Contents

- Power Query M formula language
  - Quick tour of the Power Query M formula language
  - Power Query M language specification
    - Introduction
    - Lexical Structure
    - Basic Concepts
    - Values
    - Types
    - Operators
    - Let
    - Conditionals
    - Functions
    - Error Handling
    - Sections
    - Consolidated Grammar
  - Power Query M type system
  - Expressions, values, and let expression
  - Comments
  - Evaluation model
  - Operators
  - Type conversion
  - Metadata
  - Errors
  - Power Query M functions
    - Power Query M functions overview
    - Understanding Power Query M functions
    - Accessing data functions
      - Accessing data functions overview
      - AccessControlEntry.ConditionToIdentities

AccessControlKind.Allow

AccessControlKind.Deny

Access.Database

ActiveDirectory.Domains

AdobeAnalytics.Cubes

AdoDotNet.DataSource

AdoDotNet.Query

AnalysisServices.Database

AnalysisServices.Databases

AzureStorage.BlobContents

AzureStorage.Blobs

AzureStorage.DataLake

AzureStorage.DataLakeContents

AzureStorage.Tables

Cdm.Contents

Csv.Document

CsvStyle.QuoteAfterDelimiter

CsvStyle.QuoteAlways

Cube.AddAndExpandDimensionColumn

Cube.AddMeasureColumn

Cube.ApplyParameter

Cube.AttributeMemberId

Cube.AttributeMemberProperty

Cube.CollapseAndRemoveColumns

Cube.Dimensions

Cube.DisplayFolders

Cube.MeasureProperties

Cube.MeasureProperty

Cube.Measures

Cube.Parameters

Cube.Properties

Cube.PropertyKey

Cube.ReplaceDimensions

Cube.Transform

DB2.Database

Essbase.Cubes

Excel.CurrentWorkbook

Excel.Workbook

Exchange.Contents

File.Contents

Folder.Contents

Folder.Files

GoogleAnalytics.Accounts

Hdfs.Contents

Hdfs.Files

HdInsight.Containers

HdInsight.Contents

HdInsight.Files

Html.Table

Identity.From

Identity.IsMemberOf

IdentityProvider.Default

Informix.Database

Json.Document

Json.FromValue

MySQL.Database

OData.Feed

ODataOmitValues.Nulls

Odbc.DataSource

Odbc.InferOptions

Odbc.Query

OleDb.DataSource

OleDb.Query

Oracle.Database

Pdf.Tables

PostgreSQL.Database

RData.FromBinary

Salesforce.Data

Salesforce.Reports

SapBusinessWarehouse.Cubes

SapBusinessWarehouseExecutionMode.DataStream

SapBusinessWarehouseExecutionMode.BasXml

SapBusinessWarehouseExecutionMode.BasXmlGzip

SapHana.Database

SapHanaDistribution.All

SapHanaDistribution.Connection

SapHanaDistribution.Off

SapHanaDistribution.Statement

SapHanaRangeOperator.Equals

SapHanaRangeOperator.GreaterThan

SapHanaRangeOperator.GreaterThanOrEquals

SapHanaRangeOperator.LessThan

SapHanaRangeOperator.LessThanOrEquals

SapHanaRangeOperator.NotEquals

SharePoint.Contents

SharePoint.Files

SharePoint.Tables

Soda.Feed

Sql.Database

Sql.Databases

Sybase.Database

Teradata.Database

WebAction.Request

Web.BrowserContents

Web.Contents

Web.Page

WebMethod.Delete
WebMethod.Get
WebMethod.Head
WebMethod.Patch
WebMethod.Post
WebMethod.Put
Xml.Document
Xml.Tables
Binary functions
Binary functions overview
Binary.Buffer
Binary.Combine
Binary.Compress
Binary.Decompress
Binary.From
Binary.FromList
Binary.FromText
Binary.InferContentType
Binary.Length
Binary.Range
Binary.ToList
Binary.ToText
BinaryEncoding.Base64
BinaryEncoding.Hex
BinaryFormat.7BitEncodedSignedInteger
BinaryFormat.7BitEncodedUnsignedInteger
BinaryFormat.Binary
BinaryFormat.Byte
BinaryFormat.ByteOrder
BinaryFormat.Choice
BinaryFormat.Decimal
BinaryFormat.Double

BinaryFormat.Group
BinaryFormat.Length
BinaryFormat.List
BinaryFormat.Null
BinaryFormat.Record
BinaryFormat.SignedInteger16
BinaryFormat.SignedInteger32
BinaryFormat.SignedInteger64
BinaryFormat.Single
BinaryFormat.Text
BinaryFormat.Transform
BinaryFormat.UnsignedInteger16
BinaryFormat.UnsignedInteger32
BinaryFormat.UnsignedInteger64
BinaryOccurrence.Optional
BinaryOccurrence.Repeating
BinaryOccurrence.Required
ByteOrder.BigEndian
ByteOrder.LittleEndian
Compression.Brotli
Compression.Deflate
Compression.GZip
Compression.LZ4
Compression.None
Compression.Snappy
Compression.Zstandard
Occurrence.Optional
Occurrence.Repeating
Occurrence.Required
#binary
Combiner functions
Combiner functions overview

- Combiner.CombineTextByDelimiter
- Combiner.CombineTextByEachDelimiter
- Combiner.CombineTextByLengths
- Combiner.CombineTextByPositions
- Combiner.CombineTextByRanges

Comparer functions

- Comparer functions overview
- Comparer.Equals
- Comparer.FromCulture
- Comparer.Ordinal
- Comparer.OrdinalIgnoreCase
- Culture.Current

Date functions

- Date functions overview
- Date.AddDays
- Date.AddMonths
- Date.AddQuarters
- Date.AddWeeks
- Date.AddYears
- Date.Day
- Date.DayOfWeek
- Date.DayOfWeekName
- Date.DayOfYear
- Date.DaysInMonth
- Date.EndOfDay
- Date.EndOfMonth
- Date.EndOfQuarter
- Date.EndOfWeek
- Date.EndOfYear
- Date.From
- Date.FromText
- Date.IsInCurrentDay

Date.IsInCurrentMonth

Date.IsInCurrentQuarter

Date.IsInCurrentWeek

Date.IsInCurrentYear

Date.IsInNextDay

Date.IsInNextMonth

Date.IsInNextNDays

Date.IsInNextNMonths

Date.IsInNextNQuarters

Date.IsInNextNWeeks

Date.IsInNextNYears

Date.IsInNextQuarter

Date.IsInNextWeek

Date.IsInNextYear

Date.IsInPreviousDay

Date.IsInPreviousMonth

Date.IsInPreviousNDays

Date.IsInPreviousNMonths

Date.IsInPreviousNQuarters

Date.IsInPreviousNWeeks

Date.IsInPreviousNYears

Date.IsInPreviousQuarter

Date.IsInPreviousWeek

Date.IsInPreviousYear

Date.IsInYearToDate

Date.IsLeapYear

Date.Month

Date.MonthName

Date.QuarterOfYear

Date.StartOfDay

Date.StartOfMonth

Date.StartOfQuarter

Date.StartOfWeek
Date.StartOfYear
Date.ToRecord
Date.ToText
Date.WeekOfMonth
Date.WeekOfYear
Date.Year
Day.Friday
Day.Monday
Day.Saturday
Day.Sunday
Day.Thursday
Day.Tuesday
Day.Wednesday
#date
DateTime functions
DateTime functions overview
DateTime.AddZone
DateTime.Date
DateTime.FixedLocalNow
DateTime.From
DateTime.FromFileTime
DateTime.FromText
DateTime.IsInCurrentHour
DateTime.IsInCurrentMinute
DateTime.IsInCurrentSecond
DateTime.IsInNextHour
DateTime.IsInNextMinute
DateTime.IsInNextNHours
DateTime.IsInNextNMinutes
DateTime.IsInNextNSeconds
DateTime.IsInNextSecond

- DateTime.IsInPreviousHour
- DateTime.IsInPreviousMinute
- DateTime.IsInPreviousNHours
- DateTime.IsInPreviousNMinutes
- DateTime.IsInPreviousNSeconds
- DateTime.IsInPreviousSecond
- DateTime.LocalNow
- DateTime.Time
- DateTime.ToRecord
- DateTime.ToText
- #datetime

DateTimeZone functions

- DateTimeZone functions overview
- DateTimeZone.FixedLocalNow
- DateTimeZone.FixedUtcNow
- DateTimeZone.From
- DateTimeZone.FromFileTime
- DateTimeZone.FromText
- DateTimeZone.LocalNow
- DateTimeZone.RemoveZone
- DateTimeZone.SwitchZone
- DateTimeZone.ToLocal
- DateTimeZone.ToRecord
- DateTimeZone.ToText
- DateTimeZone.ToUtc
- DateTimeZone.UtcNow
- DateTimeZone.ZoneHours
- DateTimeZone.ZoneMinutes
- #datetimezone

Duration functions

- Duration functions overview
- Duration.Days

- Duration.From
- Duration.FromText
- Duration.Hours
- Duration.Minutes
- Duration.Seconds
- Duration.ToRecord
- Duration.TotalDays
- Duration.TotalHours
- Duration.TotalMinutes
- Duration.TotalSeconds
- Duration.ToText
- #duration

Error handling

- Error handling overview
- Diagnostics.ActivityId
- Diagnostics.Trace
- Error.Record
- TraceLevel.Critical
- TraceLevel.Error
- TraceLevel.Information
- TraceLevel.Verbose
- TraceLevel.Warning

Expression functions

- Expression functions overview
- Expression.Constant
- Expression.Evaluate
- Expression.Identifier

Function values

- Function values overview
- Function.From
- Function.Invoke
- Function.InvokeAfter

- Function.IsDataSource
- Function.ScalarVector

Lines functions

- Lines functions overview
- Lines.FromBinary
- Lines.FromText
- Lines.ToBinary
- Lines.ToText

List functions

- List functions overview
- List.Accumulate
- List.AllTrue
- List.Alternate
- List.AnyTrue
- List.Average
- List.Buffer
- List.Combine
- List.ConformToPageReader
- List.Contains
- List.ContainsAll
- List.ContainsAny
- List.Count
- List.Covariance
- List.Dates
- List.DateTimes
- List.DateTimeZones
- List.Difference
- List.Distinct
- List.Durations
- List.FindText
- List.First
- List.FirstN

List.Generate

List.InsertRange

List.Intersect

List.IsDistinct

List.IsEmpty

List.Last

List.LastN

List.MatchesAll

List.MatchesAny

List.Max

List.MaxN

List.Median

List.Min

List.MinN

List.Mode

List.Modes

List.NonNullCount

List.Numbers

List.Percentile

List.PositionOf

List.PositionOfAny

List.Positions

List.Product

List.Random

List.Range

List.RemoveFirstN

List.RemoveItems

List.RemoveLastN

List.RemoveMatchingItems

List.RemoveNulls

List.RemoveRange

List.Repeat

- List.ReplaceMatchingItems
- List.ReplaceRange
- List.ReplaceValue
- List.Reverse
- List.Select
- List.Single
- List.SingleOrDefault
- List.Skip
- List.Sort
- List.Split
- List.StandardDeviation
- List.Sum
- List.Times
- List.Transform
- List.TransformMany
- List.Union
- List.Zip
- PercentileMode.ExcelExc
- PercentileMode.ExcelInc
- PercentileMode.SqlCont
- PercentileMode.SqlDisc

Logical functions

- Logical functions overview
- Logical.From
- Logical.FromText
- Logical.ToText

Number functions

- Number functions overview
- Byte.From
- Currency.From
- Decimal.From
- Double.From

Int8.From

Int16.From

Int32.From

Int64.From

Number.Abs

Number.Acos

Number.Asin

Number.Atan

Number.Atan2

Number.BitwiseAnd

Number.BitwiseNot

Number.BitwiseOr

Number.BitwiseShiftLeft

Number.BitwiseShiftRight

Number.BitwiseXor

Number.Combinations

Number.Cos

Number.Cosh

Number.E

Number.Epsilon

Number.Exp

Number.Factorial

Number.From

Number.FromText

Number.IntegerDivide

Number.IsEven

Number.IsNaN

Number.IsOdd

Number.Ln

Number.Log

Number.Log10

Number.Mod

- Number.NaN
- Number.NegativeInfinity
- Number.Permutations
- Number.PI
- Number.PositiveInfinity
- Number.Power
- Number.Random
- Number.RandomBetween
- Number.Round
- Number.RoundAwayFromZero
- Number.RoundDown
- Number.RoundTowardZero
- Number.RoundUp
- Number.Sign
- Number.Sin
- Number.Sinh
- Number.Sqrt
- Number.Tan
- Number.Tanh
- Number.ToText
- Percentage.From
- RoundingMode.AwayFromZero
- RoundingMode.Down
- RoundingMode.ToEven
- RoundingMode.TowardZero
- RoundingMode.Up
- Single.From

Record functions

- Record functions overview
- Geography.FromWellKnownText
- Geography.ToWellKnownText
- GeographyPoint.From

- Geometry.FromWellKnownText
- Geometry.ToWellKnownText
- GeometryPoint.From
- MissingField.Error
- MissingField.Ignore
- MissingField.UseNull
- Record.AddField
- Record.Combine
- Record.Field
- Record.FieldCount
- Record.FieldNames
- Record.FieldOrDefault
- Record.FieldValues
- Record.FromList
- Record.FromTable
- Record.HasFields
- Record.RemoveFields
- Record.RenameFields
- Record.ReorderFields
- Record.SelectFields
- Record.ToList
- Record.ToTable
- Record.TransformFields

Replacer functions

- Replacer functions overview
- Replacer.ReplaceText
- Replacer.ReplaceValue

Splitter functions

- Splitter functions overview
- QuoteStyle.Csv
- QuoteStyle.None
- Splitter.SplitByNothing

- Splitter.SplitTextByAnyDelimiter
- Splitter.SplitTextByCharacterTransition
- Splitter.SplitTextByDelimiter
- Splitter.SplitTextByEachDelimiter
- Splitter.SplitTextByLengths
- Splitter.SplitTextByPositions
- Splitter.SplitTextByRanges
- Splitter.SplitTextByRepeatedLengths
- Splitter.SplitTextByWhitespace

Table functions

- Table functions overview
- ExtraValues.Error
- ExtraValues.Ignore
- ExtraValues.List
- GroupKind.Global
- GroupKind.Local
- ItemExpression.From
- ItemExpression.Item
- JoinAlgorithm.Dynamic
- JoinAlgorithm.LeftHash
- JoinAlgorithm.LeftIndex
- JoinAlgorithm.PairwiseHash
- JoinAlgorithm.RightHash
- JoinAlgorithm.RightIndex
- JoinAlgorithm.SortMerge
- JoinKind.FullOuter
- JoinKind.Inner
- JoinKind.LeftAnti
- JoinKind.LeftOuter
- JoinKind.RightAnti
- JoinKind.RightOuter
- JoinSide.Left

JoinSide.Right

Occurrence.All

Occurrence.First

Occurrence.Last

Order.Ascending

Order.Descending

RowExpression.Column

RowExpression.From

RowExpression.Row

Table.AddColumn

Table.AddFuzzyClusterColumn

Table.AddIndexColumn

Table.AddJoinColumn

Table.AddKey

Table.AggregateTableColumn

Table.AlternateRows

Table.ApproximateRowCount

Table.Buffer

Table.Column

Table.ColumnCount

Table.ColumnNames

Table.ColumnsOfType

Table.Combine

Table.CombineColumns

Table.CombineColumnsToRecord

Table.ConformToPageReader

Table.Contains

Table.ContainsAll

Table.ContainsAny

Table.DemoteHeaders

Table.Distinct

Table.DuplicateColumn

Table.ExpandListColumn

Table.ExpandRecordColumn

Table.ExpandTableColumn

Table.FillDown

Table.FillUp

Table.FilterWithDataTable

Table.FindText

Table.First

Table.FirstN

Table.FirstValue

Table.FromColumns

Table.FromList

Table.FromPartitions

Table.FromRecords

Table.FromRows

Table.FromValue

Table.FuzzyGroup

Table.FuzzyJoin

Table.FuzzyNestedJoin

Table.Group

Table.HasColumns

Table.InsertRows

Table.IsDistinct

Table.IsEmpty

Table.Join

Table.Keys

Table.Last

Table.LastN

Table.MatchesAllRows

Table.MatchesAnyRows

Table.Max

Table.MaxN

Table.Min

Table.MinN

Table.NestedJoin

Table.Partition

Table.PartitionValues

Table.Pivot

Table.PositionOf

Table.PositionOfAny

Table.PrefixColumns

Table.Profile

Table.PromoteHeaders

Table.Range

Table.RemoveColumns

Table.RemoveFirstN

Table.RemoveLastN

Table.RemoveMatchingRows

Table.RemoveRows

Table.RemoveRowsWithErrors

Table.RenameColumns

Table.ReorderColumns

Table.Repeat

Table.ReplaceErrorValues

Table.ReplaceKeys

Table.ReplaceMatchingRows

Table.ReplaceRelationshipIdentity

Table.ReplaceRows

Table.ReplaceValue

Table.ReverseRows

Table.RowCount

Table.Schema

Table.SelectColumns

Table.SelectRows

- Table.SelectRowsWithErrors
- Table.SingleRow
- Table.Skip
- Table.Sort
- Table.Split
- Table.SplitAt
- Table.SplitColumn
- Table.ToColumns
- Table.ToList
- Table.ToRecords
- Table.ToRows
- Table.TransformColumnNames
- Table.TransformColumns
- Table.TransformColumnTypes
- Table.TransformRows
- Table.Transpose
- Table.Unpivot
- Table.UnpivotOtherColumns
- Table.View
- Table.ViewFunction
- Tables.GetRelationships
- #table

Text functions

- Text functions overview
- Character.FromNumber
- Character.ToNumber
- Guid.From
- Json.FromValue
- RelativePosition.FromEnd
- RelativePosition.FromStart
- Text.AfterDelimiter
- Text.At

Text.BeforeDelimiter

Text.BetweenDelimiters

Text.Clean

Text.Combine

Text.Contains

Text.End

Text.EndsWith

Text.Format

Text.From

Text.FromBinary

Text.InferNumberType

Text.Insert

Text.Length

Text.Lower

Text.Middle

Text.NewGuid

Text.PadEnd

Text.PadStart

Text.PositionOf

Text.PositionOfAny

Text.Proper

Text.Range

Text.Remove

Text.RemoveRange

Text.Repeat

Text.Replace

Text.ReplaceRange

Text.Reverse

Text.Select

Text.Split

Text.SplitAny

Text.Start

- Text.StartsWith
- Text.ToBinary
- Text.ToList
- Text.Trim
- Text.TrimEnd
- Text.TrimStart
- Text.Upper
- TextEncoding.Ascii
- TextEncoding.BigEndianUnicode
- TextEncoding.Unicode
- TextEncoding.Utf8
- TextEncoding.Utf16
- TextEncoding.Windows

Time functions

- Time functions overview
- Time.EndOfHour
- Time.From
- Time.FromText
- Time.Hour
- Time.Minute
- Time.Second
- Time.StartOfHour
- Time.ToRecord
- Time.ToText
- #time

Type functions

- Type functions overview
- Type.AddTableKey
- Type.ClosedRecord
- Type.Facets
- Type.ForFunction
- Type.ForRecord

- Type.FunctionParameters
- Type.FunctionRequiredParameters
- Type.FunctionReturn
- Type.Is
- Type.IsNullable
- Type.IsOpenRecord
- Type.ListItem
- Type.NonNullable
- Type.OpenRecord
- Type.RecordFields
- Type.ReplaceFacets
- Type.ReplaceTableKeys
- Type.TableColumn
- Type.TableKeys
- Type.TableRow
- Type.TableSchema
- Type.Union

Uri functions

- Uri functions overview
- Uri.BuildQueryString
- Uri.Combine
- Uri.EscapeDataString
- Uri.Parts

Value functions

- Value functions overview
- DirectQueryCapabilities.From
- Embedded.Value
- Graph.Nodes
- Precision.Decimal
- Precision.Double
- SqlExpression.SchemaFrom
- SqlExpression.ToExpression

Value.Add

Value.Alternate

Value.As

Value.Compare

Value.Divide

Value.Equals

Value.Expression

Value.Firewall

Value.FromText

Value.Is

Value.Lineage

Value.Metadata

Value.Multiply

Value.NativeQuery

Value.NullableEquals

Value.Optimize

Value.RemoveMetadata

Value.ReplaceMetadata

Value.ReplaceType

Value.Subtract

Value.Traits

Value.Type

Variable.Value

Quick tour of the Power Query M formula language

4/14/2021 • 2 minutes to read

This quick tour describes creating Power Query M formula language queries.

> **NOTE**
> M is a case-sensitive language.

Create a query with Query Editor

To create an advanced query, you use the **Query Editor**. A mashup query is composed of variables, expressions, and values encapsulated by a **let** expression. A variable can contain spaces by using the # identifier with the name in quotes as in #"Variable name".

A **let** expression follows this structure:

```
let
   Variablename = expression,
   #"Variable name" = expression2
in
   Variablename
```

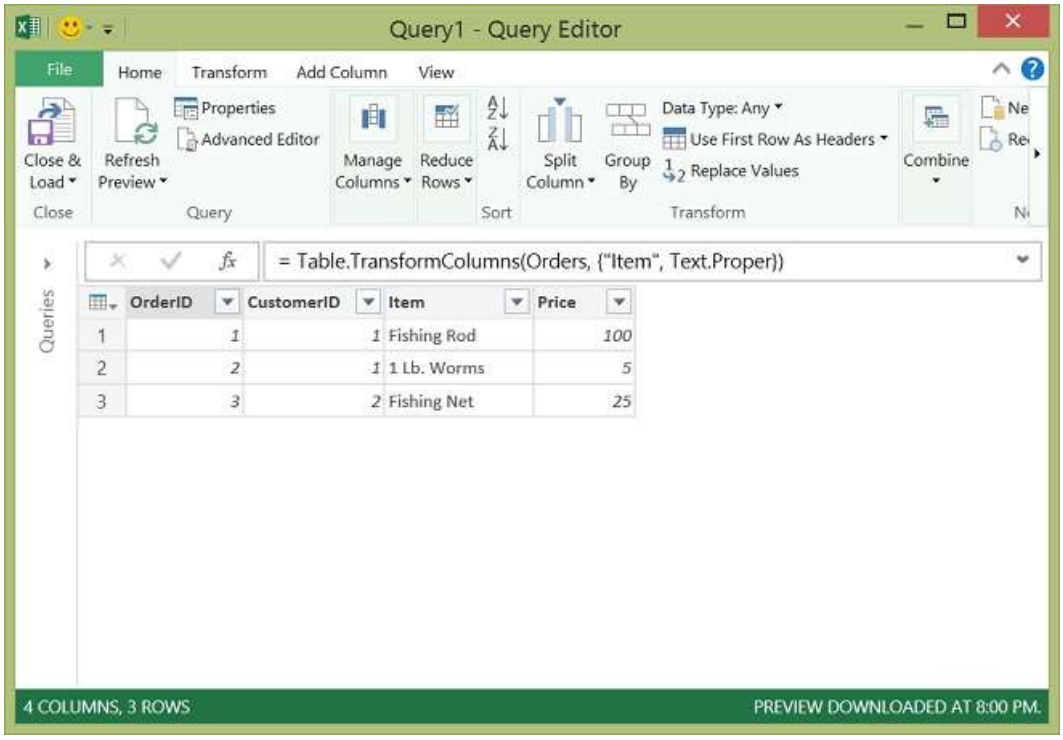
To create an M query in the **Query Editor**, you follow this basic process:

- Create a series of query formula steps that start with the **let** statement. Each step is defined by a step variable name. An M **variable** can include spaces by using the # character as #"Step Name". A formula step can be a custom formula. Please note that the Power Query Formula Language is case sensitive.
- Each query formula step builds upon a previous step by referring to a step by its variable name.
- Output a query formula step using the **in** statement. Generally, the last query step is used as the in final data set result.

To learn more about expressions and values, see Expressions, values, and let expression.

Simple Power Query M formula steps

Let's assume you created the following transform in the **Query Editor** to convert product names to proper case.



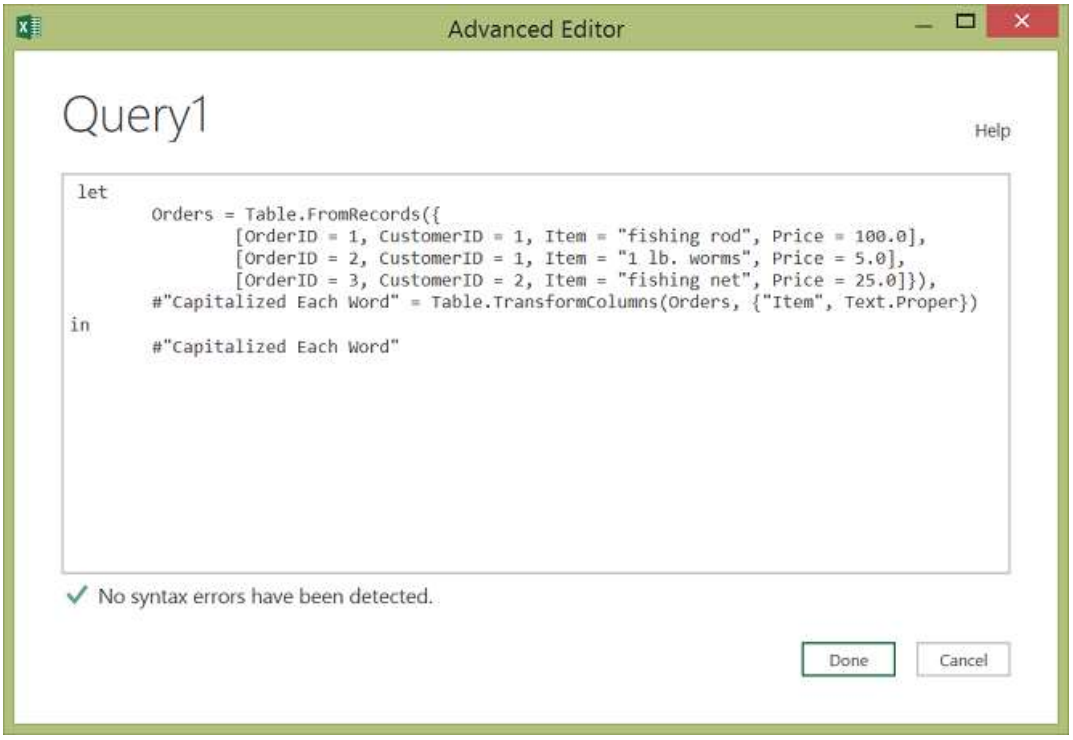
You have a table that looks like this:

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	fishing rod	100
2	1	1 lb. worms	5
3	2	fishing net	25

And, you want to capitalize each word in the Item column to produce the following table:

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	Fishing Rod	100
2	1	1 Lb. Worms	5
3	2	Fishing Net	25

The M formula steps to project the original table into the results table looks like this:



Here's the code you can paste into **Query Editor**:

```
let Orders = Table.FromRecords({
    [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
    [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
    #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item", Text.Proper})
in
    #"Capitalized Each Word"
```

Let's review each formula step.

1. **Orders** – Create a [Table](#_Table_value) with data for Orders.
2. **#"Capitalized Each Word"** – To capitalize each word, you use Table.TransformColumns().
3. **in #"Capitalized Each Word"** – Output the table with each word capitalized.

See also

Expressions, values, and let expression
Operators
Type conversion

Power Query M language specification

3/15/2021 • 2 minutes to read

The specification describes the values, expressions, environments and variables, identifiers, and the evaluation model that form the Power Query M language's basic concepts.

The specification is contained in the following topics.

- Introduction
- Lexical Structure
- Basic Concepts
- Values
- Types
- Operators
- Let
- Conditionals
- Functions
- Error Handling
- Sections
- Consolidated Grammar

Introduction

12/11/2020 • 11 minutes to read

Overview

Microsoft Power Query provides a powerful "get data" experience that encompasses many features. A core capability of Power Query is to filter and combine, that is, to "mash-up" data from one or more of a rich collection of supported data sources. Any such data mashup is expressed using the Power Query Formula Language (informally known as "M"). Power Query embeds M documents in Excel and Power BI workbooks to enable repeatable mashup of data.

This document provides the specification for M. After a brief introduction that aims at building some first intuition and familiarity with the language, the document covers the language precisely in several progressive steps:

1. The *lexical structure* defines the set of texts that are lexically valid.
2. Values, expressions, environments and variables, identifiers, and the evaluation model form the language's *basic concepts*.
3. The detailed specification of *values*, both primitive and structured, defines the target domain of the language.
4. Values have *types*, themselves a special kind of value, that both characterize the fundamental kinds of values and carry additional metadata that is specific to the shapes of structured values.
5. The set of *operators* in M defines what kinds of expressions can be formed.
6. *Functions*, another kind of special values, provide the foundation for a rich standard library for M and allow for the addition of new abstractions.
7. *Errors* can occur when applying operators or functions during expression evaluation. While errors are not values, there are ways to *handle errors* that map errors back to values.
8. *Let expressions* allow for the introduction of auxiliary definitions used to build up complex expressions in smaller steps.
9. *If expressions* support conditional evaluation.
10. *Sections* provide a simple modularity mechanism. (Sections are not yet leveraged by Power Query.)
11. Finally, a *consolidated grammar* collects the grammar fragments from all other sections of this document into a single complete definition.

For computer language theorists: the formula language specified in this document is a mostly pure, higher-order, dynamically typed, partially lazy functional language.

Expressions and values

The central construct in M is the *expression*. An expression can be evaluated (computed), yielding a single *value*.

Although many values can be written literally as an expression, a value is not an expression. For example, the expression `1` evaluates to the value *1*; the expressions `1+1` evaluates to the value *2*. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

The following examples illustrate the different kinds of values available in M. As a convention, a value is written using the literal form in which they would appear in an expression that evaluates to just that value. (Note that the `//` indicates the start of a comment which continues to the end of the line.)

- A *primitive* value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

```
123                 // A number
true                // A logical
"abc"               // A text
null                // null value
```

- A *list* value is an ordered sequence of values. M supports infinite lists, but if written as a literal, lists have a fixed length. The curly brace characters `{` and `}` denote the beginning and end of a list.

```
{123, true, "A"}     // list containing a number, a logical, and
                      //     a text
{1, 2, 3}            // list of three numbers
```

- A *record* is a set of *fields*. A field is a name/value pair where the name is a text value that is unique within the field's record. The literal syntax for record values allows the names to be written without quotes, a form also referred to as *identifiers*. The following shows a record containing three fields named "`A`", "`B`", and "`C`", which have values `1`, `2`, and `3`.

```
[
      A = 1,
      B = 2,
      C = 3
]
```

- A *table* is a set of values organized into columns (which are identified by name), and rows. There is no literal syntax for creating a table, but there are several standard functions that can be used to create tables from lists or records.

  For example:

```
#table( {"A", "B"}, { {1, 2}, {3, 4} } )
```

  This creates a table of the following shape:

A	B
1	2
3	4

- A *function* is a value which, when invoked with arguments, produces a new value. Function are written by listing the function's *parameters* in parentheses, followed by the goes-to symbol `=>`, followed by the expression defining the function. That expression typically refers to the parameters (by name).

```
(x, y) => (x + y) / 2`
```

Evaluation

The evaluation model of the M language is modeled after the evaluation model commonly found in spreadsheets, where the order of calculation can be determined based on dependencies between the formulas in the cells.

If you have written formulas in a spreadsheet such as Excel, you may recognize the formulas on the left will result in the values on the right when calculated:

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

In M, parts of an expression can reference other parts of the expression by name, and the evaluation process will automatically determine the order in which referenced expressions are calculated.

We can use a record to produce an expression which is equivalent to the above spreadsheet example. When initializing the value of a field, we can refer to other fields within the record by using the name of the field, as follows:

```
[
    A1 = A2 * 2,
    A2 = A3 + 1,
    A3 = 1
]
```

The above expression is equivalent to the following (in that both evaluate to equal values):

```
[
    A1 = 4,
    A2 = 2,
    A3 = 1
]
```

Records can be contained within, or *nest*, within other records. We can use the *lookup operator* (`[]`) to access the fields of a record by name. For example, the following record has a field named `Sales` containing a record, and a field named `Total` that accesses the `FirstHalf` and `SecondHalf` fields of the `Sales` record:

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

The above expression is equivalent to the following when it is evaluated: \

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

Records can also be contained within lists. We can use the *positional index operator* (`{}`) to access an item in a list by its numeric index. The values within a list are referred to using a zero-based index from the beginning of the list. For example, the indexes `0` and `1` are used to reference the first and second items in the list below:

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // 2500
            ]
        },
    TotalSales = Sales{0}[Total] + Sales{1}[Total] // 4600
]
```

List and record member expressions (as well as let expressions, introduced further below) are evaluated using *lazy evaluation*, which means that they are evaluated only as needed. All other expressions are evaluated using *eager evaluation*, which means that they are evaluated immediately, when encountered during the evaluation process. A good way to think about this is to remember that evaluating a list or record expression will return a list or record value that itself remembers how its list items or record fields need to be computed, when requested (by lookup or index operators).

Functions

In M, a *function* is a mapping from a set of input values to a single output value. A function is written by first naming the required set of input values (the parameters to the function) and then providing an expression that will compute the result of the function using those input values (the body of the function) following the goes-to (`=>`) symbol. For example:

```
(x) => x + 1                    // function that adds one to a value
(x, y) =>  x + y                // function that adds two values
```

A function is a value just like a number or a text value. The following example shows a function which is the value of an Add field which is then *invoked*, or executed, from several other fields. When a function is invoked, a set of values are specified which are logically substituted for the required set of input values within the function body expression.

