HL7 Arden V2.9-2012
May, 2012
Health Level Seven
Arden Syntax
Version 2.9

The Arden Syntax for Medical Logic Systems Version 2.9

Revision date: 2012-03-22
Print date: 7/16/2012

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. Subsequent versions, Version 2, Version 2.1, Version 2.5, Version 2.6, and Version 2.7 were developed and published by Health Level Seven, Inc. (HL7). These versions were accepted as standards by the American National Standards Institute (ANSI) and The International Standards Organization (ISO). The previous standard, Version 2.7, was accepted as an ANSI standard in 2008. This version, 2.8, represents an extension of the previous ANSI version.

Arden Syntax for Medical Logic Systems

TABLE OF CONTENTS

Revision date: 2012-03-22 Print date: 7/16/2012

WHAT'S NEW IN VERSION 2.9

The principal change of version 2.9 is the introduction of support for fuzzy logic. Fuzzy logic is a multi-valued logic that has gained use in formal decision-making because of its value in representing reasoning involving imprecision. Unlike the typical binary (true-false) logic that continues to be supported in this version of the Arden Syntax, fuzzy logic incorporates degrees of truth or set membership (Steimann F. On the use and usefulness of fuzzy sets in medical AI. Artif Intell Med 2001;21:131-7). Clinical practice guidelines and other forms of clinical knowledge representation may employ fuzzy logic, using linguistic variables such as "severe" and "somewhat" without necessarily formally defining them or providing an objective quantification. These changes in the Arden Syntax allow fuzzy logic to be formally represented, thus supporting the representation of clinical guidelines and clinical reasoning generally. Other explanatory material regarding fuzzy logic may be found at the HL7 Arden Syntax work group wiki at http://wiki.hl7.org/index.php?title=Arden_Syntax_Work_Group.

Changes necessitated to support fuzzy logic include the creation of new operators (Fuzzy Set, Fuzzified By, Defuzzified, Applicability Of) as well as expansion of the definition of logical operators to accommodate fuzzy reasoning.

The other significant change in the Arden Syntax is that the Resources category now is mandatory.

Changes to version 2.8 now reflected in version 2.9:

- 6.4, changed **resources** category definition from optional to required, stating that in former versions this category is optional and a default value is used
- 8.13, new data type **Truth Value** which is a generalization of Boolean
- 8.14, new **fuzzy data type** section which contains a set of data types to express fuzzy sets
- 8.15, added **applicability**, similar to "primary time" a new subcomponent is added which allows to express the applicability of a value
- 9.1.2.2, changed some **type categories** and added some new type categories to allow to use them in the operator signatures
- 9.1.3, added the new operators to **list handling** explanation
- 9.1.6, added **general applicability handling** (similar to primary time handling)
- 9.2.4, **sort** operator adjusted to be able to sort a list by the **applicability** of the values
- 9.4.1, adjusted the **or** operator to handle truth values
- 9.4.2, adjusted the **and** operator to handle truth values
- 9.4.3, adjusted the **not** operator to handle truth values
- 9.5.4, adjusted the <= operator to handle a crisp and a fuzzy data type
- 9.5.5, adjusted the >= operator to handle a crisp and a fuzzy data type
- 9.6.14, the **is [not] in** operator is now able to handle a crisp and a fuzzy data type to find the mapping of the crisp value to the given fuzzy set
- 9.6.27, new operator **is[not] fuzzy** to check a value if it is fuzzy or not
- 9.6.28, new operator **is[not] crisp** to check a value if it is crisp or not
- 9.13.5, adjusted the **at least** operator to handle truth values in a list
- 9.13.6, adjusted the **at most** operator to handle truth values in a list
- 9.19, new section **fuzzy operators** to store all operators on fuzzy sets
- 9.19.1, added new operator **fuzzy set ...** which is able to create fuzzy sets
- 9.19.2, added new operator **... fuzzified by ...** which is able to create simple triangular fuzzy sets
- 9.19.3, added new operator **defuzzified ...** which defuzzifies a fuzzy set
- 9.19.4, added new operator **applicability [of] ...** to access a values applicability (and to set it)
- 9.20, new section **type conversion operators** which contains all type conversion operators
- 9.20.1, the **as number** operator moved from 9.16.17

- 9.20.2, the **as time** operator moved from 9.17.4
- 9.20.3, the **as string** operator moved from 9.8.13
- 9.20.4, added new operator **as truth value** which converts a number into a truth value
- 10.2.2, adjusting the **if-then-statements** to describe what happens if the condition expression evaluates to a truth value
- 10.2.3, adjusting the **switch-statements** to describe what happens if the condition expression evaluates to a truth value
- 10.2.8, added reference to the **linguistic variable definition** which is a special object type
- 11.2.18, added the **linguistic variable statement** which describes an object with only fuzzy sets as fields
- A1, changes to the **BNF** to reflect the new operators and changes to the existing statements
- A2, added reserved words: **aggregate, applicability, crisp, defuzzified, endswitch, fuzzified, fuzzy, linguistic, set, truth, value, variable**
- A4, added the new operators

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 Health Level Seven Standards[1]:

HL7 Version 2.3

HL7 Version 3

2.2 ASTM Standards[2]:

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

2.3 ANSI Standards[3]:

ANSI X3.4 - 1986	Coded Character Sets-American National Standard Code for Information Interchange (7-bit ASCII)
ANSI/ISO 9899 - 1999	Programming Language C
ANSI/ISO/IEC 9075 - 2003	Information technology – Database languages – SQL
ANSI/NISO Z39.88 - 2004	The OpenURL Framework for Context-Sensitive Services

2.4 ISO Standards[4]:

ISO 8601 – 2004	Data Elements and Interchange Formats-Information Interchange (representation of dates and times)
ISO 88599 – 1998	Latin-1 Coded Character Set
ISO / IEC 9075 – 2003	Information technology – Database languages – SQL
ISO 8879 – 1986	Information processing – Text and office systems – Standard Generalized Markup Language (SGML)
ISO 639-1 - 2002	Codes for the representation of names of languages -- Part 1: Alpha-2 code
ISO 3166-1 - 1997	Codes for the representation of names of countries and their subdivisions.
ISO/IEC 10646:2003	Information technology -- Universal Multiple-Octet Coded Character Set (UCS)

---

1 Available from Health Level Seven, Inc.
3300 Washtenaw Ave, Suite 227, Ann Arbor, MI 48104, USA. www.hl7.org

2 Annual Book of ASTM Standards, Vol 14.01. Available from ASTM International ,
100 Barr Harbor Drive, West Conshohocken, PA19428-2959, USA. www.astm.org

3 Available from American National Standards Institute,
1430 Broadway, New York, NY 10018, USA. www.ansi.org

4 Available from ISO,
1 Rue de Varembe, Case Postale 56, CH 1211, Geneve, Switzerland. www.iso.ch

2.5 World Wide Web Consortium Recommendations[5]:

Extensible Markup Language (XML) 1.0 (Third Edition) 2004-02-04

Extensible Markup Language (XML) 1.1 2004-02-04

2.6 Unicode Standards[6]:

Unicode 5.0

[5] Available from World Wide Web Consortium (W3C).
MIT, 32 Vassar Street, Room 32-G515, Cambridge, MA 02139 USA or
ERCIM, 2004, route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex France. www.w3c.org.

[6] Available from The Unicode Consortium.
P.O. Box 391476, Mountain View, CA 94039-1476, U.S.A. www.unicode.org.

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 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 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)**[7], and CARE, the language of the Regenstrief Medical Record System of the Regenstrief Institute for Health Care, Indianapolis, IN **(2)**.

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

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 UNICODE encoding, 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 the library category (Section 6.2), 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.

5.4 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 X4 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;;
...
Resources:  <optional>
Slotname:  slot-body;;
.. .
end:
```

5.6 Categories

An MLM is composed of slots grouped into three required categories, maintenance, library, and knowledge, and one optional category, resources. A category is indicated by a category name followed immediately by a colon (that is, **maintenance:**, **library:**, **knowledge:**, and **resources:**).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, period (.), minus (-), 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;;
       or
mlmname: hiv_screening.mlm;;
```

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

6.1.3 Arden Syntax version (coded, optional*)

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:

```
arden: Version <Version number of Arden Syntax standard>;;
```

The text is not case sensitive. For example,

```
arden: Version 2;;
arden: version 2.1;;
arden: version 2.5;;
arden: version 2.6;;
```

* This slot is required for versions 2 and later 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 (Software Change/Configuration Control System) or RCS (Revision Control System). 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 11.2.4). 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 list, 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 complete representation in 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 Syntax must 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)

The citations slots allows for the documentation of citations to relevant literature to be documented within an MLM. 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. When using the textual format, citations to the literature should be entered in Vancouver style **(5)**.

In the structured format, 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.
;;
```

Within the structured citations format, either Vancouver style **(5)** or OpenURL format (ANSI/NISO Z39.88) are acceptable forms for representing individual citations. It is anticipated that the OpenURL format will become the preferred form in future versions of this standard. Appendix X2 contains examples of citations formatted using the OpenURL format as part of the discussion of an XML schema for representing MLMs.

6.2.5 Links (structured / textual, optional)

The links slot allows an institution to define links to other sources of information, such as an electronic textbook, teaching cases, or educational modules. 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 structured format may either use the ad-hoc format first presented in Arden Syntax Version 2.0 or the OpenURL format (ANSI/NISO Z39.88) to represent individual links. The individual links are delimited by semicolons. The contents of the links are institution-specific

Within the ad-hoc format, 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 term), comma, link text (Arden Syntax string). 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.

Appendix X2 contains examples of links formatted using the OpenURL format as part of the discussion of an XML schema for representing MLMs.

Note: This definition of the structured link differ from the 2.5 and previous versions of the structured link. This change was made to bring the structured link into conformance with the definitions of resource statements as defined in Section 6.4. Future version of the Arden Syntax standard will provide mechanisms for calling external links, it was decided to break backward compatibility on this issue to make the related constructs of links and resources have parallel structure. As the structured link has not been widely implemented it was felt that this was the proper time to make this change.

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;;
priority: 40.5;;
```

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 9.19.

6.3.6 Action (structured, required)

This slot contains the action produced when the logic slot concludes **true**. The details of this slot are explained in Section 12.

6.3.7 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. It is recommended that only integers be used as values in the urgency slot. 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;;
```

6.4 Resources category*

The resources category contains a set of language slots that specify the textual resources on which the localized operator may be applied to obtain message contents in different languages (Section 9.8.11). Each language slot defines a set of key/value pairs that represent text constants in one specific language. At least one language slot is required if the resources category is defined. Its structure is:

```
resources:
  default: <language code>;;
  language: <language code>
    <set of language specific resources> ;;
  language : <language code>
    <set of language specific resources> ;;
```

The language codes are defined either as 2-character ISO 639-1 language codes or as combination of a 2-character ISO 639-1 language code and a 2-character ISO 3166-1 geographical code concatenated by an underscore. That is,

```
en
```

or

```
en_US
```

or

```
en_GB
```

or

```
fr
```

The ISO 639-1 code is mandatory while the extended combination of language and region is optional. Implementing systems that support localization using this extended language code (that is, a locale) can further define resources for the individual use of one specific language in different regions in the world.

* This category is required for versions 2.9 and later of the syntax, but is optional in older versions of the Arden Syntax. For ensuring backward compatibility, in older versions of the Arden Syntax,

```
resources:
    default: en;;
    language: en;;
```

has to be used in the case the value is missing.

6.4.1 Default (coded, required)

When using the localized operator, the implementing system has to retrieve the current user language setting. The default slot specifies what language setting has to be applied on the MLM when this user language cannot be retrieved by the implementing system. The value of the default slot is a language code as defined in Section 6.4. That is,

```
default: de;;
```

or

```
default: en_US;;
```

6.4.2 Language (coded, required)

The resources category also consists of one or more language slots. Each language slot contains of a language code as defined in Section 6.4 followed by a set of key/value pairs. Each key is a term (see Section 7.1.7) and its associated value is a string constant (Section 7.1.8). Each key is separated from its value by a colon (:). Each string defines the result of the localized operator when applied to the corresponding term. That is,

```
language: en
  'msg': "Caution, the patient has the following allergy to penicillin
    documented: ";
  'creat': "The patient's calculated creatinine clearance is
    %f ml/min."
;;
language: de
  'msg': "Vorsicht, zu diesem Patienten wurde die folgende Penicillinallergie
    dokumentiert: ";
  'creat': "Die berechnete Kreatinin-Clearance des Patienten beträgt %f
    ml/min."
;;
```

Each language slot must contain a unique language code (ISO 639.1) or optionally, a language code concatenated with an underscore "_" followed by a region code (ISO 3166-1). If these region codes are used, every entry associated with the language must contain a region code. For example,

```
language: en_US [..] ;;
language: en_UK [..] ;;
```

is valid while

```
language: en [..] ;;
language: en_US [..] ;;
```

is not.

The resources category may contain multiple language slots with a variety of <language code>_<region code> definitions. If the implementing system is only able to determine the required language at runtime, but not the required region, the first language slot matching that language is chosen. In the following example, if only the language code 'de' was known, the German definition (de_DE) would be used:

```
language: de_DE [..] ;;
language: de_AT [..] ;;
```

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 EST (Eastern Standard Time, United States of 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.

7.1.11 Time-of-day Constants

Time-of-day constants use the ISO format (for example, **18:30, 13:23:00.123**) without the date field. Constants are defined analogously to time constants as defined in 7.1.5. Time-of-day constants must contain at least the two-digit hour and minute components – in other words, they must consist of two integers ranging from 00 to 23, one colon, and two more integers ranging from 00 to 59. Seconds, fractional seconds and time zones are optional in time-of-day constants. Midnight is expressed as **00:00:00.000** and all other time-of-day values are greater than this value.

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 9.19, 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 10.2.5). 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 11.2.4); 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). Entities of data type null may also have a primary time. 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 beyond milliseconds is left to the implementing instance. Times stored in patient databases will have varying granularities. When a time is read by the MLM, it is always truncated to the beginning of the granule interval. For example, if the time-of-day is recorded only to the minute, then zero seconds are assumed; if only the date is known, then the time-of-day is assumed to be midnight.

8.4.2 Midnight

Midnight represents the beginning of a day and is expressed as T00:00:00 in a time data type, or as 00:00 as a time-of-day. 24:00 is **not** defined.

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 will therefore 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 12.2.5), 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**.

8.4.6 Currenttime

The word **currenttime** represents the system time at the instant the word is encountered during MLM execution. **Currenttime** differs from **now** in that **currenttime** constantly changes, while **now** remains constant while an MLM runs. Thus, the time required to execute an MLM (or query) can be determined by subtracting **now** from **currenttime**. The following inequality is guaranteed within a single MLM: **eventtime <= triggertime <= now <= currenttime**.

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 9.10.7). 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 = 1990-11-26T22:57:05.4
1991-01-31T00:00:00 - 1.1 months = 1990-12-27T22:57:05.4
1991-04-30T00:00:00 - 0.1 months = 1991-04-26T22:57:05.4
```

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. They are used only in a **call** statement (see Section 10.2.5). 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, string, truth value, fuzzy number, fuzzy time, or fuzzy duration. 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
()
```

If operators that expect list arguments are presented non-list arguments, the arguments are implicitly converted to single-element lists before the operator is applied.

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) associated with it. This time is defined as the medically relevant time for that query. For different entities, the primary time might signify different times. The primary time of a blood test might be the time it was drawn from the patient (or the closest to that time), whereas the primary time of a medication order might be the time the order was placed. 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.

8.10 Object

An object results from use of the New statement (see Section 10.2.8), the **read as** statement (Section 11.2.2), the **destination as** statement (Section 11.2.9), or the **message as** statement (Section 11.2.7). It may contain multiple named attributes, each of which may contain any valid Arden type (including lists or objects). The latter capability allows for complex data structures to be manipulated by an MLM (lists within lists, for example) which would otherwise not be possible. Objects are also useful for interfacing MLMs with other object-oriented domain models (outside the scope of this document).

8.11 Time-of-day

The time-of-day data type refers to points in time that are not directly linked to a specific date. Time-of-day constants are analogously defined to time constants leaving the date portion blank. Time-of-day constants (for example, **23:20:00**) are defined in Section 7.1.6.

Operators that can use both time arguments and time-of-day arguments at the same time may follow the default time-of-day handling as defined in Section 9.1.5 .The primary time handling is unaffected by these extension.

Note: To improve readability when describing this data type, the phrase “time-of-day” is usually hyphenated. These hyphens are NOT included when TIME OF DAY is used in an MLM.

8.12 Day-of-week

The day-of-week data type is a special data type to represent specific days of the week to be used along with the "day of week" operator. Values of this data type are either expressed by constants or by integer values.

Day-of-week constants are defined by the following keywords:

MONDAY (1),
TUESDAY (2),
WEDNESDAY (3),
THURSDAY (4),
FRIDAY (5),
SATURDAY (6)
SUNDAY (7),

Note: To improve readability when describing this data type, the phrase “day-of-week” is usually hyphenated. These hyphens are NOT included when DAY OF WEEK is used in an MLM.

8.13 Truth Value

The data type of propositional variables is denoted by truth value or−for reasons of backwards compatibility−, equivalently, **Boolean**. A variable of this type stores real numbers between 0 and 1. The Boolean value **true** is equal to the truth value 1 and the Boolean value **false** is equal to the truth value 0. One may write:

```
Var := truth value 0; or, equivalently Var := false;
Var := truth value 0.667;
Var := truth value 1; or, equivalently Var := true;
```

8.14 Fuzzy Data Types

Fuzzy data types are fuzzy sets over one of the data types number, time or duration.

Fuzzy sets − as opposed to classical, crisp sets − provide a formal methodology to define and process sets or classes with unsharp boundaries.

A linguistic term in clinical descriptions/texts such as small or large, cold or warm, normal and elevated, enlarged and symmetric, diabetic and hypoxic is inherently a set, or class, with unsharp boundaries. A crisp boundary between neighboring concepts such as normal or pathological is − in the context of evaluating a particular patient − to a certain extent arbitrary and thus often not acceptable in real clinical situations.

For example, fever is defined as body temperature ≥ 38°C. What about 37.9°C or 37.8°C? Is it fever with a certain degree, e.g. 0.8 or 0.6? In clinical medicine, the answer should be yes. In computer assisted clinical medicine, a diagnostic definition including fever ought to be enabled when measured 37.9°C at least to a certain degree, even if the crisp fever definition starts at 38°C.

Fuzzy sets make it possible to formally define linguistic terms with unsharp boundaries, to calculate degrees of truth if measurements fall into the borderline range, and fuzzy logic allows propagating the results through further calculation. The example MLMs X4.9, X4.10, and X4.11 illustrate the difference between crisp, simulated fuzzy, and fuzzy representation of the same calculation.

8.14.1 Fuzzy Number

The data type **fuzzy number** is dedicated to fuzzy sets over the reals. A fuzzy number partitions the reals into a finite number of (possibly unbounded) intervals, on each of which the fuzzy set is linear and continuous.

Formally, a fuzzy set **u : R→ [0, 1]** can be stored into a variable of the type fuzzy number, if the following condition is met: There are $\mathbf{a_1} < \mathbf{a_2} < \ldots < \mathbf{a_k}$ with **k >= 1**, such that **u** is linear on each open interval $\mathbf{(a_1; a_2),}$

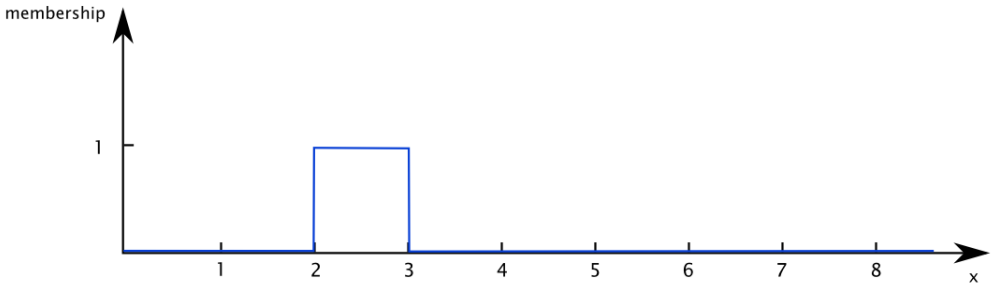
… , (a_{k-1}; a_k), **u** is constant on **(-∞; a_1)** and **(a_k; +∞)**, and for each **x** in **R**, **u(x)** coincides either with the left limit or the right limit of **u** at **x**. If **u** is continuous, we then define:

```
Fuzzyset u := fuzzy set (a1,t1), (a2,t2), ..., (ak,tk);
```

where **t_i = u(a_i) for i = 1, …, k and u(x) is called the characteristic function of the fuzzy set.**.

The characteristic functions are allowed to contain discontinuities, which are not likely to be required in applications, but should at least be definable. At discontinuity points we denote the left as well as the right limit. The first assignment is the value at that point, unless the second one appears twice. For instance,

```
TwotoThree := fuzzy set (2, 0), (2, 1), (2, 1), (3, 1), (3,0);
```



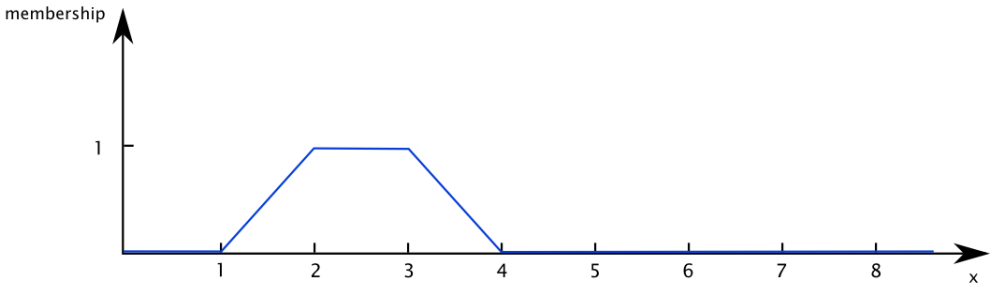
At point 2 there is a "discontinuity point" which means that approaching 2 from the left side the membership value is 0 while approaching 2 from the right side the membership value is 1. The question is which membership value is assigned to 2. The sentence "The first assignment is the value at that point, unless the second one appears twice." means in this example (fuzzy set `TwotoThree`) that the membership value at point 2 is 1, since the (2, 1) appears twice. If the fuzzy set is adjusted to

```
TwotoThree := fuzzy set (2, 0), (2, 1), (3, 1), (3,0);
```

the membership value at point 2 is 0.

The example

```
OnetoFour := fuzzy set (1, 0), (2, 1), (2, 1), (3, 1), (4,0);
```



has no such "discontinuity point" at 2 and writing the (2, 1) twice is unnecessary but should have no effect to the interpretation of the function.

Fuzzy sets describing a symmetrical triangle around a single point, which is mapped to 1, are called triangular normal fuzzy sets. A simplified notation is permitted for these: an expression of the form fuzzy set (a - b, 0), (a, 1), (a + b, 0), where a; b in R and b > 0, may also be written as:

```
a fuzzified by b
```

8.14.2 Fuzzy Time

The data type **fuzzy time** refers to fuzzy sets over times. Except for the simplified notation, all definitions of fuzzy numbers apply mutatis mutandis to fuzzy time.

For the simplified notation, a time constant can only be fuzzified by duration. Thus, we define

```
AfuzzyTime := today fuzzified by 1 day;
simple := 2009-10-10 fuzzified by 12 hours;
complex := fuzzy set (2009-10-10,0), (2009-10-11,1), (2009-11-10,1), (2009-
    11-11,0);
```

8.14.3 Fuzzy Duration

All definitions of a fuzzy number apply mutatis mutandis to **fuzzy duration**.

```
simple := 14 days fuzzified by 1 day;
complex := fuzzy set (2 days,0), (3 days,1), (14 days,1), (31 days,0);
```

8.15 Applicability

All simple data types (Truth Value, Boolean, Number, Time, Duration, String, Term, Query Results, Time-of-Day, Day-of-Week, Fuzzy Types) are endowed with an additional type of information called the **degree of applicability**. The degree of applicability stores a truth value that refers to the degree to which it is reasonable to use the value of a variable. It is 1 by default, and—whenever the program branches—reduced automatically according to the weight assigned to that particular branch. The programmer may decide to make explicit use of this concept but is not required to do so. To access the degree of applicability of an expression, the Arden Syntax programmer is referred to the **applicability [of]** operator (Chapter 8.15).

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**.

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

time-of-day—time-of-day data type

times—time and time-of-day data type

duration—duration data type

string—string data type

truth-value—truth value data type

item—not used in expressions, only in **call** statements (see 10.2.4)

any-type—null, Boolean, number, time, time-of-day, duration, string, truth-value, fuzzy-number, fuzzy-time, or fuzzy-duration

fuzzy-type—fuzzy-number, fuzzy-time, or fuzzy-duration

crisp-type—Boolean,number,time,time-of-day,duration,or string

non-null—Boolean, number, time, time-of-day, duration, string, truth-value, fuzzy-number, fuzzy-time, or fuzzy-duration

ordered—number, time, time-of-day, duration, string, or truth-value

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. Each operator must apply the here described list handling first (if applicable) before the specific list handling as described in the respective operator description is applied.

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 time of day
… is not time of day
… is duration
… is not duration
… is string
… is not string
… is fuzzy
… is not fuzzy
… is crisp
… is not crisp
… is object
… is not object
… is <object-type>
… is not <object-type>
+ …
- …
… ago
… year
… years
```

```
… month
… months
… week
… weeks
… day
… days
… hour
… hours
… minute
… minutes
… second
… seconds
… as number
… as string
… as time
… as truth value
time [of] …
time of day [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] …
extract year [of] …
extract month [of] …
extract day [of] …
extract hour [of] …
extract minute [of] …
extract second [of] …
int …
round …
sqrt …
string …
length [of] …
uppercase …
lowercase …
trim …
localized …
defuzzified …
applicability [of] …
```

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] …
sort [data|time] …
reverse …
```

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 numbers 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 …
… 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 …
… from …
localized … by …
replace year [of] … with …
replace month [of] … with …
replace day [of] … with …
replace hour [of] … with …
replace minute [of] … with …
replace second [of] … with …
```

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 earliest … from …
index of … from …
add … to …
at least … from …
```

```
at most … 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 …
remove … 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 Time-of-Day Handling

Operators that are defined for operands of "any" type, ordered types, etc. are not affected by time-of-day values. For example, aggregation operators such as the average operator still compute a result from a homogeneous list of time-of-day values, but return null if time-of-day values and time values are combined. Those operators that can be used with combined time-of-day and time values are defined in the next sections.

9.1.5.1 Default Time-of-Day Handling

Some binary and ternary operators can combine time and time-of-day values as operands as defined in the next section. In this case, as the time-of-day data type is a sub-type of the time data type, the operators automatically use the common information part of the operands, which is the time-of-day-fraction of the given time value, and ignore the date information of the other operand (see examples of simple comparison operators in Section 9.5).

Operators that follow the default time-of-day handling are

- simple comparison operators (Section 9.5)
- is after/before (Section 9.6.12, 9.6.13)

9.1.5.2 Role of midnight

Operators where the order of the arguments may indicate that the midnight boundary may be spanned are

is within ... to … (Section 9.6.6)

- is within ... to … preceding/following (Section 9.6.7, 9.6.8)
- is within … to … surrounding … (Section 9.6.9)
- Arithmetic operators (Section 9.9)

9.1.5.3 Undefined Operators for time-of-day values

Operators for which time-of-day data types are not allowed as arguments are

- 'is within same day as': **undefined** for time-of-day operands as the required information for the comparison (date) is not present; returns null
- 'within past': **undefined** for time-of-day operands as the reference of the comparison is usually a fixed date and time; returns null

9.1.6 Applicability Handling

In general, every binary and ternary operator calculates the applicability of its result as the minimum of the applicability values of all parameters if not stated otherwise in the following definitions. Every unary operator sets the applicability of its result to 1 if not otherwise stated below. These rules also apply to the component-wise application of the operator on elements of lists.

9.1.7 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.

9.1.7.1 Precedence Table

The operators are shown grouped by precedence in Annex A4.

9.1.8 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:

9.1.8.1 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**).

9.1.8.2 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**.

