Arden Syntax – Basics

Karsten Fehre

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

Identifing an MLM

- An MLM can be identified by using the following three 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: body mass index;;
   mlmname: BMI;;
   arden: Version 2.7;;
   version: 1.00;;
   institution: Medexter Healthcare;;
   ...
```

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

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

Note: If there is more than one MLM with the same name and institution, the MLM with the latest version number is used. Also, if the remote MLM is from the same institution as the current MLM, it is not necessary to write this institution explicitly.

Data Types – Fundamentals I

- **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
null
```

- **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 hour 2 minutes 54.6 seconds
```

- **String**: Stream of characters

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

Data Types – Fundamentals II

- **List**: An ordered set of elements; each element can be an arbitrary data type (lists cannot contain lists as elements)

```
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 referring to specific days of the week; represented by constants or integer

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

Data Types – Primary Time

- In addition to its value part, each data value has a **primary time** part and a degree of applicability
- Primary time represents the value part's time of creation or measurement or examination or …
- By default, primary time is `null`
- Can be accessed using the `time` operator

```
2016-03-15T00:00:00 := time of laboratory_result
```

- Database query results should contain both, the value and the primary time
  - Might be the time when a blood test was drawn from the patient
  - Might be the time when a medication order was placed
  - Which time of a database entry is taken as primary time is left to the Arden Syntax implementer

Expressions – Fundamentals

- **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 that is explicitly represented is called a constant

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

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

```
let var1 be 0;
var2 := MLM 'BMI' from institution "Medexter";
var3 := var1 + 1;
```

- **Operator**: An expression may contain an operator and a number of sub-expressions called arguments

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

Statements – Fundamentals I

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

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

- **Write:** Sends texts or coded messages to a destination

```
email_dest := destination {kf@medexter.com};
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 another MLM

```
mlm2 := mlm 'my_mlm2.mlm' from institution "my institution";
include mlm2;
```

Statements – Fundamentals I – Examples

```
// MLM that contains the object definition of patient
mlmImport := mlm 'objectDefinition' from institution "Medexter";

// include
include mlmImport;
```

- The first statement is an **assignment**, assigning the reference to the MLM; in this case `objectDefinition` to the variable `mlmImport`
- The second one is an **include** statement that imports all object, MLM, event, interface, and resource definitions from the MLM `mlmImport` (`objectDefinition`)

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

- This **write** statement concatenates the calculated BMI and its classification to a string and sends this message to the default destination

Statements – Fundamentals II

- **Loops**
  - **While Loop:** Loops as long as 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;
FOR i IN list_of_values DO ... ENDDO;
```

- **Conclude**: Ends execution in the logic slot; if the conclude statement has a single `true` as argument, the action slot is executed immediately; otherwise the MLM terminates instantly
- **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)

Statements – Fundamentals II – Example

```
conclude result.classification is present; // if there is a classification
```

- **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 begins
- 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

```
call sampleMLM with var1;
let patientID be argument;
```

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

```
return result;
```

- **Return** statement that returns the object `result` to the calling instance (if the MLM is called from another MLM, it will be returned to the calling MLM)

Statements – If-Then-ElseIf

- **If-Then:** Permits conditional execution based on the value of an expression
  - There are three 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 – List Operators

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

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

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

```
/* data1 has data value 2 and primary time 2013-01-02T00:00:00, and data2 has data values 1 and 3
and primary times 2013-01-01T00:00:00 and 2013-01-03T00:00:00 */
(1, 2, 3) := data1 MERGE data2
null := (4,3) MERGE (2,1) // no primary time -> result is null
```

- **Sort:** Reorganizes 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 (20, 10, 30);
(30, 20, 10) := REVERSE (SORT DATA (20, 10, 30));
(30, 20, 10) := SORT TIME data3; /* assuming that data3 contains the values 10, 20, 30 with
primary times 2013-01-03T00:00:00, 2013-01-02T00:00:00  and 2013-01-01T00:00:00  */
```

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 the result is null

```
false := true AND false
null := true AND null
false := false AND null
0.4 := (0.5 AS TRUTH VALUE) AND (0.4 AS TRUTH VALUE)
```

- **Or:** Performs the logical disjunction of its two arguments; if any argument is true the result is true; if both arguments are false, the result is false; otherwise the result is null

```
true := true OR false
false := false OR false
true := true OR null
null := false OR null
null := false OR 3.4
0.5 := (0.5 AS TRUTH VALUE) OR (0.4 AS TRUTH VALUE)
```

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