```
[
    Add = (x, y) => x + y,
    OnePlusOne = Add(1, 1),     // 2
    OnePlusTwo = Add(1, 2)      // 3
]
```

Library

M includes a common set of definitions available for use from an expression called the *standard library*, or just library for short. These definitions consist of a set of named values. The names of values provided by a library are available for use within an expression without having been defined explicitly by the expression. For example:

```
Number.E                        // Euler's number e (2.7182...)
Text.PositionOf("Hello", "ll")  // 2
```

Operators

M includes a set of operators that can be used in expressions. *Operators* are applied to *operands* to form symbolic expressions. For example, in the expression `1 + 2` the numbers `1` and `2` are operands and the operator is the addition operator (`+`).

The meaning of an operator can vary depending on what kind of values its operands are. For example, the plus operator can be used with other kinds of values than numbers:

```
1 + 2                   // numeric addition: 3
#time(12,23,0) + #duration(0,0,2,0)
                        // time arithmetic: #time(12,25,0)
```

Another example of an operator with operand-depending meaning is the combination operator (`&`):

```
"A" & "BC"              // text concatenation: "ABC"
{1} & {2, 3}            // list concatenation: {1, 2, 3}
[ a = 1 ] & [ b = 2 ]   // record merge: [ a = 1, b = 2 ]
```

Note that not all combinations of values may be supported by an operator. For example:

```
1 + "2"  // error: adding number and text is not supported
```

Expressions that, when evaluated, encounter undefined operator conditions evaluate to errors. More on errors in M later.

Metadata

Metadata is information about a value that is associated with a value. Metadata is represented as a record value, called a *metadata record*. The fields of a metadata record can be used to store the metadata for a value.

Every value has a metadata record. If the value of the metadata record has not been specified, then the metadata record is empty (has no fields).

Metadata records provide a way to associate additional information with any kind of value in an unobtrusive way. Associating a metadata record with a value does not change the value or its behavior.

A metadata record value `y` is associated with an existing value `x` using the syntax `x meta y`. For example, the following associates a metadata record with `Rating` and `Tags` fields with the text value `"Mozart"`:

```
"Mozart" meta [ Rating = 5, Tags = {"Classical"} ]
```

For values that already carry a non-empty metadata record, the result of applying meta is that of computing the record merge of the existing and the new metadata record. For example, the following two expressions are equivalent to each other and to the previous expression:

```
("Mozart" meta [ Rating = 5 ]) meta [ Tags = {"Classical"} ]
"Mozart" meta ([ Rating = 5 ] & [ Tags = {"Classical"} ])
```

A metadata record can be accessed for a given value using the *Value.Metadata* function. In the following example, the expression in the `ComposerRating` field accesses the metadata record of the value in the `Composer`

field, and then accesses the `Rating` field of the metadata record.

```
[
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
    ComposerRating = Value.Metadata(Composer)[Rating] // 5
]
```

Let expression

Many of the examples shown so far have included all the literal values of the expression in the result of the expression. The *let* expression allows a set of values to be computed, assigned names, and then used in a subsequent expression that follows the *in*. For example, in our sales data example, we could do:

```
let
    Sales2007 =
        [
            Year = 2007,
            FirstHalf = 1000,
            SecondHalf = 1100,
            Total = FirstHalf + SecondHalf // 2100
        ],
    Sales2008 =
        [
            Year = 2008,
            FirstHalf = 1200,
            SecondHalf = 1300,
            Total = FirstHalf + SecondHalf // 2500
        ]
  in Sales2007[Total] + Sales2008[Total] // 4600
```

The result of the above expression is a number value (`4600`) which was computed from the values bound to the names `Sales2007` and `Sales2008` .

If expression

The `if` expression selects between two expressions based on a logical condition. For example:

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression (`2 + 2`) is selected if the logical expression (`2 > 1`) is true, and the second expression (`1 + 1`) is selected if it is false. The selected expression (in this case `2 + 2`) is evaluated and becomes the result of the `if` expression (`4`).

Errors

An *error* is an indication that the process of evaluating an expression could not produce a value.

Errors are raised by operators and functions encountering error conditions or by using the error expression. Errors are handled using the try expression. When an error is raised, a value is specified that can be used to indicate why the error occurred.

```
let Sales =
    [
        Revenue = 2000,
        Units = 1000,
        UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
    ],
    UnitPrice = try Number.ToText(Sales[UnitPrice])
in "Unit Price: " &
    (if UnitPrice[HasError] then UnitPrice[Error][Message]
    else UnitPrice[Value])
```

The above example accesses the `Sales[UnitPrice]` field and formats the value producing the result:

```
"Unit Price: 2"
```

If the `Units` field had been zero, then the `UnitPrice` field would have raised an error which would have been handled by the `try`. The resulting value would then have been:

```
"No Units"
```

A `try` expression converts proper values and errors into a record value that indicates whether the try expression handled and error, or not, and either the proper value or the error record it extracted when handling the error. For example, consider the following expression that raises an error and then handles it right away:

```
try error "negative unit count"
```

This expression evaluates to the following nested record value, explaining the `[HasError]`, `[Error]`, and `[Message]` field lookups in the unit-price example before.

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

A common case is to replace errors with default values. The `try` expression can be used with an optional `otherwise` clause to achieve just that in a compact form:

```
try error "negative unit count" otherwise 42
// 42
```

Lexical Structure

12/11/2020 • 9 minutes to read

Documents

An M *document* is an ordered sequence of Unicode characters. M allows different classes of Unicode characters in different parts of an M document. For information on Unicode character classes, see *The Unicode Standard, Version 3.0*, section 4.5.

A document either consists of exactly one *expression* or of groups of *definitions* organized into *sections*. Sections are described in detail in Chapter 10. Conceptually speaking, the following steps are used to read an expression from a document:

1. The document is decoded according to its character encoding scheme into a sequence of Unicode characters.
2. Lexical analysis is performed, thereby translating the stream of Unicode characters into a stream of tokens. The remaining subsections of this section cover lexical analysis.
3. Syntactic analysis is performed, thereby translating the stream of tokens into a form that can be evaluated. This process is covered in subsequent sections.

Grammar conventions

The lexical and syntactic grammars are presented using *grammar productions*. Each grammar production defines a non-terminal symbol and the possible expansions of that nonterminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, _non-terminal+ symbols are shown in italic type, and *terminal* symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the nonterminal given as a sequence of non-terminal or terminal symbols. For example, the production:

if-expression:
    `if` *if-condition* `then` *true-expression* `else` *false-expression*

defines an *if-expression* to consist of the token `if`, followed by an *if-condition*, followed by the token `then`, followed by a *true-expression*, followed by the token `else`, followed by a *false-expression*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

variable-list:
    variable
    variable-list `,` *variable*

defines a *variable-list* to either consist of a *variable* or consist of a *variable-list* followed by a *variable*. In other words, the definition is recursive and specifies that a variable list consists of one or more variables, separated by commas.

A subscripted suffix "$_{opt}$" is used to indicate an optional symbol. The production:

field-specification:
    `optional`$_{opt}$ *field-name* `=` *field-type*

is shorthand for:

field-specification:
    field-name `=` *field-type*
    `optional` *field-name* `=` *field-type*

and defines a *field-specification* to optionally begin with the terminal symbol `optional` followed by a *field-name*, the terminal symbol `=`, and a *field-type*.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

decimal-digit: one of
    `0 1 2 3 4 5 6 7 8 9`

is shorthand for:

decimal-digit:
    `0`
    `1`
    `2`
    `3`
    `4`
    `5`
    `6`
    `7`
    `8`
    `9`

Lexical Analysis

The *lexical-unit* production defines the lexical grammar for an M document. Every valid M document conforms to this grammar.

lexical-unit:
    lexical-elements$_{opt}$
lexical-elements:
    lexical-element
    lexical-element
    lexical-elements
lexical-element:
    whitespace
    token comment

At the lexical level, an M document consists of a stream of *whitespace*, *comment*, and *token* elements. Each of these productions is covered in the following sections. Only *token* elements are significant in the syntactic grammar.

Whitespace

Whitespace is used to separate comments and tokens within an M document. Whitespace includes the space character (which is part of Unicode class Zs), as well as horizontal and vertical tab, form feed, and newline character sequences. Newline character sequences include carriage return, line feed, carriage return followed by line feed, next line, and paragraph separator characters.

whitespace:
    Any character with Unicode class Zs
    Horizontal tab character (`U+0009`)
    Vertical tab character (`U+000B`)
    Form feed character (`U+000C`)
    Carriage return character (`U+000D`) followed by line feed character (`U+000A`)
    new-line-character
new-line-character:
    Carriage return character (`U+000D`)
    Line feed character (`U+000A`)
    Next line character (`U+0085`)
    Line separator character (`U+2028`)
    Paragraph separator character (`U+2029`)

For compatibility with source code editing tools that add end-of-file markers, and to enable a document to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to an M document:

- If the last character of the document is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the document if that document is non-empty and if the last character of the document is not a carriage return (`U+000D`), a line feed (`U+000A`), a line separator (`U+2028`), or a paragraph separator (`U+2029`).

Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters `//` and extend to the end of the source line. *Delimited comments* start with the characters `/*` and end with the characters `*/`.

Delimited comments may span multiple lines.

comment:
    single-line-comment
    delimited-comment
single-line-comment:
    `//` *single-line-comment-characters*$_{opt}$
single-line-comment-characters:
    single-line-comment-character single-line-comment-characters$_{opt}$
single-line-comment-character:
    Any Unicode character except a *new-line-character*
delimited-comment:
    `/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`
delimited-comment-text:
    delimited-comment-section delimited-comment-text$_{opt}$
delimited-comment-section:
    `/`
    asterisks$_{opt}$ *not-slash-or-asterisk*
asterisks:
    `*` *asterisks*$_{opt}$
not-slash-or-asterisk:
    Any Unicode character except `*` or `/`

Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a single-line

comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

Comments are not processed within text literals. The example

```
/* Hello, world
*/
    "Hello, world"
```

includes a delimited comment.

The example

```
// Hello, world
//
"Hello, world" // This is an example of a text literal
```

shows several single-line comments.

Tokens

A *token* is an identifier, keyword, literal, operator, or punctuator. Whitespace and comments are used to separate tokens, but are not considered tokens.

token:
- *identifier*
- *keyword*
- *literal*
- *operator-or-punctuator*

Character Escape Sequences

M text values can contain arbitrary Unicode characters. Text literals, however, are limited to graphic characters and require the use of *escape sequences* for non-graphic characters. For example, to include a carriage-return, linefeed, or tab character in a text literal, the `#(cr)`, `#(lf)`, and `#(tab)` escape sequences can be used, respectively. To embed the escapesequence start characters `#(` in a text literal, the `#` itself needs to be escaped:

```
#(#)(
```

Escape sequences can also contain short (four hex digits) or long (eight hex digits) Unicode code-point values. The following three escape sequences are therefore equivalent:

```
#(000D)     // short Unicode hexadecimal value
#(0000000D) // long Unicode hexadecimal value
#(cr)       // compact escape shorthand for carriage return
```

Multiple escape codes can be included in a single escape sequence, separated by commas; the following two sequences are thus equivalent:

```
#(cr,lf)
#(cr)#(lf)
```

The following describes the standard mechanism of character escaping in an M document.

character-escape-sequence:
- `#(` *escape-sequence-list* `)`

escape-sequence-list:
    single-escape-sequence
    single-escape-sequence `,` *escape-sequence-list*
single-escape-sequence:
    long-unicode-escape-sequence
    short-unicode-escape-sequence
    control-character-escape-sequence
    escape-escape
long-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
short-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit
control-character-escape-sequence:
    control-character
control-character:
    `cr`
    `lf`
    `tab`
escape-escape:
    `#`

Literals

A *literal* is a source code representation of a value.

literal:
    logical-literal
    number-literal
    text-literal
    null-literal
    verbatim-literal

Null literals

The null literal is used to write the `null` value. The `null` value represents an absent value.

null-literal:
    `null`

Logical literals

A logical literal is used to write the values `true` and `false` and produces a logical value.

logical-literal:
    `true`
    `false`

Number literals

A number literal is used to write a numeric value and produces a number value.

number-literal:
    decimal-number-literal
    hexadecimal-number-literal
decimal-number-literal:
    decimal-digits `.` *decimal-digits exponent-part*$_{opt}$
    `.` *decimal-digits exponent-part*$_{opt}$
    decimal-digits exponent-part$_{opt}$
decimal-digits:

decimal-digit decimal-digits$_{opt}$

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

exponent-part:

`e` *sign*$_{opt}$ *decimal-digits*

`E` *sign*$_{opt}$ *decimal-digits*

sign: one of

`+ -`

hexadecimal-number-literal:

`0x` *hex-digits*

`0X` *hex-digits*

hex-digits:

hex-digit hex-digits$_{opt}$

hex-digit: one of

`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

A number can be specified in hexadecimal format by preceding the *hex-digits* with the characters `0x`. For example:

```
0xff // 255
```

Note that if a decimal point is included in a number literal, then it must have at least one digit following it. For example, `1.3` is a number literal but `1.` and `1.e3` are not.

Text literals

A text literal is used to write a sequence of Unicode characters and produces a text value.

text-literal:

`"` *text-literal-characters*$_{opt}$ `"`

text-literal-characters:

text-literal-character text-literal-characters$_{opt}$

text-literal-character:

single-text-character

character-escape-sequence

double-quote-escape-sequence

single-text-character:

Any character except `"` (`U+0022`) or `#` (`U+0023`) followed by `(` (`U+0028`)

double-quote-escape-sequence:

`""` (`U+0022`, `U+0022`)

To include quotes in a text value, the quote mark is repeated, as follows:

```
"The ""quoted"" text" // The "quoted" text
```

The *character-escape-sequence* production can be used to write characters in text values without having to directly encode them as Unicode characters in the document. For example, a carriage return and line feed can be written in a text value as:

```
"Hello world#(cr,lf)"
```

Verbatim literals

A verbatim literal is used to store a sequence of Unicode characters that were entered by a user as code, but

which cannot be correctly parsed as code. At runtime, it produces an error value.

verbatim-literal:
`#!"` *text-literal-characters*$_{opt}$ `"`

Identifiers

An *identifier* is a name used to refer to a value. Identifiers can either be regular identifiers or quoted identifiers.

identifier:
regular-identifier
quoted-identifier
regular-identifier:
available-identifier
available-identifier dot-character regular-identifier
available-identifier:
A *keyword-or-identifier* that is not a *keyword*
keyword-or-identifier:
identifier-start-character identifier-part-characters$_{opt}$
identifier-start-character:
letter-character
underscore-character
identifier-part-characters:
identifier-part-character identifier-part-characters$_{opt}$
identifier-part-character:
letter-character
decimal-digit-character
underscore-character
connecting-character
combining-character
formatting-character
dot-character:
`.` (`U+002E`)
underscore-character:
`_` (`U+005F`)
letter-character:
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
combining-character:
A Unicode character of classes Mn or Mc
decimal-digit-character:
A Unicode character of the class Nd
connecting-character:
A Unicode character of the class Pc
formatting-character:
A Unicode character of the class Cf

A *quoted-identifier* can be used to allow any sequence of zero or more Unicode characters to be used as an identifier, including keywords, whitespace, comments, operators and punctuators.

quoted-identifier:
`#"` *text-literal-characters*$_{opt}$ `"`

Note that escape sequences and double-quotes to escape quotes can be used in a *quoted identifier*, just as in a *text-literal*.

The following example uses identifier quoting for names containing a space character:

```
[
    #"1998 Sales" = 1000,
    #"1999 Sales" = 1100,
    #"Total Sales" = #"1998 Sales" + #"1999 Sales"
]
```

The following example uses identifier quoting to include the `+` operator in an identifier:

```
[
    #"A + B" = A + B,
    A = 1,
    B = 2
]
```

Generalized Identifiers

There are two places in M where no ambiguities are introduced by identifiers that contain blanks or that are otherwise keywords or number literals. These places are the names of record fields in a record literal and in a field access operator (`[ ]`) There, M allows such identifiers without having to use quoted identifiers.

```
[
    Data = [ Base Line = 100, Rate = 1.8 ],
    Progression = Data[Base Line] * Data[Rate]
]
```

The identifiers used to name and access fields are referred to as *generalized identifiers* and defined as follows:

generalized-identifier:
    generalized-identifier-part
    generalized-identifier separated only by blanks (`U+0020`)
generalized-identifier-part
generalized-identifier-part:
    generalized-identifier-segment
    decimal-digit-character generalized-identifier-segment
generalized-identifier-segment:
    keyword-or-identifier
    keyword-or-identifier dot-character keyword-or-identifier

Keywords

A *keyword* is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when using the identifier-quoting mechanism or where a generalized identifier is allowed.

keyword: one of

`and as each else error false if in is let meta not null or otherwise`
`section shared then true try type #binary #date #datetime`
`#datetimezone #duration #infinity #nan #sections #shared #table #time`

Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

operator-or-punctuator: one of

`, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ! ? => .. ...`

Basic concepts

12/11/2020 • 8 minutes to read

This section discusses basic concepts that appear throughout the subsequent sections.

Values

A single piece of data is called a *value*. Broadly speaking, there are two general categories of values: *primitive values*, which are atomic, and *structured values*, which are constructed out of primitive values and other structured values. For example, the values \

```
1
true
3.14159
"abc"
```

are primitive in that they are not made up of other values. On the other hand, the values

```
{1, 2, 3}
[ A = {1}, B = {2}, C = {3} ]
```

are constructed using primitive values and, in the case of the record, other structured values.

Expressions

An *expression* is a formula used to construct values. An expression can be formed using a variety of syntactic constructs. The following are some examples of expressions. Each line is a separate expression.

```
"Hello World"             // a text value
123                       // a number
1 + 2                     // sum of two numbers
{1, 2, 3}                 // a list of three numbers
[ x = 1, y = 2 + 3 ]      // a record containing two fields:
                          //        x and y
(x, y) => x + y           // a function that computes a sum
if 2 > 1 then 2 else 1    // a conditional expression
let x = 1 + 1  in x * 2   // a let expression
error "A"                 // error with message "A"
```

The simplest form of expression, as seen above, is a literal representing a value.

More complex expressions are built from other expressions, called *sub-expressions*. For example:

```
1 + 2
```

The above expression is actually composed of three expressions. The `1` and `2` literals are subexpressions of the parent expression `1 + 2`.

Executing the algorithm defined by the syntactic constructs used in an expression is called *evaluating* the expression. Each kind of expression has rules for how it is evaluated. For example, a literal expression like `1` will produce a constant value, while the expression `a + b` will take the resulting values produced by evaluating two

other expressions (`a` and `b`) and add them together according to some set of rules.

Environments and variables

Expressions are evaluated within a given environment. An *environment* is a set of named values, called *variables*. Each variable in an environment has a unique name within the environment called an *identifier*.

A top-level (or *root*) expression is evaluated within the *global environment*. The global environment is provided by the expression evaluator instead of being determined from the contents of the expression being evaluated. The contents of the global environment includes the standard library definitions and can be affected by exports from sections from some set of documents. (For simplicity, the examples in this section will assume an empty global environment. That is, it is assumed that there is no standard library and that there are no other section-based definitions.)

The environment used to evaluate a sub-expression is determined by the parent expression. Most parent expression kinds will evaluate a sub-expression within the same environment they were evaluated within, but some will use a different environment. The global environment is the *parent environment* within which the global expression is evaluated.

For example, the *record-initializer-expression* evaluates the sub-expression for each field with a modified environment. The modified environment includes a variable for each of the fields of the record, except the one being initialized. Including the other fields of the record allows the fields to depend upon the values of the fields. For example:

```
[
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
]
```

Similarly, the *let-expression* evaluates the sub-expression for each variable with an environment containing each of the variables of the let except the one being initialized. The *let-expression* evaluates the expression following the in with an environment containing all the variables:

```
let

    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
in
    x + y + z       // environment: x, y, z
```

(It turns out that both *record-initializer-expression* and *let-expression* actually define *two* environments, one of which does include the variable being initialized. This is useful for advanced recursive definitions and is covered in Identifier references .

To form the environments for the sub-expressions, the new variables are "merged" with the variables in the parent environment. The following example shows the environments for nested records:

```
[
    a =
    [

        x = 1,      // environment: b, y, z
        y = 2,      // environment: b, x, z
        z = x + y   // environment: b, x, y
    ],
    b = 3           // environment: a
]
```

The following example shows the environments for a record nested within a let:

```
Let
    a =
    [
        x = 1,       // environment: b, y, z
        y = 2,       // environment: b, x, z
        z = x + y    // environment: b, x, y
    ],
    b = 3            // environment: a
in
    a[z] + b         // environment: a, b
```

Merging variables with an environment may introduce a conflict between variables (since each variable in an environment must have a unique name). The conflict is resolved as follows: if the name of a new variable being merged is the same as an existing variable in the parent environment, then the new variable will take precedence in the new environment. In the following example, the inner (more deeply nested) variable `x` will take precedence over the outer variable `x`.

```
[
    a =
    [
        x = 1,       // environment: b, x (outer), y, z
        y = 2,       // environment: b, x (inner), z
        z = x + y    // environment: b, x (inner), y
    ],
    b = 3,           // environment: a, x (outer)
    x = 4            // environment: a, b
]
```

Identifier references

An *identifier-reference* is used to refer to a variable within an environment.

identifier-expression:
    identifier-reference
identifier-reference:
    exclusive-identifier-reference
    inclusive-identifier-reference

The simplest form of identifier reference is an *exclusive-identifier-reference*.

exclusive-identifier-reference:
    identifier

It is an error for an *exclusive-identifier-reference* to refer to a variable that is not part of the environment of the expression that the identifier appears within, or to refer to an identifier that is currently being initialized.

An *inclusive-identifier-reference* can be used to gain access to the environment that includes the identifier being initialized. If it used in a context where there is no identifier being initialized, then it is equivalent to an *exclusive-identifier-reference*.

inclusive-identifier-reference:
`@` *identifier*

This is useful when defining recursive functions since the name of the function would normally not be in scope.

```
[
    Factorial = (n) =>
        if n <= 1 then
            1
        else
            n * @Factorial(n - 1),  // @ is scoping operator

    x = Factorial(5)
]
```

As with a *record-initializer-expression*, an *inclusive-identifier-reference* can be used within a *let-expression* to access the environment that includes the identifier being initialized.

Order of evaluation

Consider the following expression which initializes a record:

```
[
    C = A + B,
    A = 1 + 1,
    B = 2 + 2
]
```

When evaluated, this expression produces the following record value:

```
[
    C = 6,
    A = 2,
    B = 4
]
```

The expression states that in order to perform the `A + B` calculation for field `C`, the values of both field `A` and field `B` must be known. This is an example of a *dependency ordering* of calculations that is provided by an expression. The M evaluator abides by the dependency ordering provided by expressions, but is free to perform the remaining calculations in any order it chooses. For example, the computation order could be:

```
A = 1 + 1
B = 2 + 2
C = A + B
```

Or it could be:

```
B = 2 + 2
A = 1 + 1
C = A + B
```

Or, since `A` and `B` do not depend on each other, they can be computed concurrently:

`B = 2 + 2` *concurrently with* `A = 1 + 1`
`C = A + B`

Side effects

Allowing an expression evaluator to automatically compute the order of calculations for cases where there are no explicit dependencies stated by the expression is a simple and powerful computation model.

It does, however, rely on being able to reorder computations. Since expressions can call functions, and those functions could observe state external to the expression by issuing external queries, it is possible to construct a scenario where the order of calculation does matter, but is not captured in the partial order of the expression. For example, a function may read the contents of a file. If that function is called repeatedly, then external changes to that file can be observed and, therefore, reordering can cause observable differences in program behavior. Depending on such observed evaluation ordering for the correctness of an M expression causes a dependency on particular implementation choices that might vary from one evaluator to the next or may even vary on the same evaluator under varying circumstances.

Immutability

Once a value has been calculated, it is *immutable*, meaning it can no longer be changed. This simplifies the model for evaluating an expression and makes it easier to reason about the result since it is not possible to change a value once it has been used to evaluate a subsequent part of the expression. For instance, a record field is only computed when needed. However, once computed, it remains fixed for the lifetime of the record. Even if the attempt to compute the field raised an error, that same error will be raised again on every attempt to access that record field.

An important exception to the immutable-once-calculated rule applies to list and table values. Both have *streaming semantics*. That is, repeated enumeration of the items in a list or the rows in a table can produce varying results. Streaming semantics enables the construction of M expressions that transform data sets that would not fit in memory at once.

Also, note that function application is *not* the same as value construction. Library functions may expose external state (such as the current time or the results of a query against a database that evolves over time), rendering them *non-deterministic*. While functions defined in M will not, as such, expose any such non-deterministic behavior, they can if they are defined to invoke other functions that are non-deterministic.

A final source of non-determinsm in M are *errors*. Errors stop evaluations when they occur (up to the level where they are handled by a try expression). It is not normally observable whether `a + b` caused the evaluation of `a` before `b` or `b` before `a` (ignoring concurrency here for simplicity). However, if the subexpression that was evaluated first raises an error, then it can be determined which of the two expressions was evaluated first.

Values

12/11/2020 • 17 minutes to read

A value is data produced by evaluating an expression. This section describes the kinds of values in the M language. Each kind of value is associated with a literal syntax, a set of values that are of that kind, a set of operators defined over that set of values, and an intrinsic type ascribed to newly constructed values.

KIND	LITERAL
Null	`null`
Logical	`true` `false`
Number	`0` `1` `-1` `1.5` `2.3e-5`
Time	`#time(09,15,00)`
Date	`#date(2013,02,26)`
DateTime	`#datetime(2013,02,26, 09,15,00)`
DateTimeZone	`#datetimezone(2013,02,26, 09,15,00, 09,00)`
Duration	`#duration(0,1,30,0)`
Text	`"hello"`
Binary	`#binary("AQID")`
List	`{1, 2, 3}`
Record	`[ A = 1, B = 2 ]`
Table	`#table({"X","Y"},{{0,1},{1,0}})`
Function	`(x) => x + 1`
Type	`type { number }` `type table [ A = any, B = text ]`

The following sections cover each value kind in detail. Types and type ascription are defined formally in Types. Function values are defined in Functions. The following sections list the operators defined for each value kind and give examples. The full definition of operator semantics follows in Operators.

Null

A *null value* is used to represent the absence of a value, or a value of indeterminate or unknown state. A null value is written using the literal `null`. The following operators are defined for null values:

OPERATOR	RESULT
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal

The native type of the `null` value is the intrinsic type `null`.

Logical

A *logical value* is used for Boolean operations has the value true or false. A logical value is written using the literals `true` and `false`. The following operators are defined for logical values:

OPERATOR	RESULT
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal
`x or y`	Conditional logical OR
`x and y`	Conditional logical AND
`not x`	Logical NOT

The native type of both logical values (`true` and `false`) is the intrinsic type `logical`.

Number

A *number value* is used for numeric and arithmetic operations. The following are examples of number literals:

```
3.14  // Fractional number
-1.5  // Fractional number
1.0e3 // Fractional number with exponent
123   // Whole number
1e3   // Whole number with exponent
0xff  // Whole number in hex (255)
```

A number is represented with at least the precision of a *Double* (but may retain more precision). The *Double* representation is congruent with the IEEE 64-bit double precision standard for binary floating point arithmetic defined in [IEEE 754-2008]. (The *Double* representation have an approximate dynamic range from 5.0 x 10^{324} to 1.7 x 10^{308} with a precision of 15-16 digits.)

The following special values are also considered to be *number* values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two.
- Positive infinity (`#infinity`) and negative infinity (`-#infinity`). Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The *Not-a-Number* value (`#nan`), often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.

Binary mathematical operations are performed using a *Precision*. The precision determines the domain to which the operands are rounded and the domain in which the operation is performed. In the absence of an explicitly specified precision, such operations are performed using *Double Precision*.

- If the result of a mathematical operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a mathematical operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a mathematical operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

The following operators are defined for number values:

OPERATOR	RESULT
`x > y`	Greater than
`x >= y`	Greater than or equal
`x < y`	Less than
`x <= y`	Less than or equal
`x = y`	Equal
`x <> y`	Not equal
`x + y`	Sum

OPERATOR	RESULT
`x - y`	Difference
`x * y`	Product
`x / y`	Quotient
`+x`	Unary plus
`-x`	Negation

The native type of number values is the intrinsic type `number`.

Time

A *time value* stores an opaque representation of time of day. A time is encoded as the number of *ticks since midnight*, which counts the number of 100-nanosecond ticks that have elapsed on a 24-hour clock. The maximum number of *ticks since midnight* corresponds to 23:59:59.9999999 hours.

Time values may be constructed using the #time instrinsic.

```
#time(hour, minute, second)
```

The following must hold or an error with reason code `Expression.Error` is raised:

0 ≤ hour ≤ 24
0 ≤ minute ≤ 59
0 ≤ second ≤ 59

In addition, if hour = 24, then minute and second must be zero.

The following operators are defined for time values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a date:

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
`x + y`	`time`	`duration`	Date offset by duration
`x + y`	`duration`	`time`	Date offset by duration
`x - y`	`time`	`duration`	Date offset by negated duration
`x - y`	`time`	`time`	Duration between dates
`x & y`	`date`	`time`	Merged datetime

The native type of time values is the intrinsic type `time`.

Date

A *date value* stores an opaque representation of a specific day. A date is encoded as a number of *days since epoch*, starting from January 1, 0001 Common Era on the Gregorian calendar. The maximum number of days since epoch is 3652058, corresponding to December 31, 9999.

Date values may be constructed using the `#date` intrinsic.

```
#date(year, month, day)
```

The following must hold or an error with reason code `Expression.Error` is raised:

1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31

In addition, the day must be valid for the chosen month and year.

The following operators are defined for date values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a date:

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
`x + y`	`date`	`duration`	Date offset by duration
`x + y`	`duration`	`date`	Date offset by duration
`x - y`	`date`	`duration`	Date offset by negated duration
`x - y`	`date`	`date`	Duration between dates
`x & y`	`date`	`time`	Merged datetime

The native type of date values is the intrinsic type `date`.

DateTime

A *datetime value* contains both a date and time.

DateTime values may be constructed using the `#datetime` intrinsic.

```
#datetime(year, month, day, hour, minute, second)
```

The following must hold or an error with reason code Expression.Error is raised: 1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31
0 ≤ hour ≤ 23
0 ≤ minute ≤ 59
0 ≤ second ≤ 59

In addition, the day must be valid for the chosen month and year.

The following operators are defined for datetime values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a datetime:

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
`x + y`	`datetime`	`duration`	Datetime offset by duration
`x + y`	`duration`	`datetime`	Datetime offset by duration
`x - y`	`datetime`	`duration`	Datetime offset by negated duration
`x - y`	`datetime`	`datetime`	Duration between datetimes

The native type of datetime values is the intrinsic type `datetime`.

DateTimeZone

A *datetimezone* value contains a datetime and a timezone. A *timezone* is encoded as a number of *minutes offset from UTC*, which counts the number of minutes the time portion of the *datetime* should be offset from Universal Coordinated Time (UTC). The minimum number of *minutes offset from UTC* is -840, representing a UTC offset of -14:00, or fourteen hours earlier than UTC. The maximum number of *minutes offset from UTC* is 840, corresponding to a UTC offset of 14:00.

DateTimeZone values may be constructed using the `#datetimezone` intrinsic.