9.1.8.3 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).

9.1.9 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 and applicabilities are maintained according to Section 9.1.4, unless otherwise specified.

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

Binary **,** (list concatenation) appends two lists. Primary times and applicabilities of the individual list elements are maintained. 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 and applicabilities 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 (unary, non-associative)

The **sort** operator reorders a list based on element contents, which are either the element values (keyword **data**) the primary times (keyword **time**), or the applicability (keyword **applicability**). An optional modifier may be used with the sort operator. If used, the modifier must be placed immediately after the **sort** keyword. The following keywords can be placed after the **sort** keyword: **data**, **time**, or **applicability**, which are mutually exclusive. If no modifier is used, the sort operator defaults to a data sort. Direction of sorting is always ascending. For a descending sort, **reverse** can be used.

The sort options are considered to be part of the sort operator for precedence purposes. This resolves the potential conflict with the **time [of]** operator (9.17.1). Thus the expression "**sort time x**" should be parsed as "sort the list x by time" rather than as "extract the primary times from the list x and sort the list of times."

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**). The sorting by applicabilities is defined equivalent to sorting by primary times. 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** and applicability values **truth value 0.7, truth value 0.5, truth value 0.3**):

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

The optional modifier **using ...** can be appended to the **sort** operator to control the calculation of the ordering. Thus, the following expressions can be used to sort the list by the data or the primary times of the elements:

```
<n:any-type> := sort <n:any-type> using it; // for sorting by data
<n:any-type> := sort <n:any-type> using time of it; // for sorting by time
```

The above mentioned expressions will be equivalent to the currently available expressions **sort time** and **sort data**. However, the **using** operator can be used to sort the list by an arbitrary calculation applied to each element of the list, e.g.:

```
<n:any-type> := sort <n:any-type> using sin it; // for sorting the list by
// the sin of each value
<n:any-type> := sort <n:any-type> using abs it; // for sorting the list by
// absolute values of the list elements
<n:any-type> := sort <n:any-type> using extract month it; // for sorting the
// list by month part of the list elements
```

If the **using** operator is applied to a list of objects, the list may be sorted by a specified field of the given objects, e.g.:

```
<n:object> := sort <n:object> using it.height; // for sorting the objects by
// their field "height"
<n:any-type> := sort <n:any-type> using time of it.value; // for sorting the
// objects by the primary time of their field "value"
```

The modifier using can contain any complex expression incorporating the **it** keyword.

9.2.5 Add … To … [At …] (ternary, non-associative)

The **add ... to ... [at ...]** operator expects an arbitrary data value as its first argument and a list as its second argument. It adds this element to the given list. If no position is given, the element will be added to the end of the list. If a position is provided, the element is inserted at this position and the index of all elements from this to the end of the list will be increased by one. If the given position is greater than the cardinality of the list, the element will be appended at the end of the list. In case a negative position or 0 is given, the element will be appended at the beginning of the list. If the second argument is not a list, the argument is assumed a list with one element. When more than one position is given, the positions are first identified and then the elements are inserted. The usage of the **add ... to ... [at ...]** operator is:

```
<n+1:any-type> := ADD <1:any-type> TO <n:any-type>
<n+m:any-type> := ADD <1:any-type> TO <n:any-type> AT <m:number>
(1, 2, 3, 4) := ADD 4 TO (1, 2, 3);

(4, 1, 2, 3) := ADD 4 TO (1, 2, 3) AT 1;

(1, 2, 3, null) := ADD null TO (1, 2, 3);

(null, 4) := ADD 4 TO null;

(1, 2, 3, 4) := ADD 4 TO (1, 2, 3) AT 9;

(4, 4, 1, 2, 3) := ADD 4 TO (1, 2, 3) AT (1, -1);

(1, 2, 3, 4) := ADD 2 TO (1, 3, 4) AT INDEX OF 3 WITHIN (1, 3, 4);

   (4, 1, 4, 2, 3) := ADD 4 TO (1, 2, 3) AT (1, 2);
```

9.2.6 Remove … From … (binary, non-associative)

The **remove ... from ...** operator expects a number or list of numbers as its first argument and a list as its second argument. The operator also accepts first and last as its first argument, they are interpreted as the number representing the last (the first) index in the given list. The operator removes the elements with the given indices from the list. The indices of all elements from the given index to the end of the list will be decreased by one. If the second argument is not a list, the argument is assumed a list with one element. When more than one position is given, the positions are first detected and then the elements are removed. The usage of the **remove ... from ...** operator is:

```
<n-m:any-type> := REMOVE <m:number> FROM <n:any-type>
(2, 1) := REMOVE 1 FROM (3, 2, 1);

("two", 4, 5) := REMOVE (1,3,6) FROM ("one", "two", 3, 4, 5, 6 days);

(3, 2, 1) := REMOVE null FROM (3, 2, 1);

(3, 2, 1) := REMOVE 8 FROM (3, 2, 1);

() := REMOVE (INDEX OF "3" WITHIN ("3", "3")) FROM ("3", "3");

(null) := REMOVE 2 FROM null;

() := REMOVE 1 FROM null;

   (3, 2, 1) := REMOVE () FROM (3, 2, 1);
```

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) and applicabilities 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 following 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.

The following example follows the default time-of-day handling as it combines primary times (time values) with time-of-day constraints to select those blood glucose values that have been measured after lunch:

```
post_prandial_blood_glucoses := bc_values where they occurred within 13:00:00
    to 15:00:00
```

The where operator can also be combined with day-of-week arguments, such as

```
labResults where day of week of time of them is in (SATURDAY, SUNDAY)
```

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**. If both arguments are truth values, the maximum of both arguments is returned. Otherwise the result is **null**. Its usage is as follows:

```
<n:truth-value> := <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
0.4 := false OR (0.4 AS TRUTH VALUE)//see section 9.20.4 (AS TRUTH VALUE)
true := true OR (TRUTH VALUE 0.7)//see section 8.13 (truth values)
0.5 := (0.5 AS TRUTH VALUE) OR (0.4 AS TRUTH VALUE)
(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	other truth value	Other	(Right argument)
(Left	TRUE	TRUE	TRUE	TRUE	
argument)	other truth value	TRUE	MAX(a, b)	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**. If both arguments are truth values, the minimum of both arguments is returned. Otherwise the result is **null**. Its usage is:

```
<n:truth-value> := <n:any-type> AND <n:any-type>
false := true AND false
null := true AND null
false := false AND (0.4 AS TRUTH VALUE)//see section 9.20.4 (AS TRUTH VALUE)
false := false AND (TRUTH VALUE 0.5)//see section 8.13 (truth values)
0.4 := (0.5 AS TRUTH VALUE) AND (0.4 AS TRUTH VALUE)
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	other truth value	other	(Right argument)
(Left argument)	TRUE	TRUE	FALSE	NULL	
	other truth value	FALSE	MIN(a, b)	FALSE	
	Other	NULL	FALSE	NULL	

9.4.3 Not (unary, non-associative)

The **not** operator performs the logical negation of its argument. If the argument is a truth value, the negation is the subtraction from 1. Its usage is:

```
<n:truth-value> := NOT <n:any-type>
true := NOT false
null := NOT null
0.8 := NOT (0.2 as TRUTH VALUE) //see section 9.20.4 (AS TRUTH VALUE)
0.8 := NOT (TRUTH VALUE 0.2) //see section 8.13 (truth values)
(true, false) := NOT (false, true)
() := NOT ()
```

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

NOT	TRUE	FALSE	Other truth value	other
	FALSE	TRUE	1- truth value	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:crisp-type> = <n:crisp-type>
false := 1 = 2
(null,true,false) := (1,2,"a") = (null,2,3)
null := (3/0) = (3/0)
() := 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)
true := 1979-02-25T08:20:00 = 08:20:00
```

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:crisp-type> <> <n:crisp-type>
true := 1 <> 2
(null,false,true) := (1,2,"a") <> (null,2,3)
null := (3/0) <> (3/0)
false := 1979-02-25T08:20:00 <> 08:20:00
```

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 := 1990-03-02T00:00:00 < 13:00:00
null := 13:00:00 < 14 hours
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>
<n:truth-value> := <n:crisp-type> <= <n:fuzzy-type>
true := 1 <= 2
true := 1990-03-02T00:00:00 <= 1990-03-10T00:00:00
true := 1990-03-02T00:00:00 <= 13:00:00
true := 2 days <= 1 year
true := "aaa" <= "aab"
null := "aaa" <= 1
```

In addition, the <= operators support the same arguments as the **is [in]** operator. Supposing that the first argument is a crisp type and the second a corresponding fuzzy type, the <= operator then returns the **maximum** of u(x) for all x>=r, where r is the value stored in the first argument and u(x) is the fuzzy set provided by the second argument. For example:

```
young := FUZZY SET (0,1),(15,1),(20,0);
middle_aged := FUZZY SET (15,0),(20,1),(60,1), (70,0);

truth value 0 := 25 <= young;
truth value 1 := 25 <= middle_aged;
truth value 1 := 10 <= young;
truth value 1 := 10 <= middle_aged;
truth value 0.5 := 17.5 <= young;
truth value 1 := 17.5 <= middle_aged;
```

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 := 1990-03-02T00:00:00 > 13:00:00
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>
<n:truth-value> := <n:crisp-type> >= <n:fuzzy-type>
false := 1 >= 2
false := 1990-03-02T00:00:00 >= 1990-03-10T00:00:00
false := 1990-03-02T00:00:00 >= 13:00:00
false := 2 days >= 1 year
false := "aaa" >= "aab"
null := "aaa" >= 1
```

The >= operators further support the same arguments as the **is [in]** operator. Supposing that the first argument is a crisp type and the second a fuzzy type, the >= operator then returns the **maximum** of u(x) for all x<=r, while r is the value stored in the first argument and u(x) is the fuzzy set provided by the second argument. For example:

```
young := FUZZY SET (0,1),(15,1),(20,0);
middle_aged := FUZZY SET (15,0),(20,1),(60,1), (70,0);

truth value 1 := 25 >= young;
truth value 1 := 25 >= middle_aged;
truth value 1 := 10 >= young;
truth value 0 := 10 >= middle_aged;
truth value 1 := 17.5 >= young;
truth value 0.5 := 17.5 >= young;
```

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. When used with time-of-day arguments, the order of the right and middle argument may be relevant, as the specified time frame may span over midnight.

When used with arguments that are not time-of-day arguments, operator functionally checks the following relationship

argument 2 <= argument 1 <= argument 3

and returns true if the relationship is satisfied and false if is not satisfied.

Its usage is:

```
<n:Boolean> := <n:ordered> IS WITHIN <n:ordered> TO <n:ordered>
true := 3 IS WITHIN 2 TO 5
false := 3 IS WITHIN 5 TO 2
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"
false := 1990-03-10T15:00:00 IS WITHIN 16:00:00 TO 17:00:00
```

If the middle and right argument of the last example are swapped, then the reference time frame spans midnight:

```
true := 1990-03-10T15:00:00 IS WITHIN 17:00:00 TO 16:00:00
true := time of day of time of order IS WITHIN 22:00:00 to 02:00:00
```

The last example returns true, if the order has been placed after 10 pm and 2 am, independently from the date of the order. The next example checks whether the measurement has been recorded on a weekday.

```
true := DAY OF WEEK OF TIME OF measurement IS WITHIN MONDAY TO FRIDAY
```

Note that the day of week of a primary time results in a number, as well as the keywords MONDAY and FRIDAY. The following code snippet is not valid:

```
null := measurement OCCURRED WITHIN MONDAY to FRIDAY
```

Caution must be used when using the day of week data type with the is . . . within operator, as well as the other comparison operators. Each day of the week is associated with an integer, with Monday = 1 through Sunday = 7 (see Section 8.12). Thus, the range of days specified can not begin before Monday and end after Sunday. For example.

```
 True :=  WEDNESDAY IS WITHIN TUESDAY TO FRIDAY
 True :=   SATURDAY IS WITHIN FRIDAY TO SUNDAY
 FALSE := SATURDAY IS WITHIN FRIDAY TO MONDAY
(this returns false because 6 is not within 5 to 1)
```

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

This operator may be used with small durations as a short-hand notation for some comparisons that can be also represented by using the 'is within to' operator.

Examples:

```
false := time of day of time of request is within 2 hours surrounding 14:00
(true, true, true, false, true) := time of day of time of measurements
                            are within 30 minutes surrounding 13:00
```

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:times> IS WITHIN PAST <n:duration>
true := 1990-03-08T00:00:00 IS WITHIN PAST 3 days
null := 12:00:00 IS WITHIN PAST 2 weeks
```

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
null := 12:00:00 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:times> IS BEFORE <n:times>
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:times> IS AFTER <n:times>
true := 1990-03-08T00:00:00 IS AFTER 1990-03-07T00:00:00
false := now is after 18:00:00
```

The last example assumes, that the MLM runs before 18:00 (for example, now is 2005-01-01T17:30:00).

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. If the first operand is **null**, **true** is always returned. If the second operand is **null** then **null** is returned, except the first one is also **null**. 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)
false := day of week of (time of potassium) IS IN (SATURDAY, SUNDAY)
```

The operator **is in** also checks for containment in a fuzzy set, returning a **truth value**. The arguments are of a crisp and a fuzzy type. The fuzzy type must be derived from the rough crisp type of the other argument (e.g.: if the crisp value is a number, the fuzzy value has to consist of a fuzzy number), otherwise **false** is returned. If we define a fuzzy and a crisp number as:

```
fuzzyVar := Fuzzy Set (0,0), (4,1), (5,0);
crispVar := 2;
```

The crisp number may be correlated to the fuzzy set by the expression

```
crispVar IS IN FuzzyVar
```

This simply gives the value of the fuzzy set (`fuzzyVar`) the point of the crisp value (`crispVar`). For the above example the result will be 0.5.

If one argument is **null**, then **null** is always returned.

Primary times are not used in determining the result. The primary time of the result is determined by the rules in Section 9.1.4. The usage of the **… is [in] …** operator is:

```
<n:truth-value> := <n:crisp-type> IS IN <n:fuzzy-type>
0.5 := 4 IS IN 5 fuzzified by 2;
0.5 := 2 IS IN Fuzzy Set (0,0), (4,1), (5,0);
```

See also Section 9.6.24.

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] Time of day (unary, non-associative)

The **is time of day** operator returns **true** if the argument's data type is time-of-day. Otherwise it returns **false**. **Is time of day** never returns **null**. Its usage is:

```
<n:Boolean> := <n:any-type> IS TIME OF DAY
true := 23:20:00 IS TIME OF DAY
true := 23:20:00.12 IS TIME OF DAY
false := 1991-03-12T00:00:00 IS TIME OF DAY
false := null IS TIME OF DAY
```

9.6.22 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
```

9.6.23 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
```

9.6.24 [not] In (binary, non-associative)

The operator **in** is a synonym of **is in** and behaves in the same manner. Its usage is:

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

See also Section 9.6.14.

9.6.25 Is [not] Object (unary, non-associative)

The **is object** operator returns **true** if the argument is an object (any type of object defined with an Object declaration, as described in Section 11.2.17). Otherwise it returns **false**. Its usage is:

```
<n:Boolean> := <n:any-type> IS OBJECT
```

9.6.26 Is [not] <Object-Type> (unary, non-associative)

The **is <object-type>** operator returns **true** if the argument is an object of the named type (as previously defined with an Object declaration, as described in Section 11.2.17). Otherwise it returns **false**. Its usage is:

```
<n:Boolean> := <n:any-type> IS <OBJECT-TYPE>
RectType := OBJECT [x, y, width, height];
Rect := new RectType;
true := Rect IS RectType;
```

9.6.27 Is [not] Fuzzy (unary, non-associative)

The **is fuzzy** operator returns **true** if the argument's data type is a fuzzy number, fuzzy time or fuzzy duration. Otherwise it returns **false**. **Is fuzzy** never returns null. Its usage is:

```
<n:Boolean> := <n:any-type> IS FUZZY
false := 3 IS FUZZY
true := (FUZZY SET (0,0), (1,1)) IS FUZZY
true := (today fuzzified by 2 days) IS FUZZY
```

9.6.28 Is [not] Crisp (unary, non-associative)

The **is crisp** operator returns **true** if the argument's data type is not a fuzzy number, fuzzy time or fuzzy duration. Otherwise it returns **false**. **Is crisp** never returns null. Its usage is::

```
<n:Boolean> := <n:any-type> IS CRISP
true := 3 IS CRISP
false := (FUZZY SET (0,0), (1,1)) IS CRISP
false := (today fuzzified by 2 days) IS CRISP
```

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

Day-of-week data types are not allowed as arguments to occur comparison operators at this time. Time-of-day data types are allowed and follow standard time-of-day processing.

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

```
<n:Boolean> := <n:any-type> OCCUR EQUAL <n:times>
false := query_result OCCURRED EQUAL 1990-03-01T00:00:00
```

See also Section 9.7.11.

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

```
<n:Boolean> := <n:any-type> OCCUR WITHIN <n:times> TO <n:times>
true := query_result OCCURRED 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:times>
false := query_result 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:times>
false := query_result 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:times>
false := query_result  OCCURRED WITHIN 3 days SURROUNDING 1990-03-10T00:00:00
false := request occurred within 2 hours surrounding 14:00
(true, true, true, false, true) := measurements occurred within 30 minutes
    surrounding 13:00
```

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

```
<n:Boolean> := <n:any-type> OCCUR WITHIN PAST <n:duration>
true := query_result  OCCURRED 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 := query_result  OCCURRED WITHIN SAME DAY AS 1990-03-08T01:01:01
null := query_result OCCURRED WITHIN SAME DAY AS 01:01:01
```

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

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

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

```
<n:Boolean> := <n:any-type> OCCUR AFTER <n:times>
false := query_result  OCCURRED AFTER 1990-03-08T01:01:01
```

9.7.11 Occur [not] At (binary, non-associative)

The **occur at** operator functionally identical to the **occur equal** operator.

```
<n:Boolean> := <n:any-type> OCCUR AT <n:times>
false := query_result  OCCURRED AT 1990-03-01T00:00:00
```

See Section 9.7.2.

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 **%s** 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. Its usage is

```
<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:string>
<1:string> := STRING <m:list of strings>
"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 times 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> := <n : string> MATCHES PATTERN <1 : 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 "_\%";
```

9.8.5 Length (unary, right-associative)

The **length** operator returns the number of characters in a string. Leading or trailing spaces are included in this calculation. Applying the **length** operator to an empty string returns zero, while the **length** of a non-string data type or an empty list is **null**. The **length** operator is different from the **count** operator (see Section 9.12.2), in that **length** is the number of characters in a single string, while **count** is the number of items in a list. Primary times are not preserved.

```
  <n:number> := LENGTH [OF] <n:string>
           7 := LENGTH OF "Example"
          14 := LENGTH "Example String"
           0 := LENGTH ""
        null := LENGTH ()
        null := LENGTH OF null
(8, 3, null) := LENGTH OF ("Negative", "Pos", 2)
```

9.8.6 Uppercase (unary, right-associative)

The **uppercase** operator converts all lowercase characters in a string to uppercase. Non-lowercase characters, including numeric and punctuation characters, are not affected. The **uppercase** of a non-string data type or an empty list is null. Primary times are preserved.

```
                <n:string> := UPPERCASE <n:string>
          "EXAMPLE STRING" := UPPERCASE "Example String"
                       "" := UPPERCASE ""
                     null := UPPERCASE null
                     null := UPPERCASE ()
("5-HIAA", "POS", null) := uppercase ("5-Hiaa", "Pos", 2)
```

9.8.7 Lowercase (unary, right-associative)

The **lowercase** operator converts all uppercase characters in a string to lowercase. Non-uppercase characters, including numeric and punctuation characters, are not affected. The **lowercase** of a non-string data type or empty list is **null**. Primary times are preserved.

```
                <n:string> := LOWERCASE <n:string>
          "example string" := LOWERCASE "Example String"
                       "" := LOWERCASE ""
                     null := LOWERCASE 12.8
                     null := LOWERCASE null
("5-hiaa", "pos", null) := LOWERCASE ("5-HIAA", "Pos", 2)
```

9.8.8 Trim [Left | Right] (unary, right-associative)

The **trim** operator removes leading and trailing white space from a string (see Section 7.1.10). The optional **left** or **right** modifier can be applied to remove leading or trailing white space respectively. Printable characters and embedded white space characters are not affected. The **trim** of a non-string data type or empty list is **null**. Primary times are preserved.

```
          <n:string> := TRIM [LEFT | RIGHT] <n:string>
           "example" := TRIM "  example  "
                  "" := TRIM ""
                null := TRIM ()
         "result:  " := TRIM LEFT  "  result:  "
         "  result:" := TRIM RIGHT "  result:  "
("5 N", "2 E", null) := TRIM (" 5 N", "2 E ", 2)
```

9.8.9 Find...[in] String...[starting at]... (ternary, right-associative)

The **find ... string** operator locates a substring within a target string, and returns a number that represents the starting position of the substring. **Find ... string** is similar to **matches pattern**, but returns a number (rather than a boolean), and does not support wildcards. **Find ... string** is case-sensitive, and returns a zero if the target string does not contain the exact substring. If either the substring or target is not a string data type, **null** is returned. Primary times are not preserved.

The optional modifier **starting at...** can be appended to the **find ... string** operator to control where the search for the substring begins. Omitting the modifier causes the search to begin at the first character of the string. The value following **starting at...** must be an integer, otherwise **null** is returned. If the value following **starting at...** is an integer beyond the length of the target string (i.e. less than 1 or greater than **length** target), zero is returned.

```
<n:number> := FIND <1:string> [IN] STRING <n:string>
<n:number> := FIND <1:string> [IN] STRING <n:string> [STARTING AT <n:number>]
         3 := FIND "a" IN STRING "Example Here"
         5 := FIND "ple" IN STRING "Example Here"
         0 := FIND "s" IN STRING "Example Here"
      null := FIND 2 IN STRING "Example Here"
      null := FIND "a" STRING 510
 (2, 0, 4) := FIND "t" STRING ("start", "meds", "halt")
         7 := FIND "e" IN STRING "Example Here" STARTING AT 1
         1 := FIND "e" IN STRING LOWERCASE "Example Here" STARTING AT 1
        10 := FIND "e" IN STRING "Example Here" STARTING AT 8
        10 := FIND "e" IN STRING "Example Here" STARTING AT 10
```

```
      12 := FIND "e" IN STRING "Example Here" STARTING AT 11
       0 := FIND "e" IN STRING "Example Here" STARTING AT 13
    null := FIND "e" IN STRING "Example Here" STARTING AT 1.5
    null := FIND "e" IN STRING "Example Here" STARTING AT "x"
(10,12) := FIND "e" IN STRING "Example Here" STARTING AT (10,11)
```

9.8.10 Substring … Characters [starting at …] from … (ternary, right associative)

The **substring … characters [starting at …] from …** operator returns a substring of characters from a designated target string. This substring consists of the specified number of characters from the source string beginning with the starting position (either the first character of the string or the specified location within the string). For example **substring 3 characters starting at 2 from "Example"** would return "xam" – a 3 character string beginning with the second character in the source string "Example".

The target string must be a string data type, the starting location within the string must be a positive integer, and the number of characters to be returned must be an integer, or the operator returns **null**. If a starting position is specified, its value must be an integer between 1 and the length of the string, otherwise an empty string is returned. If the requested number of characters is greater than the length of the string, the entire string is returned. If a starting point is specified, and the requested number of characters is greater than the length of the string minus the starting point, the resulting string is the original string to the right of and including the starting position. If the number of characters requested is positive the characters are counted from left to right. If the number of characters requested is negative, the characters are counted from right to left. The characters in a substring are always returned in the order that they appear in the string. Default list handling is observed. Primary times are preserved.

```
<n:string> := SUBSTRING <n:number> CHARACTERS [STARTING AT <n:number>]
                  FROM <n:string>
      "ab" := SUBSTRING 2 CHARACTERS FROM "abcdefg"
  "abcdefg" := SUBSTRING 100 CHARACTERS FROM "abcdefg"
     "def" := SUBSTRING 3 CHARACTERS STARTING AT 4 FROM "abcdefg"
    "defg" := SUBSTRING 20 CHARACTERS STARTING AT 4 FROM "abcdefg"
        null := SUBSTRING 2.3 CHARACTERS FROM "abcdefg"
        null := SUBSTRING 2 CHARACTERS STARTING AT 4.7 FROM "abcdefg"
      null := SUBSTRING 3 CHARACTERS STARTING AT "c" FROM "abcdefg"
      null := SUBSTRING "b" CHARACTERS STARTING AT 4 FROM "abcdefg"
      null := SUBSTRING 3 CHARACTERS STARTING AT 4 FROM 281471
       "d" := SUBSTRING 1 CHARACTERS STARTING AT 4 FROM "abcdefg"
       "d" := SUBSTRING –1 CHARACTERS STARTING AT 4 FROM "abcdefg"
     "bcd" := SUBSTRING –3 CHARACTERS STARTING AT 4 FROM "abcdefg"
       "a" := SUBSTRING 1 CHARACTERS FROM "abcdefg"
       "g" := SUBSTRING –1 CHARACTERS STARTING AT LENGTH OF "abcdefg"
                  FROM "abcdefg"
("Pos","Neg",null):= SUBSTRING 3 CHARACTERS FROM ("Positive","Negative",2)
```

Example: Determine the systolic and diastolic values of patient's blood pressure when observations (bp) are stored as strings like this: "98/72", "121/86", or "138/102".

