ESKORT Designer for Clearance

User Guide, Section IV - Formalization Language

icdk/Application/Designer/UserGuide/4/v1.11/2011.01.24

DOCUMENT

Title:	User Guide, Section IV - Formalization Language
Document:	icdk/Application/Designer/UserGuide/4
Date:	2011.01.24
Version:	1.11
Author:	Marco Dijkstra, Jørgen Rune Mortensen
Contributions by:	
Classification:	Commercial in Confidence
Distribution:	

Versions:	1.0	Initial Version
	1.1	Footer changed to Intracom
	1.2	Added Parameter TableUsage functions
	1.3	Aligned to Intracom Standard Updated function Reference
	1.4	Updated Function Reference
	1.5	Updated Function Reference Minor editorials corrected
	1.6	Updated Function Reference
	1.7	Updated MDC Language Definition
	1.8	Fixed syntax error in example using case
	1.9	Removed functions from Clearance which are not related to Clearance
	1.10	Added section describing ADM.
	1.11	Updated Function List according to documentation in Clearance Workbench

Printed: 16.05.12

Table of Contents

°LastInfoPage

1. Overview

This document is aimed at establishing a basic understanding of the rule formalization language.

1.1 Conventions

Definitions – The first time a specific term or concept is mentioned in a relevant subsection it is printed in italic. If required, further definition will follow shortly after the first time it is mentioned and is preceded by an icon.

Exercise – Exercises are printed in bold and are preceded by an icon.

[Menu] – Menu options, buttons, window and field names are printed within square brackets. The word option, button, field, dialog/window behind the word will specify the type.

2. Understanding Grammars

In the following sections, the syntax of the formalization language is presented in the so-called *Backus-Naur Form (BNF)* notation. Before looking at the formalization language itself, this section explains how to read this notation.

2.1 BNF Notation by Example

The BNF notation was developed to specify the syntax of a programming language in a precise, unambiguous and exhaustive form, called a *grammar* for the language. This is a pre-requisite for verification of a program to be done by a computer, and can help a user understand the possibilities provided by a language – in this case the rule formalization language.

BNF is a style of specification used for formal syntax descriptions

Although you may not be aware of it, you are probably familiar with the concept of a grammar from things you work with naturally on a daily basis.

To illustrate, as a user of a spreadsheet product – such as Microsoft Excel – you must comply with a specific grammar when specifying formulas for calculated cells.

You may think of it in terms such as:

> *In a formula I can add, subtract, multiply and divide using constants or values from other cells.*

This informal "specification", can be presented more precisely in BNF notations as follows:

Start Symbol

<expression>

Terminal Symbols

integer denotes a positive integer, e.g. 5 or 10000.

cell-address denotes any character between 'A' and 'Z', followed by a positive integer, e.g. A1 or B10.

Production Rules

```
<expression>          ::=   <expression> <operator> <expression>
                            | integer
                            | cell-address

<operator>            ::=   *
                            | /
                            | +
                            | -
```

In addition to the *start symbol* which specifies what is defined by the grammar (in this case an expression), the specification comprises the definition of a set of *terminal symbols* (or just terminals), and a list of *production rules (*or just productions*)*, specifying how a valid formula (expression) can be produced[1].

Each production rule comprises a *non-terminal symbol* (to the left of '::='), and a specification (to the right of '::=') of how the symbol can be composed. The production rule for <expression> shows three alternative ways of composing an expression. Note, that alternative productions are separated by '|', and typically written on separate lines.

The first of these alternatives could be read in the following manner:

> *One way to compose an expression is by concatenating an expression, an operator, and another expression.*

The other two alternatives could be read as follows:

> *One way to compose an expression is to write an integer.*
>
> *One way to compose an expression is to write a cell-address.*

Naturally, you will ask: how then do I compose an operator? To find the answer to this question all you need to do is to find the production rule that defines <operator>.

This is the second production rule in the grammar. As you can see, this production rule lists four alternatives, the first of which could be read in the following manner:

> *One way to compose an operator is to write the keyword* *******.

The remaining alternatives indicate that you can also use the keywords /, + and – as operators.

Similarly, you can ask: how do I compose a cell-address? As you can see, the word cell-address is underlined, indicating that it is a terminal symbol. Looking under terminal symbols, you can find that cell-address is defined to be any character between 'A' and 'Z', followed by a positive integer.

Using the production rules of the grammar, we can now produce expressions like:

- 5
- 2 + 2
- A1
- A1 + 3
- 2 * A1 + 3

Note that in this simple grammar example, the following examples are **not valid**:

- -5
- a1
- 3 * (2 + 1)

2.1.1 Exercise

Consider why the last three examples are not valid, in this grammar[2].

[1] Note: to keep the example simple, we have assumed that all values are integers, and that columns are referenced by a single character.

Consider what changes would be required in the grammar to make the expressions valid.

2.1.2 Summary

To summarize, when describing a grammar in BNF notation, the following elements are involved:

1) A *start symbol* designating the non-terminal symbol defined through the grammar.
2) A list of *terminal symbols* defining how of the grammar.
3) A list of *production rules* defining how each *non-terminal symbol* can be composed from *keywords*, terminal symbols and non-terminal symbols.

2.2 Notation

In the above example, and the remainder of this document, the following notation is used:

- Terminal symbols are written underlined (e.g. character or integer).
- Keywords are written in bold (e.g. **within** or **if**)[3].
- Non-terminal symbols are written enclosed in '<' and '>' (e.g. <rulebody> or <action>).
- Alternatives in a production are separated by '|'.
- In some instances it is possible to compose a symbol by writing nothing. To avoid confusion, the special notation 'empty' is used in these instances.

  The definition of <formalization> illustrates this:

  ```
  <formalization>    ::=      'empty'
                              | <area_head_members>  <areamembers>
  ```

  The availability of the first alternative allows a formalization list to be empty.
- In the BNF fragments, presented in the following sections, the notation '…' is used to indicate that other alternatives than the one presented are available, as illustrated in the example below.

  ```
  <area_head_member>          ::= …
                              |  let  <letbody>
  ```

  Generally, when fragments are presented, there will be non-terminals for which the reader is expected to consult the grammar overview in Appendix A, for a detailed definition.

2.3 Additional Resources

Please refer to 'Appendix E – Grammar Examples' for additional examples explaining BNF both in terms of simple general examples, and in terms of selected constructs of the formalization language.

[2] You can find answers to these exercises in 'Appendix E – Grammar Resources'.
[3] Terminal symbols are optionally quoted so that any character, including one used in BNF, can be efined as a terminal symbol.

3. Rule Formalization

Rules need to be formalized using the formalization language to be evaluated by the risk analysis server.

Formalizing rules is a specialized task, which requires knowledge of the following aspects (in addition to general knowledge about the relevant ESKORT applications):

- The (internal) multi dimensional representation of taxpayer data (once it has been extracted from the data warehouse)
- Formalization language (especially regarding how data is being accessed)
- The context (environment) in which the knowledge base is being developed (cubes and their dimensions).

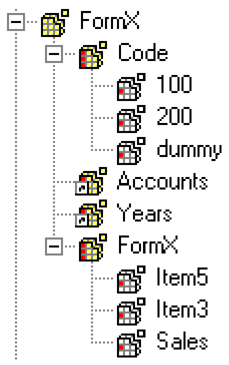
The following sections focus on the formalization language.

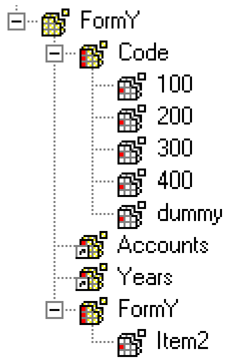
Note that it is important to have read (and understood) the main concepts (such as environment, knowledge base, cube, member, measure and tuple) described in [Introduction to Environment] and [Introduction to Knowledge Engineering], before reading these sections.

Please refer to Appendix A: MDC Language Definition for a complete overview of the grammar in BNF notation.

3.1.1 Sample environment and formalization

In the following section all examples will be based on the generic cubes FormX and FormY.





Accounts and Years are shared dimensions.

3.1.2 Terminal Symbols

The MDC grammar is based on the following special terminal symbols:

<u>identifier</u>

> *Any sequence of alpha-numeric characters starting with a non-numeric character (e.g. MyName, First_Name).*

<u>string</u>

Any sequence of alpha-numeric characters delimited with "" characters (e.g. "This is a string", or "This is \"a string\"").

unsigned-integer

An unsigned integer number constant (e.g. MyName, First_Name).

real

A decimal number constant (e.g. 1000000 or 0.5).

3.1.3 Sample Rule

The following rule will be used throughout this section to illustrate various aspects of the formalization language:

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

3.2 Specifying Ranges

The default cube, and the ranges applicable for the rules within an area, may be specified in the beginning of an area block using the following syntax:

```
<area_head_member>  ::=   within  <withinbody>
                          | …

<withinbody>        ::=   (  <cubenames>  )  for  <forbody>

<forbody>           ::=   (  <sets>  )
                          | FOR_ALL_KNOWN_CELLS
```

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The <cubename> specified in the within clause is used as the *default cubename* on all unqualified measures (all measures for which a cubename is not explicitly stated).

The part for (<sets>) specifies the *ranges* for the evaluation of the rule. The ranges are specified as a comma-separated sequence of tuples or set-value expressions.

Alternatively, the keyword **FOR_ALL_KNOWN_CELLS** can be specified to indicate that the rule should be evaluated for all cells in the cube, for which a value is known. This construct can be especially useful in very sparsely populated cubes.

Ranges specify the tuples that will form the evaluation contexts for the rule within the area.

Note: All dimensions except the measure dimension from the default cube must be in the ranges.

3.2.1 Example of Ranges

To illustrate, if a taxpayer had one account (A) and three years of data (1999, 2000 and 2001), then the set of tuples would be:

{‘A’,’1999’}

{‘A’,’2000’}

{‘A’,’2001’}

```
within ( FormX ) for ( Account, Years )
```

The rule will be evaluated for each tuple, which is said to be the evaluation context for the evaluation in question. I.e. the rule is evaluated for all Accounts and all Years.

```
within ( FormX ) for ( Account, {Years.’2000’} )
```

The rule will be evaluated only for the tuple {‘A’,’2000’}. I.e. the rule is evaluated for all Accounts but only the specific member ‘2000’ from the Years dimension.

```
within ( FormX ) for ( Account, {Years.LASTMEMBER} )
```

The rule will be evaluated only for the tuple {‘A’,’2001’}. I.e. the rule is evaluated for all Accounts and the last member of the Years dimension.

3.2.2 Using a ‘dummy’ Member in Ranges

Sometimes we would like not to evaluate a rule for all members in a given dimension, but on the other hand, we do not want to explicitly bind the dimension to a given member either. Rather we would like to bind the dimension to a so-called ‘dummy’ member and then refer to the correct member(s) in the rule body.

As an example consider the cube with the dimensions:

Account, Years, Code and a measure dimension.

In a rule we would like to test that the measure ‘Sales’ for the code ‘100’ is the same as code ‘200’. We would like to test the rule for all accounts and all years.

In this case it is not correct to write:

```
within ( FormX ) for ( Account, Years, Code )
```

since the rule then will be evaluated for all codes.

We could implement the rule as follows:

```
within ( FormX ) for ( Account, Years, {Code.’100’} )
rule
  if
    Sales = {Code.’200’, Sales}
  then
    Call Observation(…).
```

Here the unqualified Sales measure refers to the one bound in the Ranges (i.e. where Account is bound to the account currently evaluated, Years is bound to the year currently evaluated and Code is bound to code ‘100’). When we want to compare it with the value of Sales from code ‘200’ we need to bind the Code dimension to ‘200’ instead of the value (‘100’) bound in the range. Therefore we need the tuple:

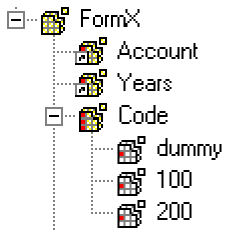
```
{Code.’200’, Sales}
```

The above implementation of the rule is valid and will do the job. Some people however find it a bit confusing that the Sales measure at one place in unqualified and not at other places. In such cases one can consider to create a ‘dummy’ member in the Code dimension and then use this in the ranges. All references to measures in the rule body will then be tuples stating the exact code.

```
within ( FormX ) for ( Account, Years, {Code.dummy} )
rule
  if
    {Code.’100’, Sales} = {Code.’200’, Sales}
```

```
then
  call Observation(...).
```

The various codes together with the dummy member must be defined in the environment:



For more information about environments refer to [ESKORT Designer, Section II – Introduction to Environments].

3.3 Defining Derived Members

A derived member is a member for which data has not been 'measured' but for which a value can be determined through calculation.

Derived members are defined using the let construct in the formalization language.

```
<area_head_member>  ::=   ...
                          | let  <letbody>

<letbody>           ::=   <new_member>  =  <value_expression>
                          | member <new_member>  =  <value_expression>
                          | double <new_member>  =  <value_expression>
                          | integer <new_member>  =  <value_expression>
                          | string <new_member>  =  <value_expression>
                          | boolean <new_member>  =  <search_condition>
                          | set <new_member>  =  <set>
```

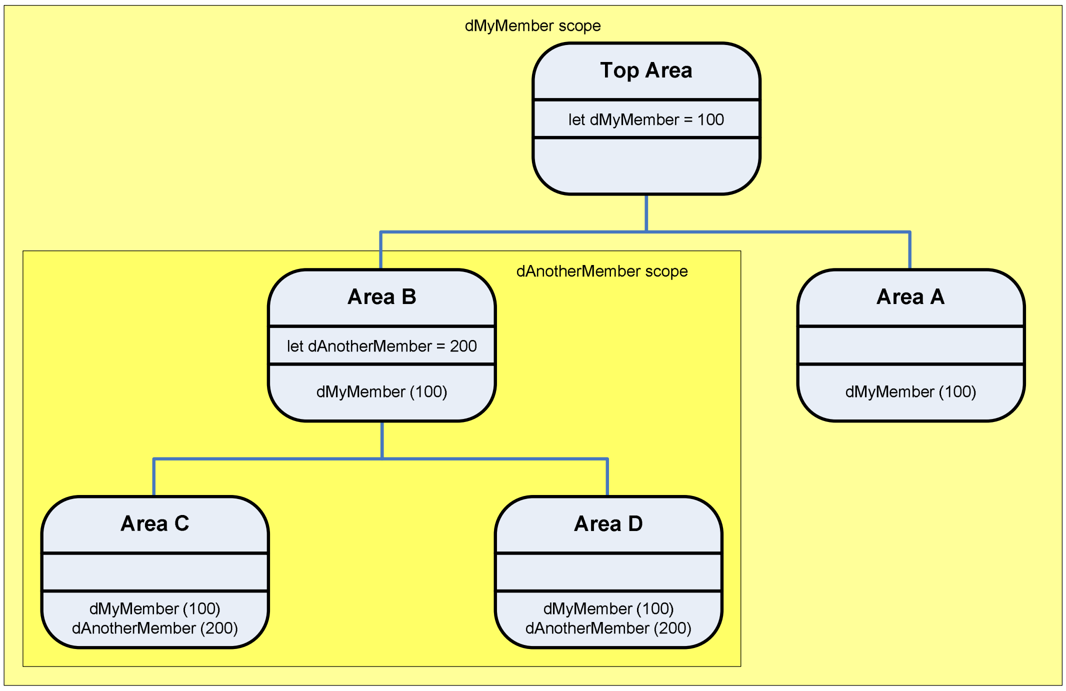
```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The rules in the BNF grammar show us that we can add (+), subtract (-), multiply (*) and divide (/). It is also correct to use functions when defining derived members.

3.3.1 Scope of Derived Members

Derived members can be defined in both area and rule formalization. If derived members are defined in an area, all child areas and child rules will be able to access them.

See the following example:



Because *dMyMember* has been defined in the top area, it is accessible in all child areas. This does not count for *dAnotherMember,* which is only accessible in the child areas 'B', 'C' and 'D'.

3.3.2 Type of Derived Members

Just like the members corresponding to data that is downloaded, derived members constructed via the let construct can have different types. When you use the plain let clause – with no type or with the **member** type indication – the member is assumed to be a double.

You can specify which type of member you want, by including one of the type keywords (double, integer, string, boolean), see the following examples:

```
let string start = SubStr(MEMBER_NAME(CalendarYears.CURRENTMEMBER), 1, 8)


let string end = SubStr(MEMBER_NAME(CalendarYears.CURRENTMEMBER), 10, 8)


let boolean PPDecrease = (PP < ValPrevPeriod(PP,1))
```

The types double, integer and string – corresponds to the member data types described in [ESKORT Designer, Section II – Introduction to Environment] section Creating Members.

3.3.2.1 Boolean Members

The type boolean can be used for members that evaluate to true or false. This allows you to define part of a condition as a boolean member.

To illustrate, you could define a member:

```
let boolean IsLarge = (Turnover > 1000000) and (NumberOfEmployees > 50)
```

and then refer to it in the rule condition

```
if
  IsLarge = true
  and …
```

3.3.2.2 Sets

The type set allows you to define a set of members, that can be used in places where you are required to provide a set, e.g. in the for clause of a rule.

To illustrate, you could define a set:

```
let set LastCalendarYears =
   {{CalendarYears.'20010101-20011231'}, {CalendarYears.'20020101-20021231'}}
```

and then use it in the for clause of a rule:

```
within (BasicInfoYear) for (Accounts, LastCalendarYears)
```

The let set construct is often used in conjunction with the KbInclude mechanism. See [ESKORT Designer, Section V – Defining and Releasing Tasks] for more information about KbIncludes.

3.3.3 Derived members in the Ranges

Derived members can also be used in the ranges of the rule:

```
within ( FormX ) for ( Accounts, { Years.YearsTotal } )
```

The derived member YearsTotal of the dimension Years must therefore be defined in a parent area to the rule.

E.g. In the top area of the knowledge base it is defined as:

```
let Years.YearsTotal = SUM( Years )
```

3.3.4 Anonymous Derived Members

By using Anonymous Derived Members (ADM) it is possible to use derived members directly in coordinates without first specifying any let-statement. An example of this is shown below.

```
if ( { IsPackages::SUM( IsPackages ), SadIsPackages:gross_weight } > 1000 ) then
….
```

The syntax used in the coordinate above is a merging of the statements below:

```
let x = SUM( IsPackages )
let total_gross_weight = { x, SadIsPackages:gross_weight }

…

if( total_gross_weight > 1000) then
…
```

The difference between the two examples is that in the last example a temporary name (**x**) is used to define a derived member which is the SUM of the IsPackages and **total_gross_weight** is used to define a derived member which is the SUM of all gross weights; whereas the first example defines the SUM on the gross weights in one line which omits the temporary names, hence it is defined using anonymous derived members.

It is also possible to specify ADM without using aggregation functions such as SUM. An example of this is shown below.

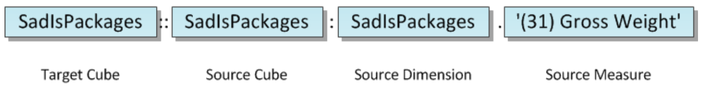
```
if ( { IsPackages::SUM( IsPackages ), iif( SadIsPackages:gross_weight > 1000, 1, 0 ) } > 5 ) then
….
```

In the example above the rule evaluates to true if there are more than 5 IsPackages with a gross weight more than 1000kg. The coordinate in the example above is composed by two ADMs – one defining the SUM and another defining the condition for the IsPackages which shall be counted.

3.3.4.1 Syntax

The syntax when using ADM introduces double colons, which is used to specify the contexts on which the ADM is defined.

The figure below shows the basic syntax of ADM.

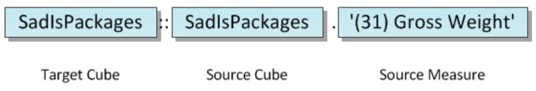


When making a derived member the part before the double colon (::) defines the target cube. The target cube is the cube which will be extended by the new, derived member.

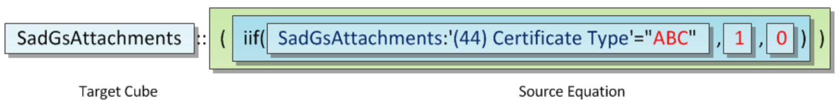
The subsequent parts are used for specifying the measure which is included in the calculations.

The part before the single colon (:) defines the name of the source cube, the part before the dot (.) specifies the dimension in the source cube (typically the measure dimension) and finally the part after the dot (.) specifies the name of the measure.

In most situations you do need to specify the complete qualified source, e.g. the name of the measure dimension shall always the same name as the cube. The system recognizes this; hence the specification can be reduced to the form shown below.



As seen in at the figure the target cube can be put in from of an equation rather than directly in front of a measure. This is used to indicate that the coordinate value that it represents is in the specific cube. This is shown in the figure below.


SadGsAttachments
::
(iif(SadGsAttachments:'(44) Certificate Type'="ABC" , 1 , 0))
Target Cube
Source Equation

3.4 Defining Filters

Filters applicable to an area may be specified in the beginning of an area block. A filter clause ensures that the rules within the area are only evaluated when the taxpayer data fulfils the filter expression (e.g. that the taxpayer is registered as having one of the stated business types or industry codes).

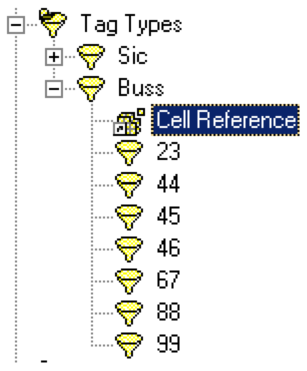
Filters are specified using the following syntax:

```
<area_head_member>  ::=   …
                          | filter  <filterbody>

<filterbody>        ::=   <filter_id>  {  <filter_set>  }
```

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The filter used in the rule above implies that the rule will only be evaluated for the business types 23, 45, 67 and 88.

Note that filters must be defined in the environment before they can be used. Filter values can either be stated explicitly or be available in a separate database. Refer to [ESKORT Designer, Section II – Introduction to Environments] for more information.



3.5 Defining Rules

Rules are defined using the rule definition constructs of the formalization language.

```
<areamember>        ::=  rule  <rulescope>
                         | …

<rulescope>         ::=  <rulebody>  <end_rule>
                         | <start_named_rule>  <rulebody>  <end_rule>

<start_named_rule>  ::=  <rule_identifier>

<rule_identifier>   ::=  identifier

<end_rule>          ::=  .

<rulebody>          ::=  if  <search_condition>  then  <action_list>
```

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

Note that according to the BNF grammar it is valid to have an unnamed rule (i.e. the rule name is not stated after the **rule** keyword):

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

Consider the following rules from the BNF grammar:

```
<action_list>          ::=   'empty'
                             | <action>
                             | <action_list>  ;  <action>

<action>               ::=   call  <oleidentifier>  (  <param_list>  )
```

The production rules tell us that the following rule would be valid as well (as far as the syntax is concerned):

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

      (dMyMember > 10000) and Item5=1

then

      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance);
      Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

In this case, two observations are created if the condition in the rule is met. However, this is an odd example since it would create two identical observations. In a typical situation, each observation would apply to a different profile, with different measures.

The following rules explain the non-terminal symbol <search_condition>, which for example reflects the syntax of the (important) condition of the rule:

```
<search_condition>    ::=   <boolean_term>
                            | <search_condition>  or  <boolean_term>

<boolean_term>        ::=   <boolean_factor>
                            | <boolean_term>  and  <boolean_factor>

<boolean_factor>      ::=   <boolean_primary>
                            | not  <boolean_primary>

<boolean_primary>     ::=   <value_expression>  <comp_op>  <value_expression>
                            | (  <search_condition>  )
                            | <function_identifier>  (  <param_list>  )  <function_end>

<comp_op>             ::=   <>
                            | >
                            | <
                            | >=
                            | <=
```

The rules allow complex conditions. The grammar allows use of operators including =, <>, >, <, >=, <= as well as ‘and’, ‘or’ and ‘not’**.** It also allows use of functions.

See the following example:

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

In this example the rule creates an observation if dMyMember (as defined as a derived member) is greater than 10000 and if Item5 (of cube FormX, because this is the default cube of the rule) is equal to 1.

3.5.1 Using Measures from more than one Cube

All unqualified measures (all measures for which a cubename is not explicitly stated) in a rule refer to measures in the default cubename.

It is possible to refer to measures from other cubes than the default cube. If this is the case the cubename must be stated:

```
<cubename> : <identifier>
```

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

As seen in the example dMyMember is the sum of Item3 from default cube (i.e. FormX) and Item2 from FormY. References to other cubes can of course also be done in the condition part of the rule.

When referring to measures from another cube in this simple way one must ensure that the cube has dimensions that match the default cube. This will be the case if the cubes have the same dimensions except the measure dimension, of if the cubes have dimensions that can be extended into each other.

If the cube that is not the default cube has other dimension that is not covered of the above, these dimensions must be bound in a tuple, see section 3.5.2.

Example:

Dimension of default cube	**Dimensions of other cube**	**Use Cube:Measure?**
Accounts, Years	Accounts, Years	Yes
Accounts, Years	Accounts, Months	Yes, if dimension Months has attribute BuildExtender = 'yes' and measure dimensions has attribute ExtendControl

		other than 'no_control'. Typically one would set the ExtendControl to 'aggregate' and the monthly values from the second dimension are then added together.
Accounts, Months	Account, Years	Yes, if dimension Months has attribute BuildExtender = 'yes' and measure dimensions has attribute ExtendControl other than 'no_control'. Typically one would set the ExtendControl to 'aggregate' and the yearly value from the second dimension are then split into monthly values.
Accounts, Years	Accounts	Yes. Even though the other cube does not have the Years dimension, the dimensions of the other cube still match the default cube
Accounts	Account, Years	No The dimension Years is not bound in the other cube, and this must be done before referring to a measure from that cube, see section 3.5.2

For information about the various ExtendControl types and how they work refer to [ESKORT Designer, Section II – Introduction to Environments].

3.5.2 Using Tuples in Rules

Writing a measure in the rule, e.g. Item5, is a simplified notation for writing a tuple {'Item5'}.

In some cases it is not enough just to write the simplified notation. This will be the case if the measure we want to refer to is from another cube and the dimension of that cube does not match or extend to the dimension of the default cube. In this case we need to write a tuple where we bind the dimensions that are already bound in the range.

Say that the cube FormY in our example has the dimensions: Account, Years, and Code. Then it is not possible to just write FormY:Item2 in our rule – the Code dimension is not bound to a value when evaluating the rule. So we must bind it. It is however not necessary to bind the Account and Years dimension of cube FormY, since they are already bound in the range.

The member of the Code dimension we bind to could be a derived member, but it could also be a specific member of the Code dimension.

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + {FormY:Code.'200', FormY:Item2}

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

Say the code dimension has the codes: 100, 200, 300, 400, and we are interested in the value of Item2 for code 200, then the tuple would be:

```
{FormY:Code.'200', FormY:Item2}
```

In the case where we want the sum of Item2 for all codes then we need to define a derived member on the Code dimension:

```
let FormY:Code.CodeTotal = SUM(Code)
```

(this can be done in the rule formalization or an area formalization on a higher level).

The tuple will then be as:

```
{FormY:Code.CodeTotal, FormY:Item2}
```

Another example of using tuples in rules is given in section 3.11.1.

3.6 Creating Observations

Observations are created using the observation() function in the action part of the rule body.

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

call Observation**(** *name-of-profile*, *name-of-area*, *name-of-measure*, *name-of-measure* **)**

Note that the number of arguments depends on the risk model.

Instead of having to write the name of the area, the CURRENT_AREA keyword may be used.

call Observation(*name-of-profile*, **CURRENT_AREA,** *name-of-measure*, *name-of-measure* **)**

By adding the following line somewhere in the knowledge base formalization (e.g. in the top area):

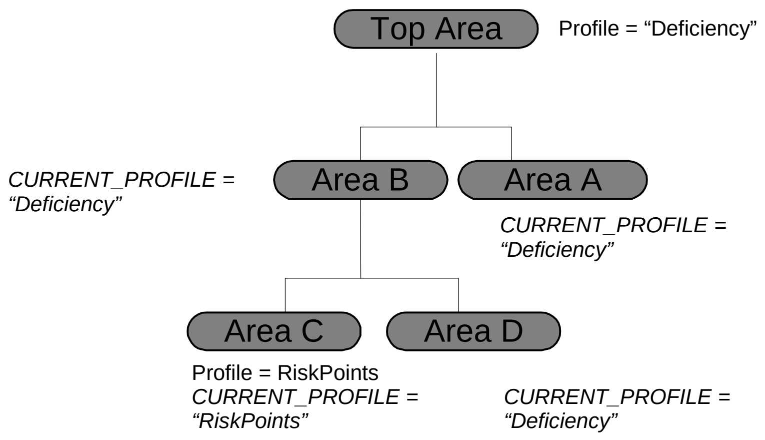
profile = “Non-Compliance”[1]

You can also use the CURRENT_PROFILE keyword in the following manner:

call Observation**(CURRENT_PROFILE, CURRENT_AREA,** *name-of-measure*, *name-of-measure* **)**

When the Profile is placed in an area formalization, all areas and rules below have this Profile at the CURRENT_PROFILE, unless it has been sat to something else.

[1] Note that this is just an example. You can use the profiles available in your system; see [ESKORT Designer User Guide Section III – Introduction to Knowledge Engineering, section 2.3] for information.


Top Area
Profile = "Deficiency"
CURRENT_PROFILE = "Deficiency"
Area B
Area A
CURRENT_PROFILE = "Deficiency"
Area C
Area D
Profile = RiskPoints
CURRENT_PROFILE = "RiskPoints"
CURRENT_PROFILE = "Deficiency"

In the example above, the Profile is sat to "Deficiency" at the top area, and all areas and rules below will therefore have CURRENT_PROFILE = "Deficiency" if they do not explicitly set it. Area C sets the Profile to "RiskPoints" and therefore CURRENT_PROFILE = "RiskPoints" for this area (and any areas and rules below that).

Instead of using CURRENT_PROFILE you may also explicitly state the name of the profile:

call Observation(**"**Non-Compliance**", CURRENT_AREA,** *name-of-measure, name-of-measure* **)**

3.7 Activate Areas or Rules

An action of a rule can also be to activate (or de-activate) another area or another rule, i.e. this area or rule will be evaluated if the rule fires. This is achieved through the functions ActivateArea and DeActivateArea respectively.

These functions should not be confused with the standard activation or deactivation of areas as described in [ESKORT Designer Guide Section III – Introduction to Knowledge Engineering, section 2.2.9].

3.8 Choice Constructs

Two choice constructs, the 'iif' and the 'case …when … then', are allowed according to the following rules from the BNF grammar:

```
<numeric_primary>     ::=   …
                            | iif  ( <search_condition>  ,  <value_expression>  ,
                            <value_expression>  )
                            | case <when_list>  else  <value_expression> end

<when_list>           ::=   <when_then>
                            | <when_list> <when_then>

<when_then>           ::=   when <search_condition> then <value_expression>
```

The following examples show how the constructs can be used to determine the value of the derived member dLikelihood.

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = iif ((dMyMember - 10000) <= 1000 , 0.1, 0.5)

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The example above shows the ’iif’ construct (immediate if).

This construct should be interpreted as follows: If dMyMember – 10000 is less than or equal to 1000, the dLikelihood is set to 0.1 otherwise dLikelihood is set to 0.5.

The ‘iif’ construct can be used if a value should have one out of two values depending on a condition.

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = case
                    when (dMyMember - 10000) <= 1000 then
                      0.1
                     when ((dMyMember - 10000) > 1000) and ((dMyMember - 10000) < 3000) then
                      0.3
                     when (dMyMember - 10000) >= 3000 then
                      0.5
                     else
                      0
rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The example above shows the ‘case …when … then’ construct.

This construct should be interpreted as follows: If dMyMember – 10000 is less than or equal to 1000, the dLikelihood is set to 0.1, otherwise if dMyMember – 10000 is greater than 1000 and less than 3000 then dLikelihood is set to 0.3, otherwise if dMyMember – 10000 is greater than 3000 then dLikelihood is set to 0.5 otherwise dLikelihood is set to 0 (but this will actually never be the case since all possibilities is covered in the other three cases).

The 'case …when … then' construct can be used if a value should have one out of two or more values depending on a condition (in the above example it should have one out of three values).

3.9 Using Functions

The grammar of the formalization language also allows us to use functions, both to calculate numeric values and to calculate Boolean values.