```
#datetimezone(
       year, month, day,
       hour, minute, second,
       offset-hours, offset-minutes)
```

The following must hold or an error with reason code `Expression.Error` is raised:

1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31
0 ≤ hour ≤ 23
0 ≤ minute ≤ 59
0 ≤ second ≤ 59
-14 ≤ offset-hours ≤ 14
-59 ≤ offset-minutes ≤ 59

In addition, the day must be valid for the chosen month and year and, if offset-hours = 14, then offset-minutes <= 0 and, if offset-hours = -14, then offset-minutes >= 0.

The following operators are defined for datetimezone values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal

OPERATOR	RESULT
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

The following operators permit one or both of their operands to be a datetimezone:

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
`x + y`	`datetimezone`	`duration`	Datetimezone offset by duration
`x + y`	`duration`	`datetimezone`	Datetimezone offset by duration
`x - y`	`datetimezone`	`duration`	Datetimezone offset by negated duration
`x - y`	`datetimezone`	`datetimezone`	Duration between datetimezones

The native type of datetimezone values is the intrinsic type `datetimezone`.

Duration

A *duration value* stores an opaque representation of the distance between two points on a timeline measured 100-nanosecond ticks. The magnitude of a *duration* can be either positive or negative, with positive values denoting progress forwards in time and negative values denoting progress backwards in time. The minimum value that can be stored in a *duration* is -9,223,372,036,854,775,808 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775808 seconds backwards in time. The maximum value that can be stored in a *duration* is 9,223,372,036,854,775,807 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775807 seconds forwards in time.

Duration values may be constructed using the `#duration` intrinsic function:

```
#duration(0, 0, 0, 5.5)          // 5.5 seconds
#duration(0, 0, 0, -5.5)         // -5.5 seconds
#duration(0, 0, 5, 30)           // 5.5 minutes
#duration(0, 0, 5, -30)          // 4.5 minutes
#duration(0, 24, 0, 0)           // 1 day
#duration(1, 0, 0, 0)            // 1 day
```

The following operators are defined on duration values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal

OPERATOR	RESULT
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal

Additionally, the following operators allow one or both of their operands to be a duration value:

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
`x + y`	`datetime`	`duration`	Datetime offset by duration
`x + y`	`duration`	`datetime`	Datetime offset by duration
`x + y`	`duration`	`duration`	Sum of durations
`x - y`	`datetime`	`duration`	Datetime offset by negated duration
`x - y`	`datetime`	`datetime`	Duration between datetimes
`x - y`	`duration`	`duration`	Difference of durations
`x * y`	`duration`	`number`	N times a duration
`x * y`	`number`	`duration`	N times a duration
`x / y`	`duration`	`number`	Fraction of a duration

The native type of duration values is the intrinsic type `duration`.

Text

A *text* value represents a sequence of Unicode characters. Text values have a literal form conformant to the following grammar:

_text-literal:
    `"` text-literal-characters$_{opt}$ `"`
text-literal-characters:
    text-literal-character text-literal-characters$_{opt}$
text-literal-character:
    single-text-character
    character-escape-sequence
    double-quote-escape-sequence
single-text-character:

Any character except `"` (`U+0022`) or `#` (`U+0023`) followed by `(` (`U+0028`)

double-quote-escape-sequence:

`""` (`U+0022` , `U+0022`)

The following is an example of a *text* value:

```
"ABC" // the text value ABC
```

The following operators are defined on *text* values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than
`x <= y`	Less than or equal
`x & y`	Concatenation

The native type of text values is the intrinsic type `text` .

Binary

A *binary value* represents a sequence of bytes. There is no literal format. Several standard library functions are provided to construct binary values. For example, `#binary` can be used to construct a binary value from a list of bytes:

```
#binary( {0x00, 0x01, 0x02, 0x03} )
```

The following operators are defined on *binary* values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x >= y`	Greater than or equal
`x > y`	Greater than
`x < y`	Less than

OPERATOR	RESULT
`x <= y`	Less than or equal

The native type of binary values is the intrinsic type *binary*.

List

A *list value* is a value which produces a sequence of values when enumerated. A value produced by a list can contain any kind of value, including a list. Lists can be constructed using the initialization syntax, as follows:

list-expression:
    { *item-list*$_{opt}$ }
item-list:
    item
    item `,` *item-list*
item:
    expression
    expression `..` *expression*

The following is an example of a *list-expression* that defines a list with three text values: `"A"`, `"B"`, and `"C"`.

```
{"A", "B", "C"}
```

The value `"A"` is the first item in the list, and the value `"C"` is the last item in the list.

- The items of a list are not evaluated until they are accessed.
- While list values constructed using the list syntax will produce items in the order they appear in *item-list*, in general, lists returned from library functions may produce a different set or a different number of values each time they are enumerated.

To include a sequence of whole number in a list, the `a..b` form can be used:

```
{ 1, 5..9, 11 }     // { 1, 5, 6, 7, 8, 9, 11 }
```

The number of items in a list, known as the *list count*, can be determined using the `List.Count` function.

```
List.Count({true, false})  // 2
List.Count({})             // 0
```

A list may effectively have an infinite number of items; `List.Count` for such lists is undefined and may either raise an error or not terminate.

If a list contains no items, it is called an *empty list*. An empty list is written as:

```
{}  // empty list
```

The following operators are defined for lists:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x & y`	Concatenate

For example:

```
{1, 2} & {3, 4, 5}   // {1, 2, 3, 4, 5}
{1, 2} = {1, 2}      // true
{2, 1} <> {1, 2}     // true
```

The native type of list values is the intrinsic type `list`, which specifies an item type of `any`.

Record

A *record value* is an ordered sequence of fields. A *field* consists of a *field name*, which is a text value that uniquely identifies the field within the record, and a *field value*. The field value can be any kind of value, including record. Records can be constructed using initialization syntax, as follows:

record-expression:
    `[` *field-list*$_{opt}$ `]`
field-list:
    field
    field `,` *field-list*
field:
    field-name `=` *expression*
field-name:
    generalized-identifier
    quoted-identifier

The following example constructs a record with a field named `x` with value `1`, and a field named `y` with value `2`.

```
[ x = 1, y = 2 ]
```

The following example constructs a record with `a` field named a with a nested record value. The nested record has a field named `b` with value `2`.

```
[ a = [ b = 2 ] ]
```

The following holds when evaluating a record expression:

- The expression assigned to each field name is used to determine the value of the associated field.
- If the expression assigned to a field name produces a value when evaluated, then that becomes the value of the field of the resulting record.
- If the expression assigned to a field name raises an error when evaluated, then the fact that an error was raised is recorded with the field along with the error value that was raised. Subsequent access to that field

will cause an error to be re-raised with the recorded error value.

- The expression is evaluated in an environment like the parent environment only with variables merged in that correspond to the value of every field of the record, except the one being initialized.
- A value in a record is not evaluated until the corresponding field is accessed.
- A value in a record is evaluated at most once.
- The result of the expression is a record value with an empty metadata record.
- The order of the fields within the record is defined by the order that they appear in the *record-initializer-expression*.
- Every field name that is specified must be unique within the record, or it is an error. Names are compared using an ordinal comparison.

```
[ x = 1, x = 2 ] // error: field names must be unique
[ X = 1, x = 2 ] // OK
```

A record with no fields is called an *empty record*, and is written as follows:

```
[] // empty record
```

Although the order of the fields of a record is not significant when accessing a field or comparing two records, it is significant in other contexts such as when the fields of a record are enumerated.

The same two records produce different results when the fields are obtained:

```
Record.FieldNames([ x = 1, y = 2 ]) // [ "x", "y" ]
Record.FieldNames([ y = 1, x = 2 ]) // [ "y", "x" ]
```

The number of fields in a record can be determined using the `Record.FieldCount` function. For example:

```
Record.FieldCount([ x = 1, y = 2 })  // 2
Record.FieldCount([])                // 0
```

In addition to using the record initialization syntax `[ ]`, records can be constructed from a list of values, and a list of field names or a record type. For example:

```
Record.FromList({1, 2}, {"a", "b"})
```

The above is equivalent to:

```
[ a = 1, b = 2 ]
```

The following operators are defined for record values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal

OPERATOR	RESULT
`x & y`	Merge

The following examples illustrate the above operators. Note that record merge uses the fields from the right operand to override fields from the left operand, should there be an overlap in field names.

```
[ a = 1, b = 2 ] & [ c = 3 ]    // [ a = 1, b = 2, c = 3 ]
[ a = 1, b = 2 ] & [ a = 3 ]    // [ a = 3, b = 2 ]
[ a = 1, b = 2 ] = [ b = 2, a = 1 ]         // true
[ a = 1, b = 2, c = 3 ] <> [ a = 1, b = 2 ] // true
```

The native type of record values is the intrinsic type `record`, which specifies an open empty list of fields.

Table

A *table value* is an ordered sequence of rows. A *row* is an ordered sequence of value. The table's type determines the length of all rows in the table, the names of the table's columns, the types of the table's columns, and the structure of the table's keys (if any).

There is no literal syntax for tables. Several standard library functions are provided to construct binary values. For example, `#table` can be used to construct a table from a list of row lists and a list of header names:

```
#table({"x", "x^2"}, {{1,1}, {2,4}, {3,9}})
```

The above example constructs a table with two columns, both of which are of `type any`.

`#table` can also be used to specify a full table type:

```
#table(
    type table [Digit = number, Name = text],
    {{1,"one"}, {2,"two"}, {3,"three"}}
    )
```

Here the new table value has a table type that specifies column names and column types.

The following operators are defined for table values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal
`x & y`	Concatenation

Table concatenation aligns like-named columns and fills in `null` for columns appearing in only one of the operand tables. The following example illustrates table concatenation:

```
  #table({"A","B"}, {{1,2}})
& #table({"B","C"}, {{3,4}})
```

A	B	C
`1`	`2`	`null`
`null`	`3`	`4`

The native type of table values is a custom table type (derived from the intrinsic type `table`) that lists the column names, specifies all column types to be any, and has no keys. (See Table types for details on table types.)

Function

A *function value* is a value that maps a set of arguments to a single value. The details of *function* values are described in Functions.

Type

A *type value* is a value that classifies other values. The details of *type* values are described in Types.

Types

12/11/2020 • 12 minutes to read

A *type value* is a value that *classifies* other values. A value that is classified by a type is said to *conform* to that type. The M type system consists of the following kinds of types:

- Primitive types, which classify primitive values (`binary` , `date` , `datetime` , `datetimezone` , `duration` , `list` , `logical` , `null` , `number` , `record` , `text` , `time` , `type`) and also include a number of abstract types (`function` , `table` , `any` , and `none`)
- Record types, which classify record values based on field names and value types
- List types, which classify lists using a single item base type
- Function types, which classify function values based on the types of their parameters and return values
- Table types, which classify table values based on column names, column types, and keys
- Nullable types, which classifies the value null in addition to all the values classified by a base type
- Type types, which classify values that are types

The set of *primitive types* includes the types of primitive values a number of *abstract types*, types that do not uniquely classify any values: `function` , `table` , `any` , and `none` . All function values conform to the abstract type `function` , all table values to the abstract type `table` , all values to the abstract type `any` , and no values to the abstract type `none` . An expression of type `none` must raise an error or fail to terminate since no value could be produced that conforms to type `none` . Note that the primitive types `function` and `table` are abstract because no function or table is directly of those types, respectively. The primitive types `record` and `list` are non-abstract because they represent an open record with no defined fields and a list of type any, respectively.

All types that are not members of the closed set of primitive types are collectively referred to as *custom types*. Custom types can be written using a `type-expression` :

type-expression:
    primary-expression
    `type` *primary-type*
type:
    parenthesized-expression
    primary-type
primary-type:
    primitive-type
    record-type
    list-type
    function-type
    table-type
    nullable-type
primitive-type: one of
    `any binary date datetime datetimezone duration function list logical`
    `none null number record table text time type`

The *primitive-type* names are *contextual keywords* recognized only in a *type* context. The use of parentheses in a *type* context moves the grammar back to a regular expression context, requiring the use of the type keyword to move back into a type context. For example, to invoke a function in a *type* context, parentheses can be used:

```
type nullable ( Type.ForList({type number}) )
// type nullable {number}
```

Parentheses can also be used to access a variable whose name collides with a *primitive-type* name:

```
let  record = type [ A = any ]  in  type {(record)}
// type {[ A = any ]}
```

The following example defines a type that classifies a list of numbers:

```
type { number }
```

Similarly, the following example defines a custom type that classifies records with mandatory fields named `X` and `Y` whose values are numbers:

```
type [ X = number, Y = number ]
```

The ascribed type of a value is obtained using the standard library function `Value.Type`, as shown in the following examples:

```
Value.Type( 2 )                // type number
Value.Type( {2} )              // type list
Value.Type( [ X = 1, Y = 2 ] )  // type record
```

The `is` operator is used to determine whether a value's type is compatible with a given type, as shown in the following examples:

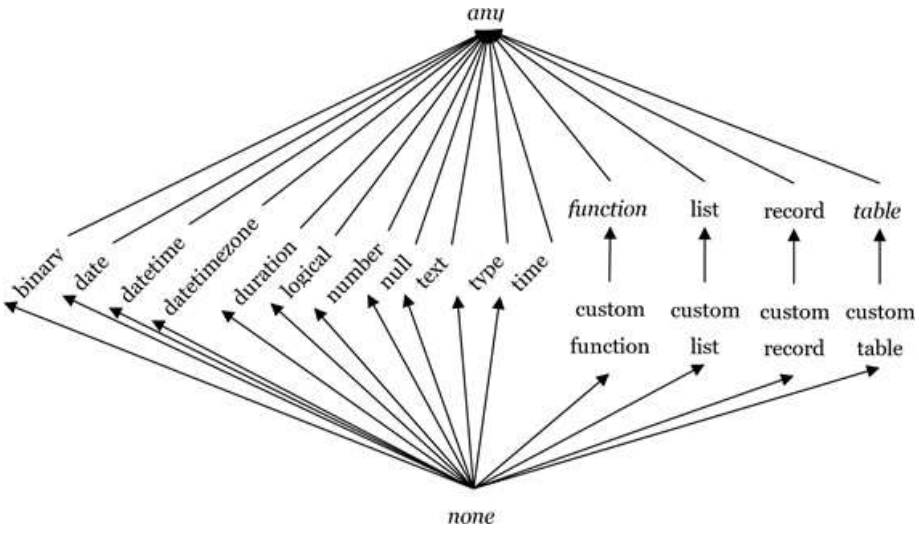
```
1 is number          // true
1 is text            // false
{2} is list          // true
```

The `as` operator checks if the value is compatible with the given type, and raises an error if it is not. Otherwise, it returns the original value.

```
Value.Type( 1 as number )   // type number
{2} as text                 // error, type mismatch
```

Note that the `is` and `as` operators only accept primitive types as their right operand. M does not provide means to check values for conformance to custom types.

A type `X` is *compatible* with a type `Y` if and only if all values that conform to `X` also conform to `Y`. All types are compatible with type `any` and no types (but `none` itself) are compatible with type `none`. The following graph shows the compatibility relation. (Type compatibility is reflexive and transitive. It forms a lattice with type `any` as the top and type `none` as the bottom value.) The names of abstract types are set in *italics*.



The following operators are defined for type values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal

The native type of type values is the intrinsic type `type`.

Primitive Types

Types in the M language form a disjoint hierarchy rooted at type `any`, which is the type that classifies all values. Any M value conforms to exactly one primitive subtype of `any`. The closed set of primitive types deriving from type `any` are as follows:

- `type null`, which classifies the null value.
- `type logical`, which classifies the values true and false.
- `type number`, which classifies number values.
- `type time`, which classifies time values.
- `type date`, which classifies date values.
- `type datetime`, which classifies datetime values.
- `type datetimezone`, which classifies datetimezone values.
- `type duration`, which classifies duration values.
- `type text`, which classifies text values.
- `type binary`, which classifies binary values.
- `type type`, which classifies type values.
- `type list`, which classifies list values.
- `type record`, which classifies record values.
- `type table`, which classifies table values.
- `type function`, which classifies function values.
- `type anynonnull`, which classifies all values excluding null. The intrinsic type `none` classifies no values.

Any Type

The type `any` is abstract, classifies all values in M, and all types in M are compatible with `any`. Variables of type `any` can be bound to all possible values. Since `any` is abstract, it cannot be ascribed to values—that is, no value is directly of type `any`.

List Types

Any value that is a list conforms to the intrinsic type `list`, which does not place any restrictions on the items within a list value.

list-type:
    `{` *item-type* `}`
item-type:
    type

The result of evaluating a *list-type* is a *list type value* whose base type is `list`.

The following examples illustrate the syntax for declaring homogeneous list types:

```
type { number }        // list of numbers type
     { record }        // list of records type
     {{ text }}        // list of lists of text values
```

A value conforms to a list type if the value is a list and each item in that list value conforms to the list type's item type.

The item type of a list type indicates a bound: all items of a conforming list conform to the item type.

Record Types

Any value that is a record conforms to the intrinsic type record, which does not place any restrictions on the field names or values within a record value. A *record-type value* is used to restrict the set of valid names as well as the types of values that are permitted to be associated with those names.

record-type:
    `[` *open-record-marker* `]`
    `[` *field-specification-list*$_{opt}$ `]`
    `[` *field-specification-list* , *open-record-marker* `]`
field-specification-list:
    field-specification
    field-specification `,` *field-specification-list*
field-specification:
    `optional`$_{opt}$ *field-name field-type-specification*$_{opt}$
field-type-specification:
    `=` *field-type*
field-type:
    type
open-record-marker:
    `...`

The result of evaluating a *record-type* is a type value whose base type is `record`.

The following examples illustrate the syntax for declaring record types:

```
type [ X = number, Y = number]
type [ Name = text, Age = number ]
type [ Title = text, optional Description = text ]
type [ Name = text, ... ]
```

Record types are *closed* by default, meaning that additional fields not present in the *fieldspecification-list* are not allowed to be present in conforming values. Including the *openrecord-marker* in the record type declares the type to be *open*, which permits fields not present in the field specification list. The following two expressions are equivalent:

```
type record   // primitive type classifying all records
type [ ... ]  // custom type classifying all records
```

A value conforms to a record type if the value is a record and each field specification in the record type is satisfied. A field specification is satisfied if any of the following are true:

- A field name matching the specification's identifier exists in the record and the associated value conforms to the specification's type
- The specification is marked as optional and no corresponding field name is found in the record

A conforming value may contain field names not listed in the field specification list if and only if the record type is open.

Function Types

Any function value conforms to the primitive type `function`, which does not place any restrictions on the types of the function's formal parameters or the function's return value. A custom *function-type value* is used to place type restrictions on the signatures of conformant function values.

function-type:
    `function (` *parameter-specification-list*$_{opt}$ `)` *function-return-type*
parameter-specification-list:
    required-parameter-specification-list
    required-parameter-specification-list `,` *optional-parameter-specification-list*
    optional-parameter-specification-list
required-parameter-specification-list:
    required-parameter-specification
    required-parameter-specification `,` *required-parameter-specification-list*
required-parameter-specification:
    parameter-specification
optional-parameter-specification-list:
    optional-parameter-specification
    optional-parameter-specification `,` *optional-parameter-specification-list*
optional-parameter-specification:
    `optional` *parameter-specification*
parameter-specification:
    parameter-name parameter-type
function-return-type:
    assertion
assertion:
    `as` *nullable-primitive-type*

The result of evaluating a *function-type* is a type value whose base type is `function`.

The following examples illustrate the syntax for declaring function types:

```
type function (x as text) as number
type function (y as number, optional z as text) as any
```

A function value conforms to a function type if the return type of the function value is compatible with the function type's return type and each parameter specification of the function type is compatible to the positionally corresponding formal parameter of the function. A parameter specification is compatible with a formal parameter if the specified *parameter-type* type is compatible with the type of the formal parameter and the parameter specification is optional if the formal parameter is optional.

Formal parameter names are ignored for the purposes of determining function type conformance.

Table types

A *table-type value* is used to define the structure of a table value.

table-type:
    `table` *row-type*
row-type:
    `[` *field-specification-list* `]`

The result of evaluating a *table-type* is a type value whose base type is `table`.

The *row type* of a table specifies the column names and column types of the table as a closed record type. So that all table values conform to the type `table`, its row type is type `record` (the empty open record type). Thus, type table is abstract since no table value can have type `table`'s row type (but all table values have a row type that is compatible with type `table`'s row type). The following example shows the construction of a table type:

```
type table [A = text, B = number, C = binary]
// a table type with three columns named A, B, and C
// of column types text, number, and binary, respectively
```

A table-type value also carries the definition of a table value's *keys*. A key is a set of column names. At most one key can be designated as the table's *primary key*. (Within M, table keys have no semantic meaning. However, it is common for external data sources, such as databases or OData feeds, to define keys over tables. Power Query uses key information to improve performance of advanced functionality, such as cross-source join operations.)

The standard library functions `Type.TableKeys`, `Type.AddTableKey`, and `Type.ReplaceTableKeys` can be used to obtain the keys of a table type, add a key to a table type, and replace all keys of a table type, respectively.

```
Type.AddTableKey(tableType, {"A", "B"}, false)
// add a non-primary key that combines values from columns A and B
Type.ReplaceTableKeys(tableType, {})
// returns type value with all keys removed
```

Nullable types

For any `type T`, a nullable variant can be derived by using *nullable-type*:

nullable-type:
    `nullable` *type*

The result is an abstract type that allows values of type *T* or the value `null`.

```
42 is nullable number            // true null is
nullable number                  // true
```

Ascription of `type nullable` *T* reduces to ascription of `type null` or `type` *T*. (Recall that nullable types are abstract and no value can be directly of abstract type.)

```
Value.Type(42 as nullable number)       // type number
Value.Type(null as nullable number)     // type null
```

The standard library functions `Type.IsNullable` and `Type.NonNullable` can be used to test a type for nullability and to remove nullability from a type.

The following hold (for any `type T`):

- `type T` is compatible with `type nullable T`
- `Type.NonNullable(type T)` is compatible with `type T`

The following are pairwise equivalent (for any `type T`):

`type nullable any`
`any`

`Type.NonNullable(type any)`
`type anynonnull`

`type nullable none`
`type null`

`Type.NonNullable(type null)`
`type none`

`type nullable nullable T`
`type nullable T`

`Type.NonNullable(Type.NonNullable(type T))`
`Type.NonNullable(type T)`

`Type.NonNullable(type nullable T)`
`Type.NonNullable(type T)`

`type nullable (Type.NonNullable(type T))`
`type nullable T`

Ascribed type of a value

A value's *ascribed type* is the type to which a value is *declared* to conform. When a value is ascribed a type, only a limited conformance check occurs. *M does not perform conformance checking beyond a nullable primitive type. M program authors that choose to ascribe values with type definitions more complex than a nullable primitive-type must ensure that such values conform to these types.*

A value may be ascribed a type using the library function `Value.ReplaceType`. The function either returns a new value with the type ascribed or raises an error if the new type is incompatible with the value's native primitive type. In particular, the function raises an error when an attempt is made to ascribe an abstract type, such as `any`.

Library functions may choose to compute and ascribe complex types to results based on the ascribed types of the input values.

The ascribed type of a value may be obtained using the library function `Value.Type`. For example:

```
Value.Type( Value.ReplaceType( {1}, type {number} )
// type {number}
```

Type equivalence and compatibility

Type equivalence is not defined in M. Any two type values that are compared for equality may or may not return `true`. However, the relation between those two types (whether `true` or `false`) will always be the same.

Compatibility between a given type and a nullable primitive type can be determined using the library function `Type.Is`, which accepts an arbitrary type value as its first and a nullable primitive type value as its second argument:

```
Type.Is(type text, type nullable text)  // true
Type.Is(type nullable text, type text)  // false
Type.Is(type number, type text)         // false
Type.Is(type [a=any], type record)      // true
Type.Is(type [a=any], type list)        // false
```

There is no support in M for determining compatibility of a given type with a custom type.

The standard library does include a collection of functions to extract the defining characteristics from a custom type, so specific compatibility tests can be implemented as M expressions. Below are some examples; consult the M library specification for full details.

```
Type.ListItem( type {number} )
  // type number
Type.NonNullable( type nullable text )
  // type text
Type.RecordFields( type [A=text, B=time] )
  // [ A = [Type = type text, Optional = false],
  //   B = [Type = type time, Optional = false] ]
Type.TableRow( type table [X=number, Y=date] )
  // type [X = number, Y = date]
Type.FunctionParameters(
        type function (x as number, optional y as text) as number)
  // [ x = type number, y = type nullable text ]
Type.FunctionRequiredParameters(
        type function (x as number, optional y as text) as number)
  // 1
Type.FunctionReturn(
        type function (x as number, optional y as text) as number)
  // type number
```

Operator behavior

3/15/2021 • 36 minutes to read

This section defines the behavior of the various M operators.

Operator precedence

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive-expression* consists of a sequence of *multiplicative-expression's* separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*` and `/` operators.

The *parenthesized-expression* production can be used to change the default precedence ordering.

parenthesized-expression:
`(` *expression* `)`

For example:

```
1 + 2 * 3       // 7
(1 + 2) * 3     // 9
```

The following table summarizes the M operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

CATEGORY	EXPRESSION	DESCRIPTION
Primary	*i* @*i*	Identifier expression
	(*x*)	Parenthesized expression
	x[*i*]	Lookup
	x{*y*}	Item access
	x(...)	Function invocation
	{*x*, *y*, ...}	List initialization
	[*i* = *x*, ...]	Record initialization
	...	Not implemented
Unary	+*x*	Identity
	-*x*	Negation

	`not` *x*	Logical negation
Metadata	*x* `meta` *y*	Associate metadata
Multiplicative	*x* * *y*	Multiplication
	x / *y*	Division
Additive	*x* + *y*	Addition
	x - *y*	Subtraction
Relational	*x* <> *y*	Less than
	x > *y*	Greater than
	x <= *y*	Less than or equal
	x >= *y*	Greater than or equal
Equality	*x* = *y*	Equal
	x <> *y*	Not equal
Type assertion	*x* `as` *y*	Is compatible nullable-primitive type or error
Type conformance	*x* `is` *y*	Test if compatible nullable-primitive type
Logical AND	*x* `and` *y*	Short-circuiting conjunction
Logical OR	*x* `or` *y*	Short-circuiting disjunction

Operators and metadata

Every value has an associated record value that can carry additional information about the value. This record is referred to as the *metadata record* for a value. A metadata record can be associated with any kind of value, even `null`. The result of such an association is a new value with the given metadata.

A metadata record is just a regular record and can contain any fields and values that a regular record can, and itself has a metadata record. Associating a metadata record with a value is "non-intrusive". It does not change the value's behavior in evaluations except for those that explicitly inspect metadata records.

Every value has a default metadata record, even if one has not been specified. The default metadata record is empty. The following examples show accessing the metadata record of a text value using the `Value.Metadata` standard library function:

```
Value.Metadata( "Mozart" )   // []
```

Metadata records are generally *not preserved* when a value is used with an operator or function that constructs a new value. For example, if two text values are concatenated using the `&` operator, the metadata of the

resulting text value is the empty record `[]`. The following expressions are equivalent:

```
"Amadeus " & ("Mozart" meta [ Rating = 5 ])
"Amadeus " & "Mozart"
```

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata).

The only operator that returns results that carry metadata is the meta operator.

Structurally recursive operators

Values can be *cyclic*. For example:

```
let l = {0, @l} in l
// {0, {0, {0, ... }}}
[A={B}, B={A}]
// [A = {{ ... }}, B = {{ ... }}]
```

M handles cyclic values by keeping construction of records, lists, and tables lazy. An attempt to construct a cyclic value that does not benefit from interjected lazy structured values yields an error:

```
[A=B, B=A]
// [A = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation"),
//  B = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation"),
// ]
```

Some operators in M are defined by structural recursion. For instance, equality of records and lists is defined by the conjoined equality of corresponding record fields and item lists, respectively.

For non-cyclic values, applying structural recursion yields a *finite expansion* of the value: shared nested values will be traversed repeatedly, but the process of recursion always terminates.

A cyclic value has an *infinite expansion* when applying structural recursion. The semantics of M makes no special accommodations for such infinite expansions—an attempt to compare cyclic values for equality, for instance, will typically run out of resources and terminate exceptionally.

Selection and Projection Operators

The selection and projection operators allow data to be extracted from list and record values.

Item Access

A value may be selected from a list or table based on its zero-based position within that list or table using an *item-access-expression*.

item-access-expression:
    item-selection
    optional-item-selection
item-selection:
    primary-expression `{` *item-selector* `}`
optional-item-selection:
    primary-expression `{` *item-selector* `} ?`

item-selector:
    expression

The *item-access-expression* `x{y}` returns:

- For a list `x` and a number `y`, the item of list `x` at position `y`. The first item of a list is considered to have an ordinal index of zero. If the requested position does not exist in the list, an error is raised.
- For a table `x` and a number `y`, the row of table `x` at position `y`. The first row of a table is considered to have an ordinal index of zero. If the requested position does not exist in the table, an error is raised.
- For a table `x` and a record `y`, the row of table `x` that matches the field values of record `y` for fields with field names that match corresponding table-column names. If there is no unique matching row in the table, an error is raised.

For example:

```
{"a","b","c"}{0}                      // "a"
{1, [A=2], 3}{1}                      // [A=2]
{true, false}{2}                      // error
#table({"A","B"},{{0,1},{2,1}}){0}    // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}  // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}  // error
#table({"A","B"},{{0,1},{2,1}}){[B=1]}  // error
```

The *item-access-expression* also supports the form `x{y}?`, which returns `null` when position (or match) `y` does not exist in list or table `x`. If there are multiple matches for `y`, an error is still raised.

For example:

```
{"a","b","c"}{0}?                     // "a"
{1, [A=2], 3}{1}?                     // [A=2]
{true, false}{2}?                     // null
#table({"A","B"},{{0,1},{2,1}}){0}    // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}  // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}  // null
#table({"A","B"},{{0,1},{2,1}}){[B=1]}  // error
```

Item access does not force the evaluation of list or table items other than the one being accessed. For example:

```
{ error "a", 1, error "c"}{1}  // 1
{ error "a", error "b"}{1}     // error "b"
```

The following holds when the item access operator `x{y}` is evaluated:

- Errors raised during the evaluation of expressions `x` or `y` are propagated.
- The expression `x` produces a list or a table value.
- The expression `y` produces a number value or, if `x` produces a table value, a record value.
- If `y` produces a number value and the value of `y` is negative, an error with reason code `"Expression.Error"` is raised.
- If `y` produces a number value and the value of `y` is greater than or equal to the count of `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.

- If `x` produces a table value and `y` produces a record value and there are no matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.
- If `x` produces a table value and `y` produces a record value and there are multiple matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised.

No items in `x` other than that at position `y` is evaluated during the process of item selection. (For streaming lists or tables, the items or rows preceding that at position `y` are skipped over, which may cause their evaluation, depending on the source of the list or table.)

Field Access

The *field-access-expression* is used to *select* a value from a record or to *project* a record or table to one with fewer fields or columns, respectively.

field-access-expression:
    field-selection
    implicit-target-field-selection
    projection
    implicit-target-projection
field-selection:
    primary-expression field-selector
field-selector:
    required-field-selector
    optional-field-selector
required-field-selector:
    `[` *field-name* `]`
optional-field-selector:
    `[` *field-name* `] ?`
field-name:
    generalized-identifier
    quoted-identifier
implicit-target-field-selection:
    field-selector
projection:
    primary-expression required-projection
    primary-expression optional-projection
required-projection:
    `[` *required-selector-list* `]`
optional-projection:
    `[` *required-selector-list* `] ?`
required-selector-list:
    required-field-selector
    required-selector-list `,` *required-field-selector*
implicit-target-projection:
    required-projection
    optional-projection

The simplest form of field access is *required field selection*. It uses the operator `x[y]` to look up a field in a record by field name. If the field `y` does not exist in `x`, an error is raised. The form `x[y]?` is used to perform *optional* field selection, and returns `null` if the requested field does not exist in the record.

For example:

```
[A=1,B=2][B]       // 2
[A=1,B=2][C]       // error
[A=1,B=2][C]?      // null
```

Collective access of multiple fields is supported by the operators for *required record projection* and *optional record projection*. The operator `x[[y1],[y2],...]` projects the record to a new record with fewer fields (selected by `y1`, `y2`, `...`). If a selected field does not exist, an error is raised. The operator `x[[y1],[y2],...]` projects the record to a new record with the fields selected by `y1`, `y2`, `...`; if a field is missing, `null` is used instead. For example:

```
[A=1,B=2][[B]]          // [B=2]
[A=1,B=2][[C]]          // error
[A=1,B=2][[B],[C]]?     // [B=2,C=null]
```

The forms `[y]` and `[y]?` are supported as a *shorthand* reference to the identifier `_` (underscore). The following two expressions are equivalent:

```
[A]
_[A]
```

The following example illustrates the shorthand form of field access:

```
let _ = [A=1,B=2] in [A] //1
```

The form `[[y1],[y2],...]` and `[[y1],[y2],...]?` are also supported as a shorthand and the following two expressions are likewise equivalent:

```
[[A],[B]]
_[[A],[B]]
```

The shorthand form is particularly useful in combination with the `each` shorthand, a way to introduce a function of a single parameter named `_` (for details, see Simplified declarations. Together, the two shorthands simplify common higher-order functional expressions:

```
List.Select( {[a=1, b=1], [a=2, b=4]}, each [a] = [b])
// {[a=1, b=1]}
```

The above expression is equivalent to the following more cryptic looking longhand:

```
List.Select( {[a=1, b=1], [a=2, b=4]}, (_) => _[a] = _[b])
// {[a=1, b=1]}
```

Field access does not force the evaluation of fields other than the one(s) being accessed. For example:

```
[A=error "a", B=1, C=error "c"][B]  // 1
[A=error "a", B=error "b"][B]       // error "b"
```

The following holds when a field access operator `x[y]`, `x[y]?`, `x[[y]]`, or `x[[y]]?` is evaluated:

- Errors raised during the evaluation of expression `x` are propagated.

- Errors raised when evaluating field `y` are permanently associated with field `y`, then propagated. Any future access to field `y` will raise the identical error.
- The expression `x` produces a record or table value, or an error is raised.
- If the identifier `y` names a field that does not exist in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `...?` is used, in which case the value `null` is returned.

No fields of `x` other than that named by `y` is evaluated during the process of field access.

Metadata operator

The metadata record for a value is amended using the *meta operator* (`x meta y`).

metadata-expression:
unary-expression
unary-expression `meta` *unary-expression*

The following example constructs a text value with a metadata record using the `meta` operator and then accesses the metadata record of the resulting value using `Value.Metadata`:

```
Value.Metadata( "Mozart" meta [ Rating = 5 ] )
// [Rating = 5 ]
Value.Metadata( "Mozart" meta [ Rating = 5 ] )[Rating]
// 5
```

The following holds when applying the metadata combining operator `x meta y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The `y` expression must be a record, or an error with reason code `"Expression.Error"` is raised.
- The resulting metadata record is `x`'s metadata record merged with `y`. (For the semantics of record merge, see Record merge.)
- The resulting value is the value from the `x` expression, without its metadata, with the newly computed metadata record attached.

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata). The following expressions are equivalent:

```
x meta y
Value.ReplaceMetadata(x, Value.Metadata(x) & y)
Value.RemoveMetadata(x) meta (Value.Metadata(x) & y)
```

Equality operators

The *equality operator* `=` is used to determine if two values are the equal. The *inequality operator* `<>` is used to determine if two values are not equal.

equality-expression:
relational-expression
relational-expression `=` *equality-expression*
relational-expression `<>` *equality-expression*

For example:

```
1 = 1           // true
1 = 2           // false
1 <> 1          // false
1 <> 2          // true
null = true     // false
null = null     // true
```

Metadata is not part of equality or inequality comparison. For example:

```
(1 meta [ a = 1 ]) = (1 meta [ a = 2 ]) // true
(1 meta [ a = 1 ]) = 1                  // true
```

The following holds when applying the equality operators `x = y` and `x <> y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The `=` operator has a result of `true` if the values are equal, and `false` otherwise.
- The `<>` operator has a result of `false` if the values are equal, and `true` otherwise.
- Metadata records are not included in the comparison.
- If values produced by evaluating the `x` and `y` expressions are not the same kind of value, then the values are not equal.
- If the values produced by evaluating the `x` and `y` expression are the same kind of value, then there are specific rules for determining if they are equal, as defined below.
- The following is always true:

```
    (x = y) = not (x <> y)
```

The equality operators are defined for the following types:

- The `null` value is only equal to itself.