```
Bp := "121/86";
slash_pos := FIND "/" IN STRING bp;
systolic := SUBSTRING (slash_pos – 1) CHARACTERS FROM  bp;
```

or

```
systolic  := SUBSTRING –3 CHARACTERS STARTING AT (slash_pos - 1) FROM bp;

diastolic := SUBSTRING 3 CHARACTERS STARTING AT (slash_pos + 1) FROM bp;
or
diastolic := SUBSTRING (LENGTH of bp) CHARACTERS STARTING AT (slash_pos + 1)
FROM bp
```

9.8.11 Localized (unary, non-associative)

The **localized** operator returns a string that has been previously defined in the language slot of the MLM's resources category. The string is looked up by choosing the key/value pair defined in the language slot that matches the current language setting of the system which executes the MLM. The argument of the operator specifies the term that is used as key to lookup the value for one specific text resource.

Retrieving the current language setting is implementation specific. If the language cannot be retrieved or no language slot is defined for the current language, the default language of the resources category is used. If the term is not defined in the chosen language slot or if the argument is not a Term, **null** is returned.

According to the examples in Section 6.4.2 its usage is:

```
<n:string> := LOCALIZED <n:term>
   "Caution, the patient has
    the following allergy to
    penicillin documented: " := localized 'msg';
   "The patient's calculated
    creatinine clearance is
    0.33 ml/min."            := creat formatted with localized 'creat';
   null                      := localized 'unknown';
```

Or in an German setting:

```
   "Vorsicht, zu diesem Patienten
    wurde die folgende
    Penicillinallergie
    dokumentiert: "            := localized 'msg';
   "Die berechnete Kreatinin-
    Clearance des Patienten
    beträgt 0,33 ml/min."      := creat formatted with localized 'creat';
   null                        := localized 'unknown';
```

9.8.12 Localized (binary, right-associative)

The binary **localized** operator acts like the unary version of this operator and additionally allows the selection of the target language as second argument. As second operator, either a string constant or a variable can be used. Other expressions are not valid.

This operator can be used if the language of the message has to be different from the current language in the system setting, for example when the system language is English (as the user operates in an English environment), but the recipient of the message text requires another language, such as German.

Regarding the lookup mechanism and the default language handling it acts in the same way like the unary version. In addition, if the second argument does not resolve to a string, the default language is used. Its usage is:

```
<n:string> := LOCALIZED <n:term> by <n:string>
   "Caution, the patient has
    the following allergy to
    penicillin documented: " := localized 'msg' by "en_US";
   "Die berechnete Kreatinin-
    Clearance des Patienten
    beträgt 0,33 ml/min."   := creat formatted with localized 'creat' by
                               lang_setting;       /* lang_setting == "de" */
```

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:times> := <n:times> + <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:times> := <n:duration> + <n:times>
   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:times> := <n:times> - <n:duration>
   1990-03-13T00:00:00 := 1990-03-15T00:00:00 - 2 days
<n:duration> := <n:times> - <n:times>
   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> ** <1: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:times> := <n:duration> AFTER <n:times>
   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:times> := <n:duration> BEFORE <n:times>
   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
```

9.10.4 From (binary, non-associative)

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

```
<n:times> := <n:duration> FROM <n:times>
   2000-09-13T00:08:00 := 2 days FROM 2000-09-11T00:08:00
```

9.10.5 Time of day [of] (unary, right-associative)

The **time of day** operator extracts the time-of-day from a time. Primary times are lost.
Its usage is:

```
<n:time-of-day> := TIME OF DAY [OF] <n:time>
```

```
14:23:17.3 := TIME OF DAY OF 1990-01-03T14:23:17.3
null := TIME OF DAY OF "this is not a time"
/* let time of data0 be 2006-01-01T12:00:00 */
12:00:00 := TIME OF DAY OF (TIME OF data0)
null := TIME OF (TIME OF DAY OF (TIME OF data0))
```

9.10.6 Day of week [of] (unary, right associative)

The **day of week** operator returns a positive integer from 1 to 7 that represents the day of the week of a specified time (Section 8.12). The number 1 corresponds to Monday, 2 corresponds to Tuesday, etc. The number 7 represents Sunday. This operator may be used with a user-defined list of strings to report an actual weekday in an appropriate language, or may be used with the reserved words representing the days of the week. The example below assumes that 2006-0526 was a Friday, 2006-06-03 was a Sunday, 2006-06-06 was a Tuesday, **potassium** is the result of a query with the primary times (2006-06-03T09:04:00, 2006-06-06T16:40:00), and the weekday of now is a Monday.

```
<n:number> := DAY OF WEEK [OF] <n:time>

5 := DAY OF WEEK OF 2006-05-26T13:20:00
(6, 2) := DAY OF WEEK OF (TIME OF potassium)
1 := DAY OF WEEK OF now
null := DAY OF WEEK 15:30:00
true := DAY OF WEEK OF 2006-05-26T13:20:00 = FRIDAY
(true, false) := DAY OF WEEK OF TIME OF potassium IS IN (SATURDAY, SUNDAY)
false := DAY OF WEEK OF now IS IN (SATURDAY, SUNDAY)
```

A more detailed example:

```
weekend := DAY OF WEEK OF eventtime is in (SATURDAY, SUNDAY);
// weekend is true if the event occurred on Saturday or Sunday
weekday := ("Monday", "Tuesday", …, "Sunday");
last_k := last potassium;
last_k_time := time last_k;
msg := "The last potassium was collected on "
    || weekday[DAY OF WEEK OF last_k_time];
//"The last potassium was collected on Tuesday"
```

9.10.7 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)
   null := EXTRACT YEAR 14:23:17.3
```

9.10.8 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
   null := EXTRACT MONTH 14:23:17.3
```

9.10.9 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"
   null := EXTRACT DAY 14:23:17.3
```

9.10.10 Extract Hour (unary, right-associative)

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

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

9.10.11 Extract minute (unary, right-associative)

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

```
<n:number> := EXTRACT MINUTE <n:times>
   23 := EXTRACT MINUTE 1990-01-03T14:23:17.3
   0 := EXTRACT MINUTE 1990-01-03
   null := EXTRACT MINUTE 0000-00-00
   23 := EXTRACT MINUTE 14:23:17.3
```

9.10.12 Extract second (unary, right-associative)

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

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

9.10.13 Replace Year [of] … With (binary, right-associative)

The **replace year of … with** operator allows the replacement of the year part of a time. The result of the **replace year of … with** operator preserves the primary time of the first argument. The numeric second argument must evaluate to a positive integer greater than or equal to 1800, otherwise **null** is returned. Any fractional part of the second argument will be removed before evaluation. For example:

```
<n:time> := REPLACE YEAR [OF] <n:time> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
2011-03-15T15:00:00 := REPLACE YEAR OF var1 WITH 2011;
(2011-03-15T15:00:00, 2010-03-15T15:00:00) := REPLACE YEAR OF var1 WITH
    (2011, 2010);
null := REPLACE YEAR OF var1 WITH -10;
null := REPLACE YEAR OF var1 WITH "7";
var2 := 19:00:00;
null := REPLACE YEAR OF var2 WITH 2011;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2011-09-21T16:30:00, 2011-03-15T15:00:00) := REPLACE YEAR OF var3 WITH 2011;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(1999-09-21T16:30:00, 2000-03-15T15:00:00) := REPLACE YEAR OF var3 WITH
    (1999, 2000);
null := REPLACE YEAR OF var3 WITH (1999, 2000, 2002);
```

9.10.14 Replace Month [of] … With (binary, right-associative)

The **replace month of … with** operator allows the replacement of the month part of a time. The result of the **replace month of … with** operator preserves the primary time of the first argument. The numeric second argument must evaluate to a positive integer between 1 and 12, otherwise **null** is returned. Any fractional part of the second argument will be removed before evaluation. For example:

```
<n:time> := REPLACE MONTH [OF] <n:time> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
1990-11-15T15:00:00 := REPLACE MONTH OF var1 WITH 11;
(1990-11-15T15:00:00, 1990-10-15T15:00:00) := REPLACE MONTH OF var1 WITH (11,
    10);
null := REPLACE MONTH OF var1 WITH 14;
```

```
null := REPLACE MONTH OF var1 WITH "7";
1990-07-15T15:00:00 := REPLACE MONTH OF var1 WITH 7.45;
var2 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-12-21T16:30:00, 2010-12-15T15:00:00) := REPLACE MONTH OF var2 WITH 12;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-06-21T16:30:00, 2010-07-15T15:00:00) := REPLACE MONTH OF var3 WITH (6,
    7);
null := REPLACE MONTH OF var3 WITH (3, 4, 7);
```

9.10.15 Replace Day [of] …With (binary, right-associative)

The **replace day of … with** operator allows the replacement of the day part of a time. The result of the **replace day of … with** operator preserves the primary time of the first argument. The numeric second argument must evaluate to a positive integer between 1 and the number of days in the existing month of the first operator, otherwise, **null** is returned. Any fractional part of the second argument will be removed before evaluation. For example:

```
<n:time> := REPLACE DAY [OF] <n:time> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
1990-03-11T15:00:00 := REPLACE DAY OF var1 WITH 11;
(1990-03-11T15:00:00, 1990-03-10T15:00:00) := REPLACE DAY OF var1 WITH (11,
    10);
null := REPLACE DAY OF var1 WITH 100;
null := REPLACE DAY OF var1 WITH "7";
1990-03-07T15:00:00 := REPLACE DAY OF var1 WITH 7.45;
null := REPLACE DAY OF 1990-02-11T15:00:00 WITH 30;
null := REPLACE DAY OF 1990-02-11T15:00:00 WITH 0.8;
1990-02-01T15:00:00:= REPLACE DAY OF 1990-02-15T15:00:00 WITH 1.8;
var2 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-07T16:30:00, 2010-03-07T15:00:00) := REPLACE DAY OF var2 WITH 7;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-12T16:30:00, 2010-03-23T15:00:00) := REPLACE DAY OF var3 WITH (12,
    23);
null := REPLACE DAY OF var3 WITH (12, 23, 24);
```

9.10.16 Replace Hour [of] … With (binary, right-associative)

The **replace hour of … with** operator allows the replacement of the hour part of a time or time-of-day. The result of the **replace hour of … with** operator preserves the primary time of the first argument. The numeric second argument must evaluate to a positive integer between 0 and 23, otherwise, **null** is returned. Any fractional part of the second argument will be removed before evaluation. For example:

```
<n:times> := REPLACE HOUR [OF] <n:times> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
1990-03-15T11:00:00 := REPLACE HOUR OF var1 WITH 11;
(1990-03-15T11:00:00, 1990-03-15T10:00:00) := REPLACE HOUR OF var1 WITH (11,
    10);
null := REPLACE HOUR OF var1 WITH 100;
null := REPLACE HOUR OF var1 WITH "7";
10:00 := REPLACE HOUR OF 18:00 WITH 10;
var2 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T20:30:00, 2010-03-15T20:00:00) := REPLACE HOUR OF var2 WITH 20;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T07:30:00, 2010-03-15T09:00:00) := REPLACE HOUR OF var3 WITH (7,
    9);
null := REPLACE HOUR OF var3 WITH (7, 9, 13);
```

9.10.17 Replace Minute [of] … With (binary, right-associative)

The **replace minute of … with** operator allows the redefinition of the minute part of a time or time-of-day. The result of the **replace minute of … with** operator preserves the primary time of the first argument. The

numeric second argument must evaluate to a positive integer between 0 and 59, otherwise, **null** is returned. Any fractional part of the second argument will be removed before evaluation. For example:

```
<n:times> := REPLACE MINUTE [OF] <n:times> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
1990-03-15T15:11:00 := REPLACE MINUTE OF var1 WITH 11;
(1990-03-15T15:11:00, 1990-03-15T15:10:00) := REPLACE MINUTE OF var1 WITH
    (11, 10);
null := REPLACE MINUTE OF var1 WITH 100;
null := REPLACE MINUTE OF var1 WITH "7";
18:10 := REPLACE MINUTE OF 18:00 WITH 10;
var2 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T16:15:00, 2010-03-15T15:15:00) := REPLACE MINUTE OF var2 WITH 15;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T16:25:00, 2010-03-15T15:23:00) := REPLACE MINUTE OF var3 WITH
    (25, 23);
null := REPLACE MINUTE OF var3 WITH (25, 23, 7);
```

9.10.18 Replace Second [of] … With (binary, right-associative)

The **replace second of … with** operator allows the redefinition of the second part of a time or time-of-day. The result of the **replace second of … with** operator preserves the primary time of the first argument. The numeric second argument must be a positive number greater than or equal to 0 and strictly lower than 60, otherwise, **null** is returned. Fractional replacement parameters are allowed for the **replace second of … with** operator. For example:

```
<n:times> := REPLACE SECOND [OF] <n:times> WITH <n:number>;
var1 := 1990-03-15T15:00:00;
1990-03-15T15:00:11 := REPLACE SECOND OF var1 WITH 11;
(1990-03-15T15:00:11, 1990-03-15T15:00:10) := REPLACE SECOND OF var1 WITH
    (11, 10);
null := REPLACE SECOND OF var1 WITH -100;
null := REPLACE SECOND OF var1 WITH "7";
18:00:10 := REPLACE SECOND OF 18:00 WITH 10;
var2 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T16:30:33, 2010-03-15T15:00:33) := REPLACE SECOND OF var2 WITH 33;
var3 := (2010-09-21T16:30:00, 2010-03-15T15:00:00);
(2010-09-21T16:30:23, 2010-03-15T15:00:42) := REPLACE SECOND OF var3 WITH
    (23, 42);
null := REPLACE SECOND OF var3 WITH (23, 42, 55);
```

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 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
```

9.11.3 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
```

9.11.4 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
```

9.11.5 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
```

9.11.6 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
```

9.11.7 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
```

9.12 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. If the list argument is a single item, then it is treated as a list of length one. **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:times>
   1990-03-11T03:10:00 := AVERAGE (1990-03-10T03:10:00, 1990-03-12T03:10:00)
   null := AVERAGE (03:10:00, 1990-03-12T03:10:00)
   04:10:00 := AVERAGE (03:10:00, 05:10:00)
<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:times> := MEDIAN <n:times>
   1990-03-11T03:10:00 := MEDIAN (1990-03-10T03:10:00, 1990-03-11T03:10:00,
    1990-03-28T03:10:00)
   03:10:00 := MEDIAN (03:10:00, 02:10:00, 23:10:00)
<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")
```

The **minimum** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the minimum. For example:

```
<1:object> := minimum <n:object> using it.age;  // will return the youngest
                    // person from a list of persons (represented by objects)
180 := minimum (0, 30, 90, 180, 200, 300) using cosine of it;
```

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")
```

The **maximum** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the maximum. For example:

```
<1:object> := maximum <n:object> using it.age;  // will return the oldest
                    // person from a list of persons (represented by objects)
90 := maximum (0, 30, 90, 180, 200, 300) using sinus of it;
```

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 [IsTrue] (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). The optional keyword "**IsTrue**" can be used to increase the readability of statements using the **any** operator. Its usage is:

```
<1:Boolean> := ANY [ISTRUE] <n:any-type>
   true := ANY IsTrue (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 [AreTrue] (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). The optional keyword "**AreTrue**" can be used to increase the readability of statements using the **all** operator. Its usage is:

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

9.12.15 No [IsTrue] (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). The optional keyword "**IsTrue**" can be used to increase the readability of statements using the **no** operator. Its usage is:

```
<1:Boolean> := NO [ISTRUE] <n:any-type>
   false := NO IsTrue (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. If more than one element has the latest primary time, the first (with the lowest index) of these elements will be 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)
```

The **latest** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the latest value. For example:

```
<1:object> := latest <n:object> using it.birthday; //will return the youngest
                     // person from a list of persons (represented by objects)
```

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 more than one element has the earliest primary time, the first (with the lowest index) of these elements will be returned. 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)
```

The **earliest** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the earliest value. For example:

```
<1:object> := earliest <n:object> using it.birthday; //will return the
          // youngest person from a list of persons (represented by objects)
```

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. Its usage is:

```
<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:string>
("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. The primary time of the selected argument is maintained. 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. The primary time of the argument is maintained. Its usage is:

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

3 := INDEX EARLIEST ("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)
   1 := INDEX MIN 3
   null := INDEX MINIMUM ()
   null := INDEX MINIMUM (1,"abc")
```

9.12.22.4 Index Maximum (unary, right associative)

The index **maximum** operator has one synonym: index **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> := INDEX MAXIMUM <n:ordered>
   3 := INDEX MAXIMUM (12,13,14)
   1 := INDEX MAX 3
   null := INDEX MAXIMUM ()
   null := INDEX MAXIMUM (1,"abc")
```

9.12.22.5 Absence of other index operators

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**.

The query aggregation operators follow the default time-of-day handling, when used with a time-of-day argument. The time-of-day value is a point in time within the current day.

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 <1:times> 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 ()

   14 := NEAREST 12:00 FROM data
   // the same as NEAREST 1990-03-18T12:00:00

   14 := NEAREST 23:00 FROM data
   // the same as NEAREST 1990-03-18T23:00:00
```

A more detailed example: a blood glucose query result contains following values **7.0, 10.0, 12.0, query_result** has the primary times 1990-03-18T12:00:00, 1990-03-18T12:30:00, 1990-03-18T13:00:00, and now is 1990-03-18T16:00:00.

The blood glucose level before lunch can be retrieved with:

```
7.0 := NEAREST 12:00 FROM query_result
```

The blood glucose level after ½ hour is:

```
12.0 := NEAREST 12:30 FROM query_result
```

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 Index Of … From … (binary, right-associative)

The **index of ... from** operator expects an arbitrary data value as its first argument and a list as its second argument. It returns a list containing the indices of the occurrences of the given data value within the provided list. If there is more than one occurrence all occurrences are returned. The result is **null** if no such value is found in the list or in case of invalid parameters. The primary times of the arguments are not maintained. The usage of the **index of ... from** operator is:

```
<n:number> := INDEX OF <1:any-type> FROM <m:any-type>
   (4) := INDEX OF 4 FROM (1, 2, 3, 4, "5", "six", 7);
   (5) := INDEX OF “5” FROM (1, 2, 3, 4, "5", "six", 7);
null := INDEX OF 5 FROM (1, 2, 3, 4, "5", "six", 7);

null := INDEX OF null FROM (1, 2, 3, 4, "5", "six", 7);

null := INDEX OF 5 FROM null;
```

```
   (1) := INDEX OF null FROM null;
   (1) := INDEX OF 5 FROM 5;
(1,3,5) := INDEX OF 1 FROM (1, 2, 1, 4, 1, "six", 7);

(3,5) := INDEX OF null FROM (1, 2, null, 4, null, "six", 7);
```

9.13.5 At Least ... [IsTrue|AreTrue] From … (binary, right-associative)

The **at least ... from** operator expects a number (call it N) as its first argument and a homogeneous list of truth values or Boolean as its second argument. The **at least … from** operator returns the n^{th} largest value of the list of truth values or Boolean. If the first argument is not a number or the second parameter contains a non truth value or non Boolean, **null** is returned. If N is greater than the cardinality of the list, **false** is returned. The primary times of the arguments are not maintained. The optional keywords "**IsTrue**" and "**AreTrue**" can be used to increase the readability of statements using the **at least … from** operator. The usage of the operator is:

```
<1:Boolean> := AT LEAST <1:number> [ISTRUE|ARETRUE] FROM <n:Boolean>
TRUE := AT LEAST 1 IsTrue FROM (TRUE, TRUE, FALSE, FALSE)
TRUE := AT LEAST 2 AreTrue FROM (TRUE, TRUE, TRUE, FALSE)
FALSE := AT LEAST 2 FROM (TRUE, FALSE, FALSE, FALSE)
FALSE := AT LEAST 7 AreTrue FROM (TRUE, FALSE, FALSE)
null := AT LEAST 2 YEARS FROM (TRUE, FALSE, FALSE)
null := AT LEAST 2 FROM (TRUE, "true", FALSE)

<1:truth-value> := AT LEAST <1:number> [ISTRUE|ARETRUE] OF <n:truth-value>
0.7 := AT LEAST 2 OF (TRUE, truth value 0.7, truth value 0.1, FALSE)
1 := APPLICABILITY OF (AT LEAST 2 OF (TRUE, Truth value 0.7, FALSE))
FALSE := AT LEAST 7 OF (TRUE, truth value 0.1,FALSE)
null := AT LEAST 2 YEARS OF (TRUE, truth value 0.1,FALSE)
null := AT LEAST 2 OF (TRUE, "true", truth value 0.1,FALSE)
1 := APPLICABILITY OF (AT LEAST 2 OF (TRUE, "true", truth value 0.1,FALSE))
```

9.13.6 At Most ... [IsTrue|AreTrue] From … (binary, right-associative)

The **at most ... from** operator expects a number (call it N) as its first argument and a homogeneous list of truth values or Boolean as its second argument. The **at most … from** operator returns the n^{th} smallest value of the list of truth values or Boolean. If the first argument is not a number or the second parameter contains a non truth value or non Boolean, **null** is returned. If N is greater than the cardinality of the list, **false** is returned. The primary times of the arguments are not maintained. The optional keywords "**IsTrue**" and "**AreTrue**" can be used to increase the readability of statements using the **at most … from** operator. The usage of the operator is:

```
<1:Boolean> := AT MOST <1:number> [ISTRUE|ARETRUE] FROM <n:Boolean>
TRUE := AT MOST 2 AreTrue FROM (TRUE, TRUE, FALSE, FALSE)
FALSE := AT MOST 1 IsTrue FROM (TRUE, TRUE, TRUE, FALSE)
TRUE := AT MOST 2 FROM (TRUE, FALSE, FALSE, FALSE)
FALSE := AT MOST 7 FROM (TRUE, FALSE, FALSE)
null := AT MOST 2 YEARS FROM (TRUE, FALSE, FALSE)
null := AT MOST 2 FROM (TRUE, "true", FALSE)

<1:truth-value> := AT MOST <1:number> [ISTRUE|ARETRUE] OF <n:truth-value>
0.6 := AT MOST 2 OF (TRUE, truth value 0.4, truth value 0.7, FALSE)
1 := APPLICABILITY OF (AT MOST 2 OF (TRUE,0.5,0.7,0.1,FALSE))
FALSE := AT MOST 7 OF (TRUE, truth value 0.5, FALSE)
null := AT MOST 2 YEARS OF (TRUE,0.5,0.7,0.1,FALSE)
null := AT MOST 2 OF (TRUE,"true",0.7,0.1,FALSE)
1 := APPLICABILITY OF (AT MOST 2 OF (TRUE,"true",0.7,0.1,FALSE))
```

9.13.7 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)
```

The **minimum … from** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the minimum. For example:

```
<n:object> := minimum 2 from <n:object> using it.age; //will return the two
          // youngest persons from a list of persons (represented by objects)
```

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)
```

The **maximum … from** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the maximum. For example:

```
<n:object> := maximum 2 from <n:object> using it.age; //will return the two
          // oldest persons from a list of persons (represented by objects)
```

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 Sublist …Elements [Starting at …] From … (ternary, right-associative)

The **sublist … elements [starting at …] from** operator returns a sublist of elements from a designated target list and is similar to the **substring** operator (see 9.8.10). This sublist consists of the specified number of elements from the source list beginning with the starting position (either the first elements of the list or the specified location within the list). For example **sublist 3 elements starting at 2 from (“E”, ”x”, ”a”, ”m”, ”p”, ”l”, ”e”)** would **return ("x”, ”a”, ”m")**–a 3 element list beginning with the second element in the source list.

The target list must be a list data type, the starting location within the list must be a positive integer, and the number of elements to be returned must be an integer, or the operator returns **null**. If target is not a list data type, a list with one element is assumed. If a starting position is specified, its value must be an integer between 1 and the length of the list, otherwise an empty list is returned. If the requested number of elements is greater than the length of the list, the entire list is returned. If a starting point is specified, and the requested number of elements is greater than the size of the list minus the starting point, the resulting list is the original list to the right of and including the starting position. If the number of elements requested is positive the elements are counted from left to right. If the number of elements requested is negative, the elements are counted from right to left. The elements in a sublist are always returned in the order that they appear in the original list. Default list handling is observed. Primary times are preserved.

```
<n:any-type> := SUBLIST <1:number> ELEMENTS [STARTING AT <1:number>] FROM
    <m:any-type>
(1, 2) := SUBLIST 2 ELEMENTS FROM (1, 2, 3, 4, 5)
```

```
(1, 2, 3, 4, 5)  := SUBLIST 100 ELEMENTS FROM (1, 2, 3, 4, 5)
(4, 5, 6) := SUBLIST 3 ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
(4, 5, 6, 7) := SUBLIST 20 ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
null := SUBLIST 2.3 ELEMENTS FROM (1, 2, 3, 4, 5, 6, 7)
null := SUBLIST 2 ELEMENTS STARTING AT 4.7 FROM (1, 2, 3, 4, 5, 6, 7)
null := SUBLIST 3 ELEMENTS STARTING AT "c" FROM (1, 2, 3, 4, 5, 6, 7)
null := SUBLIST "b" ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
() := SUBLIST 3 ELEMENTS STARTING AT 4 FROM 281471
(4) := SUBLIST 1 ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
(4) := SUBLIST –1 ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
(2, 3, 4) := SUBLIST –3 ELEMENTS STARTING AT 4 FROM (1, 2, 3, 4, 5, 6, 7)
(1) := SUBLIST 1 ELEMENTS FROM (1, 2, 3, 4, 5, 6, 7)
```

9.14.7 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:times>
   (1 day) := INCREASE (1990-03-01,1990-03-02)
(1 hour) := INCREASE (13:00:00,14:00:00)

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

9.14.8 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:times>
   ((-1) day) := DECREASE (1990-03-01,1990-03-02)
((-1) hour) := DECREASE (13:00:00,14:00:00)

<n:duration> := DECREASE <m:duration>
   ((-1) day) := DECREASE (1 day, 2 days)
```

9.14.9 % 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)
```

9.14.10 % 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)
```

9.14.11 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 ()
```

The **earliest … from** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the earliest value. For example:

```
<n:object> := earliest 2 from <n:object> using it.birthday; //will return the
      // two oldest persons from a list of persons (represented by objects)
```

9.14.12 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 ()
```

The **latest … from** operator can also be extended by the using modifier as defined for the sort operator (see 9.2.4) to allow more complex calculations of the latest value. For example:

```
<n:object> := latest 2 from <n:object> using it.birthday; //will return the
      // two youngest persons from a list of persons (represented by objects)
```

9.14.13 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.

9.14.13.1 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)
```

9.14.13.2 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)
```

9.14.13.3 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)
   -3 := INT (-2.5)
   -4 := INT (-3.1)
   -4 := INT (-4)
```

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
   -2 :=CEILING (-2.5)
   -3 := CEILING (-3.9)
```

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 (unary, 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
   -4 := 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)
```

9.17 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 **data0** 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 data0
   1990-03-15T15:00:00 := TIME TIME data0
   (null,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;
```

If the identifier on the left hand side of an assignment statement refers to a list, the behavior of the time assignment is undefined. Future versions of the Arden Syntax standard may formally define this behavior. If the right side of the assignment statement does not refer to a time value, the **time** operator assigns **null** to the primary time of the identifier at the left hand side.

9.17.2 Time of Objects

When an object is passed to the **time** operator, the result will be null if one or more attributes do not reference a data item with a primary time, if the data contain primary times but those times differ, or if the object contains no attributes. If all the objects attributes refer to data items with primary times, and all those times are equivalent, then this time is returned as the time of the object. If an attribute contains a list, then the primary time of the object is not defined (returns null) since lists do not have a specific primary time.

```
LabResult := OBJECT [id, value];
result := new LabResult;
```

```
result.id := 123;
time of result.id := 2004-01-16T00:00:00;
result.value := 1.0;
time of result.value := 2004-01-16T00:00:00;

2004-01-16T00:00:00 := time of result; // all attributes have same primary
    time
2004-01-16T00:00:00 := time of result.id;

time of result.id := 2004-01-17T00:00:00;
null := time of result;              // primary times differ
2004-01-17T00:00:00 := time of result.id;
```

9.17.3 Attime (binary, right associative)

The **attime** operator constructs a time value from two time and time-of-day arguments. The result consists of the date of the time arguments and the time of the time-of-day argument. **Null** is returned if it is used with other arguments than time and time-of-day. The primary times are lost.

```
<n:time> := <n:time> ATTIME <n:time-of-day>
2006-06-20T15:00:00 := now ATTIME 15:00:00
2001-01-01T14:30:00 := TIME OF intuitive_new_millenium ATTIME 14:30:00
```

This operator was known as the **at** operator in Arden Syntax 2.6. The change from **at** to **attime** was made to resolve a conflict in context-free grammar (Annex 1) and remove the need for precedence rules to properly parse write statements (12.2.1) that utilize destinations.

9.18 Object Operators

9.18.1 Dot (binary, right associative)

The **dot** operator (".") selects an attribute from an object based on the name following the dot. It takes an expression and an identifier. The expression typically evaluates to an object or a list of objects.

```
<n:any-type> := <expr> "." <identifier>
```

If the expression does not evaluate to an object, or if the object does not contain the named attribute, then null is returned. If the expression evaluates to a list, normal Arden list handling is used and a list is returned. Therefore, if the expression is a list of objects, then a list (of the same length) of the attribute values named by the identifier is returned (a common usage).

```
NameType := object [FirstName, LastName];
/* Assume namelist contains a list of 2 NameType objects */
("John", "Paul")  := namelist.FirstName;
("Lennon", "McCartney") := namelist.LastName;
"John" := namelist[1].FirstName;
 null := namelist[1].Height;
(null, null) := namelist.Height;
```

The dot operator maintains the primary time of the attribute it references.

```
chemistry_panel := object [albumin, calcium, phosphorus];
/* assume patientResult is a single chemistry_panel object with albumin = 4.0
    mg/dL, calcium = 8.7 mg/dL and phosphorus = 3.0 mg/dL on 15 December 2004
    */
calciumPhosphorusProduct := patientResult.calcium * patientResult.phosphorus;
26.1 := calciumPhosphorusProduct;
2004-12-15T16:00:00 := time of patientResult.calcium;
```

Dot operators may be used together, when objects are stored as attributes of other objects.

```
PatientInfo := object [Name, Birthdate];
```

```
/* Assume patient contains an object of type PatientInfo, and the Name
    attribute contains an object of type NameType */
"John" := patient.Name.FirstName;
```

9.18.2 Clone (unary, right associative)

The **clone** operator returns a copy of its argument. Practically, this only affects objects, because these are the only data types which retain identity across multiple operations. (See Annex A6 for details of object identity). When an object is copied, a new object of the same type is created, and all its fields are initialized by assigning values from corresponding fields in the argument object. The fields, which may contain objects, are themselves cloned, resulting in a deep copy. If any field contains a list, that list is cloned, and any objects stored in the list are also cloned.

The **clone** operator insures that no objects are shared between the argument and the result. The **clone** returns another, distinct object that has the same structure and value as the original object.

