DMN 1.1 Reference Guide

Quickly understanding and using the Decision Model and Notation standard

actico
Smarter Decisions

Table of Content

Decision Requirements Diagram & Decision Services **2**
Decision Requirements Diagram (DRD) 2
Decision Services 2
Information Items 2
Boxed Expressions **4**
Data Types **8**
Null 8
Basic Data Types 9
Item Definitions 9
FEEL Expressions **10**
Arithmetic 10
Ranges and Range Comparison (Interval) 10
Logical Comparison 10
Conjunction & Disjunction 11
Semantics of Date, Time, Date and Time and Duration Properties 11
Other 13
FEEL Function Reference **16**
Conversion Functions 17
List Functions 19
Sort Function 20
Boolean Function 20
String Functions 21
Numeric Functions 21

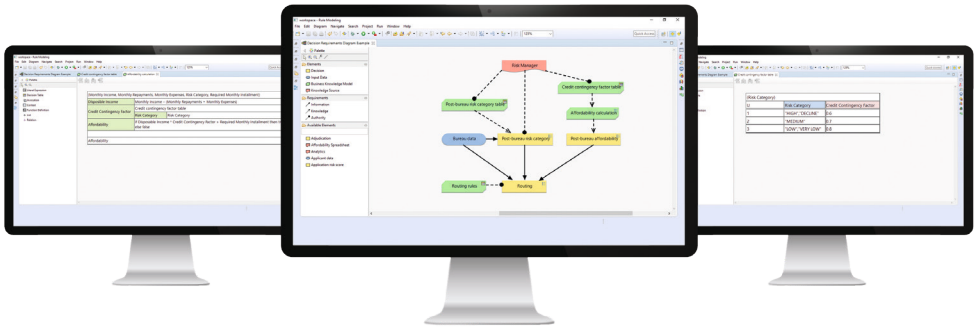
Welcome to the DMN 1.1 Reference Guide!

With this reference guide, we want to help decision modelers around the globe successfully adopt the Decision Model and Notation (DMN). DMN is a standard defined by the Object Management Group (OMG®). It defines a business-friendly notation to describe how decisions are made. It also defines a way to express the actual decision logic used to make decisions, allowing companies to automate operational decision-making.

DMN key concepts are:

- **Decision Requirements Diagram (DRD)**: The standard defines Decision Requirements Diagrams (DRD) to illustrate business decisions, the information required to make these decisions and their dependencies.
- **Decision Logic**: The standard defines how the actual decision logic of individual decisions can be described using so-called „boxed expressions“. This includes but is not limited to decision tables.
- **Expression Language (FEEL)**: Finally, the standard defines an expression language named FEEL (Friendly Enough Expression Language) that defines how to express the conditions and calculations in the decision logic.

ACTICO Platform fully supports all parts of the DMN standard. You can create DMN models with multiple Decision Requirement Diagrams (DRD), describe the decision logic using all boxed expressions defined by DMN, and use the full FEEL expression language.



Start with DMN now! www.actico.com/actico-platform

Decision Requirements Diagram & Decision Services

Decision Requirements Diagram (DRD)

The *Decision Requirements Diagram* (DRD) is a business-friendly illustration of decisions and their dependencies. It can be used to describe human or automated decision-making or a mix thereof.

- A DRD often has a tree-like structure with the main decision at the top. However, you can draw the diagram in any way you like so that it makes sense to you.
- A DRD may show only a subset of the elements of a DMN model. It may show also elements of imported (other) DMN models.
- Names are used for *Decisions*, *Input Data*, *Business Knowledge Models*, *Context* entries, *Relation* columns, *Function* parameters, *Decision Table* output clauses and *Item Definitions*.
- Names are case-sensitive and must not start with a keyword.
- Names must be unique within a model (namespace).
- Names can contain upper- and lowercase letters and digits. They can also include single spaces, dashes (-), plus signs (+), asterisks (*), dashes (/), apostrophes ('), dots (.) and ampersands (&).

