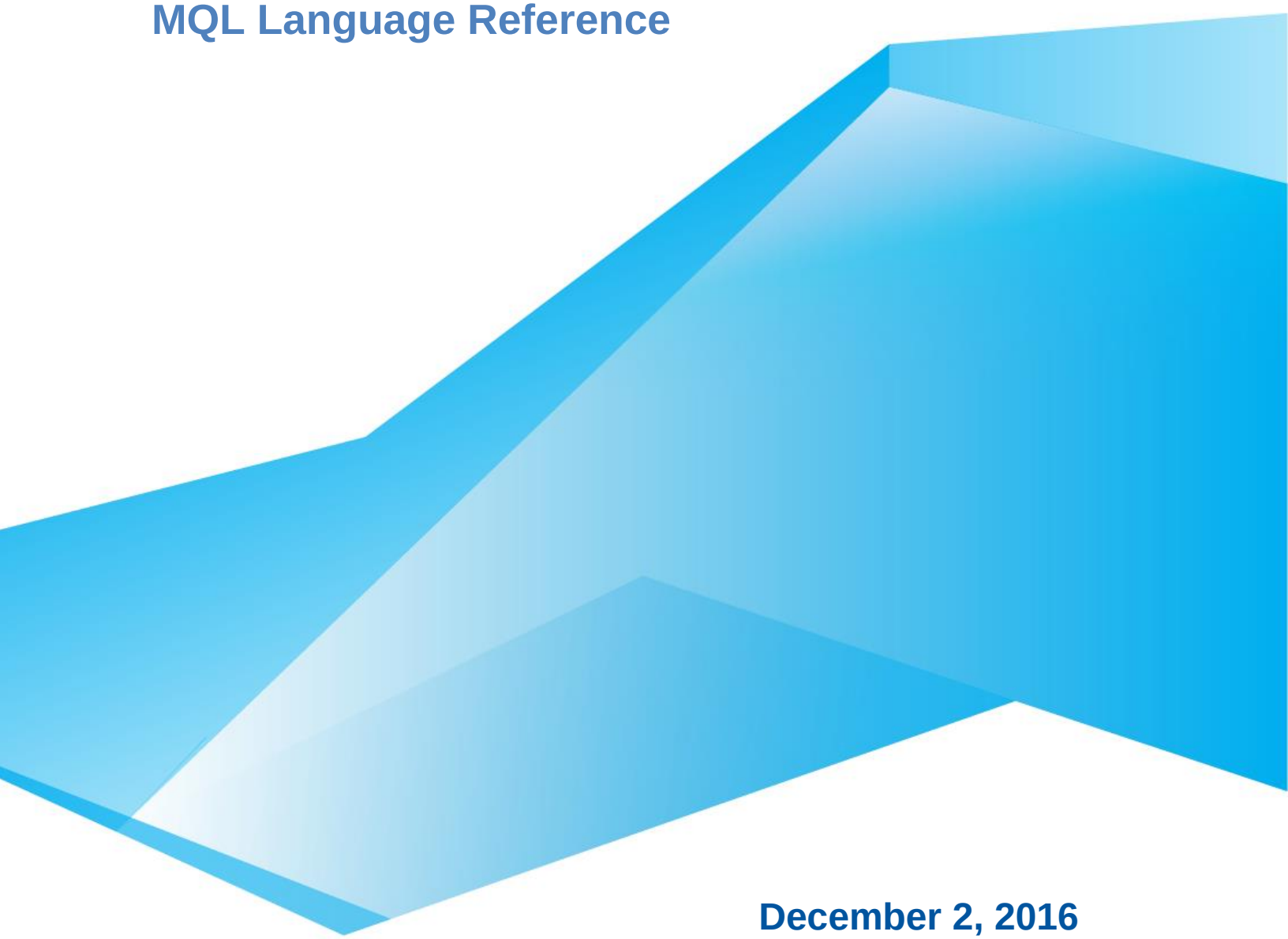
Deltek Know more. Do more.™

Deltek Maconomy 2.3 GA

MQL Language Reference



December 2, 2016

This edition published December 2016.

Contents

Introduction

The Maconomy Query Language (MQL) is a language for interacting with the Maconomy database. Currently only data selection is supported by MQL.