Effectively, **clone** works like this depending on the argument type:

Object	A deep copy of the object is returned.
List	A copy of the list is returned, which contains a clone of each item in the original list, in the same order.
Other types	The original item is returned.

```
<n:any-type>   := CLONE [OF] <n:any-type>
<Copy of Object> := CLONE OF <Object>
1990-03-15T15:00:00        := CLONE OF 1990-03-15T15:00:00
(1,2, <Copy of Object>) := CLONE (1,2, <Object>)
null := CLONE null
```

When the **clone** operator is applied, the resulting object will contain the same primary times as the argument object. Application of the clone operator to a top level object or any embedded objects ensures that the fields in any new object have the same primary time as the original fields.

9.18.3 Extract Attribute Names ... (unary, right associative)

The **extract attribute names** operator expects an object as its argument. It returns a list containing the attribute names of the object argument. Only the immediate attribute names of the argument are returned. If an attribute is itself an object, the attribute names of the embedded object are not returned, i.e. no nested lists. If the argument is not an object, **null** is returned.

```
<n:string> := EXTRACT ATTRIBUTE NAMES <1:any-type>

(in data slot)
MedicationDose := OBJECT [Medication, Dose, Status];
dose := NEW MedicationDose with "Ampicillin", "500mg", "Active";

(in data slot or logic slot)
dose_attributes := extract attribute names dose
dose_attributes = ("Medication","Dose","Status")
```

9.18.4 Attribute … From … (binary, right associative)

The **attribute … from …** operator expects a string containing the name of an attribute and an object as arguments. It returns the value of the named attribute. If the named attribute is itself an object, the sub-object is returned. If no attributes with the supplied name exists within the named object, null is returned. This is analogous to referring to attributes using dot notation. However, the **attribute … from …** operator allows the name of the attribute to be supplied at run-time rather than requiring knowing the attribute name at design-time.

```
<n:any-type> := attribute <m:string> FROM <m:object>

(in data slot)
MedicationDose := OBJECT [Medication, Dose, Status];
dose := NEW MedicationDose with "Ampicillin", "500mg", "Active";

(in data slot or logic slot)
medication_name := attribute "Medication" from dose
medication_name = "Ampicillin"

medication_name := dose.Medication
medication_name = "Ampicillin"

dose_attributes := extract attribute names dose
medication_name := attribute dose_attributes[1] from dose
medication_name = "Ampicillin"
```

9.19 Fuzzy Operators

9.19.1 Fuzzy Set … (unary, right associative)

The **fuzzy set …** operator creates a new fuzzy set as described in 8.14.1, 8.14.2, or 8.14.3, according to the provided parameters. The operator returns null if the data types are not compatible. Its usage is:

```
<1:fuzzy-number> := FUZZY SET "(" <1:number>, <1:number>")", "(" <1:number>,
    <1:number>")", ...;
Var1 := fuzzy set (2, 0), (3, 1), (4, 1), (5, 0);

<1:fuzzy-time> := FUZZY SET "(" <1:time>, <1:truth-value>")", "(" <1:time>,
    <1:truth-value>")", ...;
Var2 := fuzzy set (now – 2 days, 0), (now, 1), (now + 1 day, 0);
Var3 := fuzzy set (2001-12-12, 0), (2003-12-12, 1), (2009-01-01, 0);

<1:fuzzy-duration> := FUZZY SET "(" <1:duration>, <1:truth-value>")", "("
    <1:duration>, <1:truth-value>")", ...;
Var4 := fuzzy set (2 days, 0), (3 days, 1), (4 days, 1), (5 days, 0);
```

9.19.2 Fuzzified By (binary, non-associative)

The **… fuzzified by …** operator creates a new triangular fuzzy set as described in 8.14.1, 8.14.2, or 8.14.3, according to the provided parameters. The operator returns **null** if the data types are not compatible. Its usage is:

```
<1:fuzzy-number> := <1:number> FUZZIFIED BY <1:number>;
Var1 := 7 fuzzified by 2;

<1:fuzzy-time> := <1:time> FUZZIFIED BY <1:duration>;
```

```
Var2 := now fuzzified by 2 days;

<1:fuzzy-duration> := <1:duration> FUZZIFIED BY <1:duration>;
Var3 := 7 days fuzzified by 2 hours;
```

9.19.3 Defuzzified … (unary, right associative)

The **defuzzified** operator expects a fuzzy data type value as its argument. The operator converts a fuzzy set into a crisp data type. To calculate the result, the **mean of maximum** method is used. This method calculates the average of those intervals' midpoints, which are mapped to the maximum of the fuzzy set image. The usage of the operator is:

```
<n:crisp-type> := DEFUZZIFIED <n:fuzzy-type>
7 := Defuzzified 7 fuzzified by 2;
```

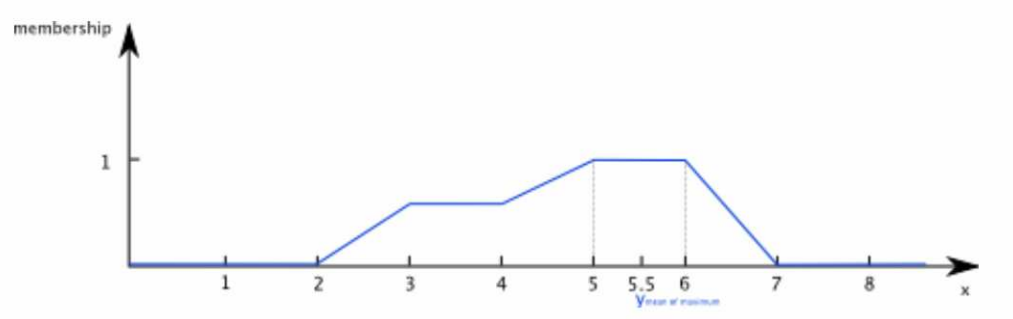


Figure 1: Mean Of Maximum

9.19.4 Applicability [of] … (unary, non-associative)

The **applicability of** operator returns the degree of applicability of a value. Since **null** is not allowed as degree of applicability, a value is always returned (default degree of applicability is 1). The result of the **applicability** operator preserves the primary time and the degree of applicability of its argument; therefore, **applicability applicability x** is equivalent to **applicability x**. Its usage is (assuming that data0 has the degree of applicability of 0.44):

```
<n:truth-value> := APPLICABILITY [OF] <n:any-type>
TRUTH VALUE 0.44 := APPLICABILITY OF data0
TRUTH VALUE 0.44 := APPLICABILITY APPLICABILITY data0
(TRUTH VALUE 1, TRUTH VALUE 1) := APPLICABILITY (3,4)
```

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

```
APPLICABILITY [OF] <n:any-type> := <n:truth-value>;
APPLICABILITY data1 := TRUTH VALUE 0.44;
```

If the identifier on the left hand side of an assignment statement refers to a list, the behavior of the **applicability** assignment is undefined.

9.19.5 Applicability of Objects

When an object is passed to the **applicability** operator, the result will be null if the attributes are referring to data items with different applicabilities, or if the object contains no attributes. If all the objects attributes refer to data items with equivalent applicability, then this applicability is returned as the applicability of the object. If an attribute contains a list, then the applicability of the object is not defined (returns null) since lists do not have a specific applicability.

```
LabResult := OBJECT [id, value];
result := new LabResult;
```

```
result.id := 123;
APPLICABILITY of result.id := TRUTH VALUE 0.44;
result.value := 1.0;
APPLICABILITY of result.value := TRUTH VALUE 0.44;

TRUTH VALUE 0.44 := APPLICABILITY of result; // all attributes have same
    applicability
TRUTH VALUE 0.44 := APPLICABILITY of result.id;

APPLICABILITY of result.id := TRUTH VALUE 0.5;
null := APPLICABILITY of result;            // applicabilities differ
TRUTH VALUE 0.5 := APPLICABILITY of result.id;
```

9.20 Type Conversion Operator

9.20.1 As Number (unary, non-associative)

The **as number** operator attempts to convert a string or Boolean into a number. If conversion into a number is possible, the number is returned, otherwise **null** is returned. The primary time of the argument is preserved. The regular use of this operator would be the conversion of a string that contains a valid number representation i.e. "123" into the represented number. If the string does not contain a valid number, the result will be null. Boolean values are translated as 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;
```

9.20.2 As Time (unary, non-associative)

The **as time** operator attempts to convert a given string into a time. If conversion into a time is possible, the time is returned, otherwise null is returned. The primary time of the argument is preserved. The common use of this operator is the conversion of a string containing a valid date/time format as described in ISO 8601:1988(E), e.g., "1999-12-12" or "1999-12-12T13:41", into a time.

```
<n:time> := <n:string> AS TIME;

<n:time> := <n:time> AS TIME;

1999-12-12 := "1999-12-12" AS TIME;

null := "xyz" AS TIME;
```

```
(1999-12-12, 1999-12-12, null, null, null, 1997-10-31T00:00:00, null) :=
    ("1999-12-12", 1999-12-12, "ABC", Null, True, "1997-10-31T00:00:00", 3
    days) AS TIME;

():= () AS TIME;
```

9.20.3 As String (unary, non-associative)

The **as string** operator attempts to convert any data type into a string. If conversion into a string is possible, the string is returned, otherwise null is returned. The primary time of the argument is preserved.

```
<n:string> := <n:any-type> AS STRING;

"5" := 5 AS STRING;

"null" := null AS STRING;

"true" := True AS STRING;

"false" := False AS STRING;

("7", "8", "4100", "ABC", "null", "true", "false", "1997-10-31T00:00:00", "3
    days") := ("7", 8, 4.1E+3, "ABC", Null, True, False, 1997-10-31T00:00:00,
    3 days) AS STRING;

():= () AS STRING;
```

9.20.4 As Truth Value (unary, non-associative)

The **as truth value** operator attempts to convert a number or Boolean into a truth value. If the conversion into a truth value is possible, the truth value is returned, otherwise null is returned. The primary time of the argument is preserved. The regular use of this operator is to convert a calculated number into the corresponding truth value. If the number is not between 0 and 1, the result will be null. Boolean values are translated as follows: Boolean true is represented as truth value 1 and Boolean false is represented as truth value 0.

```
<n:truth-value> := <n:number> AS TRUTH VALUE;

0.33 := 0.33 AS TRUTH VALUE;

null := "xyz" AS TRUTH VALUE;

null := 400 AS TRUTH VALUE;

<n:truth value> := <n:Boolean> AS TRUTH VALUE;

1 := True AS TRUTH VALUE;

0 := False AS TRUTH VALUE;

 (0, 1, null, 0.33, null, 1, 0, null, null, null) := (0, 1, 4.1E+3, 0.33,
    Null, True, False, 1997-10-31T00:00:00, 3 days, "ABC") AS TRUTH VALUE;

() := () AS TRUTH VALUE;
```

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. However, it is poor programming practice to depend on this.

The following variables cannot be re-assigned outside of the data slot after they have been assigned in the data slot: **event** (Section 11.2.3), **mlm** (Section 11.2.4), and **interface** (Section 11.2.16). 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;
```

10.2.1.1 Object Attribute Assignment

The identifier on the left side of an assignment statement may be specified by an object attribute reference, using the following form:

<identifier> . <attribute-name>

This allows the assignment to individual attributes of an object. The identifier should name a variable. When the statement is executed, if the variable references an object whose type contains an attribute of the specified name, then that attribute value will be set to the result of evaluating the expression on the right side of the assignment statement.

If at execution time the named variable does not refer to an object, or that object does not contain an attribute of the specified name, then this statement will still evaluate the expression but will not assign the result.

```
Rectangle := Object [Left, Top, Width, Height];
rect := new Rectangle;
// assign attributes
rect.Left := 0;
rect.Top := 0;
rect.Width := 10;
rect.Height := 20;

// incorrect assignment
rect.Depth := 30;
null := rect.Depth
```

Note that objects in Arden retain their identity during assignment, references, etc. If more than one reference to an object exists, and that object is modified, other references to the same object will be affected.

```
rect1 := new Rectangle;
// assign attributes
```

```
rect1.Left := 0;
rect1.Top := 0;
rect1.Width := 10;
rect1.Height := 20;

rect2 := rect1;       // references the same Rectangle
rect1.Width := 50;

50 := rect2.Width;       // rect2.width reflects change to shared object
```

10.2.1.2 Enhanced Assignment Statement

In addition to the basic assignment and simple object assignment statements described above, any expression that ends with a **dot** operation (Section 9.18.1) or **element** operation (Section 9.12.18) may be placed on the left hand side of an assignment statement. This does not apply for multiple-assignment. If the left side contains a parenthesised list of variables, then this arbitrary expression syntax may not be used.

This enhancement streamlines the processing of lists and objects. For example,

```
//simple example using index
my_list := 5, 10, 15;
my_list[3] := 20;  //contents of my_list are now 5, 10, 20

//create one object with three nested objects
message_type := OBJECT [id, msg];
my_collection_type := OBJECT [name, message_list];
message_list := ();
for i in 1 seqto 3 do
    message_text := new message_type with i , "this is message " || i;
    message_list := message_list, message_text;
enddo;
my_obj := new my_collection_type with "Reminders", message_list;

//traditional syntax
n := 2;
obj1 := my_obj.message_list [n];
obj1.msg := "this is a replacement message";
message2 := new message_type with 10, "this is message 10";
my_obj.message_list := first (n-1) from  my_obj.message_list,
    message2, last (count of my_obj.message_list - n) from
    my_obj.message_list;

var1 := first (n-1) from  my_obj.message_list;
var2 := last (count of my_obj.message_list - n) from my_obj.message_list;

//enhanced syntax
n := 2;
my_obj.message_list[n].msg := "this is a replacement message";  //modify nth
    item
my_obj.message_list[n] := new message_type with 10, "this is message 10";
    //replace nth item

//additional examples
my_obj.message_list.msg := "This is a test"; //modifies message in all
    objects
my_var := my_obj.message_list.msg; //contents of my_var are "This is a test",
    "This is a test", "This is a test"
my_list[1] := my_var;  //contents of my_list changed to "This is a test",
    "This is a test", "This is a test", 10, 20
```

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 results in any truth value (t) between 0 and 1, the block of statements (<**block**>) is executed and the degree of applicability of each variable is multiplied by t. If the result of the expression is a list, or any single item other than **true or a truth value**, 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
   <block>
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;
```

If, however, <**expr1**> is any truth value (t) between 0 and 1, the program splits: <**block1**> and <**block2**> will be executed in parallel. To this end, each branch is provided with its own set of variables which, accordingly, are duplicated.

Moreover, the degree of applicability of each variable is in the case of <**block1**> multiplied by t, in the case of <**block2**> multiplied by 1 − t. t and 1 − t are called **relative weights** of <**block1**> and <**block2**>, respectively. The weight of an MLM evaluation is 1 as long as it does not split.

The program may branch several times. When the weight of the current branch is w and the MLM evaluation enters a branch with relative weight t, the weight will be reduced to w · t.

In a branch of weight **w**, the range of the degree of applicability of any variable is [0, **w**]. Whenever the content of a variable is changed, its degree of applicability will be reduced to **w,** if necessary. For example:

```
Var := 0;
Con := truth value 0.2;
If con then
  Var := Var + 1;
Else
  Var := Var + 3;
Endif
```

The result of this example are two branches of the MLM execution, where in the first branch Var has the value 1 and the degree of applicability 0.2 and in the second branch Var has the value 3 and the degree of applicability 0.8. That is, the execution of the MLM returns two different values with different degrees of applicability. A nested example would be:

```
Logic:
   Var := 0;
   Bool_true := true;
   Con := truth value 0.2;
   con_second := truth value 0.3;
   IF Con THEN
       Var := Var + 1;
       IF con_second THEN
```

```
            Var := Var + 1;
        ELSE
            Var := Var + 3;
        ENDIF;
    ELSE
        Var := Var + 3;
    ENDIF
    CONCLUDE TRUE;
Action:
    WRITE Var;
```

As above, the MLM execution splits into 2 branches on the first IF statement. The second IF statement branches the first branch again into 2 separate executions. Those 3 Branches are executed in parallel.

The expected output:

- 2 with applicability=0.06 (THEN->THEN)
- 4 with applicability=0.14 (THEN->ELSE)
- 3 with applicability=0.8 (ELSE)

The sum of all applicabilities of a variable is 1.

If a conclude statement occurs within a branch the execution of this branch stops immediately. The execution of all other branches continues. For example:

```
Logic:
    Var := 0;
    Bool_true := true;
    Con := truth value 0.2;
    IF Con THEN
        Var := Var + 1;
        CONCLUDE FALSE;
    ELSE
        Var := Var + 3;
    ENDIF;
    CONCLUDE TRUE;
Action:
    WRITE Var;
```

As specified, the CONCLUDE statement stops the execution of the affected branch. The other branch is executed until the end of the MLM and the expected output is 3 with applicability of 0.8.

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;
```

If the expressions are truth values, the execution of the MLM is split into n branches. Branching into n + 1 blocks is possible by the following statement:

```
IF <expr1> then <block1>
ELSEIF <expr2> then <block2>
...
ELSEIF <exprN> then <blockN>
ELSE <blockN+1>
ENDIF;
```

In this case, the relative weight t_i of the i^{th} branch is given by **<expr$_i$>**, where i = 1, ..., n. If **<expr$_i$>** is undefined, it is treated as $t_i = 0$, in which case the branch is not executed. Moreover, if the sum of the t_i is strictly smaller than 1, the relative weight of $block_{n+1}$ will be $1 - t_1 - ... - t_n$, otherwise this block is skipped.

10.2.2.4 If-Then-Elseif-Aggregate Statement

As shown in chapter 10.2.2.2, the program execution is split, if the condition of an if-then-else statement evaluates to any truth value between 0 and 1.

Once all branches of a program have completed their execution in parallel, because of an unsharp condition, it is difficult to give a general recommendation on how the program should proceed. Two possibilities exist:

(A) The program remains split, that is, all subsequent commands are executed in parallel as well, the action slot included.

(B) The program reunifies. The multiplied variables are merged into single ones.

Both options are available and (A) is the default. Which possibility is chosen should be decided individually, according to the characteristics of each situation.

If (A) is selected, the MLM's results will be provided by each branch separately. The application to which the results are sent—the host system or the calling MLM—must be prepared to deal with the situation. If the MLM is called by another MLM and returns data, the calling MLM splits accordingly as well.

The possibility (B) implies that the task of combining divergent pieces of information is executed within the MLM itself. To opt for (B), the final line of an if-then-else statement is modified: after the keyword endif, the keyword **aggregate** is added. Thus, when writing

```
IF <expr> then <block1>
ELSE <block2>
ENDIF AGGREGATE;
```

the two branches unify after their execution. The program weight is then set to the sum of the weight of the branches, i.e., to the same value as before.

Moreover, corresponding variables are aggregated.

Let Var be a variable defined in at least one branch. As far as the main component is concerned, the procedure is as follows:

- If the content of Var is the same in each branch, the content is taken over.
- Otherwise, if Var is defined in all branches and of the same simple data type except string, the contents are aggregated according to their **weighted middle**.
- If Var is of the same compound type in all branches, we proceed successively with the components in the same manner.
- In the remaining cases, Var is set to null.

The aggregation of the contents of variables, with respect to the degree of applicability and the primary time, is straightforward. The primary time of Var is transferred if coincident in all branches. If distinct times appear, the primary time will be set to **null**.

Furthermore, as might be expected, the degrees of applicability are added. Thus, if left unchanged during the execution of all branches, the degree of applicability prior to the execution of the if-then-else statement will be restored. For example:

```
Logic:
   Var := 0;
   Bool_true := true;
   Con := truth value 0.2;
   IF Con THEN
       Var := Var + 1;
   ELSE
       Var := Var + 3;
   ENDIF AGGREGATE
   CONCLUDE TRUE;
Action:
   WRITE Var;
```

The MLM execution branches at the IF statement and the expressions in the THEN and ELSE branch are executed in parallel. When execution reaches the ENDIF AGGREGATE statement the aggregated value of Var is calculated. After the ENDIF AGGREGATE statement Var has the value (0.2*1 + 0.8*3) / (0.2 + 0.8) = 2.6. The expected output is 2.6 with the applicability of 1.

Example X4.12 illustrates the difference between using the aggregate and not using aggregate. If the MLM is called with an age of 19.9 years the results will be 8 with applicability of 0.1 and 15 with applicability of 0.9. If the MLM is adjusted to use aggregate at the endif the recommended dose will be 14.3 (8*0.1+15*0.9).

10.2.2.5 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;
```

10.2.2.6 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 Switch-Case Statement

The **switch-case** statement permits conditional execution based on the value of an expression. It tests whether an expression (**<expr1>, <expr2>, <expr3> …**) is equal to the value of the provided variable (**<var>**). If the expression is a fuzzy set the **is [in]** operator is used to test equality, in all other cases the **equals** operator is used. If the equality check does not return **false**, the corresponding block of statements (**<block1>, <block2>, <block3> …**) is executed. A block of statements is simply a collection of valid statements, possibly including other **switch-case** statements; thus the **switch-case** statement is a nested structure. If the expression does not match the value of the provided variable, then the corresponding block of statements is not executed. The flow of control then continues with subsequent statements.

The switch-case statement has several forms:

10.2.3.1 Simple Switch-Case Statement

This form executes **<block1>** if the value of **<var>** equals **<expr1>** and **<block2>** if the value is equal to **<expr2>**:

```
SWITCH <var>
    CASE <expr1>
            <block1>
    CASE <expr2>
            <block2>
ENDSWITCH;
```

The following example will set the variable “returnVal” to 7 if the value of the incoming variable “inVal” is equal to 1 and to 9 if the value of the incoming variable “inVal” is equal to 2.

```
switch inVal
  case 1
    returnVal := 7;
  case 2
    returnVal := 9;
endswitch;
```

Equivalent to the **if-then-elseif statement** (see Chapter 10.2.2.3), the execution of **a switch-case** statement can split the program execution into several program branches which will be executed in parallel. This happens if the comparison between the value of a variable and an **<expr>** evaluates to a truth value between 0 and 1.

10.2.3.2 Switch-Case-Default Statement

This form executes **<block1>** if the value of **<var>** equals **<expr1>** and **<block2>** if the value is equal to **<expr2>**. If none of the both match with the value of **<var>** then the default block **<block3>** is executed:

```
SWITCH <var>
    CASE <expr1>
            <block1>
    CASE <expr2>
            <block2>
    DEFAULT
            <block3>
ENDSWITCH
```

The following example will set the variable “returnVal” to 7 if the value of the incoming variable “inVal” is equal to 1, to 9 if the value of the incoming variable “inVal” is equal to 2 and to 0 otherwise.

```
switch inVal
  case 1
    returnVal := 7;
  case 2
    returnVal := 9;
  default
    returnVal := 0; //error state
endswitch;
```

Equivalent to the **if-then-else statement** (see Chapter 10.2.2.3), the execution of **a switch-case-default** statement can split the program execution into several program branches which will be executed in parallel. This happens if the comparison between the value of a variable and an **<expr>** evaluates to a truth value between 0 and 1.

10.2.3.3 Switch-Case-Aggregate Statement

The aggregate operator in the switch-case-aggregate or switch-case-default-aggregate statement acts exactly like in the **if-then-elseif-aggregate** statement. See chapter 10.2.2.4 for more details.

10.2.4 Conclude Statement

The **conclude** statement ends execution in the logic slot. If the expression (**<expr>**) in the conclude statement is a truth value > 0, the applicabilities of all variables are multiplied by this value, and the action slot is executed immediately. Otherwise the whole MLM or the current branch of the 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 10.2.1.2) 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;
```

Furthermore, the reserved word **conclude** can be used in the action slot to retrieve the **degree of applicability** the action slot is executed with.

```
Applicability_of_action_slot:= conclude;
```

10.2.5 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 (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>;
```

10.2.5.1 Commas

Because arguments to a call are separated by commas (see **argument**, Section 11.2.5), 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);
```

10.2.5.2 <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 11.2.4), a valid event variable as defined by the event statement in the data slot (see Section 11.2.3), a valid interface variable as defined by the interface statement in the data slot (see Section 11.2.16), or an MLM, event, or interface variable defined through the use of an include statement (Section 11.2.20).

10.2.5.3 <exprs>

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

10.2.5.4 <var>

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

10.2.5.5 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 11.2.5), 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;
```

10.2.5.6 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;
```

10.2.5.7 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;
```

10.2.5.8 Example: Call Statement

Here is a valid **call** statement:

```
/* Define find_allergies MLM */
find_allergies := MLM 'find_allergies';
/* 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;
```

10.2.5.9 Example: Interface 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;
```

10.2.5.10 Enhanced Assignment in Call Statement

The call statement also supports the same enhanced assignment syntax described in the assignment statement (Section 10.2.1.2)

10.2.6 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 10.2.1.2). 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;
```

10.2.6.1 Breakloop Statement

The block of statements (**<block>**) of a while loop may contain a **breakloop** statement. If the execution reaches such a **breakloop** statement, the direct superior loop will be aborted immediately. If the breakloop statement occurs within a nested loop, it will always apply to the innermost loop only. **Breakloop** statements are only allowed inside of loops.

An example is:

```
num:= 1;
/* Checks each allergen in the medications and stops if patient is allergic
    to it */
while num <= (count med_allergen) do
   allergen:= last(first num from med_allergens);
   allergy_found:= (patient_allergies = allergen);
   /* be returned to the calling MLM */
   If any allergy_found then
       breakloop;  // execution of the while-loop will stop immediately
```

```
   endif;
   /* Increments the counter that is used to stop the while-loop */
   num:= num + 1 ;
   [...]
enddo;
```

10.2.7 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;
```

10.2.7.1 Breakloop Statement

The **breakloop** statement, defined in Section10.2.6.1, is also permitted in the <**block**> of the for loop. When a **breakloop** statement is executed, the <**identifier**> becomes undefined and its value should not be used.

10.2.8 New Statement

The **new** statement causes a new object to be created, and assigns it to the named variable.

```
<var> := NEW <object-identifier>;
<var> := NEW <object-identifier> WITH <expr 1>, <expr 2>, <expr n>;
LET <var> BE NEW <object-identifier>;
LET <var> BE NEW <object-identifier> WITH <expr 1>, <expr 2>, <expr n>;
```

<**object-identifier**> is a name which represents an object type declared previously by an **object or linguistic variable** declaration (see Section 11.2.17, 11.2.18).

```
MedicationDose := OBJECT [Medication, Dose, Status];
dose := NEW MedicationDose with "Ampicillin", "500mg", "Active";
```

In the simple case (without the **with** clause) all attributes of the object are initialized to null. In the full statement, a set of 1 or more comma-separated expressions should follow the **with** reserved word. Each expression is evaluated and assigned as a value of an attribute of the object. They are assigned in the order the attributes were declared in the **object** statement. If the number of expressions is less than the number of attributes, remaining attributes are initialized to null. If the number of expressions is greater than the number of attributes, the extra expressions are evaluated but the results are silently discarded.

As with a **call** statement, commas between expressions will be considered as separating successive attribute initializer expressions rather than as defining a list. If you want to initialize an attribute with a list you need to enclose the list in parentheses. See Section 10.2.5.1 for detailed information.

```
dose := NEW MedicationDose with "Ampicillin", ("500", "700"), "Active";
```

10.2.8.1 New Statement with Named Initializer

There are times when the MLM author may wish to initialize one or more fields explicitly, not necessarily in the order they are declared. It is desirable to have an easy way to initialize certain fields (attributes) directly by name. Allowing field initialization by name is clearer in the MLM code, especially when the object has a large number of fields.

```
my_var := NEW <object-type>
   { WITH <expr_1>, <expr_2>,  ..., <expr_n> }
   { WITH [ attribute_1 := <expr_1>, attribute_2 := <expr_2>, ...,
      attribute_3 := <expr_3>] };
```

The first WITH clause is optional, and allows one or more Arden expressions to be specified. They will get evaluated in order and initialize attributes of the object beginning with the first field specified in the OBJECT declaration.

The second WITH clause is also optional, and uses the square braces [,] to distinguish itself from the ordered parameters of the first WITH clause. The attribute_1,... should be declared names of object attributes. The attribute names may occur in any order, and allow the MLM author to indicate that one or more attributes should be set following the ordered attribute initialization (the first WITH clause). In many cases this may be clearer and more succinct, such as when you wish to set one of the last fields in the attribute list and allow previous fields to have default (null) values.

