassium) > 5.0
```

13.3.4 Where Trigger Statement

A **where** trigger statement contains a simple trigger statement along with a **where** phrase that provides additional criteria that must be satisfied for the MLM to be evoked. Its form is the same as the simple trigger statement except that the event variables in **<event-expr>** may be replaced with:

```
<event> WHERE <Boolean-expr>
```

<event> is an event variable as defined in 11.2.2.

<Boolean-expr> is an expression whose result is a Boolean value and whose variables are the result of read statements. This construct usually contains a Boolean restriction on data that are stored as part of a data storage event.

Or and **any** are permitted as in the simple trigger statement.

13.3.4.1 Operation

The MLM is triggered whenever an event occurs and its corresponding **where** clause results in **true**. If the **where** clause results in anything else, the MLM is not triggered.

13.3.4.2 Example

In this example, the data and evoke slots are both shown.

```
data:
    penicillin_storage := event {store penicillin order};
    penicillin_dose := read last {dose of stored penicillin order};
    cephalosporin_storage := event {store cephalosporin order};
    cephalosporin_dose := read last {dose of stored cephalosporin order};
    ;;
evoke:
    (penicillin_storage WHERE penicillin_dose > 500)
    OR
    (cephalosporin_storage WHERE cephalosporin_dose > 500);;
```

13.3.4.3

The **where** trigger statement is provided only for systems that must use it for execution efficiency. It is recommended that it not be used, if possible. Instead, put all logical constraints in the logic slot. The above example would become (assuming that a query for the dosage of a medication that was not stored is **null**):

```
data:
    penicillin_storage := event {store penicillin order};
    penicillin_dose := read last {dose of stored penicillin order};
    cephalosporin_storage := event {store cephalosporin order};
    cephalosporin_dose := read last {dose of stored cephalosporin order};
    ;;
evoke:
    penicillin_storage OR cephalosporin_storage;;
logic:
    IF penicillin_dose > 500 OR cephalosporin_dose > 500 THEN …
    ENDIF;;
```

13.4 Evoke Slot Usage

The evoke slot usually contains a single statement that specifies when an MLM is triggered. If the evoke slot has more than one statement, then the MLM is evoked whenever any of the criteria in any of the statements occurs.

Annexes
(Mandatory Information)

A1 BACKUS-NAUR FORM

The MLM syntax is defined using Backus-Naur Form (BNF) **(3).** In the interest of readability and computability, the context free grammar is expressed in Backus-Naur Form rather than in the more compact Extended Backus-Naur Form (EBNF) **(3)**. The following definitions hold:

<expression> **-** represents the non-terminal expression

"IF" **-** represents the terminal **if**, **iF**, **If**, or **IF**

":=" **-** represents the terminal **:=**

::= **-** is defined as

/*...*/ **-** a comment about the grammar

| **-** or

Terminals are listed in uppercase, but the language is case insensitive outside of character strings. In structured slots, space, carriage return, line feed, horizontal tab, vertical tab, and form feed are considered white space and are ignored. in addition, the terminal **the** is treated as white space (that is, the word **the** is ignored).
With minor modifications, the following grammar can be processed by an LALR(1) parser generator, except where noted by comments against individual rules

```
/****** physical file containing one or more MLMs ******/

/****** file of individual MLMs ******/

<mlms> ::=
         <mlm>
       | <mlm> <mlms>

/****** categories ******/

<mlm> ::=
         <maintenance_category>
         <library_category>
         <knowledge_category>
         "END:"

<maintenance_category> ::=
         "MAINTENANCE:" <maintenance_body>

<maintenance_body> ::=
         <title_slot>
         <mlmname_slot>
         <arden_version_slot>
         <version_slot>
         <institution_slot>
         <author_slot>
         <specialist_slot>
         <date_slot>
         <validation_slot>

<library_category> ::=
         "LIBRARY:" <library_body>

<library_body> ::=
         <purpose_slot>
         <explanation_slot>
         <keywords_slot>
         <citations_slot>
         <links_slot>

<knowledge_category> ::=
         "KNOWLEDGE:" <knowledge_body>
```

```
        <knowledge_body> ::=
                <type_slot>
                <data_slot>
                <priority_slot>
                <evoke_slot>
                <logic_slot>
                <action_slot>
                <urgency_slot>
```

/****** slots ******/

/****** maintenance slots ******/

```
    <title_slot> ::=
            "TITLE:" <text> ";;"

    <mlmname_slot> ::=
            "MLMNAME:" <identifier> ";;"
          | "FILENAME:" <identifier> ";;"
                                    /* the "FILENAME:" form is only valid */
                                    /* combination with the empty version */
                                    /* of <arden_version_slot>            */

    <arden_version_slot> ::=
            "ARDEN:" <arden_version> ";;"
          | /*empty*/
                                    /* the empty version is only valid    */
                                    /* combination with the "FILENAME"    */
                                    /* form of < mlmname_slot >           */

    <arden_version> ::=
            "VERSION" "2"

    <version_slot> ::=
            "VERSION:" <mlm_version> ";;"

    <mlm_version>
            <text>

    <institution_slot> ::=
            "INSTITUTION:" <text> ";;"      /* text limited to 80 characters */

    <author_slot> ::=
            "AUTHOR:" <text> ";;"           /* see 6.1.6 for details */

    <specialist_slot> ::=
            "SPECIALIST:" <text> ";;"       /* see 6.1.7 for details */

    <date_slot> ::=
            "DATE:" <mlm_date> ";;"

    <mlm_date> ::=
            <iso_date>
          | <iso_date_time>

    <validation_slot> ::=
            "VALIDATION:" <validation_code> ";;"

        <validation_code> ::=
                "PRODUCTION"
              | "RESEARCH"
              | "TESTING"
              | "EXPIRED"
```

/****** library slots ******/

```
    <purpose_slot> ::=
            "PURPOSE:" <text> ";;"

    <explanation_slot> ::=
            "EXPLANATION:" <text> ";;"

    <keywords_slot> ::=
            "KEYWORDS:" <text> ";;"
```

/* May require special processing to handle both list and text versions */

```
    <citations_slot> ::=
            /* empty */
          | "CITATIONS:" <citations_list> ";;"
          | "CITATIONS:" <text> ";;"/* deprecated -- supported for backward
        compatibility */

    <citations_list> ::=
            /* empty */
          | <single_citation>
          | <single_citation> ";" <citations_list>
```

```
            <single_citation> ::=
                    <digits> "." <citation_type> <citation_text>
                  | <citation_text>
```

/* This is a separate definition to allow for future expansion */

```
                <citation_text>
                       <string>

                <citation_type> ::=
                         /* empty */
                       | "SUPPORT"
                       | "REFUTE"
```

/* May require special processing to handle both list and text versions */

```
            <links_slot> ::=
                    /* empty */
                  | "LINKS:" <links_list> ";;"
                  | "LINKS:" <text> ";;"    /* deprecated - supported for backward*/
                                            /* compatibility */

            <links_list> ::=
                    /* empty */
                  | <single link>
                  | <links_list> ";" <single link>

            <single_link> ::=
                  <link_type> <link_name> <link_text>

            <link_type> ::=
                  /* empty */
                  | "URL_LINK"
                  | "MESH_LINK"
                  | "OTHER_LINK"
                  | "EXE_LINK"

                <link_name> ::=
                   /* empty */
                   | <string>
```

/* This is a separate definition to allow for future expansion */

```
            <link_text> ::=
                  <term>
```

/****** knowledge slots ******/

```
            <type_slot> ::=
                    "TYPE:" <type_code> ";;"
```

/* This is a separate definition to allow for future expansion */