```
    null = null    // true
    null = true    // false
    null = false   // false
```

- The logical values `true` and `false` are only equal to themselves. For example:

```
    true = true      // true
    false = false    // true
    true = false     // false
    true = 1         // false
```

- Numbers are compared using the specified precision:
  - If either number is `#nan`, then the numbers are not the same.
  - When neither number is `#nan`, then the numbers are compared using a bit-wise comparison of the numeric value.
  - `#nan` is the only value that is not equal to itself.

For example:

```
1 = 1,              // true
1.0 = 1             // true
2 = 1               // false
#nan = #nan         // false
#nan <> #nan        // true
```

- Two durations are equal if they represent the same number of 100-nanosecond ticks.
- Two times are equal if the magnitudes of their parts (hour, minute, second) are equal.
- Two dates are equal if the magnitudes of their parts (year, month, day) are equal.
- Two datetimes are equal if the magnitudes of their parts (year, month, day, hour, minute, second) are equal.
- Two datetimezones are equal if the corresponding UTC datetimes are equal. To arrive at the corresponding UTC datetime, the hours/minutes offset is subtracted from the datetime component of the datetimezone.
- Two text values are equal if using an ordinal, case-sensitive, culture-insensitive comparison they have the same length and equal characters at corresponding positions.
- Two list values are equal if all of the following are true:
  - Both lists contain the same number of items.
  - The values of each positionally corresponding item in the lists are equal. This means that not only do the lists need to contain equal items, the items need to be in the same order.

    For example:

```
{1, 2} = {1, 2}     // true
{2, 1} = {1, 2}     // false
{1, 2, 3} = {1, 2}  // false
```

- Two records are equal if all of the following are true:
  - The number of fields is the same.
  - Each field name of one record is also present in the other record.
  - The value of each field of one record is equal to the like-named field in the other record.

    For example:

```
[ A = 1, B = 2 ] = [ A = 1, B = 2 ]        // true
[ B = 2, A = 1 ] = [ A = 1, B = 2 ]        // true
[ A = 1, B = 2, C = 3 ] = [ A = 1, B = 2 ] // false
[ A = 1 ] = [ A = 1, B = 2 ]               // false
```

- Two tables are equal if all of the following are true:
  - The number of columns is the same.
  - Each column name in one table is also present in the other table.
  - The number of rows is the same.

- Each row has equal values in corresponding cells.

  For example:

```
#table({"A","B"},{{1,2}}) = #table({"A","B"},{{1,2}}) // true
#table({"A","B"},{{1,2}}) = #table({"X","Y"},{{1,2}}) // false
#table({"A","B"},{{1,2}}) = #table({"B","A"},{{2,1}}) // true
```

- A function value is equal to itself, but may or may not be equal to another function value. If two function values are considered equal, then they will behave identically when invoked.

  Two given function values will always have the same equality relationship.

- A type value is equal to itself, but may or may not be equal to another type value. If two type values are considered equal, then they will behave identically when queried for conformance.

  Two given type values will always have the same equality relationship.

Relational operators

The `<`, `>`, `<=`, and `>=` operators are called the *relational operators*.

relational-expression:
additive-expression
additive-expression `<` *relational-expression*
additive-expression `>` *relational-expression*
additive-expression `<=` _relational-expression
additive-expression `>=` *relational-expression*

These operators are used to determine the relative ordering relationship between two values, as shown in the following table:

OPERATION	RESULT
`x < y`	`true` if `x` is less than `y`, `false` otherwise
`x > y`	`true` if `x` is greater than `y`, `false` otherwise
`x <= y`	`true` if `x` is less than or equal to `y`, `false` otherwise
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

For example:

```
0 <= 1           // true
null < 1         // null
null <= null     // null
"ab" < "abc"     // true
#nan >= #nan     // false
#nan <= #nan     // false
```

The following holds when evaluating an expression containing the relational operators:

- Errors raised when evaluating the `x` or `y` operand expressions are propagated.

- The values produced by evaluating both the `x` and `y` expressions must be a number, date, datetime, datetimezone, duration, logical, null or time value. Otherwise, an error with reason code `"Expression.Error"` is raised.
- If either or both operands are `null`, the result is the `null` value.
- If both operands are logical, the value `true` is considered to be greater than `false`.
- If both operands are durations, then the values are compared according to the total number of 100-nanosecond ticks they represent.
- Two times are compared by comparing their hour parts and, if equal, their minute parts and, if equal, their second parts.
- Two dates are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts.
- Two datetimes are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts and, if equal, their hour parts and, if equal, their minute parts and, if equal, their second parts.
- Two datetimezones are compared by normalizing them to UTC by subtracting their hour/minute offset and then comparing their datetime components.
- Two numbers `x` and `y` are compared according to the rules of the IEEE 754 standard:
  - If either operand is `#nan`, the result is `false` for all relational operators.
  - When neither operand is `#nan`, the operators compare the values of the two floatingpoint operands with respect to the ordering `-∞ < -max < ... < -min < -0.0 = +0.0 < +min < ... < +max < +∞` where min and max are the smallest and largest positive finite values that can be represented. The M names for -° and +° are `-#infinity` and `#infinity`.

    Notable effects of this ordering are:

    - Negative and positive zeros are considered equal.
    - A `-#infinity` value is considered less than all other number values, but equal to another `-#infinity`.
    - A `#infinity` value is considered greater than all other number values, but equal to another `#infinity`.

Conditional logical operators

The `and` and `or` operators are called the conditional logical operators.

logical-or-expression:
    logical-and-expression
    logical-and-expression `or` *logical-or-expression*
logical-and-expression:
    is-expression
    is-expression `and` *logical-and-expression*

The `or` operator returns `true` when at least one of its operands is `true`. The right operand is evaluated if and only if the left operand is not `true`.

The `and` operator returns `false` when at least one of its operands is `false`. The right operand is evaluated if

and only if the left operand is not `false`.

Truth tables for the `or` and `and` operators are shown below, with the result of evaluating the left operand expression on the vertical axis and the result of evaluating the right operand expression on the horizontal axis.

AND	TRUE	FALSE	NULL	ERROR
true	true	false	null	error
false	false	false	false	false
null	null	false	null	error
error	error	error	error	error

OR	TRUE	FALSE	NULL	ERROR
or	true	false	null	error
true	true	true	true	true
false	true	false	null	error
null	true	null	null	error
error	error	error	error	error

The following holds when evaluating an expression containing conditional logical operators:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The conditional logical operators are defined over the types `logical` and `null`. If the operand values are not of those types, an error with reason code `"Expression.Error"` is raised.
- The result is a logical value.
- In the expression `x` or `y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `true`.
- In the expression `x` and `y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `false`.

The last two properties give the conditional logical operators their "conditional" qualification; properties also referred to as "short-circuiting". These properties are useful to write compact *guarded predicates*. For example, the following expressions are equivalent:

```
d <> 0 and n/d > 1 if d <> 0 then n/d > 1 else false
```

Arithmetic Operators

The `+`, `-`, `*` and `/` operators are the *arithmetic operators*.

additive-expression:
    multiplicative-expression
    additive-expression `+` *multiplicative-expression*
    additive-expression `-` *multiplicative-expression*
multiplicative-expression:
    metadata- expression
    multiplicative-expression `*` *metadata-expression*
    multiplicative-expression `/` *metadata-expression*

Precision

Numbers in M are stored using a variety of representations to retain as much information as possible about numbers coming from a variety of sources. Numbers are only converted from one representation to another as needed by operators applied to them. Two precisions are supported in M:

PRECISION	SEMANTICS
`Precision.Decimal`	128-bit decimal representation with a range of ±1.0 x 10-28 to ±7.9 x 1028 and 28-29 significant digits.
`Precision.Double`	Scientific representation using mantissa and exponent; conforms to the 64-bit binary double-precision IEEE 754 arithmetic standard IEEE 754-2008.

Arithmetic operations are performed by choosing a precision, converting both operands to that precision (if necessary), then performing the actual operation, and finally returning a number in the chosen precision.

The built-in arithmetic operators (`+` , `-` , `*` , `/`) use Double Precision. Standard library functions (`Value.Add` , `Value.Subtract` , `Value.Multiply` , `Value.Divide`) can be used to request these operations using a specific precision model.

- No numeric overflow is possible: `#infinity` or `-#infinity` represent values of magnitudes too large to be represented.
- No numeric underflow is possible: `0` and `-0` represent values of magnitudes too small to be represented.
- The IEEE 754 special value `#nan` (NaN—Not a Number) is used to cover arithmetically invalid cases, such as a division of zero by zero.
- Conversion from Decimal to Double precision is performed by rounding decimal numbers to the nearest equivalent double value.
- Conversion from Double to Decimal precision is performed by rounding double numbers to the nearest equivalent decimal value and, if necessary, overflowing to `#infinity` or `-#infinity` values.

Addition operator

The interpretation of the addition operator (`x + y`) is dependent on the kind of value of the evaluated expressions x and y, as follows:

X	Y	RESULT	INTERPRETATION
`type number`	`type number`	`type number`	Numeric sum
`type number`	`null`	`null`	

x	y	result	interpretation
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	Numeric sum of magnitudes
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` *datetime*	`type duration`	`type` *datetime*	Datetime offset by duration
`type duration`	`type` *datetime*	`type` *datetime*	
`type` *datetime*	`null`	`null`	
`null`	`type` *datetime*	`null`	

In the table, `type` *datetime* stands for any of `type date`, `type datetime`, `type datetimezone`, or `type time`. When adding a duration and a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric sum

The sum of two numbers is computed using the *addition operator*, producing a number.

For example:

```
1 + 1            // 2
#nan + #infinity  // #nan
```

The addition operator `+` over numbers uses Double Precision; the standard library function `Value.Add` can be used to specify Decimal Precision. The following holds when computing a sum of numbers:

- The sum in Double Precision is computed according to the rules of 64-bit binary doubleprecision IEEE 754 arithmetic IEEE 754-2008. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x + y`.

+	y	+0	-0	+∞	-∞	NaN
x	z	x	x	+∞	-∞	NaN
+0	y	+0	+0	+∞	-∞	NaN
-0	y	+0	-0	+∞	-∞	NaN

+	Y	+0	-0	+°	-°	NAN
+°	+°	+°	+°	+°	NaN	NaN
-°	-°	-°	-°	NaN	-°	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Sum of durations

The sum of two durations is the duration representing the sum of the number of 100nanosecond ticks represented by the durations. For example:

```
#duration(2,1,0,15.1) + #duration(0,1,30,45.3)
// #duration(2, 2, 31, 0.4)
```

Datetime offset by duration

A *datetime* `x` and a duration `y` may be added using `x + y` to compute a new *datetime* whose distance from `x` on a linear timeline is exactly the magnitude of `y`. Here, *datetime* stands for any of `Date`, `DateTime`, `DateTimeZone`, or `Time` and a non-null result will be of the same type. The datetime offset by duration may be computed as follows:

- If the datetime's days since epoch value is specified, construct a new datetime with the following information elements:
  - Calculate a new days since epoch equivalent to dividing the magnitude of y by the number of 100-nanosecond ticks in a 24-hour period, truncating the decimal portion of the result, and adding this value to the x's days since epoch.
  - Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
  - Copy x's value for minutes offset from UTC unchanged.
- If the datetime's days since epoch value is unspecified, construct a new datetime with the following information elements specified:
  - Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
  - Copy x's values for days since epoch and minutes offset from UTC unchanged.

The following examples show calculating the absolute temporal sum when the datetime specifies the *days since epoch*:

```
#date(2010,05,20) + #duration(0,8,0,0)
    //#datetime( 2010, 5, 20, 8, 0, 0 )
    //2010-05-20T08:00:00

#date(2010,01,31) + #duration(30,08,0,0)
    //#datetime(2010, 3, 2, 8, 0, 0)
    //2010-03-02T08:00:00

#datetime(2010,05,20,12,00,00,-08) + #duration(0,04,30,00)
    //#datetime(2010, 5, 20, 16, 30, 0, -8, 0)
    //2010-05-20T16:30:00-08:00

#datetime(2010,10,10,0,0,0,0) + #duration(1,0,0,0)
   //#datetime(2010, 10, 11, 0, 0, 0, 0, 0)
   //2010-10-11T00:00:00+00:00
```

The following example shows calculating the datetime offset by duration for a given time:

```
#time(8,0,0) + #duration(30,5,0,0)
   //#time(13, 0, 0)
   //13:00:00
```

Subtraction operator

The interpretation of the subtraction operator (`x - y`) is dependent on the kind of the value of the evaluated expressions `x` and `y`, as follows:

X	Y	RESULT	INTERPRETATION
`type number`	`type number`	`type number`	Numeric difference
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	Numeric difference of magnitudes
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` *datetime*	`type` *datetime*	`type duration`	Duration between datetimes
`type` *datetime*	`type duration`	`type` *datetime*	Datetime offset by negated duration
`type` *datetime*	`null`	`null`	
`null`	`type` *datetime*	`null`	

In the table, `type` *datetime* stands for any of `type date`, `type datetime`, `type datetimezone`, or `type time`. When subtracting a duration from a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric difference

The difference between two numbers is computed using the *subtraction operator*, producing a number. For example:

```
1 - 1               // 0
#nan - #infinity    // #nan
```

The subtraction operator `-` over numbers uses Double Precision; the standard library function `Value.Subtract` can be used to specify Decimal Precision. The following holds when computing a difference of numbers:

- The difference in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic IEEE 754-2008. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x - y`.

-	Y	+0	-0	+°	-°	NAN
x	z	x	x	-°	+°	NaN
+0	-y	+0	+0	-°	+°	NaN
-0	-y	-0	+0	-°	+°	NaN
+°	+°	+°	+°	NaN	+°	NaN
-°	-°	-°	-°	-°	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The difference in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Difference of durations

The difference of two durations is the duration representing the difference between the number of 100-nanosecond ticks represented by each duration. For example:

```
#duration(1,2,30,0) - #duration(0,0,0,30.45)
// #duration(1, 2, 29, 29.55)
```

Datetime offset by negated duration

A *datetime* `x` and a duration `y` may be subtracted using `x - y` to compute a new *datetime*. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The resulting *datetime* has a distance from `x` on a linear timeline that is exactly the magnitude of `y`, in the direction opposite the sign of `y`. Subtracting positive durations yields results that are backwards in time relative to `x`, while subtracting negative values yields results that are forwards in time.

```
#date(2010,05,20) - #duration(00,08,00,00)
   //#datetime(2010, 5, 19, 16, 0, 0)
   //2010-05-19T16:00:00
#date(2010,01,31) - #duration( 30,08,00,00)
   //#datetime(2009, 12, 31, 16, 0, 0)
   //2009-12-31T16:00:00
```

Duration between two datetimes

Two *datetimes* `t` and `u` may be subtracted using `t - u` to compute the duration between them. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The duration produced by subtracting `u` from `t` must yield `t` when added to `u`.

```
#date(2010,01,31) - #date(2010,01,15)
// #duration(16,00,00,00)
// 16.00:00:00

#date(2010,01,15)- #date(2010,01,31)
// #duration(-16,00,00,00)
// -16.00:00:00

#datetime(2010,05,20,16,06,00,-08,00) -
#datetime(2008,12,15,04,19,19,03,00)
// #duration(521,22,46,41)
// 521.22:46:41
```

Subtracting `t - u` when `u > t` results in a negative duration:

```
#time(01,30,00) - #time(08,00,00)
// #duration(0, -6, -30, 0)
```

The following holds when subtracting two *datetimes* using `t - u`:

- u + (t - u) = t

Multiplication operator

The interpretation of the multiplication operator (`x * y`) is dependent on the kind of value of the evaluated expressions x and y, as follows:

X	Y	RESULT	INTERPRETATION
`type number`	`type number`	`type number`	Numeric product
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	Multiple of duration
`type number`	`type duration`	`type duration`	Multiple of duration
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric product

The product of two numbers is computed using the *multiplication operator*, producing a number. For example:

```
2 * 4               // 8
6 * null            // null
#nan * #infinity    // #nan
```

The multiplication operator `*` over numbers uses Double Precision; the standard library function `Value.Multiply` can be used to specify Decimal Precision. The following holds when computing a product of numbers:

- The product in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic IEEE 754-2008. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

*	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+0	-0	+∞	-∞	NaN
-x	-z	+z	-0	+0	-∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	-∞	NaN	NaN	+∞	-∞	NaN
-∞	-∞	+∞	NaN	NaN	-∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The product in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Multiples of durations

The product of a duration and a number is the duration representing the number of 100nanosecond ticks represented by the duration operand times the number operand. For example:

```
#duration(2,1,0,15.1) * 2
// #duration(4, 2, 0, 30.2)
```

Division operator

The interpretation of the division operator (`x / y`) is dependent on the kind of value of the evaluated expressions `x` and `y`, as follows:

X	Y	RESULT	INTERPRETATION
`type number`	`type number`	`type number`	Numeric quotient
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	Fraction of duration
`type duration`	`type duration`	`type duration`	Numeric quotient of durations
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric quotient

The quotient of two numbers is computed using the *division operator*, producing a number. For example:

```
8 / 2              // 4
8 / 0              // #infinity
0 / 0              // #nan
0 / null           // null
#nan / #infinity   // #nan
```

The division operator `/` over numbers uses Double Precision; the standard library function `Value.Divide` can be used to specify Decimal Precision. The following holds when computing a quotient of numbers:

- The quotient in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic IEEE 754-2008. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

/	+Y	-Y	+0	-0	+°	-°	NAN
+X	+z	-z	+°	-°	+0	-0	NaN
-X	-z	+z	-°	+°	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+°	+°	-°	+°	-°	NaN	NaN	NaN

/	+Y	-Y	+0	-0	+°	-°	NAN
-°	-°	+°	-°	+°	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Quotient of durations

The quotient of two durations is the number representing the quotient of the number of 100nanosecond ticks represented by the durations. For example:

```
#duration(2,0,0,0) / #duration(0,1,30,0)
// 32
```

Scaled durations

The quotient of a duration `x` and a number `y` is the duration representing the quotient of the number of 100-nanosecond ticks represented by the duration `x` and the number `y`. For example:

```
#duration(2,0,0,0) / 32
// #duration(0,1,30,0)
```

Structure Combination

The combination operator (`x & y`) is defined over the following kinds of values:

X	Y	RESULT	INTERPRETATION
`type text`	`type text`	`type text`	Concatenation
`type text`	`null`	`null`	
`null`	`type text`	`null`	
`type date`	`type time`	`type datetime`	Merge
`type date`	`null`	`null`	
`null`	`type time`	`null`	
`type list`	`type list`	`type list`	Concatenation
`type record`	`type record`	`type record`	Merge
`type table`	`type table`	`type table`	Concatenation

Concatenation

Two text, two list, or two table values can be concatenated using `x & y`.

The following example illustrates concatenating text values:

```
"AB" & "CDE"     // "ABCDE"
```

The following example illustrates concatenating lists:

```
{1, 2} & {3}     // {1, 2, 3}
```

The following holds when concatenating two values using `x & y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- No error is propagated if an item of either `x` or `y` contains an error.
- The result of concatenating two text values is a text value that contains the value of x immediately followed by y. If either of the operands is null and the other is a text value, the result is null.
- The result of concatenating two lists is a list that contains all the items of `x` followed by all the items of `y`.
- The result of concatenating two tables is a table that has the union of the two operand table's columns. The column ordering of `x` is preserved, followed by the columns only appearing in `y`, preserving their relative ordering. For columns appearing only in one of the operands, `null` is used to fill in cell values for the other operand.

Merge

Record merge

Two records can be merged using `x & y`, producing a record that includes fields from both `x` and `y`.

The following examples illustrate merging records:

```
[ x = 1 ] & [ y = 2 ]               // [ x = 1, y = 2 ]
[ x = 1, y = 2 ] & [ x = 3, z = 4 ] // [ x = 3, y = 2, z = 4 ]
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- If a field appears in both `x` and `y`, the value from `y` is used.
- The order of the fields in the resulting record is that of `x`, followed by fields in `y` that are not part of `x`, in the same order that they appear in `y`.
- Merging records does not cause evaluation of the values.
- No error is raised because a field contains an error.
- The result is a record.

Date-time merge

A date `x` can be merged with a time `y` using `x & y`, producing a datetime that combines the parts from both `x` and `y`.

The following example illustrates merging a date and a time:

```
#date(2013,02,26) & #time(09,17,00)
// #datetime(2013,02,26,09,17,00)
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The result is a datetime.

Unary operators

The `+`, `-`, and `not` operators are unary operators.

unary-expression:
  type-expression
  `+` *unary expression*
  `-` *unary expression*
  `not` *unary expression*

Unary plus operator

The unary plus operator (`+x`) is defined for the following kinds of values:

X	RESULT	INTERPRETATION
`type number`	`type number`	Unary plus
`type duration`	`type duration`	Unary plus
`null`	`null	

For other values, an error with reason code `"Expression.Error"` is raised.

The unary plus operator allows a `+` sign to be applied to a number, datetime, or null value. The result is that same value. For example:

```
+ - 1                // -1
+ + 1                // 1
+ #nan               // #nan
+ #duration(0,1,30,0) // #duration(0,1,30,0)
```

The following holds when evaluating the unary plus operator `+x`:

- Errors raised when evaluating `x` are propagated.
- If the result of evaluating `x` is not a number value, then an error with reason code `"Expression.Error"` is raised.

Unary minus operator

The unary minus operator (`-x`) is defined for the following kinds of values:

X	RESULT	INTERPRETATION
`type number`	`type number`	Negation

x	RESULT	INTERPRETATION
`type duration`	`type duration`	Negation
`null`	`null`	

For other values, an error with reason code `"Expression.Error"` is raised.

The unary minus operator is used to change the sign of a number or duration. For example:

```
- (1 + 1)       // -2
- - 1           // 1
- - - 1         // -1
- #nan          // #nan
- #infinity     // -#infinity
- #duration(1,0,0,0)  // #duration(-1,0,0,0)
- #duration(0,1,30,0) // #duration(0,-1,-30,0)
```

The following holds when evaluating the unary minus operator `-x`:

- Errors raised when evaluating `x` are propagated.
- If the expression is a number, then the result is the number value from expression `x` with its sign changed. If the value is NaN, then the result is also NaN.

Logical negation operator

The logical negation operator (`not`) is defined for the following kinds of values:

x	RESULT	INTERPRETATION
`type logical`	`type logical`	Negation
`null`	`null`	

This operator computes the logical `not` operation on a given logical value. For example:

```
not true             // false
not false            // true
not (true and true)  // false
```

The following holds when evaluating the logical negation operator `not x`:

- Errors raised when evaluating `x` are propagated.
- The value produced from evaluating expression x must be a logical value, or an error with reason code `"Expression.Error"` must be raised. If the value is `true`, the result is `false`. If the operand is `false`, the result is `true`.

The result is a logical value.

Type operators

The operators `is` and `as` are known as the type operators.

Type compatibility operator

The type compatibility operator `x is y` is defined for the following types of values:

X	Y	RESULT
`type any`	*nullable-primitive-type*	`type logical`

The expression `x is y` returns `true` if the ascribed type of `x` is compatible with `y`, and returns `false` if the ascribed type of `x` is incompatible with `y`. `y` must be a *nullable-primitivetype*.

is-expression:
    as-expression
    is-expression `is` *nullable-primitive-type*
nullable-primitive-type:
    `nullable` $_{opt}$ *primitive-type*

Type compatibility, as supported by the `is` operator, is a subset of general type compatibility and is defined using the following rules:

- If `x` is null then it is compatible iff y is a nullable type or the type `any`.
- If `x` is non-null then if it is a compatible if the the primitive type of `x` is the same as `y`.

The following holds when evaluating the expression `x is y`:

- An error raised when evaluating expression `x` is propagated.

Type assertion operator

The type assertion operator `x as y` is defined for the following types of values:

X	Y	RESULT
`type any`	*nullable-primitive-type*	`type any`

The expression `x as y` asserts that the value `x` is compatible with `y` as per the `is` operator. If it is not compatible, an error is raised. `y` must be a *nullable-primitive-type*.

as-expression:
    equality-expression
    as-expression `as` *nullable-primitive-type*

The expression `x as y` is evaluated as follows:

- A type compatibility check `x is y` is performed and the assertion returns `x` unchanged if that test succeeds.
- If the compatibility check fails, an error with reason code `"Expression.Error"` is raised.

Examples:

```
1 as number              // 1
"A" as number            // error
null as nullable number  // null
```

The following holds when evaluating the expression `x as y`:

- An error raised when evaluating expression `x` is propagated.

Let

12/11/2020 • 2 minutes to read

Let expression

A let expression can be used to capture a value from an intermediate calculation in a variable.

let-expression:
    `let` *variable-list* `in` *expression*
variable-list:
    variable
    variable `,` *variable-list*
variable:
    variable-name `=` *expression*
variable-name:
    identifier

The following example shows intermediate results being calculated and stored in variables `x`, `y`, and `z` which are then used in a subsequent calculation `x + y + z`:

```
let     x = 1 + 1,
        y = 2 + 2,
        z = y + 1
in
        x + y + z
```

The result of this expression is:

```
11  // (1 + 1) + (2 + 2) + (2 + 2 + 1)
```

The following holds when evaluating expressions within the *let-expression*:

- The expressions in the variable list define a new scope containing the identifiers from the *variable-list* production and must be present when evaluating the expressions within the *variable-list* productions. Expressions within the *variable-list* may refer to one-another.
- The expressions within the *variable-list* must be evaluated before the expression in the *let-expression* is evaluated.
- Unless the expressions in the *variable-list* are accessed, they must not be evaluated.
- Errors that are raised during the evaluation of the expressions in the *let-expression* are propagated.

A let expression can be seen as syntactic sugar over an implicit record expression. The following expression is equivalent to the example above:

```
[     x = 1 + 1,
      y = 2 + 2,
      z = y + 1,
      result = x + y + z
][result]
```

Conditionals

12/11/2020 • 2 minutes to read

The *if-expression* selects from two expressions based on the value of a logical input value and evaluates only the selected expression.

if-expression:
    `if` *if-condition* `then` *true-expression* `else` *false-expression*
if-condition:
    expression
true-expression:
    expression
false-expression:
    expression

The following are examples of *if-expressions*:

```
if 2 > 1 then 2 else 1          // 2
if 1 = 1 then "yes" else "no"   // "yes"
```

The following holds when evaluating an *if-expression*:

- If the value produced by evaluating the *if-condition* is not a logical value, then an error with reason code `"Expression.Error"` is raised.
- The *true-expression* is only evaluated if the *if-condition* evaluates to the value `true`.
- The *false-expression* is only evaluated if the *if-condition* evaluates to the value `false`.
- The result of the *if-expression* is the value of the *true-expression* if the *if-condition* is `true`, and the value of the *false-expression* if the *if-condition* is `false`.
- Errors raised during the evaluation of the *if-condition*, *true-expression*, or *falseexpression* are propagated.

Functions

12/11/2020 • 6 minutes to read

A *function* is a value that represents a mapping from a set of argument values to a single value. A function is invoked by provided a set of input values (the argument values), and produces a single output value (the return value).

Writing functions

Functions are written using a *function-expression*:

function-expression:
    `(` *parameter-list*$_{opt}$ `)` *function-return-type*$_{opt}$ `=>` *function-body*
function-body:
    expression
parameter-list:
    fixed-parameter-list
    fixed-parameter-list `,` *optional-parameter-list*
    optional-parameter-list
fixed-parameter-list:
    parameter
    parameter `,` *fixed-parameter-list*
parameter:
    parameter-name parameter-type$_{opt}$
parameter-name:
    identifier
parameter-type:
    assertion
function-return-type:
    assertion
assertion:
    `as` *nullable-primiitve-type*
optional-parameter-list:
    optional-parameter
    optional-parameter `,` *optional-parameter-list*
optional-parameter:
    `optional` *parameter*
nullable-primitve-type
    `nullable`$_{opt}$ primitive-type_

The following is an example of a function that requires exactly two values `x` and `y`, and produces the result of applying the `+` operator to those values. The `x` and `y` are *parameters* that are part of the *formal-parameter-list* of the function, and the `x + y` is the *function body*.

