Arden Syntax –
An introduction (with fuzzy concepts)

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

- What is Arden Syntax?
- Arden Syntax – Fundamentals
- General MLM Layout
- Concepts: Data Types – Expressions – Statements – Operators
- Special Concepts: Curly Braces
- MLM to MLM Interaction
- Fuzzy Arden Syntax
What is Arden Syntax?

- ... a language used for representing and sharing medical knowledge.

- ... used for sharing of computerized health knowledge bases across personnel, information systems, and institutions.

- ... organized using modules, while each module, referred to as a Medical Logic Module (MLM), contains sufficient knowledge to make a single decision.

- ... an executable format which can be used by clinical decision support systems.
History

• A first draft of the standard was prepared at a meeting at the Arden Home-stead, New York, in 1989. Arden Syntax was previously adopted as a standard by the American Society for Testing and Materials (ASTM) as document E 1460, under subcommittee E31.15 Health Knowledge Representation.

• 1992: Arden Syntax version 1.0

• 1998: sponsorship moved to HL7 International (Arden Syntax Work Group)

• 1999: Arden Syntax version 2.0 adopted by HL7 and the American National Standards Institute (ANSI)

• 2014: Arden Syntax version 2.10
## History

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
<th>Important changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>2002</td>
<td>new string operators; reserved word “currenttime” returns the system time</td>
</tr>
<tr>
<td>2.5</td>
<td>2005</td>
<td>object capabilities: create and edit objects; XML representation of MLMs (except logic, action and data slot)</td>
</tr>
<tr>
<td>2.6</td>
<td>2007</td>
<td>UNICODE encoding; additional resources category to define text resources for specific languages; time-of-day and day-of-week data types; “localized” operator to access texts in specific languages</td>
</tr>
<tr>
<td>2.7</td>
<td>2008</td>
<td>enhanced assignment statement; extended “new” operator to allow easy and flexible object instantiation</td>
</tr>
<tr>
<td>2.8</td>
<td>2012</td>
<td>additional operators for list manipulation; operators to manipulate parts of given date and time values; switch statements; keyword “breakloop” for aborting a loop; number of editorial corrections</td>
</tr>
<tr>
<td>2.9</td>
<td>2013</td>
<td>fuzzification: fuzzy data types and fuzzy sets; adjustment of all available operators to be able to handle fuzzy data types</td>
</tr>
<tr>
<td>2.10</td>
<td>2014</td>
<td>XML representation of whole MLMs (including logic, action and data slot)</td>
</tr>
</tbody>
</table>
In Arden Syntax, medical knowledge is arranged within Medical Logic Modules (MLMs) each of which represents sufficient knowledge to make a single clinical 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.
MLMs are working in close contact with their 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 that are maintained by the host system
- **Output**: Analogous to the input parameter, data can be committed from the MLM to the host system after the execution of the MLM has finished

In order to start the execution of an MLM, an engine is needed that handles communication with the host system and can tell which of the MLMs are available

Ways to start running an MLM:
- **MLM call**: An MLM is directly called
- **Event call**: Any MLM that listens to a specific event is executed
Basic MLM Layout

- An MLM is composed of slots, grouped into the following **four** required **categories**: **maintenance**, **library**, **knowledge**, and **resources**
- Categories must appear in the correct (predefined) order
- Within each category is a set of **slots** that must appear in the correct order, too
- In general, an MLM is arranged such as:

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

- Contains slots that specify general information, unrelated to the MLM’s health knowledge
- These slots are used for MLM knowledge base maintenance and change control
- Contains information about the Arden Syntax version 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 the MLM
Library Category

- Contains the slots pertinent to knowledge base maintenance that are related to the MLM’s medical knowledge
- Slots provide health personnel with explanatory information as well as links to relevant health literature
- Slots
  - Purpose
  - Explanation
  - Keywords
  - Citations
  - Links
Knowledge Category