```
true := NOT false
null := NOT null
0.8 := NOT (0.2 as TRUTH VALUE)
```

Operators – Comparison Operators

- **<,>,<=,=>,=, <>:** These operators have their common meaning; they can handle any data type; if one argument is null or types do not match, null is returned

- **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 under 19 years

Operators – String Operators I

- **Concatenation:** Converts its arguments into strings and concatenates them afterwards

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

- **Formatted with:** Formats a string with a given pattern

```
"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 I

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

- The **concatenation operator** concatenates the string the `bmi` field of the object `result` refers to with the string the `classification` field of the object `result` refers to 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 II

```
let the result.classification be localized ‘msg’;
```

- The **localized operator** will return the string that is assigned to the term `'over'` in the **resources category**
- The operator will obtain the string from the **language slot** that 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:** Determines if a string matches a pattern (similar to LIKE in SQL)

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

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

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

Operators – Arithmetic Operators

- **+, -, *, /, **:** Are used in their common meaning, except one argument is null or types do not match

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

- **Cosine, Sine:** Calculates the cosine/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 given 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 // Friday
1 := DAY OF WEEK OF now // in case the current day is Monday
```

Operators – Aggregation Operators I

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

```
var1 := (12,13,17);
3 := COUNT var1;
```

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

```
true := EXIST var1
false := EXIST null
```

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

```
14 := AVERAGE var1
04:10:00 := AVERAGE (03:10:00, 05:10:00)
```

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

```
42 := SUM var1
7 days := SUM (1 day, 6 days)
```

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

```
13 := MEDIAN var1
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 reverse order

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

Operators – Aggregation Operators II – Example

```
// read all measured weights from the data base
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 that denotes the time when the value was measured or inserted into the database)
- The latest weight is **assigned** to the variable `weight`

Operators – Time 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 (mostly used for objects)

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

Statements – 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;
```

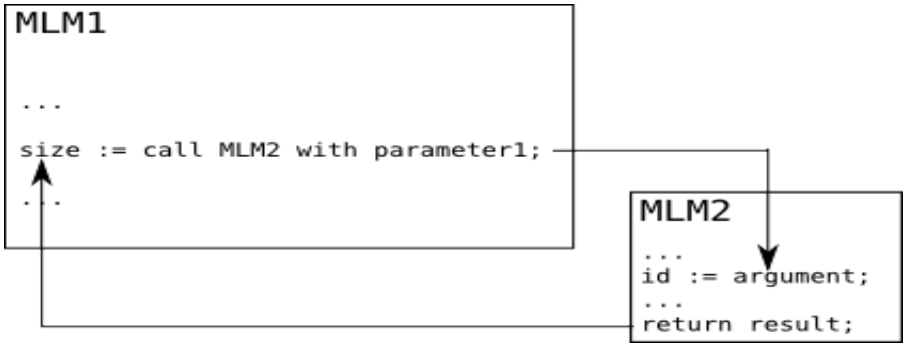
- **Event calls:** When the event call statement is executed, the current MLM is interrupted, and all the MLMs whose evoke slots refer to the named event are executed; parameters are passed to the named MLMs

```
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;
```

Statements – Call Statements – Nested MLMs



- MLM calls are used to externalize blocks of calculation which may be used by several MLMs or are additionally used in other knowledge bases
- The **call statement** in MLM1 immediately invokes MLM2 (the execution of MLM1 suspends)
- The parameter (`parameter1`) is passed to MLM2 and is accessed using 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` using the return statement

Statements – Call Statements – Example

```
mlmForReadSize := MLM ‘read_Size_MLM’;
size := call mlmForReadSize with patientID;
```

- The **MLM statement** assigns a reference pointing to the MLM `read_Size_MLM`, to the variable `mlmForReadSize`
- This variable is used in the **call statement** to call the referred MLM
- The **call statement** passes the content of the variable `patientID` (the patient ID that constitutes 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`

Statements – Triggers

- **Simple Trigger:** A trigger statement specifies an event or a set of events; as soon as any of the events occur, the MLM is triggered; they may only be used 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:** Allows the MLM to be triggered at a specific time

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

- **Periodic Event Trigger:** Allows the MLM to be triggered at specified time intervals after the occurrence of an event

```
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
```

Statements – Triggers – Example

```
Let userEvent be event {getBMI};

evoke:
     userEvent;
;;
```

- The **event statement** assigns the reference of the event getBMI to the variable userEvent
- This variable is used in the **evoke slot**
- The MLM is triggered immediately after the referred to event 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 following Monday at 13:00, after the occurrence of the referred to event

THE END