```
(x, y) => x + y
```

The result of evaluating a *function-expression* is to produce a function value (not to evaluate the *function-body*). As a convention in this document, function values (as opposed to function expressions) are shown with the *formal-parameter-list* but with an ellipsis (`...`) instead of the *function-body*. For example, once the function

expression above has been evaluated, it would be shown as the following function value:

```
(x, y) => ...
```

The following operators are defined for function values:

OPERATOR	RESULT
`x = y`	Equal
`x <> y`	Not equal

The native type of function values is a custom function type (derived from the intrinsic type `function`) that lists the parameter names and specifies all parameter types and the return type to be `any`. (See Function types for details on function types.)

Invoking functions

The *function-body* of a function is executed by *invoking* the function value using an *invokeexpression*. Invoking a function value means the *function-body* of the function value is evaluated and a value is returned or an error is raised.

invoke-expression:
    primary-expression `(` *argument-list*$_{opt}$ `)`
argument-list:
    expression-list

Each time a function value is invoked, a set of values are specified as an *argument-list*, called the *arguments* to the function.

An *argument-list* is used to specify a fixed number of arguments directly as a list of expressions. The following example defines a record with a function value in a field, and then invokes the function from another field of the record:

```
[
    MyFunction = (x, y, z) => x + y + z,
    Result1 = MyFunction(1, 2, 3)           // 6
]
```

The following holds when invoking a function:

- The environment used to evaluate the *function-body* of the function includes a variable that corresponds to each parameter, with the same name as the parameter. The value of each parameter corresponds to a value constructed from the *argument-list* of the *invokeexpression*, as defined in Parameters.
- All of the expressions corresponding to the function arguments are evaluated before the *function-body* is evaluated.
- Errors raised when evaluating the expressions in the *expression-list* or *functionexpression* are propagated.
- The number of arguments constructed from the *argument-list* must be compatible with the formal parameters of the function, or an error is raised with reason code `"Expression.Error"`. The process for determining compatibility is defined in Parameters.

Parameters

There are two kinds of formal parameters that may be present in a *formal-parameter-list*:

- A *required* parameter indicates that an argument corresponding to the parameter must always be specified when a function is invoked. Required parameters must be specified first in the *formal-parameter-list*. The function in the following example defines required parameters `x` and `y`:

```
[
    MyFunction = (x, y) => x + y,

    Result1 = MyFunction(1, 1),     // 2
    Result2 = MyFunction(2, 2)      // 4
]
```

- An *optional* parameter indicates that an argument corresponding to the parameter may be specified when a function is invoked, but is not required to be specified. If an argument that corresponds to an optional parameter is not specified when the function is invoked, then the value `null` is used instead. Optional parameters must appear after any required parameters in a *formal-parameter-list*. The function in the following example defines a fixed parameter `x` and an optional parameter `y`:

```
[
    MyFunction = fn(x, optional y) =>
                    if (y = null) x else x + y,
    Result1 = MyFunction(1),        // 1
    Result2 = MyFunction(1, null),  // 1
    Result3 = MyFunction(2, 2),     // 4
]
```

The number of arguments that are specified when a function is invoked must be compatible with the formal parameter list. Compatibility of a set of arguments `A` for a function `F` is computed as follows:

- Let the value *N* represent the number of arguments `A` constructed from the *argumentlist*. For example:

```
MyFunction()             // N = 0
MyFunction(1)            // N = 1
MyFunction(null)         // N = 1
MyFunction(null, 2)      // N = 2
MyFunction(1, 2, 3)      // N = 3
MyFunction(1, 2, null)   // N = 3
MyFunction(1, 2, {3, 4}) // N = 3
```

- Let the value *Required* represent the number of fixed parameters of `F` and *Optional* the number of optional parameters of `F`. For example:

```
()              // Required = 0, Optional = 0
(x)             // Required = 1, Optional = 0
(optional x)    // Required = 0, Optional = 1
(x, optional y) // Required = 1, Optional = 1
```

- Arguments `A` are compatible with function `F` if the following are true:
  - (*N >= Fixed*) and (*N <= (Fixed + Optional)*)
  - The argument types are compatible with `F`'s corresponding parameter types
- If the function has a declared return type, then the result value of the body of function `F` is compatible

with `F`'s return type if the following is true:

  - The value yielded by evaluating the function body with the supplied arguments for the function parameters has a type that is compatible with the return type.
- If the function body yields a value incompatible with the function's return type, an error with reason code `"Expression.Error"` is raised.

Recursive functions

In order to write a function value that is recursive, it is necessary to use the scoping operator (`@`) to reference the function within its scope. For example, the following record contains a field that defines the `Factorial` function, and another field that invokes it:

```
[
    Factorial = (x) =>
                if x = 0 then 1 else x * @Factorial(x - 1),
    Result = Factorial(3)  // 6
]
```

Similarly, mutually recursive functions can be written as long as each function that needs to be accessed has a name. In the following example, part of the `Factorial` function has been refactored into a second `Factorial2` function.

```
[
    Factorial = (x) => if x = 0 then 1 else Factorial2(x),
    Factorial2 = (x) => x * Factorial(x - 1),
    Result = Factorial(3)     // 6
]
```

Closures

A function can return another function as a value. This function can in turn depend on one or more parameters to the original function. In the following example, the function associated with the field `MyFunction` returns a function that returns the parameter specified to it:

```
[
    MyFunction = (x) => () => x,
    MyFunction1 = MyFunction(1),
    MyFunction2 = MyFunction(2),
    Result = MyFunction1() + MyFunction2()  // 3
]
```

Each time the function is invoked, a new function value will be returned that maintains the value of the parameter so that when it is invoked, the parameter value will be returned.

Functions and environments

In addition to parameters, the *function-body* of a *function-expression* can reference variables that are present in the environment when the function is initialized. For example, the function defined by the field `MyFunction` accesses the field `C` of the enclosing record `A`:

```
[
A =
    [
        MyFunction = () => C,
        C = 1
    ],
B = A[MyFunction]()           // 1
]
```

When `MyFunction` is invoked, it accesses the value of the variable `C`, even though it is being invoked from an environment (`B`) that does not contain a variable `C`.

Simplified declarations

The *each-expression* is a syntactic shorthand for declaring untyped functions taking a single formal parameter named `_` (underscore).

each-expression:
    `each` *each-expression-body*
each-expression-body:
    function-body

Simplified declarations are commonly used to improve the readability of higher-order function invocation.

For example, the following pairs of declarations are semantically equivalent:

```
each _ + 1
(_) => _ + 1
each [A]
(_) => _[A]

Table.SelectRows( aTable, each [Weight] > 12 )
Table.SelectRows( aTable, (_) => _[Weight] > 12 )
```

Error Handling

12/11/2020 • 4 minutes to read

The result of evaluating an M expression produces one of the following outcomes:

- A single value is produced.
- An *error is raised*, indicating the process of evaluating the expression could not produce a value. An error contains a single record value that can be used to provide additional information about what caused the incomplete evaluation.

Errors can be raised from within an expression, and can be handled from within an expression.

Raising errors

The syntax for raising an error is as follows:

error-raising-expression:
`error` *expression*

Text values can be used as shorthand for error values. For example:

```
error "Hello, world" // error with message "Hello, world"
```

Full error values are records and can be constructed using the `Error.Record` function:

```
error Error.Record("FileNotFound", "File my.txt not found",
     "my.txt")
```

The above expression is equivalent to:

```
error [
    Reason = "FileNotFound",
    Message = "File my.txt not found",
    Detail = "my.txt"
]
```

Raising an error will cause the current expression evaluation to stop, and the expression evaluation stack will unwind until one of the following occurs:

- A record field, section member, or let variable—collectively: an *entry*—is reached. The entry is marked as having an error, the error value is saved with that entry, and then propagated. Any subsequent access to that entry will cause an identical error to be raised. Other entries of the record, section, or let expression are not necessarily affected (unless they access an entry previously marked as having an error).
- The top-level expression is reached. In this case, the result of evaluating the top-level expression is an error instead of a value.
- A `try` expression is reached. In this case, the error is captured and returned as a value.

Handling errors

An *error-handling-expression* is used to handle an error:

_error-handling-expression:
    `try` *protected-expression otherwise-clause*$_{opt}$
protected-expression:
    expression
otherwise-clause:
    `otherwise` *default-expression*
default-expression:
    expression

The following holds when evaluating an *error-handling-expression* without an *otherwiseclause*:

- If the evaluation of the protected-expression does not result in an error and produces a value x, the value produced by the error-handling-expression is a record of the following form:

```
    [ HasErrors = false, Value = x ]
```

- If the evaluation of the protected-expression raises an error value e, the result of the error-handling-expression is a record of the following form:

```
    [ HasErrors = true, Error = e ]
```

The following holds when evaluating an *error-handling-expression* with an *otherwiseclause*:

- The protected-expression must be evaluated before the otherwise-clause.
- The otherwise-clause must be evaluated if and only if the evaluation of the protectedexpression raises an error.
- If the evaluation of the *protected-expression* raises an error, the value produced by the *error-handling-expression* is the result of evaluating the otherwise-clause.
- Errors raised during the evaluation of the otherwise-clause are propagated.

The following example illustrates an *error-handling-expression* in a case where no error is raised:

```
let
    x = try "A"
in
    if x[HasError] then x[Error] else x[Value]
// "A"
```

The following example shows raising an error and then handling it:

```
let
    x = try error "A"
in
    if x[HasError] then x[Error] else x[Value]
// [ Reason = "Expression.Error", Message = "A", Detail = null ]
```

An otherwise clause can be used to replace errors handled by a try expression with an alternative value:

```
try error "A" otherwise 1
// 1
```

If the otherwise clause also raises an error, then so does the entire try expression:

```
try error "A" otherwise error "B"
// error with message "B"
```

Errors in record and let initializers

The following example shows a record initializer with a field `A` that raises an error and is accessed by two other fields `B` and `C`. Field `B` does not handle the error that is raised by `A`, but `C` does. The final field `D` does not access `A` and so it is not affected by the error in `A`.

```
[
    A = error "A",
    B = A + 1,
    C = let x =
            try A in
                if not x[HasError] then x[Value]
                else x[Error],
    D = 1 + 1
]
```

The result of evaluating the above expression is:

```
[
    A = // error with message "A"
    B = // error with message "A"
    C = "A",
    D = 2
]
```

Error handling in M should be performed close to the cause of errors to deal with the effects of lazy field initialization and deferred closure evaluations. The following example shows an unsuccessful attempt at handling an error using a `try` expression:

```
let
    f = (x) => [ a = error "bad", b = x ],
    g = try f(42) otherwise 123
in
    g[a]  // error "bad"
```

In this example, the definition `g` was meant to handle the error raised when calling `f`. However, the error is raised by a field initializer that only runs when needed and thus after the record was returned from f and passed through the `try` expression.

Not implemented error

While an expression is being developed, an author may want to leave out the implementation for some parts of the expression, but may still want to be able to execute the expression. One way to handle this case is to raise an error for the unimplemented parts. For example:

```
(x, y) =>
     if x > y then
         x - y
     else
         error Error.Record("Expression.Error",
            "Not Implemented")
```

The ellipsis symbol (`...`) can be used as a shortcut for `error` .

not-implemented-expression:

`...`

For example, the following is equivalent to the previous example:

```
(x, y) => if x > y then x - y else ...
```

Sections

12/11/2020 • 4 minutes to read

A *section-document* is an M program that consists of multiple named expressions.

section-document:
    section
section:
    literal-attributes$_{opt}$ `section` *section-name* `;` *section-members*$_{opt}$
section-name:
    identifier
section-members:
    section-member section-members$_{opt}$
section-member:
    literal-attributes$_{opt}$ `shared`$_{opt}$ *section-member-name* `=` *expression* `;`
section-member-name:
    identifier

In M, a section is an organizational concept that allows related expressions to be named and grouped within a document. Each section has a *section-name*, which identifies the section and qualifies the names of the *section-members* declared within the section. A *sectionmember* consists of a *member-name* and an *expression*. Section member expressions may refer to other section members within the same section directly by member name.

The following example shows a section-document that contains one section:

```
section Section1;

A = 1;                          //1
B = 2;                          //2
C = A + B;                      //3
```

Section member expressions may refer to section members located in other sections by means of a *section-access-expression*, which qualifies a section member name with the name of the containing section.

section-access-expression:
    identifier `!` *identifier*

The following example shows a document containing two sections that are mutually referential:

```
section Section1;
A = "Hello";                    //"Hello"
B = 1 + Section2!A;             //3

section Section2;
A = 2;                          //2
B = Section1!A & " world!";     /"Hello, world"
```

Section members may optionally be declared as `shared`, which omits the requirement to use a *section-access-expression* when referring to shared members outside of the containing section. Shared members in external sections may be referred to by their unqualified member name so long as no member of the same name is declared in the referring section and no other section has a like-named shared member.

The following example illustrates the behavior of shared members when used across sections within the same document:

```
section Section1;
shared A = 1;        // 1

section Section2;
B = A + 2;           // 3 (refers to shared A from Section1)

section Section3;
A = "Hello";         // "Hello"
B = A + " world";    // "Hello world" (refers to local A)
C = Section1!A + 2;  // 3
```

Defining a shared member with the same name in different sections will produce a valid global environment, however accessing the shared member will raise an error when accessed.

```
section Section1;
shared A = 1;

section Section2;
shared A = "Hello";

section Section3;
B = A;    //Error: shared member A has multiple definitions
```

The following holds when evaluating a section-document:

- Each *section-name* must be unique in the global environment.
- Within a section, each *section-member* must have a unique *section-member-name*.
- Shared section members with more than one definition raise an error when the shared member is accessed.
- The expression component of a *section-member* must not be evaluated before the section member is accessed.
- Errors raised while the expression component of a *section-member* is evaluated are associated with that section member before propagating outward and then re-raised each time the section member is accessed.

Document Linking

A set of M section documents can be linked into an opaque record value that has one field per shared member of the section documents. If shared members have ambiguous names, an error is raised.

The resulting record value fully closes over the global environment in which the link process was performed. Such records are, therefore, suitable components to compose M documents from other (linked) sets of M documents. There are no opportunities for naming conflicts.

The standard library functions `Embedded.Value` can be used to retrieve such "embedded" record values that correspond to reused M components.

Document Introspection

M provides programmatic access to the global environment by means of the `#sections` and `#shared` keywords.

#sections

The `#sections` intrinsic variable returns all sections within the global environment as a record. This record is keyed by section name and each value is a record representation of the corresponding section indexed by section member name.

The following example shows a document consisting of two sections and the record produced by evaluating the `#sections` intrinsic variable within the context of that document:

```
section Section1;
A = 1;
B = 2;

section Section2;
C = "Hello";
D = "world";

#sections
//[
//  Section1 = [ A = 1, B = 2],
//  Section2 = [ C = "Hello", D = "world" ]
//]
```

The following holds when evaluating `#sections`:

- The `#sections` intrinsic variable preserves the evaluation state of all section member expressions within the document.
- The `#sections` intrinsic variable does not force the evaluation of any unevaluated section members.

#shared

The `#shared` intrinsic variable returns a record containing the names and values of all shared section members currently in scope.

The following example shows a document with two shared members and the corresponding record produced by evaluating the #shared intrinsic variable within the context of that document:

```
section Section1;
shared A = 1;
B = 2;

Section Section2;
C = "Hello";
shared D = "world";

//[
//  A = 1,
//  D = "world"
//]
```

The following holds when evaluating `#shared`:

- The `#shared` intrinsic variable preserves the evaluation state of all shared member expressions within the document.
- The `#shared` intrinsic variable does not force the evaluation of any unevaluated section members.

Consolidated Grammar

12/11/2020 • 5 minutes to read

Lexical grammar

lexical-unit:
    lexical-elements$_{opt}$
lexical-elements:
    lexical-element lexical-elements$_{opt}$
lexical-element:
    whitespace
    token comment

White space

whitespace:
    Any character with Unicode class Zs
    Horizontal tab character (`U+0009`)
    Vertical tab character (`U+000B`)
    Form feed character (`U+000C`)
    Carriage return character (`U+000D`) followed by line feed character (`U+000A`) *new-line-character*
new-line-character:
    Carriage return character (`U+000D`)
    Line feed character (`U+000A`)
    Next line character (`U+0085`)
    Line separator character (`U+2028`)
    Paragraph separator character (`U+2029`)

Comment

comment:
    single-line-comment
    delimited-comment
single-line-comment:
    `//` *single-line-comment-characters*$_{opt}$
single-line-comment-characters:
    single-line-comment-character single-line-comment-characters$_{opt}$
single-line-comment-character:
    Any Unicode character except a *new-line-character*
delimited-comment:
    `/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`
delimited-comment-text:
    delimited-comment-section delimited-comment-text$_{opt}$
delimited-comment-section:
    `/`
    asterisks$_{opt}$ *not-slash-or-asterisk*
asterisks:
    `*` *asterisks*$_{opt}$
not-slash-or-asterisk:
    Any Unicode character except `*` or `/`

Tokens

token:
    identifier
    keyword
    literal
    operator-or-punctuator

Character escape sequences

character-escape-sequence:
    `#(` *escape-sequence-list* `)`
escape-sequence-list:
    single-escape-sequence
    escape-sequence-list `,` *single-escape-sequence*
single-escape-sequence:
    long-unicode-escape-sequence
    short-unicode-escape-sequence
    control-character-escape-sequence
    escape-escape
long-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
short-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit
control-character-escape-sequence:
    control-character
control-character:
    `cr`
    `lf`
    `tab`
escape-escape:
    `#`

Literals

literal:
    logical-literal
    number-literal
    text-literal
    null-literal
    verbatim-literal
logical-literal:
    `true`
    `false`
number-literal:
    decimal-number-literal
    hexadecimal-number-literal
decimal-digits:
    decimal-digit decimal-digits$_{opt}$
decimal-digit: one of
    `0 1 2 3 4 5 6 7 8 9`
hexadecimal-number-literal:
    `0x` *hex-digits*
    `0X` *hex-digits*
hex-digits:

hex-digit *hex-digits*$_{opt}$

hex-digit: one of

`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

decimal-number-literal:

decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$

`.` *decimal-digits* *exponent-part*$_{opt}$

decimal-digits *exponent-part*$_{opt}$

exponent-part:

`e` *sign*$_{opt}$ *decimal-digits*

`E` *sign*$_{opt}$ *decimal-digits*

sign: one of

`+ -`

text-literal:

`"` *text-literal-characters*$_{opt}$ `"`

text-literal-characters:

text-literal-character *text-literal-characters*$_{opt}$

text-literal-character:

single-text-character

character-escape-sequence

double-quote-escape-sequence

single-text-character:

Any character except `"` (`U+0022`) or `#` (`U+0023`) followed by `(` (`U+0028`)

double-quote-escape-sequence:

`""` (`U+0022`, `U+0022`)

null-literal:

`null`

verbatim-literal:

`#!"` *text-literal-characters*$_{opt}$ `"`

Identifiers

identifier:

regular-identifier

quoted-identifier

regular-identifier:

available-identifier

available-identifier *dot-character* *regular-identifier*

available-identifier:

A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:

letter-character

underscore-character

identifier-start-character *identifier-part-characters*

identifier-start-character:

letter-character

underscore-character

identifier-part-characters:

identifier-part-character *identifier-part-characters*$_{opt}$

identifier-part-character:

letter-character

decimal-digit-character

underscore-character

connecting-character

combining-character
formatting-character

generalized-identifier:
generalized-identifier-part
generalized-identifier separated only by blanks (`U+0020`) *generalized-identifier-part*

generalized-identifier-part:
generalized-identifier-segment
decimal-digit-character generalized-identifier-segment

generalized-identifier-segment:
keyword-or-identifier
keyword-or-identifier dot-character keyword-or-identifier

dot-character:
`.` (`U+002E`)

underscore-character:
`_` (`U+005F`)

letter-character:_
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:
A Unicode character of classes Mn or Mc

decimal-digit-character:
A Unicode character of the class Nd

connecting-character:
A Unicode character of the class Pc

formatting-character:
A Unicode character of the class Cf

quoted-identifier:
`#"` *text-literal-characters*$_{opt}$ `"`

Keywords and predefined identifiers

Predefined identifiers and keywords cannot be redefined. A quoted identifier can be used to handle identifiers that would otherwise collide with predefined identifiers or keywords.

keyword: one of

```
and as each else error false if in is let meta not null or otherwise
section shared then true try type #binary #date #datetime
#datetimezone #duration #infinity #nan #sections #shared #table #time
```

Operators and punctuators

operator-or-punctuator: one of

```
, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ? => .. ...
```

Syntactic grammar

Documents

document:
section-document
expression-document

Section Documents

section-document:
section

section:
literal-attributes$_{opt}$ `section` *section-name* `;` *section-members*$_{opt}$

section-name:
    identifier
section-members:
    section-member section-members$_{opt}$
section-member:
    literal-attributes$_{opt}$ *shared*$_{opt}$ *section-member-name* `=` *expression* `;`
section-member-name:
    identifier

Expression Documents

Expressions

expression-document:
    expression
expression:
    logical-or-expression
    each-expression
    function-expression
    let-expression
    if-expression
    error-raising-expression
    error-handling-expression

Logical expressions

logical-or-expression:
    logical-and-expression
    logical-and-expression `or` *logical-or-expression*
logical-and-expression:
    is-expression
    logical-and-expression `and` *is-expression*

Is expression

is-expression:
    as-expression
    is-expression `is` *nullable-primitive-type*
nullable-primitive-type:
    `nullable`$_{opt}$ *primitive-type*

As expression

as-expression:
    equality-expression
    as-expression `as` *nullable-primitive-type*

Equality expression

equality-expression:
    relational-expression
    relational-expression `=` *equality-expression*
    relational-expression `<>` *equality-expression*

Relational expression

relational-expression:
    additive-expression
    additive-expression `<` *relational-expression*
    additive-expression `>` *relational-expression*
    additive-expression `<=` *relational-expression*
    additive-expression `>=` *relational-expression*

Arithmetic expressions

additive-expression:
- *multiplicative-expression*
- *multiplicative-expression* `+` *additive-expression*
- *multiplicative-expression* `-` *additive-expression*
- *multiplicative-expression* `&` _additive-expression

multiplicative-expression:
- *metadata-expression*
- *metadata-expression* `*` *multiplicative-expression*
- *metadata-expression* `/` *multiplicative-expression*

Metadata expression

metadata-expression:
- *unary-expression*
- *unary-expression* `meta` *unary-expression*

Unary expression

unary-expression:
- *type-expression*
- `+` *unary-expression*
- `-` *unary-expression*
- `not` *unary-expression*

Primary expression

primary-expression:
- *literal-expression*
- *list-expression*
- *record-expression*
- *identifier-expression*
- *section-access-expression*
- *parenthesized-expression*
- *field-access-expression*
- *item-access-expression*
- *invoke-expression*
- *not-implemented-expression*

Literal expression

literal-expression:
- *literal*

Identifier expression

identifier-expression:
- *identifier-reference*

identifier-reference:
- *exclusive-identifier-reference*
- *inclusive-identifier-reference*

exclusive-identifier-reference:
- *identifier*

inclusive-identifier-reference:
- `@` *identifier*

Section-access expression

section-access-expression:
- *identifier* `!` *identifier*

Parenthesized expression

parenthesized-expression:

`(` *expression* `)`

Not-implemented expression

not-implemented-expression:

`...`

Invoke expression

invoke-expression:

primary-expression `(` *argument-list*$_{opt}$ `)`

argument-list:

expression

expression `,` *argument-list*

List expression

list-expression:

`{` *item-list*$_{opt}$ `}`

item-list:

item

item `,` *item-list*

item:

expression

expression `..` *expression*

Record expression

record-expression:

`[` *field-list*$_{opt}$ `]`

field-list:

field

field `,` *field-list*

field:

field-name `=` *expression*

field-name:

generalized-identifier

quoted-identifier

Item access expression

item-access-expression:

item-selection

optional-item-selection

item-selection:

primary-expression `{` *item-selector* `}`

optional-item-selection:

primary-expression `{` *item-selector* `}?`

item-selector:

expression

Field access expressions

field-access-expression:

field-selection

implicit-target-field-selection

projection

implicit-target-projection

field-selection:

primary-expression field-selector

field-selector:
    required-field-selector
    optional-field-selector
required-field-selector:
    `[` *field-name* `]`
optional-field-selector:
    `[` *field-name* `] ?`
field-name:
    generalized-identifier
    quoted-identifier
implicit-target-field-selection:
    field-selector
projection:
    primary-expression required-projection
    primary-expression optional-projection
required-projection:
    `[` *required-selector-list* `]`
optional-projection:
    `[` *required-selector-list* `] ?`
required-selector-list:
    required-field-selector
    required-field-selector `,` *required-selector-list*
implicit-target-projection:
    required-projection
    optional-projection

Function expression

function-expression:
    `(` $parameter\text{-}list_{opt}$ `)` $return\text{-}type_{opt}$ `=>` *function-body*
function-body:
    expression
parameter-list:
    fixed-parameter-list
    fixed-parameter-list `,` *optional-parameter-list*
    optional-parameter-list
fixed-parameter-list:
    parameter
    parameter `,` *fixed-parameter-list*
parameter:
    parameter-name $parameter\text{-}type_{opt}$
parameter-name:
    identifier
parameter-type:
    assertion
return-type:
    assertion
assertion:
    `as` *nullable-primitive-type*
optional-parameter-list:
    optional-parameter
    optional-parameter `,` *optional-parameter-list*
optional-parameter:

`optional` *parameter*

Each expression

each-expression:
`each` *each-expression-body*
each-expression-body:
function-body

Let expression

let-expression:
`let` *variable-list* `in` *expression*
variable-list:
variable
variable `,` *variable-list*
variable:
variable-name `=` *expression*
variable-name:
identifier

If expression

if-expression:
`if` *if-condition* `then` *true-expression* `else` *false-expression*
if-condition:
expression
true-expression:
expression
false-expression:
expression

Type expression

type-expression:
primary-expression
`type` *primary-type*
type:
parenthesized-expression
primary-type
primary-type:
primitive-type
record-type
list-type
function-type
table-type
nullable-type
primitive-type: one of
`any anynonnull binary date datetime datetimezone duration function`
`list logical none null number record table text type`
record-type:
`[` *open-record-marker* `]`
`[` *field-specification-list*$_{opt}$ `]`
`[` *field-specification-list* `,` *open-record-marker* `]`
field-specification-list:
field-specification
field-specification `,` *field-specification-list*
field-specification:

`optional` $_{opt}$ *field-name field-type-specification*$_{opt}$

field-type-specification:

`=` *field-type*

field-type:

type

open-record-marker:

`...`

list-type:

`{` *item-type* `}`

item-type:

type

function-type:

`function (` *parameter-specification-listopt* `)` *return-type*

parameter-specification-list:

required-parameter-specification-list

required-parameter-specification-list `,` *optional-parameter-specification-list*

optional-parameter-specification-list

required-parameter-specification-list:

required-parameter-specification

required-parameter-specification `,` *required-parameter-specification-list*

required-parameter-specification:

parameter-specification

optional-parameter-specification-list:

optional-parameter-specification

optional-parameter-specification `,` *optional-parameter-specification-list*

optional-parameter-specification:

`optional` *parameter-specification*

parameter-specification:

parameter-name parameter-type

table-type:

`table` *row-type*

row-type:

`[` *field-specification-list* `]`

nullable-type:

`nullable` *type*

Error raising expression

error-raising-expression:

`error` expression_

Error handling expression

error-handling-expression:

`try` *protected-expression otherwise-clause*$_{opt}$

protected-expression:

expression

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression

Literal Attributes

literal-attributes:

record-literal

record-literal:

`[` *literal-field-list*$_{opt}$ `]`

literal-field-list:

literal-field

literal-field `,` *literal-field-list*

literal-field:

field-name `=` *any-literal*

list-literal:

`{` *literal-item-list*$_{opt}$ `}`

literal-item-list:

any-literal

any-literal `,` *literal-item-list*

any-literal:

record-literal

list-literal

logical-literal

number-literal

text-literal

null-literal

Types in the Power Query M formula language

3/15/2021 • 8 minutes to read

The Power Query M Formula Language is a useful and expressive data mashup language. But it does have some limitations. For example, there is no strong enforcement of the type system. In some cases, a more rigorous validation is needed. Fortunately, M provides a built-in library with support for types to make stronger validation feasible.

Developers should have a thorough understanding of the type system in-order to do this with any generality. And, while the Power Query M language specification explains the type system well, it does leave a few surprises. For example, validation of function instances requires a way to compare types for compatibility.

By exploring the M type system more carefully, many of these issues can be clarified, and developers will be empowered to craft the solutions they need.

Knowledge of predicate calculus and naïve set theory should be adequate to understand the notation used.

PRELIMINARIES

(1) B := { *true*, *false* }
B is the typical set of Boolean values

(2) N := { valid M identifiers }
N is the set of all valid names in M. This is defined elsewhere.

(3) P := $\square B, T\square$
P is the set of function parameters. Each one is possibly optional, and has a type. Parameter names are irrelevant.

(4) $P^n := \bigcup_{0 \le i \le n} \square i, P^i \square$
P^n is the set of all ordered sequences of n function parameters.

(5) $P^* := \bigcup_{0 \le i \le \circ} P^i$
P^* is the set of all possible sequences of function parameters, from length 0 on up.

(6) $F := \square B, N, T\square$
F is the set of all record fields. Each field is possibly optional, has a name, and a type.

(7) $F^n := \prod_{0 \le i \le n} F$
F^n is the set of all sets of n record fields.

(8) $F^* := \left(\bigcup_{0 \le i \le \circ} F^i \right) \setminus \{ F \mid \square b_1, n_1, t_1 \square, \square b_2, n_2, t_2 \square \in F \wedge n_1 = n_2 \}$
F^* is the set of all sets (of any length) of record fields, except for the sets where more than one field has the same name.

(9) $C := \square N, T \square$
C is the set of column types, for tables. Each column has a name and a type.

(10) $C^n \subset \bigcup_{0 \le i \le n} \square i, C \square$
C^n is the set of all ordered sequences of n column types.

(11) $C^* := \left(\bigcup_{0 \le i \le \circ} C^i \right) \setminus \{ C^m \mid \square a, \square n_1, t_1 \square \square, \square b, \square n_2, t_2 \square \square \in C^m \wedge n_1 = n_2 \}$
C^* is the set of all combinations (of any length) of column types, except for those where more than one column

has the same name.

M TYPES

(12) T_F := ⟦P, P^*⟧
A Function Type consists of a return type, and an ordered list of zero-or-more function parameters.

(13) T_L := ⟦T⟧
A List type is indicated by a given type (called the "item type") wrapped in curly braces. Since curly braces are used in the metalanguage, ⟦ ⟧ *brackets are used in this document.*

(14) T_R := ⟦B, F^*⟧
A Record Type has a flag indicating whether it's "open", and zero-or-more unordered record fields.

(15) T_R^o := ⟦*true*, F⟧

(16) $T_R^•$:= ⟦*false*, F⟧
T_R^o *and* $T_R^•$ *are notational shortcuts for open and closed record types, respectively.*

(17) T_T := C^*
A Table Type is an ordered sequence of zero-or-more column types, where there are no name collisions.

(18) T_P := { any; none; null; logical; number; time; date; datetime; datetimezone; duration; text; binary; type; list; record; table; function; anynonnull }
A Primitive Type is one from this list of M keywords.

(19) T_N := { t_n, u ∈ T | t_n = u+null } = nullable t
Any type can additionally be marked as being nullable, by using the "nullable" *keyword.*

(20) T := T_F ⋃ T_L ⋃ T_R ⋃ T_T ⋃ T_P ⋃ T_N
The set of all M types is the union of these six sets of types:
Function Types, List Types, Record Types, Table Types, Primitive Types, and Nullable Types.

FUNCTIONS

One function needs to be defined: *NonNullable* : $T \leftarrow T$
This function takes a type, and returns a type that is equivalent except it does not conform with the null value.

IDENTITIES

Some identities are needed to define some special cases, and may also help elucidate the above.

(21) nullable any = any
(22) nullable anynonnull = any
(23) nullable null = null
(24) nullable none = null
(25) nullable nullable t ∈ T = nullable t
(26) *NonNullable*(nullable t ∈ T) = *NonNullable*(t)
(27) *NonNullable*(any) = anynonnull

TYPE COMPATIBILITY

As defined elsewhere, an M type is compatable with another M type if and only if all values that conform to the first type also conform to the second type.

Here is defined a compatability relation that does not depend on conforming values, and is based on the properties of the types themselves. It is anticiplated that this relation, as defined in this document, is completely

equivalent to the original semantic definition.

The "is compatible with" relation : $\leq : B \leftarrow T \times T$
In the below section, a lowercase t will always represent an M Type, an element of T.

A Φ will represent a subset of F^*, or of C^*.

(28) $t \leq t$
This relation is reflexive.

(29) $t_a \leq t_b \land t_b \leq t_c \rightarrow t_a \leq t_c$
This relation is transitive.

(30) none $\leq t \leq$ any
M types form a lattice over this relation; none *is the bottom, and* any *is the top.*

(31) $t_a, t_b \in T_N \land t_a \leq t_a \rightarrow NonNullable(t_a) \leq NonNullable(t_b)$
If two types are compatible, then the NonNullable equivalents are also compatible.

(32) null $\leq t \in T_N$
The primitive type null *is compatible with all nullable types.*

(33) $t \notin T_N \leq$ anynonnull
All nonnullable types are compatible with anynonnull.

(34) $NonNullable(t) \leq t$
A NonNullible type is compatible with the nullable equivalent.

(35) $t \in T_F \rightarrow t \leq$ function
All function types are compatible with function.

(36) $t \in T_L \rightarrow t \leq$ list
All list types are compatible with list.

(37) $t \in T_R \rightarrow t \leq$ record
All record types are compatible with record.

(38) $t \in T_T \rightarrow t \leq$ table
All table types are compatible with table.

(39) $t_a \leq t_b \leftrightarrow \llbracket t_a \rrbracket \leq \llbracket t_b \rrbracket$
A list type is compaible with another list type if the item types are compatible, and vice-versa.

(40) $t_a \in T_F = \square\, p_a, p^* \square, t_b \in T_F = \square\, p_b, p^* \square \land p_a \leq p_b \rightarrow t_a \leq t_b$
A function type is compatible with another function type if the return types are compatible, and the parameter lists are identical.

(41) $t_a \in T_R^{\circ}, t_b \in T_R^{\bullet} \rightarrow t_a \not\leq t_b$
An open record type is never compatible with a closed record type.

(42) $t_a \in T_R^{\bullet} = \square false, \Phi \square, t_b \in T_R^{\circ} = \square true, \Phi \square \rightarrow t_a \leq t_b$
A closed record type is compatible with an otherwise identical open record type.

(43) $t_a \in T_R^{\circ} = \square true, (\Phi, \square true, n, \text{any}\square)\square, t_b \in T_R^{\circ} = \square true, \Phi \square \rightarrow t_a \leq t_b \land t_b \leq t_a$
An optional field with the type any *may be ignored when comparing two open record types.*

(44) $t_a \in T_R = \square b, (\Phi, \square \beta, n, u_a \square)\square, t_b \in T_R = \square b, (\Phi, \square \beta, n, u_b \square)\square \land u_a \leq u_b \rightarrow t_a \leq t_b$
Two record types that differ only by one field are compatible if the name and optionality of the field are identical, and the types of said field are compatible.

(45) $t_a \in T_R = \square b, (\Phi, \square false, n, u \square)\square, t_b \in T_R = \square b, (\Phi, \square true, n, u \square)\square \rightarrow t_a \leq t_b$

A record type with a non-optional field is compatible with a record type identical but for that field being optional.

(46) $t_a \in T_R^{\circ} = [true, (\Phi, [b, n, u])], t_b \in T_R^{\circ} = [true, \Phi[]] \rightarrow t_a \leq t_b$
An open record type is compatible with another open record type with one fewer field.

(47) $t_a \in T_T = (\Phi, [i, [n, u_a]]), t_b \in T_T = (\Phi, [i, [n, u_b]]) \wedge u_a \leq u_b \rightarrow t_a \leq t_b$
A table type is compatible with a second table type, which is identical but for one column having a differing type, when the types for that column are compatible.

REFERENCES

Microsoft Corporation (2015 August)
Microsoft Power Query for Excel Formula Language Specification [PDF]
Retrieved from https://msdn.microsoft.com/library/mt807488.aspx

Microsoft Corporation (n.d.)
Power Query M function reference [web page]
Retrieved from https://msdn.microsoft.com/library/mt779182.aspx

Expressions, values, and let expression

4/14/2021 • 5 minutes to read

A Power Query M formula language query is composed of formula **expression** steps that create a mashup query. A formula expression can be evaluated (computed), yielding a value. The **let** expression encapsulates a set of values to be computed, assigned names, and then used in a subsequent expression that follows the **in** statement. For example, a let expression could contain a **Source** variable that equals the value of **Text.Proper()** and yields a text value in proper case.

Let expression

