How to write Arden Syntax MLMs – An introduction

Karsten Fehre

Medexter Healthcare GmbH
Borschkegasse 7/5
A-1090 Vienna, Austria
www.medexter.com

Content

- **Arden Syntax – General Information**
- **General MLM Layout**
  - Maintenance Category
  - Library Category
  - Knowledge Category
  - Resources Category
- **Identify an MLM**
- **Data Types**
- **Operators**
  - Basic Operators
  - Curly Braces
  - List Operators
  - Logical Operators
  - Comparison Operators
  - String Operators
  - Arithmetic Operators
  - Other Operators
- **Control Statements**
- **Call / Write Statements and Trigger**

Arden Syntax – General Information I

- Medical knowledge in Arden Syntax is arranged within Medical Logic Modules (MLMs)
- Each MLM represents sufficient knowledge to make a single decision
- One or more MLMs are stored within a file that has the extension "mlm"
- Each MLM is well organized and structured into categories and slots with specific content

```
maintenance:
  title:        [TITLE_(needed)];;
  mlmname:      [MLM-NAME_(needed)];;
  arden:        Version 2.5;;
  version:      [MLM-VERSION_(needed)];;
  institution:  [INSTITUTION_(needed)];;
  author:       ;;
  specialist:   ;;
  date:         [DATE];;
  validation:   testing;;
library:
  purpose:      ;;
  explanation:  ;;
  keywords:     ;;
  citations:    ;;
  links:        ;;
knowledge:
  type:         data_driven;;
  data:         ;;
  priority:     ;;
  evoke:        ;;
  logic:
    conclude true;
  ;;
  action:
  ;;
  urgency:      ;;
end:
```

Arden Syntax – General Information II

- An MLM works in close contact with a host system. Ways of interaction are:
  - **Input**: By calling an MLM, an input parameter can be committed.
  - **Curly Brace Expressions**: So called "curly brace expressions" implement a special kind of dynamic interaction between MLMs and host systems.
  - **Write Statements**: Texts can be written to destinations which are maintained by the host system.
  - **Output**: Analogous to the input parameter, some data can be committed from the MLM to the host system when the execution of the MLM is finished.
- To start the execution of an MLM, an engine is needed which handles communication with the host system and knows which MLMs are available
- Ways to start running an MLM
  - **MLM call**: an MLM is directly called
  - **Event call**: any MLM which listens to a specific event is executed

General MLM Layout

- An MLM is composed of slots grouped into four required **categories: maintenance**, **library**, **knowledge**, and **resources**.
- A category starts with its name followed immediately by a colon (e.g., maintenance:).
- Categories must appear in the correct order.
- Within each category is a set of **slots**.
- Slots must appear in the correct order, too.
- In general, an MLM is arranged such as:

```
maintenance:
   slotname: slot-body;;
   slotname: slot-body;;
   ...
library:
   slotname: slot-body;;
   ...
knowledge:
   slotname: slot-body;;
   ...
resources:  <optional>
   slotname: slot-body;;
```

Maintenance Category

- Contains the slots that specify some general information unrelated to the health knowledge in the MLM
- Slots are used for MLM knowledge base maintenance and change control
- Contains the version of the Arden Syntax in use

- Slots:
  - Title
  - Mlmname (content required)
  - Arden Syntax version (content required)
  - Version (content required)
  - Institution (content required)
  - Author
  - Specialist
  - Date
  - Validation (one of: production, research, testing, expired)