```
<value_expression_primary> ::=
                               …
                             | <numeric_value_function>

<numeric_value_function> ::=
                               SUM ( <set> )
                             | AVG ( <set> )      |
                             | <function_identifier> ( <param_list> )

<function_identifier>   ::=   identifier

<boolean_primary>       ::=   …
                              $ <function_identifier> ( <param_list> )
```

See the example below:

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > 10000) and Item5=1

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

The rule uses the 'slope()' and 'py()' functions. For an extensive list of available functions, see Appendix D.

Functions are useful when more complex calculations are needed. The ‘slope’ function is an example of that (the ‘slope’ function determines a value (e.g. a points score) determined by a slope).

Function can also be used for parameter lookup from the parameters database. The ‘py’ function is an example of that.

Some functions are part of the standard system and others are local to each project. It is possible to define local functions on your own (however it is strongly recommended that only personnel with programming skills do this).

Functions can be used in both definitions of derived members (as shown in the above example) and in the rule formalization.

```
within (FormX) for (Accounts, Years)

filter Buss {23, 45, 67, 88}

let dMyMember = Item3 + FormY:Item2

let dSignificance = (dMyMember - 10000) * py("CITTaxRate",MEMBER_NAME(Years.CURRENTMEMBER))

let dLikelihood = Slope(0.1,0.5,0.05,10000,1000,(dMyMember-10000))

rule R_001
if

    (dMyMember > p("R001Value")) and $IsKnown(Item5)

then

    Call Observation(CURRENT_PROFILE,CURRENT_AREA,dLikelihood,dSignificance).
```

When the result of a function is used as a logical expression (i.e. **true** or **false**) a $-sign needs to be put in front of the function name, as in $IsKnown in the above example. Not putting the $-sign in front of the function name would require a comparison of the return value of the functions, e.g. IsKnown(…) = 1. Note that this only works for functions returning a Boolean value, ie. **true** or **false**.

3.10 Including External Definitions

It is possible to include definitions, which have been specified in a KbInclude element specified for the task being evaluated. Please refer to [ESKORT Designer User Guide Section V - Defining and Releasing Tasks] for information about how to include such an element.

The include clause has the following form:

```
<area_head_member>  ::=   …
                        | include  <identifier>
```

Where <identifier> corresponds to the name given to the KbInclude element in the task.

3.11 Example of Tuple Evaluations

This section will explain the mechanics of a simple rule. It is assumed that the rule operates within the cube VAT, defined by the cell set:

Coordinate	Value
{'341001', '01-1999', 'Sales'}	100,00
{'341001', '02-1999', 'Sales'}	110,00
{'341001', '03-1999', 'Sales'}	130,00
{'341002', '01-1999', 'Sales'}	20,00
{'341002', '02-1999', 'Sales'}	18,00
{'341002', '03-1999', 'Sales'}	25,00
{'341003', '01-1999', 'Sales'}	500,00
{'341003', '02-1999', 'Sales'}	550,00
{'341003', '03-1999', 'Sales'}	540,00
{'341001', '01-1999', 'Purchases'}	40,00
{'341001', '02-1999', 'Purchases'}	45,00
{'341001', '03-1999', 'Purchases'}	50,00
{'341002', '01-1999', 'Purchases'}	10,00
{'341002', '02-1999', 'Purchases'}	10,00
{'341002', '03-1999', 'Purchases'}	15,00
{'341003', '01-1999', 'Purchases'}	200,00
{'341003', '02-1999', 'Purchases'}	250,00
{'341003', '03-1999', 'Purchases'}	270,00

A simple rule using this cube could be:

```
within VAT for (Accounts, Month)
rule
 if
  Purchases < 0,5 * Sales
 then
  call Observation(...).
```

The first step in evaluating this rule is determining the set of tuples identified by the range (the within clause). In this example, the range would identify the following tuples:

{'341001', '01-1999'},
{'341001', '02-1999'},
{'341001', '03-1999'},
{'341002', '01-1999'},
{'341002', '02-1999'},
{'341002', '03-1999'},
{'341003', '01-1999'},
{'341003', '02-1999'},
{'341003', '03-1999'}

After this, each of these tuples are processed individually.

For each tuple, the condition part of the rule is evaluated using the tuple as evaluation *context*.

The following illustrates the processing for the first tuple, {'341001', '01-1999'}:

In evaluating the condition, the first step will be determine the value of Purchases. Writing Purchases as in the rule, is a simplified notation for writing the tuple {'Purchases'}.

While {'Purchases'} in itself refers to a cell-set with 9 different cells, when combined with the evaluation context {'341001', '01-1999'}, a tuple, {'341001', '01-1999', 'Purchases'}, emerges, which identifies exactly one cell. The value of Purchases is the value that can be looked up in this cell: 40,00.

3.11.1 Understanding Derived Members

Consider the following variation of the example above:

```
within VAT for (Accounts, Months)
let Accounts.AccountsTotal = SUM(Accounts)
rule
  if
    {Accounts.AccountsTotal, Purchases} < 0,1 * Sales
  then
    call Observation(...).
```

Like in the original example, we consider the first context tuple. We have to establish the value of {'Accounts.AccountsTotal', 'Purchases'}.

Combining with members from the context tuple, we get:

{'Accounts.AccountsTotal', '01-1999', 'Purchases'}

Note that this coordinate does not correspond to a physical cell in the cube. In processing the tuple, the analysis engine notes that one of the members in the tuple is a derived (or calculated) member. The engine continues processing this member by replacing the member with the underlying expression.

{SUM(Accounts), '01-1999', 'Purchases'},

which is equivalent to

{SUM('341001', '341002', '341003'), '01-1999', 'Purchases'},

since SUM(Accounts) is equivalent to writing SUM('341001', '341002', '341003'). Using the dimension name is just a simplified notation for listing all physical members in the dimension.

The derived member is equivalent to

SUM({'341001', '01-1999', 'Purchases'}, {'341002', '01-1999', 'Purchases'}, {'341003', '01-1999', 'Purchases'})

By looking up each of the physical cells, this can be reduced to

SUM(40,00, 10,00, 200,00)

Completing the SUM operation yields the value for the tuple:

250,00.

To take the example one step further, consider the variation:

```
within VAT for (Accounts, Months)
let Accounts.AccountsTotal = SUM(Accounts)
let VAT.Profit = Sales – Purchases
rule
 if
  {Accounts.AccountsTotal, VAT.Profit} < 0,1 * VAT.Profit
 then
  call Observation(...).
```

Here we have to establish the value of {'Accounts.AccountsTotal', 'VAT.Profit'}.

Combining with members from the context tuple, we get:

{'Accounts.AccountsTotal', '01-1999', 'VAT.Profit'}

This coordinate has several derived members.The engine continues processing by choosing one of the derived members and proceeding as above.

{SUM(Accounts), '01-1999', 'VAT.Profit'},

which is equivalent to

{SUM('341001', '341002', '341003'), '01-1999', 'VAT.Profit'},

The derived member is equivalent to

SUM({'341001', '01-1999', 'VAT.Profit'}, {'341002', '01-1999', 'VAT.Profit'}, {'341003', '01-1999', 'VAT.Profit'})

Unlike in the above example, the individual tuples can not be looked up directly, since they contain a derived member ('VAT.Profit'). Instead, the engine proceeds to reduce these one by one.

{'341001', '01-1999', 'VAT.Profit'}

Becomes

{'341001', '01-1999', 'Sales - Purchases'}

Which becomes

{'341001', '01-1999', 'Sales'} - {'341001', '01-1999', 'Purchase'}

By looking up each of the physical cells, this can be reduced to

100,00 - 40,00

Yielding

60,00.

Completing equivalent steps for the remaining two tuples, results in

SUM(60,00, 10,00, 300,00)

Completing the SUM operation yields the value for the tuple:

370,00.

4. Troubleshooting Formalization Problems

The following sections are intended as inspiration on how to address problems that may occur during work with the knowledge engineering. Troubleshooting by its nature is a creative process. This section tries to provide inspiration on steps to take to resolve the problem you experience.

Errors typically fall into one of the following three categories:

- Syntax errors

  Syntax errors occur when the formalization does not adhere to the MDC language definition.

  In Designer you can verify the syntax before releasing a knowledge base.

- Semantic errors

  Semantic errors occur when the formalization is valid but does not correspond to the intent. Usually this is detected when the observations made are not what you expected for a given taxpayer. This type of error is by far the most common error.

- Execution errors

  Execution errors are characterized by the generation of an error message during evaluation of the rule.

4.1 Correcting Syntax Errors

Sometimes you will get a syntax error that is not immediately visible. One approach in this situation is to start reducing the complexity of the rule – e.g. removing one clause at time, or replacing a complex expression with a simpler.

When the rule is syntactically correct again, you can then reintroduce the complexity again. Instead of writing the new expressions from scratch, try introducing equivalent clauses from a working rule, and then modify (e.g. changing the member name referenced).

4.2 Reread the Error Message

Read the error message carefully, often is does include a hint as to what is wrong, although it can be cluttered by other information.

4.3 Compare with Working Rules

Find a similar rule that works, or is semantically correct, and look for differences – e.g. a set of parentheses missing. Consider whether the differences could explain the problem you are experiencing.

4.4 Examining Error Annotations

Generally the error messages provided by the Designer (or during analysis) include an error number, and a textual description of the problem.

In the online Help, you will find an entry Errors in the Contents. This entry holds links to Error Annotations and Custom Error Annotations. General error annotations provide annotations for selected errors of a general nature, while Custom Error Annotations can provide annotations for errors specific to you configuration. The contents of Custom Error Annotations are primarily the responsibility of you as knowledge engineer.

4.5 Build a History

In general it is recommended that you built a history of the errors that you have encountered, and document the way you fixed them. This will help you when you encounter the same error at a later point in time.

4.6 Verify your Test Data

If a taxpayer is selected as test subject based on a reporting tool, make sure that the data available in the report is the same as the data used by the Analysis Server.

If you are not sure you can review data by downloading for the taxpayer in Designer or you can review data by using standard database query tools.

Investigation of a possible error should always be done with the intent of the formalization in mind and the key data for the taxpayer selected as test subject at hand.

4.7 Look at Intermediate Results

In some situations, you have a rule that does not fire as you expect when you are using [Test in Testbed]. To find the explanation why the rule is – or is not – firing as expected, you can temporarily add observation clauses that provide information on intermediate results.

Consider the following example:

```
within VAT for (Accounts, Years)
if
  Purchases < 0,5 * Sales
then
  Call Observation(CURRENT_PROFILE,CURRENT_AREA,1,1);
```

Since the Observation function can be called with a numeric value, which is displayed on the observation tab, you can use it to see intermediate information. Since the Boolean values **true** and **false** are considered equvalent to 1 and 0 respectively, it is also possible to set intermediate results of a Boolean nature.

To illustrate, in the above example we could temporarily modify the definition to the following:

```
let boolean t1 = (0,5 * Sales)
let boolean t2 = (Purchases < 0,5 * Sales)

within VAT for (Accounts, Years)
if
  1 = 1 or
  Purchases < 0,5 * Sales
then
  Call Observation(CURRENT_PROFILE,CURRENT_AREA,101,Purchases);
  Call Observation(CURRENT_PROFILE,CURRENT_AREA,102,Sales);
```

```
    Call Observation(CURRENT_PROFILE,CURRENT_AREA,103,t1);
    Call Observation(CURRENT_PROFILE,CURRENT_AREA,104,t2);
    Call Observation(CURRENT_PROFILE,CURRENT_AREA,1,1).
```

Note: the following changes have been made:

The condition has been modified to ensure that the rule always fires.

Two Boolean derived members have been added: one representing the right hand side of the relational expression in the condition (t1), and one representing the entire relational expression (t2).

Four new observations have been – using as risk points, Purchases, Sales, t1 and t2. Note that each have been given a distinct value for likelihood.

You can now use [Test in Testbed] on the modified version of the rule, and go to the Observation tab. In the observations you can now see the intermediate values. Hopefully these can help you understand why the rule is firing or not firing.

Before proceeding, remember to remove all the elements you added.

4.8 Troubleshooting

The following table may help you to troubleshoot semantic errors and execution errors:

Symptom	Possible cause	Suggested action
No observations were made	The status of the rule may be "Under construction" or "Ready-for-Test"	Check the status of the rule in the knowledge base.
	There is an error in the formalization syntax.	Check the error log on the server.
	There is a semantic error in the formalization.	Test the rule using the testbed in Designer and check the calculation log to trace the evaluation.
Observations are made for every period.	Periodic data is being compared to non-periodic data.	Review intent of rule, change formalization or change environment.
The observations made are not what you expected	There is a semantic error in the formalization.	Verify the evaluation order of conditions separated by "AND" and "OR". Test the rule using the test-bed in Designer and check the calculation log to trace the evaluation.
	One of the extract definitions used may not be correct.	Review extract definitions.
	Intent of rule is not matched by formalization	Change formalization.

The data used were not as expected	A change was made to the data warehouse structure.	Change extract definition.
	Information is not available in the data warehouse before a given date.	Review intent of rule. Change formalization to work after the given date.

4.9 Contacting Support

In case you have an operational support agreement, you can contact you support contact.

You should include as much information about the problem as needed to understand it. This could be a description of how you discovered the problem, what actions you have taken to pinpoint the cause and error logs from the analysis if needed. Screenshots, text pasted from the Designer and attached configuration files might also be relevant.

Appendix A: MDC Language Definition

Revision: 1.49

The following provides a complete overview of the MDC Language Grammar.

Contents:

Notation

The grammar is presented in EBNF, using the following conventions:

Style	Meaning
<symbol>	Indicates a non-terminal symbol of the grammer (e.g. <area> or <rule>).
keyword	Indicates a keyword, syntactical symbol or operator (e.g. **within** or **;**).
string	Indicates a terminal symbol (e.g. identifier, string, real or unsigned-integer).
X | Y	Indicates that X and Y are alternatives.
[X]	Indicates that X is optional.
X*	Indicates zero or more repetitions of X.
X+	Indicates one or more repetitions of X
(XY)	Indicates grouping for use with other operaters.

Symbols

area ::=

"area" **"{"** <areamember> * **"}"**
| **"area"** <identifier> **"{"** <areamember> * **"}"**

derived_area ::=

"area" **"{"** <areamember> * **"}"**
| **"area"** <identifier> **"{"** <areamember> * **"}"**

area_head_member ::=

"within" <withinbody>
| **"let"** <letbody>
| **"filter"** <filterbody>
| **"profile"** **"="** <str>
| **"include"** <identifier>

areamember ::=

<area_head_member>
| **"rule"** <rulescope>
| <derived_area>

withinbody ::=

"(" <cubename> (**","** <cubename>)* **")"** <forbody>

forbody ::=

empty
| **"for"** **"("** <set> (**","** <set>)* **")"**
| **"FOR_ALL_KNOWN_CELLS"**

identifier ::=

<regular_identifier>

regular_identifier ::=

identifier1

uint ::=

uint1

real ::=

real1

str ::=

str1

cubename ::=

<identifier>

dimension ::=

<identifier>

qualified_dimension ::=

<cubename> **":"** <dimension>

member ::=

<identifier>
| <dimension> **"."** <identifier>
| <dimension> **"."** **"FIRSTMEMBER"**
| <qualified_dimension> **"."** **"FIRSTMEMBER"**
| <dimension> **"."** **"LASTMEMBER"**
| <qualified_dimension> **"."** **"LASTMEMBER"**
| <dimension> **"."** **"PREVMEMBER"**
| <qualified_dimension> **"."** **"PREVMEMBER"**
| <dimension> **"."** **"NEXTMEMBER"**
| <qualified_dimension> **"."** **"NEXTMEMBER"**
| <qualified_dimension> **"."** <identifier>
| <cubename> **":"** <identifier>
| <member_value_expression>
| <dimension> **"."** **"member"|"MEMBER"** <member_index>

member_index ::=

index_str1

new_member ::=

<member>

tuple ::=

"{" <member> (**","** <member>)* **"}"**
| <member>

set ::=

<set_value_expression>
| **"{"** <tuple> (**","** <tuple>)* **"}"**