- Contains the slots that actually specify the MLM’s action

- This category’s slots define:
  - Terms used in the MLM (data slot)
  - The order of execution if more than one MLM is called (priority and urgency slot)
  - The context in which the MLM should be evoked (evoke slot)
  - The condition to be tested (logic slot)
  - The action to be taken if the condition is true (action slot)

- Slots:
  - **Type**: only “data-driven” available
  - **Data**: preparation, preprocessing, and query of data
  - **Priority**: defines the order of MLM evaluation
  - **Evoke**: checks if the MLM has to be executed if an event call occurs
  - **Logic**: contains the actual logic of the MLM
  - **Action**: is executed if logic slot concludes **true**
  - **Emergency**: defines the urgency of the action slot
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 that 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:

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

```plaintext
maintenance:
  title: simple body mass index;;
  mlmname: BMI_HowTo;;
  arden: Version 2.7;;
  version: 1.00;;
  institution: Medexter Healthcare GmbH;;
  author: Karsten Pehre;;
  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.)

知识:

类型: data_driven;
数据:

// 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.)

```r
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 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

```plaintext
maintenance:
  title: simple body mass index;;
  mlmname: BMI;;
  arden: Version 2.7;;
  version: 1.00;;
  institution: Medexter Healthcare GmbH;;
...
```

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

```plaintext
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. 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 an applicability

• Primary time represents the value part’s time of creation or measurement

• By default, primary time is **null**

• Can be accessed using the `time` operator
  
  \[2011-03-15T00:00:00 := 2 \text{ days AFTER} \ 2011-03-13T00:00:00\]

• 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 used Arden Syntax implementation
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 an other MLM

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

<table>
<thead>
<tr>
<th>If-Then:</th>
<th>If-Then-Else:</th>
<th>If-Then-ElseIf:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block1 is executed if condition is true</td>
<td>Block1 is executed if condition is true, otherwise (if condition is false or anything other than true) block2 is executed</td>
<td>Block1 is executed if condition1 is true, if condition2 is true block2 is executed, in all other cases block3 is executed</td>
</tr>
</tbody>
</table>

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

```
IF <cond1> THEN
      <block1>
ELSEIF <cond2> THEN
      <block2>
ELSE
      <block3>
ENDIF;
```
Statements – If-Then-ElseIf – Example

```plaintext
// 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 signifying the following:
  
  - 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 >=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
Statements – Object Statements

- **Object:** Assigns an object declaration to a variable (objects are the only data types in Arden Syntax that are first declared and then “instantiated”)
  
  ```
  MedicationDose := OBJECT [Medication, Dose, Status];
  ```

- **New:** Causes the creation of a new object (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. The dot operator is used to access the fields of an object
  
  ```
  "John" := patient.Name.FirstName;
  NameType := object [FirstName, LastName];
  /* Assume namelist contains a list of 2 NameType objects */
  ("John", "Paul") := namelist.FirstName;
  ```
Statements – Object Statements – Example

// object declaration
bmiResult := object [bmi, classification];
result := new bmiResult; // create an empty result object

• The first statement creates an object declaration with two fields and assigns this declaration to the variable bmiResult
• The second statement 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
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 */
  
  \[(1, 2, 3) := data1 \text{ MERGE data2}\]

  \text{null := (4,3) MERGE (2,1) // no primary time -> result is null}\n
- **Sort**: Reorganizes a list based on either the element values (keyword data) or the primary times (keyword time); default keyword is data

  \[(1, 2, 3, 3) := \text{SORT (1,3,2,3);}\]

  \[(10, 20, 30) := \text{SORT DATA (20, 10, 30);}\]

  \[(30, 20, 10) := \text{REVERSE (SORT DATA (20, 10, 30));}\]

  \[(30, 20, 10) := \text{SORT TIME data3; /* assuming that data3 contains the values 10, 20, 30 with primary times 2013-01-03T00:00:00, 2013-01-02T00:00:00 and 2013-01-01T00:00:00 */}\]
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