```
                <type_code> ::=
                       "DATA_DRIVEN"
                       | "DATA-DRIVEN"    /* deprecated -- supported for backwards */
                                          /* compatibility */

            <data_slot> ::=
                "DATA:" <data_block> ";;"

            <priority_slot> ::=
                    /* empty */
                  | "PRIORITY:" <number> ";;"

            <evoke_slot> ::=
                    "EVOKE:" <evoke_block> ";;"

            <logic_slot> ::=
                    "LOGIC:" <logic_block> ";;"

            <action_slot> ::=
                    "ACTION:" <action_block> ";;"

            <urgency_slot> ::=
                    /* empty */
                  | "URGENCY:" <urgency_val> ";;"

            <urgency_val> ::=
                    <number>
                  | <identifier>
```

/****** logic block ******/

```
            <logic_block> ::=
                    <logic_block> ';' <logic_statement>
                  | <logic_statement>
```

```
<logic_statement> ::=
        /* empty */
      |  <logic_assignment>
      |  "IF" <logic_if_then_else2>
      |  "FOR" <identifier> "IN" <expr> "DO" <logic_block> ";" "ENDDO"
      |  "WHILE" <epxr> "DO" <logic_block> ";" "ENDDO"
      |  "CONCLUDE" <expr>

<logic_if_then_else2> ::=
        <expr> "THEN" <logic_block> ";" <logic_elseif> ";"

<logic_elseif> ::=
        "ENDIF"
      | "ELSE" <logic_block> ";" "ENDIF"
      | "ELSEIF" <logic_if_then_else2>

<logic_assignment> ::=
        <identifier_becomes> <expr>
      | <time_becomes> <expr>
      | <identifier_becomes> <call_phrase>

<identifier_becomes> ::=
        <identifier> ":="
      | "LET" <identifier> "BE"
      | "NOW" ":="

<time_becomes> ::=
        "TIME" "OF" <identifier> :=
      | "TIME" <identifier> :=
      | "LET" "TIME" "OF" <identifier> "BE"
      | "LET" "TIME" <identifier> "BE"

    <call_phrase> ::=
            "CALL" <identifier>
          | "CALL" <identifier> "WITH" <expr>
```

/****** expressions ******/

```
<expr> ::=
        <expr_sort>
      | <expr> "," <expr_sort>

<expr_sort> ::=
        <expr_where>
      | <expr_where> "MERGE" <expr_sort>
      | "SORT" "DATA" <expr_sort>
      | "SORT" "TIME" <expr_sort>

<expr_where> ::=
        <expr_range>
      | <expr_range> "WHERE" <expr_range>

<expr_range> ::=
        <expr_or>
      | <expr_or> "SEQTO" <expr_or>

<expr_or> ::=
        expr_or OR expr_and
      | expr_and

<expr_and> ::=
        expr_and AND expr_not
      |  expr_not

<expr_not> ::=
        NOT expr_comparison
      | expr_comparison

<expr_comparison> ::=
        <expr_string>
      | <expr_string> <simple_comp_op> <expr_string>
      | <expr_string> <is> <main_comp_op>
      | <expr_string> <is> "NOT" <main_comp_op>
      | <expr_string> <occur> <temporal_comp_op>
      | <expr_string> <occur> "NOT" <temporal_comp_op>
      | <expr_string> "MATCHES" "PATTERN" <expr_string>

<expr_string> ::=
        <expr_plus>
      | <expr_string> "||" <expr_plus>
      | <expr_plus> "FORMATTED" "WITH" <format_string>
```

```
<format_string> ::=
         """ <format_specification> """  /* The format string is a true     */
                                         /* Arden Syntax string, enclosed   */
                                                                            */
       | <string>

<format_specifiation> ::=                /*  See section 9.8.2 (p. 34) and  */
                                         /* Annex 5 for explanation of      */
                                         /* valid combinations and meanings.*/
         <format_specificaton> <format_specification_single>
       | <format_specification_single>

<format_specification_single>
         "%"<format_options><format_flag><width_precision>
       /*  No spaces are permitted between elements in above form */
       | <text>

<format_options> ::=
       /* empty */
       | "+"
       | "-"
       | "0"
       | " "  /* space */
       | "#"

<format_flag> ::=     /* Format flags are case sensitive */
         "c"
       | "C"
       | "d"
       | "I"
       | "o"
       | "u"
       | "x"
       | "X"
       | "e"
       | "E"
       | "f"
       | "g"
       | "G"
       | "n"
       | "p"
       | "s"
       | "t"

<width_precision> ::=
       /* empty */
       | <digits>
       | <digits>"."<digits>

<expr_plus> ::=
         <expr_times>
       | <expr_plus> "+" <expr_times>
       | <expr_plus> "-" <expr_times>
       | "+" <expr_times>
       | "-" <expr_times>

<expr_times> ::=
         <expr_power>
       | <expr_times> "*" <expr_power>
       | <expr_times> "/" <expr_power>

<expr_power> ::=
         <expr_before>
       | <expr_function> "**" <expr_function>

<expr_before> ::=
         <expr_ago>
       | <expr_ago> "BEFORE" <expr_ago>
       | <expr_ago> "AFTER" <expr_ago>

<expr_ago> ::=
         <expr_function>
       | <expr_duration>
       | <expr_duration> "AGO"

<expr_duration> ::=
         <expr_function> <duration_op>
```

```
<expr_function> ::=
        <expr_factor>
      | <of_func_op> <expr_function>
      | <of_func_op> "OF" <expr_function>
      | <from_of_func_op> <expr_function>
      | <from_of_func_op> "OF" <expr_function>
      | <from_of_func_op> <expr_factor> "FROM" <expr_function>
      | <from_func_op> <expr_factor> "FROM" <expr_function>
      | <index_from_of_func_op> <expr_function>
      | <index_from_of_func_op> "OF" <expr_function>
      | <index_from_of_func_op> <expr_factor> "FROM" <expr_function>
      | <index_from_func_op> <expr_factor> "FROM" <expr_function>
      | <expr_factor> "AS" <as_func_op>
   <expr_factor> ::=
        <expr_factor_atom>
      | <expr_factor_atom> "[" <expr> "]"      /* number [<expr>] is not */
                                               /* a valid construct      */

<expr_factor_atom> ::=
        <identifier>
      | <number>
      | <string>
      | <time_value>
      | <boolean_value>
      | "NULL"
      | <it>                        /* Value is NULL outside of a where   */
                                    /* clause and may be flagged as an    */
                                    /* error in some implementations.     */
      | "(" ")"
      | "(" <expr> ")"
```

/****** for readability *******/

```
<it> ::= "IT" | "THEY"
```

/****** comparison synonyms ******/

```
<is> ::= "IS" | "ARE" | "WAS" | "WERE"

<occur> ::= "OCCUR" | "OCCURS" | "OCCURRED"
```

/****** operators ******/

```
<simple_comp_op> ::=
        "="  | "EQ"
      | "<"  | "LT"
      | ">"  | "GT"
      | "<=" | "LE"
      | ">=" | "GE"
      | "<>" | "NE"

<main_comp_op> ::=
        <temporal_comp_op>
      | <range_comp_op>
      | <unary_comp_op>
      | <binary_comp_op> <expr_string>
```

/* the WITHIN TO operator will accept any ordered parameter, */
/* including numbers, strings (single characters), times, Boolean /*