```
let
    Source = Text.Proper("hello world")
in
    Source
```

In the example above, Text.Proper("hello world") is evaluated to "Hello World".

The next sections describe value types in the language.

Primitive value

A **primitive** value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

TYPE	EXAMPLE VALUE
Binary	00 00 00 02 // number of points (2)
Date	5/23/2015
DateTime	5/23/2015 12:00:00 AM
DateTimeZone	5/23/2015 12:00:00 AM -08:00
Duration	15:35:00
Logical	true and false
Null	null
Number	0, 1, -1, 1.5, and 2.3e-5
Text	"abc"
Time	12:34:12 PM

Function value

A **Function** is a value which, when invoked with arguments, produces a new value. Functions are written by listing the function's **parameters** in parentheses, followed by the goes-to symbol =>, followed by the expression defining the function. For example, to create a function called "MyFunction" that has two parameters and performs a calculation on parameter1 and parameter2:

```
let
    MyFunction = (parameter1, parameter2) => (parameter1 + parameter2) / 2
in
    MyFunction

Calling the MyFunction() returns the result:

let
    Source = MyFunction(2, 4)
in
    Source
```

This code produces the value of 3.

Structured data values

The M language supports the following structured data values:

- List
- Record
- Table
- Additional structured data examples

> **NOTE**
> Structured data can contain any M value. To see a couple of examples, see Additional structured data examples.

List

A List is a zero-based ordered sequence of values enclosed in curly brace characters { }. The curly brace characters { } are also used to retrieve an item from a List by index position. See [List value](#_List_value).

> **NOTE**
> Power Query M supports an infinite list size, but if a list is written as a literal, the list has a fixed length. For example, {1, 2, 3} has a fixed length of 3.

The following are some **List** examples.

VALUE	TYPE
{123, true, "A"}	List containing a number, a logical, and text.
{1, 2, 3}	List of numbers

{
{1, 2, 3},
{4, 5, 6}
}
	List of List of numbers

VALUE	TYPE

{
[CustomerID = 1, Name = "Bob", Phone = "123-4567"],
[CustomerID = 2, Name = "Jim", Phone = "987-6543"]
}
	List of Records

{123, true, "A"}{0}	Get the value of the first item in a List. This expression returns the value 123.
{ {1, 2, 3}, {4, 5, 6} }{0}{1}	Get the value of the second item from the first List element. This expression returns the value 2.

Record

A **Record** is a set of fields. A **field** is a name/value pair where the name is a text value that is unique within the field's record. The syntax for record values allows the names to be written without quotes, a form also referred to as **identifiers**. An identifier can take the following two forms:

- identifier_name such as OrderID.
- #"identifier name" such as #"Today's data is: ".

The following is a record containing fields named "OrderID", "CustomerID", "Item", and "Price" with values 1, 1, "Fishing rod", and 100.00. Square brace characters [ ] denote the beginning and end of a record expression, and are used to get a field value from a record. The follow examples show a record and how to get the Item field value.

Here's an example record:

```
let Source =
        [
              OrderID = 1,
              CustomerID = 1,
              Item = "Fishing rod",
              Price = 100.00
        ]
in Source
```

To get the value of an Item, you use square brackets as Source[Item]:

```
let Source =
    [
          OrderID = 1,
          CustomerID = 1,
          Item = "Fishing rod",
          Price = 100.00
    ]
in Source[Item] //equals "Fishing rod"
```

Table

A **Table** is a set of values organized into named columns and rows. The column type can be implicit or explicit. You can use #table to create a list of column names and list of rows. A **Table** of values is a List in a **List**. The curly brace characters { } are also used to retrieve a row from a **Table** by index position (see Example 3 – Get a row from a table by index position).

Example 1 - Create a table with implicit column types

```
let
  Source = #table(
    {"OrderID", "CustomerID", "Item", "Price"},
      {
          {1, 1, "Fishing rod", 100.00},
          {2, 1, "1 lb. worms", 5.00}
      })
in
    Source
```

Example 2 – Create a table with explicit column types

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price = number],
        {
                {1, 1, "Fishing rod", 100.00},
              {2, 1, "1 lb. worms", 5.00}
        }
    )
in
    Source
```

Both of the examples above creates a table with the following shape:

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	Fishing rod	100.00
2	1	1 lb. worms	5.00

Example 3 – Get a row from a table by index position

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price = number],
        {
              {1, 1, "Fishing rod", 100.00},
              {2, 1, "1 lb. worms", 5.00}
         }
    )
in
    Source{1}
```

This expression returns the follow record:

FIELD	VALUE
OrderID	2
CustomerID	1
Item	1 lb. worms
Price	5

Additional structured data examples

Structured data can contain any M value. Here are some examples:

Example 1 - List with [Primitive](#_Primitive_value_1) values, [Function](#_Function_value), and [Record](#_Record_value)

```
let
    Source =
{
   1,
   "Bob",
   DateTime.ToText(DateTime.LocalNow(), "yyyy-MM-dd"),
   [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
}
in
    Source
```

Evaluating this expression can be visualized as:

A List containing a Record
1
"Bob"
2015-05-22

OrderID	1
CustomerID	1
Item	"Fishing rod"
Price	100.0

Example 2 - Record containing Primitive values and nested Records

```
let
    Source = [CustomerID = 1, Name = "Bob", Phone = "123-4567", Orders =
        {
              [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
            [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0]
        }]
in
    Source
```

Evaluating this expression can be visualized as:

A record containing a List of Records	
CustomerID	1
Name	"Bob"
Phone	"123-4567"
Orders	OrderID: 1 CustomerID: 1 Item: "Fishing rod" Price: 100.0
	OrderID: 2 CustomerID: 1 Item: "1 lb. worms" Price: 5.0

> **NOTE**
>
> Although many values can be written literally as an expression, a value is not an expression. For example, the expression 1 evaluates to the value 1; the expression 1+1 evaluates to the value 2. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

If expression

The **if** expression selects between two expressions based on a logical condition. For example:

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression (2 + 2) is selected if the logical expression (2 > 1) is true, and the second expression (1 + 1) is selected if it is false. The selected expression (in this case 2 + 2) is evaluated and becomes the result of the **if** expression (4).

Comments

3/15/2021 • 2 minutes to read

You can add comments to your code with single-line comments `//` or multi-line comments that begin with `/*` and end with `*/`.

Example - Single-line comment

```
let
   //Convert to proper case.
    Source = Text.Proper("hello world")
in
    Source
```

Example - Multi-line comment

```
/* Capitalize each word in the Item column in the Orders table. Text.Proper
is evaluated for each Item in each table row. */
let
    Orders = Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
          [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
          [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
        #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item", Text.Proper})
in
        #"Capitalized Each Word"
```

Evaluation model

3/15/2021 • 2 minutes to read

The evaluation model of the Power Query M formula language is modeled after the evaluation model commonly found in spreadsheets, where the order of calculations can be determined based on dependencies between the formulas in the cells.
If you have written formulas in a spreadsheet such as Excel, you may recognize the formulas on the left will result in the values on the right when calculated:

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

In M, an expression can reference previous expressions by name, and the evaluation process will automatically determine the order in which referenced expressions are calculated.

Let's use a record to produce an expression which is equivalent to the above spreadsheet example. When initializing the value of a field, you refer to other fields within the record by the name of the field, as follows:

```
[
    A1 = A2 * 2,
    A2 = A3 + 1,
    A3 = 1
]
```

The above expression evaluates to the following record:

```
[
    A1 = 4,
    A2 = 2,
    A3 = 1
]
```

Records can be contained within, or **nested**, within other records. You can use the **lookup operator** ([ ]) to access the fields of a record by name. For example, the following record has a field named Sales containing a record, and a field named Total that accesses the FirstHalf and SecondHalf fields of the Sales record:

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

The above expression evaluates to the following record:

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

You use the **positional index operator** ({ }) to access an item in a list by its numeric index. The values within a list are referred to using a zero-based index from the beginning of the list. For example, the indexes 0 and 1 are used to reference the first and second items in the list below:

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // equals 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // equals 2500
            ]
        },
    #"Total Sales" = Sales{0}[Total] + Sales{1}[Total] // equals 4600
]
```

Lazy and eager evaluation

List, **Record**, and **Table** member expressions, as well as **let** expressions (See Expressions, values, and let expression), are evaluated using **lazy evaluation**: they are evaluated when needed. All other expressions are evaluated using **eager evaluation**: they are evaluated immediately, when encountered during the evaluation process. A good way to think about this is to remember that evaluating a list or record expression will return a list or record value that knows how its list items or record fields need to computed, when requested (by lookup or index operators).

Operators

3/15/2021 • 2 minutes to read

The Power Query M formula language includes a set of operators that can be used in an expression. **Operators** are applied to **operands** to form symbolic expressions. For example, in the expression 1 + 2 the numbers 1 and 2 are operands and the operator is the addition operator (+).

The meaning of an operator can vary depending on the type of operand values. The language has the following operators:

Plus operator (+)

EXPRESSION	EQUALS
1 + 2	Numeric addition: 3
#time(12,23,0) + #duration(0,0,2,0)	Time arithmetic: #time(12,25,0)

Combination operator (&)

FUNCTION	EQUALS
"A" & "BC"	Text concatenation: "ABC"
{1} & {2, 3}	List concatenation: {1, 2, 3}
[a = 1] & [b = 2]	Record merge: [a = 1, b = 2]

List of M operators

Common operators which apply to null, **logical**, number, time, date, datetime, datetimezone, duration, text, binary)

OPERATOR	DESCRIPTION
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
=	Equal
<>	Not equal

Logical operators (In addition to **Common operators**)

OPERATOR	DESCRIPTION
or	Conditional logical OR
and	Conditional logical AND
not	Logical NOT

Number operators (In addition to Common operators)

OPERATOR	DESCRIPTION
+	Sum
-	Difference
*	Product
/	Quotient
+x	Unary plus
-x	Negation

Text operators (In addition to Common operators)

OPERATOR	DESCRIPTION
&	Concatenation

List, record, table operators

OPERATOR	DESCRIPTION
=	Equal
<>	Not equal
&	Concatenation

Record lookup operator

OPERATOR	DESCRIPTION
[]	Access the fields of a record by name.

List indexer operator

OPERATOR	DESCRIPTION
{}	Access an item in a list by its zero-based numeric index.

Type compatibility and assertion operators

OPERATOR	DESCRIPTION
is	The expression x is y returns true if the type of x is compatible with y, and returns false if the type of x is not compatible with y.
as	The expression x as y asserts that the value x is compatible with y as per the is operator.

Date operators

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
x + y	time	duration	Date offset by duration
x + y	duration	time	Date offset by duration
x - y	time	duration	Date offset by negated duration
x - y	time	time	Duration between dates
x & y	date	time	Merged datetime

Datetime operators

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
x + y	datetime	duration	Datetime offset by duration
x + y	duration	datetime	Datetime offset by duration
x - y	datetime	duration	Datetime offset by negated duration
x - y	datetime	datetime	Duration between datetimes

Datetimezone operators

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
x + y	datetimezone	duration	Datetimezone offset by duration
x + y	duration	datetimezone	Datetimezone offset by duration
x - y	datetimezone	duration	Datetimezone offset by negated duration
x - y	datetimezone	datetimezone	Duration between datetimezones

Duration operators

OPERATOR	LEFT OPERAND	RIGHT OPERAND	MEANING
x + y	datetime	duration	Datetime offset by duration
x + y	duration	datetime	Datetime offset by duration
x + y	duration	duration	Sum of durations
x - y	datetime	duration	Datetime offset by negated duration
x - y	datetime	datetime	Duration between datetimes
x - y	duration	duration	Difference of durations
x * y	duration	number	N times a duration
x * y	number	duration	N times a duration
x / y	duration	number	Fraction of a duration

> **NOTE**
> Not all combinations of values may be supported by an operator. Expressions that, when evaluated, encounter undefined operator conditions evaluate to errors. For more information about errors in M, see Errors

Error example:

FUNCTION	EQUALS
1 + "2"	Error: adding number and text is not supported

Type conversion

3/15/2021 • 2 minutes to read

The Power Query M formula language has formulas to convert between types. The following is a summary of conversion formulas in M.

Number

TYPE CONVERSION	DESCRIPTION
Number.FromText(text as text) as number	Returns a number value from a text value.
Number.ToText(number as number) as text	Returns a text value from a number value.
Number.From(value as any) as number	Returns a number value from a value.
Int32.From(value as any) as number	Returns a 32-bit integer number value from the given value.
Int64.From(value as any) as number	Returns a 64-bit integer number value from the given value.
Single.From(value as any) as number	Returns a Single number value from the given value.
Double.From(value as any) as number	Returns a Double number value from the given value.
Decimal.From(value as any) as number	Returns a Decimal number value from the given value.
Currency.From(value as any) as number	Returns a Currency number value from the given value.

Text

TYPE CONVERSION	DESCRIPTION
Text.From(value as any) as text	Returns the text representation of a number, date, time, datetime, datetimezone, logical, duration or binary value.

Logical

TYPE CONVERSION	DESCRIPTION
Logical.FromText(text as text) as logical	Returns a logical value of true or false from a text value.
Logical.ToText(logical as logical) as text	Returns a text value from a logical value.
Logical.From(value as any) as logical	Returns a logical value from a value.

Date, Time, DateTime, and DateTimeZone

TYPE CONVERSION	DESCRIPTION
.FromText(text as text) as date, time, datetime, or datetimezone	Returns a date, time, datetime, or datetimezone value from a set of date formats and culture value.
.ToText(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone) as text	Returns a text value from a date, time, datetime, or datetimezone value.
.From(value as any)	Returns a date, time, datetime, or datetimezone value from a value.
.ToRecord(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone)	Returns a record containing parts of a date, time, datetime, or datetimezone value.

Metadata

3/15/2021 • 2 minutes to read

Metadata is information about a value that is associated with a value. **Metadata** is represented as a record value, called a metadata record. The fields of a **metadata record** can be used to store the metadata for a value. Every value has a metadata record. If the value of the metadata record has not been specified, then the metadata record is empty (has no fields). Associating a metadata record with a value does not change the value's behavior in evaluations except for those that explicitly inspect metadata records.

A metadata record value is associated with a value x using the syntax value meta [record]. For example, the following associates a metadata record with Rating and Tags fields with the text value "Mozart":

```
"Mozart" meta [ Rating = 5,
Tags = {"Classical"} ]
```

A metadata record can be accessed for a value using the `Value.Metadata` function. In the following example, the expression in the ComposerRating field accesses the metadata record of the value in the Composer field, and then accesses the Rating field of the metadata record.

```
[
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
    ComposerRating = Value.Metadata(Composer)[Rating]   // 5
]
```

Metadata records are not preserved when a value is used with an operator or function that constructs a new value. For example, if two text values are concatenated using the & operator, the metadata of the resulting text value is an empty record [].

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata.

Errors

3/15/2021 • 2 minutes to read

An **error** in Power Query M formula language is an indication that the process of evaluating an expression could not produce a value. Errors are raised by operators and functions encountering **error** conditions or by using the **error** expression. Errors are handled using the **try** expression. When an error is raised, a value is specified that can be used to indicate why the error occurred.

Try expression

A try expression converts values and errors into a record value that indicates whether the try expression handled an error, or not, and either the proper value or the error record it extracted when handling the error. For example, consider the following expression that raises an error and then handles it right away:

```
try error "negative unit count"
```

This expression evaluates to the following nested record value, explaining the `[HasError], [Error]`, and `[Message]` field lookups in the unit-price example before.

Error record

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

A common case is to replace errors with default values. The try expression can be used with an optional otherwise clause to achieve just that in a compact form:

```
try error "negative unit count" otherwise 42
// equals 42
```

Error example

```
let Sales =
        [
        ProductName = "Fishing rod",
            Revenue = 2000,
            Units = 1000,
            UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
        ],

    //Get UnitPrice from Sales record
        textUnitPrice = try Number.ToText(Sales[UnitPrice]),
    Label = "Unit Price: " &
        (if textUnitPrice[HasError] then textUnitPrice[Error][Message]
        //Continue expression flow
            else textUnitPrice[Value])
in
    Label
```

The above example accesses the `Sales[UnitPrice]` field and formats the value producing the result:

```
"Unit Price: 2"
```

If the Units field had been zero, then the `UnitPrice` field would have raised an error which would have been handled by the try. The resulting value would then have been:

```
"No Units"
```

Power Query M function reference

3/15/2021 • 2 minutes to read

The Power Query M function reference includes articles for each of the over 700 functions. The reference articles you see here on docs.microsoft.com are auto-generated from in-product help. To learn more about functions and how they work in an expression, see Understanding Power Query M functions.

Functions by category

- Accessing data functions
- Binary functions
- Combiner functions
- Comparer functions
- Date functions
- DateTime functions
- DateTimeZone functions
- Duration functions
- Error handling
- Expression functions
- Function values
- List functions
- Lines functions
- Logical functions
- Number functions
- Record functions
- Replacer functions
- Splitter functions
- Table functions
- Text functions
- Time functions
- Type functions
- Uri functions
- Value functions

Understanding Power Query M functions

3/15/2021 • 2 minutes to read

In the Power Query M formula language, a **function** is a mapping from a set of input values to a single output value. A function is written by first naming the function parameters, and then providing an expression to compute the result of the function. The body of the function follows the goes-to (=>) symbol. Optionally, type information can be included on parameters and the function return value. A function is defined and invoked in the body of a **let** statement. Parameters and/or return value can be implicit or explicit. Implicit parameters and/or return value are of type **any**. Type **any** is similar to an object type in other languages. All types in M derive from type **any**.

A **function** is a value just like a number or a text value, and can be included in-line just like any other expression. The following example shows a function which is the value of an Add variable which is then invoked, or executed, from several other variables. When a function is invoked, a set of values are specified which are logically substituted for the required set of input values within the function body expression.

Example – Explicit parameters and return value

```
let
    AddOne = (x as number) as number => x + 1,
    //additional expression steps
    CalcAddOne = AddOne(5)
in
    CalcAddOne
```

Example – Implicit parameters and return value

```
let
    Add = (x, y) => x + y,
    AddResults =
        [
            OnePlusOne = Add(1, 1),     // equals 2
            OnePlusTwo = Add(1, 2)      // equals 3
    ]
in
    AddResults
```

Find the first element of a list greater than 5, or null otherwise

```
let
    FirstGreaterThan5 = (list) =>
        let
            GreaterThan5 = List.Select(list, (n) => n> 5),
            First = List.First(GreaterThan5)
        in
            First,
    Results =
    [
            Found    = FirstGreaterThan5({3,7,9}),  // equals 7
            NotFound = FirstGreaterThan5({1,3,4})   // equals null
    ]
in
    Results
```

Functions can be used recursively. In order to recursively reference the function, prefix the identifier with @.

```
let
    fact = (num) => if num = 0 then 1 else num * @fact (num-1)
in
    fact(5) // equals 120
```

Each keyword

The **each** keyword is used to easily create simple functions. "each ..." is syntactic sugar for a function signature that takes the _ parameter "(_) => ..."

Each is useful when combined with the lookup operator, which is applied by default to _
For example, each [CustomerID] is the same as each _[CustomerID], which is the same as (_) => _[CustomerID]

Example – Using each in table row filter

```
Table.SelectRows(
      Table.FromRecords({
            [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
            [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
            [CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,
            [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
      }),
      each [CustomerID] = 2
)[Name]

// equals "Jim"
```

Accessing data functions

6/22/2021 • 8 minutes to read

Accessing data

These functions access data and return table values. Most of these functions return a table value called a **navigation table**. Navigation tables are primarily used by the Power Query user interface to provide a navigation experience over the potentially large hierarchical data sets returned.

FUNCTION	DESCRIPTION
AccessControlEntry.ConditionToIdentities	Returns a list of identities that the condition will accept.
AccessControlKind.Allow	Access is allowed.
AccessControlKind.Deny	Access is denied.
Access.Database	Returns a structural representation of an Microsoft Access database.
ActiveDirectory.Domains	Returns a list of Active Directory domains in the same forest as the specified domain or of the current machine's domain if none is specified.
AdobeAnalytics.Cubes	Returns the report suites in Adobe Analytics.
AdoDotNet.DataSource	Returns the schema collection for an ADO.NET data source.
AdoDotNet.Query	Returns the schema collection for an ADO.NET data source.
AnalysisServices.Database	Returns a table of multidimensional cubes or tabular models from the Analysis Services database.
AnalysisServices.Databases	Returns the Analysis Services databases on a particular host.
AzureStorage.BlobContents	Returns the content of the specified blob from an Azure storage vault.
AzureStorage.Blobs	Returns a navigational table containing all containers found in the Azure Storage account. Each row has the container name and a link to the container blobs.
AzureStorage.DataLake	Returns a navigational table containing the documents found in the specified container and its subfolders from Azure Data Lake Storage.
AzureStorage.DataLakeContents	Returns the content of the specified file from an Azure Data Lake Storage filesystem.

FUNCTION	DESCRIPTION
AzureStorage.Tables	Returns a navigational table containing a row for each table found at the account URL from an Azure storage vault. Each row contains a link to the azure table.
Cdm.Contents	This function is unavailable because it requires .NET 4.5.
Csv.Document	Returns the contents of a CSV document as a table using the specified encoding.
CsvStyle.QuoteAfterDelimiter	Quotes in a field are only significant immediately following the delimiter.
CsvStyle.QuoteAlways	Quotes in a field are always significant regardless of where they appear.
Cube.AddAndExpandDimensionColumn	Merges the specified dimension table, dimensionSelector, into the cube's, cube, filter context and changes the dimensional granularity by expanding the specified set, attributeNames, of dimension attributes.
Cube.AddMeasureColumn	Adds a column with the name column to the cube that contains the results of the measure measureSelector applied in the row context of each row.
Cube.ApplyParameter	Returns a cube after applying parameter with arguments to cube.
Cube.AttributeMemberId	Returns the unique member identifier from a member property value.
Cube.AttributeMemberProperty	Returns the property `propertyName` of dimension attribute `attribute`.
Cube.CollapseAndRemoveColumns	Changes the dimensional granularity of the filter context for the cube by collapsing the attributes mapped to the specified columns columnNames.
Cube.Dimensions	Returns a table containing the set of available dimensions within the cube.
Cube.DisplayFolders	Returns a nested tree of tables representing the display folder hierarchy of the objects (e.g. dimensions and measures) available for use in the cube.
Cube.MeasureProperties	Returns a table containing the set of available properties for measures that are expanded in the cube.
Cube.MeasureProperty	Returns the property of a measure.
Cube.Measures	Returns a table containing the set of available measures within the cube.
Cube.Parameters	Returns a table containing the set of parameters that can be applied to cube.

FUNCTION	DESCRIPTION
Cube.Properties	Returns a table containing the set of available properties for dimensions that are expanded in the cube.
Cube.PropertyKey	Returns the key of property `property`.
Cube.ReplaceDimensions	
Cube.Transform	Applies the list cube functions, transforms, on the cube.
DB2.Database	Returns a table of SQL tables and views available in a Db2 database.
Essbase.Cubes	Returns the cubes in an Essbase instance grouped by Essbase server.
Excel.CurrentWorkbook	Returns the tables in the current Excel Workbook.
Excel.Workbook	Returns a table representing sheets in the given excel workbook.
Exchange.Contents	Returns a table of contents from a Microsoft Exchange account.
File.Contents	Returns the binary contents of the file located at a path.
Folder.Contents	Returns a table containing the properties and contents of the files and folders found in the specified folder.
Folder.Files	Returns a table containing a row for each file found at a folder path, and subfolders. Each row contains properties of the folder or file and a link to its content.
GoogleAnalytics.Accounts	Returns the Google Analytics accounts for the current credential.
Hdfs.Contents	Returns a table containing a row for each folder and file found at the folder url, {0}, from a Hadoop file system. Each row contains properties of the folder or file and a link to its content.
Hdfs.Files	Returns a table containing a row for each file found at the folder url, {0}, and subfolders from a Hadoop file system. Each row contains properties of the file and a link to its content.
HdInsight.Containers	Returns a navigational table containing all containers found in the HDInsight account. Each row has the container name and table containing its files.
HdInsight.Contents	Returns a navigational table containing all containers found in the HDInsight account. Each row has the container name and table containing its files.

FUNCTION	DESCRIPTION
HdInsight.Files	Returns a table containing a row for each folder and file found at the container URL, and subfolders from an HDInsight account. Each row contains properties of the file/folder and a link to its content.
Html.Table	Returns a table containing the results of running the specified CSS selectors against the provided html.
Identity.From	Creates an identity.
Identity.IsMemberOf	Determines whether an identity is a member of an identity collection.
IdentityProvider.Default	The default identity provider for the current host.
Informix.Database	Returns a table of SQL tables and views available in an Informix database on server `server` in the database instance named `database`.
Json.Document	Returns the contents of a JSON document. The contents may be directly passed to the function as text, or it may be the binary value returned by a function like File.Contents.
Json.FromValue	Produces a JSON representation of a given value value with a text encoding specified by encoding.
MySQL.Database	Returns a table with data relating to the tables in the specified MySQL Database.
OData.Feed	Returns a table of OData feeds offered by an OData serviceUri.
ODataOmitValues.Nulls	Allows the OData service to omit null values.
Odbc.DataSource	Returns a table of SQL tables and views from the ODBC data source specified by the connection string `connectionString`.
Odbc.InferOptions	Returns the result of trying to infer SQL capabilities for an ODBC driver.
Odbc.Query	Connects to a generic provider with the given connection string and returns the result of evaluating the query.
OleDb.DataSource	Returns a table of SQL tables and views from the OLE DB data source specified by the connection string.
OleDb.Query	Returns the result of running a native query on an OLE DB data source.
Oracle.Database	Returns a table with data relating to the tables in the specified Oracle Database.

FUNCTION	DESCRIPTION
Pdf.Tables	Returns any tables found in pdf.
PostgreSQL.Database	Returns a table with data relating to the tables in the specified PostgreSQL Database.
RData.FromBinary	Returns a record of data frames from the RData file.
Salesforce.Data	Connects to the Salesforce Objects API and returns the set of available objects (i.e. Accounts).
Salesforce.Reports	Connects to the Salesforce Reports API and returns the set of available reports.
SapBusinessWarehouse.Cubes	Returns the InfoCubes and queries in an SAP Business Warehouse system grouped by InfoArea.
SapBusinessWarehouseExecutionMode.DataStream	'DataStream flattening mode' option for MDX execution in SAP Business Warehouse.
SapBusinessWarehouseExecutionMode.BasXml	'bXML flattening mode' option for MDX execution in SAP Business Warehouse.
SapBusinessWarehouseExecutionMode.BasXmlGzip	'Gzip compressed bXML flattening mode' option for MDX execution in SAP Business Warehouse. Recommended for low latency or high volume queries.
SapHana.Database	Returns the packages in an SAP HANA database.
SapHanaDistribution.All	Returns the packages in an SAP HANA database.
SapHanaDistribution.Connection	'Connection' distribution option for SAP HANA.
SapHanaDistribution.Off	'Off' distribution option for SAP HANA.
SapHanaDistribution.Statement	'Statement' distribution option for SAP HANA.
SapHanaRangeOperator.Equals	'Equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.GreaterThan	'Greater than' range operator for SAP HANA input parameters.
SapHanaRangeOperator.GreaterThanOrEquals	'Greater than or equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.LessThan	'Less than' range operator for SAP HANA input parameters.
SapHanaRangeOperator.LessThanOrEquals	'Less than or equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.NotEquals	'Not equals' range operator for SAP HANA input parameters.

FUNCTION	DESCRIPTION
SharePoint.Contents	Returns a table containing a row for each folder and document found at the SharePoint site url. Each row contains properties of the folder or file and a link to its content.
SharePoint.Files	Returns a table containing a row for each document found at the SharePoint site url, and subfolders. Each row contains properties of the folder or file and a link to its content.
SharePoint.Tables	Returns a table containing the result of a SharePoint List as an OData feed.
Soda.Feed	Returns the resulting table of a CSV file that can be accessed using the SODA 2.0 API. The URL must point to a valid SODA-compliant source that ends in a .csv extension.
Sql.Database	Returns a table containing SQL tables located on a SQL Server instance database.
Sql.Databases	Returns a table with references to databases located on a SQL Server instance. Returns a navigation table.
Sybase.Database	Returns a table with data relating to the tables in the specified Sybase Database.
Teradata.Database	Returns a table with data relating to the tables in the specified Teradata Database.
WebAction.Request	Creates an action that, when executed, will return the results of performing a method request against url using HTTP as a binary value.
Web.BrowserContents	Returns the HTML for the specified url, as viewed by a web browser.
Web.Contents	Returns the contents downloaded from a web url as a binary value.
Web.Page	Returns the contents of an HTML webpage as a table.
WebMethod.Delete	Specifies the DELETE method for HTTP.
WebMethod.Get	Specifies the GET method for HTTP.
WebMethod.Head	Specifies the HEAD method for HTTP.
WebMethod.Patch	Specifies the PATCH method for HTTP.
WebMethod.Post	Specifies the POST method for HTTP.
WebMethod.Put	Specifies the PUT method for HTTP.

FUNCTION	DESCRIPTION
Xml.Document	Returns the contents of an XML document as a hierarchical table (list of records).
Xml.Tables	Returns the contents of an XML document as a nested collection of flattened tables.

AccessControlEntry.ConditionToIdentities

3/15/2021 • 2 minutes to read

Syntax

```
AccessControlEntry.ConditionToIdentities(identityProvider as function, condition as function) as
list
```

About

Using the specified `identityProvider`, converts the `condition` into the list of identities for which `condition` would return `true` in all authorization contexts with `identityProvider` as the identity provider. An error is raised if it is not possible to convert `condition` into a list of identities, for example if `condition` consults attributes other than user or group identities to make a decision.

Note that the list of identities represents the identities as they appear in `condition` and no normalization (such as group expansion) is performed on them.

AccessControlKind.Allow

3/15/2021 • 2 minutes to read

About

Access is allowed.

AccessControlKind.Deny

3/15/2021 • 2 minutes to read

About

Access is denied.

Access.Database

3/15/2021 • 2 minutes to read

Syntax

```
Access.Database(database as binary, optional options as nullable record) as table
```

About

Returns a structural representation of an Access database, `database`. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is false).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.

The record parameter is specified as [option1 = value1, option2 = value2...], for example.

ActiveDirectory.Domains

3/15/2021 • 2 minutes to read

Syntax

```
ActiveDirectory.Domains(optional forestRootDomainName as nullable text) as table
```

About

Returns a list of Active Directory domains in the same forest as the specified domain or of the current machine's domain if none is specified.

AdobeAnalytics.Cubes

6/14/2021 • 2 minutes to read

Syntax

```
AdobeAnalytics.Cubes(optional options as nullable record) as table
```

About

Returns a table of multidimensional packages from Adobe Analyics. An optional record parameter, `options`, may be specified to control the following options:

- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MaxRetryCount`: The number of retries to perform when polling for the result of the query. The default value is 120.
- `RetryInterval`: The duration of time between retry attempts. The default value is 1 second.
- `Implementation`: Specifies the internal database provider implementation to use. Valid values are: "IBM" and "Microsoft".