```
letbody ::=

<new_member> "=" <value_expression>
| "member"|"MEMBER" <new_member> "=" <value_expression>
| "double" <new_member> "=" <value_expression>
| "integer" <new_member> "=" <value_expression>
| "string" <new_member> "=" <value_expression>
| "boolean" <new_member> "=" <search_condition>
| "SET"|"set" <identifier> "=" <set>

filterbody ::=

<identifier> "{" <filter_element> ( "," <filter_element> )* "}"

filter_element ::=

<uint>
| <str>
| <uint> "-" <uint>

value_expression ::=

<numeric_value_expression>

numeric_value_expression ::=

<value>

value ::=

<term>
| <value> "+" <term>
| <value> "-" <term>
| "ADD" "(" <value_expression> ( "," <value_expression> )* ")"

term ::=

<factor>
| <term> "*" <factor>
| <term> "/" <factor>

factor ::=

<numeric_primary>
| "+" <numeric_primary>
| "-" <numeric_primary>

numeric_primary ::=

<value_expression_primary>
| <numeric_value_function>
| "iif" "(" <search_condition> "," <value_expression> "," <value_expression> ")"
| "case" <when_then> + "else" <value_expression> "end"

when_then ::=

"when" <search_condition> "then" <value_expression>

value_expression_primary ::=

<unsigned_numeric_literal>
| <character_string_literal>
| "null"|"NULL"
| "(" <value_expression> ")"
| <tuple>
| "MEMBER_NAME" "(" <member> ")"
| "CURRENT_PROFILE"
| "CURRENT_AREA"
| "CURRENT_CUBE"
| "CUBE_SET"
| "TRUE"|"true"|"sand"
| "FALSE"|"false"|"falsk"

unsigned_numeric_literal  ::=

<exact_numeric_literal>

character_string_literal  ::=

<str>

exact_numeric_literal  ::=

<real>
|  <uint>

rulescope  ::=

<rulebody>  "."
|  <identifier>  <rulebody>  "."

rulebody  ::=

"if"  <search_condition>  "then"  <action>  (  ";"  <action>  )*

search_condition  ::=

<boolean_term>
|  <search_condition>  "||"|"or"|"eller"  <boolean_term>

boolean_term  ::=

<boolean_factor>
|  <boolean_term>  "&&"|"and"|"og"  <boolean_factor>

boolean_factor  ::=

<boolean_primary_opt_tagged>
| "!"|"not"|"ikke"  <boolean_primary_opt_tagged>

boolean_primary_opt_tagged  ::=

tag1  "("  <boolean_primary>  ")"
|  <boolean_primary>
|  tag1

boolean_primary  ::=

<value_expression>  "="  <value_expression>
|  <value_expression>  "<>"  <value_expression>
|  <value_expression>  ">"  <value_expression>
|  <value_expression>  "<"  <value_expression>
|  <value_expression>  "@"  <value_expression>
|  <value_expression>  "!@"  <value_expression>
|  <value_expression>  ">="  <value_expression>
|  <value_expression>  "<="  <value_expression>
|  <value_expression>  "like"  <value_expression>
|  <value_expression>  "in"|"i"  <value_expression>
| "("  <search_condition>  ")"
| "$"  <identifier>  "("  [ <parameter>  (  ","  <parameter>  )* ]  ")"

numeric_value_function  ::=

"SUM"|"AGGREGATE"  "("  <set>  ")"
| "_SUM"  "("  <set>  ")"
| "AVG"  "("  <set>  ")"
| "MIN"  "("  <set>  ")"
| "MAX"  "("  <set>  ")"
```

| **"SET"|"set"** **"("** <set> **")"**
| <identifier> **"("** [<parameter> (**","** <parameter>)*] **")"**

set_value_expression ::=

<identifier>
| <dimension> **"."** **"MEMBERS"**
| <dimension> **"("** <member_index> **"-"** <member_index> **")"**
| <dimension> **"."** **"EXCEPT"** **"("** <member> **")"**

member_value_expression ::=

<dimension> **"."**
| <dimension> **"."** **"CURRENTMEMBER"**

action ::=

"call" <identifier> **"("** [<parameter> (**","** <parameter>)*] **")"**

parameter ::=

<value>

Terminal Symbols

identifier1

Any sequence of alpha-numeric characters starting with a non-numeric character;
(e.g. MyName, First_Name).

uint1

An unsigned integer number constant;
(e.g. 5 or 3000).

real1

A decimal number constant;
(e.g. 1000000 or 0.5).

str1

Any sequence of alpha-numeric characters delimited with " characters;
A "character in the string is achieved via the escape sequence \".
A \ character in the string is achieved via the escape sequence \\.
(e.g. "This is a string", or "This is \"a string\"").

index_str1

Any sequence of alpha-numeric characters enclosed in square parenthesis [and];
(e.g. [This is an index], or [123]).

index_str2

Not in use!

tag1

A sequence of the form: %<uint1> optionally used to tag simple relational expressions.
(e.g. %1 or %5).

Appendix B: Operator Precedence and Associatively

If no parentheses are used to separate operands then the precedence and associatively of operators in expression are given in the following table:

Precedence (1)	Operator(s)	Associativity	Description
1	+ - (unary)	Left	Unary sign operators, like in –3
2	not	Right	Logical negation.
3	* /	Left	Multiplicative operators. \ is integer division.
4	+ - (binary)	Left	Additive operator.
5	< > <= >= = <>	Left	Relative/comparison operators - equality and inequality.
6	and	Left	Logical and.
7	or	Left	Logical or.

(1) 1 is highest precedence.

What is operator precedence?

Operator precedence determines which parts of an expression are evaluated before the other parts. For example, the expression

2 + 2 * 7

evaluates to 16, not 28, because the * operator has a higher precedence than the + operator. Thus the 2 * 7 part of the expression is evaluated before the 2 + 2 part. If you wish, you can use parentheses in expressions to clarify evaluation order or to override precedence. For example, if you really wanted the result of the expression above to be 28, you could write the expression like this:

(2 + 2) * 7

What is operator associativity?

Operator associativity is why the expression 8 - 3 - 2 is calculated as (8 - 3) - 2, giving 3, and not as 8 - (3 - 2), giving 7.

When multiple operators of the same precedence appear side by side in an expression, the associativity of the operators determines the order of evaluation. In EBC based applications, all operators except the logical negation operator (**not**) are left-associative.

Examples:

2 * 5 / 7 is evaluated as (2 * 5) / 7.

$IsKnown(x) and $IsKnown(y) and $IsKnown(z) is evaluated as

($IsKnown(x) and $IsKnown(y)) and $IsKnown(z) .

Order of Evaluation

After taking into account precedence and parentheses it is guaranteed that expressions will be evaluated left to right.

A simple example:

8 + 3 * 6 – 2 / 4 will be evaluated as (8 + (3 * 6)) - (2 / 4)

An example of evaluating a Boolean condition:

dRatio / dIndustryAverage < 0.75 and not (“2” = SBTCalculationTypeCode) and not (“6” = SBTCalculationTypeCode)

will be evaluated as

((((dRatio / dIndustryAverage) < 0.75)   and (not (“2” = SBTCalculationTypeCode)))

and (not (“6” = SBTCalculationTypeCode)))

Appendix C: Rule Examples

In the following some example rules using various parts of the formalization language are explained.

In the examples we assume that we have three cubes:

- IncomeTax with dimensions Accounts, Years and IncomeTax (measure).
- VAT with dimensions Accounts, Months, VATCode and VAT (measure).
- SalesTax with dimensions Accounts, Years, SalesCode and SalesTax (measure).

Example 1

```
within (IncomeTax) for (Accounts, Years)

let deviance = (Item6 - Item7) * 100/Item6

rule Example1
if
   deviance < 10
then
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,2).
```

The rule used cube IncomeTax as default cube. Therefore we need to bind the Accounts and Years dimensions in the ranges. This rule is evaluated for all accounts and all years.

This rule defines a derived member called ‘deviance’. It is calculated from the members Item6 and Item7 in the IncomeTax measure dimension. This illustrates how complex calculations can be defined as derived members to make the rule condition simpler.

The rule could also be implemented as follows:

```
within (IncomeTax) for (Accounts, Years)

rule Example1
if
   (Item6 - Item7) * 100/Item6 < 10
then
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,2).
```

Example 2

```
within (IncomeTax) for (Accounts, Years)

rule Example2
if
   MEMBER_NAME(Years.CURRENTMEMBER) >= "20000101-20001231" and
   $IsZeroOrNotKnown (Item4) and
   {VAT:VATCode.'220', VAT:Amount1} > 10000
```

```
then
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,5).
```

The rule used cube IncomeTax as default cube. Therefore we need to bind the Accounts and Years dimensions in the ranges. This rule is evaluated for all accounts and all years.

However the rule will only fire year 2000 and later, since it is put as a condition, MEMBER_NAME(Years.CURRENTMEMBER) >= "20000101-20001231". This illustrates how we can get name of the members from the ranges that are currently evaluated.

The rule uses a function, IsZeroOrNotKnown. Since the return value is used as a logical expression it has a $-sign in front – otherwise we needed to write IsZeroOrNotKnown(…) = 1.

In the rule we use a member from the VAT cube. Since this cube has other dimensions than the IncomeTax cube, we need to bind these. However the Months dimension is automatically extended into the Year dimension by adding values for all months within the year currently evaluated (The environment has been defined so this happens). We then just need to bind the VATCode dimension which is done by the tuple:

{VAT:VATCode.’220’, VAT:Amount1}

Example 3

```
within (VAT) for (Accounts, Months, {VATCode.'200'})

filter LoB { 20, 21, 22, 23, 24, 25, 26, 27, 28 }

rule Example3
if
   $IsKnownAndNotZero(Amount2) and
   IncomeTax:Item7 > 1000
then
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,4).
```

The rule used cube VAT as default cube. Therefore we need to bind the Accounts, Months and VATCode dimensions in the ranges. This rule is evaluated for all accounts and all months but only for VATCode = 200.

The rule uses the filter mechanism, so it is only evaluated for the taxpayers that fulfill the filter condition – in this case has a LoB (Line-of-Business) code that is either 20, 21, 22, 23, 24, 25, 26, 27 or 28.

In the rule we use a member from the IncomeTax cube. We do not need to bind any of the IncomeTax dimensions, since it has Accounts which is already bound in the ranges, and Years which is automatically extended into the Months dimension (normally by distributing the yearly value into monthly values). So we can just write the measure from IncomeTax we want to use:

IncomeTax:Item7

Example 4

```
within (IncomeTax) for (Accounts, Years)

let Rate = Item3/Item9 * 100
let standardRate = py("Rate", MEMBER_NAME(Years.CURRENTMEMBER))
let difference = standardRate - Rate
let rp = Slope(1,5,1,0,3,difference)

rule Example4
if
   Rate < standardRate
then
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,rp).
```

The rule used cube IncomeTax as default cube. Therefore we need to bind the Accounts and Years dimensions in the ranges. This rule is evaluated for all accounts and all years.

This rule defines four different derived members in order to keep the rule condition very simple.

One of the derived members is a parameter lookup in the parameter database. It is a yearly parameter lookup (using function ‘py’) and it finds the standard rate for the year currently evaluated.

This rule also uses the slope function to generate the risk points. This illustrates risk points can vary dependent on e.g. how big a difference between two values is.

Example 5

```
within (SalesTax) for (Accounts, Years, {SalesCode.dummy})

let Val1 =
  case
    when MEMBER_NAME(Years.CURRENTMEMBER) < "20000101-20001231" then
       {SalesCode.'42', Amount1}
    else
       {SalesCode.'43', Amount1}
   end

let Val2 =
  case
    when MEMBER_NAME(Years.CURRENTMEMBER) < "20000101-20001231" then
       {SalesCode.'63', Amount1}
    else
       {SalesCode.'66', Amount1}
   end

rule Example5
if
   Val1 > 0.8 * Val2
then
```

```
   call Observation(CURRENT_PROFILE,CURRENT_AREA,1,2).
```

The rule used cube SalesTax as default cube. Therefore we need to bind the Accounts, Years and SalesCode dimensions in the ranges. This rule is evaluated for all accounts and all years. As the SalesCode in the ranges we use a “dummy” member, since we explicitly state the SalesCodes we are interested in in the rule.

The rule defines two different derived members using the ‘case …when … then’ construct. In our case the SalesCodes has changed in year 2000, so we need to get our values from different codes dependent on the year, i.e. if the year evaluated is 1999 or earlier we have:

Val1 = {SalesCode.'42', Amount1}

Val2 = {SalesCode.'63', Amount1}

and if the year evaluated is 2000 or later

Val1 = {SalesCode.'43', Amount1}

Val2 = {SalesCode.'66', Amount1}

Appendix D: Function Reference

AbsValue (double x)

Description: Returns the absolute value of the parameter (*x*). For positive values and zero the result will be the same as the parameter (*x*). For negative values the result will be the positive value (the minus sign will be removed), i.e. –*x*.

Syntax: double AbsValue(double x)

Parameters:

x	The double value which absolute value is returned.

Example:

`AbsValue``(-1000)`

Returns 1000

`AbsValue``(3.14)`

Returns 3.14

ActivateArea (string AreaName)

Description: Activates the area (or rule) specified by the parameter.

Syntax: void ActivateArea(string Area*Name*)

Parameters:

AreaName	The name of the area (or rule) which are activated.

Example:

`ActivateArea``(”MyArea”)`

Activates the area (or rule) named “MyArea”.

Add (TUPLE Measure1, TUPLE Measure2, …, TUPLE MeasureN)

Description: Adds the specified measures together, ignoring measures that are ***NULL***.

Syntax: double Add(Measure$_1$, Measure$_2$, …, Measure$_N$)

Parameters:

Measure$_i$	The i’th measure that added to the total.

Example:

```
Add(MyMember1, MyMember2, MyMember3)
```

Adds together the values kept by the specified members.

AllEmpty (TUPLE Measure, string DimensionName)

Description: Returns ***TRUE*** if all cells identified by the measure (*Measure*) are empty for the dimension (*DimensionName*).

Syntax: boolean AllEmpty(TUPLE *Measure*, string *DimensionName*)

Parameters:

Measure	The measure member which are investigated for a specified dimension (*DimensionName*)**.**
DimensionName	The name of the dimension in which the member’s existence is investigated.

Example:

```
AllEmpty(TaxableIncome, ”Years”)
```

Returns ***TRUE*** if no measures in the years-dimension exist. If one or more exists the function returns ***FALSE***.

AllZeroOrNotKnown (TUPLE Measure)

Description: Returns ***TRUE*** when all cells built with the measure (*Measure*) and all the ordinary members of the period dimension are NULL/EMPTY or zero.

Syntax: boolean AllZeroOrNotKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether all the measures are not known or is zero.

Example:

```
AllZeroOrNotKnown(TaxableIncome)
```

Returned ***TRUE*** if no declared taxable incomes are not known by the system or if all the declared incomes are zero; otherwise ***FALSE*** is returned.

See also:
IsEmpty
IsKnown
IsKnownAndNotZero
IsNotKnown

AnyZeroOrNotKnown (TUPLE Measure)

Description: Returns ***TRUE*** when any cells built with the measure (*Measure*) and all the ordinary members of the period dimension are NULL/EMPTY or zero.

Syntax: boolean AnyZeroOrNotKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether any of the measures are not known or are zero.

Example:

```
AnyZeroOrNotKnown(TaxableIncome)
```

Returned ***TRUE*** if any declared taxable incomes are known as zero by the system or if any the declared incomes are unknown; otherwise ***FALSE*** is returned.

See also:
IsEmpty
IsKnown
IsKnownAndNotZero
IsNotKnown

Assign (TUPLE Tuple, ANY_TYPE Value)

Description: Sets the value (*Value*) in the cell identified by the tuple (*Tuple*).

Syntax: void Assign(TUPLE *Tuple*, ANY_TYPE *Value*)

Parameters:

Tuple	The coordinate of the cell into which the value is assigned.
Value	The value that is assigned into the cube.

Example:

```
Assign(StatusCode, ”1”)
```

Sets the *StatusCode* member in the current context to the value “1”.

Average (TUPLE Measure, string DimensionName)

Description: Returns the average value of measure *Measure* for the dimension *DimensionName*.

Syntax: double Average(TUPLE *Measure*, string *DimensionName*)

Parameters:

Measure	The measure member which average will be calculated over the specified dimension (*DimensionName*)**.**
DimensionName	The name of the dimension over which the average is calculated.

Example:

```
Average(TaxableIncome, ”Years”)
```

Returns the average taxable income over the years.

BindMacro (string parameter, UNKNOWN_TYPE macro)

Description: Binds the name of a parameter to a macro so the macro can be expanded inside a text.

Syntax: BindMacro(string *Parameter*, UNKNOWN_TYPE *Macro*)

Parameters:

Parameter	The name of the parameter which will be bound to the macro.
Macro	The macro which will be identified by the specified parameter name.

Example:

```
BindMacro(”Volume”, FormatNumber(volume,
"%G3',’%D'.'%P2%TZ"))
```

Formats the value represented by the variable *volume* and binds it to the parameter name *volume*. Subsequent the tag: **`%Volume%`** can be used in a text – e.g.;

```
High risk due to a volume of %Volume%
```

If e.g. the volume is calculated and formatted as 41.4 the produced text will be:

```
High risk due to a volume of 41.4
```

BuildDate (ANY_TYPE date, string modification)

Description: Builds a date by applying one or more modification steps (separated by semicolons) to the base date (*date*) passed. The passed base date and the returned date will be in the format *yyyymmdd* or *yyyy-mm-dd* (determined by the length of the string).

The following step types are supported:

- [*y*|*q*|*m*|*d*]<*count*> will add *count* years (*y*), quarters (*q*), months (*m*) or days (*d*) respectively.
- *fd*[[*y*|*q*|*m*] will change date to the first day of year (*fdy*), quarter (*fdq*) and month (*fdm*) respectively.

ld[[*y*|*q*|*m*] will change date to the last day (*ld*) of year, quarter (*ld*) and month (*lm*) respectively.

Syntax: string BuildDate(ANY_TYPE date, string modification)

Parameters:

date	The base date that is modified.
modification	Patterns describing the modification applied.

Example:

```
BuildDate(LatestVATReturnDate, ”m:-5;fdm”)
```

BuildDate2 (ANY_TYPE date, string modification, double format)

Description: Builds a date by applying one or more modification steps (separated by semicolons) to the base date (*date*) passed. The passed base date and the returned date will be in the format *yyyymmdd* or *yyyy-mm-dd* (determined by the length of the string).

The *BuildDate2*-function is a copy of the *BuildDate*-function – only the options for the formatting of the result has been added.

The following step types are supported:

- [*y*|*q*|*m*|*d*]<*count*> will add *count* years (*y*), quarters (*q*), months (*m*) or days (*d*) respectively.
- *fd*[[*y*|*q*|*m*] will change date to the first day of year (*fdy*), quarter (*fdq*) and month (*fdm*) respectively.

ld[[*y*|*q*|*m*] will change date to the last day (*ld*) of year, quarter (*ld*) and month (*lm*) respectively.

The following formats (specified by *format*) are supported for the resulting date:

format	**Pattern**
1	yyyymmdd
2	yyyy-mm-dd
3	dd-mm-yyyy
Otherwise	yyyymmdd

Syntax: string BuildDate2(ANY_TYPE date, string modification, double format)

Parameters:

date	The base date that is modified.
modification	Patterns describing the modification applied.
format	A number indicating which format in which the build date should be returned.

Example:

```
BuildDate2(LatestVATReturnDate, ”m:-5;fdm”, 2)
```

Cardinality (UNKNOWN x)

Description: Returns the cardinality of the specified set, i.e. the number of members contained in the set. If the specified argument represents another type than a set 1 is returned. If the specified argument is ***NULL*** then 0 is returned.

Syntax: long Cardinality(UNKNOWN x)

Parameters:

x The set which cardinality is determined by the function.

ConcatStr (string s_1, string s_2)

Description: Returns a string which contents is the concatenation of the two argument strings (s_1 and s_2)

Syntax: string ConcatStr(string s1, string s2)

Parameters:

s1 The string which will form the left part of the concatenated string

s2 The string which will form the right part of the concatenated string

Example:

```
ConcateStr(”The first string”, ”The second string”)
```

Returns the text “The first stringThe second string”

CountTo (integer n)

Description: Increases a general counter. If the counter reaches the specified threshold (*n*) then the counter is reset and the function returns ***TRUE***. If the threshold is not reached then the function returns ***FALSE***.

Hereby it is possible to make a rule which “fires” with a fixed specified interval.

Syntax: boolean CountTo(integer n)

Parameters:

n The threshold at which the counter reset and the function returns ***TRUE***.

Example:

```
CountTo(36)
```

Returns the ***TRUE*** one out of 36 calls. The other 35 calls returns ***FALSE***.

CountToWithKey (integer n, string key)

Description: Increases the counter identified by the specified key (*key*). If the counter reaches the specified threshold (*n*) then the counter is reset and the function returns ***TRUE***. If the threshold is not reached then the function returns ***FALSE***.

Hereby it is possible to make a rule which “fires” with a fixed specified interval.

By giving the counter a key it is possible to have multiple counters configured each having a current count and threshold.

Syntax: boolean CountToWithKey(integer n, string key)

Parameters:

n	The threshold at which the counter reset and the function returns ***TRUE***.
key	The name by which the counter is identified.

Example:

```
CountToWithKey(36, ”MyCounter”)
```

Returns the ***TRUE*** one out of 36 calls by the key “MyCounter”. The other 35 calls returns ***FALSE***.

CubeEmpty (string CubeName)

Description: Returns ***TRUE*** if the entire cube (*CubeName*) is empty.

Syntax: boolean CubeEmpty(string *CubeName*)

Parameters:

CubeName	The name of the cube which is investigated.

Example:

```
CubeEmpty(”MyCube”)
```

Returns ***TRUE*** if the cube named MyCube is empty; otherwise ***FALSE*** is returned.

DateHalfAYearBack (string date)

Description: Returns the date that is half a year earlier than the specified date (*date*).

Both the specified date and the returned date has the format: *YYYYMMDD*

The specified date is a string which is assumed to hold a date in the leftmost 8 positions. This means that valid arguments are of a form *YYYYMMDD* or *YYYYMMDD-YYYYMMDD*.

If the specified date is an empty string, then the current date is used.

Syntax: string DateHalfAYearBack("19660714")

Parameters:

date	The date which is used as offset for determining the date a half year back.

Example:

```
DateAHalfYearBack(MEMBER_NAME(CalendarYears.CURRENTMEMBER))
```

DateToLong (string date)

Description: Converts the specified date (*date*) from a string to an integer value (*n*) calculated as:

$$n = 10000 \cdot YYYY + 100 \cdot MM + DD$$

If *date* specifies an invalid date then 0 is returned.

Syntax: long DateToLong("19660714")

Parameters:

date The date which is converted into an integer value. The *date* must be a string of either the length 8 or 10.

If the string is 8 characters long the date must be specified at the following format:

YYYYMMDD

If the string is 10 character long the date must be specified at the following format:

YYYY□MM□DD, where □ can be any character.

Example:

```
DateToLong(”2010x05x04”)
```

Returns the integer value 20100504

DateXYearBack (string date, double nYears)

Description: Returns the date that is the specified number of years (*nYears*) earlier than the specified date (*date*).

The returned date has the format: *YYYYMMD*D.

If *date* is an empty string, then current date is used instead.

Syntax: long DateXYearBack(string date, double nYears)

Parameters:

date	The date from which the number of years are withdrawn before the final date is returned. The *date* argument is a string which is assumed to hold a date in the leftmost 8 positions. This means that valid arguments are of a form *YYYYMMDD* or *YYYYMMDD-YYYYMMDD*.

Example:

```
DateXYearBack(”20100504”, 3)
```

Returns the integer value 20100504

DaysBetween (string startDate, string endDate)

Description: Returns the number of days between the two specified dates. If the two dates are equal 0 is returned.

Syntax: double DaysBetween(string startDate, string endDate)

Parameters:

startDate	The date from which the number of days are counted. The argument is a string which is assumed to hold a date in the leftmost 8 positions. This means that valid arguments are of a form *YYYYMMDD* or *YYYYMMDD-YYYYMMDD*.
endDate	The date until which the number of days are counted. The argument is a string which is assumed to hold a date in the leftmost 8 positions. This means that valid arguments are of a form *YYYYMMDD* or *YYYYMMDD-YYYYMMDD*.

Example:

```
DaysBetween(”20100504”, ”20100501”)
```

returns the value 3.

DaysPassed (string since)

Description: Returns the number of days that has passed since the specified date (*since*). If the specified date is the current day then 0 is returned.

Syntax: double DaysPassed(string since)

Parameters:

since	The date from which until now the number of days are counted. The argument is a string which is assumed to hold a date in the leftmost 8 positions. This means that valid arguments are of a form *YYYYMMDD* or *YYYYMMDD-YYYYMMDD.*

Example:

```
DaysPassed(”19660714”)
```

Returns the number of days since July 14 1966.

DeActivateArea (string AreaName)

Description: Deactivates the area (or rule) specified by the parameter.

Syntax: void DeActivateArea(string Area*Name*)

Parameters:

AreaName	The name of the area (or rule) which are deactivated.

Example:

```
DeActivateArea(”MyArea”)
```

Deactivates the area (or rule) named “MyArea”.

DblToLng (double x)

Description: Converts a decimal value (*x*) into an integer value by truncating (flooring) the decimal value.

Syntax: long DblToLng(double x)

Parameters:

x	The decimal value that is converted into an integer value.

Example:

```
DblToLng(turnover/nEmployee)
```

Returns the turnover per employee truncated into an integer value.

DblToStr (double x)

Description: Converts a decimal value (*x*) into a string (a text).

Syntax: string DblToStr(double x)

Parameters:

x	The decimal value that is converted into a string.

Example:

```
DblToStr(3.14)
```

Returns the number 3.14 as the text “3.14”

Difference (double x1, double x2)

Description: Returns the difference between x_1 and x_2, i.e. the value calculated as $x_2 – x_1$.

Syntax: double Difference(double x_1, double x_2)

Parameters:

x_1	The value to which the difference to x_2 is calculated.
x_2	The value to which the difference from x_1 is calculated.

Example:

```
Difference(5.5, 7)
```

Returns the number 7 – 5.5. = 1.5

```
Difference(13, 11.3)
```

Returns the number 11.3 – 13 = – 1.7

EndDate (string period)

Description: Returns the end date of the specified period (*period*) as an integer value.

Syntax: long EndDate(string period)

Parameters:

period	A period that complies to the specified pattern: *YYYYMMDD-YYYYMMDD.*

Example:

```
EndDate(”19660714-20100504”)
```

Returns the integer value 20100504

ExecuteArea (string AreaName)

Description: Executes the area (or rule) specified by the parameter. When the specified area (or rule) has been executed the evaluation will continue from the calling point. The area can be located anywhere inside the knowledge base.

Syntax: void ExecuteArea(string Area*Name*)

Parameters:

AreaName	The name of the area (or rule) which are executed.

Example:

```
ExecuteArea(”MyArea”)
```

Executes the area (or rule) named “MyArea”.

FilterString (string s, string trimChars)

Description: Returns the contents of the specified string (*s*) but where all the occurrences of the characters in the specified string (*trimChars*) have been removed.

Syntax: string FilterString(string s, string trimChars)

Parameters:

s	The string which are filtered.
trimChars	A string specifying the characters which shall be removed from *s*.

Example:

```
FilterString(”Intracom IT-Services”, ” IT”)
```

Returns the text “ntracom-Services”

FormatNumber (double value, string format)

Description: Formats the specified value (*value*) into a string using the specified format.

Syntax: string FormatNumber(double value, string format)

Parameters:

value	The value that is formatted into a string
format	A string specifying the format according to Appendix F.

Example:

```
FormatNumber(7/3, ”Amount=%N”)
```

Returns the text “Amount=2”

GetCubeName (CURRENT_CUBE)

Description:	Returns a string containing the name of the specified cube (*cube*). Hereby it is possible to get the name of a cube into a string which then can be used rules.
Syntax:	string GetCubeName(OBJECT cube)

Parameters:

CURRENT_CUBE	The keyword CURRENT_CUBE. Note that this is a keyword of the language and must be spelled exactly like that. This is the only valid input to this argument and specifies the current cube which name should be returned.

Example:

```
GetCubeName(CURRENT_CUBE)
```

Returns a string containing the text: “IncomeTax” if current cube is IncomeTax.

GetNoOfMembers (CURRENT_CUBE, string dimensionName)

Description:	Returns the number of members in the dimension specified by the second parameter *dimensionName*.
Syntax:	double getNoOfMembers(CURRENT_CUBE, string DimensionName)

Parameters:

CURRENT_CUBE	The keyword CURRENT_CUBE. Note that this is a keyword of the language and must be spelled exactly like that. This is the only valid input to this argument! This argument has no effect on the result but is retained for backward compatibility and must be supplied.
dimensionName	The name of the dimension which members are counted.

Example:

```
GetNoOfMembers(CURRENT_CUBE, ”Years”)
```

Returns the number of members in the *Years*-dimension in the current cube.

GetParamDoubleValue(string UsageName)

Description: Returns the result from the specified Parameter Table Usage as a double value.

Syntax: double GetParamDoubleValue(string Usage*Name*)

Parameters:

UsageName The name of the usage which are called.

Example:

`GetParamDoubleValue``(”MyDoubleUsage”)`

Calls the usage named “MyDoubleUsage” and returns the double value returned by the usage.

GetParamIntValue (string UsageName)

Description: Returns the result from the specified Parameter Table Usage as an integer value.

Syntax: integer GetParamIntValue(string Usage*Name*)

Parameters:

UsageName The name of the usage which are called.

Example:

`GetParamIntValue``(”NumberOfTickets”)`

Calls the usage named “`NumberOfTickets`” and returns the integer value returned by the usage.

GetParamStrValue (string UsageName)

Description: Returns the result from the specified Parameter Table Usage as a string.

Syntax: string GetParamStrValue(string Usage*Name*)

Parameters:

UsageName The name of the usage which are called.

Example:

`GetParamStrValue``(”StandardTaxOfficeName”)`

Calls the usage named “`StandardTaxOfficeName`” and returns the string returned by the usage.

GetRandomNo (double min, double max)

Description: Returns a pseudo random number within the range limited by the specified values (*min* as the lower limit and *max* the upper limit).

Syntax: double GetRandomNo(double min, double max)

Parameters:

min	The lower limit of the range within in which the pseudo random number will be given.
max	The upper limit of the range within in which the pseudo random number will be given.

Example:

```
GetRandomNo(20, 30)
```

Returns a number between 20 and 30.

InString (string candidate, string s)

Description: Returns ***TRUE*** if the specified string (*candidate*) is contained in the specified string (*s*); Otherwise ***FALSE*** is returned.

Syntax: boolean InString(string candidate, string s)

Parameters:

candidate	The string for which *s* is searched
s	The string which is searched for the occurrence of *candidate*.

Example:

```
InString(”Abe”, ”Abe Lincoln”)
```

Returns true because “Abe” is contained in “Abe Lincoln”

IsEmpty (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is empty. If it is empty then ***TRUE*** is returned; otherwise ***FALSE***.

Syntax: boolean IsEmpty(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is empty.

Example:

```
IsEmpty(TaxableIncome)
```

Returned ***TRUE*** if no taxable income is declared; otherwise ***FALSE*** is returned.

See also:

AllZeroOrNotKnown
IsKnow
IsKnownAndNotZero
IsNotKnown
IsZeroOrNotKnown

IsKnown (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is known (a.k.a. NULL/EMPTY – I.e. has been assigned a value during data extraction or assignment). If it is known then ***TRUE*** is returned; otherwise ***FALSE***.

Syntax: boolean IsKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is known.

Example:

```
IsKnown(TaxableIncome)
```

Returned ***TRUE*** if no taxable income is known by the system; otherwise ***FALSE*** is returned.

See also:

AllZeroOrNotKnown
IsEmpty
IsKnownAndNotZero
IsNotKnown
IsZeroOrNotKnown

IsKnownAndNotZero (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is known (a.k.a. not NULL/EMPTY – I.e. has been assigned a value during data extraction or assignment). If it is not known then ***FALSE*** is returned; otherwise the function returns ***TRUE*** if the value is not zero; otherwise it returned ***FALSE***.

Syntax: boolean IsNotKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is known and not zero.

Example:

```
IsKnownAndNotZero(TaxableIncome)
```

Returned ***FALSE*** if no taxable income is known by the system; otherwise ***TRUE*** is returned if the taxable income is different from zero.

See also:

AllZeroOrNotKnown
IsEmpty
IsKnown
IsNotKnown
IsZeroOrNotKnown

IsNotKnown (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is not known (a.k.a. NULL/EMPTY – I.e. has not been assigned a value during data extraction or assignment). If it is not known then ***TRUE*** is returned; otherwise ***FALSE***.

Syntax: boolean IsNotKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is not known.

Example:

```
IsKnown(TaxableIncome)
```

Returned ***FALSE*** if no taxable income is known by the system; otherwise ***TRUE*** is returned.

See also:

AllZeroOrNotKnown
IsEmpty
IsKnown
IsKnownAndNotZero
IsZeroOrNotKnown

IsStandardContainerCode (string code)

Description: Interprets the format of the specified container code (*code*). If the container code represents what is considered to be a standard container code then the function returns ***TRUE***; otherwise ***FALSE*** is returned.

As shown in the table below, all prefix numbers have their own values. The value of a number in the container code is equal to the number itself.

Prefix number	Value
A	10
B	12
C	13
D	14
E	15
F	16
G	17
H	18
I	19

Prefix number	Value
J	20
K	21
L	23
M	24
N	25
O	26
P	27
Q	28
R	29

Prefix number	Value
S	30
T	31
U	32
V	34
W	35
X	36
Y	37
Z	38

Values of Prefix Number

The last, seventh number (placed on the container in a small square) is not included, but is the last digit of the calculation and by this the control digit. For example, take the container number:

MWCU 605978-4.

The pre-fix MWCU has the following value:

M = 24
W = 35
C = 13
U = 32

With this the following calculation is made, by which every value is multiplied with twice the number of which the previous value was multiplied (starting with the digit 1).

M	24	x	1	=	24
W	35	x	2	=	70
C	13	x	4	=	52
U	32	x	8	=	256
6	6	x	16	=	96
0	0	x	32	=	0
5	5	x	64	=	320

```
9        9  x  128  =   1152
7        7  x  256  =   1792
8        8  x  512  =   4096
                      ______
                        7858
                      ======
```

The result of the calculation is then divided by eleven and the remainder of the division is the last digit.

```
7858 : 11 = 714
77
 15
 11
  48
  44
   4
```

With shipper owned containers completely different digit combinations are being used, such that this calculation will often give a false outcome.

Syntax: boolean IsStandardContainerCode(string code)

Parameters:

code The code that is interpreted.

Example:

IsStandardContainerCode(”MWCU 605978-4”)

Returns the ***TRUE*** because the specified container code is a standard container code.

IsSubSet (Set SetA, Set SetB)

Description: Determines whether *SetB* is a sub set of *SetA*. If *SetB* is a sub set of *SetA* ***TRUE*** is returned; otherwise ***FALSE*** is returned.

Syntax: boolean IsSuBSet(Set SetA, Set SetB)

Parameters:

SetA	The set to which *SetB* is compared.
SetB	The set which is compared as a sub set to *SetA*.

IsZero (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is zero – I.e. has been assigned a value that is zero during data extraction or assignment). If the value is zero then ***TRUE*** is returned; Otherwise ***FALSE*** is returned.

Syntax: boolean IsZero (TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is zero.

Example:

```
IsZero (TaxableIncome)
```

Returned ***TRUE*** if the declared income is zero; otherwise ***FALSE*** is returned.

IsZeroOrNotKnown (TUPLE Measure)

Description: Determines if the specified measure (*Measure*) is not known (a.k.a. NULL/EMPTY – I.e. has not been assigned a value during data extraction or assignment). If it is not known then ***TRUE*** is returned; otherwise it returns ***TRUE*** if the known value is zero; Otherwise ***FALSE*** is returned.

Syntax: boolean IsZeroOrNotKnown(TUPPLE Measure)

Parameters:

Measure	The measure that the function determines whether it is not known or is zero.

Example:

```
IsZeroOrNotKnown(TaxableIncome)
```

Returned ***TRUE*** if no taxable income is known by the system or if the declared income is zero; otherwise ***FALSE*** is returned.

See also:

AllZeroOrNotKnown
IsEmpty
IsKnown
IsKnownAndNotZero
IsNotKnown

LeftStr (string s, double length)

Description: Returns the first *length* characters from the string (*s*). If *s* is shorter than *length* then the whole *s* is returned.

Syntax: string LeftStr(string s, double length)

Parameters:

s	The string from which the first (left) characters is returned.
length	The maximum number of characters returned.

Example:

```
LeftStr(“Intracom IT-Services”, 8)
```

Returns the text “Intracom”

LngToDbl (long value)

Description: Converts an integer value (*value*) into a decimal value.

Syntax: double LngToDbl (long value)

Parameters:

value	The integer value that is converted into a decimal value.

Example:

```
LngToDbl(36)
```

Returns the decimal value 36.0

LngToStr (long value)

Description: Converts an integer value (*value*) into a string.

Syntax: string LngToStr (long value)

Parameters:

value	The integer value that is converted into a string.

Example:

```
LngToStr(36)
```

Returns the string value “36”

LowerCase (string s)

Description: Returns the contents of the supplied string but with all cases converted into lower case.

The character sets supported depends on the setup of the actual server platform.

Syntax: string LowerCase(string s)

Parameters:

s	The string which is converted into lower cases.

Example:

```
LowerCase(”This is a text”)
```

Returns the text “this is a text”

Maximum (double x_1, double x_2)

Description: Returns the maximum value of the two specified values (x_1 and x_2)

Syntax: double Maximum(double x_1, double $x_{2)}$

Parameters:

x_1	The value returned if it is greater than x_2.
x_2	The value returned if it is greater than x_1.

Example:

```
Maximum(3, 6)
```

Returns the value 6.

MaximumElement (Set s)

Description: Returns the maximum value contained in the set *s*. If *s* is the empty set *NULL* is returned.

Syntax: double MaximumElement(Set s)

Parameters:

s	The set from which the maximum value is returned.

MaximumValue (TUPLE Measure, string DimensionName)

Description: Returns the maximum value of measure *Measure* for the dimension *DimensionName.*

Syntax: double MaximumValue(TUPLE *Measure*, string *DimensionName*)

Parameters:

Measure	The measure member from which maximum value will be derived from the specified dimension (*DimensionName*)**.**
DimensionName	The name of the dimension from which the maximum value is derived.

Example:

```
MaximumValue(TaxableIncome, ”Years”)
```

Returns the maximum taxable income found in the years.

Minimum (double x_1, double x_2)

Description: Returns the minimum value of the two specified values (x_1 and x_2)

Syntax: double Minimum(double x_1, double $x_{2)}$

Parameters:

x_1	The value returned if it is less than x_2.
x_2	The value returned if it is less than x_1.

Example:

```
Minimum(11, 3.14)
```

Returns the value 3.14.

MinimumElement (Set s)

Description: Returns the minimum value contained in the set *s*. If *s* is the empty set *NULL* is returned.

Syntax: double MinimumElement(Set s)

Parameters:

s	The set from which the minimum value is returned.

MinimumValue (TUPLE Measure, string DimensionName)

Description: Returns the minimum value of measure *Measure* for the dimension *DimensionName*.

Syntax: double MinimumValue(TUPLE *Measure*, string *DimensionName*)

Parameters:

Measure	The measure member from which minimum value will be derived from the specified dimension (*DimensionName*)**.**
DimensionName	The name of the dimension from which the minimum value is derived.

Example:

```
MinimumValue(TaxableIncome, ”Years”)
```

Returns the minimum taxable income found in the years.

Nz (double x)

Description: See ValOrZero(double x)

Observation (string Name, Rule Rule, double Likelihood, double Value)

Description: Makes an observation for the current rule with likelihood and a value.

Syntax: Observation(string Name, Rule Rule, double Likelihood, double Value)

Parameters:

Name	The name of the observation that is made.
Rule	The rule which makes this observation. Typically this will be CURRENT_AREA.
Likelihood	The likelihood for the risk identified by the observation
Value	THe risk value identified by the observation

Example:

```
call Observation(”Generic”, CURRENT_AREA, 1, volumn)
```

OnChanged (string key)

Description: Returns ***TRUE*** if the specified key (*key*) is different from the key specified last time the function was called; otherwise the function returns ***FALSE***. If it is the first time the function is called ***TRUE*** is also returned.

Syntax: boolean OnChanged(string key)

Parameters:

key	The key that is compared to the key used in previous call.

Example:

```
OnChanged(”MyKey”)
```

Returns ***FALSE*** if the last call to OnChanged also was done with the key “MyKey”; Otherwise it returns ***TRUE***.

p (string ParameterTableName)

Description: Returns a parameter value from the specified parameter table (*ParameterTableName*) (for the taxpayers industry code).

Syntax: double p(string *ParameterTableName*)

Parameters:

ParameterTableName	The name of the parameter table from which the value is returned.

Example:

```
p(”SalesToIncomeRatio”)
```

Returns the value stored in “SalesToIncomeRatio” parameter table for the tax payers industry code.

PeriodValue (TUPLE Tuple, string StartDate, string EndDate)

Description: Returns a value determine by the following steps:

1. The tuple corresponding to cell specified by *Tuple* is retrieved.
2. The period member is replaced with an extended period member covering the period indicated by *startDate* and *endDate*.
3. The value of the resulting cell is returned.

Syntax: PeriodValue(TUPLE Tuple, string StartDate, string EndDate)

Parameters:

Tuple	The tuple identifying to the cell that is retrieved.
StartDate	The first date of the period from which the value is determined.
EndDate	The end date of the period from which the value is determined.

PeriodValueSet (TUPLE Tuple, string StartDate, string EndDate)

Description: Returns a value determine by the following steps:

1. The tuple corresponding to cell specified by *Tuple* is retrieved.
2. The period member is replaced with an extended period member covering the period indicated by *startDate* and *endDate*.
3. A request is made that the value should be calculated as a set, regardless of the dimension aggregation setting
4. The value set of the resulting cell is returned.

Syntax: PeriodValueSet(TUPLE Tuple, string StartDate, string EndDate)

Parameters:

Tuple	The coordinate to the cell that is retrieved.
StartDate	The first date of the period from which the value is determined.
EndDate	The end date of the period from which the value is determined.

Partition (SET Set, string Operator, UNKNOWN Value)

Description: Returns the set of elements that fulfill some criteria specified by the operator and value argument.

For each value contained in the specified set (*Set*) the value is compared to the specified value (*Value)* by using the specified operator (*Operator*) – only elements which together with the operator and value provides a true equation is add to the returned set.

Syntax: Partition(SET Set, string Operator, UNKNOWN Value)

Parameters:

Set The set of values which are being filtered by the specified operator and value.

Operator The operator (*Operator*) which is used to compare the value in the specified set with the provided value (*Value*).

The following operators are supported:

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Value The value to which each value in the provided set (*Set*) is compared.

Example:

```
Partition(MySet, ">", 13)
```

Returns a sub set of MySet containing only values which are greater than 13.

pa (string ParameterTableName)

Description:	Parameter lookup function. Returns the average of the specified parameter value (*ParameterTableName*) within the years in CalendarYears dimension.
Syntax:	double p(string *ParameterTableName*)

Parameters:

ParameterTableName	The name of the parameter table from which the value is returned.

Example:

```
pa(”SalesToIncomeRatio”)
```

Returns the average value stored in “SalesToIncomeRatio” parameter table for the tax payers industry code.

py (string ParameterTableName, string Year)

Description:	Returns a parameter value from the specified parameter table (*ParameterTableName*), for the year Year.
Syntax:	double py(string *ParameterTableName*)

Parameters:

ParameterTableName	The name of the parameter table from which the value is returned.

Example:

```
py(”IncomeTaxRate”, ”2010”)
```

Returns the income tax rate for the year 2010.

QualifiedAverage (TUPLE Measure, double Threshold, string DimensionName)

Description: Returns the average for values of measure (*Measure*), exceeding the threshold value (*Threshold*), for the dimension (*DimensionName*).

Syntax: double QualifiedAverage(TUPLE Measure, double Threshold, string DimensionName)

Parameters:

Measure	The measure member from which the total deviation will be derived**.**
Threshold	The threshold of the measures which are qualified for the average. Measures below this threshold will not be taken into account.
DimensionName	The name of the dimension from which the qualified average is derived.

Example:

```
QualifiedAverage(TaxableIncome,0,”Years”)
```

Returns the qualified average of the positive taxable income over the years.

QualifiedCount (TUPLE Measure, double Threshold, string DimensionName)

Description: Returns the number of values for measure (*Measure*) that exceed the threshold value (*Threshold*), for the dimension (*DimensionName*).

Syntax: double QualifiedCount(TUPLE Measure, double Threshold, string DimensionName)

Parameters:

Measure	The measure member from which number is counted**.**
Threshold	The threshold of the measures which are qualified for the qualified count. Measures below this threshold will not be counted.
DimensionName	The name of the dimension from which the values are counted.

Example:

```
QualifiedCount(TaxableIncome,0,”Years”)
```

Returns the count of measures of positive taxable income over the years.

randomsample (double rate)

Description: Returns ***TRUE*** by the likelihood specified by *rate*; and ***FALSE*** by the likelihood specified by 1 – *rate*. The function is based on a stochastic pseudo random function.

Syntax: boolean randomsample(double rate)

Parameters:

rate	The probability by which ***TRUE*** is returned.

Example:

```
randomsample(0.2) == true
```

Ratio (double arg1, double arg2, double arg3, double arg4)

Description: Implements a safe division of $\frac{\text{arg}_1}{\text{arg}_2}$.

The result is calculated as follows:

If arg_2 is zero then arg_3 is returned.

If arg_1 or arg_2 (or both) are undefined (null) then arg_4 is returned.

Otherwise the value calculated as $\frac{\text{arg}_1}{\text{arg}_2}$ is returned.

If any errors occur then arg_4 is returned.

Syntax: double Ratio(arg_1, arg_2, arg_3, arg_4)

Parameters:

arg_1	The nominator of the division
arg_2	The denominator of the division
arg_3	The value returned if division by zero
arg_4	The value returned in case of errors or in case of arg_1 or arg_2 (or both) is undefined.

Example:

```
Ratio(3, 6 1, 2)
```

Returns the value 0.5.