```
<range_comp_op> ::=
      "WITHIN" <expr_string> "TO" <expr_string>

<temporal_comp_op> ::=
        "WITHIN" <expr_string> "PRECEDING" <expr_string>
      | "WITHIN" <expr_string> "FOLLOWING" <expr_string>
      | "WITHIN" <expr_string> "SURROUNDING" <expr_string>
      | "WITHIN" "PAST" <expr_string>
      | "WITHIN" "SAME" "DAY" "AS" <expr_string>
      | "BEFORE" <expr_string>
      | "AFTER" <expr_string>
      | "EQUAL" <expr_string>

<unary_comp_op> ::=
        "PRESENT"
      | "NULL"
      | "BOOLEAN"
      | "NUMBER"
      | "TIME"
      | "DURATION"
      | "STRING"
      | "LIST"
```

```
<binary_comp_op> ::=
         "LESS" "THAN"
       | "GREATER" "THAN"
       | "GREATER" "THAN" "OR" "EQUAL"
       | "LESS" "THAN" "OR" "EQUAL"
       | "IN"

    <of_func_op> ::=
             <of_read_func_op>
           | <of_noread_func_op>

<of_read_func_op> ::=
         "AVERAGE"| "AVG"
       | "COUNT"
       | "EXIST"   | "EXISTS"
       | "SUM"
       | "MEDIAN"


<of_noread_func_op> ::=
         "ANY"
       | "ALL"
       | "NO"
       | "SLOPE"
       | "STDDEV"
       | "VARIANCE"
       | "INCREASE"
       | "PERCENT" "INCREASE" | "%" "INCREASE"
       | "DECREASE"
       | "PERCENT" "DECREASE" | "%" "DECREASE"
       | "INTERVAL"
       | "TIME"
       | "ARCCOS"
       | "ARCSIN"
       | "ARCTAN"
       | "COSINE"  | "COS"
       | "SINE"    | "SIN"
       | "TANGENT"| "TAN"
       | "EXP"
       | "FLOOR"
       | "INT"
       | "ROUND"
       | "CEILING"
       | "TRUNCATE"
       | "LOG"
       | "LOG10"
       | "ABS"
       | "SQRT"
       | "EXTRACT" "YEAR"
       | "EXTRACT" "MONTH"
       | "EXTRACT" "DAY"
       | "EXTRACT" "HOUR"
       | "EXTRACT" "MINUTE"
       | "EXTRACT" "SECOND"
       | "TIME"
       | "STRING"
       | "EXTRACT" "CHARACTERS"

<from_func_op> ::=
         "NEAREST"

<index_from_func_op> ::=
         "INDEX" "NEAREST"

<from_of_func_op> ::=
         <from_of_read_func_op>
       | <from_of_noread_func_op>

<from_of_read_func_op>
         "MINIMUM"| "MIN"
       | "MAXIMUM"| "MAX"
       | "LAST"
       | "FIRST"
       | "EARLIEST"
       | "LATEST"

<from_of_noread_func_op>
         "REVERSE"           /* must be careful as reverse 5 from x does */
                             /* not make sense                           */
```

```
            <index_from_of_func_op> ::=
                    "INDEX" "MINIMUM"    | "INDEX" "MIN"
                  | "INDEX" "MAXIMUM"    | "INDEX" "MAX"
                  | "INDEX" "EARLIEST"
                  | "INDEX" "LATEST"

            <as_func_op> ::=
                    "NUMBER"

                <duration_op> ::=
                        "YEAR"    | "YEARS"
                      | "MONTH"   | "MONTHS"
                      | "WEEK"    | "WEEKS"
                      | "DAY"     | "DAYS"
                      | "HOUR"    | "HOURS"
                      | "MINUTE"  | "MINUTES"
                      | "SECOND"  | "SECONDS"
```

/****** factors ******/

```
            <boolean_value> ::=
                    "TRUE"
                  | "FALSE"

            <time_value> ::=
                    "NOW"
                  | <iso_date_time>
                  | <iso_date>
                  | "EVENTTIME"
                  | "TRIGGERTIME"
```

/****** data block ******/

```
            <data_block> ::=
                    <data_block> ';' <data_statement>
                  | <data_statement>

            <data_statement> ::=
                  /* empty */
                  | <data_assignment>
                  | "IF" <data_if_then_else2>
                  | "FOR" <identifier> "IN" <expr> "DO" <data_block> ";" "ENDDO"
                  | "WHILE" <epxr> "DO" <data_block> ";" "ENDDO"

            <data_if_then_else2> ::=
                    <expr> "THEN" <data_block> ";" <data_elseif>

            <data_elseif> ::=
                    "ENDIF"
                  | "ELSE" <data_block> ";" "ENDIF"
                  | "ELSEIF" <data_if_then_else2>

            <data_assignment> ::=
                    <identifier_becomes> <data_assign_phrase>
                  | "(" <data_var_list> ")" ":=" "READ" <read_phrase>
                  | "LET" "(" <data_var_list> ")" "BE" "READ" <read_phrase>
                  | "(" <data_var_list> ")" ":=" "ARGUMENT"
                  | "LET" "(" <data_var_list> ")" "BE" "ARGUMENT"

            <data_var_list> ::=
                    <identifier>
                  | <identifier> "," <data_var_list>

                <data_assign_phrase> ::=
                        "READ" <read_phrase>
                      | "MLM" <term>
                      | "MLM" <term> "FROM" "INSTITUTION" <string>
                      | "MLM" "MLM_SELF"
                      | "INTERFACE" <mapping_factor>
                      | "EVENT" <mapping_factor>
                      | "MESSAGE" <mapping_factor>
                      | "DESTINATION" <mapping_factor>
                      | "ARGUMENT"
                      | <call_phrase>
                      | <expr>

            <read_phrase> ::=
                    <read_where>
                  | <of_read_func_op> <read_where>
                  | <of_read_func_op> "OF" <read_where>
                  | <from_of_read_func_op> <read_where>
                  | <from_of_read_func_op> "OF" <read_where>
                  | <from_of_read_func_op> <expr_factor> "FROM" <read_where>
```

```
<read_where> ::=
        <mapping_factor>
      | <mapping_factor> "WHERE" <it> <occur> <temporal_comp_op>
      | <mapping_factor> "WHERE" <it> <occur> "NOT" <temporal_comp_op>
      | "(" <read_where> ")"


<mapping_factor> ::=
        "{" <data_mapping> "}"
```

/****** evoke block ******/

```
<evoke_block> ::=
        <evoke_statement>
      | <evoke_block> ";" <evoke_statement>

<evoke_statement> ::=
      /* empty */
      | <event_or>
      | <evoke_time>
      | <qualified_evoke_cycle>
      | <event_where>
      | "CALL"           /* deprecated -- kept for backward compatibility */

<event_list> ::=
        <event_or>
      | <event_list> "," <event_or>

<event_or> ::=
        <event_or> "OR" <event_any>
      | <event_any>

<event_any> ::=
        "ANY" "(" <event_list> ")"
        "ANY" "OF" "(" <event_list> ")"
      | "ANY" <identifier>
      | "ANY" "OF" <identifier>
      | <event_factor>

<event_factor> ::=
        "(" <event_or> ")"
      | <identifier>

<event_where> ::=
      <event_or> "WHERE" <expr>     /* this expression must evaluate to a  */
                                    /* boolean value (see 13.3.4)
   */

<evoke_time> ::=
        <evoke_duration> "AFTER" <evoke_time>
      | "TIME" <event_any>
      | "TIME" "OF" <event_any>
      | <iso_date_time>
      | <iso_date>

<qualified_evoke_cycle> ::=
        <simple_evoke_cycle>
      | <simple_evoke_cycle> "UNTIL" <expr>