AdoDotNet.DataSource

3/15/2021 • 2 minutes to read

Syntax

```
AdoDotNet.DataSource(providerName as text, connectionString as any, optional options as nullable
record) as table
```

About

Returns the schema collection for the ADO.NET data source with provider name `providerName` and connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.
- `TypeMap`

AdoDotNet.Query

3/15/2021 • 2 minutes to read

Syntax

```
AdoDotNet.Query(providerName as text, connectionString as any, query as text, optional options
as nullable record) as table
```

About

Returns the result of running `query` with the connection string `connectionString` using the ADO.NET provider `providerName`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

AnalysisServices.Database

3/15/2021 • 2 minutes to read

Syntax

```
AnalysisServices.Database(server as text, database as text, optional options as nullable record)
as table
```

About

Returns a table of multidimensional cubes or tabular models from the Analysis Services database `database` on server `server`. An optional record parameter, `options`, may be specified to control the following options:

- `Query` : A native MDX query used to retrieve data.
- `TypedMeasureColumns` : A logical value indicating if the types specified in the multidimensional or tabular model will be used for the types of the added measure columns. When set to false, the type "number" will be used for all measure columns. The default value for this option is false.
- `Culture` : A culture name specifying the culture for the data. This corresponds to the 'Locale Identifier' connection string property.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is driver-dependent.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `SubQueries` : A number (0, 1 or 2) that sets the value of the "SubQueries" property in the connection string. This controls the behavior of calculated members on subselects or subcubes. (The default value is 2).
- `Implementation`

AnalysisServices.Databases

3/15/2021 • 2 minutes to read

Syntax

```
AnalysisServices.Databases(server as text, optional options as nullable record) as table
```

About

Returns databases on an Analysis Services instance, `server`. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `TypedMeasureColumns` : A logical value indicating if the types specified in the multidimensional or tabular model will be used for the types of the added measure columns. When set to false, the type "number" will be used for all measure columns. The default value for this option is false.
- `Culture` : A culture name specifying the culture for the data. This corresponds to the 'Locale Identifier' connection string property.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is driver-dependent.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `SubQueries` : A number (0, 1 or 2) that sets the value of the "SubQueries" property in the connection string. This controls the behavior of calculated members on subselects or subcubes. (The default value is 2).
- `Implementation`

AzureStorage.BlobContents

3/15/2021 • 2 minutes to read

Syntax

```
AzureStorage.BlobContents(url as text, optional options as nullable record) as binary
```

About

Returns the content of the blob at the URL, `url`, from an Azure storage vault. `options` may be specified to control the following options:

- `BlockSize` : The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize` : The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests` : The ConcurrentRequests option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (ConcurrentRequest * RequestSize). The default value is 16.

AzureStorage.Blobs

3/15/2021 • 2 minutes to read

Syntax

```
AzureStorage.Blobs(account as text, optional options as nullable record) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs. `options` may be specified to control the following options:

- `BlockSize` : The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize` : The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests` : The ConcurrentRequests option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (ConcurrentRequest * RequestSize). The default value is 16.

AzureStorage.DataLake

3/15/2021 • 2 minutes to read

Syntax

```
AzureStorage.DataLake(endpoint as text, optional options as nullable record) as table
```

About

Returns a navigational table containing the documents found in the specified container and its subfolders at the account URL, `endpoint`, from an Azure Data Lake Storage filesystem. `options` may be specified to control the following options:

- `BlockSize` : The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize` : The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests` : The ConcurrentRequests option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (ConcurrentRequest * RequestSize). The default value is 16.
- `HierarchicalNavigation` : A logical (true/false) that controls whether the files are returned in a tree-like directory view or in a flat list. The default value is false.

AzureStorage.DataLakeContents

3/15/2021 • 2 minutes to read

Syntax

```
AzureStorage.DataLakeContents(url as text, optional options as nullable record) as binary
```

About

Returns the content of the file at the URL, `url`, from an Azure Data Lake Storage filesystem. `options` may be specified to control the following options:

- `BlockSize` : The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize` : The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests` : The ConcurrentRequests option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (ConcurrentRequest * RequestSize). The default value is 16.

AzureStorage.Tables

6/14/2021 • 2 minutes to read

Syntax

```
AzureStorage.Tables(account as text, optional options as nullable record) as table
```

About

Returns a navigational table containing a row for each table found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the azure table. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Timeout`: A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

Cdm.Contents

3/15/2021 • 2 minutes to read

Syntax

```
Cdm.Contents(table as table) as table
```

About

This function is unavailable because it requires .NET 4.5.

Csv.Document

3/15/2021 • 2 minutes to read

Syntax

```
Csv.Document(source as any, optional columns as any, optional delimiter as any, optional
extraValues as nullable number, optional encoding as nullable number) as table
```

About

Returns the contents of the CSV document as a table.

- `columns` can be null, the number of columns, a list of column names, a table type, or an options record. (See below for more details on the options record.)
- `delimiter` can be a single character, or a list of characters. Default: `","`.
- Please refer to `ExtraValues.Type` for the supported values of `extraValues`.
- `encoding` specifies the text encoding type.

If a record is specified for `columns` (and `delimiter`, `extraValues`, and `encoding` are null), the following record fields may be provided:

- `Delimiter`: The column delimiter. Default: `","`.
- `Columns`: Can be null, the number of columns, a list of column names, or a table type. If the number of columns is lower than the number found in the input, the additional columns will be ignored. If the number of columns is higher than the number found in the input, the additional columns will be null. When not specified, the number of columns will be determined by what is found in the input.
- `Encoding`: The text encoding of the file. Default: 65001 (UTF-8).
- `CsvStyle`: Specifies how quotes are handled. `CsvStyle.QuoteAfterDelimiter` (default): Quotes in a field are only significant immediately following the delimiter. `CsvStyle.QuoteAlways`: Quotes in a field are always significant, regardless of where they appear.
- `QuoteStyle`: Specifies how quoted line breaks are handled. `QuoteStyle.None` (default): All line breaks are treated as the end of the current row, even when they occur inside a quoted value. `QuoteStyle.Csv`: Quoted line breaks are treated as part of the data, not as the end of the current row.

Example 1

Process CSV text with column headers.

```
let
    csv = Text.Combine({"OrderID,Item", "1,Fishing rod", "2,1 lb. worms"}, "#(cr)#(lf)")
in
    Table.PromoteHeaders(Csv.Document(csv))
```

ORDERID	ITEM
1	Fishing rod

ORDERID	ITEM
2	1 lb. worms

CsvStyle.QuoteAfterDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
CsvStyle.QuoteAfterDelimiter
```

About

Quotes in a field are only significant immediately following the delimiter.

CsvStyle.QuoteAlways

3/15/2021 • 2 minutes to read

Syntax

```
CsvStyle.QuoteAlways
```

About

Quotes in a field are always significant regardless of where they appear.

Cube.AddAndExpandDimensionColumn

3/15/2021 • 2 minutes to read

Syntax

```
Cube.AddAndExpandDimensionColumn(**cube** as table, **dimensionSelector** as any,
**attributeNames** as list, optional **newColumnNames** as any) as table
```

About

Merges the specified dimension table, `dimensionSelector`, into the cube's, `cube`, filter context and changes the dimensional granularity by expanding the specified set, `attributeNames`, of dimension attributes. The dimension attributes are added to the tabular view with columns named `newColumnNames`, or `attributeNames` if not specified.

Cube.AddMeasureColumn

3/15/2021 • 2 minutes to read

Syntax

```
Cube.AddMeasureColumn(**cube** as table, **column** as text, **measureSelector** as any) as
table
```

About

Adds a column with the name `column` to the `cube` that contains the results of the measure `measureSelector` applied in the row context of each row. Measure application is affected by changes to dimension granularity and slicing. Measure values will be adjusted after certain cube operations are performed.

Cube.ApplyParameter

3/15/2021 • 2 minutes to read

Syntax

```
Cube.ApplyParameter(cube as table, parameter as any, optional arguments as nullable list) as
table
```

About

Returns a cube after applying `parameter` with `arguments` to `cube`.

Cube.AttributeMemberId

3/15/2021 • 2 minutes to read

Syntax

```
Cube.AttributeMemberId(attribute as any) as any
```

About

Returns the unique member identifier from a member property value. `attribute`. Returns null for any other values.

Cube.AttributeMemberProperty

3/15/2021 • 2 minutes to read

Syntax

```
Cube.AttributeMemberProperty(attribute as any, propertyName as text) as any
```

About

Returns the property `propertyName` of dimension attribute `attribute`.

Cube.CollapseAndRemoveColumns

3/15/2021 • 2 minutes to read

Syntax

```
Cube.CollapseAndRemoveColumns(**cube** as table, **columnNames** as list) as table
```

About

Changes the dimensional granularity of the filter context for the `cube` by collapsing the attributes mapped to the specified columns `columnNames`. The columns are also removed from the tabular view of the cube.

Cube.Dimensions

3/15/2021 • 2 minutes to read

Syntax

```
Cube.Dimensions(**cube** as table) as table
```

About

Returns a table containing the set of available dimensions within the `cube`. Each dimension is a table containing a set of dimension attributes and each dimension attribute is represented as a column in the dimension table. Dimensions can be expanded in the cube using Cube.AddAndExpandDimensionColumn.

Cube.DisplayFolders

3/15/2021 • 2 minutes to read

Syntax

```
Cube.DisplayFolders(**cube** as table) as table
```

About

Returns a nested tree of tables representing the display folder hierarchy of the objects (e.g. dimensions and measures) available for use in the `cube`.

Cube.MeasureProperties

3/15/2021 • 2 minutes to read

Syntax

```
Cube.MeasureProperties(cube as table) as table
```

About

Returns a table containing the set of available properties for measures that are expanded in the cube.

Cube.MeasureProperty

3/15/2021 • 2 minutes to read

Syntax

```
Cube.MeasureProperty(measure as any, propertyName as text) as any
```

About

Returns the property `propertyName` of measure `measure`.

Cube.Measures

3/15/2021 • 2 minutes to read

Syntax

```
Cube.Measures(**cube** as any) as table
```

About

Returns a table containing the set of available measures within the `cube`. Each measure is represented as a function. Measures can be applied to the cube using Cube.AddMeasureColumn.

Cube.Parameters

3/15/2021 • 2 minutes to read

Syntax

```
Cube.Parameters(cube as table) as table
```

About

Returns a table containing the set of parameters that can be applied to `cube`. Each parameter is a function that can be invoked to get `cube` with the parameter and its arguments applied.

Cube.Properties

3/15/2021 • 2 minutes to read

Syntax

```
Cube.Properties(cube as table) as table
```

About

Returns a table containing the set of available properties for dimensions that are expanded in the cube.

Cube.PropertyKey

3/15/2021 • 2 minutes to read

Syntax

```
Cube.PropertyKey(property as any) as any
```

About

Returns the key of property `property`.

Cube.ReplaceDimensions

3/15/2021 • 2 minutes to read

Syntax

```
Cube.ReplaceDimensions(cube as table, dimensions as table) as table
```

About

Cube.ReplaceDimensions

Cube.Transform

3/15/2021 • 2 minutes to read

Syntax

```
Cube.Transform(cube as table, transforms as list) as table
```

About

Applies the list cube functions, `transforms`, on the `cube`.

DB2.Database

3/15/2021 • 2 minutes to read

Syntax

```
DB2.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views available in a Db2 database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `Implementation` : Specifies the internal database provider implementation to use. Valid values are: "IBM" and "Microsoft".
- `BinaryCodePage` : A number for the CCSID (Coded Character Set Identifier) to decode Db2 FOR BIT binary data into character strings. Applies to Implementation = "Microsoft". Set 0 to disable conversion (default). Set 1 to convert based on database encoding. Set other CCSID number to convert to application encoding.
- `PackageCollection` : Specifies a string value for package collection (default is "NULLID") to enable use of shared packages required to process SQL statements. Applies to Implementation = "Microsoft".
- `UseDb2ConnectGateway` : Specifies whether the connection is being made through a Db2 Connect gateway. Applies to Implementation = "Microsoft".

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Essbase.Cubes

3/15/2021 • 2 minutes to read

Syntax

```
Essbase.Cubes(url as text, optional options as nullable record) as table
```

About

Returns a table of cubes grouped by Essbase server from an Essbase instance at APS server `url`. An optional record parameter, `options`, may be specified to control the following options:

- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.

Excel.CurrentWorkbook

3/15/2021 • 2 minutes to read

Syntax

```
Excel.CurrentWorkbook() as table
```

About

Returns the contents of the current Excel workbook.

Excel.Workbook

3/15/2021 • 2 minutes to read

Syntax

```
Excel.Workbook(workbook as binary, optional useHeaders as any, optional delayTypes as nullable
logical) as table
```

About

Returns the contents of the Excel workbook.

- `useHeaders` can be null, a logical (true/false) value indicating whether the first row of each returned table should be treated as a header, or an options record. (See below for more details on the options record.) Default: false.
- `delayTypes` can be null or a logical (true/false) value indicating whether the columns of each returned table should be left untyped. Default: false.

If a record is specified for `useHeaders` (and `delayTypes` is null), the following record fields may be provided:

- `UseHeaders`: Can be null or a logical (true/false) value indicating whether the first row of each returned table should be treated as a header. Default: false.
- `DelayTypes`: Can be null or a logical (true/false) value indicating whether the columns of each returned table should be left untyped. Default: false.
- `InferSheetDimensions`: Can be null or a logical (true/false) value indicating whether the area of a worksheet that contains data should be inferred by reading the worksheet itself, rather than by reading the dimensions metadata from the file. This can be useful in cases where the dimensions metadata is incorrect. Note that this option is only supported for Open XML Excel files, not for legacy Excel files. Default: false.

Exchange.Contents

3/15/2021 • 2 minutes to read

Syntax

```
Exchange.Contents (optional mailboxAddress as nullable text) as table
```

About

Returns a table of contents from the Microsoft Exchange account `mailboxAddress`. If `mailboxAddress` is not specified, the default account for the credential will be used.

File.Contents

3/15/2021 • 2 minutes to read

Syntax

```
File.Contents(path as text, optional options as nullable record) as binary
```

About

Returns the contents of the file, `path`, as binary.

Folder.Contents

3/15/2021 • 2 minutes to read

Syntax

```
Folder.Contents(path as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each folder and file found at the folder path, `path`. Each row contains properties of the folder or file and a link to its content.

Folder.Files

3/15/2021 • 2 minutes to read

Syntax

```
Folder.Files(path as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each file found at the folder path, `path`, and subfolders. Each row contains properties of the file and a link to its content.

GoogleAnalytics.Accounts

3/15/2021 • 2 minutes to read

Syntax

```
GoogleAnalytics.Accounts() as table
```

About

Returns Google Analytics accounts that are accessible from the current credential.

Hdfs.Contents

3/15/2021 • 2 minutes to read

Syntax

```
Hdfs.Contents(url as text) as table
```

About

Returns a table containing a row for each folder and file found at the folder URL, `url`, from a Hadoop file system. Each row contains properties of the folder or file and a link to its content.

Hdfs.Files

3/15/2021 • 2 minutes to read

Syntax

```
Hdfs.Files(url as text) as table
```

About

Returns a table containing a row for each file found at the folder URL, `url`, and subfolders from a Hadoop file system. Each row contains properties of the file and a link to its content.

HdInsight.Containers

3/15/2021 • 2 minutes to read

Syntax

```
HdInsight.Containers(account as text) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs.

HdInsight.Contents

3/15/2021 • 2 minutes to read

Syntax

```
HdInsight.Contents(account as text) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs.

HdInsight.Files

3/15/2021 • 2 minutes to read

Syntax

```
HdInsight.Files(account as text, containerName as text) as table
```

About

Returns a table containing a row for each blob file found at the container URL, `account`, from an Azure storage vault. Each row contains properties of the file and a link to its content.

Html.Table

3/15/2021 • 2 minutes to read

Syntax

```
Html.Table(html as any, columnNameSelectorPairs as list, optional options as nullable record) as table
```

About

Returns a table containing the results of running the specified CSS selectors against the provided `html`. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `RowSelector`

Example 1

Returns a table from a sample html text value.

```
Html.Table("<div class=""name"">Jo</div><span>Manager</span>", {{"Name", ".name"}, {"Title", "span"}}, [RowSelector=".name"])\
```

```
#table({"Name", "Title"}, {{"Jo", "Manager"}})
```

Example 2

Extracts all the hrefs from a sample html text value.

```
Html.Table("<a href=""/test.html"">Test</a>", {{"Link", "a", each [Attributes][href]}})
```

```
#table({"Link"}, {{"/test.html"}})
```

Identity.From

3/15/2021 • 2 minutes to read

Syntax

```
Identity.From(identityProvider as function, value as any) as record
```

About

Creates an identity.

Identity.IsMemberOf

3/15/2021 • 2 minutes to read

Syntax

```
Identity.IsMemberOf(identity as record, collection as record) as logical
```

About

Determines whether an identity is a member of an identity collection.

IdentityProvider.Default

3/15/2021 • 2 minutes to read

Syntax

```
IdentityProvider.Default() as any
```

About

The default identity provider for the current host.

Informix.Database

3/15/2021 • 2 minutes to read

Syntax

```
Informix.Database(server as text, database as text, optional options as nullable record) as
table
```

About

Returns a table of SQL tables and views available in an Informix database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Json.Document

3/15/2021 • 2 minutes to read

Syntax

```
Json.Document(jsonText as any, optional encoding as nullable number) as any
```

About

Returns the content of the JSON document.

Json.FromValue

3/15/2021 • 2 minutes to read

Syntax

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

About

Produces a JSON representation of a given value `value` with a text encoding specified by `encoding`. If `encoding` is omitted, UTF8 is used. Values are represented as follows:

- Null, text and logical values are represented as the corresponding JSON types
- Numbers are represented as numbers in JSON, except that `#infinity`, `-#infinity` and `#nan` are converted to null
- Lists are represented as JSON arrays
- Records are represnted as JSON objects
- Tables are represented as an array of objects
- Dates, times, datetimes, datetimezones and durations are represented as ISO-8601 text
- Binary values are represented as base-64 encoded text
- Types and functions produce an error

Example 1

Convert a complex value to JSON.

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

```
"{""A"":[1,true,""3""],""B"":""2012-03-25""}"
```

MySQL.Database

3/15/2021 • 2 minutes to read

Syntax

```
MySQL.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables, views, and stored scalar functions available in a MySQL database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `Encoding` : A TextEncoding value that specifies the character set used to encode all queries sent to the server (default is null).
- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `TreatTinyAsBoolean` : A logical (true/false) that determines whether to force tinyint columns on the server as logical values. The default value is true.
- `OldGuids` : A logical (true/false) that sets whether char(36) columns (if false) or binary(16) columns (if true) will be treated as GUIDs. The default value is false.
- `ReturnSingleDatabase` : A logical (true/false) that sets whether to return all tables of all databases (if false) or to return tables and views of the specified database (if true). The default value is false.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

OData.Feed

3/15/2021 • 2 minutes to read

Syntax

```
OData.Feed(serviceUri as text, optional headers as nullable record, optional options as any) as any
```

About

Returns a table of OData feeds offered by an OData service from a uri `serviceUri`, headers `headers`. A boolean value specifying whether to use concurrent connections or an optional record parameter, `options`, may be specified to control the following options:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `Headers` : Specifying this value as a record will supply additional headers to an HTTP request.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `ApiKeyName` : If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Timeout` : Specifying this value as a duration will change the timeout for an HTTP request. The default value is 600 seconds.
- `EnableBatch` : A logical (true/false) that sets whether to allow generation of an OData $batch request if the MaxUriLength is exceeded (default is false).
- `MaxUriLength` : A number that indicates the max length of an allowed uri sent to an OData service. If exceeded and EnableBatch is true then the request will be made to an OData $batch endpoint, otherwise it will fail (default is 2048).
- `Concurrent` : A logical (true/false) when set to true, requests to the service will be made concurrently. When set to false, requests will be made sequentially. When not specified, the value will be determined by the service's AsynchronousRequestsSupported annotation. If the service does not specify whether AsynchronousRequestsSupported is supported, requests will be made sequentially.
- `ODataVersion` : A number (3 or 4) that specifies the OData protocol version to use for this OData service. When not specified, all supported versions will be requested. The service version will be determined by the OData-Version header returned by the service.
- `FunctionOverloads` : A logical (true/false) when set to true, function import overloads will be listed in the navigator as separate entries, when set to false, function import overloads will be listed as one union function in the navigator. Default value for V3: false. Default value for V4: true.
- `MoreColumns` : A logical (true/false) when set to true, adds a "More Columns" column to each entity feed containing open types and polymorphic types. This will contain the fields not declared in the base type. When false, this field is not present. Defaults to false.
- `IncludeAnnotations` : A comma separated list of namespace qualified term names or patterns to include with "*" as a wildcard. By default, none of the annotations are included.
- `IncludeMetadataAnnotations` : A comma separated list of namespace qualified term names or patterns to include on metadata document requests, with "*" as a wildcard. By default, includes the same annotations as IncludeAnnotations.

- `OmitValues` : Allows the OData service to avoid writing out certain values in responses. If acknowledged, we will infer those values from the omitted fields. Options include:
- `ODataOmitValues.Nulls` : Allows the OData service to omit null values.
- `Implementation` : Specifies the implementation of the OData connector to use. Valid values are "2.0" or null.

ODataOmitValues.Nulls

3/15/2021 • 2 minutes to read

About

Allows the OData service to omit null values.

Odbc.DataSource

3/15/2021 • 2 minutes to read

Syntax

```
Odbc.DataSource(connectionString as any, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the ODBC data source specified by the connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is 15 seconds.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

Odbc.InferOptions

3/15/2021 • 2 minutes to read

Syntax

```
Odbc.InferOptions(connectionString as any) as record
```

About

Returns the result of trying to infer SQL capbabilities with the connection string `connectionString` using ODBC. `connectionString` can be text or a record of property value pairs. Property values can either be text or number.

Odbc.Query

3/15/2021 • 2 minutes to read

Syntax

```
Odbc.Query(connectionString as any, query as text, optional options as nullable record) as table
```

About

Returns the result of running `query` with the connection string `connectionString` using ODBC. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is 15 seconds.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

OleDb.DataSource

3/15/2021 • 2 minutes to read

Syntax

```
OleDb.DataSource(connectionString as any, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the OLE DB data source specified by the connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is true).
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

OleDb.Query

3/15/2021 • 2 minutes to read

Syntax

```
OleDb.Query(connectionString as any, query as text, optional options as nullable record) as
table
```

About

Returns the result of running `query` with the connection string `connectionString` using OLE DB. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth` : A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

Oracle.Database

3/15/2021 • 2 minutes to read

Syntax

```
Oracle.Database(server as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the Oracle database on server `server`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Pdf.Tables

6/22/2021 • 2 minutes to read

Syntax

```
Pdf.Tables(pdf as binary, optional options as nullable record) as table
```

About

Returns any tables found in `pdf`. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Implementation`: The version of the algorithm to use when identifying tables. Old versions are available only for backwards compatibility, to prevent old queries from being broken by algorithm updates. The newest version should always give the best results. Valid values are "1.3", "1.2", "1.1", or null.
- `StartPage`: Specifies the first page in the range of pages to examine. Default: 1.
- `EndPage`: Specifies the last page in the range of pages to examine. Default: the last page of the document.
- `MultiPageTables`: Controls whether similar tables on consecutive pages will be automatically combined into a single table. Default: true.
- `EnforceBorderLines`: Controls whether border lines are always enforced as cell boundaries (when true), or simply used as one hint among many for determining cell boundaries (when false). Default: false.

Example 1

Returns the tables contained in sample.pdf.

```
Pdf.Tables(File.Contents("c:\sample.pdf"))
```

```
#table({"Name", "Kind", "Data"}, ...)
```

PostgreSQL.Database

3/15/2021 • 2 minutes to read

Syntax

```
PostgreSQL.Database(server as text, database as text, optional options as nullable record) as
table
```

About

Returns a table of SQL tables and views available in a PostgreSQL database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

RData.FromBinary

3/15/2021 • 2 minutes to read

Syntax

```
RData.FromBinary(stream as binary) as any
```

About

Returns a record of data frames from the RData file.

Salesforce.Data

3/15/2021 • 2 minutes to read

Syntax

```
Salesforce.Data(optional loginUrl as any, optional options as nullable record) as table
```

About

Returns the objects on the Salesforce account provided in the credentials. The account will be connected through the provided environment `loginUrl`. If no environment is provided then the account will connect to production (https://login.salesforce.com). An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is false).
- `ApiVersion` : The Salesforce API version to use for this query. When not specified, API version 29.0 is used.
- `Timeout` : A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

Salesforce.Reports

3/15/2021 • 2 minutes to read

Syntax

```
Salesforce.Reports(optional loginUrl as nullable text, optional options as nullable record) as
table
```

About

Returns the reports on the Salesforce account provided in the credentials. The account will be connected through the provided environment `loginUrl`. If no environment is provided then the account will connect to production (https://login.salesforce.com). An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ApiVersion`: The Salesforce API version to use for this query. When not specified, API version 29.0 is used.
- `Timeout`: A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

SapBusinessWarehouse.Cubes

3/15/2021 • 2 minutes to read

Syntax

```
SapBusinessWarehouse.Cubes(server as text, systemNumberOrSystemId as text, clientId as text,
optional optionsOrLogonGroup as any, optional options as nullable record) as table
```

About

Returns a table of InfoCubes and queries grouped by InfoArea from an SAP Business Warehouse instance at server `server` with system number `systemNumberOrSystemId` and Client ID `clientId`. An optional record parameter, `optionsOrLogonGroup`, may be specified to control options.

sapbusinesswarehouseexecutionmode.datastream

3/15/2021 • 2 minutes to read

About

'DataStream flattening mode' option for MDX execution in SAP Business Warehouse.

SapBusinessWarehouseExecutionMode.BasXml

3/15/2021 • 2 minutes to read

About

'bXML flattening mode' option for MDX execution in SAP Business Warehouse.

SapBusinessWarehouseExecutionMode.BasXmlGzip

3/15/2021 • 2 minutes to read

About

'Gzip compressed bXML flattening mode' option for MDX execution in SAP Business Warehouse. Recommended for low latency or high volume queries.

SapHana.Database

3/15/2021 • 2 minutes to read

Syntax

```
SapHana.Database(**server** as text, optional **options** as nullable record) as table
```

About

Returns a table of multidimensional packages from the SAP HANA database `server`. An optional record parameter, `options`, may be specified to control the following options:

- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `Distribution` : A SapHanaDistribution that sets the value of the "Distribution" property in the connection string. Statement routing is the method of evaluating the correct server node of a distributed system before statement execution. The default value is SapHanaDistribution.All.

SapHanaDistribution.All

3/15/2021 • 2 minutes to read

About

'All' distribution option for SAP HANA.

SapHanaDistribution.Connection

3/15/2021 • 2 minutes to read

About

'Connection' distribution option for SAP HANA.

SapHanaDistribution.Off

3/15/2021 • 2 minutes to read

About

'Off' distribution option for SAP HANA.

SapHanaDistribution.Statement

3/15/2021 • 2 minutes to read

About

'Statement' distribution option for SAP HANA.

SapHanaRangeOperator.Equals

3/15/2021 • 2 minutes to read

About

'Equals' range operator for SAP HANA input parameters.

SapHanaRangeOperator.GreaterThan

3/15/2021 • 2 minutes to read

About

'Greater than' range operator for SAP HANA input parameters.

SapHanaRangeOperator.GreaterThanOrEquals

3/15/2021 • 2 minutes to read

About

'Greater than or equals' range operator for SAP HANA input parameters.

SapHanaRangeOperator.LessThan

3/15/2021 • 2 minutes to read

About

'Less than' range operator for SAP HANA input parameters.

SapHanaRangeOperator.LessThanOrEquals

3/15/2021 • 2 minutes to read

About

'Less than or equals' range operator for SAP HANA input parameters.

SapHanaRangeOperator.NotEquals

3/15/2021 • 2 minutes to read

About

'Not equals' range operator for SAP HANA input parameters.

SharePoint.Contents

3/15/2021 • 2 minutes to read

Syntax

```
SharePoint.Contents(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each folder and document found at the specified SharePoint site, `url`. Each row contains properties of the folder or file and a link to its content. `options` may be specified to control the following options:

- `ApiVersion` : A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.

SharePoint.Files

3/15/2021 • 2 minutes to read

Syntax

```
SharePoint.Files(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each document found at the specified SharePoint site, `url`, and subfolders. Each row contains properties of the folder or file and a link to its content. `options` may be specified to control the following options:

- `ApiVersion` : A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.

SharePoint.Tables

3/15/2021 • 2 minutes to read

Syntax

```
SharePoint.Tables(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each List item found at the specified SharePoint list, `url`. Each row contains properties of the List. `options` may be specified to control the following options:

- `ApiVersion` : A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.
- `Implementation`
- `ViewMode`

Soda.Feed

3/15/2021 • 2 minutes to read

Syntax

```
Soda.Feed(url as text) as table
```

About

Returns a table from the contents at the specified URL `url` formatted according to the SODA 2.0 API. The URL must point to a valid SODA-compliant source that ends in a .csv extension.

Sql.Database

3/15/2021 • 2 minutes to read

Syntax

```
Sql.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables, views, and stored functions from the SQL Server database `database` on server `server`. The port may be optionally specified with the server, separated by a colon or a comma. An optional record parameter, `options`, may be specified to control the following options:

- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `MaxDegreeOfParallelism`: A number that sets the value of the "maxdop" query clause in the generated SQL query.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MultiSubnetFailover`: A logical (true/false) that sets the value of the "MultiSubnetFailover" property in the connection string (default is false).
- `UnsafeTypeConversions`: A logical (true/false) that, if true, attempts to fold type conversions which could fail and cause the entire query to fail. Not recommended for general use.
- `ContextInfo`: A binary value that is used to set the CONTEXT_INFO before running each command.
- `OmitSRID`: A logical (true/false) that, if true, omits the SRID when producing Well-Known Text from geometry and geography types.

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Sql.Databases

3/15/2021 • 2 minutes to read

Syntax

```
Sql.Databases(server as text, optional options as nullable record) as table
```

About

Returns a table of databases on the specified SQL server, `server`. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `MaxDegreeOfParallelism`: A number that sets the value of the "maxdop" query clause in the generated SQL query.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MultiSubnetFailover`: A logical (true/false) that sets the value of the "MultiSubnetFailover" property in the connection string (default is false).
- `UnsafeTypeConversions`: A logical (true/false) that, if true, attempts to fold type conversions which could fail and cause the entire query to fail. Not recommended for general use.
- `ContextInfo`: A binary value that is used to set the CONTEXT_INFO before running each command.
- `OmitSRID`: A logical (true/false) that, if true, omits the SRID when producing Well-Known Text from geometry and geography types.

The record parameter is specified as [option1 = value1, option2 = value2...] for example.

Does not support setting a SQL query to run on the server. `Sql.Database` should be used instead to run a SQL query.

Sybase.Database

3/15/2021 • 2 minutes to read

Syntax

```
Sybase.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views available in a Sybase database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Teradata.Database

3/15/2021 • 2 minutes to read

Syntax

```
Teradata.Database(server as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the Teradata database on server `server`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties` : A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator` : A function that is used for the creation of names for navigation properties.
- `Query` : A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout` : A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout` : A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation` : A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

WebAction.Request

3/15/2021 • 2 minutes to read

Syntax

```
WebAction.Request(method as text, url as text, optional options as nullable record) as action
```

About

Creates an action that, when executed, will return the results of performing a `method` request against `url` using HTTP as a binary value. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 100 seconds.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `IsRetry`: Specifying this logical value as true will ignore any existing response in the cache when fetching data.
- `ManualStatusHandling`: Specifying this value as a list will prevent any builtin handling for HTTP requests whose response has one of these status codes.
- `RelativePath`: Specifying this value as text appends it to the base URL before making the request.
- `Content`: Specifying this value will cause its contents to become the body of the HTTP request.

Web.BrowserContents

3/15/2021 • 2 minutes to read

Syntax

```
Web.BrowserContents(url as text, optional options as nullable record) as text
```

About

Returns the HTML for the specified `url`, as viewed by a web browser. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `WaitFor`: Specifies a condition to wait for before downloading the HTML, in addition to waiting for the page to load (which is always done). Can be a record containing Timeout and/or Selector fields. If only a Timeout is specified, the function will wait the amount of time specified before downloading the HTML. If both a Selector and Timeout are specified, and the Timeout elapses before the Selector exists on the page, an error will be thrown. If a Selector is specified with no Timeout, a default Timeout of 30 seconds is applied.

Example 1

Returns the HTML for https://microsoft.com.

```
Web.BrowserContents("https://microsoft.com")
```

```
"<!DOCTYPE html><html xmlns=..."
```

Example 2

Returns the HTML for https://microsoft.com after waiting for a CSS selector to exist.

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Selector = "div.ready"]])
```

```
"<!DOCTYPE html><html xmlns=..."
```

Example 3

Returns the HTML for https://microsoft.com after waiting ten seconds.