- \(<, \>, \leq, \geq, =, \neq\): These operators have their common meaning; these operators 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 strictly under 19
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 (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 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"
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 := "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

- **+, -, *, /, ****: 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

  /* 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
  
  \[ 2.5 := \text{VARIANCE} (12,13,14,15,16) \]

- **Min, Max**: Returns the smallest/largest value in a homogeneous list of an ordered type
  
  \[ 14 := \text{MAXIMUM} (12,13,14) \]

- **Last, First**: Returns the value at the end/beginning of a list
  
  \[ 14 := \text{LAST} (12,13,14) \]

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

- **Seqto**: Generates a list of integers in ascending order
  
  \[ (2,3,4) := 2 \ \text{SEQTO} \ 4 \]
  \[ (-3,-2,-1) := (-3) \ \text{SEQTO} \ (-1) \]
  \[ () := 4 \ \text{SEQTO} \ 2 \]

- **Reverse**: Generates a new list with the elements in reverse order
  
  \[ (3,2,1) := \text{reverse} (1,2,3) \]
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 and Object Operators

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

  \[ 2011-03-15T15:00:00 := \text{TIME OF} \text{ data0}; \]

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

  \[ 2011-06-20T15:00:00 := \text{now ATTIME} \text{ 15:00:00}; \]
  \[ 2001-01-01T14:30:00 := \text{TIME OF intuitive_new_millenium ATTIME} \text{ 14:30:00}; \]

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

  \[ 2011-03-15T15:00:00 := \text{CLONE OF} \text{ 2011-03-15T15:00:00}; \]
Operators – Time and Object Operators – Example

```c
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
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
EXAMPLE
Expressions – Curly Braces (Mapping Clauses)

- Are used in the data slot to signify institution-specific definitions such as database queries

  - **Read statement:** Reads data from the host system
    
    \[
    \text{var1} := \text{READ } \{ \text{select potassium from results where specimen = 'serum'} \};
    \]

  - **Event statement:** Defines an event; an event can be used to call MLMs
    
    \[
    \text{event1} := \text{EVENT } \{ \text{storage of serum potassium} \};
    \]

  - **Message statement:** Text that is used by write statements
    
    \[
    \text{message1} := \text{MESSAGE } \{ \text{increased body temperature} \};
    \]

  - **Destination statement:** Target that is used by write statements
    
    \[
    \text{destination1} := \text{DESTINATION } \{ \text{email: user@cuasdf.bitnet} \};
    \]

  - **Interface statement:** function that is evaluated by host system
    
    \[
    \text{func\_drugint} := \text{INTERFACE } \{ \text{char* API:Interaction (char*, char*) } \};
    \]
Expressions – Curly Braces (Mapping Clauses) – Example

// read all measured weights from the data base
Let weights be read \{select measured_weights from DB where patID = patientID\};

- This **assignment** statement assigns the result of the read statement (using mapping clause "SELECT measured_weight FROM DB WHERE patID = patientID") to the variable *weights*
- *patientID* is a variable that contains the patient ID currently in use 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
- The content of the **curly brace** must be evaluated by the host system and its syntax is not part of the Arden Syntax

Let userEvent be event \{getBMI\};

- **Assignment** statement that assigns the event *getBMI* to the variable *userEvent*

  Evoke: userEvent;;

- If the event variable is used in the **evoke slot**, the MLM is always executed, when this event occurs
EXAMPLE
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 event occurs

```
evoke:
        Monday at time 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
EXAMPLE
Fuzzy Sets – Background I

- Function that maps a given data value to a truth value between 0 and 1
- A fuzzy set represents a linguistic/clinical concept with fuzzy (non-sharp) boundaries

\[ x \text{ is in } \text{BMI.normal} \]
Fuzzy Sets – Background II

- **Crisp** border
  - Defines a **sharp** border
  - Checking if a given measurement is greater or less than the defined crisp border results in either true or false
  - Borderline cases are not detected