Decision Services

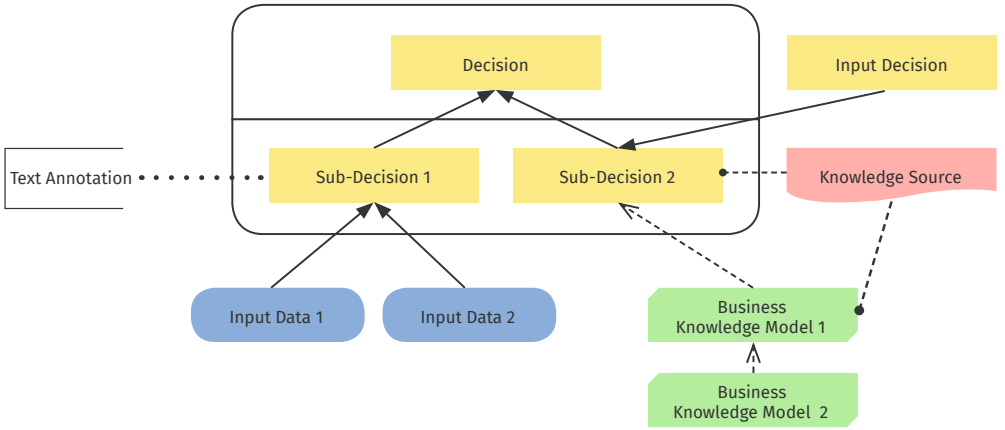
Decision Services are a layer on top of the model illustrated as a box with two compartments.

- The top compartment contains evaluated *Decisions* whose results shall be the result of the *Decision Service*.
- The bottom compartment contains all *Decisions* that shall be evaluated during *Decision Service* execution but are not part of the result.
- Any *Decision* and *Input Data* elements outside the *Decision Service* box with information requirements to *Decisions* inside the *Decision Service* are inputs to the *Decision Service*. Their values need to be provided when calling the *Decision Service*.
- *BKM*s or *Knowledge Sources* can be placed anywhere. Their location has no effect on the definition of the *Decision Service*.

Information Items

Information Items are variables and consist of a name and a type (basic or custom).

- An *Information Item* must be defined for an *Input Data*, *Decision*, *BKM*, *Context* entry, *Relation* column, *Function Definition* parameter, *Function Invocation* parameter, and also for *Decision Table* output columns, if the *Decision Table* has more then one output column. During evaluation a value is assigned to an *Information Item* and it can be accessed via its name.
- Nested name access for nested *Item Definitions* is possible via qualified names.
- For a *Decision* and *BKM* all incoming *Information Requirements* and *Knowledge Requirements* are in scope. For a boxed expression all *Information Items* that are defined before and above that boxed expression are in scope.



A *Decision* represents the act of determining an outcome from several inputs using decision logic.

Input Data denotes the information needed as input by one or multiple *Decisions*.

A *Business Knowledge Model* (BKM) represents reusable business logic. It can be invoked from *Decisions*, other *BKMs or FEEL expressions*.

Knowledge Sources represent authorities for a *Decision*, a *BKM* or another *Knowledge Source*, e.g. policies, regulations or people.

Information Requirements connect an *Input Data* or a *Decision* with a *Decision* that needs the *Input Data* or *Decision*.

A *Decision Service* defines a technical boundary for execution and automation of *Decisions*.

Knowledge Requirements are used to invoke a *BKM*. They point from the *BKM* to the *Decision* or *BKM* invoking it.

Authority Requirements point from a *Knowledge Source* to other elements that are influenced by the *Knowledge Source*.

Text annotation

Text Annotations are used to add explanations or comments.

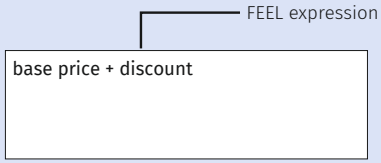
An *Association* links a *Text Annotation* to a DRD element.

Boxed Expressions

The DMN standard defines how the actual decision logic of individual *Decisions* and *BKMs* can be described using so-called „Boxed Expressions“. Boxed expressions can be nested within other boxed expressions. However, *Literal Expressions* and *Decision Tables* do not allow nesting.

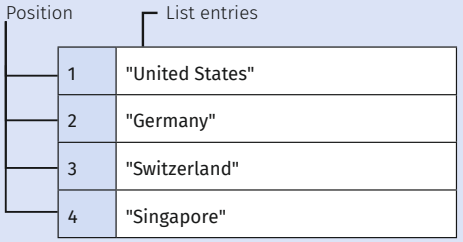
Literal Expression

A *Literal Expression* is a box containing just one expression that defines how an output value is derived from its input values. Almost every box within the other boxed expressions is a *Literal Expression*.