```
Ratio(3, 0, 1, 2)
```

Returns the value 1.

RatioChange(TUPLE numerator, TUPLE denom, long year)

Description: Same as RatioIncrease.

RatioDrop (TUPLE numerator, TUPLE denom, long year)

Description: This function returns the negated value of *RatioIncrease*.

Returns the decrease (drop) in the ratio between the passed entries between a year and the following year.

The two passed coordinates must each have exactly one member from a period dimension. This function finds the next period on the same level for both coordinates, and calculates the ratio increase as follows:

$$\frac{numerator}{denom} - \frac{\{numerator, nextperiod\}}{\{denom, nextperiod\}}$$

The functions test for NULL or EMPTY conditions, and returns NULL or EMPTY if the ratio cannot be calculated without zero-division.

Note: an increase in the ratio will result in a negative result.

Syntax: double RatioDrop(TUPLE numerator, TUPLE denom, long year)

Parameters:

numerator	The measure used as numerator.
denom	The measure used as denominator.
year	The number of years over which the ratio is derived.

Example:

```
RatioDrop(GrossReceipts, YearDebtSales, 0)
```

RatioIncrease(TUPLE numerator, TUPLE denom, long year)

Description: Returns the Increase in the ratio between the passed entries between a year and the following year.

The two passed coordinates must each have exactly one member from a period dimension. This function finds the next period on the same level for both coordinates, and calculates the ratio increase as follows:

$$\frac{\{numerator, nextperiod\}}{\{denom, nextperiod\}} - \frac{numerator}{denom}$$

The functions test for NULL or EMPTY conditions, and returns NULL or EMPTY if the ratio can not be calculated without zero-division.

Note: a decrease in the ratio will result in a negative result.

Syntax: double RatioIncrease(TUPLE numerator, TUPLE denom, long year)

Parameters:

numerator	The measure used as numerator.
denom	The measure used as denominator.
year	The number of years over which the ratio is derived.

Example:

```
RatioIncrease(GrossReceipts, YearDebtSales, 0)
```

RightStr (string s, double length)

Description: Returns the last *length* characters from the string (*s*). If *s* is shorter than *length* then the whole *s* is returned.

Syntax: string RightStr(string s, double length)

Parameters:

s	The string from which the last (right) characters is returned.
length	The maximum number of characters returned.

Example:

```
RightStr(“Intracom IT-Services”, 8)
```

Returns the text “Services”

Roundsum (double value)

Description: Returns the number of continuous zeros in the specified value (*value*) taken from the right.

Syntax: double Roundsum(VARIENT value)

Parameters:

value	The value which trailing zeros are counted.

Example:

```
Roundsum(9000)
```

Returns the value 3.0

SetDiff (SET SetA, SET SetB)

Description: Returns the differences between two sets as a new set. The difference is defined as the elements which are exclusive members of *SetA* or *SetB* – not both sets.

Syntax: SET SetDiff(SET SetA, SET SetB)

Parameters:

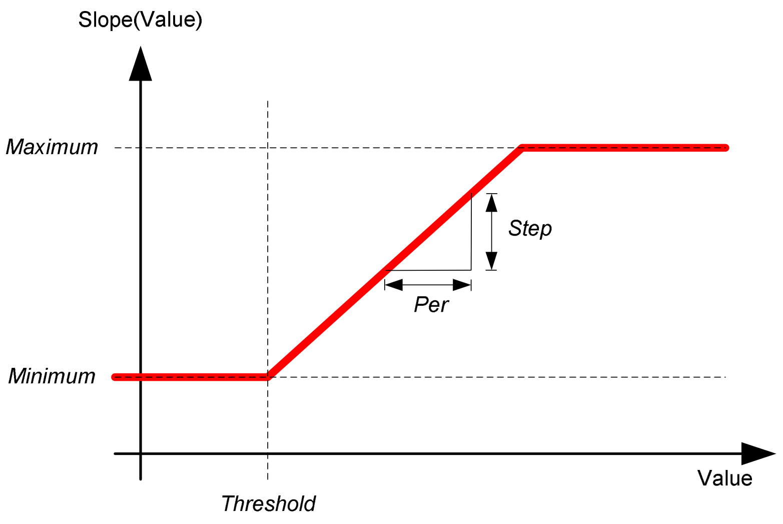
SetA	The first of the two sets which difference is being determined. If a single value is provided it will be treated as a set containing the single value.
SetB	The second of the two sets which difference is being determined. If a single value is provided it will be treated as a set containing the single value.

Slope (double Min, double Max, double Step, double Threshold, double Per, double Value)

Description: Determines a value (e.g. a points score) determined from *Value*. Slope will calculate a value, always within the range *Minimum* to *Maximum*, based on the following principles:

- Step is positive and *Per* is positive:
  The result is determined starting with *Minimum*, and adding *Step* for every *Per* that *Value* exceeds *Threshold*, until *Maximum* is reached.

  Illustration for positive Step and Per:



- *Step* is positive and *Per* is negative:
  The result is determined starting with *Minimum*, and adding *Step* for every |*Per*| that *Value* is lower than *Threshold*, until *Maximum* is reached.
- *Step* is negative and *Per* is positive:
  The result is determined starting with *Maximum*, and subtracting |*Step*| for every *Per* that *Value* is exceeds *Threshold*, until *Minimum* is reached.
- *Step* is negative and *Per* is negative:
  The result is determined starting with *Maximum*, and subtracting |*Step*| for every |*Per*| that *Value* is lower than *Threshold*, until *Minimum* is reached.

Syntax: double Slope(double Minimum, double Maximum, double Step, double Threshold, double Per, double Value)

Parameters:

Min	The minimum return value from the function.
Max	The maximum return value from the function.

Step	The rate at which values above *Threshold* are amplified until they are cut by *Maximum*.
Threshold	The threshold above which the values are amplified.
Per	The denominator which together with the *Step* value forms the slope.
Value	The input value which is mapped by the function.

Example:

```
Slope(0.1,0.5,0.05,10000,1000,myMember)
```

Returns a value which is the value *myMember* limited into the range 0.1- 0.5 for values above 10000. The values above 10000 are amplified by 0.05 for each 1000 until the value 0.5.

SoundsLike (string s_1, string s_2)

Description: Compares two strings (s_1 and s_2) using SOUNDEX (see http://en.wikipedia.org/wiki/Soundex). I.e. by converting each of the strings by using SOUNDEX and then string comparing the two results. Hereby a simple pattern comparison can be made.

Syntax: boolean SoundsLike(string s1, string s2)

Parameters:

s_1	The string which SOUNDEX value is compared to the SOUNDEX value of s_2.
s_2	The string which SOUNDEX value is compared to the SOUNDEX value of s_1.

Example:

```
SoundsLike(”Intracom”, ”Intrekom”)
```

Returns ***TRUE*** because the two words “Intracom” and “Intrekom” sounds a lot like each other.

```
SoundsLike(”Intracom”, ”Entrecote”)
```

Returns ***FALSE*** because the two words “Intracom” and “Entrecote” sound too different.

StartDate (string period)

Description: Gets the start date from a period string (format *YYYYMMDD-YYYYMMDD*) and return it as an integer in the format *YYYYMMDD*.

Syntax: long StartDate(string period)

Parameters:

period	The string that is interpreted as a period and which star date is returned as an integer.

Example:

```
StartDate(”19660714-20100505”)
```

Returns the integer value 19660714

StrToDbl (string s)

Description: Converts (if possible) the specified string into a decimal number.

Syntax: double StrToDbl(string s)

Parameters:

s	The string is converted into a decimal number.

Example:

```
StrToDbl(”7.913”)
```

Returns the double value 7.913

StrToLng (string s)

Description: Converts (if possible) the specified string into an integer.

Syntax: long StrToLng(string s)

Parameters:

s	The string is converted into an integer.

Example:

```
StrToLng(”7.913”)
```

Returns the integer number 7

SubStr (string s, double pos, double length)

Description: Returns a substring as the *length* characters from the string (*s*) starting at position *pos*.

Syntax: string SubStr(string s, double pos, double length)

Parameters:

s	The string from which the substring is returned.
pos	The position of the first characters in the returned substring, where 1 is the first character in the string (*s*).
length	The maximum number of characters returned.

Example:

```
SubStr(”Intracom IT-Services”, 10, 2)
```

Returns the text “IT”

Today ()

Description: Returns the current date as a string in the format *YYYYMMDD*.

Syntax: string Today()

Example:

```
Today()
```

TotalDeviation (TUPLE Measure)

Description: Returns the total deviation for the measure (*Measure*), based on the members on the period dimension.

Syntax: DOUBLE TotalDeviation(TUPLE Measure)

Parameters:

Measure	The measure member from which the total deviation will be derived**.**

Example:

```
TotalDeviation(TaxableIncome)
```

Returns the total deviation of the taxable income over the period dimension.

TotalDeviationPct (TUPLE Measure)

Description: Returns the total deviation percentage for the measure (*Measure*), based on the members on the period dimension.

Syntax: double TotalDeviation(TUPLE Measure)

Parameters:

Measure	The measure member from which the total deviation percentage will be derived**.**

Example:

```
TotalDeviationPct(TaxableIncome)
```

Returns the total deviation percentage of the taxable income over the period dimension.

TrimString (string s)

Description: Returns a copy of the specified string (*s*) but where all trailing spaces, tabs and other whitespace characters have been removed.

Syntax: string TrimString(string s)

Parameters:

s	The string which is trimmed.

Example:

```
TrimString(”Intracom   ”)
```

Returns the text “Intracom”

UpperCase (string s)

Description: Returns the contents of the supplied string but with all cases converted into upper case.

The character sets supported depends on the setup of the actual server platform.

Syntax: string UpperCase(string s)

Parameters:

s	The string which is converted into upper cases.

Example:

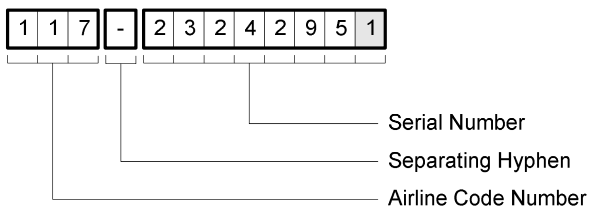
```
UpperCase(”This is a text”)
```

Returns the text “THIS IS A TEXT”

ValidAirWaybillNumber(string awbn)

Description: Validates the Air Waybill Number specified by *awbn*. If *awbn* specifies a valid airway bill number then the function returns ***TRUE***; otherwise the function returns ***FALSE***.

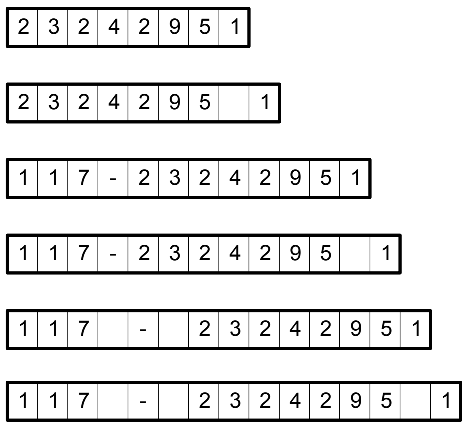
As shown in the figure below the Air Waybill Number consists of the following parts:



The Air Waybill Number

1) Airline Code Number
2) Separating Hyphen
3) Serial Number

The validation is made on the Serial Number only; the function however is implemented to handle both complete Air Waybill Numbers and the Serial Number part only. This is done by letting the function look at the last 8 digits of the filtered *awbn*-parameter only – (Filtered means that if separating white spaces exists they will be ignored) – hereby the following *awbn*'s will be treated equally as the Serial Number 23242951:



A Serial Number is validated by dividing the first 7 digits by 7 (in the example this will be 2324295). For a valid Serial Number the remainder is the same number as the last digit (in the example this is 1). In such case the Serial Number and thereby the whole Air Waybill Number is considered valid and the function returns ***TRUE***; otherwise it is considered invalid and the function returns ***FALSE***.