- Mlmname, Institution, and Version are used to identify an MLM

Library Category

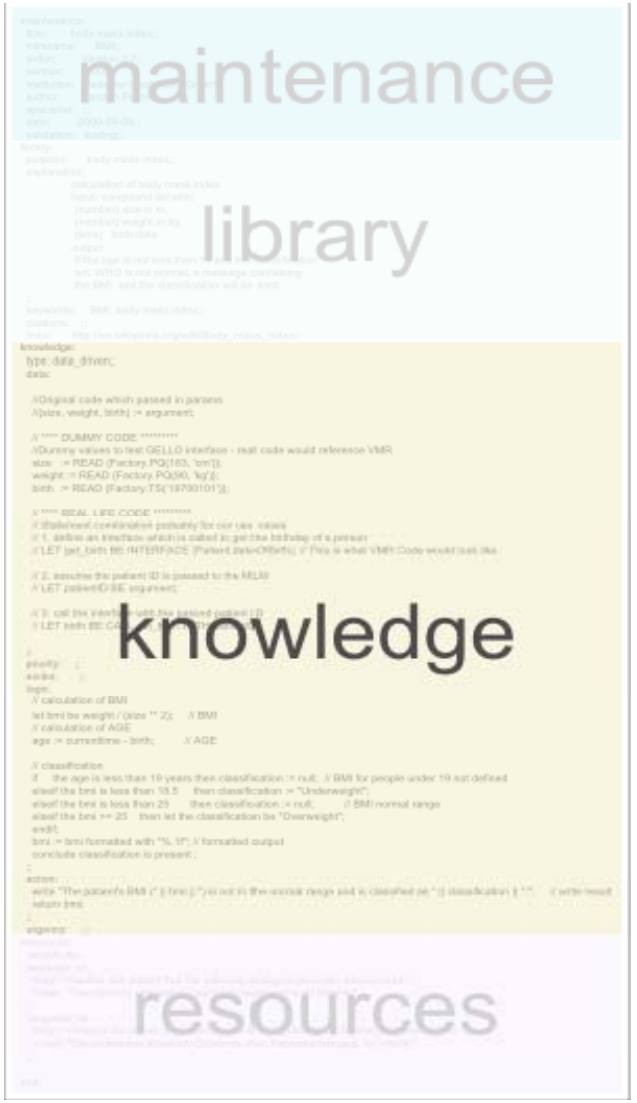
- Contains the slots pertinent to knowledge base maintenance that are related to the MLM's knowledge
- Slots provide health personnel with predefined explanatory information and links to relevant health literature

- Slots
  - Purpose
  - Explanation
  - Keywords
  - Citations
  - Links

Knowledge Category

- Contains the slots that actually specify what the MLM does
- Slots of the knowledge category 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),
  - The action to be taken should the condition be true (action slot).

- Slots:
  - Type (only "data-driven" available)
  - Data
  - Priority
  - Evoke
  - Logic (contains the actual logic of the MLM)
  - Action (is executed if logic slot concludes **true**)
  - Urgency



Resources Category

- Contains a set of language slots that specify the textual resources from which the localized operator should draw to obtain message content in different languages
- Each language slot defines a set of key/value pairs which represent text constants in one specific language
- At least one language slot is required

- Slots:
  - Default (defines the default language to be used)
  - Language (one language slot for each language to be used)

- Example:

```
resources:
  default: de;;
  language: en
   'msg' : "The patient's BMI %.1f is not in the normal range and is
    classified as ";
  ;;
  language: de
   'msg' : "Der BMI %.1f des Patienten ist nicht im normalen Bereich und
    wird klassifiziert als ";
  ;;
```

Sample MLM

- Most of the examples for operator and concept explanation are taken from the following sample MLM which calculates the body mass index (BMI) of a patient

```
maintenance:
  title: simple body mass index;;
  mlmname: BMI_HowTo;;
  arden: Version 2.7;;
  version: 1.00;;
  institution: Medexter Healthcare GmbH;;
  author: Karsten Fehre;;
  specialist: ;;
  date: 2010-09-09;;
  validation: testing;;
library:
  purpose: body mass index;;
  explanation: calculation of body mass index;;
  keywords: BMI, body mass index;;
  citations: ;;
  links: http://en.wikipedia.org/wiki/Body_mass_index;;
```

Sample MLM (cont.)

```
knowledge:
  type: data_driven;;
  data:

    // MLM that contains the interface definition "LET get_birth BE INTERFACE {Patient.dateOfBirth}; "
    mlmImport       := MLM 'interface_birthday_definition';

    // include
    include mlmImport;

    mlmForReadSize  := MLM 'read_Size_MLM'; // MLM which can read the current size of the patient from the DB

    LET patientID BE argument; // the patient ID is passed to the MLM

    LET birth     BE CALL get_birth WITH patientID; // call the interface with the passed patient ID

    // read all measured weights from the database
    LET weights   BE READ {SELECT measured_weight FROM DB WHERE patID = patientID };

    LET userEvent BE EVENT {getBMI};

    // object declaration
    bmiResult := object [bmi, classification];

  ;;
  priority: ;;
  evoke:
    userEvent;
  ;;
```