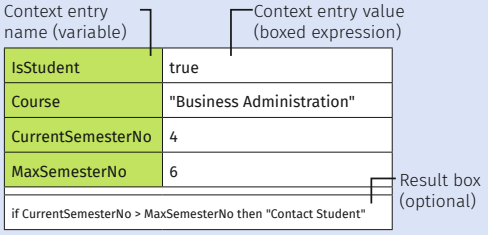
List

A *List* is used to represent multiple values. It is represented as a vertical list of boxed expressions that are numbered starting from 1.



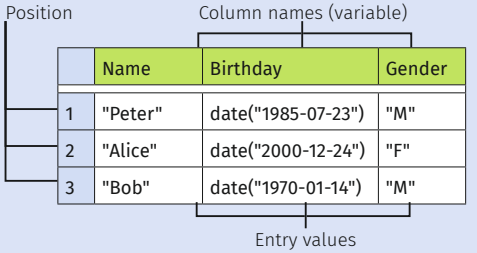
Context

A *Context* is a table with two columns with an optional result box at the bottom. A *Context* allows to define names for partial or intermediate results. This way, decision logic can be broken down into smaller steps.



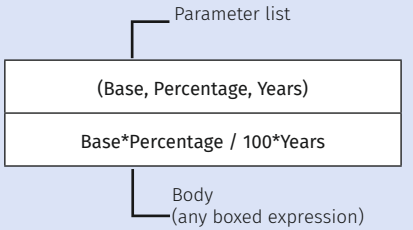
Relation

A *Relation* is like a spreadsheet or a relational database table. It is a list but every element is a *Context* with the same entries. Elements are vertically listed and numbered. Every row is an element and specifies the values for its *Context* entries in its columns.



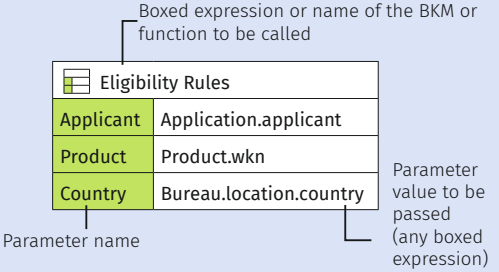
Function Definition

A *Function Definition* allows to define a custom function. It can be invoked either from *Literal Expressions* using FEEL or from a *Function Invocation* boxed expression. The *Function Definition* consists of two cells: a parameter list in the top and the body of the function in the bottom cell.



Function Invocation

Allows to call a *BKM*, a *Function Definition* or a FEEL built-in function, pass parameters and receive the result.
A *Function Invocation* is similar to a FEEL function call. However, a *Function Invocation* requires at least one parameter while FEEL can call functions without parameters.



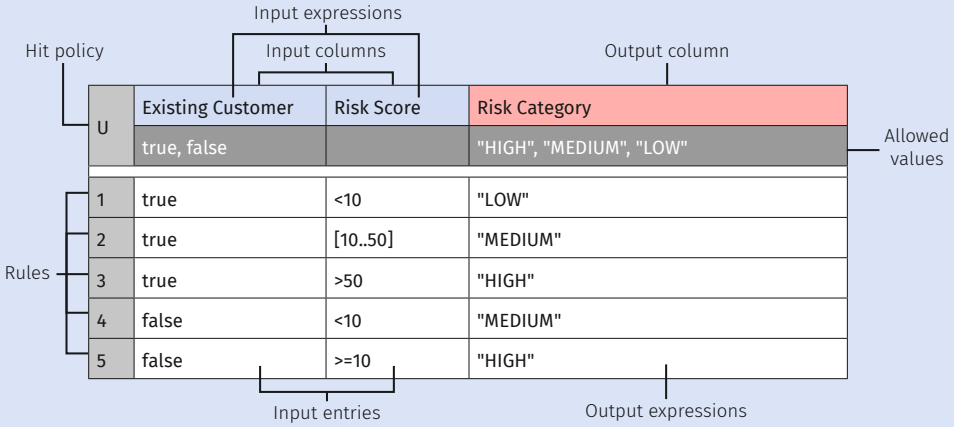
Boxed Expressions

Decision Table

A *Decision Table* is a tabular representation of multiple rules to make a decision. The rules in a *Decision Table* are numbered starting from 1. Rules fire based on the values of one or multiple inputs (blue input columns). In its simplest form, the rules (= rows or columns, depending on the orientation) of the *Decision Table* define different conditions for the inputs and if all of a rule's conditions are fulfilled, the *Decision Table* produces the output values specified in one or multiple output columns (red) of that rule. However, depending on the *Hit Policy* of the *Decision Table*, its behavior may be different from that.