Note that although both WITH clauses are optional, if they both occur, the ordered attribute list must precede the named initializer list. The named initializer list will also take precedence in the case that an attribute gets initialized in both the ordered list and the named list.

Example:

```
obj_def := object [x, y, z];
testobj := NEW obj_def with [z:=10, y:="roger"];
```

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. To simplify maintenance, it is recommended that, in the absence of conditional assignments, **include**, **object**, **mlm**, **interface**, and **event** statements appear before read statements within the data slot.

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 11.2.3), **mlm** (Section 11.2.4), **interface** (Section 11.2.16), and **object** (Section 11.2.17). 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 the 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 last {select potassium from results};
LET var1 BE READ {select potassium from results} WHERE it occurred within the
    past 1 week;
var1 := READ first 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 last one is assumed to be correct. For example,

```
birthdate := READ last {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 Read As Statement

The **read as** statement is very similar to the **read** statement (11.2.1.1). However, rather than returning query results as a set of lists, where each list represents a collection of values for a particular query field (or column), it returns a single list of objects, each of which consist of named attributes (fields) and values. The attribute names are specified in the **object** declaration, which should have been declared previously (see Section 11.2.17).

```
<var> := READ AS <object-type> <aggregation> <mapping> WHERE <constraint>;
LET <var> BE READ AS <object-type> <aggregation> <mapping> WHERE
    <constraint>;
```

<**object-type**> is a name which represents an object type declared previously by an **object** declaration (see Section 11.2.17).

```
MedicationDose := object [Medication, Dose, Status];
med_doses := read as MedicationDose
     { select med, dosage, status from client where status != "inactive" };
```

It is often easier to manipulate data in this format, because it allows associated values to stay together when lists of data are appended or otherwise manipulated.

It is up to the MLM author to assure that the implementation-specific contents of the curly braces produces the values to be assigned to attributes, and in the correct order.

The following example shows two ways to retrieve three anion gap values, first using **read** and then using **read as**. Note that the text of the implementation-dependent section (curly braces) did not need to change in this example, although of course this standard does not specify anything about this section. The point here is that the same data is retrieved in each case, but it is just returned in a different form.

```
/* in this example the data to calculate three anion gaps are retrieved */
(Na,Cl,HCO3) := read last 3 from {select sodium, chloride, bicarb from
    electro};
/* using READ AS */
AnionGap := Object [Na, Cl, HCO3];
gaps := read as AnionGap last 3 from {select sodium, chloride, bicarb from
    electro};
```

11.2.3 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 10.2.5), and as a Boolean value in a **logic** or **action** slot.. There are two equivalent versions:

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

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

11.2.3.1 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 Boolean value of the variable is false until the MLM is called by the referred event.

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.

11.2.3.2 Example

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

11.2.4 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 10.2.5. 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>;
```

11.2.4.1 Examples

```
LET MLM1 BE MLM 'my_mlm1';
mlm2 := MLM 'my_mlm2.mlm' FROM INSTITUTION "my institution";
```

11.2.4.2 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).

11.2.4.3 Examples

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

11.2.5 Argument Statement

The **argument** statement is used by an MLM that is called by another MLM, as defined in Section 10.2.5. 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 <var1> refers to the first passed argument, the second variable <var2>to the second argument, etc.

If there is a mismatch of variables where 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.

11.2.5.1 Example

In the calling MLM:

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

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

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

11.2.6 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). 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.

11.2.6.1 Example

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

11.2.7 Message As Statement

The **message as** statement is very similar to the **message** statement (11.2.5). However, rather than returning a variable, it returns a single object, which consists of named attributes (fields) and values. The attribute names are specified in the **object** statement, which should have occurred previously in the MLM (see Section 11.2.13). If the mapping clause is empty, it may be omitted in this statement. However, it is up to the implementation if a non-empty mapping clause is allowed.

```
<var> := MESSAGE AS <object-type> <mapping>;
<var> := MESSAGE AS <object-type>;
LET <var> BE MESSAGE AS <object-type> <mapping>;
LET <var> BE MESSAGE AS <object-type>;
```

<object-type> is a name which represents an object type declared previously by an **object** statement (see Section 11.2.17).

11.2.7.1 Example

```
message_obj := OBJECT [subject, text];
high_PTT_msg := MESSAGE AS message_obj {Elevated PTT};
def_msg := MESSAGE AS message_obj; // default mapping clause
```

11.2.8 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.

11.2.8.1 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" };
```

11.2.9 Destination As Statement

The **destination as** statement is very similar to the **destination** statement (11.2.6.1). However, rather than returning a variable, it returns a single object, which consists of named attributes (fields) and values. The attribute names are specified in the **object** statement, which should have occurred previously in the MLM (see Section 11.2.17). If the mapping clause is empty, it may be omitted in this statement. However, it is up to the implementation if a non-empty mapping clause is allowed.

```
<var> := DESTINATION AS <object-type> <mapping>;
<var> := DESTINATION AS <object-type>;
LET <var> BE DESTINATION AS <object-type> <mapping>;
LET <var> BE DESTINATION AS <object-type>;
```

<object-type> is a name which represents an object type declared previously by an **object** statement (see Section 11.2.17).

It is up to the MLM author to assure that the implementation-specific contents of the mapping produces the values to be assigned to attributes, and in the correct order.

11.2.9.1 Example

```
dest_obj := object [dest_method, recip_name, recip_address];
email_attending := DESTINATION AS dest_obj {Attending Phys Email};
def_destination := DESTINATION AS dest_obj;
```

11.2.10 Assignment Statement

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

11.2.11 If-Then Statement

The **if-then** statement, defined in Section 10.2.1.2, is also permitted in the data slot.

11.2.12 Switch-Case Statement

The **switch-case** statement, defined in Section 10.2.3, is also permitted in the data slot.

11.2.13 Call Statement

The **call** statement, defined in Section 10.2.5, is also permitted in the data slot.

11.2.14 While Loop

The **while** loop (with an optional **breakloop** statement), defined in Section 10.2.6, is also permitted in the data slot.

11.2.15 For Loop

The **for** loop (with an optional **breakloop** statement), defined in Section 10.2.7, is also permitted in the data slot.

11.2.16 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 10.2.5). 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.

11.2.16.1 Example

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

11.2.17 Object Statement

The **object** statement assigns object declaration to a variable. This variable should not be reassigned in another statement, and the variable name becomes the object type name (as used in a **read as** statement (Section 11.2.2) or **new** statement (Section 10.2.8). The object statement permits specification of the attributes and attribute ordering of an object type.

```
<var> := OBJECT "[" <attribute-name-1>, <attribute-name-2> ... "]";
LET <var> BE OBJECT "[" <attribute-name-1>, <attribute-name-2> ... "]";
MedicationDose := OBJECT [Medication, Dose, Status];
```

Object attributes follow the same rules as variable names regarding allowed characters. As with variable names, character case is not significant.

11.2.18 Linguistic Variable Statement

Linguistic variables are used to use fuzzy sets in conjunction with other fuzzy sets in order to define a subset of a value range. Assume a value, stored in the variable parameter, out of an arbitrary interval W. Furthermore, assume three fuzzy sets u_1, u_2, and u_3 over W representing the ranges "low", "middle", and "high". In such a case, it is necessary to save these three fuzzy sets together in a single variable of the type **object** whose fields are named according to the ranges, such as:

```
Range :=  object  [low, middle, high];
Value :=  new  Range;
Value.low :=  /definition of the fuzzy set u1 /;
Value.middle := /definition of the fuzzy set u2 /;
Value.high :=  /definition of the fuzzy set u3 /;
```

Whenever a parameter has a low, medium, or high value, it can be evaluated by the following expressions, which provide three truth values, whose sum is truth value 1.

```
Parameter  =  Value.low
Parameter  =  Value.middle
Parameter  =  Value.high
```

To clarify the significance of the fuzzy sets, the keyword **linguistic variable** is used for object declarations where all components are fuzzy data types.

```
RangeOfAge  := linguistic  variable  [young,  middleAge,  old];
Age  :=  new  RangeOfAge;
Age.young := FUZZY SET (0 years, 1), (25  year, 1), (35  years, 0);
Age.middleAge := FUZZY SET(25 years, 0), (35 years, 1), (55 years, 1), (65
    years,  0);
Age.old := FUZZY SET (55  years,  0),  (65  years,  1);
```

Now, if the variable myAge is interpreted as the age of a person, **myAge is Age.young** returns a truth value that indicates the degree to which the statement "is the person young" is justified.

11.2.19 New Statement

The **new** statement, defined in Section 10.2.8, is also permitted in the data slot.

11.2.20 Include Statement

The include statement is analogous to the include statement in C-based languages in that indicates an external MLM may be consulted for object, MLM, event, interface variable and resource definitions. The **include** statement references a variable previously assigned in an MLM statement (11.2.3). When object definitions or resource definitions occur in both the local MLM and a remote MLM, the definition in the local scope always takes precedence. If two remote MLMs define objects or resource definitions with the same name or key, the definitions in MLMs referred to later in the local MLM take precedence. The basic form of the statement is

```
INCLUDE <var>;
```

11.2.20.1 Example

```
mlm2 := MLM 'my_mlm2.mlm' FROM INSTITUTION "my institution";
INCLUDE mlm2;
```

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 12.2.5) 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 11.2.8. 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 11.2.6. The message variable permits institutions to write institution-specific coded MLM messages to databases that will not accommodate the <**expr**> form.

<**expr**> is often a string. If a particular implementation or deployment of Arden Syntax needs to use XML to structure messages, a string expression can be used to compose this message. Appendix X1 shows the recommended DTD for structured messages.

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 12.2.5), 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.

Note that embedding the AT operator (Section 9.17.3) in a WRITE statement can introduce ambiguity. The use of the operator in this context is implementation-specific.

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 10.2.5). 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 10.2.1.2, is also permitted in the action slot.

12.2.4 Switch-Case Statement

The **switch-case** statement, defined in Section 10.2.3, is also permitted in the action slot.

12.2.5 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 10.2.5; the arguments can be accessed with the **argument** statement in Section 11.2.5. 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 11.2.4), or a valid event variable as defined by an event statement in the data slot (see Section 11.2.3).

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

12.2.5.1 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.

12.2.5.2 Example

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

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

12.2.6 While Loop

The **while** loop (with an optional **breakloop** statement), defined in Section 10.2.5.10, is also permitted in the action slot

12.2.7 For Loop

The **for** loop (with an optional **breakloop** statement), defined in Section 10.2.6.1, is also permitted in the action slot.

12.2.8 Assignment Statement

The **assignment** statement, defined in Section 10.2.1, is also permitted in the action slot. Note that with Arden versions prior to 2.5, **assignment** statements were not permitted in the action slot. This capability was added in 2.5 to allow increased flexibility for things like **while** loops, which are not usable without assignment. MLM authors should remember to keep the logic to the logic slot, as much as possible. Refer to Section 12.3, below, for details.

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.

13.1.4 A Constant Time Trigger

For example, on 07-27-2007 at 12:00:00.

13.1.5 A Constant Periodic Time Trigger

For example, start on Friday at 18:00:00, trigger again every week for one year.

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 of** 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 11.2.3.

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 11.2.3, 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 Event Trigger Statement

A **delayed event trigger** statement permits the MLM to be triggered some time after an event occurs. It is of this form:

```
<time-expr> AFTER TIME [OF] <event>
```

<time-expr> is an expression that contains only times expressed as one of the following.

- time constants (see Section 7.1.5),
- as time-of-day constants applied to the at operator in combination with a day-of-week keyword or the reserved words **today,** and **tomorrow** using the **attime** reserved word to combine a day-of-week with a time-of-day in the form <day of week> ATTIME <time of day>
- a duration constant formed by using a number constant with a duration operator

combined using the OR keyword

<event> is an event variable.

<day of week> is a day-of week-variable (see Section 8.12) or the reserved words **today** or **tomorrow.**

<time of day> is a time-of-day variable (see Section 8.11)

For example:

```
TODAY ATTIME 15:00 AFTER TIME OF penicillin_storage
```

The MLM execution is delayed until 15:00 of the day the penicillin_storage event occurs. If the time of day is after 15:00 the MLM will execute immediately unless the evoke slot contains another time constant (see

subsection "use of or"). If the MLM has to be executed the following day, tomorrow can be used as time constant, for example:

```
TOMORROW ATTIME 02:30 AFTER TIME OF penicillin_storage
```

Here, the MLM execution is delayed until 02:30 of the next day. If the execution of the MLM has scheduled for a given day of the week, that day can be also specified within the evoke slot:

```
MONDAY ATTIME 13:00 AFTER TIME OF penicillin_storage
```

The day-of-week is one of the literals Sunday, Monday, Tuesday etc. The MLM execution is delayed until 13:00 of the designated day. If the day of week of "eventtime" is the same as the designated day and eventtime is later than 13:00, the MLM execution is delayed until the following week.

13.3.2.1 *Use of OR*

Time expressions for the delayed trigger can be combined using OR. In this case the whole expression is evaluated to find the next earliest trigger time. For example:

```
MONDAY ATTIME 13:00 OR FRIDAY ATTIME 12:00 AFTER TIME OF penicillin_storage
```

This triggers the MLM on Monday if the event occurs between Friday after 12:00 and Monday before 13:00. If the event occurs outside of this time interval, the MLM is triggered on Friday.

13.3.2.2 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. In the special case, that the delay time is given as an absolute point in time, the triggering is delayed to this timestamp, as soon as the event occurs. If the event occurs after this timestamp, the MLM triggers immediately.

13.3.2.3 Examples

In the following examples, all variables are event variables:

```
3 days after time of penicillin_storage
1992-01-01T00:00:00 AFTER TIME OF penicillin_storage
TOMORROW ATTIME 02:00 AFTER TIME OF penicillin_storage
```

If time expressions are combined with OR, the MLM will be executed at the next scheduled time.

```
TODAY ATTIME 13:00 OR TOMORROW AT 02:00 AFTER TIME OF penicillin_storage
```

13.3.3 Constant Time Trigger Statement

A **constant time trigger statement** permits the MLM to be triggered at a specific instance in time. It has two forms:

```
<time-expr>
<duration-expr> AFTER <time-expr- simple>
```

<duration-expr> is a duration constant formed by using a number constant (see Section 7.1.4) with a duration operator (see Section 9.10.4).

<time-expr> as defined for the delayed event trigger statement above

<time-expr-simple> is defined as <time-expr> but without <duration-expr>

13.3.3.1 Operation

The MLM is triggered at the time specified by the time expression. This is either an absolute point in time, or a relative date (such as tomorrow or simply a duration). A relative date is always evaluated relative to the timepoint when the MLM becomes executable in the system. If a time expression evaluates to a point in time which lies in the past, the MLM is triggered immediately.

For example:

```
TOMORROW ATTIME 02:30
```

The MLM is triggered the day after it got executable at 02:30.

```
20 hours
```

The MLM is triggered 20 hours after it got executable.

13.3.3.2 Examples

In the following examples, variables are event variables:

```
1992-01-01T00:00:00
3 days AFTER 01-01-2007
TOMORROW ATTIME 02:30
```

If used with time-of-day-constants and more than one time constant is specified in the evoke slot, the MLM will be executed at the next scheduled time.

```
TODAY ATTIME 13:00 OR TOMORROW AT 02:00
```

13.3.4 Periodic Event Trigger Statement

A **periodic event 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 <delayed-event-trigger>
EVERY <duration-expr> FOR <duration-expr> STARTING <delayed-event-trigger>
    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 9.10.4).

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

<delayed-event-trigger> is a **delayed event trigger** as defined above.

Simple trigger statements not using a delayed event trigger also are supported. Example:

```
EVERY 1 day FOR 14 days STARTING time of event2
```

13.3.4.1 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.

13.3.4.2 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).

13.3.4.3 Examples

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

```
every 1 day for 14 days starting 1992-01-01T00:00:00 after time of event1
every 1 day for 14 days starting time of event2
every 2 hours for 1 day starting today at 12:00 after time of event3
every 1 week for 1 month starting 3 days after time of event4 until
    last(serum_potassium) > 5.0
```

13.3.5 Constant Periodic Time Trigger Statement

A **constant periodic time trigger** statement permits the repeatedly execution of a MLM at specific instances of time, independent of events. It has two forms:

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

<duration-expr> as defined for the periodic event trigger statement

<Boolean-expr> as defined for the periodic event trigger statement

<constant-time-trigger> is a **constant time trigger** as defined above.

Consider the following evoke slot:

```
EVERY 1 DAY FOR 5 months STARTING 2008-10-01T06:30;
```

This evoke slot could be used to run an influenza rule every day for the five months of the 2008 flu season.

13.3.5.1 Operation

As defined for the **periodic event trigger** statement, but the first execution is determined by a constant time trigger statement.

13.3.5.2 Until

As defined for the **periodic event trigger** statement.

13.3.5.3 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 2 hours for 1 day starting today at 12:00
every 1 week for 1 month starting 3 days after 1992-01-01T00:00:00 until
    last(serum_potassium) > 5.0
```

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>
         <resources_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>
<resources_category> ::=
        /* empty */
      | "RESOURCES:" <resources_body>
<resources_body> ::=
        <default_slot>
        <language_slots>
```

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

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

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

```
< <mlmname_slot> ::=
          "MLMNAME:" <mlmname_text> ";;"
        | "FILENAME:" <mlmname_text> ";;"
                                        /* the "FILENAME:" form is only valid */
                                        /* combination with the empty version */
                                        /* of <arden_version_slot>            */
<mlmname_text> ::=
          <letter>
        | <mlmname_text><mlmname_text_rest>
 <mlmname_text_rest> ::=
          <letter>
        | <digit>
        | "."
        | "-"
        | "_"
 <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" "2.1"
        | "VERSION" "2.5"
        | "VERSION" "2.6"
        | "VERSION" "2.7"
        | "VERSION" "2.8"
 <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>       /* see ANSI/NISO Z39.88          */
                         /* for preferred OpenURL format  */
    <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>            /* see ANSI/NISO Z39.88          */
                         /* for preferred OpenURL format  */
```

/****** 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>
```

/****** resource slots ******/

```
<default_slot> ::=
        "DEFAULT:" <iso639-1> ";;"   /* 2-character language code */
<language_slots> ::=
        <language_slots> <language_slot>
      | <language_slot>
<language_slot> ::=
        "LANGUAGE:" <iso639-1>
        <resource_terms>
        ";;"
<resource_terms> ::=
        <resource_terms> ";" <term> ":" <string>
      | <term> ":" <string>
```

/****** 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" <expr> "DO" <logic_block> ";" "ENDDO"
      | <logic_switch>
      | "BREAKLOOP"
      | "CONCLUDE" <expr>
<logic_if_then_else2> ::=
        <expr> "THEN" <logic_block> ";" <logic_elseif> ";"
<logic_elseif> ::=
        <logic_endif>
      | "ELSE" <logic_block> ";" <logic_endif>
      | "ELSEIF" <logic_if_then_else2>
```

```
<logic_endif> ::=
         "ENDIF"
       | "ENDIF" "AGGREGATE"
<logic_assignment> ::=
         <identifier_becomes> <expr>
       | <time_becomes> <expr>
       | <applicability_becomes> <expr>
       | <identifier_becomes> <call_phrase>
       | "(" <data_var_list> ")" ":=" <call_phrase>
       | "LET" "(" <data_var_list> ")" "BE" <call_phrase>
       | <identifier_becomes> <new_object_phrase>
       | <identifier_becomes> <fuzzy_set_phrase>
<expr_fuzzy_set> ::=
         <expr>
       | <fuzzy_set_phrase>
<identifier_becomes> ::=
         <identifier_or_object_ref> ":="
       | "LET" <identifier_or_object_ref> "BE"  | "NOW" ":="
<logic_switch> ::=
         "SWITCH" <identifier> ":"
         <logic_switch_cases>
         <logic_endswitch>
<logic_endswitch> ::=
         "ENDSWITCH" ";"
       | "ENDSWITCH" "AGGREGATE" ";"
<logic_switch_cases> ::=
         /* empty */
       | "CASE" <expr_factor> <logic_block> <logic_switch_cases>
       | "DEFAULT" <expr_factor> <logic_block>


<identifier_or_object_ref> ::=
       <identifier>
       | <identifier_or_object_ref> "[" <expr> "]"
       | <identifier_or_object_ref> "." <identifier_or_object_ref>
                                                    /* field reference */
<time_becomes> ::=
         "TIME" "OF" <identifier> ":="
       | "TIME" <identifier> ":="
       | "LET" "TIME" "OF" <identifier> "BE"
       | "LET" "TIME" <identifier> "BE"
```

```
<applicability_becomes> ::=
         "APPLICABILITY" "OF" <identifier> ":="
       | "APPLICABILITY" <identifier> ":="
       | "LET" "APPLICABILITY" "OF" <identifier> "BE"
       | "LET" "APPLICABILITY" <identifier> "BE"


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

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

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

<expr_sort> ::=
         <expr_add_list>
       | <expr_add_list> "MERGE" <expr_sort>
       | "SORT"  <sort_option>  <expr_sort>
       | <expr_add_list> "MERGE" <expr_sort>  "USING" <expr_function>
       | "SORT"  <sort_option>  <expr_sort>  "USING" <expr_function>

<sort_option> ::=
         /*empty*/
       | "TIME"
       | "DATA"
       | "APPLICABILITY"

<expr_add_list> ::=
         <expr_remove_list>
       | "ADD" <expr_where> "TO" <expr_where>
       | "ADD" <expr_where> "TO" <expr_where> "AT" <expr_where>

<expr_remove_list> ::=
         <expr_where>
       | "REMOVE" <expr_where> "FROM" <expr_where>

<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_find_string>
       | <expr_string> <simple_comp_op> <expr_string>
       | <expr_string> <is> <main_comp_op>
       | <expr_string> <is> "NOT" <main_comp_op>
       | <expr_string> <in_comp_op>
       | <expr_string> "NOT" <in_comp_op>
       | <expr_string> <occur> <temporal_comp_op>
       | <expr_string> <occur> "NOT" <temporal_comp_op>
       | <expr_string> <occur> <range_comp_op>
       | <expr_string> <occur> "NOT" <range_comp_op>
       | <expr_string> "MATCHES" "PATTERN" <expr_string>
<expr_find_string> ::=
         "FIND" <expr_string> "IN" "STRING" <expr_string>
    <string_search_start>
       | "FIND" <expr_string> "STRING" <expr_string> <string_search_start>
<expr_string> ::=
         <expr_plus>
       | <expr_string> "||" <expr_plus>
       | <expr_string> "FORMATTED" "WITH" <format_string>
       | <expr_string> "FORMATTED" "WITH" <expr_plus>
       | "TRIM" <trim_option> <expr_string>
       | <case_option> <expr_string>
       | "SUBSTRING" <expr_plus> "CHARACTERS" <string_search_start> "FROM"
    <expr_string>
<format_string> ::=
       """ <format_specification> """    /* The format string is a true     */
                                         /* Arden Syntax string, enclosed   */
                                         /* in a single pair of double      */
                                         /* quotes (" )                     */
<format_specification> ::=   /* See Section 9.8.2 and Annex 5 for          */
                             /* explanation of valid combination and their */
                             /* meanings.                                  */
         <format_specification> <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> ::=
         /* empty */
       | <digits>
<precision> ::=
         /* empty */
       | "."<digits>
<trim_option> ::=
         /* empty */
       | "LEFT"
       | "RIGHT"
<case_option> ::=
       "UPPERCASE"
       | "LOWERCASE"
<string_search_start> ::=
        /* empty */
       | "STARTING" "AT" <expr_plus>
```

```
<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_attime>
       | <expr_function> "**" <expr_function>
                 /* exponent (second argument) must be an expression    */
                 /* that evaluates to a scalar number                   */

<expr_attime> ::=
         <expr_before>
       | <expr_before> "ATTIME" <expr_attime>

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

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

<expr_duration> ::=
         <expr_function>
       | <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>
```

```
        | "REPLACE" <timepart> "OF" <expr_function> "WITH" <expr_factor>
        | "REPLACE" <timepart> <expr_function> "WITH" <expr_factor>
        | <from_of_func_op> <expr_function> "USING" <expr_function>
        | <from_of_func_op> "OF" <expr_function> "USING" <expr_function>
        | <from_of_func_op> <expr_factor> "FROM" <expr_function> "USING"
    <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>
        | <at_least_most_op> <expr_factor> "FROM" <expr_function>
        | <at_least_most_op> <expr_factor> "ISTRUE" "FROM" <expr_function>
        | <at_least_most_op> <expr_factor> "ARETRUE" "FROM" <expr_function>
        | "INDEX" "OF" <expr_factor> "FROM" <expr_function>
        | <index_from_func_op> <expr_factor> "FROM" <expr_function>
        | <expr_factor> "AS" <as_func_op>
        | <expr_attribute_from>
        | <expr_sublist_from>

<expr_attribute_from> ::=
         "ATTRIBUTE" <expr_factor> "FROM" <expr_factor>

<expr_sublist_from> ::=
         "SUBLIST" <expr_factor> "FROM" <expr_factor>
        | "SUBLIST" <expr_factor> "STARTING" "AT" <expr_factor> "FROM"
    <expr_factor>

<expr_factor> ::=
         <expr_factor_atom>
        | <expr_factor_atom> "[" <expr> "]"      /* number [<expr>] is not */
                                                 /* a valid construct      */
        | <expr_factor> "." <identifier>         /* object dot notation    */

<expr_factor_atom> ::=
         <identifier>
        | <number>
        | <string>
        | <time_value>
        | <boolean_value>
        | <weekday_literal>
        | "TODAY"
        | "TOMORROW"
        | "NULL"
        | "CONCLUDE" /* only available in the action slot */
        | <it>                    /* Value of <it> is NULL outside of a     */
                                  /* where clause and may be flagged as an  */
                                  /* error in some implementations.         */
        | "(" ")"
        | "(" <expr> ")"
        | "(" <expr_fuzzy_set> ")"
```

/****** 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>
                   | "AT" <expr_string>
```

```
<unary_comp_op> ::=
         "PRESENT"
       | "NULL"
       | "BOOLEAN"
       | "TRUTH VALUE"
       | "CRISP"
       | "FUZZY"
       | "NUMBER"
       | "TIME"
       | "DURATION"
       | "STRING"
       | "LIST"
       | "OBJECT"
       | "LINGUISTIC VARIABLE"
       | <identifier>   /*names an object i.e. left side of OBJECT statement*/
       | "TIME" "OF" "DAY"
<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>
<in_comp_op> ::=
         "IN"
<of_read_func_op> ::=
         "AVERAGE"| "AVG"
       | "COUNT"
       | "EXIST"  | "EXISTS"
       | "SUM"
       | "MEDIAN"
```

```
<of_noread_func_op> ::=
         "ANY"
       | "ANY" "ISTRUE"
       | "ALL"
       | "ALL" "ARETRUE"
       | "NO"
       | "NO"  "ISTRUE"
       | "SLOPE"
       | "STDDEV"
       | "VARIANCE"
       | "INCREASE"
       | "PERCENT" "INCREASE" | "%" "INCREASE"
       | "DECREASE"
       | "PERCENT" "DECREASE" | "%" "DECREASE"
       | "INTERVAL"
       | "TIME"
       | "TIME" "OF" "DAY"
       | "DAY" "OF" "WEEK"
       | "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"
       | "EXTRACT" "TIME" "OF" "DAY"
       | "STRING"
       | "EXTRACT" "CHARACTERS"
       | "REVERSE"
       | "LENGTH"
       | "CLONE"
       | "EXTRACT" "ATTRIBUTE" "NAMES"
```

```
       | "APPLICABILITY"
       | "DEFUZZIFIED"

<from_func_op> ::=
         "NEAREST"

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

<from_of_func_op> ::=
         "MINIMUM"| "MIN"
       | "MAXIMUM"| "MAX"
       | "LAST"
       | "FIRST"
       | "EARLIEST"
       | "LATEST"

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

<as_func_op> ::=
         "NUMBER"
       | "TIME"
       | "STRING"
       | "TRUTH VALUE"


<at_least_most_op> ::=
         "AT" "LEAST"
       | "AT" "MOST"


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

<timepart> ::=
         "YEAR"
       | "MONTH"
       | "DAY"
       | "HOUR"
       | "MINUTE"
       | "SECOND"
```