```
2324295 : 7 = 332042
21
 22
 21
  14
  14
   02
    0
    29
    28
     15
     14
      1
```

Syntax: boolean ValidAirWaybillNumber(string awbn)

Parameters:

awbn The string representing the Air Waybill Number that is validated.

Example:

`ValidAirWaybillNumber``(”117-2324295 1”)`

Returns the ***TRUE*** because 117-2324295 1 is a valid Air Waybill Number.

ValOrZero (double x)

Description: Returns 0 (zero) if the specified value is *NULL*, otherwise value is returned.

Syntax: double ValOrZero(double x)

Parameters:

x The value that returned unless it is *NULL*.

Example:

`ValOrZero``(3.14)`

Returns the value 3.14 because the value is neither *NULL*.

`ValOrZero``(x)`

Returns *NULL* if *x* is unknown (*NULL*); otherwise the double value represented by *x* is returned.

WriteString

Description: Writes the text contents of a given node to the output made by a subsequent observation (see *Observation*). The available nodes depend on the configuration of the system.
During the write of the node contents the associated macros will be expanded in the written text (see *BindMacro*).

Syntax: WriteString (string NodeName)

Example:

```
call BindMacro("Volume", FormatNumber(volume,
"%G3','%D'.'%P2%TZ"));

call WriteString("ReasonOutput");

call Observation("Generic", CURRENT_AREA, 1, volume).
```

Formats the contents of the volume variables and writes the contents of the node named *ReasonOutput* to the associated observation. If the text contents of the *ReasonOutput*-node contains the text %Volume% it will be substituted by the formatted volume value.

Appendix E: Grammar Examples

This appendix presents additional examples, aimed at building an understanding of the BNF notation, as well as answers to the small exercises presented in section 2.1.1.

Simple BNF Examples

The following examples illustrate the use of production rules in grammars described via BNF grammars. Please note, that the example grammars used here has no relation to the formalization language.

Multiplication

The following example illustrates the process involved in producing an expression in accordance with a simple multiplication grammar, through a process of repeated substitution. A similar process can be followed to build valid expressions in any grammar.

Consider the following simple grammar:

<multiplication>	::=	<term> ***** <term>
<term>	::=	<number> | <variable>
<number>	::=	**2 | 3**
<variable>	::=	**a | b**

We start with the non-terminal <multiplication>, for which there is only one alternative, leaving us with:

<term> ***** <term>

To make a valid multiplication expression using the grammar we must then substitute the occurrences of <term>. Since the production rule for <term> contains two alternatives one of them must be chosen for each <term>. For the first substitution we choose <number> and for the second we choose <variable>. This leaves us with:

<number> ***** <variable>

Next we substitute the occurrences of <number> and <variable> with one of the alternatives, e.g leaving us with:

2 * b

The set of possible combinations of substitutions in this grammar is 2 * 2, 2 * 3, 3 * 2, 3 * 3, a * a, a * b, b * b, b * a, 2 * a, 2 * b, 3 * a, 3 * b, a * 2, a * 3, b * 2 and b * 3.

English language sub-set

Consider a grammar for a very small part of English:

The set of productions rules, the following:

```
<A>                    ::=   < B>  <C>  <D>

<B>                    ::=   " I"  |  "You"  |  "We"

<C>                    ::=   "read"  | "wrote"

<D>                    ::=   "this"
```

Using this grammar you can construct six different simple sentences: “I read this”, “You read this”, “We read this”, “I wrote this”, “You wrote this” and “We wrote this”. The set of possible sentences that can be described by the grammar is called the language described by the grammar.

Simple String Example

Consider a simple grammar for generating a string of a’s and b’s:

The set of productions rules are the following:

```
<T>                    ::=   <R> | "a"<T>"a"

<R>                    ::=   "b" | "b" <R>
```

The grammar denotes the set of strings that start with any number of a’s followed by non-zero number of b’s and then the same number of a’s with which it started.

For example are “aaaabaaaa” and “abba” strings generated by the above grammar.

Formalization Language Example

The following example illustrates the process involved in producing a clause from the formalization language through the process of repeated substitution. Lets assume that we want to produce a valid within clause. The starting point is the following production:

```
<area_head_member>  ::=   within  <withinbody>
                          | …
```

giving us the following clause as starting point:

```
within  <withinbody>
```

We now need to substitute <withinbody>, and find the following relevant production rules:

```
<withinbody>           ::=   (  <cubenames>  )  for  <forbody>

<forbody>              ::=   (  <sets>  )
                             | FOR_ALL_KNOWN_CELLS
```

These two production rules tell us that the withinbody construct called contains two non-terminal characters: <cubenames> and <forbody>.

The non-terminal symbol <forbody> is further explained by the rule <forbody> ::= (<sets>), which in turn contains the non-terminal symbol <sets>. We now have the following clause:

```
within ( <cubenames> ) for ( <sets> )
```

From this example it is clear that more rules are required to fully explain the syntax of the <withinbody> construct.

In general, one could say that if non-terminal symbols exist, further definition (more production rules) will be required.

To further explain the <forbody> symbol, we must consider the production rules for <sets>:

```
<sets>                 ::=   <set>
                             | <sets>  ,  <set>

<set>::=               ::=   <set_value_expression>
                             | {  <tuples>  }
```

This first production rule (for <sets>) implies that the non-terminal symbol <sets> either consists of the non-terminal symbol <set> or of the non-terminal symbols <sets> and <set>.

This rule effectively means that <sets> can be replaced with one or more non-terminal <set> symbols, separated by commas.

```
within ( <cubenames> ) for ( <set> , <set> )
```

Using the second production rule (<set>), we can substitute further (e.g. using <set_value_expression>).

```
within ( <cubenames> ) for ( <set_value_expression> , <set_value_expression> )
```

More production rules further explain the syntax:

```
<set_value_expression> ::=  <dimension>
                            | <dimension>  .  MEMBERS
                            | <dimension>  .  EXCEPT  (  <member>  )

<dimension>            ::=  <identifier>

<members>              ::=  <member>
                            | <members>  ,  <member>
```

```
<member>               ::=  <identifier>
                            |
                            | <dimension> . <identifier>
                            | <dimension> . FIRSTMEMBER
                            | <dimension> . LASTMEMBER
                            | <dimension> . PREVMEMBER
                            | <dimension> . NEXTMEMBER
                            | <qualified_dimension> . <identifier>
                            | <cubename> : <identifier>
                            | <member_value_expression>

<identifier>           ::=  <regular_identifier>

<regular_identifier>   ::=  identifier

<cubenames>            ::=  <cubename>
                            | <cubenames> , <cubename>

<cubename>             ::=  <identifier>
```

Combined with the definition of the terminal symbol identifier:

identifier

Any sequence of aplha-numeric characters starting with a non-numeric character (e.g. MyName, First_Name).

We are able to build valid within clauses like:

```
within ( FormX ) for ( Accounts , CalendarYears )
```

or

```
within ( FormX ) for ( Accounts , CalendarYears.FIRSTMEMBER )
```

or

```
within ( FormX ) for ( Accounts, CalendarYears.EXCEPT( CalendarYears.FIRSTMEMBER ) )
```

Answers to Exercises

Consider why the last three examples are not valid, in this grammar

Answer:

The first example presumes that availability of a unary minus operator. To get a negative value you would have to write something like 0 – 5.

The second example uses a lowercase column reference; this is not allowed according to the definition of the cell-address terminal.

The third example uses parantheses, which are not mentioned anywhere in the grammar.

Consider what changes would be required in the grammar to make the expressions valid.

Answer:

Unary minus could be made valid by adding the following alternative to the <expression> production:

```
<expression>      ::=       …
                            | - <expression>
```

Lowercase column identifiers could be made valid by redefining the definition of the cell-address terminal.

The use of paranthesis could be made valid by adding the following alternative to the <expression> production:

```
<expression>      ::=       …
                            | ( <expression> )
```

Appendix F: Numeric Display Format

This appendix describes how the numeric values are formatted into strings.

The display format for numbers provides a way of defining different settings of how numbers such as amounts, decimals and integers should be displayed. The format can have one to four sections with semicolons (;) as the list separator. Each section contains the display format for a different type of number conditions:

Section	Description
First	The format for positive numbers.
Second	The format for negative numbers.
Third	The format for zero value numbers.
Fourth	The format for Unknown value. *Note that the use of this format is not currently implemented!

If you use multiple sections but don't specify a format for each section, entries for which there is no format will default to the formatting of the first section.

Within a formatting section the amount placeholder (%N) may be present. The placeholder specifies where in the format the number will be displayed. Any literal characters before the placeholder will be displayed as prefix to the number and any literal characters after the placeholder will be displayed after the number. If the amount placeholder is not present there is no suffix. The amount placeholder is generally only needed if you have literal characters in the suffix.

Just before the number placeholder you may specify one or more formatting flags, that each specifies a special display setting. See "Formal Format Display Syntax" section below for a more formal definition of the format.

If an invalid display format is specified the displayed text will be "###" indicating an error.

Precision Flag

Flag Syntax	"%P" <Digit> * Note: The current implementation pads with trailing zeros accordingly to precision - see %TZ flag.
Description	Specifies the precision of the amount. This corresponds to the maximum number of significant digits after the decimal point character. If precision is set to 0 the decimal point character will not be used. The amount will be rounded if necessary thus a amount of 1.238 with a specified precision of 2 will be rounded to 1.24. Note that an amount of 1.2 with a specified precision of 2 is 1.2. To print trailing zeros use %TZ flag, e.g. 1.20.
Default setting	The default precision is taken from regional settings - see below.

Digit Grouping

Flag Syntax	"%G" [<Digit>* [""" <Char> """]] * Note: Currently only one digit for group sizes are implemented.
Description	Specifies the digit grouping and the digit grouping separator character. This flag will be ignored if scientific notation is specified. The digit grouping is specified by a series of digits. Each digit specifies the number of digits in a grouping starting from right to left (from the decimal point character). The last digit specifies the grouping for subsequent groups. The 0 digit may only be used as the last digit. Example: Thus a specification of 324 will group first with 3 digits and then with 2 digits followed by subsequent groups of 4 digits. Example "111 22 3333 4444"
Default setting	The default grouping units and the default grouping separator character is taken from the regional settings - see below. Grouping is not used unless %G is specified.

Decimal Point Character

Flag Syntax	"%D" """ <Char> """
Description	Specifies the decimal point character. The decimal point character will not be displayed if precision is set to 0.
Default setting	The default decimal point character is taken from the regional settings - see below.

Scientific notation

Flag Syntax	"%E" [<Sign>] Note: Currently implementation ignores <Sign> and forces the sign on.
Description	Specifies that the number should be displayed with scientific notation. The sign specifies whether + or - after the 'E' should be forced/removed. The exponent will always be displayed with 3 digits padded if necessary with zeros. Note: The exponent is always prefixed with a capital 'E'. Currently it's not possible to control whether the exponent is prefixed with an 'E' or 'e'.
Default setting	Do not display with forced sign. Scientific notation is not used unless %E is specified.

Force or remove sign

Flag Syntax	"%+" or "%-"
Description	In positive number format specification this forces the sign to be presented. In negative number format specifications it removes the sign to be presented. Note: Be careful with only specifying a positive number format using this flag. When using this flag you should always supply a negative number format, unless you're sure that the sign have no importance in negative numbers.
Default setting	No forced sign in positive format and no remove sign in negative format.

Trailing zeros

Flag Syntax	"%TZ" *Note: Current implementation assumes this flag - always.
Description	Specifies that trailing zeros should be padded accordingly to the precision. Thus an amount of 1.2 with precision 2 and trailing zeros will be padded with an extra zero, thus becoming "1.20".
Default setting	No trailing zeros.

Leading zeros

Flag Syntax	"%LZ" <Digit> *Note: Currently not implemented.
Description	Specifies that leading zeros should be padded in front of the first digit in the amount. The digit specifies the maximum number of zeros to pad with. Thus an amount of 1.2 with a leading zero specification of 3 will be displayed as "001.2".
Default setting	No leading zeros.

Multiplication

Flag Syntax	"%M" <Integer>
Description	Specifies that the value should be multiplied with an integer value as specified. The integer value should be different from zero. This can be used in e.g. percent values to show 0.5 as 50 (if integer value is specified as 100).
Default setting	Does not multiply (integer equals zero).

Formal Syntax

```
<Display format string> ::=
          <Positive number format>
          [ ";" <Negative number format> [ ";" <Zero number format>
          [ ";" <Unknown number format> ]]]
<Positive number format> ::=
          <Format>
<Negative number format> ::=
          <Format>
<Zero number format> ::=
          <Format>
<Unknown number format> ::=
          <Format>
```

Format

```
<Format> ::=
          <Prefix> <Flag>* [ <Number placeholder> <Suffix> ]
<Prefix> ::=
          <Characters>
<Suffix> ::=
          <Characters>
<Number placeholder> ::=
          "%N"
```

Format Flags

```
<Flag> ::=
          <Precision> | <Decimal point
char> | <Grouping> | <Scientific notation> |
          <Force sign>
<Precision> ::=
          "%P" <Digit>
<Decimal point char> ::=
          "%D" <Single quote> <Char> <Single quote>
<Grouping> ::=
          "%G" [ <Digit> [ <Single quote> <Char> <Single quote> ]]
<Scientific notation> ::=
          "%E" [ <Sign> ]
<Force sign> ::=
          "%+" | "%-"
```

The Default Settings

The default values used in some of the flags are pr. default taken from the Numbers-tab from the 'Control Panel | Regional Options' dialog within your Windows operating system. This does not apply to UNIX installations.

Examples

This section describes some Number display formats. You should note that some of the examples below the result may depend on the current settings within the regional options dialog, since these are used per default. Also in the examples below it is assumed that the "English (United States)" user locale is selected. Thus the default precision is 2, the default decimal point character is '.', the default grouping character is ',' and the default grouping is "

Display Format	***Sample amount***	***Will be displayed as (excl. quotes)***	***Comments***
"" or "%N"	*123456.789*	*"123457"*	*The most simple display format. No prefix, suffix or formatting flags specified.*

Display Format	***Sample amount***	***Will be displayed as (excl. quotes)***	***Comments***
"" or "%N"	*-123456.789*	*"-123457"*	
"Amt= %N."	*123456.789*	*"Amt=123457."*	*Use of literal characters in prefix and suffix. Prefix equals "Amt= " and suffix equals ".".*
";-;Null;Unknown"	*123456.789*	*"123457"*	
";-;Null;Unknown"	*-123456.789*	*"-123457"*	*Need minus character as prefix.*
";-;Null;Unknown"	*0*	*"Null"*	*Null is displayed literally.*
";-;Null;Unknown"	*Unknown*	*"Unknown"*	*Unknown is displayed literally.*
"%+;%-"	*123456.789*	*"+123457"*	*Use of force sign. It doesn't matter whether %+ or %- are used.*
"%+;(%-)"	*-123456.789*	*"(123457)"*	*Use of force sign. It doesn't matter whether %+ or %- are used.*
"%P2%TZ"	*123456.789*	*"123456.79"*	
"%P2%TZ"	*123456.7*	*"123456.70"*	
"%G"	*123456.789*	*"123 457"*	*Use of default grouping of 3 digits and default grouping separator character " " (space).*
"%G32'.'%D',' %P2"	*123456.789*	*"1.23.456,79"*	
"%G32'.'%D',' %P2"	*123456.7*	*"1.23.456,7"*	
"%G32'.'%D',' %P2%TZ"	*123456.7*	*"1.23.456,70"*	
"%G3','%D'.'%P2%TZ; (%-G3','%D'.'%P2%TZ); %P2%TZ"	*123456.7*	*"123,456.70"*	

Display Format	*Sample amount*	*Will be displayed as (excl. quotes)*	*Comments*
"%G3','%D'.'%P2%TZ; (%- %G3','%D'.'%P2%TZ); %P2%TZ"	-123456.7	"(123,456.70)"	
"%G3','%D'.'%P2%TZ; (%- %G3','%D'.'%P2%TZ); %P2%TZ"	0	"0.00"	
"%E"	123456.789	"1E005"	If precision is 0 within regional settings.
"%E%P3"	123456.789	"1.235E005"	
"%E+%P3"	123456.789	"1.234E+005"	
"%E+%P3"	-123456.789	"-1.234E+005"	
"%E%P3"	0.123456789	"1.235E-001"	
"%E%P3%+"	0.123456789	"+1.235E-001"	
"%E%P3%+"	-0.123456789	"-1.235E-001"	
"%LZ2"	-0.123456789	"-00"	
"%LZ2"	1.23456789	"01"	
"%P2%LZ2"	1.23456789	"01.23"	
"%P2%LZ2"	1.2	"01.2"	
"%P2%LZ2%TZ"	1.2	"01.20"	
"%P2%LZ2%TZ"	1.234	"01.23"	
"%X"	1.234	"###"	Error in display format.
"%M100 %%"	0.25	"25 %"	