MQL, Universes, and SQL

To reduce the MQL learning curve, MQL looks similar to SQL. However, this resemblance is only on the surface, because MQL is a language on its own.

When writing a SQL command, the developer must know the join structure of the relations in the database. The join structure and semantic information about fields in the database, also called the data model, is very complex in the Maconomy system.

The main difference between MQL and SQL is the separation of the data model from the command. In SQL, the data model is specified in each command; in MQL, the data model is specified in a Universe. This separation enables the reuse of the data model in multiple commands, and it enables the MQL command to be developed without knowledge of the data model. For further information on Universes, see the *Deltek Maconomy Language Reference MUL.*

Unlike SQL, MQL is aware of types, and especially the Maconomy type system. The types of all expressions in an MQL command are validated before execution. This sort of validation can reveal many errors during development.

Unlike SQL, MQL is database-independent. The Maconomy Server knows which database is actually used, and performs a runtime translation of MQL into SQL in accordance with the requirements of the specific database.

Reading this Manual

The formal syntax of MQL is presented in BNF (Bachus Naur Form).

Where to Use MQL

MQL is designed to be the Maconomy language for direct database access. Currently, MQL is only used for reporting purposes, and is only available in M-Script and in MRL.

M-Script

MQL can be used from M-Script with the maconomy::mql* set of functions.

The version of the MQL command executed is dependent on the M-Script version, unless explicitly defined in the MQL command.

In the M-Script context, actual values for parameters can be given through an M-Script object, and the result of a command is returned as an M-Script object. Cursor definitions and result structuring features are ignored in this context.

For further information on MQL command integration in M-Script, see the *Deltek M-Script Maconomy API Reference*.

MRL (Maconomy Report Language)

MRL is used for specifying Universe Reports. A Universe Report is installed and executed on the Maconomy Server. MQL is used in MRL for query specification.

The version of the MQL query executed is dependent on the MRL version used.

In the MRL context, formal parameters are defined outside the MQL query, but are implicitly available inside the query. The MRL runtime framework handles actual values for the parameters. Cursor definitions and result structuring features are available in this context.

For further information on MQL command integration in MRL, see the *Deltek Maconomy Language Reference MRL* or the introduction to Universe reporting, *Getting Started with Universe Reports*.

Commands

This section describes the current version of MQL. A version history of MQL is supplied in the "Version History" section. Unlike SQL, an MQL command contains version identification, used for controlling syntax and/or semantic changes. The version identification can be implicit, given by the context of the MQL command. See Where to Use MQL for more information.

MQL commands are not case-sensitive; for instance, the command mselect equals MSelect.

```
mql ::=
 (<MQL 1.5> | MQL 1.5)
 mselect
```

MSelect Command

The `mselect` command retrieves rows, columns, and derived values from a Maconomy Universe. The syntax for the command is as follows.

```
mselect ::=
  MSELECT [DISTINCT] (fieldlist|cursor)
  [AGGREGATE [(ALL|SUM|MINMAX),] [aggregatedef (,aggregatedef)*] ]
  FROM module [INTERFACE id]
  [WHERE expressionshort]
  [ORDER BY qualifiedid [ASC|DESC] (, qualifiedid [ASC|DESC])* ]
  [USING PARAMETERS parmfield (, parmfield)* ]

fieldlist ::=
  field (, field)*

field ::=
  qualifiedfieldid |
  expressionshort AS id [TITLE string]

cursor ::=
[ fieldlist [, cursor] ] AS CURSOR id

aggregatedef ::=
  aggregateexp [AS id [TITLE string]] [ON idOrQualifiedid]

aggregateexp ::=
  SUM() |
  MIN() |
  MAX() |
  PCT(idOrFunctionfieldid, idOrFunctionfieldid) |
  MUL(idOrFunctionfieldid, idOrFunctionfieldid) |
  DIV(idOrFunctionfieldid, idOrFunctionfieldid) |
  constExpressionShort

parmfield ::=

  id (TYPE|:) typeid [TITLE string] [DEFAULT constExpressionShort]
```

Universe and Field Selection

The basic functionality of the `mselect` command is the selection of fields from a Universe. The Universe is selected in the from clause, and fields used elsewhere in the query are taken from this Universe.