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

```
<string> ::=
         <plainstring>
       | "LOCALIZED" <term> <localize_option>
<localize_option> ::=
         /* empty */
       | "BY" <plainstring>
       | "BY" <identifier>
<boolean_value> ::=
         "TRUE"
       | "FALSE"
       | "TRUTH VALUE" <number>
       | "TRUTH VALUE" "TRUE"
       | "TRUTH VALUE" "FALSE"
<time_value> ::=
         "NOW"
       | <iso_date_time>
       | <iso_date>
       | "EVENTTIME"
       | "TRIGGERTIME"
       | "CURRENTTIME"
       | <time_of_day>
```

/****** 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" <expr> "DO" <data_block> ";" "ENDDO"
       | <data_switch>
       | "BREAKLOOP"
       | "INCLUDE" <identifier>
<data_if_then_else2> ::=
         <expr> "THEN" <data_block> ";" <data_elseif>
<data_elseif> ::=
         <data_endif>
       | "ELSE" <data_block> ";" <data_endif>
       | "ELSEIF" <data_if_then_else2>
```

```
<data_endif> ::=
         "ENDIF"
       | "ENDIF" "AGGREGATE"
<data_switch> ::=
         "SWITCH" <identifier> ":"
         <data_switch_cases>
         <data_endswitch>
<data_endswitch> ::=
         "ENDSWITCH" ";"
       | "ENDSWITCH" "AGGREGATE" ";"
<data_switch_cases> ::=
         /* empty */
       | "CASE" <expr_factor> <data_block> <data_switch_cases>
       | "DEFAULT" <expr_factor> <data_block>


<data_assignment> ::=
         <identifier_becomes> <data_assign_phrase>
       | <time_becomes> <expr>
       | <applicability_becomes> <expr>
       | "(" <data_var_list> ")" ":=" "READ" <read_phrase>
       | "LET" "(" <data_var_list> ")" "BE" "READ" <read_phrase>
       | "(" <data_var_list> ")" ":=" "READ" "AS" <identifier> <read_phrase>
       | "LET" "(" <data_var_list> ")" "BE" "READ"
                 "AS" <identifier> <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>
       | "MESSAGE" "AS" <identifier> <mapping_factor>
       | "MESSAGE" "AS" <identifier>
       | "DESTINATION" <mapping_factor>
       | "DESTINATION" "AS" <identifier> <mapping_factor>
       | "DESTINATION" "AS" <identifier>
       | "ARGUMENT"
       | "OBJECT" <object_definition>
       | "LINGUISTIC VARIABLE" <object_definition>
       | <call_phrase>
       | <new_object_phrase>
       | <fuzzy_set_phrase>
       | <expr>
<fuzzy_set_phrase> ::=
         "FUZZY SET" <fuzzy_set_init_list>
       | <expr_duration> "FUZZIFIED BY" <expr_duration>
       | <expr_factor> "FUZZIFIED BY" <expr_factor>
<fuzzy_set_init_list> ::=
         <fuzzy_set_init_element>
       | <fuzzy_set_init_list> "," <fuzzy_set_init_element>
<fuzzy_set_init_element> ::=
         "(" <fuzzy_set_init_factor> "," <expr_factor> ")"
<fuzzy_set_init_factor> ::=
         <expr_factor>
       | <number> <duration_op>
<read_phrase> ::=
         <read_where>
       | <of_read_func_op> <read_where>
       | <of_read_func_op> "OF" <read_where>
       | <from_of_func_op> <read_where>
       | <from_of_func_op> "OF" <read_where>
       | <from_of_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>
       | <mapping_factor> "WHERE" <it> <occur> <range_comp_op>
       | <mapping_factor> "WHERE" <it> <occur> "NOT" <range_comp_op>
       | "(" <read_where> ")"
<mapping_factor> ::=
         "{" <data_mapping> "}"
<object_definition> ::=
         "[" <object_attribute_list> "]"
   <object_attribute_list> ::=
         <identifier>
           | <identifier> "," <object_attribute_list>
   <new_object_phrase> ::=
         "NEW" <identifier>
       | "NEW" <identifier> "WITH" <expr>
       | "NEW" <identifier> "WITH" "[" <object_init_list> "]"
       | "NEW" <identifier> "WITH" <expr> "WITH" "[" <object_init_list> "]"
```

```
<object_init_list> ::=
        <object_init_element>
 | <object_init_list> "," <object_init_element>
<object_init_element> ::=
        <identifier> ":=" <expr>
```

/****** evoke block ******/

```
<evoke_block> ::=
        <evoke_statement>
      | <evoke_block> ";" <evoke_statement>
<evoke_statement> ::=
        /* empty */
      | <event_or>
      | <evoke_time>
      | <delayed_evoke>
      | <qualified_evoke_cycle>
      | "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> ::=e
        "(" <event_or> ")"
      | <identifier>
<delayed_evoke>::=
    <evoke_time_expr_or> "AFTER" <event_time>
    |<evoke_time_expr_or>
    |<evoke_duration> "AFTER" <evoke_time_or>

<event_time> ::=
 "TIME" <event_any>
    | "TIME" "OF" <event_any>

<evoke_time_or>::=
```

```
    <evoke_time>
    | <evoke_time> "OR" <evoke_time_or>

<evoke_time_expr_or> ::=
    <evoke_time_expr>
| <evoke_time_expr> "OR" <evoke_time_expr_or>

<evoke_time_expr>::=
    <evoke_duration>
    | <evoke_time>

<evoke_time> ::=
      <iso_date_time>
    | <iso_date>
    | <relative_evoke_time_expr>

<evoke_duration> ::=
      <number> <duration_op>

<relative_evoke_time_expr>::=
      "TODAY" "ATTIME" <time_of_day>
    | "TOMORROW" "ATTIME" <time_of_day>
    | <weekday_literal> "ATTIME" <time_of_day>

<weekday_literal> ::=
      "SUNDAY"
    | "MONDAY"
    | "TUESDAY"
    | "WEDNESDAY"
    | "THURSDAY"
    | "FRIDAY"
    | "SATURDAY"

<qualified_evoke_cycle> ::=
      <simple_evoke_cycle>
    | <simple_evoke_cycle> "UNTIL" <expr>

<simple_evoke_cycle> ::=
```

```
      "EVERY" <evoke_duration> "FOR" <evoke_duration> "STARTING"
    <starting_delay>


<starting_delay>::=
      <event_time>
    | <delayed_evoke>
/****** 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" <expr> "DO" <action_block> ";" "ENDDO"
       | <action_switch>
       | "BREAKLOOP"
       | <call_phrase>
       | <call_phrase> "DELAY" <expr>
       | "WRITE" <expr>
       | "WRITE" <expr> "AT" <identifier>
       | "RETURN" <expr>
       | <identifier_becomes> <expr>
       | <time_becomes> <expr>
       | <applicability_becomes> <expr>
       | <identifier_becomes> <new_object_phrase>
<action_if_then_else2> ::=
       <expr> "THEN" <action_block> ";" <action_elseif>
<action_elseif> ::=
         <action_endif>
       | "ELSE" <action_block> ";" <action_endif>
       | "ELSEIF" <action_if_then_else2>
<action_endif> ::=
         "ENDIF"
       | "ENDIF" "AGGREGATE"
<action_switch> ::=
         "SWITCH" <identifier> ":"
         <action_switch_cases>
         <action_endswitch>
<action_endswitch> ::=
         "ENDSWITCH" ";"
       | "ENDSWITCH" "AGGREGATE" ";"
```

```
<action_switch_cases> ::=
        /* empty */
      | "CASE" <expr_factor> <action_block> <action_switch_cases>
      | "DEFAULT" <expr_factor> <action_block>
```

/****** lexical constructs ******/

```
/*  Unless otherwise specificed, characters are the printable ASCII        */
/*  characters (ASCII 33 through and including 126),  ( See 5.2 )          */
/*  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.                                               */

<plainstring> ::=
        /* any string of characters enclosed in double quotes (" ASCII 22) */
        /* with nested ""                                                  */
        /*  (character set limitations do not apply here)                  */
        /*  one possible regular expression to match Arden Syntax strings: */
        /*      /"([^"]|/"/")*/"                                           */

<identifier> ::=
      /* up to 80 characters total (no reserved words allowed) */
        <letter> <identifier_rest>

<identifier_rest> ::=       /* no spaces are permitted between elements */
        /* empty */
      | <letter> <identifier>
      | <digit> <identifier>
      | "_" <identifier>

<text> ::=
        /* any string of characters without ";;" */

<format_text> ::=
        /* any string of characters */

<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>

<time_of_day> ::=           /* no spaces are permitted between elements */
      <digit> <digit> ":" <digit> <digit>
      <seconds>
      <time_zone>
```

```
<seconds> ::=                /* no spaces are permitted between elements */
         ":" <digit> <digit> <fractional_seconds>
       | /* empty */
<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                    */
<multi_line_comment> ::=
       /* any string of characters enclosed between pairs of "/*" and"*/"  */
       /* (character set limitations do not apply here)                    */
<single_line_comment> ::=
       /* any string of characters located between "//" and                 */
       /* an end-of-line markner (CR, LF, or CR/LF pair)                   */
       /* (character set limitations do not apply here)                    */
<iso639-1> ::=
       /* 2-letter character code as defined by standard ISO 639-1         */
```

A2 RESERVED WORDS

Listed here in alphabetic order are all the reserved words. None of these words may be used as variable names.

Abs	citations	every	interval
action	conclude	evoke	is
add	cos	exist	istrue
after	cosine	exists	it
aggregate	count	exp	keywords
ago	clone	expired	knowledge
alert	crisp	explanation	language
all	currenttime	extract	last
and	data	false	latest
any	data_driven	filename	le
applicability	data-driven	find	least
arccos	date	first	left
arcsin	day	floor	length
arctan	days	following	less
arden	decrease	for	let
are	default	formatted	library
aretrue	defuzzified	friday	linguistic
argument	delay	from	links
as	destination	fuzzified	list
at	do	fuzzy	localized
attribute	duration	ge	log
author	earliest	greater	log10
average	elements	gt	logic
avg	else	hour	lowercase
be	elseif	hours	lt
before	enddo	if	maintenance
Boolean	endif	in	matches
breakloop	endswitch	include	max
by	end	increase	maximum
call	eq	index	median
case	equal	institution	merge
ceiling	event	int	message
characters	eventtime	interface	min

minimum	percent	sqrt	truncate
minute	preceding	starting	truth
minutes	present	stddev	tuesday
mlm	priority	string	type
mlmname	production	substring	unique
mlm_self	purpose	sublist	until
month	read	sum	uppercase
monday	refute	sunday	urgency
months	remove	support	using
most	replace	surrounding	validation
names	research	switch	value
ne	resources	tan	variable
nearest	return	tangent	variance
new	reverse	testing	version
no	right	than	was
not	round	the	wednesday
now	same	then	week
null	saturday	they	weeks
number	second	thursday	were
object	seconds	time	where
occur	seqto	title	while
occurred	set	to	with
occurs	sin	today	within
of	sine	tomorrow	write
or	slope	triggertime	year
past	sort	trim	years
pattern	specialist	true	

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 º. The symbol [of] means that the word **of** is optional, and does not affect the logic of the operator. The symbol [in] means that the work **in** 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.

... fuzzified by ... (non-associative)

Fuzzy Set ... (Right associative)

... [...] (non-associative)

, ... (non-associative)

... , ... (left associative)
... merge ... (left associative)
... merge ... using … (left-associative)

---

sort ... (non-associative)
sort ... using …(non-associative)

---

add ... to ... (non-associative)
add ... to ... at … (non-associative)
remove ... from ... (non-associative)

... where ... (non-associative)

... or ... (left associative)

... and ... (left associative)

---

not ... (non-associative)

... = ... º ... eq ... º ... is equal ... (non-associative)
... <> ... º ... ne ... º ... is not equal ... (non-associative)
... < ... º ... lt ... º ... is less than ... º ... is not greater than or equal ... (non-associative)
... <= ... º ... le ... º ... is less than or equal ... º ... is not greater than ... (non-associative)
... > ... º ... gt ... º ... is greater than ... º ... is not less than or equal ... (non-associative)
... >= ... º ... ge ... º ... is greater than or equal ... º ... 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 ... º … occur at … (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 ... º … in … (non-associative)
... is not in ... º … not in … (non-associative)
... is present º ... is not null (non-associative)
... is not present º ... 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 time of day (non-associative)
... is not time of day (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)
... is object (non-associative)
... is not object (non-associative)
... is fuzzy (non-associative)
... is not fuzzy (non-associative)
... is crisp (non-associative)
... is not crisp (non-associative)
... is linguistic variable (non-associative)

... is not linguistic variable (non-associative)
... is <object-name> (non-associative)
... is not <object-name> (non-associative)

... || ... (left-associative)

... formatted with ... (non- associative)

uppercase … (right associative)
lowercase … (right associative)
trim … (right associative)
trim left … (right associative)
trim right … (right associative)
substring … characters from … (right associative)
substring … characters from … starting at … (right associative)
localized … by … (right associative)

localized … (non-associative)

+ ... (non-associative)
- ... (non-associative)

... + ... (left-associative)
... - ... (left-associative)

... * ... (left associative)
... / ... (left associative)

... ** ... (non-associative)

... before ... (non-associative)
... after ... º … from … (non-associative)

... ago (non-associative)

... year º ... years (non-associative)
... month º ... months (non-associative)
... week º ... weeks (non-associative)
... day º ... days (non-associative)
... hour º ... hours (non-associative)
... minute º ... minutes (non-associative)
... second º ... seconds (non-associative)
… matches pattern … (non-associative)

find … [in] … (right-associative)
find … [in] … starting at … (right-associative)

count [of] ... (right associative)
exist [of] ... (right associative)
avg [of] ... º 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 º minimum ... from ... (right associative)
min [of] ... º minimum [of] ... (right associative)
min ... from … using º minimum ... from ... using … (right-associative)
min [of] ... using º minimum [of] ... using … (right-associative)
max ... from ... º maximum ... from ... (right associative)
max [of] ... º maximum [of] ... (right associative)
max ... from ... using …º maximum ... from ... using … (right-associative)
max [of] ... using … º maximum [of] ... using (right-associative)
index min ... from º index minimum ... from ... (right associative)
index min [of] ... º index minimum [of] ... (right associative)
index max ... from ... º index maximum ... from ... (right associative)
index max [of] ... º 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 ... from ... using (right-associative)
sublist ... elements from ... (right-associative)
sublist ... elements starting at … from ... (right-associative)
latest [of] ... (right associative)
latest [of] ... using …(right-associative)
earliest ... from ... (right associative)
earliest [of] ... (right associative)
earliest ... from ... using … (right-associative)
earliest [of] ... using …(right-associative)
nearest ... from ... (right associative)
index nearest ... from ... (right associative)
index of ... within ... (right-associative)
at least ... from ... (right-associative)
at most ... from ... (right-associative)
increase [of] ... (right associative)
decrease [of] ... (right associative)
percent increase [of] ... º % increase [of] ... (right associative)
percent decrease [of] ... º % decrease [of] ... (right associative)
interval [of] ... (right associative)
time [of] ... (right associative)
applicability [of] ... (right associative)
defuzzified ... (right associative)
time of day [of] ... (right associative)
day of week [of] ... (right associative)
arccos [of] ... (right associative)
arcsin [of] ... (right associative)
arctan [of] ... (right associative)
cos [of] ... º cosine [of] ... (right associative)
sin [of] ... º sine [of] ... (right associative)
tan [of] ... º tangent [of] ... (right associative)
exp [of] ... (right associative)
floor [of] ... (right associative)
ceiling [of] ... (right associative)

truncate [of] ... (right associative)
round [of] ... (right associative)
log [of] ... (right associative)
log10 [of] ... (right associative)
int [of] ... (right associative)
abs [of] ... (right associative)
sqrt [of] ... (right associative)
extract year [of] ... (right associative)
extract month [of] ... (right associative)
extract day [of] ... (right associative)
extract hour [of] ... (right associative)
extract minute [of] ... (right associative)
extract second [of] ... (right associative)
replace year [of] ... with … (right-associative)
replace month [of] ... with … (right-associative)
replace day [of] ... with … (right-associative)
replace hour [of] ... with … (right-associative)
replace minute [of] ... with … (right-associative)
replace second [of] ... with …(right-associative)
reverse [of] ... (right associative)
extract characters [of] ... (right associate)
string [of] ... (right associative)
length [of] … (right associative)
… . … (right associative)
attribute … from … (right associative)
extract attribute names … (right associative)
clone … (right associative)

… seqto … (non-associative)

… as number (non-associative)
… as time (non-associative)
… as string (non-associative)
… as truth value (non-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 (–).
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.
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. The order and format of the fields are implementation specific. 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.

A6 OBJECTS IN ARDEN SYNTAX

A6.1 Rationale

Objects were introduced in Arden 2.5. These have been added as an enhancement to the Arden Syntax to address user and vendor concerns about Arden limitations, and to dramatically increase the capabilities of the syntax. This is an evolutionary step toward full support for receiving data in HL7 V3 messages.

Arden Syntax was originally designed to be very simple, and was limited to a single method of combining data: the ordered list. To simplify list handling, and avoid complexities of things like lists containing lists, the syntax specifies that lists contain only individual items. This has some significant limitations.

Repository data, which typically is represented logically as tables, is returned by the READ statement as a set of columnar lists which are then assigned to separately named variables. After a READ, it can be difficult to maintain the links between values which are naturally associated with each other. For example, the last item of **firstnames** and the last item of **lastnames** may correspond, but what happens if a new item is added to one of the lists, or one of the lists is reordered? The correspondence is lost.

As MLMs evolve, they typically gain features, size and complexity via successive refinement. Declaring new variables for every temporary computation in an MLM clutters the MLM name space with names which often have little meaning. MLM authors tend to start using lists as ad-hoc data structures, where the first, second etc. items, rather than representing multiple instances of a piece of data, instead represent several different types of data which are united by a common relationship. In most computer languages these would be stored in a specialized data structure, with a declared name for each item. Items in a list can only be referred to via their index (a number) which is not easy to read or understand.

Structured data is actually a simplifying concept. Introducing structured data types, while allowing complex structures, tends to make any given usage simpler because of the ability to declare names and relationships. The addition of Objects to the 2.5 standard acknowledges this, and this enhances the Arden Syntax in a number of ways:

- Database queries can be returned as a list of rows, each of which contains named attributes and values.
- Object domain models, such as the HL7 models, may be adapted to Arden and referenced in a natural way as objects by MLMs.
- Complex data structures, as needed, may be created and manipulated easily. Object attributes can contain lists or other object instances, allowing arbitrary depth.

A design goal, in incorporating objects into Arden, was full backward compatibility, and to introduce as few reserved words and as little new syntax as possible. New reserved words cause a compatibility problem because existing MLMs may use those reserved words as variable names. We have only added two more reserved words. Syntax changes are summarized:

- New reserved words: **new**, **object**
- New syntax (special characters): dot (**.**) for object reference, square braces (**[]**) to denote attributes in an object declaration statement.
- New operators: **dot** (**.**) operator, **is object**, **is not object** , **read as**

A6.2 Object Details

The term **object** is used in the domain model sense, rather than as a programming language artifact. In Arden an object is a structured data type, which has a name, and an ordered collection of attributes. Each of these attributes may refer to any valid Arden data item, or be null. Each of these data items may have a primary time associated with it, but the object itself does not have a primary time independent of its attributes. For convenience, if all attributes of an object share a common primary time, the **time of** operator will return that time when applied to the object.

A6.3 Object Identity

Objects in Arden have an identity, which is preserved when assigned or used as an argument to an operator, added to lists or extracted from lists. Objects only are created when the **new** statement is called (from either the data slot or logic slot) or via the **read as** statement in the data slot. An MLM may also reference objects which are passed as arguments or returned from calls to other MLMs. Object identity is not maintained when passed as an argument to an MLM call or a foreign interface, or returned from MLMs. That is, an object is always copied when passed to or returned from an MLM (objects are passed by value to other MLMs, not by reference).

Objects allow an MLM to create structured data, store it in a list, modify it while it still exists in the list, and later reference it as part of the list. While this may sound complicated, it is an important feature. It allows Arden syntax to remain fairly simple while still allowing the easy reference and manipulation of query results.

```
// Assume a list of order objects, with attributes including status and
// message.
// This MLM wants to set the message based on the status.
For order_obj in order_obj_list do
   if order_obj.status = "Cancel" then
       order_obj.message = "This order has been cancelled.";
   elseif order_obj.status = "Modify" then
       order_obj.message = "This order has been modified.";
   elseif order_obj.status = "Suspend" then
       order_obj.message = "This order has been suspended.";
   endif;
enddo;
```

This code only works correctly because of object identity. The order_obj in the loop corresponds to the order referenced in the list of order objects. Without object identity it would not be possible to do this type of manipulation on lists of items.

At this time it is not possible to determine in Arden if two variables refer to the same object. That is, the equality operator is not defined for objects, and there is no substitute method defined. This may be a something to add in a future version.

A6.4 Objects In Expressions

If an object is passed to a standard Arden operator (equality operator, addition, etc) which does not explicitly define behavior with objects, the result of the operation will be null. To effectively use an object as an argument to these standard operators, you must reference a particular field within the object (using the **dot** operator) so that the resulting type is not an object.

A6.4 Creating Objects

The **new** statement can be used to create object instances, with all attributes initialized to null. Using attribute assignment statements (Section 10.2.1.1) it is possible to set these fields explicitly after creating the object. Sometimes however it is preferable to create an MLM which acts as a constructor, to create an object and initialize attributes to the desired default values. Any time one of these objects needs to be created, that MLM can be called. Here is an example of using an MLM as a constructor:

```
Create_field_mlm := MLM 'create_form_field';
Form_field := Call Create_field_mlm with name, value, status;

/* MLM 'create_form_field' segment */
Data:
    form_field_type :=
       Object [name, value, status];
    field := new form_field_type;
    field.name  := argument 1;
    field.value := argument 2;
    field.status := argument 3;;

Evoke: /* called directly */ ;;
Logic:  conclude true;;
Action: return field;;
```

Appendices

(Nonmandatory Information)

X1 STRUCTURED WRITE STATEMENT SUGGESTED SCHEMA

X1.1 Structured Message

The <structured.message> should be parsed and interpreted as an XML document. This message is distinguished from other coded messages by the first line:

```
<?xml version="1.0"?>
```

X1.2 Usage Notes

Structured messages are created by concatenating string literals and variables in the same manner as other strings. This means that if double quotes (") are to be used within the XML message, they must be "escaped" by using two double quotes (""). Likewise if XML reserved symbols (e.g., <, >) are desired within the structured write statement, CDATA escape tokens must be placed at the beginning and end of the element.

X1.3 Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xs:element name="structured.message">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="body" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="subject" type="xs:string" minOccurs="0"/>
       <xs:element name="context" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element name="conclusion" type="xs:string"/>
       <xs:element name="recommendation" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="instruction" type="xs:string"/>
          <xs:element name="choice.list" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="choice" maxOccurs="unbounded">
              <xs:complexType>
               <xs:simpleContent>
                <xs:extension base="xs:string">
                 <xs:attribute name="id" type="xs:string"/>
                </xs:extension>
               </xs:simpleContent>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="type">
             <xs:simpleType>
              <xs:restriction base="xs:NMTOKEN">
```

```
            <xs:enumeration value="at.least"/>
            <xs:enumeration value="at.most"/>
            <xs:enumeration value="exactly"/>
           </xs:restriction>
          </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="number" type="xs:string"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="citation" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:simpleContent>
       <xs:extension base="xs:string">
        <xs:attribute name="position">
         <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
           <xs:enumeration value="support"/>
           <xs:enumeration value="refute"/>
          </xs:restriction>
         </xs:simpleType>
        </xs:attribute>
       </xs:extension>
      </xs:simpleContent>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="distribution.list" minOccurs="0">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="distribution" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="recipient" maxOccurs="unbounded">
        <xs:complexType>
         <xs:simpleContent>
          <xs:extension base="xs:string">
           <xs:attribute name="role" type="xs:string"/>
          </xs:extension>
         </xs:simpleContent>
        </xs:complexType>
       </xs:element>
       <xs:element name="workflow" minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="closure.select" type="xs:boolean" minOccurs="0"/>
          <xs:element name="forwarding.select" type="xs:boolean" minOccurs="0"/>
          <xs:element name="coverage.select" type="xs:boolean" minOccurs="0"/>
          <xs:element name="timers.select" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="timeout" maxOccurs="unbounded">
```

```
                <xs:complexType>
                 <xs:simpleContent>
                  <xs:extension base="xs:string">
                   <xs:attribute name="type">
                    <xs:simpleType>
                     <xs:restriction base="xs:NMTOKEN">
                      <xs:enumeration value="submission.ack"/>
                      <xs:enumeration value="delivery.ack"/>
                      <xs:enumeration value="display.ack"/>
                      <xs:enumeration value="closure.ack"/>
                     </xs:restriction>
                    </xs:simpleType>
                   </xs:attribute>
                  </xs:extension>
                 </xs:simpleContent>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

X1.4 Description of elements and attributes

The elements and attributes utilized in the <structured.message> are described below:

X1.4.1 <structured.message>

This element is the root element of the XML-based structured MLM output message. The element represents all the information that can be defined by the MLM author in the structured message. The element contains one or more <body> sub-elements, followed by an optional <distribution.list> sub-element.

X1.4.2 <body>

This element encompasses the main textual portion of the MLM message output. The element contains an optional <subject> sub-element, followed by zero or more <context> sub-elements, followed by a <conclusion> sub-element, followed by zero or more <recommendation> sub-elements, followed by zero or more <citation> sub-elements.

X1.4.3 <subject>

This element is the subject of the MLM message output. The element indicates the nature of the MLM output (e.g. "Panic Lab Result".) The element contains character data.

X1.4.4 <context>

This element is contextual information relevant to the MLM conclusion or the associated patient, including previous lab results, existing allergies, etc. The element contains character data.

X1.4.5 <conclusion>

This element is the main conclusion statement of the MLM after the condition specified in the MLM logic slot is evaluated as true (e.g. "Glucose level has significantly dropped".) The element contains character data.

X1.4.6 <recommendation>

This element encompasses the MLM author's recommended response to the detected condition in the MLM logic slot. The element contains an <instruction> sub-element, followed by an optional <choice.list> sub-element.

X1.4.7 <instruction>

This element is a recommended action item possibly with instruction to select one of the subsequent options (e.g. "Select a treatment".) The element contains character data.

X1.4.8 <choice.list>

This element encompasses the selectable options appropriate for the instruction. The element contains one or more <choice> sub-elements. The element has two attributes named type and number. These attributes allow the MLM author to indicate the nature and number of the selections of the subsequent option choices. The value of the type attribute must be the string "at.least", "at.most", or "exactly." The value of the number attribute can be any string of characters (excluding reserved characters "<", ">", "&"—unless the XML CDATA escape notation is used); it ishowever is expected to be a natural number. If the attribute values are not supplied, interpretation of these values is left to the discretion of the message consumer.

X1.4.9 <choice>

This element is an action option. The element contains character data. The element typically identifies a single option (e.g. "50% dextrose intravenous"), but can also be utilized to identify an aggregation of options (e.g. "All of the above") or no options (e.g. "None of the above".) The element has an attribute named id. The attribute is a unique identifier assigned to the action option that can be subsequently used to reference a selected choice. The value of the attribute can be any string of characters (excluding reserved characters "<", ">", "&"—unless XML CDATA escape notation is used). If the id value is not supplied, interpretation of the value is left to the discretion of the message consumer.

X1.4.10 <citation>

This element is reference information for the algorithm provided in the MLM logic slot as described in Section 6.2.4. The element contains character data. The element has an attribute named position. The value of the attribute must be either the string "support" (indicating a citation that verifies the algorithm in the logic slot) or "refute" (indicating a citation that refutes the algorithm in the logic slot.) If the position value is not supplied, interpretation of the value is left to the discretion of the message consumer.

X1.4.11 <distribution.list>

This element consists of the MLM author preferences for disseminating the message. Multiple distributions can be utilized concurrently, each with its own recipient list and distribution workflow. The element contains one or more <distribution> sub-elements.

X1.4.12 <distribution>

This element consists of information about the dissemination of the message. The distribution includes the message recipients and the associated workflow that is to be used in disseminating the message to these recipients. The element contains one or more <recipient> sub-elements, followed by an optional <workflow> sub-element.

X1.4.13 <recipient>

This element identifies a message recipient. The element contains character data. The element has an attribute named type. The value of the attribute must be the string "person", "role", "group", or "unspecified." "Person" designates that the specified recipient is a person. "Group" designates that the specified recipient is a group. This group must be subsequently resolved into its individual member persons. "Role" designates that the specified recipient is a role. This role must be subsequently resolved to determine which of the persons provisioned in the role is "filling" the role. "Unspecified" designates that the appropriate message recipient is unknown to the MLM author. In this case the contents of this element should be ignored. The dynamic nature of patient-provider relationships and coverage schedules in medical institutions may prevent the MLM author from specifying the recipient. "Unspecified" indicates that the appropriate recipient will be subsequently identified by an external system with knowledge of appropriate patient coverage. In the absence of a value for the type attribute the default value of "person" is used.

X1.4.14 <workflow>

This element consists of the MLM author specified workflow to be employed in delivering the message to the recipients of the distribution, including the various acknowledgment timers associated with the distribution. The element contains an optional <closure.select> sub-element, followed by an optional <forwarding.select> sub-element, followed by an optional <coverage.select> sub-element, followed by an optional <timers.select> sub-element.

X1.4.15 <closure.select>

Closure is the indication of completion of the workflow by the recipient(s) associated with the message delivery. This element does not contain anything, however the element has an attribute named required. The attribute specifies that the MLM author requires closure for all deliveries originating from this distribution. The value of the attribute must be either the string "true" or "false." In the absence of a value for the required attribute the default value of "true" is used.

X1.4.16 <forwarding.select>

Forwarding is the redirection of a message delivery to another recipient. This element does not contain anything, however the element has an attribute named enabled. The attribute specifies that the MLM author allows deliveries originating from this distribution to be forwarded to other recipients. The value of the attribute must be either the string "true" or "false." In the absence of a value for the enabled attribute the default value of "true" is used.

X1.4.17 <coverage.select>

Coverage indicates that a message will be delivered to an alternate recipient if the specified distribution recipient is not available. This element does not contain anything, however the element has an attribute named enabled. The attribute specifies that the MLM author requires alternate recipients be identified if the distribution recipients are unavailable. The value of the attribute must be either the string "true" or "false." In the absence of a value for the enabled attribute the default value of "true" is used.

X1.4.18 <timers.select>

This element specifies the timeout values utilized in the management of the delivery process. The element contains one or more <timeout> sub-elements.

X1.4.19 <timeout>

This element is the time utilized as a timeout for an aspect of the delivery process. The element contains character data. The element has an attribute named type. The attribute indicates the timeout type. The value of the attribute must be the string "submission.ack" (acknowledgement of message submission from the delivery network), "delivery.ack" (acknowledgement of message delivery from the delivery device), "display.ack" (acknowledgement of message presentation from the delivery device) or "closure.ack" (acknowledgement of workflow completion.) If the type value is not supplied, interpretation of the value is left to the discretion of the message consumer.

X1.5 Example