Sample MLM (cont.)

```
logic:
  result := new bmiResult; // create an empty result object

  weight := latest of weights; // get the latest weight from the list

  size := call mlmForReadSize with patientID; // get the size of the patient calculated by another MLM

  result.bmi := weight / (size ** 2); // calculation of BMI
  age := currenttime - birth; // calculation of AGE

  // classification - the classification is only valid for patients older than 19
  if the age is less than 19 years then result.classification := null;
  elseif the result.bmi is less than 18.5 then result.classification := localized 'under';
  elseif the result.bmi is less than 25 then result.classification := null;
  else let the result.classification be localized 'over';
  endif;

  result.bmi := result.bmi formatted with localized 'msg'; // construct the localized message

  if (time of weight) is before (currenttime - 6 months) then
    conclude false; //no bmi calculation if the latest measure was 6 months ago
  else
    conclude result.classification is present ; // if there is a classification, execute the action slot
  endif;

;;
```

Sample MLM (cont.)

```
  action:

    write result.bmi || result.classification || "."; // return result

    return result;
  ;;
  urgency: ;;
  resources:
    default: de;;
    language: en
      'msg' : "The patient's BMI %.1f is not in the normal range and is classified as ";
      'under' : "Underweight";
      'over' : "Overweight"
    ;;
    language: de
      'msg' : "Der BMI %.1f des Patienten ist nicht im normalen Bereich und wird klassifiziert als ";
      'under' : "Untergewicht";
      'over' : "Übergewicht"
    ;;
end:
```

Identify an MLM

- An MLM is identified using the following 3 pieces of information:
  - **Name**, as given in the mlmname-slot
  - **Institution**, as given in the institution-slot
  - **Version**, as given in the version-slot

- Example:
  The MLM with the following maintenance category

```
maintenance:
   title: simple body mass index;;
   mlmname: BMI;;
   arden: Version 2.7;;
   version: 1.00;;
   institution: Medexter Healthcare GmbH;;
   author: Karsten Fehre;;
   specialist: ;;
   date: 2010-09-09;;
   validation: testing;;
```

  can be addressed using the following MLM definition in the data-slot:

```
bmiMLM := MLM 'BMI' from institution "Medexter Healthcare GmbH";
```

  Note: If there is more than one MLM with the same name and institution, the MLM with the latest version number is used.

General Language Expressions

- **Statement**: A statement specifies a logical constraint or an action to be performed. All statements except for the last statement in a slot must end with a semicolon (;).

```
let var1 be 0; // equal to: var1 := 0;
```

- **Constant**: Any data value represented explicitly is called a constant.

```
true
 "this is a string"
```

- **Variable**: A variable is a placeholder for a data value or special constructs (event, MLM, message, and destination) and represents this value in subsequent expressions. An assignment statement is used to assign to a variable a value.

```
let var1 be 0; // equal to: var1 := 0;
var2 := MLM 'BMI' from institution "medexter";
```

- **Operator**: An expression may contain an operator and sub-expressions called arguments.

```
3 + 5 //where + is the operator, 3 and 5 are the arguments
```

- **Primary Time**: Each data value consists of a value part and a primary time part which represents the time of creation or measurement of the value part.

Data Types

- **Null**: special data type that signifies unknown/uncertainty
- **Boolean**: includes two truth values, true and false; logical operators use tri-state logic by using null to signify the third state, unknown/uncertainty

```
true
false
```

- **Number**: no distinction is made between integer and floating point numbers

```
7
7.34323
```

- **Time**: refers to points in time; times before 1800-01-01 are not valid

```
2011-07-12T00:00:12
2011-07-12
```

- **Duration**: signifies an interval of time

```
19.01 years
3 days 1 hours 2 minutes 54.6 seconds
```

- **String**: streams of characters

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

Data Types (cont.)

- **List**: an ordered set of elements; each element can be an arbitrary data type

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

- **Object**: may contain multiple named attributes, each of which may contain any valid data type

```
MedicationDose := OBJECT [Medication, Dose, Status];
dose := NEW MedicationDose with "Ampicillin", "500mg", "Active";
// dose refers to an object with the fields Medication, Dose, Status
"Ampicillin" := dose.Medication;
```

- **Time-of-day**: refers to points in time that are not directly linked to a specific date

```
23:20:00
```

- **Day-of-week**: special data type to represent specific days of the week; represented by constants or integer

```
MONDAY (1)
TUESDAY (2)
...
```