Note:

- The rule conditions (input entries) are so-called *Unary Tests*.
- For a *Decision Table*, a default output value can be defined for an output column that is used if no rule matches.
- Input expressions and output expressions can optionally be restricted by specifying *Allowed Values*. *Allowed Values* are *Unary Tests*. They can be separated by comma.



Hit Policies

U: Unique	Only a single rule can match. Otherwise the *Decision Table* fails.
A: Any	Multiple rules can match that must all produce the same result. This result is returned. If matching rules produce different results, the decision table fails.
F: First	Multiple rules can match. The result of the first matching rule is returned.
P: Priority	Multiple rules can match and they can produce different results. Only one result is returned which is the first to appear in the list of allowed values.
O: Output Order	A list of the results of all matching rules is returned, in the order of decreasing priority. Priority is determined by the list of allowed values.
R: Rule Order	A list of the results of all matching rules is returned, in the order of the rules.
C: Collect	A list of the results of all matching rules is returned.
C+: Collect (Sum)	The sum of the results of all matching rules is returned.
C<: Collect (Minimum)	The minimum of the results of all matching rules is returned.
C>: Collect (Maximum)	The maximum of the results of all matching rules is returned.
C#: Collect (Count)	The number of matching rules is returned.

Data Types

Basic Data Types

Name	Example	Description
number	23 -766.991 0.0006544	Decimal number (with up to 34 digits of precision).
boolean	true false	Logical value of either 'true' or 'false'.
string	"Honolulu" "4200 Main St."	Sequence of characters (text). It is written with double quotes and can contain unicode characters. Special characters like new line are not allowed. No escaping is possible.
date	date(1970,1,14) date("1970-01-14")	Value consisting of year, month (1-12), day (1-31). A date value can also be assigned to an information item or parameter of type date and time.
time	time("12:45:00@Europe/Paris") time("08:00:00+02:00") time("12:45:00") time(12,45,0,duration("PT2H"))	Value consisting of hours, minutes, seconds, optionally including fractions of a second, a timezone or a timezone offset from GMT.
date and time	date and time("2017-12-31T11:22:33@Europe/Paris") date and time("2017-12-31T08:00:00+02:00") date and time("2017-12-31T11:22:33")	Combination of a 'date' and 'time' value, optionally including fractions of a second, a timezone or a timezone offset from GMT.
days and time duration	duration("P1DT2H3M4.123456789S")	Value specifying a time period in days, hours, minutes, seconds, optionally including fractions of a second.
years and months duration	duration("P1Y2M")	Value specifying a time period in years and months.

Null

Null	DMN allows every *Information Item* or expression result to be *null*. Consequently, the possible values for each of the *Basic Data Types* always include the *null* value. Note: Whenever a FEEL expression or boxed expression can not be evaluated due to an error condition, its value is null.

Item Definitions

Item Definitions	*Item Definitions* are custom types for *Information Items* that can be used in addition to the *Basic Data Types*. An *Item Definition* can reference another data type (basic or custom) or it consists of other nested *Item Definitions*. An *Item Definition* can be a collection and may define allowed values, which are a list of *Unary Tests*.

FEEL Expressions

Arithmetic

Name	Example	Description
+	10+5	Addition
-	10-5	Subtraction
*	n*5	Multiplication
/	10/5	Division
**	n**5	Exponentiation
-	-(2+3)	Negation

Logical Comparison

Name	Example	Description
=	a = b	Equality
!=	a != b	Inequality
<	a < b	Less than
>	a > b	Greater than
<=	a <= b	Less than or equal to
>=	a >= b	Greater than or equal to

Conjunction & Disjunction

a	b	a and b	a or b
true	true	true	true
true	false	false	true
true	otherwise	null	true
false	true	false	true
false	false	false	false
false	otherwise	false	null
otherwise	true	null	true
otherwise	false	false	null
otherwise	otherwise	null	null

Ranges and Range Comparison (Interval)