- **Fuzzified** border
  - Defines a **gradual** border
  - Checking if a given measurement is greater or less than the defined fuzzified border results in a truth value between 0 and 1
  - Borderline cases are detected
  - Weighted results for borderline cases, all other are as usual
Fuzzy Sets – Example I

- **Usual** Arden Syntax

  fever_limit := 38;
  temperature := 37.9;

  message := "patient has no fever";
  IF temperature > fever_limit THEN
    message := "patient has fever"
  END IF

  - Result message: “patient has no fever”
  - Borderline case is not detected

- **Fuzzy** Arden Syntax

  fever_limit := FUZZY SET (37.5, 0), (38, 1);
  temperature := 37.9;

  message := "patient has no fever";
  IF temperature > fever_limit THEN
    message := "patient has fever"
  END IF

  - Result message: “patient has fever” (with applicability 0.8)
**Fuzzy Sets – Example II**

- **Usual Arden Syntax**

  ```
  fever_border := 38;
  sub_border := 37.5;
  temperature := 37.9;
  
  message := "patient has no fever";
  
  IF temperature > fever_border THEN
    message := "patient has fever";
    app := 1;
  ELSE IF temperature > sub_border THEN
    message := "patient has fever";
    app := (temperature - sub_border)/0.5;
  END IF
  
  - Variable message contains the string "patient has fever"
  - Applicability (variable app) is the truth value 0.8
  ```

- **Fuzzy Arden Syntax**

  ```
  fever_border := FUZZY SET (37.5,0), (38,1);
  temperature := 37.9;
  message := "patient has no fever";
  
  IF temperature > fever_border THEN
    message := "patient has fever";
  END IF
  
  app := applicability of message;
  
  - Variable message contains the string "patient has fever"
  - Applicability (variable app) is the truth value 0.8
  ```
Data Types – Fuzzy Sets

- Definition of a fuzzy set
  \[ \text{Fuzzyset}_u := \text{FUZZY SET} \ (18.5,0), \ (19.5,1), \ (24,1), \ (25,0); \]
  \[ \text{Fuzzyset}_v := 7 \text{ fuzzified by 2}; \]

- Fuzzy set based on other data types
  \[ \text{Fuzzyset}_{\text{duration}} := \text{FUZZY SET} \ (3 \text{ days},0), \ (10 \text{ days},1), \ (20 \text{ days},1), \ (25 \text{ days},0); \]
  \[ \text{simple} := 2009-10-10 \text{ fuzzified by 12 hours}; \]
  \[ \text{complex} := \text{FUZZY SET} \ (2009-10-10,0), \ (2009-10-11,1), \ (2009-11-10,1), \ (2009-11-11,0); \]
Applicability

- Arden Syntax contains two types of fuzziness:
  - Data types: for explicit calculations e.g., truth value, fuzzy set
  - Applicability: for weighting MLM evaluation and weighting of branches

- All simple data types are endowed with additional information concerning 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

- Default applicability is 1 and the applicability is never null

- Can be accessed using the applicability operator

- If-then statements with a condition that evaluates to a truth value [0,1] result in a split of the MLM execution
  - Each branch will be executed under corresponding applicability
  - The applicability is implicit attached to each variable of the branch
Data Types – Truth Value

- Generalization of the Boolean data type
- Value between 0 and 1
- Boolean value true is equal to the truth value 1 and the Boolean value false is equal to the truth value 0
- May be the result of mapping a clinical value to a fuzzy set
- Can also be defined explicitly
Data Types – Linguistic Variable

- Construct to represent a linguistic concept and its sub-concepts
- Subsumes the sub-concepts of a concept under one term
- Definition of a linguistic variable

```
- data:
  simpleBMI := LINGUISTIC VARIABLE [underweight, normal, overweight];

- logic:
  BMI := new simpleBMI;
  BMI.underweight := FUZZY SET (18.5,1), (19.5,0);
  BMI.normal := FUZZY SET (18.5,0), (19.5,1), (24,1), (25,0);
  BMI.overweight := FUZZY SET (24,0), (25,1);
```
Statements – If-Then-Else – Fuzzy Condition

- If the used condition in an **If-Then-ElseIf** statement evaluates to a truth value between 0 and 1, both blocks are executed.