Operators – Basic Statements I

- **Assignment:** places the value of an expression into a variable

```
<identifier> := <expression>;
LET <identifier> be <expression>;
```

- **Write:** sends text or coded message to a destination

```
write dose.Medication || " with " || dose.Dose;
write "this is an email alert" AT email_dest;
```

- **Include:** includes object, MLM, event, interface, and resource definitions from an external MLM

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

Operators – Basic Statements I – Example

```
// MLM that contains the interface definition "LET get_birth BE INTERFACE {Patient.dateOfBirth}; "
mlmImport       := MLM 'interface_birthday_definition';

// include
include mlmImport;
```

- First is an assignment, assigning the reference to the MLM `interface_birthday_definition` to the variable `mlmImport`
- Second is an include statement which imports all object, MLM, event, interface, and resource definitions from the MLM `mlmImport` (`interface_birthday_definition`)

```
write result.bmi || result.classification || "."; // return result
```

- Concatenates the calculated BMI and its classification to a string and sends this message to the default destination

Operators – Basic Statements II

- **If-Then:** permits conditional execution based on the value of an expression
  - 3 different types of if-then statements:

If-Then:
block1 is executed if condition is true

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

If-Then-Else:
block1 is executed if condition is true, otherwise (if condition is false or anything other than true) block2 is executed

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

If-Then-ElseIf:
block1 is executed if condition1 is true, if condition2 is true block2 is executed, in all other cases block3 is executed

```
IF <cond1> THEN
   <block1>
ELSEIF <cond2> THEN
   <block2>
ELSE
   <block3>
ENDIF;
```

Operators – Basic Statements II – Example

```
// classification - the classification is only valid for patients older than 19
if the age is less than 19 years then result.classification := null;
elseif the result.bmi is less than 18.5 then result.classification := localized 'under';
elseif the result.bmi is less than 25 then result.classification := null;
else let the result.classification be localized 'over';
endif;
```

- This is an If-Then-ElseIf statement
- If the age of the current patient is less than 19 years, `null` is assigned to the classification variable (the BMI specification is only valid for persons over 19 years)
- Otherwise, if the calculated BMI is less than 18.5, the localized string for underweight is assigned to the classification variable
- Otherwise, if the calculated BMI is less than 25 (this means between 18.5 and 25), `null` is assigned to the classification variable (no alert is required if the patient is in normal BMI)
- Otherwise (i.e., all BMIs greater than 25), the localized string for overweight is assigned to the classification variable

Operators – Basic Statements III

- **Loops**
  - **While Loop:** loops as long the condition is equal to true

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

  - **For Loop:** loops over the elements of a list

```
FOR i IN (1 seqto 10) DO
  ... // i can be used inside of the loop
ENDDO;
```

- **Conclude**: ends execution in the logic slot; if the conclude statement has a single true as argument then the action slot is executed immediately; otherwise the MLM terminates immediately

- **Argument**: if a calling instance passes parameters to the called MLM, the MLM retrieves the parameters via the argument statement

- **Return:** returns the provided parameter to the calling instance (which may be another MLM or an external instance)

Operators – Basic Statements III – Example

```
conclude result.classification is present ; // if there is a classification, execute the action slot
```

- **Conclude statement**
- "`result.classification is present`" will evaluate to `true`, if the classification variable does not refer to null
- If "`result.classification is present`" evaluates to `true`, the execution of the logic slot stops immediately and the execution of the action slot will start
- If "`result.classification is present`" evaluates to `false`, the execution of the logic slot also stops immediately but the action slot will not be executed and the evaluation of the MLM terminates

```
LET patientID BE argument;
```

- Argument statement which assigns all incoming parameters to the variable `patientID`

```
return result;
```

- Returns the object `result` to the calling instance (if the MLM is called from an other MLM, it will be returned to the calling MLM)

Operators – Object Statements

- **Object:** assigns an object declaration to a variable (objects are the only data types in Arden Syntax which are first declared and then "instantiated")

```
MedicationDose := OBJECT [Medication, Dose, Status];
```

- **New:** causes a new object to be created (based on the used object declaration)

```
dose1 := NEW MedicationDose; //empty object
dose2 := NEW MedicationDose with "Ampicillin", "500mg", "Active";
```

- **Dot:** selects an attribute from an object based on the name following the dot

```
"John" := patient.Name.FirstName;
NameType := object [FirstName, LastName];
/* Assume namelist contains a list of 2 NameType objects */
("John", "Paul") := namelist.FirstName;
```