Name	Example	Description
[], (), [), (],]], [[	[1.. 10] [2..5) [2..5) = [2..5[	A *range* defines an interval. A round bracket excludes the endpoint. A square bracket includes or excludes the endpoint, depending on the direction of the square bracket.
in	2 in (2..5] → false 2 in (>0, [10..20], 2)	The *in* expression can be used to test if an expression fulfills a unary test or a single unary test in a list of unary tests.
[from..to]	{startDate: date("2018-01-01"), endDate: date("2018-01-31"), range: [startDate..endDate] }	Endpoints can be names of *Information Items* (e. g. to create ranges of dates or times).
between ... and ...	x between a and b 5 between 0 and 100	Another way to define a range check. Between always includes the endpoints.

Semantics of Date, Time, Date and Time and Duration Properties

Name	*e* . name	Property Names
date	Result is the named component of the *date* object *e*. Valid names are shown to the right.	year, month, day
date and time	Result is the named component of the *date and time* object *e*. Valid names are shown to the right. *time offset* and *timezone* may be null.	year, month, day, hour, minute, second, time offset, timezone
time	Result is the named component of the *time* object *e*. Valid names are shown to the right. *time offset* and *timezone* may be null.	hour, minute, second, time offset, timezone
years and months duration	Result is the named component of the *years and months duration* object *e*. Valid names are shown to the right.	years, months
days and time duration	Result is the named component of the *days and time duration* object *e*. Valid names are shown to the right.	days, hours, minutes, seconds

FEEL Expressions

Other

Name	Example	Description
List [n1, n2, n3]	["Peter", "Lisa", "Pepe"]	A list of values. The empty *List* is just written as []. *Lists* can be nested. A *List* with a single element behaves also like the single element and vice versa.
Context {key1: expr1, key2: expr2}	{name: "Peter", age: 34}	A *Context* defines structured data. Each entry is a pair of name and value. *Contexts* can be nested. An empty *Context* is just written as {}.
Path .	Customer.Age {name: „Max“, result: string length(name)}.result → 3 [{a: 1, b: true}, {a: 2, b: false}].b → [true, false]	Use the dot '.' to access an individual component, entry or result. A full path expression is called a *Qualified Name*. The first part of a *Qualified Name* can be the namespace prefix of an imported DMN model (e.g. lib.max(5,2,8)).
Filter list[condition]	[1, 2, 3, 4, 5, 6][item > 4] → [5, 6] [{x:1, y:2}, {x:2, y:3}][x=1] → [{x:1, y:2}] [1, 2, 3, 4][-2] → 3	Use a *Filter* to find the elements of a list that satisfy a condition. Use ‘item‘ to refer to individual elements or, if the elements are *Contexts*, use the *Context* entry name, e.g. age. If condition is a number value, it defines the position of the element that is filtered. Negative positions are allowed. -1 is the last element.
Some and Every *some/every in* list *satisfies* condition	some User in Users satisfies User.Age < 40	Use ‘*some*’ or ‘*every*’ to check if elements in a list satisfy a condition. It returns either *true* or *false*. Multiple lists can be specified.
For *for* variable *in* list *return* expression	for i in [1,2,3] return i * 2 → [2,4,6]	Use ‘*for*’ to process all items from a list. ‘*for*’ is often used in *Decisions* to call decision logic in a *BKM* multiple times for every element of a list. If multiple lists are specified, the return expression is called for the cartesian product of all list items.
If *if* condition *then* expression *else* expression	if Balance > 0 then "ok" else "not ok"	Use ‘*if*’ to check a condition and return one thing or the other. ‘*else*’ expression is evaluated if ‘*if*’ expression is not true (e.g. null). Use ‚*if*’ only for simple checks and resort to *Decision Tables* when things get more complicated.
Instance of expression *instance of* type	if Value instance of number then Value else 1.0	Use the ‘*instance of*’ operator to check if a value is of a certain data type (basic or *Item Definition*), e.g. if the value is a number, a valid date or a year and months duration.