- Each branch is provided with its own set of variables which are duplicated accordingly.

- The **degree of applicability** of each variable in the if-block is multiplied with the truth value of the condition.

- In the else-block the applicability of each variable is multiplied with 1 minus the truth value of the condition.

- The general applicabilities of these blocks are called relative weights.

- The weight of an MLM evaluation is 1, as long as it does not split.

- The program may branch several times.

- If the branches are not subsumed using the **aggregate** keyword, the branches are executed in parallel and the MLM will finish with 2 or more return values (with different applicabilities).
Statements – If-Then-ElseIf – Fuzzy Condition – Example

Source

maintenance: [...] 
knowledge: [...] 
logic:
  //define linguistic variable
  //BMI as above
  [...] 
  myBMI := 24.8;
  x := myBMI <= BMI.overweight;
  if x then
    // this branch is executed
    // with applicability 0.8
    <then_block>
  else
    // this branch is executed
    // with applicability 0.2
    <else_block>
  endif;
  [...] 
end:

Arden Syntax

Fuzzy Arden Syntax
Statements – If-Then-Aggregate

```plaintext
if x then
  <then_block>
else
  <else_block>
endif AGGREGATE;
```

- Combination of the variable values in each execution branch according to their applicability

- Aggregations are common in fuzzy control
Operators – Comparison Operators – Fuzzy Comparison

- The behavior of the comparison operators is different to the standard case when a crisp value is compared to a fuzzy set

- \textit{x is in BMI.normal}
  Returns the truth value \( u(x) \) to that the crisp value \( x \) is mapped by the fuzzy set

- \textit{x <= BMI.normal}
  Returns the maximum of the mappings of all \( y \geq x \) (green shape)

- \textit{x >= BMI.normal}
  Returns the maximum of the mappings of all \( y \leq x \) (blue shape)
EXAMPLE
ArdenML: Objectives and applications

• Provide a complete XML schema for version 2.10 of the Arden Syntax to express MLMs in XML

• Thus, Arden Syntax is now compatible with all other HL7 standards based on XML (HL7 version 3, VmR, and others)

• Further benefit: To be able to use available XML tools
ArdenML: Example

```xml
<Library>
  <Purpose>Test</Purpose>
  <Explanation></Explanation>
  <Keywords></Keywords>
</Library>
<Knowledge>
  <Type>data driven</Type>
  <Data></Data>
  <Evoke></Evoke>
  <Logic>
    <Assignment>
      <Identifier var='var1' />
      <Assign>
        <Value otype='time'>1990-03-15T15:00:00</Value>
      </Assign>
    </Assignment>
    <Assignment>
      <Identifier var='res1' />
      <Assign>
        <ReplaceYearWith
          <Identifier var='var1' />
          <Value otype='number'>2011</Value>
        </ReplaceYearWith>
      </Assign>
    </Assignment>
    <Assignment>
      <Identifier var='res2' />
      <Assign>
        <ReplaceYearWith
          <Identifier var='var1' />
          <List>
            <Value otype='number'>2011</Value>
            <Value otype='number'>2010</Value>
          </List>
        </ReplaceYearWith>
      </Assign>
    </Assignment>
  </Logic>
</Knowledge>
```
Cross compilation/transformation of Arden Syntax to/from ArdenML

Team effort by Intermountain Hospitals, Salt Lake City, Utah, U.S.A., and Medexter Healthcare, Vienna, Austria
Ongoing effort for transformation to/from HeD*

*HeD (Health electronic decisions)—Clinical decision support within Meaningful Use, stage 3, U.S.A.
THE END