```
<?xml version="1.0"?>
<!DOCTYPE structured.message SYSTEM " StructuredMessage.dtd">
<structured.message>
    <body>
       <subject>Critical Lab Result</subject>
       <context>Mary Smith had a blood sugar of 120 two weeks ago.</context>
       <conclusion>Mary Smith's blood sugar is critical at 435 mg/dl.
    </conclusion>
       <citation>See Pharmacy and Therapeutics committee standard criteria for
    abnormal lab studies.</citation>
    </body>
    <distribution.list>
       <distribution>
           <recipient role="personal physician">Dr. John Jones</recipient>
           <workflow>
               <closure.select required="true"/>
               <forwarding.select enabled="true"/>
               <coverage.select enabled="false"/>
               <timers.select>
                   <timeout type="closure.ack"/>
               </timers.select>
           </workflow>
       </distribution>
    </distribution.list>
</structured.message>
```

X2 XML SCHEMA FOR MLMS

The following sections detail a sample schema that may be used to represent MLMs in XML. Later version of Arden Syntax may include alternate, non-textual representations of medical logic modules as part of the normative standard. This informative appendix illustrates one possible coded form.

The XML schema included in this appendix represents a high-level decomposition of a textual MLM into XML elements that the Arden Syntax SIG could justify as being useful for MLM management functions such as indexing, searching, and retrieval of specific MLMs from a knowledge base library in XML-centric information systems or databases.. Although some Arden Syntax SIG members argued for greater detail, the consensus of the SIG was that additional decomposition could not be justified within the scope of the current Arden Syntax.

X2.1 Graphic Representation of Schema

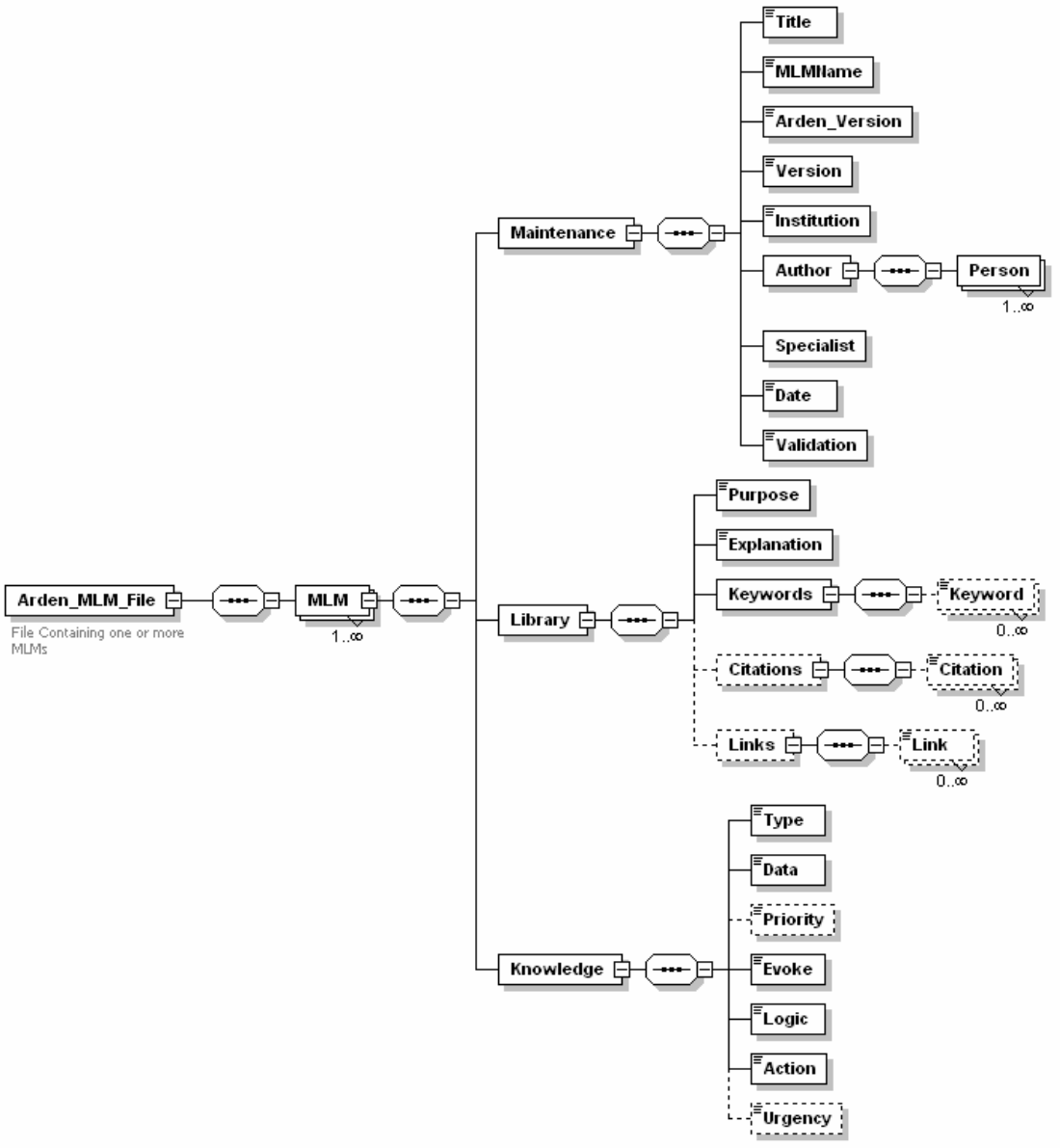


Figure X2.1 Graphic Representation of XML Schema for Arden Syntax MLMs

X2.2 Textual Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified">
 <xs:complexType name="Citation">
  <xs:sequence>
   <xs:element name="Citation_Number" type="xs:positiveInteger" minOccurs="0"/>
   <xs:element name="Citation_Type" minOccurs="0">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="support"/>
      <xs:enumeration value="refute"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="Citation_Text" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Link">
  <xs:sequence>
   <xs:element name="Link_Type" type="xs:string" minOccurs="0"/>
   <xs:element name="Link_Description" type="xs:string" minOccurs="0"/>
   <xs:element name="Link_Text" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="Arden_MLM_File">
  <xs:annotation>
   <xs:documentation>File Containing one or more MLMs</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="MLM" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Maintenance">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Title">
           <xs:simpleType>
            <xs:restriction base="xs:string">
             <xs:whiteSpace value="preserve"/>
            </xs:restriction>
           </xs:simpleType>
          </xs:element>
          <xs:element name="MLMName">
           <xs:simpleType>
            <xs:restriction base="xs:string">
             <xs:minLength value="1"/>
             <xs:maxLength value="80"/>
             <xs:whiteSpace value="collapse"/>
            </xs:restriction>
           </xs:simpleType>
          </xs:element>
          <xs:element name="Arden_Version">
           <xs:simpleType>
```

```
      <xs:restriction base="xs:string">
       <xs:enumeration value="Version 2.0"/>
       <xs:enumeration value="Version 2.1"/>
       <xs:enumeration value="Version 2.5"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Version">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:maxLength value="80"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Institution">
     <xs:simpleType>
      <xs:restriction base="xs:string">
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Author">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Person" type="Person" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Specialist" type="Person"/>
    <xs:element name="Date">
     <xs:simpleType>
      <xs:union memberTypes="xs:dateTime xs:date"/>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Validation">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:enumeration value="Production"/>
       <xs:enumeration value="Research"/>
       <xs:enumeration value="Testing"/>
       <xs:enumeration value="Expired"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="Library">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Purpose">
     <xs:simpleType>
      <xs:restriction base="xs:string">
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Explanation">
```

```
      <xs:simpleType>
       <xs:restriction base="xs:string">
       </xs:restriction>
      </xs:simpleType>
     </xs:element>
     <xs:element name="Keywords">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Keyword" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="Citations" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Citation" type="Citation" minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="Links" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Link" type="Link" minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="Knowledge">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Type">
      <xs:simpleType>
       <xs:restriction base="xs:string">
        <xs:pattern value=""/>
        <xs:enumeration value="data-driven"/>
        <xs:enumeration value="data_driven"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:element>
     <xs:element name="Data" type="xs:string"/>
     <xs:element name="Priority" default="50" minOccurs="0">
      <xs:simpleType>
       <xs:restriction base="xs:decimal">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="99"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:element>
     <xs:element name="Evoke" type="xs:string"/>
     <xs:element name="Logic" type="xs:string"/>
     <xs:element name="Action" type="xs:string"/>
     <xs:element name="Urgency" default="50" minOccurs="0">
      <xs:simpleType>
       <xs:restriction base="xs:decimal">
```

```
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="99"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 </xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType name="Person">
 <xs:annotation>
  <xs:documentation>Name and degree of the person with optional email address -- Use / derive from standard RIM
-derived XML schema for person.  Must include e-mail address.</xs:documentation>
 </xs:annotation>
</xs:complexType>
</xs:schema>
```

X2.3 Description of Elements

X2.3.1 element **Arden_MLM_File**

children **MLM**

annotation documentation File Containing one or more MLMs

The <Arden_MLM_File> element is the root element. The <Arden_MLM_File> element may contain one or more of element <MLM>, each of which would be a specific Arden MLM

X2.3.2 element **Arden_MLM_File/MLM**

children **Maintenance Library Knowledge**

Each <MLM> element would contain a single sequence of elements namely one <Maintenance> containing elements describing slots in maintenance category, one <Library> containing elements describing slots in the library category, and one <Knowledge> containing elements describing slots in the Knowledge category of the MLM.

*X2.3.2.1 element **Arden_MLM_File/MLM/Maintenance***

children **Title MLMName Arden_Version Version Institution Author Specialist Date Validation**

The <Maintenance> element contains a definite sequence of elements defining the sequence of slots of Maintenance category of an Arden MLM. The <Maintenance> element contains a definite sequence of elements as described in the contents below.

X2.3.2.1.1 element **Arden_MLM_File/MLM/Maintenance/Title**

type restriction of **xs:string**

facets whiteSpace preserve

pattern

The <Title> element corresponds to the Title slot of an Arden MLM and describes the title of the MLM. <Title> is a required element. CDATA tags may need to be placed around the text within the Title element if XML-reserved symbols (e.g., < >) are used within the text.

X2.3.2.2.2element **Arden_MLM_File/MLM/Maintenance/MLMName**

type restriction of **xs:string**

facets minLength 1

maxLength 80

whiteSpace collapse

pattern

The <MLMName> is an element corresponding to MLMName slot of an Arden MLM. It contains the Name of the MLM and has similar characteristics as defined for a MLMName slot, i.e. length between 1 and 80 characters and without any white spaces. <MLMName> is a required element.

X2.3.2.2.3 element **Arden_MLM_File/MLM/Maintenance/Arden_Version**

type restriction of **xs:string**

facets pattern

enumeration Version 2.0

enumeration Version 2.1

enumeration Version 2.5

The <Arden_Version> element corresponds to the 'Arden Version' slot of an Arden MLM. <Arden_Version> gives the version of Arden Syntax Standard in which that particular MLM is written. <Arden_Version> is a required element.

X2.3.2.2.4 element **Arden_MLM_File/MLM/Maintenance/Version**

type restriction of **xs:string**

facets maxLength 80

pattern

The <Version> element corresponds to the 'Version' slot of an Arden MLM. <Version> element tells the version of the particular Arden MLM. It is suggested that version should start at 1.0, increasing by 0.1 for minor revision and by 1.0 for major revision. <Version> is a required element.

X2.3.2.2.5 element **Arden_MLM_File/MLM/Maintenance/Institution**

type restriction of **xs:string**

facets pattern

The <Institution> element corresponds to 'Institution' slot of an Arden MLM. <Institution> contains name of the institution in which the MLM is being used. <Institution> is a required element.

*X2.3.2.2.6 element **Arden_MLM_File/MLM/Maintenance/Author***

children **Person**

The <Author> element corresponds to 'Author' slot of an Arden MLM. <Author> contains one or more <Person> elements that enumerate the name(s) of the MLM authors. <Author> is a required element.

X2.3.2.2.6.1 element **Arden_MLM_File/MLM/Maintenance/Author/Person**

type **Person**

The <Person> element contains the name of one MLM author. <Person> is a required element.

*X2.3.2.2.7 element **Arden_MLM_File/MLM/Maintenance/Specialist***

type **Person**

The <Specialist> element corresponds to 'Specialist' slot of an Arden MLM. <Specialist> contains information about the person responsible for maintenance and implementation of the Arden MLM in a particular element. <Specialist> is a required element

*X2.3.2.2.8 element **Arden_MLM_File/MLM/Maintenance/Date***

type union of (**xs:dateTime**, **xs:date**)

The <Date> element corresponds to 'Date' slot of an Arden MLM. <Date> contains the date on which the MLM was last modified. <Date> is a required element

*X2.3.2.2.9 element **Arden_MLM_File/MLM/Maintenance/Validation***

type restriction of **xs:string**

facets pattern

enumeration Production

enumeration Research

enumeration Testing

enumeration Expired

The <Validation> element corresponds to 'Validation' slot of an Arden MLM. <Validation> contains the string indicating validation status of the Arden MLM. <Validation> is a required element.

X2.3.3.3 element **Arden_MLM_File/MLM/Library**

children **Purpose Explanation Keywords Citations Links**

The <Library> element corresponds to the <Library> category of an Arden MLM. <Library> contains elements pertaining to slots in a library category in a particular sequence as described in the contents below. <Library> is a required element.

*X2.3.3.3.1 element **Arden_MLM_File/MLM/Library/Purpose***

type restriction of **xs:string**

facets pattern

The <Purpose> element corresponds to the 'Purpose' slot of an Arden MLM. <Purpose> contains information as to the purpose of the Arden MLM. <Purpose> is a required element.

*X2.3.3.3.2 element **Arden_MLM_File/MLM/Library/Explanation***

type restriction of **xs:string**

facets pattern

The <Explanation> element corresponds to the 'Explanation' slot of an Arden MLM. <Explanation> contains the explanation as to the logic of the Arden MLM. The explanation can be used to show user as to why the MLM came to a decision. <Explanation> is a required element.

*X2.3.3.3.3 element **Arden_MLM_File/MLM/Library/Keywords***

Children **Keyword**

The <Keywords> element corresponds to the 'Keywords' slot of an Arden MLM. <Keywords> contains zero or more <Keyword> elements each representing a single keyword. <Keywords> is a required element.

X2.3.3.3.3.1 element **Arden_MLM_File/MLM/Library/Keywords/Keyword**

Type **xs:string**

Each <Keyword> element corresponds to one of the keywords from the semi-colon delimited list in a textual MLM.

*X2.3.3.3.4 element **Arden_MLM_File/MLM/Library/Citations***

children **Citation**

The <Citations> element corresponds to the 'Citations' slot of an Arden MLM. <Citations> contains zero or more <Citation> elements. <Citations> is an optional element.

X2.3.3.3.4.1 element **Arden_MLM_File/MLM/Library/Citations/Citation**

type **Citation**

children **Citation_Number Citation_Type Citation_Text**

Each <Citation> element contains the information describing a single citation to the literature. The Citation complex type is described below.

*X2.3.3.3.5 element **Arden_MLM_File/MLM/Library/Links***

children **Link**

The <Links> element corresponds to the 'Links' slot of an Arden MLM. <Links> contains zero or more <Link> elements. <Links> is an optional element.

X2.3.3.3.5.1 element **Arden_MLM_File/MLM/Library/Links/Link**

type **Link**

children **Link_Type Link_Description Link_Text**

The <Link> element contains an individual link. The Link complex type is described below.

X2.3.3.4 element **Arden_MLM_File/MLM/Knowledge**

children **Type Data Priority Evoke Logic Action Urgency**

The <Knowledge> element corresponds to the 'Knowledge' slot of an Arden MLM. The <Knowledge> element contains a sequence of elements corresponding to the slots in Knowledge category, as described in the contents below. <Knowledge> is a required element.

*X2.3.3.4.1 element **Arden_MLM_File/MLM/Knowledge/Type***

Type restriction of **xs:string**

Facets pattern

enumeration data-driven

enumeration data_driven

The <Type> element corresponds to the 'Type' slot of an Arden MLM. The <Type> element contains the type indicating the type of Arden MLM. Currently, only one type is defined, i.e. data-driven or data_driven. <Type> is a required element.

*X2.3.3.4.2 element **Arden_MLM_File/MLM/Knowledge/Data***

Type **xs:string**

The <Data> element corresponds to the 'Data' slot of an Arden MLM. The <Data> element contains the data variables used in the Arden MLM and their mapping to the local database of the institution. <Data> is a required element. CDATA tags should be placed around the text in this element to avoid problems with XML reserved characters (e.g. <, >) occurring within the text.

*X2.3.3.4.3 element **Arden_MLM_File/MLM/Knowledge/Priority***

Type restriction of **xs:decimal**

Facets minInclusive 0

maxInclusive 99

The <Priority> element corresponds to the 'Priority' slot of an Arden MLM. The <Priority> element contains a number between 0 to 99 indicating the priority of execution of the MLM with 0 indicating least priority and 99 indicating highest priority. The priority is to be used when more than one MLM is fired by a particular event. <Priority> is an optional element.

*X2.3.3.4.4 element **Arden_MLM_File/MLM/Knowledge/Evoke***

Type **xs:string**

The <Evoke> element corresponds to the 'Evoke' slot of an Arden MLM. The <Evoke> element specifies the condition under which the MLM would be evoked or called. Evoke is a required element. CDATA tags should be placed around the text in this element to avoid problems with XML reserved characters (e.g. <, >) occurring within the text.

*X2.3.3.4.5 element **Arden_MLM_File/MLM/Knowledge/Logic***

Type **xs:string**

The <Logic> element corresponds to the 'Logic' slot of an Arden MLM. The <Logic> element contains main decision making logic of the Arden MLM written using Arden Syntax. It can conclude true or false only. <Logic> is a required element. CDATA tags should be placed around the text in this element to avoid problems with XML reserved characters (e.g. <, >) occurring within the text.

*X2.3.3.4.6 element **Arden_MLM_File/MLM/Knowledge/Action***

Type **xs:string**

The <Action> element corresponds to the 'Action' slot of an Arden MLM. The <Action> element specifies the action to be taken if the Logic concludes true. <Action> is a required element. CDATA tags should be placed around the text in this element to avoid problems with XML reserved characters (e.g. <, >) occurring within the text.

*X2.3.3.4.7 element **Arden_MLM_File/MLM/Knowledge/Urgency***

Type restriction of **xs:decimal**

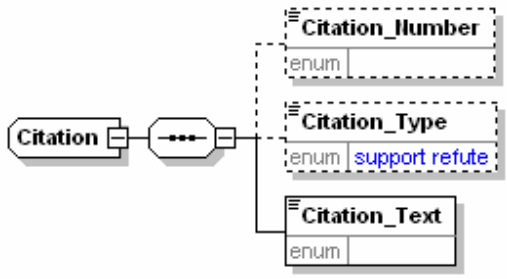
Facets minInclusive 0

maxInclusive 99

The <Urgency> element corresponds to the 'Urgency' slot of an Arden MLM. The <Urgency> element contains the number indicating the urgency of execution of action of the MLM. The number can be between 0 (least urgent) to 99 (highest level urgent). <Urgency> should be used to decide the order of execution of various actions when more than one MLMs, are executed at the same time. <Urgency> is an optional element.

X2.4 Defined Complex Types

X2.4.1 complexType **Citation**



Children **Citation_Number** **Citation_Type** **Citation_Text**

used by element **Arden_MLM_File/MLM/Library/Citations/Citation**

The Citation complexType represents the individual citations as described in 6.2.4

X2.4.1.1 element **Citation/Citation_Number**

Type **xs:positiveInteger**

X2.4.1.2 element **Citation/Citation_Type**

Type restriction of **xs:string**

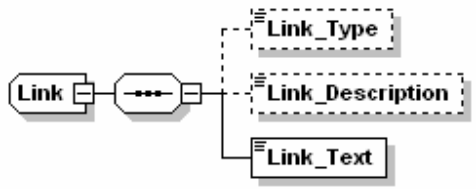
Facets enumeration support

enumeration refute

X2.4.1.3 element **Citation/Citation_Text**

Type **xs:string**

X2.4.2 complexType **Link**



Children **Link_Type** **Link_Description** **Link_Text**

used by element **Arden_MLM_File/MLM/Library/Links/Link**

The Link complexType represents the individual links as described in 6.2.5

X2.4.2.1 element **Link/Link_Type**

Type **xs:string**

X2.4.2.2 element **Link/Link_Description**

Type **xs:string**

X2.4.2.3 element **Link/Link_Text**

Type **xs:string**

X2.4.3 complexType **Person**

used by elements **Arden_MLM_File/MLM/Maintenance/Author/Person** **Arden_MLM_File/MLM/Maintenance/Specialist**

Annotation documentation Name and degree of the person with optional email address -- Use / derive from standard RIM -derived XML schema for person. Must include e-mail address.

The Person complex type is a stub definition to the XML ITS representation of the Entity>LivingSubject>Person class from the HL7 V3 RIM.

X2.5 Example MLM

```
<Arden_MLM_File xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Arden Syntax 2.5 Schema 20040804.xsd">
  <MLM>
    <Maintenance>
      <Title>Dosing for gentamicin in renal failure</Title>
      <MLMName>gentamicin_dosing</MLMName>
      <Arden_Version>Version 2.5</Arden_Version>
      <Version>1.0</Version>
      <Institution>Columbia-Presbyterian Medical Center</Institution>
      <Author>
        <Person>
          <Entity.Attrs>
            <name>
              </delimiter>
              <family>Hripcsak</family>
              <given>George</given>
              <prefix/>
              <suffix/>
            </name>
            <telecom value="mailto:george.hripcsak@columbia.edu"
          </Entity.Attrs>
          <Person.Attrs>
            <educationLevelCode>MD</educationLevelCode>
          </Person.Attrs>
        </Person>
      </Author>
      <Specialist/>
      <Date>1991-03-18</Date>
      <Validation>Testing</Validation>
    </Maintenance>
    <Library>
      <Purpose>
        Suggest an appropriate gentamicin dose in the setting of renal
        insufficiency. (This MLM demonstrates a management suggestion.)
      </Purpose>
      <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.
      </Explanation>
      <Keywords>
        <Keyword>gentamicin</Keyword>
        <Keyword>dosing</Keyword>
      </Keywords>
    </Library>
    <Knowledge>
      <Type>data-driven</Type>
      <Data>
        <![CDATA[
        /* 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 ;
        ]]>
      </Data>
      <Priority>50</Priority>
      <Evoke>gentamicin_order;</Evoke>
      <Logic>
        <![CDATA[
        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 ;
        ]]>
      </Logic>
      <Action>
        <![CDATA[
        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."
        ]]>
      </Action>
      <Urgency>50</Urgency>
    </Knowledge>
  </MLM>
</Arden_MLM_File>
```

X3 LANGUAGE AND COUNTRY CODES FOR HL7 INTERNATIONAL AFFILIATE COUNTRIES

X3.1

This appendix lists language and country codes as defined by ISO 639.1 and ISO 3166 for countries with HL7 Affiliates. Languages and country codes are arranged in alphabetic order by their English-language name. For additional language and country codes consult the appropriate ISO language / country registrars via ISO (www.iso.ch).

X3.2 Language codes

Language	Code	Language	Code
Assamese	as	Kannada	kn
Basque	eu	Kashmiri	ks
Bengali	bn	Korean	ko
Catalan; Valencian	ca	Kurdish	ku
Chinese	zh	Malayalam	ml
Croatian	hr	Maori	mi
Czech	cs	Marathi	mr
Danish	da	Oriya	or
Dutch; Flemish	nl	Portuguese	pt
English	en	Punjabi; Panjabi	pa
Faroese	fo	Russian	ru
Finnish	fi	Sanskrit	sa
French	fr	Sindhi	sd
Gaelic; Scottish Gaelic	gd	Slovak	sk
Galician	gl	Slovenian	sl
German	de	Spanish; Castilian	es
Greek, Modern (1453-)	el	Swedish	sv
Greenlandic; Kalaallisut	kl	Tamil	ta
Gujarati	gu	Telugu	te
Hindi	hi	Turkish	tr
Irish	ga	Urdu	ur
Italian	it	Welsh	cy
Japanese	ja		

X3.3 Country codes

Country	Code	Country	Code
Argentina	Ar	Italy	It
Australia	Au	Korea, Republic Of	Kr
Brazil	Br	Mexico	Mx
China	Cn	Netherlands	Nl
Croatia (Local Name: Hrvatska)	Hr	New Zealand	Nz
Czech Republic	Cz	Spain	Es
Denmark	Dk	Sweden	Se
Finland	Fi	Switzerland	Ch
France	Fr	Taiwan	Tw
Germany	De	Turkey	Tr
Greece	Gr	United Kingdom	Gb
India	In	United States	Us
Ireland	Ie		

X4 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.