FEEL Expressions

Other

Name	Example	Description
Unary Test	>10 "DE" [1..100] ["Germany", "USA", "India"] not("Germany", "USA", "Singapore")	Use *Unary Test* to just check a condition. • A list of *Unary Tests* can be specified for allowed values in an *Item Definition*, for the allowed values of an input expression and output expression in a *Decision Table* and for the FEEL *in* expression. • A single *Unary Test* can be used as input entry in a *Decision Table* and for the FEEL *in* expression. • To evaluate a *Unary Test*, a left operand is necessary. Depending on the context, it is automatically set. The result of a *Unary Test* is either *true* or *false*. • The *Unary Test* '-' is always *true*. In a *Decision Table*, the dash marks an irrelevant input. • A *Unary Test* can be a range, an endpoint (e.g. 3, *true*, "ACTICO") or a comparison with missing left operand (e.g. > 10). Available operators are: <, <=, >, >=. • To negate a *Unary Test*, use '*not*' and parentheses around the whole test.
Function Definition *function*(param1, param2) <body> *function*(param1, param2) *external* { java: { class: "<classname>", method signature: "<methodname>(paramType1, paramType2)" } }	{ add: function(a,b) a + b }.add(3,5) → 8 { max: function(a, b) external { java: { class: "java.lang.Math", method signature: "max(int, int)" } } }.max(-1, 5) → 5	Use a *Function Definition* to define custom functions or to define how public static Java methods are called. Functions should be modeled using a *Function Definition* boxed expression whenever possible Use *user-defined functions* to define custom functions. • Use parentheses to define parameter names and replace <body> with the function body. • They can be called by name, if they are assigned to an *Information Item*, for example a *Context* entry. Use *externally-defined functions* to define how public static Java methods are called.
Function Invocation positional: *function*(param 1, param2) named: *function*(name2: param2)	string length("actico") max([4,9,2,1]) list contains(element: 1, list: [5,7,-1,1])	Allows you to invoke a *Function*, that is either a built-in function listed in the *FEEL Function Reference*, a custom function or a *BKM*. Parameters can be passed positional or by name. By passing it per name, not all parameter values must be specified. For missing parameters *null* is used.

FEEL Function Reference

Conversion Functions

Name	Example	Description
number(*from* : string, *grouping separator* : string, *decimal separator* : string)	number("1 000,0", " ", ",") → 1000.0	convert *from* to a number
string(from : any)	string(1.1) → "1.1"	convert *from* to a string
date(*from* : string)	date("2012-12-25") – date("2012-12-24") → duration("P1D")	convert *from* to a date
date(*from* : date and time)	date(date and time("2012-12-25T11:00:00Z")) → date("2012-12-25")	convert *from* to a date (set time components to null)
date(*year* : number, *month* : number, *day* : number)	date(2012, 12, 25) → date("2012-12-25")	creates a date from *year, month, day* component values
date and time(*date* : date or date and time, *time* : time)	date and time (date(„2012-12-24“), time(„23:59:00“)) → date and time („2012-12-24T23:59:00“)	creates a date and time from the given *date* and the given *time*
date and time(*from* : string)	date and time("2012-12-24T23:59:00") + duration("PT1M") → date and time("2012-12-25T00:00:00")	convert *from* to a date and time
time(*from* : string)	time("23:59:00z") + duration("PT2M") → time("00:01:00Z")	convert *from* to time
time(*from* : date and time)	time(date and time("2012-12-25T11:00:00Z")) → time("11:00:00Z")	convert *from* to time (ignoring date components)
time(*hour* : number, *minute* : number, *second* : number, *offset* : days and time duration)	time(23, 59, 0, duration("PT0H")) → time("23:59:00Z")	creates a time from the given component values
duration(*from* : string)	duration("P2Y14M") → duration("P3Y2M") duration("P2D") duration("PT1H2M3.456S")	convert *from* to a days and time or years and months duration
years and months duration (*from* : date, *to* : date)	years and months duration(date("2011-12-22"), date("2013-08-24")) → duration("P1Y8M")	return years and months duration between *from* and *to*
years and months duration (*from* : date and time, *to* : date and time)	years and months duration(date and time("2011-12-22"), date and time("2013-08-24")) → duration("P1Y8M")	return years and months duration between *from* and *to*
get entries(*m* : context)	get entries({a: 1, b: true}) → [{key: "a", value: 1},{key: "b", value: true}]	returns the context entries as a relation with the keys "key" and "value".