Operators – Object Statements – Example

```
// object declaration
bmiResult := object [bmi, classification];

result := new bmiResult; // create an empty result object
```

- The first creates an object declaration with two fields and assigns this declaration to the variable `bmiResult`
- The second creates an empty instance of the `bmiResult` object and assigns this to the variable `result`

```
write result.bmi || result.classification || "."; // return result
```

- Concatenates the content of the field `bmi` and the content of the field `classification` of the object `result` to a string and sends this message to the default destination

```
result.bmi := result.bmi formatted with localized 'msg';
```

- The **dot operator (". ")** is used to access the fields of an object
- The field `BMI` of the object `result` will be filled with the formatted text containing the calculated BMI

Curly Braces (Mapping Clauses)

- Used in the data slot to signify institution-specific definitions such as database queries
- **Read statement:**

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

- **Event statement:** defines an event; an event can be used to call MLMs

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

- **Message statement:**

```
message1 := MESSAGE {increased body temperature};
```

- **Destination statement:**

```
destination1 := DESTINATION {email: user@cuasdf.bitnet};
```

- **Interface statement:**

```
func_drugint := INTERFACE {char* API:Interaction (char*,char*) };
```

Curly Braces (Mapping Clauses) – Example

```
// read all measured weights from the database
LET weights   BE READ {SELECT measured_weight FROM DB WHERE patID = patientID };
```

- Assignment statement which assigns the result from the read statement (using mapping clause "`SELECT measured_weight FROM DB WHERE patID = patientID`") to the variable `weights`
- `patientID` is a variable which contains the current used patient ID and is substituted before execution of the mapping clause
- After evaluation of this statement, the variable weights refers to the result which is a list of all measured weights of the patient with the given patient ID

```
LET userEvent BE EVENT {getBMI};
```

- Assignment statement which assigns the event `getBMI` to the variable `userEvent`

```
evoke:
  userEvent;
;;
```

- Using the event variable in the evoke slot means that this MLM is always executed if this event occurs

Operators – List Operators

- **Concatenation:** appends two lists or turns a single element into a list of length one

```
(4,2) := 4, 2;
(,3) := , 3;
```

- **Merge:** appends two lists, appends a single item to a list, or creates a list from two single items; then sorts the result in chronological order based on the primary times of the elements

```
/* data1 has data value 2 and primary time 2011-01-02T00:00:00, and data2
has data values 1 and 3 and primary times 2011-01-01T00:00:00 and 2011-01-
03T00:00:00 */
(1, 2, 3) := data1 MERGE data2
null := (4,3) MERGE (2,1) // no primary time -> result is null
```

- **Sort:** reorders a list based on either the element values (keyword data) or the primary times (keyword time); default keyword is data

```
(1, 2, 3, 3) := SORT (1,3,2,3);
(10, 20, 30) := SORT DATA data1;
(30, 20, 10) := REVERSE (SORT DATA data1);
(30, 20, 10) := SORT TIME data1;
```

Operators – Logical Operators

- **And:** performs the logical conjunction of its two arguments; if either argument is false (even if the other is not Boolean), the result is false; if both arguments are true, the result is true; otherwise null is the result

```
false := true AND false
null := true AND null
false := false AND null
```

- **Or:** performs the logical disjunction of its two arguments; if either argument is true (even if the other is not Boolean) the result is true; if both arguments are false, the result is false; otherwise null is the result

```
true := true OR false
false := false OR false
true := true OR null
null := false OR null
null := false OR 3.4
```

- **Not:** true becomes false, false becomes true, and anything else becomes null

```
true := NOT false
null := NOT null
```

Operators – Comparison Operators

- **<,>,<=,=>,=, <>:** as usual, except one argument is null or types do not match; can handle any data type

- **Is within … to …:** checks if the first argument is within the range specified by the second and third argument (inclusive)

```
true := 3 IS WITHIN 2 TO 5
false := 3 IS WITHIN 5 TO 2
```

- **Is within … following …:** checks if a time is within a defined time period

```
false := 2011-03-08T00:00:00 IS WITHIN 3 days FOLLOWING 2011-03-10T00:00:00
```

- **Is in:** checks membership of the first argument in the second argument (list)

```
false := 2 IS IN (4,5,6)
(false,true) := (3,4) IS IN (4,5,6)
```