X4.1 Data Interpretation MLM

```
maintenance:
    title: Fractional excretion of sodium;;
    mlmname: fractional_na;;
    arden: Version 2;;
    version: 1.00;;
    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-driven;;
    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 they are not null}
                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 frational 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 diurectic use or pre-existing chronic renal " ||
                      "disease, and up to 10 % of cases of uncomplicated " ||
                      "volume depletion.";
            endif;
            ;;
end:
```

X4.2 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 necessarily 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:
```

X4.3 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 allergy to penicillin documented:"
            || penicillin_allergy;;
    urgency: 50;;
end:
```

X4.4 Management Suggestion MLM

```
maintenance:
    title: Dosing for gentamicin in renal failure;;
    mlmname: gentamicin_dosing;;
    arden: Version 2.1;;
    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:
```

X4.5 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;
           endif;
           ;;
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:
```

X4.6 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 Agranulocytosis 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:
```

X4.7 MLM Translated from CARE

```
maintenance:
    title: Cardiology MLM from CARE, p. 85;;
    mlmname: care_cardiology_mlm;;
    arden: Version 2;;
    version: 1.00;;
    institution: Regenstrief Institute;;
    author: Clement J. McDonald, M.D.; George Hripcsak, M.D.;;
    specialist: ;;
    data: 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
            propanolol or less than 200 mg of metoprolol, 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, Gillam PM. Assessment of propranolol in angina
               pectoris: clinical dose response curve and effect on
               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;;
    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 = 'metoprolol'
                    and
                     the last beta_doses <= 200) then
                    let low_dose_beta_use 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.";
                   conclude true;
               else
                   if premature_beat_frequency is greater than 5 then
                       let message be
                           "Increased dose of beta blockers may "||
                           "be needed to control PVC's.";
                       conclude true;
                   endif;
               endif;
           endif;
           conclude false;
           ;;
action:
       write message;;
end:
```

X4.8 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:
```

X4.9 MLM Fever Calculation - Crisp

```
maintenance:
  title:        Increased body temperature - crisp;;
  mlmname:      increased_body_temperature_crisp;;
  arden:        version 2.9;;
  version:      ;;
  institution:  ;;
  author:       ;;
  specialist:   ;;
  date:         2011-07-06;;
  validation:   testing;;
library:
  purpose:      detects an increased body temperatur over a day - absolute criterion;;
  explanation:  Check if maximum of body temperature is increased with
                crisp logic.
                reads parameter: "TempMax" (in degree Celsius).
  ;;
  keywords:     body temperature, temperature, data to symbol conversion;;
  citations:    ;;
knowledge:
  type:         data_driven;;
  data: ////////////////////////////////////
    readParam            := interface   {read param};       // read single parameter
  ;;
  evoke:    ;;
  logic:  ////////////////////////////////////

    // read precondition from host
    paramTempMax              := call readParam with "TempMax";

    // calculation of result
    if paramTempMax is present then

      if     paramTempMax >= 38   then
        tempratureIncreased               := 1;
        time tempratureIncreased          := time paramTempMax;
      else
        tempratureIncreased               := 0;
        time tempratureIncreased          := time paramTempMax;
      endif;
    endif;
    conclude true;
  ;;

  action: ////////////////////////////////////
    write tempratureIncreased;
  ;;
end:
```

X4.10 MLM Fever Calculation – Fuzzy Simulation

```
maintenance:

  title:        Increased body temperature - fuzzy simulation;;
  mlmname:      increased_body_temperature_fuzzy_simulation;;
  arden:        version 2.9;;
  version:      ;;
  institution:  ;;
  author:       ;;
  specialist:   ;;
  date:         2011-07-06;;
  validation:   testing;;
library:
  purpose:      detects an increased body temperatur over a day - absolute criterion;;
  explanation:  Check if maximum of body temperature is increased with
                explicit coded fuzzy logic.
                reads parameter: "TempMax" (in degree Celsius).
  ;;
  keywords:     body temperature, temperature, data to symbol conversion;;
  citations:    ;;

knowledge:
  type:         data_driven;;
  data: ////////////////////////////////////////

    // interface
    readParam            := interface   {read param};       // read single parameter
  ;;
  evoke:  ;;
  logic:  ////////////////////////////////////

    // read precondition from host
    paramTempMax             := call readParam with "TempMax";

    // calculation of result
    if paramTempMax is present then

      if     paramTempMax >= 38   then
        tempratureIncreased               := 1;
        time tempratureIncreased          := time paramTempMax;
      elseif paramTempMax >  37.5 then
        tempratureIncreased               := (paramTempMax - 37.5) / 0.5;
        time tempratureIncreased          := time paramTempMax;
      else
        tempratureIncreased               := 0;
        time tempratureIncreased          := time paramTempMax;
      endif;

    endif;

    conclude true;
  ;;

  action: ////////////////////////////////////
    write tempratureIncreased;
  ;;
end:
```

X4.11 MLM Fever Calculation – Fuzzy Logic

```
maintenance:

  title:        Increased body temperature - fuzzy;;
  mlmname:      increased_body_temperature_fuzzy;;
  arden:        version 2.9;;
  version:      ;;
  institution:  ;;
  author:       ;;
  specialist:   ;;
  date:         2011-07-06;;
  validation:   testing;;
library:
  purpose:      detects an increased body temperatur over a day - absolute criterion;;
  explanation:  Check if maximum of body temperature is increased with
                fuzzy logic.
                reads parameter: "TempMax" (in degree Celsius).
  ;;
  keywords:     body temperature, temperature, data to symbol conversion;;
  citations:    ;;

knowledge:
  type:         data_driven;;
  data: ////////////////////////////////////////

    // interface
    readParam            := interface   {read param};       // read single parameter
  ;;

  evoke:  ;;
  logic:  ////////////////////////////////////

    // read precondition from host
    paramTempMax              := call readParam with "TempMax";

    // calculation of result
    if paramTempMax is present then

       // <= 37.5 °C: 0; >= 38 °C: 1; inbetween: linear
      tempErh                     := fuzzy set (37.5, 0), (38, 1);
      tempratureIncreased         := paramTempMax is in tempErh;
      time tempratureIncreased    := time paramTempMax;

    endif;

    conclude true;
  ;;

  action: ////////////////////////////////////
    write tempratureIncreased;
  ;;
end:
```

X4.12 MLM for Doses Calculation

```
maintenance:

  title:        Doses Calculation Theophylline - fuzzy;;
  mlmname:      dose_calculation_ theophylline_fuzzy;;
  arden:        version 2.9;;
  version:      ;;
  institution:  ;;
  author:       ;;
  specialist:   ;;
  date:         2012-07-10;;
  validation:   testing;;
library:
  purpose:      calculates the suggested daily doses based on the patients age;;
  explanation:  ;;
  keywords:     ;;
  citations:
    http://library.buffalo.edu/libraries/projects/cases/drug_dosing/drug_dosing_notes.htm
  ;;

knowledge:
  type:         data_driven;;
  data: ////////////////////////////////////////
    patientAge := argument;
  ;;

  evoke:  ;;
  logic:  ////////////////////////////////////////

    AgeGroup := linguistic variable [young, middleAged, old];
    age := new AgeGroup;

    // Age less than 20 years old:
    age.young := fuzzy set (0 years, 1), (19 year, 1), (20 years, 0);
    // Age more than 20 years old and less than 40 years old:
    age.middleAge := fuzzy set (19 years, 0), (20 years, 1), (39 years, 1),
        (40 years, 0);
    // Age greater than 40 years old:
    age.old := fuzzy set (39 years, 0), (40 years, 1);

    //  Theophylline Dose
    if patientAge is age.young then
        dose := 8;
    else if patientAge is age.middleAged then
        dose := 15;
    else if patientAge is age.old then
        dose := 20;
    endif;
    conclude true;
  ;;

  action: ////////////////////////////////////////
    write dose;
  ;;
end:
```

X5 SUMMARY OF CHANGES

X5.1 Summary of changes from the 1992 standard (Version 1) to Version 2:

- 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 (11.2.16)
- 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** (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 (**Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**)
  - **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. (**Error! Reference source not found.**)
- 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. (10.2.5, 11.2.5, 12.2.2, 12.2.5)
- Looping constructs have been added: for loop, while loop. (10.2.5.10, 10.2.6.1)
- The **continue** statement may have an **unless** added to it (this 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).

X5.2 Summary of changes from Version 2 to Version 2.1:

- A structured message for the **write** statement, represented as a Document Type Definition to be encoded in the Extensible Markup Language (XML), has been included. (X1.4.1 <structured.message>)
- The **in** operator is now a synonym for **is in**; similarly, **not in** is synonymous with **is not in**. (9.6.23)
- **Occur/occurs/occurred at** is now synonymous with **occur/occurs/occurred equal**. (9.7.11)
- The syntax **from <time>** is now synonymous with **after <time>**. (9.10.4)
- A period punctuation mark (".") now is permissible in the **Mlmname** slot. (6.1.2)
- New reserved word **currenttime** returns the system time at any point during an MLM's execution. (8.4.6)
- Six new string-handling operators are now available. These include **length** (9.8.5), **uppercase** (9.8.6), **lowercase** (9.8.7), **trim** (9.8.8), **find…in string** (9.8.9), and **substring…characters from** (9.8.10).
- The **where trigger** statement has been removed.
- Added new code for Arden Syntax version slot—Version 2.1—to distinguish Version 2 and Version 2.1 compliant MLMs.

X5.3 Summary of changes from Version 2.1 to Version 2.5:

The following relate to new Object capabilities:

Added new sections:

- 10.2.7, **New** statement.
- 11.2.1.9, **Read As** statement.
- 11.2.5.2 **Message As** statement
- 11.2.5.6 **Destination As** statement
- 11.2.13, **Object** statement.
- 10.2.1.1, **Attribute assignment** statement.
- 9.18, **Dot** notation (attribute reference)
- 9.19, **Clone** operator (attribute reference)
- 8.10, **Object** data type
- Annex A6, Objects in Arden: rationale, details, etc.

Section A4.3, new operators is object, is not object, is <object-name>, is not <object-name> were added.

The following updates relate to new recommendations for formatting structured citations and links

- 6.2.4, **Citations** slot now recommends ANSI/NISO OpenURL format for structured citations
- 6.2.5, **Links** slot now recommends ANSI/NISO OpenURL format for structured links
- Annex A1, XML schema for MLMs replaces DTD

The following updates relate to new recommendations for representing MLMs using XML

- Appendix X1, XML schema for structured write replaces DTD for structured write
- Appendix X2, XML schema for MLMs added

Annex A1 Backus-Naur Form updated to include new operators, statements, and correct errors from previous versions

Updated B/N forms for:

- <data_assign_phrase>
- <expr_factor>
- <logic_assignment> (fixed a problem in 2.1 B/N form relating to calling MLMs that return multiple values)
- <identifier_becomes>
- <unary_comp_op>
- <data_assignment>
- <expr_function>
- <of_noread_func_op>

These B/N Forms were added:

- <object_definition>
- <object_attribute_list>
- <new_object_phrase>
- <identifier_or_object_ref>
- <expr_attribute_from>
-

- **Annex A2 Reserved Words updated to include new operators and statements**
- **Annex A4 Operator Precedence and Associativity updated to include new operators**

X5.4 Summary of Changes from Version 2.5 to 2.6

- **5.1 Character set allows UNICODE encoding within certain limitations**
- **6.2.5 Changes to structured version of links slot.**
- **6.4 Resource category defines text resources for specific languages**
- **7.1.11 Time of day constants**
- **8.11 Time-of-day data type**
- **8.12 Day-of-week datatype**
- **9.1.5 Time of day handling**
- **9.6.21 Is [not] time of day**
- **9.10.5 Time of day operator**
- **9.10.6 Day of week operator**
- **9.8.11 Localized operator (unary)**
- **9.8.12 Localized operator (binary)**
- **9.17.3 At**
- **11.2.15 Extension of include statement to include resources**
- **X3 Selected language and country codes for use with resource category slots.**

This version features new data types and operators to represent time-of-day and day-of-week. In addition, new capabilities have been added to let an MLM report messages in a variety of languages. The modifications include:

X5.5 Summary of Changes from Version 2.6 to 2.7

- 9.17.3 **AT** (time) changed to **ATTIME** to remove need for precedence rules to proper parse use of **AT** (time) in write statement with destination.
- 10.2.1.2 Enhanced Assignment Statement changed to support directly assigning to nested attributes of objects and specific elements in a list
- 10.2.4.10 Enhanced Assignment in Call Statement
- 10.2.7.1 New Statement with Named Initializer (objects)
- Evoke slot chapter reorganized and rewritten
- Changes to BNF to reflect updates to text of standard and fix typographical errors

X5.6 Summary of Editorial Corrections of ANSI/HL7 Arden V2.7-2008 December 10, 2008

- TOC Updated numbering of chapter 11.2.10 to 11.2.18 in the table of contents
- 9.1.3 Added "*Each operator must apply the here described list handling first (if applicable) before the specific list handling as described in the respective operator description is applied.*" to make the correct application of list handling clearer.
- 9.1.3.4 Removed **… matches pattern …** because this does not belong into this chapter.
- 9.1.3.4 added missing **… from …** operator
- 9.3.1 Correction of first example since the stated operator **… is within … after …** does not exist, **… is within … following …** must be used.
- 9.4.1 Type constraint updated because **… or …** is also applicable to lists.
- 9.7 Several "occured" changed to "occurred".
- 9.8.1 Corrected **%z** to **%s** because there is no such operator **%z**.
- 9.8.4 2nd type constraint removed. <k:list of strings> means a list with k elements of "list of strings", which is a list of lists and not allowed in Arden Syntax.
- 9.9.7 Type constraint corrected to ensure that the right side of the **… ** …** operator is not a list.
- 9.12.19 Updated the type constraint for **extract characters** operator to ensure that the list of arguments is of type string.
- 10.2.1.2 Operator corrected (**element** instead of **index**), corrected examples ("msg" instead of "message", "message" not allowed as variable name)
- 10.2.7 Definition of non-terminal **<object-identifier>** added.
- 11.2.5 Removed "*[...] If the MLM is evoked instead of called, all the arguments are treated as null. [...]*" since this sentence is in contradiction with Chapter 10.2.4.6.
- 11.2.8 to 11.2.18 Updated numbering of chapters.
- A1 BNF expression for `<read_where>` updated with missing "<" and ">".
- A1 BNF expression for `<evoke_statement>` updated with the missing non-terminal `<delayed_evoke>`
- A1 BNF expression for `<delayed_evoke>` updated with the missing quotation marks.
- A1 BNF expression for `<relative_evoke_time_expr>` updated, since this non-terminal was still using "AT" instead of "ATTIME"
- A2 **arccos** instead of **arcos**
- A4 Operators added to precedence groups: 9.16.10, 9.16.14
- A4 **arccos** instead of **arcos**
- A5.1 Some letter must be lowercase instead of using them in uppercase twice.

X5.7 Summary of Changes from Version 2.7 with Editorial Corrections to 2.8

- 3.2.1, Removed the "a point in absolute time" term
- 6.1.2, Added the **minus** sign, since the BNF (non-terminal <mlmname_text_rest>) allows this sign inside of an MLM name
- 6.1.7, Changed slot type to "textual list" since the informal description claims the same format as the author slot
- 6.1.8, Added short term, which makes clear that only the complete representation (given in the **ISO**) is allowed
- 8.1, Added a sentence to make clear that null may have a primary time
- 8.4.1, Changed the granularity of time from infinitesimal to implementation specific (beyond milliseconds)
- 9.1.2.2, Additional data type "**times**" introduced, which subsumes **time** and **time-of-day**
- 9.1.2.2, Added "**time-of-day**" within the types: **any-type**, **non-null**, and **ordered**
- 9.1.3.1, Added the operators "**… As Number**", "**… As String**", and "**… As Time**" to the general list handling
- 9.1.3.4, Added the operators "**Replace Year Of … With**", "**Replace Month Of … With**", "**Replace Day Of … With**", "**Replace Hour Of … With**", "**Replace Minute Of … With**", and "**Replace Second Of … With**" to the general list handling
- 9.1.3.6, Added the operators "**Index Of … From …**" , "**Add … To …**" , "**At Least … From …**", and "**At Most … From …**" to the general list handling
- 9.1.3.7, Added the "**Remove … From …**" operator to the general list handling
- 9.2.4, Added the "**Using …**" modifier as extension to the **sort** operator. This modifier will allow to sort lists by any complex calculation
- 9.2.5, Added new operator "**Add … To … [At …]**" for simple list manipulation by insertion of elements at arbitrary positions
- 9.2.6, Added new operator "**Remove … From …**" for simple removing arbitrary elements from a list
- 9.6.7, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.8, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.9, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.10, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.12, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.13, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.6.14, Added 2 sentences to make the null handling of the "**… Is [Not] In …**" operator clearer
- 9.7.2, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.3, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.4, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.5, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed

- 9.7.6, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.9, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.10, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.7.11, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.8.13, Added new operator "**As String**" to convert any data into a string
- 9.9.1, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.9.3, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.1, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.2, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.4, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.7, Moved operator "**Extract Year**" from chapter 9.11.2
- 9.10.8, Moved operator "**Extract Month**" from chapter 9.11.4
- 9.10.9, Moved operator "**Extract Day**" from chapter 9.11.7
- 9.10.10, Moved operator "**Extract Hour**" from chapter 9.11.9
- 9.10.10, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.11, Moved operator "**Extract Minute**" from chapter 9.11.11
- 9.10.11, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.12, Moved operator "**Extract Second**" from chapter 9.11.13
- 9.10.12, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.10.13, Added new operator "**Replace Year [Of] … With**" to set the year part of a given date
- 9.10.14, Added new operator "**Replace Month [Of] … With**" to set the month part of a given date
- 9.10.15, Added new operator "**Replace Day [Of] … With**" to set the day part of a given date
- 9.10.16, Added new operator "**Replace Hour [Of] … With**" to set the hour part of a given date
- 9.10.17, Added new operator "**Replace Minute [Of] … With**" to set the minute part of a given date
- 9.10.18, Added new operator "**Replace Second [Of] … With**" to set the second part of a given date
- 9.12.3, Added a sentence to make clear what the **exists** operator does if the parameter is a single item
- 9.12.4, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.12.5, Changed the operator type constraint from <n:time> to <n:times> to describe that **time-of-day** values are also allowed
- 9.12.9, Added the ability to use the "**using**" modifier, too
- 9.12.10, Added the ability to use the "**using**" modifier, too
- 9.12.13, Added the optional keyword "**IsTrue**"

- 9.12.14, Added the optional keyword "**AreTrue**"
- 9.12.15, Added the optional keyword "**IsTrue**"
- 9.12.16, Added a sentence to make clear what happens if there is more than one element with the latest primary time
- 9.12.16, Added the ability to use the "**using**" modifier, too
- 9.12.17, Added a sentence to make clear what happens if there is more than one element with the earliest primary time
- 9.12.17, Added the ability to use the "**using**" modifier, too
- 9.12.20, Corrected an example (added brackets) since **seqto** operator has higher precedence than unary minus
- 9.13.2, Corrected the operators type constraint, since the formal description only allows single times as first parameter
- 9.13.4, Added new operator "**Index Of … From …**" to find the index of a specific list element
- 9.13.5, Added the "**At Least … [IsTrue|AreTrue] From …**" operator to determine if a list contains at least N elements which are true
- 9.13.6, Added the "**At Most … [IsTrue|AreTrue] From …**" operator to determine if a list contains at most N elements which are true
- 9.14.2, Added the ability to use the "**using**" modifier, too
- 9.14.3, Added the ability to use the "**using**" modifier, too
- 9.14.6, Added new operator "**Sublist … Elements [Starting at …] From …**" to extract sub-lists from given data lists
- 9.14.7, Adjusted the second type constraint such that the operator can handle lists of **time-of-day** values and added an example
- 9.14.8, Adjusted the second type constraint such that the operator can handle lists of **time-of-day** values and added an example
- 9.14.11, Added the ability to use the "**using**" modifier, too
- 9.14.12, Added the ability to use the "**using**" modifier, too
- 9.16.10, Corrected the first two examples (added brackets) since **int** operator has higher precedence than unary minus
- 9.16.12, Corrected the first two examples (added brackets) since **ceiling** operator has higher precedence than unary minus
- 9.16.13, Corrected the first two examples (added brackets) since **truncate** operator has higher precedence than unary minus
- 9.16.14, Corrected the last three examples (added brackets) since **round** operator has higher precedence than unary minus
- 9.17.1, Added a sentence to make clear what happens if a non-time value is used for the assignment
- 9.17.4, Added new operator "**As Time**" to convert a string into a time data type
- 9.18.3, Changed the operators type constraint such that only one object can be passed
- 10.2.1, Changed the description such that it will be clear that a re-assignment is allowed nowhere outside of the data slot
- 10.2.3, Added the "**Switch-Case**" statement for simple distinction of different states of a variable
- 10.2.3.1, Added a chapter to describe the "**Simple Switch-Case**" statement
- 10.2.3.2, Added a chapter to describe the "**Switch-Case-Default**" statement
- 10.2.6.1, Added the possibility to use the terminal "**BreakLoop**" for aborting a while loop
- 10.2.7.1, Added the possibility to use the terminal "**BreakLoop**" for aborting a for loop

- 11.2.3.1, Added a sentence to describe the default Boolean value of a variable that represents an event
- 11.2.12, Added the "**Switch-Case**" statement to the data slot, too
- 11.2.14, Added a reference to the **breakloop** statement
- 11.2.15, Added a reference to the **breakloop** statement
- 11.2.19, Added MLM, event, and interface variable to the listing, since chapter 10.2.5.2 claims that they are also included
- 12.2.4, Added the "**Switch-Case**" statement to the action slot, too
- 12.2.6, Added a reference to the **breakloop** statement
- 12.2.7, Added a reference to the **breakloop** statement
- A1 BNF, Added version 2.7 and 2.8 to the non-terminal and <arden_version>
- A1 BNF, added multiple non-terminals (<action_switch>, <logic_switch>, and <data_switch>) and added them to the general statements for the **data**, **action**, and **logic** slot to allow switch statements in all of these slots
- A1 BNF, Added the terminal "**BREAKLOOP**" to the non-terminals <logic_statement>, <data_statement>, and <action_statement>
- A1 BNF, Adjusted non-terminals <identifier_becomes> and <identifier_or_object_ref> to allow the enhanced assignment statements described in 10.2.1.2
- A1 BNF, Added **using** modifier to the non-terminal <expr_function> and to the non-terminal <expr_sort>
- A1 BNF, Added the new operator "**Add … To …**" to the non-terminal <expr_sort> and inserted a new non-terminal <expr_add_list>
- A1 BNF, Added the new operator "**Remove … From …**" as non-terminal <expr_remove_list>
- A1 BNF, Added an additional "**… Formatted With …**" line to the non-terminal <expr_string> to allow complex format strings
- A1 BNF, Removed the terminals "**Uppercase**" and "**Lowercase**" from the non-terminal <of_noread_func_op> and added them to the non-terminal <expr_string> as non-terminal <case_option>
- A1 BNF, Added non-terminal <expr_attime> to prevent infinite loops while parsing **attime** statements
- A1 BNF, Added alternative non-terminal to the BNF-expression <expr_duration>
- A1 BNF, Added the new operators "**Replace <Timepart> Of … With …**" to the non-terminal <expr_funtion>
- A1 BNF, Added the **at least** and the **at most** operator as non-terminal <at_least_most_op> to the non-terminal <expr_function>
- A1 BNF, Added the "**Index Of … from …** " operator to the non-terminal <expr_function>
- A1 BNF, Added the **sublist** operator to the non-terminal <expr_function> by adding the non-terminal <expr_sublist_from>
- A1 BNF, Added the optional keywords "**IsTrue**" and "**AreTrue**" to the operators **no**, **any** and **all** in the non-terminal <of_noread_func_op>
- A1 BNF, Added the new operator "**… As Time**" to the non-terminal <as_func_op>
- A1 BNF, Added the new operator "**… As String**" to the non-terminal <as_func_op>
- A1 BNF, Added an additional non-terminal <timepart>
- A1 BNF, Changed the non-terminal <delayed_evoke> to fit the informal description which does allow only simple duration statements on the left side of constant time trigger statements
- A1, BNF, Change description of the <plainstring> non-terminal since both, the regular expression and the informal description (7.1.6) does allow ";;" in a string
- A1 BNF, Added non-terminal <seconds> and adjusted the <time_of_day> non-terminal definition

- A2, Added the following words to the list of reserved words: **add**, **aretrue, breakloop, case**, **elements**, **istrue, least**, **most**, **remove**, **replace, sublist**, **switch**, **using**
- A4, Added the **element** operator
- A4, Added the unary **comma** operator to the list of precedence
- A4, Added the "**Add … To … [At …]**" operator
- A4, Added the "**Remove … From …**" operator
- A4, Removed binary "**… Round …**" operator, which is not defined in the specification
- A4, Added the "**Sublist … elements [Starting At …] From …**" operator in its two occurrences
- A4, Added the "**Index Of … Within …**" operator
- A4, Added the "**At Least …**" operator
- A4, Added the "**At Most …**" operator
- A4, Added the "**Replace <timepart> Of … With …**" operators
- A4, Added "**… Seqto …**" operator as new group at the end of the list
- A4, Added new precedence group for "**… As Number**", "**… As Time**", and "**… As String**"
- A4, Split some precedence groups since operators with different associativity should not be in the same precedence group
- A4, Added the operators extended by the **using** modifier

Summary of Changes from Version 2.8 to 2.9

- 6.4, changed **resources** category definition from optional to required, stating that in former versions this category is optional and a default value is used
- 8.13, new data type **Truth Value** which is a generalization of Boolean
- 8.14, new **fuzzy data type** section which contains a set of data types to express fuzzy sets
- 8.15, added **applicability**, similar to "primary time" a new subcomponent is added which allows to express the applicability of a value
- 9.1.2.2, changed some **type categories** and added some new type categories to allow to use them in the operator signatures
- 9.1.3, added the new operators to **list handling** explanation
- 9.1.6, added **general applicability handling** (similar to primary time handling)
- 9.2.4, **sort** operator adjusted to be able to sort a list by the **applicability** of the values
- 9.4.1, adjusted the **or** operator to handle truth values
- 9.4.2, adjusted the **and** operator to handle truth values
- 9.4.3, adjusted the **not** operator to handle truth values
- 9.5.4, adjusted the <= operator to handle a crisp and a fuzzy data type
- 9.5.5, adjusted the >= operator to handle a crisp and a fuzzy data type
- 9.6.14, the **is [not] in** operator is now able to handle a crisp and a fuzzy data type to find the mapping of the crisp value to the given fuzzy set
- 9.6.27, new operator **is[not] fuzzy** to check a value if it is fuzzy or not
- 9.6.28, new operator **is[not] crisp** to check a value if it is crisp or not
- 9.13.5, adjusted the **at least** operator to handle truth values in a list
- 9.13.6, adjusted the **at most** operator to handle truth values in a list
- 9.19, new section **fuzzy operators** to store all operators on fuzzy sets
- 9.19.1, added new operator **fuzzy set ...** which is able to create fuzzy sets
- 9.19.2, added new operator **... fuzzified by ...** which is able to create simple triangular fuzzy sets
- 9.19.3, added new operator **defuzzified ...** which defuzzifies a fuzzy set
- 9.19.4, added new operator **applicability [of] ...** to access a values applicability (and to set it)
- 9.20, new section **type conversion operators** which contains all type conversion operators
- 9.20.1, the **as number** operator moved from 9.16.17
- 9.20.2, the **as time** operator moved from 9.17.4
- 9.20.3, the **as string** operator moved from 9.8.13
- 9.20.4, added new operator **as truth value** which converts a number into a truth value
- 10.2.2, adjusting the **if-then-statements** to describe what happens if the condition expression evaluates to a truth value
- 10.2.3, adjusting the **switch-statements** to describe what happens if the condition expression evaluates to a truth value
- 10.2.8, added reference to the **linguistic variable definition** which is a special object type
- 11.2.18, added the **linguistic variable statement** which describes an object with only fuzzy sets as fields
- A1, changes to the **BNF** to reflect the new operators and changes to the existing statements
- A2, added reserved words: **aggregate, applicability, crisp, defuzzified, endswitch, fuzzified, fuzzy, linguistic, set, truth, value, variable**
- A4, added the new operators

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.