Unlike SQL, Universes cannot be joined in the from clause. If a join of Universes is required, then a new Universe must be defined. Note that all Maconomy relations are also available as Universes.

Column Selection

The fields in the select clause define the order of the columns in the result. The fields also determine the root-selection of the used Universe. The root-selection is quite complex, but normally you do not need to be concerned with this issue. See the *Deltek Maconomy Language Reference MUL* for further information.

The following example shows the selection of two fields from the 'Employee' Universe, which is also a Maconomy relation:

```
mselect EmployeeNumber, Name1 from Employee
```

EmployeeNumber EmployeeNumber String	Name1 (Name) Name 1 (Title) String (Type)
1011	Hansa Mujaf
1012	Joe Daniels

Unlike SQL, there is no group-by clause in MQL where explicit grouping of rows can be defined. If a field in the select clause is defined using one of the row group functions (see the "Row Group Functions" section), implicit row grouping is done on the selected fields not defined using a row group function.

In the following example, the field 'EmployeeNumberCOUNT' is defined using a row group function, and therefore an implicit row grouping is done on the field 'Country'.

```
mselect Country,
        COUNT(EmployeeNumber) as EmployeeNumberCOUNT
from Employee
order by Country
```

Country Country CountryType	EmployeeNumberCOUNT (Name) EmployeeNumberCOUNT (Title) Integer (Type)
UK	2
USA	10

Universe Interface Option

Different interfaces can be defined in a Universe, allowing different views of the data model defined in the Universe. The interface used in the query is selected in the from clause. If no interface is specified, the default interface is used.

Distinct Option

The distinct option in the select clause specifies how to handle duplicate rows in the result. If the option is selected, duplicate rows are filtered out. By default, the distinct option is not selected, that is, all rows in the result are shown.

Field Definition

New fields can be defined in the select clause using the Maconomy functions defined in the "Maconomy Functions" section. Unlike SQL, all fields have an associated title. The title of a new field is defined by the title option. If no title is defined, the name of the field is used as the title.

In the following example, the Boolean field "Employed2003" is defined, indicating if the employee was employed in the year 2003. Also, the Boolean field "Employed" is defined, indicating if the employee is currently employed:

```
mselect EmployeeNumber,
        Name1,
        DateEmployed inrange [2003.01.01 .. 2003.12.31] as
Employed2003,
        DateEndEmployment = date'nil as Employed title "Employed ?"
from Employee
```

EmployeeNumber EmployeeNumber String	Name1 Name 1 String	Employed2003 Employed2003 Boolean	Employed Employed ? Boolean
1011	Hansa Mujaf	Yes	Yes
1012	Joe Daniels	No	No
...			

Structuring the Result

Unlike SQL, Mselect contains features for structuring the result of the query. The result structure might be ignored if the location where the command is executed does not support this feature. See "Where to Use SQL."

Aggregate Definition

With an aggregate definition, the sum, minimum, and maximum of values in a result column can be calculated. This kind of aggregate is called column aggregates. Further aggregates can be defined using the column aggregates as input for some simple calculations. This kind of aggregate is called aggregate aggregates.

Column Aggregates

Column aggregates can be defined explicitly for each column or defined implicitly for all integer, real, and amount columns for which a row group function is used. With the explicit definition, the aggregate can be assigned to a name and given a title, whereas with the implicit definition, names are constructed from the column name. The names of aggregates can be used when referring to the aggregates, for example in Universe Reports, where the aggregates can be used in the layout file.

Implicit column aggregates are performed on all integer, real, and amount columns defined using a row group function. Valid values for the implicit column aggregate definition are as follows.

ALL	Sum, minimum, and maximum of all columns defined using a row group function are calculated.
SUM	Sum of all columns defined using a row group function is calculated.
MINMAX	Minimum and maximum of all columns defined using a row group function are calculated.

If no name is specified in an explicit column aggregate definition, a name is assigned to the aggregate exactly as if it was implicitly defined.

In the following example, aggregate calculation is performed on the real column 'NumberOfWeekSUM', which is a field in the Universe "JobUniverse" defined using the row group function SUM. Aggregate calculation is also performed on the integer column "cTrans," defined in the query using the row group function COUNT. Note that aggregate calculation is not performed on the integer column "integerField" because it is not defined using a row group function. Notice also the names assigned to the aggregates.