- **Is string|number|null etc.:** returns true if the argument is of the given type

Operators – Comparison Operators – Example

```
// classification - the classification is only valid for patients older than 19
if the age is less than 19 years then result.classification := null;
elseif the result.bmi is less than 18.5 then result.classification := localized 'under';
elseif the result.bmi is less than 25 then result.classification := null;
else let the result.classification be localized 'over';
endif;
```

- "less than" is a synonym to <
- "the age is less than 19 years" clearly returns true if the age is strictly under 19

Operators – String Operators I

- **Concatenation:** converts its arguments into strings and then concatenates them

```
"null3" := null || 3
"45" := 4 || 5
"list=(1,2,3)" := "list=" || (1,2,3)
```

- **Formatted with:** formats a string with a given pattern (like printf in ANSI C)

```
"The result was 10.61 mg"
:= 10.60528 FORMATTED WITH "The result was %.2f mg"

 "The date was Jan 10 2011"
:= 2011-01-10T17:25:00 FORMATTED WITH "The date was %.2t"
```

- **Localized:** returns a string that has been previously defined in the language slot of the MLM's resources category using a given or the current system's language

```
"Caution, the patient ..." := LOCALIZED 'msg' by "en_US";
"Achtung, der Patient ..." := LOCALIZED 'msg' by "de";
"Caution, the patient ..." := LOCALIZED 'msg'; //use system language
```

Operators – String Operators I – Example

```
write result.bmi || result.classification || "."; // return result
```

- The **concatenation operator** concatenates the string referred by the `BMI` field of the object `result` with the string referred by the `classification` field of the object `result` and the string `"."`

```
result.bmi := result.bmi formatted with localized 'msg';
```

- `"localized 'msg'"` will return the format pattern in the current system language
- The **formatted with** operator will then apply this pattern to the calculated BMI
- The result (a string) is assigned to the BMI field of the object `result`
- Assuming the calculated BMI is `29.4324` and the system language is English, the result of this formatted with expression is `"The patient's BMI 29.4 is not in the normal range and is classified as"`

Operators – String Operators I – Example

```
let the result.classification be localized 'over';
```

- The **localized operator** will return that string which is assigned to the term `'over'` in the **resources category**
- The operator will obtain the string from the **language slot** which matches the current language of the system the engine is running on
- If there is no language slot for the current system language, the defined default language is used
- Assuming English as the current system language, the whole statement will assign `"Overweight"` to the field `classification` of the object `result`

Operators – String Operators II

- **Uppercase, Lowercase:** converts all characters of a given string to lowercase/uppercase

```
"EXAMPLE STRING" := UPPERCASE "Example String";
"example string" := LOWERCASE "Example String";
```

- **Substring:** returns a substring of characters from a given string

```
"ab" := SUBSTRING 2 CHARACTERS FROM "abcdef";
"def" := SUBSTRING 3 CHARACTERS STARTING AT 4 FROM "abcdef";
```

- **Matches pattern:** determine if a string matches a pattern (similar to LIKE in SQL)

```
true := "fatal heart attack" MATCHES PATTERN "%heart%";
false := "fatal heart attack" MATCHES PATTERN "heart";
```

- **Length:** returns the length of a given string

```
7 := LENGTH OF "Example";
```

Operators – Arithmetic Operators

- **+, -, *, /, **:** as usual, except one argument is null or types do not match

```
2 days := 6 days / 3;
9 := 3 ** 2;
```

- **Cosine, Sine:** calculates the cosine (resp. sine) of its argument

```
1 := COSINE 0;
```

- **Log:** returns the natural logarithm of its argument

```
0 := LOG 1;
```

- **Abs:** returns the absolute value of its argument

```
1.5 := ABS (-1.5);
```

- **Ceiling:** returns the smallest integer greater than or equal to its argument

```
-3 := CEILING (-3.9);
```

- **Truncate:** removes any fractional part of a number

```
-1 := TRUNCATE (-1.5)
```

Operators – Arithmetic Operators – Example

```
result.bmi := weight / (size ** 2); // calculation of BMI
age := currenttime - birth; // calculation of AGE
```

- The BMI is calculated by dividing the current weight of the patient through the square of the current size
- The result is assigned to the field `BMI` of the object `result`
- The current age of the patient is calculated by subtracting the birthday from the current time
- The keyword `currenttime` is used to refer to the current system time
- Assuming that the birthday is `1977-12-12` and the current time is `2011-06-12T00:00:00`, after evaluating the statement the variable age will refer to the duration `33.5 years`