<simple_evoke_cycle> ::=
        "EVERY" <evoke_duration> "FOR" <evoke_duration> "STARTING"
   <evoke_time>

<evoke_duration> ::=
        <number> <duration_op>
```

/****** action block ******/

```
<action_block> ::=
        <action_statement>
      | <action_block> ";" <action_statement>

<action_statement> ::=
       /* empty */
      | "IF" <action_if_then_else2>
      | "FOR" <identifier> "IN" <expr> "DO" <action_block> ";" "ENDDO"
      | "WHILE" <epxr> "DO" <action_block> ";" "ENDDO"
      | <call_phrase>
      | <call_phrase> "DELAY" <expr>
      | "WRITE" <expr>
      | "WRITE" <expr> "AT" <identifier>
      | "RETURN" <expr>

<action_if_then_else2> ::=
      <expr> "THEN" <action_block> ";" <action_elseif>
```

```
<action_elseif> ::=
        "ENDIF"
      | "ELSE" <action_block> ";" "ENDIF"
      | "ELSEIF" <action_if_then_else2>
```

/****** lexical constructs ******/

```
<string> ::=
        /* any string of characters enclosed in double quotes (" ASCII 22)
    with nested "" but without ";;" */

<identifier> ::=
      /* up to 80 characters total (no reserved words allowed) */
        <letter> <identifier_rest>

<identifier_rest> ::=        /* no spaces are permitted between elements */
        /* empty */
      | <letter> <identifier_rest>
      | <digit> <identifier_rest>
      | "_" <identifier_rest>

<text> ::=
        /* any string of characters without ";;" */

<number> ::=                 /* no spaces are permitted between elements */
        <digits> <exponent>
      | <digits> "." <exponent>
      | <digits> "." <digits> <exponent>
      | "." <digits> <exponent>

<exponent> ::=               /* no spaces are permitted between elements */
        /* null */
        | <e> <sign> <digits>

<e> ::=
        "E"
      | "e"

<sign> ::=
        /* null */
      | "+"
      | "-"

<digits> ::=                 /* no spaces are permitted between elements */
        <digit>
      | <digit> <digits>

<digit> ::=
        "0"
      | "1"
      | "2"
      | "3"
      | "4"
      | "5"
      | "6"
      | "7"
      | "8"
      | "9"

<letter> ::=
        "a"  | "b"  | "c"  | "d"
      | "e"  | "f"  | "g"  | "h"
      | "i"  | "j"  | "k"  | "l"
      | "m"  | "n"  | "o"  | "p"
      | "q"  | "r"  | "s"  | "t"
      | "u"  | "v"  | "w"  | "x"
      | "y"  | "z"  |

         "A" | "B"  | "C"  | "D"
      | "E"  | "F"  | "G"  | "H"
      | "I"  | "J"  | "K"  | "L"
      | "M"  | "N"  | "O"  | "P"
      | "Q"  | "R"  | "S"  | "T"
      | "U"  | "V"  | "W"  | "X"
      | "Y"  | "Z"

<iso_date> ::=               /* no spaces are permitted between elements */
      <digit> <digit> <digit> <digit> "-" <digit> <digit> "-" <digit> <digit>

<iso_date_time> ::=          /* no spaces are permitted between elements */
      <digit> <digit> <digit> <digit> "-" <digit> <digit> "-" <digit> <digit>
   <t>
      <digit> <digit> ":" <digit> <digit> ":" <digit> <digit>
      <fractional_seconds>
      <time_zone>
```

```
<t> ::=
         "T"
       | "t"

<fractional_seconds> ::=                /* no spaces are permitted between
    elements */
         "." <digits>
       | /* empty */

<time_zone> ::=                     /* no spaces are permitted between
    elements */
         /* null */
       | <zulu>
       | "+" <digit> <digit> ":" <digit> <digit>
       | "-" <digit> <digit> ":" <digit> <digit>

<zulu> ::=
         "Z"
       | "z" )

<term> ::=
       /* any string of characters enclosed in single quotes (' , ASCII 44)
    without ";;" */