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        1 as iField,
        count(Employee.EmployeeNumber) as cTrans
aggregate all
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	hours ??? Real	iField ??? Integer	cTrans ??? Integer
1011	Hansa Mujaf	70.0	1	2
1012	Joe Daniels	47.5	1	7
		117.5 *(hours$SUM)*		*9* *(cTrans$SUM)*
		47.5 *(hours$MIN)*		*2* *(cTrans$MIN)*
		70.0 *(hours$MAX)*		*7* *(cTrans$MAX)*

In the following example, an explicit column aggregate is defined to calculate the sum of the "iField" column, which is not included in the implicit column aggregations because the column is not defined using a row group function. Note the name "iFieldMin" assigned to the "min" explicit-aggregate definition of the "iField" column.

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        1 as iField,
        count(Employee.EmployeeNumber) as cTrans
aggregate all,
          sum() on iField,
          min() as iFieldMin on iField
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	hours ??? Real	iField ??? Integer	cTrans ??? Integer
1011	Hansa Mujaf	70.0	1	2
1012	Joe Daniels	47.5	1	7
		117.5 *(hours$SUM)*	*2* *(iField$SUM)*	*9* *(cTrans$SUM)*
		47.5 *(hours$MIN)*	*1* *(iFieldMin)*	*2* *(cTrans$MIN)*
		70.0 *(hours$MAX)*		*7* *(cTrans$MAX)*

Aggregate Aggregates

Aggregate aggregates can be defined using column aggregates and constants as input for some simple calculations. This can, for example, be used for calculation of overall averages.

The simple calculations available are division using “div,” multiplication using “mul,” and %-calculation using “pct.” An aggregate aggregate cannot be associated with a column; that is, the “on” option is not valid.

In the following example, implicit column aggregates are defined as in the previous examples. Additionally, the aggregate aggregate “aaOverallAvg” is defined using the column aggregates “hours$SUM” and “cTrans$SUM.”

```
mselect Employee.EmployeeNumber,
        Employee.Name,
        NumberOfWeekSUM as hours,
        count(Employee.EmployeeNumber) as cTrans
aggregate all,
          div(hours$SUM, cTrans$SUM) as aaOverallAvg
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]
order by Employee.EmployeeNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	hours ??? Real	cTrans ??? Integer
1011	Hansa Mujaf	70.0	2
1012	Joe Daniels	47.5	7
		117.5 (hours$SUM)	*9 (cTrans$SUM)*
		47.5 (hours$MIN)	*2 (cTrans$MIN)*
		70.0 (hours$MAX)	*7 (cTrans$MAX)*
13.056 (aaOverallAvg)			

Cursor Definition

A cursor is a named group of result columns. Naming a group of columns is needed when references are made to multiple queries. This is the case in MRL Reports where multiple queries can be defined and referred to in the layout of the report.

A cursor can contain a subcursor. A subcursor specifies that for each different value of the result row outside the subcursor, a list of rows with values inside the subcursor is to be generated. A subcursor is needed in connection with the aggregate option, when a subtotal is needed for a group of columns.

The cursor definitions are ignored if the location where the command is executed does not support this feature. See Where to Use MQL.

In the following example, a subcursor named “JobCursor” is defined, grouping the job related columns. The effect is that for each employee row, a list of rows (one row for each job), and a subtotal for each employee is generated.

```
mselect [ Employee.EmployeeNumber,
          Employee.Name,
          [Job.JobNumber, NumberOfWeekSUM] as cursor JobCursor
        ] as cursor EmployeeCursor
        aggregate sum
from JobUniverse
where TimeSheet.PeriodStart inrange [2003.01.01 .. 2003.12.31]

order by Employee.EmployeeNumber, Job.JobNumber
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	Job.JobNumber ??? String	NumberOfWeekSUM ??? Real
1011	Hansa Mujaf	10	40.0
		20	30.0
			70.0 *(NumberOfWeekSUM$SUM)*
1012	Joe Daniels	10	37.5
		30	10.0
			47.5 *(NumberOfWeekSUM$SUM)*
			117.5 *(NumberOfWeekSUM$SUM)*

Restriction

Restrictions on the result rows can be specified in the where clause of the `mselect` command. A large number of Maconomy functions are available for restriction specifications. See Maconomy Functions.