Operators – Temporal Operators

- **After, Before:** addition/subtraction of a duration and a time

```
2011-03-15T00:00:00 := 2 days AFTER 2011-03-13T00:00:00
2011-03-11T00:00:00 := 2 days BEFORE 2011-03-13T00:00:00
```

- **Time of day:** extracts the time-of-day from a time

```
14:23:17.3 := TIME OF DAY OF 2011-01-03T14:23:17.3
/* let time of data0 be 2011-01-01T12:00:00 */
12:00:00 := TIME OF DAY OF (TIME OF data0)
```

- **Day of week:** returns a positive integer from 1 to 7 that represents the day of the week of a specified time

```
5 := DAY OF WEEK OF 2011-08-27T13:20:00
1 := DAY OF WEEK OF now
```

Operators – Aggregation Operators I

- **Count:** returns the number of items of a list

```
4 := COUNT (12,13,14,null);
```

- **Exist:** returns true if there is at least one non-null item in a list

```
true := EXIST (12,13,14)
false := EXIST null
```

- **Average:** calculates the average of a number, time, or duration list

```
14 := AVERAGE (12,13,17)
04:10:00 := AVERAGE (03:10:00, 05:10:00)
```

- **Sum:** calculates the sum of a number or duration list

```
39 := SUM (12,13,14)
7 days := SUM (1 day, 6 days)
```

- **Median:** calculates the median value of a number, time, or duration list

```
13 := MEDIAN (12,17,13)
3 days := MEDIAN (1 hour, 3 days, 4 years)
```

Operators – Aggregation Operators II

- **Variance:** returns the sample variance of a numeric list

```
2.5 := VARIANCE (12,13,14,15,16)
```

- **Min, Max:** returns the smallest/largest value in a homogeneous list of an ordered type

```
14 := MAXIMUM (12,13,14)
```

- **Last, First:** returns the value at the end/beginning of a list

```
14 := LAST (12,13,14)
```

- **Latest, Earliest:** returns the value with the latest /earliest primary time in a list

- **Seqto:** generates a list of integers in ascending order

```
(2,3,4) := 2 SEQTO 4
(-3,-2,-1) := (-3) SEQTO (-1)
() := 4 SEQTO 2
```

- **Reverse:** generates a new list with the elements in the reverse order

```
(3,2,1) := reverse (1,2,3)
```

Operators – Aggregation Operators II – Example

```
// read all measured weights from the database
LET weights   BE READ {SELECT measured_weight FROM DB WHERE patID = patientID };

weight := latest of weights;
```

- After evaluating the **read statement**, the variable `weights` refers to a list containing all weights ever measured for the specific patient
- For calculating the BMI only the latest measured weight is relevant
- The **latest operator** extracts the weight with the latest primary time (each result item from the read statement has both a value and a primary time which denotes the time when the value was measured or inserted into the database)
- The latest weight is assigned to the variable `weight`

Operators – Time and Object Operators

- **Time:** returns the primary time of the provided parameter

```
2011-03-15T15:00:00 := TIME OF data0;
```

- **Attime:** constructs a time value from two time and time-of-day arguments

```
2011-06-20T15:00:00 := now ATTIME 15:00:00;
2001-01-01T14:30:00 := TIME OF intuitive_new_millenium ATTIME 14:30:00;
```

- **Clone:** returns a copy of its argument (used for objects)

```
2011-03-15T15:00:00 := CLONE OF 2011-03-15T15:00:00;
```

Operators – Time and Object Operators – Example

```
if (time of weight) is before (currenttime - 6 months) then
  conclude false; //no bmi calculation if the latest measure was 6 months ago
else
  conclude result.classification is present ; // if there is a classification, execute the action slot
endif;
```

- The condition of the If-Then-Else statement uses the **time operator** to access the primary time of the variable `weight`
- It is checked whenever the primary time is 6 months before the current system time
- If the primary time is not within the last 6 months the MLM concludes false

Call Statements

- **MLM calls:** when the MLM call statement is executed, the current MLM is interrupted, and the named MLM is called; parameters are passed to the named MLM

```
/* Define find_allergies MLM */
find_allergies := MLM 'find_allergies';
(allergens, reactions):= call find_allergies with patientID;
```