List Functions

Name	Example	Description
list contains (*list* : list, *element* : any)	list contains([1,2,3], 2) → true	does the *list* contain the *element*?
count(*list* : list)	count([1,2,3]) → 3	return size of *list*, or 0 if *list* is empty.
min(*list* : list) **min**(c_1 : any,... ,c_n : any) **max**(*list* : list) **max**(c_1 : any,... ,c_n : any)	min([1,2,3]) → 1 min(1,2,3) → 1 max([1,2,3]) → 3 max(1,2,3) → 3	return minimum or maximum item from *list* (or from c_1,...,c_n), or **null** if list is empty.
sum(*list* : list) **sum**(n_1 : number,... ,n_n : number)	sum([1,2,3]) → 6 sum(1,2,3) → 6	return sum of numbers, or **null** if *list* is empty.
mean(*list* : list) **mean**(n_1 : number,... ,n_n : number)	mean([1,2,3]) → 2 mean(1,2,3) → 2	return arithmetic mean (average) of numbers, or **null** if *list* is empty.
and(*list* : list) **and**(b_1 : boolean,... ,b_n : boolean)	and([false,null,true]) → false and([]) → true	return *false* if any item is *false*, else *true* if empty or all items are *true*, else **null**. In DMN 1.2 this function is renamed to *all()*. ACTICO supports both names.
or(*list* : list) **or**(b_1 : number,... ,b_n : number)	or([false,null,true]) → true or([]) → false	return *true* if any item is *true*, else *false* if empty or all items are *false*, else **null.** In DMN 1.2 this function is renamed to *any()*. ACTICO supports both names.
sublist(*list* : list, *start position* : number, *length?* : number)	sublist([4,5,6], 1, 2) → [4,5]	return list of *length* (or all) elements of *list*, starting at start position. 1st position is 1, last position is -1. Parameter *length* is optional.
append(*list* : list, *item...* : any)	append([1], 2, 3) → [1,2,3]	*return* new *list* with *items* appended. *items* can be *null.*
concatenate(*list...* : list)	concatenate([1,2],[3]) → [1,2,3]	return new *list* that is a concatenation of the arguments.
insert before(*list* : list, *position* : number, *newItem* : any)	insert before([1,3],1,2) → [2,1,3]	return new *list* with *newItem* inserted at position.
remove(*list* : list, *position* : number)	remove([1,2,3], 2) → [1,3]	*list* with item at position removed.
reverse(*list* : list)	reverse([1,2,3]) → [3,2,1]	reverse the *list.*
index of(*list* : list, *match* : any)	index of([1,2,3,2],2) → [2,4]	return ascending list of *list* positions containing *match.*
union(*list...* : list)	union([1,2],[2,3]) → [1,2,3]	concatenate with duplicate removal.
distinct values(*list* : list)	distinct values([1,2,3,2,1]) → [1,2,3]	duplicate removal.
flatten(*list* : list)	flatten([[1,2],[[3]], 4]) → [1,2,3,4]	flatten nested lists.

String Functions

Name	Example	Description
substring(*string* : string, *start position* : number, *length?* : number)	substring("actico",3) → "tico"	return *length* (or all) characters in *string*. Parameter *length* is optional.
string length(*string* : string)	string length("act") → 3	return length of *string*
upper case(*string* : string)	upper case("aBc4") → "ABC4"	return uppercased *string*
lower case(*string* : string)	lower case(„aBc4“) → „abc4“	return lowercased *string*
substring before(*string* : string, *match* : string)	substring before("actico", "ico") → "act"	return substring of *string* before the *match* in *string*
substring after(*string* : string, *match* : string)	substring after("actico", "ti") → "co"	return substring of *string* after the *match* in *string*
contains(*string* : string, *match* : string)	contains("actico", "ca") → false	does the *string* contain the *match*?
starts with(*string* : string, *match* : string)	starts with("actico", "ac") → true	does the *string* start with the *match*?
ends with(*string* : string, *match* : string)	ends with("actico", "o") → true	does the *string* end with the *match*?
replace(*input* : string, *pattern* : string, *replacement* : string, *flags?* : string)	replace("abcd", "(ab)|(a)", "[1=$1][2=$2]") → "[1=ab][2=]cd"	regular expression pattern matching and replacement. Parameter *flags* is a string with the following options: s, m, i, x. Parameter *flags* is optional.*
matches(*input* : string, *pattern* : string, *flags?* : string)	matches("actico", "^AC*T", "i") → true	does the *input* match the regexp *pattern*? Parameter *flags* is a string with the following options: s, m, i, x. Parameter *flags* is optional.*

Sort Function

Name	Example	Description
sort(*list* : list, *precedes?* : function)	sort(list: [3,1,4,5,2], precedes: function(x,y) x < y) → [1,2,3,4,5]	sort a *list* using an ordering function *precedes*, which must be a boolean function with 2 arguments.