<data_mapping> ::=
       /* any balanced string of characters enclosed in curly brackets { } */
       /* (ASCII 123 and 125, respectively) without ";;" the data mapping  */
       /* does not include the curly bracket characters                    */
```

A2 RESERVED WORDS

Listed here in alphabetic order are all the reserved words. None of these words may be used as variable names.

abs	destination	increase	month
action	do	index	months
after	duration	institution	ne
ago	earliest	int	nearest
alert	else	interface	no
all	elseif	interval	not
and	enddo	is	now
any	endif	it	null
arccos	end	keywords	number
arcsin	eq	knowledge	occur
arctan	equal	last	occurred
arden	event	latest	occurs
are	eventtime	le	of
argument	every	less	or
as	evoke	let	past
at	exist	library	pattern
author	exists	links	percent
average	exp	list	preceding
avg	expired	log	present
be	explanation	log10	priority
before	extract	logic	production
Boolean	false	lt	purpose
call	filename	maintenance	read
ceiling	first	matches	refute
characters	floor	max	research
citations	following	maximum	return
conclude	for	median	reverse
cos	formatted	merge	round
cosine	from	message	same
count	ge	min	second
data	greater	minimum	seconds
date	gt	minute	seqto
day	hour	minutes	sin
days	hours	mlm	sine
decrease	if	mlmname	slope
delay	in	mlm_self	sort

specialist	testing	truncate	weeks
sqrt	than	type	were
starting	the	unique	where
stddev	then	until	while
string	they	urgency	with
sum	time	validation	within
support	title	variance	write
surrounding	to	version	year
tan	triggertime	was	year
tangent	true	week	

The following identifiers are reserved for future use:

union	intersect	excluding	citation	select

A3 SPECIAL SYMBOLS

Listed here are all the special symbols.

||	:=	,	=	>=
>	<=	<	{	(
[	-	<>	%	+
}	)	]	;	#
/	*	**	;;	:
/*	*/	//	'	"

A4 OPERATOR PRECEDENCE AND ASSOCIATIVITY

A4.1

The operators for the structured slots are shown here grouped by precedence. Groups are separated by horizontal lines. Within groups, operators have equal precedence. Groups are arranged from lowest to highest precedence.

A4.2

Synonyms are listed on the same line, separated by °. The symbol [of] means that the word of is optional, and does not affect the logic of the operator.

A4.3

The position of the arguments relative to the operator is indicated by the ellipsis … The operator's associativity is shown in italics after each operator. Some operators have both a unary form (one argument) and a binary form (two arguments); each form is listed separately.

… , … (left associative)
… merge … (left associative)
sort data … (non-associative)
sort time … (non-associative)

… where … (non-associative)

… or … (left associative)

… and … (left associative)   … not … (non-associative)

… = … ° … is equal … (non-associative)
… <> … ° … is not equal … (non-associative)
… < … ° … is less than … ° … is not greater than or equal … (non-associative)
… <= … ° … is less than or equal … ° … is not greater than … (non-associative)
… > … ° … is greater than … ° … is not less than or equal … (non-associative)
… >= … ° … is greater than or equal … ° … is not less than (non-associative)
… is within … to … (non-associative)
… is not within … to … (non-associative)
… is within … preceding … (non-associative)
… is not within … preceding (non-associative)
… is within … following … (non-associative)   is not within … following … (non-associative)
… is within … surrounding … (non-associative)
… is not within … surrounding … (non-associative)
… is within past … (non-associative)
… is not within past … (non-associative)
… is within same day as … (non-associative)
… is not within same day as (non-associative)
… is before … (non-associative)
… is not before … (non-associative)
… is after … (non-associative)
… is not after … (non-associative)
… occur equal … (non-associative)
… occur within … to … (non-associative)
… occur not within … to … (non-associative)
… occur within … preceding … (non-associative)
… occur not within … preceding … (non-associative)
… occur within … following … (non-associative)
… occur not within … following … (non-associative)
… occur within … surrounding … (non-associative)
… occur not within … surrounding … (non-associative)
… occur within past … (non-associative)
… occur not within … past … (non-associative)

… occur within same day as … (non-associative)
… occur not within same day as … (non-associative)
… occur before … (non-associative)
… occur not before … (non-associative)
… occur after … (non-associative)
… occur not after … (non-associative)
… is in … (non-associative)
… is not in … (non-associative)
… is present ° … is not null (non-associative)
… is not present ° … is null (non-associative)
… is Boolean (non-associative)
… is not Boolean (non-associative)
… is number (non-associative)
… is not number (non-associative)
… is time (non-associative)
… is not time (non-associative)
… is duration (non-associative)
… is not duration (non-associative)
… is string (non-associative)
… is not string (non-associative)
… is list (non-associative)
… is not list (non-associative)

… … (left associative)
… formatted with … (non-associative)

+ … (non-associative)
… + … (left associative)
- … (non-associative)
… - … (left associative)

… * … (left associative)
… / … (left associative)

… ** … (non-associative)

… round … (non-associative)

… before … (non-associative)
… after … (non-associative)

… ago … (non-associative)

… year ° … years (non-associative)
… month ° … months (non-associative)
… week ° … weeks (non-associative)
… day ° … days (non-associative)
… hour ° … hours (non-associative)
… minute ° minutes (non-associative)
… second ° … seconds (non-associative)
… matches pattern … (non-associative)

count [of] … (right associative)
exist [of] … (right associative)
avg [of] … ° average [of] … (right associative)
median [of] … (right associative)
sum [of] … (right associative)
stddev [of] … (right associative)
variance [of] … (right associative)
any [of] … (right associative)
all [of] … (right associative)
no [of] … (right associative)
slope [of] … (right associative)
min … from … ° minimum … from … (right associative)
min [of] … ° minimum [of] … (right associative)

max … from … ° maximum … from … (right associative)
max [of] … ° maximum [of] … (right associative)
index min … from … ° index minimum … from … (right associative)
index min [of] … ° index minimum [of] … (right associative)
index max … from … ° index maximum … from … (right associative)
index max [of] … ° index maximum [of] … (right associative)
last … from … (right associative)
last [of] … (right associative)
first … from … (right associative)
first [of] … (right associative)
latest … from … (right associative)
latest [of] … (right associative)
earliest … from … (right associative)
earliest [of] … (right associative)
nearest … from … (right associative)
index nearest … from … (right associative)
increase [of] … (right associative)
decrease [of] … (right associative)
percent increase [of] … ° % increase [of] … (right associative)
percent decrease [of] … ° % decrease [of] … (right associative)
interval [of] … (right associative)
time [of] … (right associative)
arccos [of] … (right associative)
arcsin [of] … (right associative)
arctan [of] … (right associative)
cos [of] … ° cosine [of] … (right associative)
sin [of] … ° sine [of] … (right associative)
tan [of] … ° tangent [of] … (right associative)
exp [of] … (right associative)
floor [of] … (right associative)
ceiling [of] … (right associative)
truncate [of] … (right associative)
log [of] … (right associative)
log10 [of] … (right associative)
abs [of] … (right associative)
sqrt [of] … (right associative)
extract year [of] … (right associative)
extract month [of] … (right associative)
extract hour [of] … (right associative)
extract minute [of] … (right associative)
extract second [of] … (right associative)
reverse [of] … (right associative)
extract characters [of] … (right associative)
string [of] … (right associative)

A5 FORMAT SPECIFICATION (SEE 9.8.2)

A5.1 The following is a complete description of supported types within the format specification:

```
type   Required character that determines whether the associated argument is
    interpreted as a character, a string, or a number.
```

Table A5-1

Character	Type	Output Format
c	number	The number is assumed to represent a character code to be output as a character.
C	number	The number is assumed to represent a character code to be output as a character.
d	number	Signed decimal integer.
i	number	Signed decimal integer.
o	number	Unsigned octal integer.
u	number	Unsigned decimal integer.
x	number	Unsigned hexadecimal integer, using "abcdef."
X	number	Unsigned hexadecimal integer, using "ABCDEF."
e	number	Signed value having the form [–]d.dddd e [sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or – .
E	number	Identical to the e format, except that E, rather than e, introduces the exponent.
f	double	Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	double	Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).
n	Not supported.	Not supported.
p	Not supported.	Not supported.
s	string	Specifies a character. Characters are printed until the precision value is reached.
t	time	A time is printed based on the user's environment settings and the precision value.

A5.2 The optional fields, which appear before the type character, control other aspects of the formatting, as follows:

flags Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.

Table A5-2

Flag	Meaning	Default
–	Left align the result within the given field width.	Right align.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).

Flag	Meaning	Default
0	If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (I, u, x, X, o, d) the 0 is ignored.	No padding.
Space	Prefix the output value with a space if the output value is signed and positive; the space is ignored if both the space and + flags appear.	No space appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
#	When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
#	Ignored when used with c, d, i, u, or s.	

The second optional field of the format specification is the width specification. The width argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values - depending on whether the – flag (for left alignment) is specified - until the minimum width is reached. If width is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers).
The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if width is not given, all characters of the value are printed (subject to the precision specification).
If the width specification is an asterisk (*), an integer argument from the argument list supplies the value. The width argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause the truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

width Optional number that specifies the minimum number of characters output.

precision Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values.

Table A5-3

Type	Meaning	Default
c, C	The precision has no effect.	Character is printed.
d, i, u, o, x, X	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision.	Default precision is 1.
e, E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if precision is 0, or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed. The last printed digit is rounded.	Six significant digits are printed, with any trailing zeros truncated.

Type	Meaning	Default
s	The precision specifies the maximum number of characters to be printed. Characters in excess of precision are not printed.	Characters are printed until a null character is encountered.
t	The precision specifies how many of the date and time fields are printed. Non-printed fields are truncated (rounded down). 0: Year only 1: Year, Month 2: Date (Year, Month, Day) 3: Date, hour 4: Date, hour, minute 5: Date, hour, minute, second	All fields are printed.

Appendices

(Nonmandatory Information)

X1 SAMPLE MLMS

The following are sample MLMS to be used only to demonstrate the syntax. They have not been tested, and they have not been used in clinical care. More example MLMs, many of which are in current live use, are available at Columbia's web site, www.cpmc.columbia.edu/resources/arden.

X1.1 Data Interpretation MLM:

```
maintenance:

    title: Fractional excretion of sodium;;

    mlmname: fractional_na;;

    arden: Version 2;;

    institution: Columbia-Presbyterian Medical Center;;

    author:   George Hripcsak, M.D.
              (hripcsak@cucis.cis.columbia.edu);;

    specialist: ;;

    date: 1991-03-13;;

    validation: testing;;

library:

    purpose:
           Calculate the fractional excretion of sodium whenever urine:
           electrolytes are stored.  (This MLM demonstrates data
           interpretation across independent laboratory results.);;

    explanation:
           The fractional excretion of sodium is calculated from the urine
           sodium and creatinine and the most recent serum sodium and
           creatinine (where they occurred within the past 24 hours).  A
           value less than 1.0 % is considered low.;;

    keywords: fractional excretion; serum sodium; azotemia;;

    citations:
           1. Steiner RW.  Interpreting the fractional excretion of sodium.
              Am J Med 1984;77:699-702.;;

knowledge:

    type: data-drivein;;

data:
           let (urine_ na, urine_creat) be read last
               ({urine electrolytes where evoking}
               where they occurred within the past 24 hours) ;
           let (serum_na, serum_creat) be read last
               ({serum electrolytes where evoking}
               where they occurred within the past 24 hours) ;
           let urine_electrolyte_storage be event
               {storage of urine electrolytes}

evoke:
           urine_electrolyte_storage;;
```

```
logic:
          /* calculate fractional excretion of sodium */
          let fractional_na be 100 * (urine_na / urine_creat)/
                              (serum_na / serum_creat) ;
          /* if the fractional Na is invalid (e.g., if the */
          /* urine or serum sample is QNS) then stop here */
          if fractional_na is null then
              conclude false ;
          endif ;
          /* check whether the fractional Na is low */
          let low_fractional_na be fractional_na < 1.0 ;
          /* send the message */
          conclude true ;
          ;;

    action:
          if low_fractional_na then
              write "The calculated fractional excretion of sodium is low ("
                     || fractional_na || "). If the patient is azotemic, " ||
                    "this number may indicate volume depletion, " ||
                    "hepatic failure, congestive heart failure, acute " ||
                    "glomerulonephritis, oliguric myoglobinuric or " ||
                    "hemoglobinuric renal failure, oliguric contrast " ||
                    "nephrotoxicity, polyuric renal failure with severe " ||
                    "burns, renal transplant rejection, 10 % of cases " ||
                    "with non-oliguric acute tubular necrosis, and " ||
                    "several other forms of renal injury."
          else:
              write "The calculated fractional excretion of sodium is " ||
                    "not low ("|| fractional_na || ").  If the patient " ||
                    "is azotemic, this may indicate: acute renal " ||
                    "parenchymal injury, volume depletion coexisting " ||
                    "with diuretic use or pre-existing chronic renal " ||
                    "disease, and up to 10 % of cases of uncomplicated " ||
                    "volume depletion."
          endif
          ;;

end
```

X1.2 Research Study Screening MLM:

```
maintenance:

    title: Screen for hypercalcemia for Dr. B.'s study;;

    mlmname: hypercalcemia_for_b;;

    arden: Version 2;;

    version: 2.02;;

    institution: Columbia-Presbyterian Medical Center;;

    author: George Hripcsak, M.D.;;

    specialist: ;;

    date: 1990-12-04;;

    validation: research;;

library:

    purpose:
          Screen for hypercalcemia for Dr. B.'s study. (This MLM demonstrates
          screening patients for clinical trials.);;

    explanation:
          The storage of a serum calcium value evokes this MLM. If a serum
          albumin is available from the same blood sample as the calcium,
          then the corrected calcium is calculated, and patients with actual
          or corrected calcium greater than or equal 11.5 are accepted; if
          such a serum albumin is not available, then patients with actual
          calcium greater than or equal 11.0 are accepted.  Patients with
          serum creatinine greater than 6.0 are excluded from the study.;;

    keywords: hypercalcemia;;

    citations: ;;

    knowledge:

       type: data-driven;;
```

```
data:
          /* the storage of a calcium value evokes this MLM */
          storage_of_calcium " = event {'06210519','06210669'}
          /* total calcium in mg/dL */
          calcium := read last {'06210519','06210669','CALCIUM'}
          /* albumin in g/dL */
          evoking_albumin := read last {'06210669';'ALBUMIN' where evoking}
          /*albumin in g/dL; not necessary from same test as Ca */
          last_albumin := read last ({'06210669';'ALBUMIN'}
              where it occurred within the past 2 weeks)
          /* creatinine in mg/dL; not necessarily from same test as Ca */
          creatinine := read last ({'06210669','06210545','06000545','CREAT'}
          where it occurred within the past 2 weeks
          ;;

evoke
          storage_of_calcium;

logic:
          /* make sure the Ca is present (vs. hemolyzed , …) */
          IF calcium is not present THEN
              conclude false
          ENDIF
          /* if creatinine is present and greater than 6, then stop now */
          IF creatinine is present THEN
              IF creatinine is greater than 6.0 THEN
                  conclude false
              ENDIF
          ENDIF
          /* is an albumin present for the same sample as the calcium */
          IF evoking_albumin is present THEN
          /* calculate the corrected calcium */
              IF evoking_albumin is less than 4.0 THEN
                  corrected_calcium := calcium + (4.0-evoking_albumin)*0.8
              ELSE
                  /* corrected is never less than actual */
                  corrected_calcium := calcium
              ENDIF
              /* test for total or corrected calcium >= 11.5 */
              IF calcium >= 11.5 OR corrected_calcium >= 11.5 THEN
                  message := "calcium = " || calcium ||
                             " on " || time of calcium ||
                             " (corrected calcium = " ||
                             corrected calcium || ")"
                  message := message||"; albumin = " ||evoking_albumin
                  IF creatinine is present THEN
                      message := message||
                                 "; last creatinine = ||creatinine
                      message := message||
                                 "; (total or corrected calcium " ||
                                 "was at least 11.5)"
                      conclude true
                  ELSE
                      conclude false
                  ENDIF
              ENDIF
          /* no evoking albumin was present */
          ELSE
          /* check for true calcium >= 11.0 */
              IF calcium >= 11.0 THEN
                  message := "calcium = "||calcium||" on "||time of calcium
                  IF last_albumin is present THEN
                      message := message||"; last albumin "||
                              "(not from same blood sample as calcium) = "||
                              last_albumin
                      IF creatinine is present THEN
                          message := message|| "; last creatinine = "
                                     ||creatinine
                          message := message||
                                     "; (total calcium was at least 11.0; "||
                                     "corrected calcium was not calculated)" ;
                          conclude true
                      ELSE
                          conclude false
                      ENDIF
                  ENDIF
              ENDIF
          ENDIF
      ;;

    action: write "hypercalcemia study: " || message;;

    urgency: 50;;
```

```
end:
```

X1.3 Contraindication Alert MLM:

```
maintenance:
    title: Check for penicillin allergy;;
    mlmname: pen_allergy;;
    arden: ASTM-E1460-1995;;
    version: 1.00;
    institution: Columbia-Presbyterian Medical Center;;
    author: George Hripcsak, M.D.;;
    specialist: ;;
    date: 1991-03-18;;
    validation: testing;;
library:
    purpose:
           when a penicillin is prescribed, check for an allergy.  (This MLM
           demonstrates checking for contraindications.);;
    explanation:
           This MLM is evoked when a penicillin medication is ordered. An
           alert is generated because the patient has an allergy to penicillin
           recorded.;;
    keywords: penicillin; allergy;;
    citations: ;;
knowledge:
    type: data-driven;;
    data:
           /* an order for a penicillin evokes this MLM */
           penicillin_order := event {medication order where
                                      class = penicillin};
           /* find allergies */
           penicillin_allergy := read last {allergy where
                                            agent_class = penicillin};
           ;;
    evoke:
           penicillin_order;;
    logic
           if exist (penicillin_allergy) then
               conclude true;
           endif;
           ;;
    action:
           write "Caution, the patient has the following allerge to penicillin
    documented:   " || penicillin_allergy;;
    urgency: 50;;
end:
```

X1.4 Management Suggestion MLM:

```
maintenance:
    title: Dosing for gentamicin in renal failure;;
    mlmname: gentamicin_dosing;;
    arden: ASTM-E1460-1995;;
    version: 1.00;;
    institution: Columbia-Presbyterian Medical Center;;
    author: George Hripcsak, M.D.;;
    specialist: ;;

    date: 1991-03-18;;

    validation: testing;;

library:

    purpose:
           Suggest an appropriate gentamicin dose in the setting of renal
           insufficiency. (This MLM demonstrates a management suggestion.);;

    explanation:
           Patients with renal insufficiency require the same loading dose of
           gentamicin as those with normal renal function, but they require a
           reduced daily dose. The creatinine clearance is calculated by serum
           creatinine, age, and weight. If it is less than 30 ml/min, then an
           appropriate dose is calculated based on the clearance. If the
           ordered dose differs from the calculated dose by more than 20%,
           then an alert is generated.;;

    keywords: gentamicin; dosing;;

    citations: ;;

knowledge:

    type: data-driven;;

    data:
           /* an order for gentamicin evokes this MLM */
           gentamicin_order := event {medication_order where
                                      class = gentamicin}
           /* gentamicin doses */
           (loading_dose,periodic_dose,periodic_interval) :=
               read last {medication_order initial dose,
                          periodic dose, interval}
           /* serum creatinine mg/dl */
           serum_creatinine := read last ({serum_creatinine}
               where it occurred within the past 1 week)
           /* birthdate */
           birthdate := read last {birthdate}
           /* weight kg */
           weight := read last ({weight}
               where it occurred within the past 3 months
           ;;

    evoke:
           gentamicin_order;;

    logic:
           age := (now – birthdate)/1 year
           creatinine_clearance := 140 – age) * (weight)/
                                   (72 * serum_creatinine)
           /* the algorithm can be adjusted to handle higher clearances */
           if creatinine_clearance < 30 then
               calc_loading_dose := 1.7 * weight
               calc_daily_dose := 3 * (0.05 + creatinine_clearance / 100)
               ordered_daily_dose := periodic_dose *
                                      periodic_interval/(1 day)
               /* check whether order is appropriate */
               if abs (loading_dose-calc_loading_dose/calc_loading_dose > 0.2
                 or
                   abs(ordered_daily_dose – calc_daily_dose)/
                         calc_daily_dose > 0.2 then
                 conclude true
               endif
           endif
           ;;

    action:
           write  "Due to renal insufficiency, the dose of gentamicin " ||
                  "should be adjusted. The patient's calculated " ||
                  "creatinine clearance is " || creatinine_clearance ||
                  " ml/min. A single loading dose of " ||
                  calc_loading_dose || "mg should be given, followed by " ||
                  calc_daily_dose || "mg daily. Note that dialysis may " ||
                  "necessitate additional loading doses."

           ;;

    urgency: 50;;

end:
```

X1.5 Monitoring MLM:

```
maintenance:
    title: Monitor renal function while taking gentamicin;;
    mlmname: gentamicin_monitoring;;
    arden: Version 2;;
    version: 1.00;;
    institution: Columbia-Presbyterian Medical Center;;
    author: George Hripcsak, M.D.;;
    specialist: ;;
    date: 1991-03-19;;
    validation: testing;;
library:
    purpose:
           Monitor the patient's renal function when the patient is taking
           gentamicin. (This MLM demonstrates periodic monitoring.);;
    explanation:
           This MLM runs every five days after the patient is placed on
           gentamicin until the medication is stopped. If the serum creatinine
           has not been checked recently, then an alert is generated
           requesting follow-up. If the serum creatinine has been checked, is
           greater than 2.0, and has risen by more than 20 %, then an alert is
           generated warning that the patient may be developing renal
           insufficiency due to gentamicin.
    keywords: gentamicin; renal function;;
    citations: ;;
knowledge:
    type: data-driven;;
    data:
           /* an order for gentamicin evokes this MLM */
           gentamicin_order := event {medication_order where
                                       class = gentamicin};
           /* check whether gentamicin has been discontinued */
           gentamicin_discontinued :=
               read exist ({medication_cancellation where class = gentamicin}
                   where it occurs after eventtime);
           /* baseline serum creatinine mg/dl */
           baseline_creatinine := read last ({serum_creatinine}
               where it occurred before eventtime);
           /* followup serum creatinine mg/dl */
           recent_creatinine := read last ({serum_creatinine}
               where it occurred within the past 3 days;
           ;;
    evoke:
           every 5 days for 10 years starting 5 days after time of
               gentamicin_order until gentamicin_discontinued;;
    logic:
           if recent_creatinine is not present then
               no_recent_creatinine := true;
               conclude true;
           else
               no_recent_creatinine := false;
               if % increase of (serum_creatinine,
                   recent_creatinine) > 20 /* % * /
                   and recent_creatinine > 2.0 then
                   conclude true;
               endif;
           endiif;
           ;;
    action:
            if no_recent_creatinine then
                write  "Suggest obtaining a serum creatinine to follow up " ||
                        "on renal function in the setting of gentamicin.";
            else
                write  "Recent serum creatinine ("|| recent_creatinine ||
                        "mg/dl) has increased, possibly due to renal " ||
                        "insufficiency related to gentamicin use.";
            endif;
            ;;

    urgency: 50;;

end:
```

X1.6 Management Suggestion MLM:

```
maintenance:

    title: Granulocytopenia and Trimethoprim/Sulfamethoxazole;;

    mlmname: anctms;;

    arden: Version 2;;

    version: 2.00;;

    institution: Columbia-Presbyterian Medical Center;;

    author: George Hripcsak, M.D.;;

    specialist: ;;

    date: 1991-05-28;;

    validation: testing;;

library:

    purpose:
            Detect granulocytopenia possibly due to
            trimethoprim/sulfamethoxazole;;

    explanation:
            This MLM detects patients that are currently taking
            trimethoprim/sulfamethoxazole whose absolute neutrophile count is
            less than 1000 and falling;

    keywords:
            granulocytopenia; agranulocytosis; trimethoprim; sulfamethoxazole;;

    citations:
            1. Anti-infective drug use in relation to the risk of
               agranulocytosis and aplastic anemia. A report from the
               International Agranulosis and Aplastic Anemia Study.
               Archives of Internal Medicine, May 1989, 149(5):1036-40;;

    links:
            'CTIM .34.56.78';
            'MeSH agranulocytosis/ci and sulfamethoxazole/ae';;

knowledge:

    type: data-driven;;

    data:
            /*capitalized text within curly brackets would be replaced with */
            /*an institution's own query*/
            let anc_storage be event {STORAGE OF ABSOLUTE_NEUTROPHILE_COUNT};
            let anc be read last 2 from ({ABSOLUTE_NEUTROPHILE_COUNT}
                where they occurred within the past 1 week);
            let pt_is_taking_tms be read exist
                {TRIMETHOPRIM_SULFAMETHOXAZOLE_ORDER};
            ;;

evoke: anc_storage;;
```

```
logic:
          if pt_is_taking_tms
          and the last anc is less than 1000
          and the last anc is less than the first anc
           /* is anc falling? */
          then
              conclude true;
          else
              conclude false;
          endif;;

action:
          write   "Caution: patient's relative granulocytopenia may be " ||
                  "exacerbated by trimethoprim/sulfamethoxazole.";

end:
```

X1.7 MLM Translated from CARE:

```
maintenance:

    title: Cardiology MLM from CARE, p. 85;;

    mlmname: care_cardiology_mlm;;

    arden: Version 2;;

    version: 1.00;;

    author: Clement J. McDonald, M.D.; George Hripcsak, M.D.;;

    specialist: ;;

    date: 1991-05-28;;

    validation: testing;;

library:

    purpose:
          Recommend higher beta-blocker dosage if it is currently low and the
          patient is having excessive angina or premature ventricular
          beats;;

    explanation:
          If the patient is not bradycardic and is taking less than 360 mg of
          propolanol or less than 200 mg of metroprolol, then if the patient
          is having more than 4 episodes of angina per month or more than 5
          premature ventricular beats per minute, recommend a higher dose.;;

    keywords:
          beta-blocker; angina; premature ventricular beats; bradycardia;;

    citations:
          1. McDonald CJ. Action-oriented decisions in ambulatory medicine.
             Chicago: Year Book Medical Publishers, 1981, p. 85.
          2. Prichard NC, Gillim PM. Assessment of propolanol in angina
             electrocardiogram at rest and on exercise.  Br Heart J,
             33:473-480 (1971)
          3. Jackson G, Atkinson L, Oram S. Reassessment of failed beta-
             blocker treatment in angina pectoris by peak exercise heart rate
             measurements. Br Med J, 3:616-619 (1975).
          ;;

knowledge:

    type: data-driven;;
```

Health Level Seven © 1999. All rights reserved.
Final Standard.

```
    data:
           let last_clinic_visit be read last {CLINIC_VISIT};
           let (beta_meds,beta_doses,beta_statuses) be read
               {MEDICATION,DOSE,STATUS
               where the beta_statuses are 'current'
               and beta_meds are a kind of 'beta_blocker'};
           let low_dose_beta_use be false;
           /* if patient is on one beta blocker, check if it is low dose */
           if the count of beta_meds = 1 then
               if (the last beta_meds = 'propanolol'
                   and
                    last beta_doses < 360)
               or (the last_beta_meds = 'metroprolol'
                   and
                    the last beta_doses <= 200 then
                   let low_dose_beta be true;
               endif;
           endif;
           let cutoff_time be the maximum of
           ((1 month ago),(time of last_clinic_visit),
            (time of last_beta_meds));
           /* a system-specific query to angina frequency, PVC frequency, */
           /* and pulse rate would replace capitalized terms */
           let angina_frequency be read last ({ANGINA_FREQUENCY}
               where it occurred after cutoff_time);
           let premature_beat_frequency be read last
               ({PREMATURE_BEAT_FREQUENCY}
               where it occurred after cutoff_time);
           let last_pulse_rate be read last {PULSE_RATE};
           ;;

evoke: /* this MLM is called directly */;;

logic:
           if last_pulse_rate is greater than 60 and
               low_dose_beta_use then
               if angina_frequency is greater than 4 then
                   let message be
                       "Increased dose of beta blockers may be " ||
                       "needed to control angina.";
               else
                   if premature_beat_frequency is greater than 5 then
                       let message be
                           "Increased dose of beta blockers may be " ||
                           "needed to control PVCs.";
                       conclude true;
                   endif;
               endif;
           endif;
           conclude false;
           ;;

action:
           write message;;

end:
```

X1.8-MLM Using While Loop:

```
maintenance:

    title:  Allergy_test_with_while_loop;;

    filename:  test_for_allergies_while_loop;;

    version:  0.00;;

    institution:  ;;

    author:  ;;

    specialist:  ;;

    date:  1997-11-06;;

    validation:  testing;;

library:

    purpose:
           Illustrates the use of a WHILE-LOOP that processes an entire list
           ;;
    explanation:
           ;;

    keywords:
       ;;
knowledge:

    type: data-driven;;

    data:
           /* Receives four arguments from the calling MLM: */
           (med_orders,
            med_allergens,
            patient_allergies,
            patient_reactions) := ARGUMENT;
           ;;

evoke:
           ;;

logic:
           /* Initializes variables */
           a_list:= ();
           m_list:= ();
           r_list:= ();
           num:= 1;
           /* Checks each allergen in the medications to determine */
           /* if the patient is allergic to it */
           while num <= (count med_allergen) do
               allergen:= last(first num from med_allergens);
               allergy_found:= (patient_allergies = allergen);
               reaction:= patient_reactions where allergy_found;
               medication:= med_orders where (med_allergens = allergen);

               /* Adds the allergen, medication, and reaction to */
               /* variables that will be returned to the calling MLM */
               If any allergy_found then
                   a_list:= a_list, allergen;
                   m_list:= m_list, medication;
                   r_list:= r_list, reaction;
               endif;
           /* Increments the counter that is used to stop the while-loop */
               num:= num + 1 ;
           enddo;
           /* Concludes true if the patient is allergic to one of */
           /* the medications */
           If exist m_list
               then conclude true;
           endif;
           ;;

action:
           /* Returns three lists to the calling MLM */
           return  m_list, a_list, r_list;
           ;;

end:
```

X2 SUMMARY OF CHANGES

Summary of changes from the 1992 standard

- Clarification of many details of operator definitions.
- **Arden syntax version** slot required (6.1.3).
- Citations must be numbered, and can be classified as supporting or refuting (6.2.4).
- Specification of Links slot (6.2.5).
- Times can be constructed from durations via + operator (7.1.5.3).
- **Triggertime** is the time the MLM was triggered (8.4.5).
- Query retrieval order is not necessarily by primary time (8.9.2).
- **Interface** statement for using external functions (11.2.12).
- Single-line comments may be introduced with "//" (7.1.9).
- The **filename** slot has been renamed to **mlmname** (6.1.2).
- Some new operators have been introduced:
  - **sort data** (9.2.4)
  - **sort time** (9.2.4)
  - **reverse** (9.12.21)
  - **format** (9.8.2)
  - **earliest**, **latest** (9.12.17, 9.12.16)
  - **floor**, **ceiling**, **truncate**, **round** (9.16.11, 9.16.12, 9.16.13, 9.16.14)
  - **index** (…[…]) (9.12.18)
  - **year**, **month**, **day**, **hour**, **minute**, **second** field extraction (9.11.2, 9.11.4, 9.11.7, 9.11.9, 9.11.11, 9.11.13)
  - **seqto** (9.12.20)
  - **string**, **extract characters** (9.8.3, 9.12.19)
- Operators which select from lists may be annotated to return indexes instead of the elements (9.12.18).
- **As number** operator which converts strings and Booleans to numbers (9.16.17).
- Some restrictions have been removed (e.g., double semi-colon inside strings).
- The **call** expression and statement can now pass multiple arguments; arguments may also be passed from an action slot (10.2.4, 11.2.4, 12.2.2, 12.2.4).
- Looping constructs have been added: for loop, while loop (10.2.5, 10.2.6).
- The **continue** statement may have an **unless** added to it (this is a readability aid).
- A new form of conditional execution, by allowing **unless** in a conclude statement.
- The "read… where… " no longer requires parentheses.
- A read query may specify a sort order (different from the default of chronological by primary time).

REFERENCES

(1) HELP Frame Manual, 1991, LDS Hospital, 325 8th Ave., Salt Lake City, UT 84143.

(2) McDonald, C.J., Action-Oriented Decisions in Ambulatory Medicine, Chicago: Year Book Medical Publishers, 1981.

(3) Wirth, N., "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" Communications of the ACM, Vol. 20, 1977, pp. 822-823.

(4) UMLS Knowledge Sources, Experimental Edition, Bethesda, MD: National Library of Medicine, September 1990.

(5) International Committee of Medical Journal Editors, Special Report, "Uniform Requirements for Manuscripts Submitted to Biomedical Journals," The New England Journal of Medicine, Vol 324, No. 6, 1991, pp. 424-428.