In the following example, all employees who are employed later than the beginning of this month are listed:

```
mselect EmployeeNumber, Name1
from Employee
where DateEmployed >= getfirstofmonth( getdate() )
order by EmployeeNumber
```

Unlike SQL, the `mselect` command does not have a having clause. In SQL, the having clause is used for restrictions that contain a row group function. In Mselect, such restrictions are also specified in the where clause.

In the following example, all countries with fewer than 5 employees are shown.

```
mselect Country
from Employee
where COUNT(EmployeeNumber) < 5
order by Country
```

Unlike in SQL, in MQL subqueries cannot be used in a restriction. A subquery is a query used in the where clause. Subqueries will be introduced in a future version of MQL. Note that subqueries can be used in a restriction defined in the Universe.

Ordering

The order of the result rows can be specified in the order-by clause. Unlike in SQL, in MQL only fields selected in the select clause can be used for ordering. It is always a good idea to specify the order-by clause, because if it is not specified, the order is undefined.

The result rows may be sorted in ascending or descending order. Valid values for the order option are as follows.

ASC	Ascending sorting, this is the default value.
DESC	Descending sorting.

In the following example, the result rows are ordered ascending on the field "Employee.EmployeeNumber" and then descending on the field "Job.JobNumber."

```
mselect Employee.EmployeeNumber,
        Employee.Name,

        Job.JobNumber,

               NumberOfWeekSUM
        from JobUniverse
        order by Employee.EmployeeNumber asc,

                 Job.JobNumber desc
```

Employee.EmployeeNumber EmployeeNumber String	Employee.Name Name 1 String	Job.JobNumber ??? String	NumberOfWeekSUM ??? Real
1011	Hansa Mujaf	20	30.0
1011	Hansa Mujaf	10	40.0
1012	Joe Daniels	30	10.0
1012	Joe Daniels	10	37.5

Parameter Definition

The `mselect` command can be parameterized using the parameter section of the command. Parameterization of an `mselect` command can be used in parts of the Maconomy system that support this feature. See Where to Use MQL.

The advantage of formal parameter specification is that type-safe values can be assigned at execution time, and that a command can be validated without being executed (static validation).

A parameter defined in the parameter section can be used as any other field reference. The type of the parameter must be specified, whereas the title and default value are optional. If a default value is specified, this value is used if no actual value is given at execution time.

In the following example, all employees employed within a given date range are listed. If "parmDateBegin" equals 2003.01.01 at execution time, all employees employed after the first of January 2003 are listed. If no actual values for the parameters are given at execution time, employees employed after today are listed.

Example:

```
mselect EmployeeNumber, Name1
from Employee
where DateEmployed inrange [parmDateBegin .. parmDateEnd]
order by EmployeeNumber
using parameters parmDateBegin type date default date'today,
                 parmDateEnd   type date
```

Common Syntax

This section describes common elements of the MQL language.

Expressions

Expressions are used in restrictions and in field definitions. Every expression has an associated type, which is inferred from the explicit type given by field reference, by type given by constant, or by type given by function type schema. A type error occurs if the inferred type does not match the expected type of an expression. If for instance an expression is used as a restriction, the type of the expression must be the type Boolean. Note that implicit type conversions are not performed. Functions are available for doing explicit type conversions.

```
expressionShort ::=
  subExpressionShort      |
  fieldExpressionShort    |
  constExpressionShort    |
  functionExpressionShort

subExpressionShort :: =
  ( expressionShort )

fieldExpressionShort :: =
  qualifiedfieldid | .id | functionfieldid

functionExpressionShort :: =
  functionExpressionShortInfix   |
  functionExpressionShortPrefix

functionExpressionShortInfix :: =
  expressionShort  *  expressionShort   |
  expressionShort  /  expressionShort   |
  expressionShort DIV expressionShort   |
  expressionShort  +  expressionShort   |
  expressionShort  -  expressionShort   |
  expressionShort  <  expressionShort   |
  expressionShort  <= expressionShort   |
  expressionShort  >  expressionShort   |
  expressionShort  >= expressionShort   |
  expressionShort  =  expressionShort   |
  expressionShort  != expressionShort   |
  expressionShort  <> expressionShort   |
  expressionShort AND expressionShort   |
  expressionShort OR  expressionShort   |
  expressionShort INRANGE [ expressionShort .. expressionShort ] |
  expressionShort IN expressionShort           Special function

FunctionExpressionShortPrefix ::=
  - expressionShort                          |
  ! expressionShort                          |
  NOT expressionShort                        |
  id ( expressionShort {,expressionShort}* ) |
```

```
  id ()
```