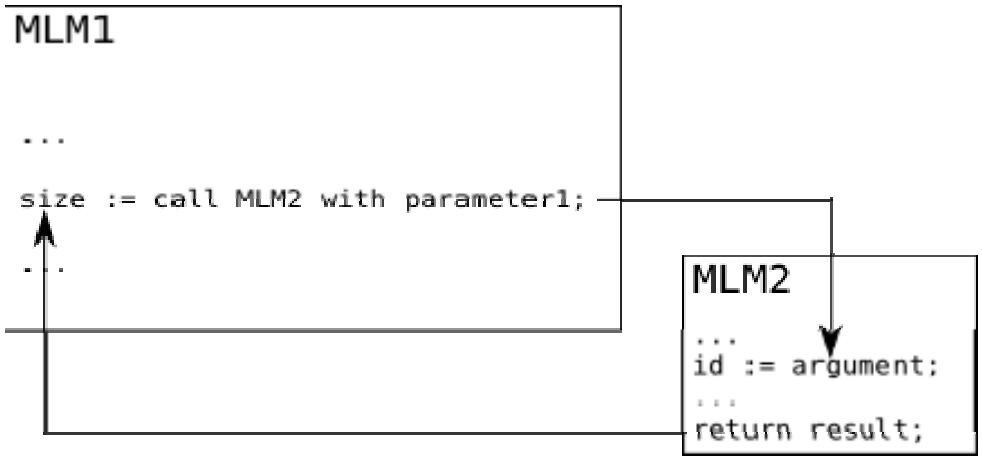
- **Event calls:** when the event call statement is executed, the current MLM is interrupted, and all the MLMs whose evoke slots refer to the named event are executed; parameters are passed to the named MLMs

```
allergy_found := EVENT {allergy found};
reactions := call allergy_found with allergy, patientID;
```

- **Interface calls:** when the interface call statement is executed, the current MLM is interrupted, and the interface is executed; parameters are passed to the interface

```
/* Define find_allergies external function*/
find_allergies := INTERFACE
{\\RuleServer\AllergyRules\my_institution\find_allergies.exe};
(allergens, reactions):= call find_allergies with patientID;
```

Nested MLMs



- MLM calls are used to externalize blocks of calculation which may be used be several MLMs or are also used in other knowledge bases
- The **call statement** in the MLM1 immediately invokes the MLM2 (the execution of the MLM1 suspends)
- The parameter (`parameter1`) is passed to the MLM2 and is accessed with the **argument expression**
- The passed parameter is assigned to the variable `id`
- When MLM2 is completed, the result of MLM2 is passed back to MLM1 and assigned to the variable `size`

Call Statements – Example

```
mlmForReadSize  := MLM 'read_Size_MLM';

size := call mlmForReadSize with patientID;
```

- The **MLM statement** assigns a reference to the MLM `read_Size_MLM` to the variable `mlmForReadSize`
- This variable is used in the **call statement** to call the MLM referred to
- The **call statement** passes the content of the variable `patientID` (the patient ID which is the context of the current MLM) to the MLM `read_Size_MLM`
- The execution of the current MLM is suspended while the called MLM is evaluated
- The return value of the called MLM is assigned to the variable `size`

Trigger

- **Simple Trigger:** a trigger statement specifies an event or a set of events; when any of the events occurs, the MLM is triggered; they occur only in the evoke slot

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

- **Delayed Trigger:** permits the MLM to be triggered some time after an event occurs

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

- **Constant Time Trigger:** permits the MLM to be triggered at a specific time

```
2011-01-01T00:00:00
```

- **Periodic Event Trigger:** permits the MLM to be triggered at specified time intervals after an event occurs

```
every 2 hours for 1 day starting today at 12:00 after time of event3
every 1 day for 14 days starting 2011-01-01T00:00:00
```

Trigger – Example

```
LET userEvent BE EVENT {getBMI};

evoke:
  userEvent;
;;
```

- The **event statement** assigns the reference to the event `getBMI` to the variable `userEvent`
- This variable is used in the **evoke slot**
- The MLM is triggered immediately after the event referred to occurs

```
evoke:
  MONDAY ATTIME 13:00 AFTER TIME OF userEvent;
;;
```

- If the evoke slot is changed to the above version, the MLM is triggered on the next Monday at 13:00 after the occurrence of the referred event

THE END