Boolean Function

Name	Example	Description
not(*negand* : boolean)	not(true) → false not(null) → null	Logical negation

Numeric Functions

Name	Example	Description
decimal(*n* : number, *scale* : number)	decimal(1/3, 2) → .33 decimal(1.5, 0) → 2	return *n* with given *scale*. *scale* is in the range [-6111..6176].
floor(*n* : number)	floor(1.5) → 1 floor(-1.5) → -2	return greatest integer <= *n*
ceiling(*n* : number)	ceiling(1.5) → 2 ceiling(-1.5) → -1	return smallest integer >= *n*

* Valid flag options: s (dot all mode), m (multiline), i (case insensitive), x (whitespace removal)

ACTICO

Europe

ACTICO GmbH
Ziegelei 5
88090 Immenstaad
Germany

info@actico.com
www.actico.com

America

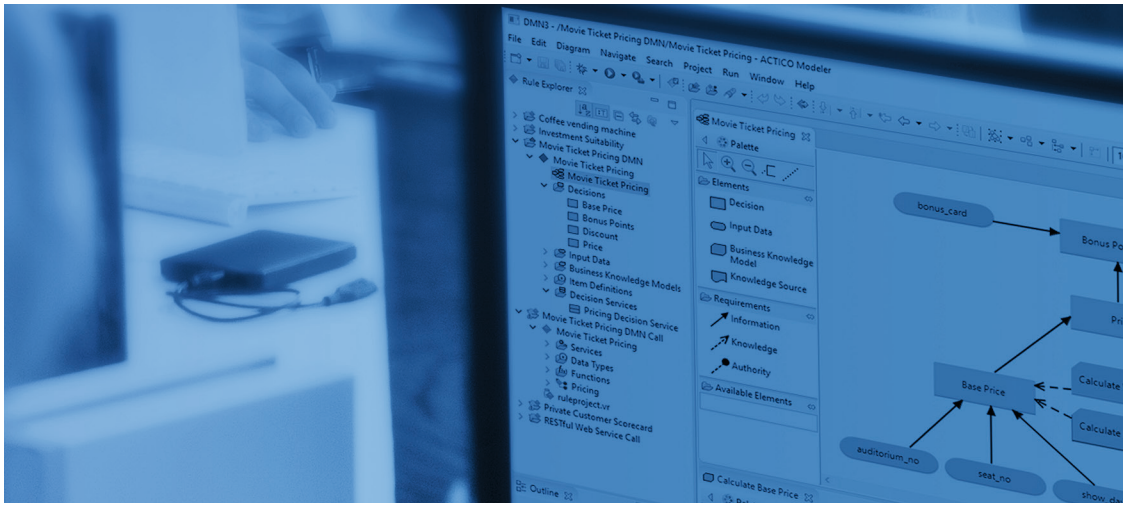
ACTICO Corp.
200 S. Wacker
Dr. Suite 3100
Chicago, IL 60606/USA

info@actico.com
www.actico.com

Asia

ACTICO Pte. Ltd.
#11 - 04, The Arcade
11 Collyer Quay
049317 Singapore

info@actico.com
www.actico.com



ACTICO is a leading international provider of software solutions and technologies for decision management.
In a digital world, it is necessary to process large volumes of data and make real-time, consistent and auditable decisions. ACTICO software allows companies to implement highly flexible applications to optimize their daily decision-making on a continuous basis. This enables them to accelerate growth, innovate effectively, stay compliant and as a result, increase profits.

ACTICO offers solutions in these areas:

- Credit Risk Management: Monitor, assess and manage credit risk
- Loan Origination: Automate credit decisions
- Compliance: Enable transparency, avoid fraud and comply with regulations
- Client Management: Process sensitive customer data securely - from onboarding to reporting
- Underwriting & Claims: Make claim settlement processes quicker, consistent and cost-effective

Since 1997, ACTICO has delivered software and services to our customers' benefits. Headquartered in Germany with offices in USA and Singapore.
More information at **www.actico.com**