Identifiers

```
id ::=
  char (char | num)*

module ::=
  id(::id)*                                  Job::EmployeeReport

qualifiedid ::=
  id(.id)*                                   Jobheader.JobNumber

functionfieldid ::=
  qualifiedid $ id                           Jobheader.JobNumber$SUM

idOrQualifiedid ::=
  id | qualifiedid

idOrFunctionfieldid ::=
  id | functionfieldid
```

Types

```
typeid ::=
  STRING | INTEGER | REAL | AMOUNT | BOOLEAN |
  id                                                        CountryType
```

Apart from the basis types, the Maconomy pop-up types are also available.

Literals

```
constExpressionShort ::=
  kernelString | templateString | rawString    |
  integer      | real           | amountValue |
  booleanValue | dateValue      | timeValue   |
  typedValue

templateString ::=
  "(char)*"                                                 "Template
string"

kernelString ::=
  @(char)*@                                                   @Kernel
string@

rawString ::=
  '(char)*'                                                              'Raw
string'

integer ::=
  [-]num+

real ::=
  [-]num+ . num+
```

```
0.5434
                                                                        _
1010.999

amountValue ::=
  [-]num+ . num num A
100.50A
                                                                        _
2000.50A
booleanValue ::=
  true | false

dateValue ::=
  num num num num . num num . num num
2001.12.31

timeValue ::=
  num num : num num : num num [AM|PM]
23:50:01

10:03:00AM

typedValue ::=
  typeid '_ null |
  typeid '_ id

num ::=
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

char ::=
  A | .. | Z | Æ | Ø | Å | a | .. | å | ø | å
```

A template string is localized dynamically before execution of the command. This means that the string is translated using the dictionary currently selected by the user. Kernel and raw strings are **not** dynamically localized.

Maconomy Functions

A wide range of functions is available for use in expressions. For each function, a type scheme is defined. If the types of the arguments to a function do not match the type scheme, a type error is given.

Note that even if the database is known, database functions are not available.

Predefined Functions

A list of functions exists for every released TPU.

Special Functions

Function	Type schema(s) Description
in (infix operator)	(α , string) → Boolean Returns true if the first argument is in the second argument. The second argument must be a constant, which is either a literal or a parameter reference. The format of the second argument is a user interface range, for example, "1..10", "1<" for integers, and "T*" for strings. See elsewhere for format specification.

Row Group Functions

The row group functions create one row from a number of rows using the corresponding database row group functions.

Function	Type schema(s) Description
sum	integer → integer real → real amount → amount Computes the total sum of all values in a group of rows.
min	integer → integer real → real amount → amount string → string Computes the minimum of all values in a group of rows.

Function	Type schema(s) Description
max	integer → integer real → real amount → amount string → string Computes the maximum of all values in a group of rows.
avg	integer → integer real → real amount → amount Computes the average of all values in a group of rows.
count	α → integer Count the number of rows in a group. Note that unlike sql, null values appearing from outer joins are included in the count.

Version History

MQL 1.5

The aggregation feature has been extended with explicit column aggregation and aggregate aggregation. This extension can be used for overall average calculation. See Aggregate Definition.

MQL 1.4

Internal release, no changes relevant for documentation.

MQL 1.3

Support for raw strings added.

Support for template strings added—that is, dynamic translation of strings.

The special in-operator added, supporting in-range functionality on user interface values.

MQL 1.2

Initial version.

Deltek is the leading global provider of enterprise software and information solutions for government contractors, professional services firms and other project- and people-based businesses. For decades, we have delivered actionable insight that empowers our customers to unlock their business potential. 20,000 organizations and millions of users in over 80 countries around the world rely on Deltek to research and identify opportunities, win new business, recruit and develop talent, optimize resources, streamline operations and deliver more profitable projects. Deltek – Know more. Do more.®

deltek.com