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Timeout = #duration(0,0,0,10)]])
```

```
"<!DOCTYPE html><html xmlns=..."
```

Example 4

Returns the HTML for https://microsoft.com after waiting up to ten seconds for a CSS selector to exist.

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Selector = "div.ready", Timeout =
#duration(0,0,0,10)]])
```

```
"<!DOCTYPE html><html xmlns=..."
```

Web.Contents

3/15/2021 • 2 minutes to read

Syntax

```
Web.Contents(url as text, optional options as nullable record) as binary
```

About

Returns the contents downloaded from `url` as binary. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 100 seconds.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `IsRetry`: Specifying this logical value as true will ignore any existing response in the cache when fetching data.
- `ManualStatusHandling`: Specifying this value as a list will prevent any builtin handling for HTTP requests whose response has one of these status codes.
- `RelativePath`: Specifying this value as text appends it to the base URL before making the request.
- `Content`: Specifying this value changes the web request from a GET to a POST, using the value of the option as the content of the POST.

The headers of the HTTP response are available as metadata on the binary result. Outside of a custom data connector context, only the Content-Type header is available.

Web.Page

3/15/2021 • 2 minutes to read

Syntax

```
Web.Page(html as any) as table
```

About

Returns the contents of the HTML document broken into its constituent structures, as well as a representation of the full document and its text after removing tags.

WebMethod.Delete

3/15/2021 • 2 minutes to read

About

Specifies the DELETE method for HTTP.

WebMethod.Get

3/15/2021 • 2 minutes to read

About

Specifies the GET method for HTTP.

WebMethod.Head

3/15/2021 • 2 minutes to read

About

Specifies the HEAD method for HTTP.

WebMethod.Patch

3/15/2021 • 2 minutes to read

About

Specifies the PATCH method for HTTP.

WebMethod.Post

3/15/2021 • 2 minutes to read

About

Specifies the POST method for HTTP.

WebMethod.Put

3/15/2021 • 2 minutes to read

About

Specifies the PUT method for HTTP.

Xml.Document

3/15/2021 • 2 minutes to read

Syntax

```
Xml.Document(contents as any, optional encoding as nullable number) as table
```

About

Returns the contents of the XML document as a hierarchical table.

Xml.Tables

3/15/2021 • 2 minutes to read

Syntax

```
Xml.Tables(contents as any, optional options as nullable record, optional encoding as nullable number) as table
```

About

Returns the contents of the XML document as a nested collection of flattened tables.

Binary functions

3/15/2021 • 3 minutes to read

These functions create and manipulate binary data.

Binary Formats

Reading numbers

FUNCTION	DESCRIPTION
BinaryFormat.7BitEncodedSignedInteger	A binary format that reads a 64-bit signed integer that was encoded using a 7-bit variable-length encoding.
BinaryFormat.7BitEncodedUnsignedInteger	A binary format that reads a 64-bit unsigned integer that was encoded using a 7-bit variable-length encoding.
BinaryFormat.Binary	Returns a binary format that reads a binary value.
BinaryFormat.Byte	A binary format that reads an 8-bit unsigned integer.
BinaryFormat.Choice	Returns a binary format that chooses the next binary format based on a value that has already been read.
BinaryFormat.Decimal	A binary format that reads a .NET 16-byte decimal value.
BinaryFormat.Double	A binary format that reads an 8-byte IEEE double-precision floating point value.
BinaryFormat.Group	Returns a binary format that reads a group of items. Each item value is preceded by a unique key value. The result is a list of item values.
BinaryFormat.Length	Returns a binary format that limits the amount of data that can be read. Both BinaryFormat.List and BinaryFormat.Binary can be used to read until end of the data. BinaryFormat.Length can be used to limit the number of bytes that are read.
BinaryFormat.List	Returns a binary format that reads a sequence of items and returns a list.
BinaryFormat.Null	A binary format that reads zero bytes and returns null.
BinaryFormat.Record	Returns a binary format that reads a record. Each field in the record can have a different binary format.
BinaryFormat.SignedInteger16	A binary format that reads a 16-bit signed integer.
BinaryFormat.SignedInteger32	A binary format that reads a 32-bit signed integer.

FUNCTION	DESCRIPTION
BinaryFormat.SignedInteger64	A binary format that reads a 64-bit signed integer.
BinaryFormat.Single	A binary format that reads a 4-byte IEEE single-precision floating point value.
BinaryFormat.Text	Returns a binary format that reads a text value. The optional encoding value specifies the encoding of the text.
BinaryFormat.Transform	Returns a binary format that will transform the values read by another binary format.
BinaryFormat.UnsignedInteger16	A binary format that reads a 16-bit unsigned integer.
BinaryFormat.UnsignedInteger32	A binary format that reads a 32-bit unsigned integer.
BinaryFormat.UnsignedInteger64	A binary format that reads a 64-bit unsigned integer.

CONTROLLING BYTE ORDER	DESCRIPTION
BinaryFormat.ByteOrder	Returns a binary format with the byte order specified by a function.
Table.PartitionValues	Returns information about how a table is partitioned.

Binary

FUNCTION	DESCRIPTION
Binary.Buffer	Buffers the binary value in memory. The result of this call is a stable binary value, which means it will have a deterministic length and order of bytes.
Binary.Combine	Combines a list of binaries into a single binary.
Binary.Compress	Compresses a binary value using the given compression type.
Binary.Decompress	Decompresses a binary value using the given compression type.
Binary.From	Returns a binary value from the given value.
Binary.FromList	Converts a list of numbers into a binary value
Binary.FromText	Decodes data from a text form into binary.
Binary.InferContentType	Returns a record with field Content.Type that contains the inferred MIME-type.
Binary.Length	Returns the length of binary values.

FUNCTION	DESCRIPTION
Binary.Range	Returns a subset of the binary value beginning at an offset.
Binary.ToList	Converts a binary value into a list of numbers
Binary.ToText	Encodes binary data into a text form.
BinaryEncoding.Base64	Constant to use as the encoding type when base-64 encoding is required.
BinaryEncoding.Hex	Constant to use as the encoding type when hexadecimal encoding is required.
BinaryOccurrence.Optional	The item is expected to appear zero or one time in the input.
BinaryOccurrence.Repeating	The item is expected to appear zero or more times in the input.
BinaryOccurrence.Required	The item is expected to appear once in the input.
ByteOrder.BigEndian	A possible value for the `byteOrder` parameter in `BinaryFormat.ByteOrder`. The most signficant byte appears first in Big Endian byte order.
ByteOrder.LittleEndian	A possible value for the `byteOrder` parameter in `BinaryFormat.ByteOrder`. The least signficant byte appears first in Little Endian byte order.
Compression.Brotli	The compressed data is in the 'Brotli' format.
Compression.Deflate	The compressed data is in the 'Deflate' format.
Compression.GZip	The compressed data is in the 'GZip' format.
Compression.LZ4	The compressed data is in the 'LZ4' format.
Compression.None	The data is uncompressed.
Compression.Snappy	The compressed data is in the 'Snappy' format.
Compression.Zstandard	The compressed data is in the 'Zstandard' format.
Occurrence.Optional	The item is expected to appear zero or one time in the input.
Occurrence.Repeating	The item is expected to appear zero or more times in the input.
Occurrence.Required	The item is expected to appear once in the input.
#binary	Creates a binary value from numbers or text.

Binary.Buffer

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Buffer(binary as nullable binary) as nullable binary
```

About

Buffers the binary value in memory. The result of this call is a stable binary value, which means it will have a deterministic length and order of bytes.

Example 1

Create a stable version of the binary value.

```
Binary.Buffer(Binary.FromList({0..10}))
```

```
#binary({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```

Binary.Combine

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Combine(binaries as list) as binary
```

About

Combines a list of binaries into a single binary.

Binary.Compress

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Compress(binary as nullable binary, compressionType as number) as nullable binary
```

About

Compresses a binary value using the given compression type. The result of this call is a compressed copy of the input. Compression types include:

- `Compression.GZip`
- `Compression.Deflate`

Example 1

Compress the binary value.

```
Binary.Compress(Binary.FromList(List.Repeat({10}, 1000)), Compression.Deflate)
```

```
#binary({227, 226, 26, 5, 163, 96, 20, 12, 119, 0, 0})
```

Binary.Decompress

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Decompress(binary as nullable binary, compressionType as number) as nullable binary
```

About

Decompresses a binary value using the given compression type. The result of this call is a decompressed copy of the input. Compression types include:

- `Compression.GZip`
- `Compression.Deflate`

Example 1

Decompress the binary value.

```
Binary.Decompress(#binary({115, 103, 200, 7, 194, 20, 134, 36, 134, 74, 134, 84, 6, 0}),
Compression.Deflate)
```

```
#binary({71, 0, 111, 0, 111, 0, 100, 0, 98, 0, 121, 0, 101, 0})
```

Binary.From

3/15/2021 • 2 minutes to read

Syntax

```
Binary.From(value as any, optional encoding as nullable number) as nullable binary
```

About

Returns a `binary` value from the given `value`. If the given `value` is `null`, `Binary.From` returns `null`. If the given `value` is `binary`, `value` is returned. Values of the following types can be converted to a `binary` value:

- `text`: A `binary` value from the text representation. See `Binary.FromText` for details.

If `value` is of any other type, an error is returned.

Example 1

Get the `binary` value of `"1011"`.

```
Binary.From("1011")
```

`Binary.FromText("1011", BinaryEncoding.Base64)`

Binary.FromList

3/15/2021 • 2 minutes to read

Syntax

```
Binary.FromList(list as list) as binary
```

About

Converts a list of numbers into a binary value.

Binary.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Binary.FromText(text as nullable text, optional encoding as nullable number) as nullable binary
```

About

Returns the result of converting text value `text` to a binary (list of `number`). `encoding` may be specified to indicate the encoding used in the text value. The following `BinaryEncoding` values may be used for `encoding`.

- `BinaryEncoding.Base64`: Base 64 encoding
- `BinaryEncoding.Hex`: Hex encoding

Example 1

Decode `"1011"` into binary.

```
Binary.FromText("1011")
```

```
Binary.FromText("1011", BinaryEncoding.Base64)
```

Example 2

Decode `"1011"` into binary with Hex encoding.

```
Binary.FromText("1011", BinaryEncoding.Hex)
```

```
Binary.FromText("EBE=", BinaryEncoding.Base64)
```

Binary.InferContentType

3/15/2021 • 2 minutes to read

Syntax

```
Binary.InferContentType(source as binary) as record
```

About

Returns a record with field Content.Type that contains the inferred MIME-type. If the inferred content type is text/*, and an encoding code page is detected, then additionally returns field Content.Encoding that contains the encoding of the stream. If the inferred content type is text/csv, and the format is delimited, additionally returns field Csv.PotentialDelimiter containing a table for analysis of potential delimiters. If the inferred content type is text/csv, and the format is fixed-width, additionally returns field Csv.PotentialPositions containing a list for analysis of potential fixed width column positions.

Binary.Length

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Length(binary as nullable binary) as nullable number
```

About

Returns the number of characters.

Binary.Range

3/15/2021 • 2 minutes to read

Syntax

```
Binary.Range(binary as binary, offset as number, optional count as nullable number) as binary
```

About

Returns a subset of the binary value beginning at the offset `binary`. An optional parameter, `offset`, sets the maximum length of the subset.

Example 1

Returns a subset of the binary value starting at offset 6.

```
Binary.Range(#binary({0..10}), 6)
```

```
#binary({6, 7, 8, 9, 10})
```

Example 2

Returns a subset of length 2 from offset 6 of the binary value.

```
Binary.Range(#binary({0..10}), 6, 2)
```

```
#binary({6, 7})
```

Binary.ToList

3/15/2021 • 2 minutes to read

Syntax

```
Binary.ToList(binary as binary) as list
```

About

Converts a binary value into a list of numbers.

Binary.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Binary.ToText(binary as nullable binary, optional encoding as nullable number) as nullable text
```

About

Returns the result of converting a binary list of numbers `binary` into a text value. Optionally, `encoding` may be specified to indicate the encoding to be used in the text value produced The following `BinaryEncoding` values may be used for `encoding`.

- `BinaryEncoding.Base64`: Base 64 encoding
- `BinaryEncoding.Hex`: Hex encoding

BinaryEncoding.Base64

3/15/2021 • 2 minutes to read

About

Constant to use as the encoding type when base-64 encoding is required.

BinaryEncoding.Hex

3/15/2021 • 2 minutes to read

About

Constant to use as the encoding type when hexadecimal encoding is required.

BinaryFormat.7BitEncodedSignedInteger

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.7BitEncodedSignedInteger(binary as binary) as any
```

About

A binary format that reads a 64-bit signed integer that was encoded using a 7-bit variable-length encoding.

BinaryFormat.7BitEncodedUnsignedInteger

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.7BitEncodedUnsignedInteger(binary as binary) as any
```

About

A binary format that reads a 64-bit unsigned integer that was encoded using a 7-bit variable-length encoding.

BinaryFormat.Binary

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Binary(optional length as any) as function
```

About

Returns a binary format that reads a binary value. If `length` is specified, the binary value will contain that many bytes. If `length` is not specified, the binary value will contain the remaining bytes. The `length` can be specified either as a number, or as a binary format of the length that preceeds the binary data.

BinaryFormat.Byte

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Byte(binary as binary) as any
```

About

A binary format that reads an 8-bit unsigned integer.

BinaryFormat.ByteOrder

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.ByteOrder(binaryFormat as function, byteOrder as number) as function
```

About

Returns a binary format with the byte order specified by `binaryFormat`. The default byte order is `ByteOrder.BigEndian`.

BinaryFormat.Choice

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Choice(binaryFormat as function, chooseFunction as function, optional type as
nullable type, optional combineFunction as nullable function) as function
```

About

Returns a binary format that chooses the next binary format based on a value that has already been read. The binary format value produced by this function works in stages:

- The binary format specified by the `binaryFormat` parameter is used to read a value.
- The value is passed to the choice function specified by the `chooseFunction` parameter.
- The choice function inspects the value and returns a second binary format.
- The second binary format is used to read a second value.
- If the combine function is specified, then the first and second values are passed to the combine function, and the resulting value is returned.
- If the combine function is not specified, the second value is returned.
- The second value is returned.

The optional `type` parameter indicates the type of binary format that will be returned by the choice function. Either `type any`, `type list`, or `type binary` may be specified. If the `type` parameter is not specified, then `type any` is used. If `type list` or `type binary` is used, then the system may be able to return a streaming `binary` or `list` value instead of a buffered one, which may reduce the amount of memory necessary to read the format.

Example 1

Read a list of bytes where the number of elements is determined by the first byte.

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.List(BinaryFormat.Byte, length)
    )
in
    listFormat(binaryData)
```

```
3
```

```
4
```

Example 2

Read a list of bytes where the number of elements is determined by the first byte, and preserve the first byte read.

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.Record([
            length = length,
            list = BinaryFormat.List(BinaryFormat.Byte, length)
        ])
    )
in
    listFormat(binaryData)
```

LENGTH	2
LIST	[List]

Example 3

Read a list of bytes where the number of elements is determined by the first byte using a streaming list.

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.List(BinaryFormat.Byte, length),
        type list
    )
in
    listFormat(binaryData)
```

3
4

BinaryFormat.Decimal

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Decimal(binary as binary) as any
```

About

A binary format that reads a .NET 16-byte decimal value.

BinaryFormat.Double

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Double(binary as binary) as any
```

About

A binary format that reads an 8-byte IEEE double-precision floating point value.

BinaryFormat.Group

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Group(binaryFormat as function, group as list, optional extra as nullable function,
optional lastKey as any) as function
```

About

The parameters are as follows:

- The `binaryFormat` parameter specifies the binary format of the key value.
- The `group` parameter provides information about the group of known items.
- The optional `extra` parameter can be used to specify a function that will return a binary format value for the value following any key that was unexpected. If the `extra` parameter is not specified, then an error will be raised if there are unexpected key values.

The `group` parameter specifies a list of item definitions. Each item definition is a list, containing 3-5 values, as follows:

- Key value. The value of the key that corresponds to the item. This must be unique within the set of items.
- Item format. The binary format corresponding to the value of the item. This allows each item to have a different format.
- Item occurrence. The `BinaryOccurrence.Type` value for how many times the item is expected to appear in the group. Required items that are not present cause an error. Required or optional duplicate items are handled like unexpected key values.
- Default item value (optional). If the default item value appears in the item definition list and is not null, then it will be used instead of the default. The default for repeating or optional items is null, and the default for repeating values is an empty list { }.
- Item value transform (optional). If the item value transform function is present in the item definition list and is not null, then it will be called to transform the item value before it is returned. The transform function is only called if the item appears in the input (it will never be called with the default value).

Example 1

The following assumes a key value that is a single byte, with 4 expected items in the group, all of which have a byte of data following the key. The items appear in the input as follows:

- Key 1 is required, and does appear with value 11.
- Key 2 repeats, and appears twice with value 22, and results in a value of { 22, 22 }.
- Key 3 is optional, and does not appear, and results in a value of null.
- Key 4 repeats, but does not appear, and results in a value of { }.
- Key 5 is not part of the group, but appears once with value 55. The extra function is called with the key value 5, and returns the format corresponding to that value (BinaryFormat.Byte). The value 55 is read and discarded.

```
let
    b = #binary({
        1, 11,
        2, 22,
        2, 22,
        5, 55,
        1, 11
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Required},
            {2, BinaryFormat.Byte, BinaryOccurrence.Repeating},
            {3, BinaryFormat.Byte, BinaryOccurrence.Optional},
            {4, BinaryFormat.Byte, BinaryOccurrence.Repeating}
        },
        (extra) => BinaryFormat.Byte
    )
in
    f(b)
```

11

[List]

[List]

Example 2

The following example illustrates the item value transform and default item value. The repeating item with key 1 sums the list of values read using List.Sum. The optional item with key 2 has a default value of 123 instead of null.

```
let
    b = #binary({
        1, 101,
        1, 102
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Repeating,
              0, (list) => List.Sum(list)},
            {2, BinaryFormat.Byte, BinaryOccurrence.Optional, 123}
        }
    )
in
    f(b)
```

203

123

BinaryFormat.Length

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Length(binaryFormat as function, length as any) as function
```

About

Returns a binary format that limits the amount of data that can be read. Both `BinaryFormat.List` and `BinaryFormat.Binary` can be used to read until end of the data. `BinaryFormat.Length` can be used to limit the number of bytes that are read. The `binaryFormat` parameter specifies the binary format to limit. The `length` parameter specifies the number of bytes to read. The `length` parameter may either be a number value, or a binary format value that specifies the format of the length value that appears that precedes the value being read.

Example 1

Limit the number of bytes read to 2 when reading a list of bytes.

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        2
    )
in
    listFormat(binaryData)
```

```
1
```

```
2
```

Example 2

Limit the number of byte read when reading a list of bytes to the byte value preceding the list.

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        BinaryFormat.Byte
    )
in
    listFormat(binaryData)
```

```
2
```

BinaryFormat.List

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.List(binaryFormat as function, optional countOrCondition as any) as function
```

About

Returns a binary format that reads a sequence of items and returns a `list`. The `binaryFormat` parameter specifies the binary format of each item. There are three ways to determine the number of items read:

- If the `countOrCondition` is not specified, then the binary format will read until there are no more items.
- If the `countOrCondition` is a number, then the binary format will read that many items.
- If the `countOrCondition` is a function, then that function will be invoked for each item read. The function returns true to continue, and false to stop reading items. The final item is included in the list.
- If the countOrCondition is a binary format, then the count of items is expected to precedes the list, and the specified format is used to read the count.

Example 1

Read bytes until the end of the data.

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte)
in
    listFormat(binaryData)
```

1
2
3

Example 2

Read two bytes.

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, 2)
in
    listFormat(binaryData)
```

1

```
2
```

Example 3

Read bytes until the byte value is greater than or equal to two.

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, (x) => x < 2)
in
    listFormat(binaryData)
```

```
1
```

```
2
```

BinaryFormat.Null

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Null(binary as binary) as any
```

About

A binary format that reads zero bytes and returns null.

BinaryFormat.Record

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Record(record as record) as function
```

About

Returns a binary format that reads a record. The `record` parameter specifies the format of the record. Each field in the record can have a different binary format. If a field contains a value that is not a binary format value, then no data is read for that field, and the field value is echoed to the result.

Example 1

Read a record containing one 16-bit integer and one 32-bit integer.

```
let
    binaryData = #binary({
        0x00, 0x01,
        0x00, 0x00, 0x00, 0x02
    }),
    recordFormat = BinaryFormat.Record([
        A = BinaryFormat.UnsignedInteger16,
        B = BinaryFormat.UnsignedInteger32
    ])
in
    recordFormat(binaryData)
```

A	1
B	2

BinaryFormat.SignedInteger16

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.SignedInteger16(binary as binary) as any
```

About

A binary format that reads a 16-bit signed integer.

BinaryFormat.SignedInteger32

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.SignedInteger32(binary as binary) as any
```

About

A binary format that reads a 32-bit signed integer.

BinaryFormat.SignedInteger64

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.SignedInteger64(binary as binary) as any
```

About

A binary format that reads a 64-bit signed integer.

BinaryFormat.Single

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Single(binary as binary) as any
```

About

A binary format that reads a 4-byte IEEE single-precision floating point value.

BinaryFormat.Text

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Text(length as any, optional encoding as nullable number) as function
```

About

Returns a binary format that reads a text value. The `length` specifies the number of bytes to decode, or the binary format of the length that precedes the text. The optional `encoding` value specifies the encoding of the text. If the `encoding` is not specified, then the encoding is determined from the Unicode byte order marks. If no byte order marks are present, then `TextEncoding.Utf8` is used.

Example 1

Decode two bytes as ASCII text.

```
let
    binaryData = #binary({65, 66, 67}),
    textFormat = BinaryFormat.Text(2, TextEncoding.Ascii)
in
    textFormat(binaryData)
```

`"AB"`

Example 2

Decode ASCII text where the length of the text in bytes appears before the text as a byte.

```
let
    binaryData = #binary({2, 65, 66}),
    textFormat = BinaryFormat.Text(
        BinaryFormat.Byte,
        TextEncoding.Ascii
    )
in
    textFormat(binaryData)
```

`"AB"`

BinaryFormat.Transform

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.Transform(binaryFormat as function, function as function) as function
```

About

Returns a binary format that will transform the values read by another binary format. The `binaryFormat` parameter specifies the binary format that will be used to read the value. The `function` is invoked with the value read, and returns the transformed value.

Example 1

Read a byte and add one to it.

```
let
    binaryData = #binary({1}),
    transformFormat = BinaryFormat.Transform(
        BinaryFormat.Byte,
        (x) => x + 1
    )
in
    transformFormat(binaryData)
```

`2`

BinaryFormat.UnsignedInteger16

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.UnsignedInteger16(binary as binary) as any
```

About

A binary format that reads a 16-bit unsigned integer.

BinaryFormat.UnsignedInteger32

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.UnsignedInteger32(binary as binary) as any
```

About

A binary format that reads a 32-bit unsigned integer.

BinaryFormat.UnsignedInteger64

3/15/2021 • 2 minutes to read

Syntax

```
BinaryFormat.UnsignedInteger64(binary as binary) as any
```

About

A binary format that reads a 64-bit unsigned integer.

BinaryOccurrence.Optional

3/15/2021 • 2 minutes to read

About

The item is expected to appear zero or one time in the input.

BinaryOccurrence.Repeating

3/15/2021 • 2 minutes to read

About

The item is expected to appear zero or more times in the input.

BinaryOccurrence.Required

3/15/2021 • 2 minutes to read

About

The item is expected to appear once in the input.

ByteOrder.BigEndian

3/15/2021 • 2 minutes to read

About

A possible value for the `byteOrder` parameter in `BinaryFormat.ByteOrder`. The most significant byte appears first in Big Endian byte order.

ByteOrder.LittleEndian

3/15/2021 • 2 minutes to read

About

A possible value for the `byteOrder` parameter in `BinaryFormat.ByteOrder`. The least significant byte appears first in Little Endian byte order.

Compression.Brotli

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'Brotli' format.

Compression.Deflate

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'Deflate' format.

Compression.GZip

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'GZip' format.

Compression.LZ4

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'LZ4' format.

Compression.None

3/15/2021 • 2 minutes to read

About

The data is uncompressed.

Compression.Snappy

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'Snappy' format.

Compression.Zstandard

3/15/2021 • 2 minutes to read

About

The compressed data is in the 'Zstandard' format.

Occurrence.Optional

3/15/2021 • 2 minutes to read

About

The item is expected to appear zero or one time in the input.

Occurrence.Repeating

3/15/2021 • 2 minutes to read

About

The item is expected to appear zero or more times in the input.

Occurrence.Required

3/15/2021 • 2 minutes to read

About

The item is expected to appear once in the input.

#binary

3/15/2021 • 2 minutes to read

Syntax

```
#binary(value as any) as any
```

About

Creates a binary value from a list of numbers or a base 64 encoded text value.

Example 1

Create a binary value from a list of numbers.

```
#binary({0x30, 0x31, 0x32})
```

`Text.ToBinary("012")`

Example 2

Create a binary value from a base 64 encoded text value.

```
#binary("1011")
```

`Binary.FromText("1011", BinaryEncoding.Base64)`

Combiner functions

3/15/2021 • 2 minutes to read

These functions are used by other library functions that merge values. For example, `Table.ToList` and `Table.CombineColumns` apply a combiner function to each row in a table to produce a single value for each row.

Combiner

FUNCTION	DESCRIPTION
Combiner.CombineTextByDelimiter	Returns a function that combines a list of text into a single text using the specified delimiter.
Combiner.CombineTextByEachDelimiter	Returns a function that combines a list of text into a single text using each specified delimiter in sequence.
Combiner.CombineTextByLengths	Returns a function that combines a list of text into a single text using the specified lengths.
Combiner.CombineTextByPositions	Returns a function that combines a list of text into a single text using the specified positions.
Combiner.CombineTextByRanges	Returns a function that combines a list of text into a single text using the specified positions and lengths.

Combiner.CombineTextByDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Combiner.CombineTextByDelimiter(delimiter as text, optional quoteStyle as nullable number) as
function
```

About

Returns a function that combines a list of text into a single text using the specified delimiter.

Combiner.CombineTextByEachDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Combiner.CombineTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number)
as function
```

About

Returns a function that combines a list of text into a single text using each specified delimiter in sequence.

Combiner.CombineTextByLengths

3/15/2021 • 2 minutes to read

Syntax

```
Combiner.CombineTextByLengths(lengths as list, optional template as nullable text) as function
```

About

Returns a function that combines a list of text into a single text using the specified lengths.

Combiner.CombineTextByPositions

3/15/2021 • 2 minutes to read

Syntax

```
Combiner.CombineTextByPositions(positions as list, optional template as nullable text) as
function
```

About

Returns a function that combines a list of text into a single text using the specified positions.

Combiner.CombineTextByRanges

3/15/2021 • 2 minutes to read

Syntax

```
Combiner.CombineTextByRanges(ranges as list, optional template as nullable text) as function
```

About

Returns a function that combines a list of text into a single text using the specified positions and lengths.

Comparer functions

3/15/2021 • 2 minutes to read

These functions test equality and determine ordering.

Comparer

FUNCTION	DESCRIPTION
Comparer.Equals	Returns a logical value based on the equality check over the two given values.
Comparer.FromCulture	Returns a comparer function given the culture and a logical value for case sensitivity for the comparison. The default value for ignoreCase is false. The value for culture are well known text representations of locales used in the .NET framework.
Comparer.Ordinal	Returns a comparer function which uses Ordinal rules to compare values.
Comparer.OrdinalIgnoreCase	Returns a case-insensitive comparer function which uses Ordinal rules to compare the provided values x and y.
Culture.Current	Returns the current culture of the system.

Comparer.Equals

3/15/2021 • 2 minutes to read

Syntax

```
Comparer.Equals(comparer as function, x as any, y as any) as logical
```

About

Returns a `logical` value based on the equality check over the two given values, `x` and `y`, using the provided `comparer`.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Compare "1" and "A" using "en-US" locale to determine if the values are equal.

```
Comparer.Equals(Comparer.FromCulture("en-us"), "1", "A")
```

`false`

Comparer.FromCulture

3/15/2021 • 2 minutes to read

Syntax

```
Comparer.FromCulture(culture as text, optional ignoreCase as nullable logical) as function
```

About

Returns a comparer function given the `culture` and a logical value `ignoreCase` for case sensitivity for the comparison. The default value for `ignoreCase` is false. The value for culture are well known text representations of locales used in the .NET framework.

Example 1

Compare "a" and "A" using "en-US" locale to determine if the values are equal.

```
Comparer.FromCulture("en-us")("a", "A")
```

`-1`

Example 2

Compare "a" and "A" using "en-US" locale ignoring the case to determine if the values are equal.

```
Comparer.FromCulture("en-us", true)("a", "A")
```

`0`

Comparer.Ordinal

3/15/2021 • 2 minutes to read

Syntax

```
Comparer.Ordinal(x as any, y as any) as number
```

About

Returns a comparer function which uses Ordinal rules to compare the provided values `x` and `y`.

Example 1

Using Ordinal rules, compare if "encyclopædia" and "encyclopaedia" are equivalent. Note these are equivalent using `Comparer.FromCulture("en-us")`.

```
Comparer.Equals(Comparer.Ordinal, "encyclopædia", "encyclopaedia")
```

`false`

Comparer.OrdinalIgnoreCase

3/15/2021 • 2 minutes to read

Syntax

```
Comparer.OrdinalIgnoreCase(x as any, y as any) as number
```

About

Returns a case-insensitive comparer function which uses Ordinal rules to compare the provided values `x` and `y`.

Example

Using case-insensitive Ordinal rules, compare "Abc" with "abc". Note "Abc" is less than "abc" using `Comparer.Ordinal`.

```
Comparer.OrdinalIgnoreCase("Abc", "abc")
```

```
0
```

Culture.Current

3/15/2021 • 2 minutes to read

About

Returns the name of the current culture for the application.

Date functions

3/15/2021 • 5 minutes to read

These functions create and manipulate the date component of date, datetime, and datetimezone values.

Date

FUNCTION	DESCRIPTION
Date.AddDays	Returns a Date/DateTime/DateTimeZone value with the day portion incremented by the number of days provided. It also handles incrementing the month and year potions of the value as appropriate.
Date.AddMonths	Returns a DateTime value with the month portion incremented by n months.
Date.AddQuarters	Returns a Date/DateTime/DateTimeZone value incremented by the number of quarters provided. Each quarter is defined as a duration of three months. It also handles incrementing the year potion of the value as appropriate.
Date.AddWeeks	Returns a Date/DateTime/DateTimeZone value incremented by the number of weeks provided. Each week is defined as a duration of seven days. It also handles incrementing the month and year potions of the value as appropriate.
Date.AddYears	Returns a DateTime value with the year portion incremented by n years.
Date.Day	Returns the day for a DateTime value.
Date.DayOfWeek	Returns a number (from 0 to 6) indicating the day of the week of the provided value.
Date.DayOfWeekName	Returns the day of the week name.
Date.DayOfYear	Returns a number that represents the day of the year from a DateTime value.
Date.DaysInMonth	Returns the number of days in the month from a DateTime value.
Date.EndOfDay	Returns a DateTime value for the end of the day.
Date.EndOfMonth	Returns a DateTime value for the end of the month.
Date.EndOfQuarter	Returns a Date/DateTime/DateTimeZone value representing the end of the quarter. The date and time portions are reset to their terminating values for the quarter. The timezone information is persisted.

FUNCTION	DESCRIPTION
Date.EndOfWeek	Returns a DateTime value for the end of the week.
Date.EndOfYear	Returns a DateTime value for the end of the year.
Date.From	Returns a date value from a value.
Date.FromText	Returns a Date value from a set of date formats and culture value.
Date.IsInCurrentDay	Indicates whether the given datetime value `dateTime` occurs during the current day, as determined by the current date and time on the system.
Date.IsInCurrentMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current month, as determined by the current date and time on the system.
Date.IsInCurrentQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current quarter, as determined by the current date and time on the system.
Date.IsInCurrentWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current week, as determined by the current date and time on the system.
Date.IsInCurrentYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current year, as determined by the current date and time on the system.
Date.IsInNextDay	Indicates whether the given datetime value dateTime occurs during the next day, as determined by the current date and time on the system.
Date.IsInNextMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next month, as determined by the current date and time on the system.
Date.IsInNextNDays	Indicates whether the given datetime value dateTime occurs during the next number of days, as determined by the current date and time on the system.
Date.IsInNextNMonths	Indicates whether the given datetime value dateTime occurs during the next number of months, as determined by the current date and time on the system.
Date.IsInNextNQuarters	Indicates whether the given datetime value dateTime occurs during the next number of quarters, as determined by the current date and time on the system.

FUNCTION	DESCRIPTION
Date.IsInNextNWeeks	Indicates whether the given datetime value dateTime occurs during the next number of weeks, as determined by the current date and time on the system.
Date.IsInNextNYears	Indicates whether the given datetime value dateTime occurs during the next number of years, as determined by the current date and time on the system.
Date.IsInNextQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next quarter, as determined by the current date and time on the system.
Date.IsInNextWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next week, as determined by the current date and time on the system.
Date.IsInNextYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next year, as determined by the current date and time on the system.
Date.IsInPreviousDay	Indicates whether the given datetime value dateTime occurs during the previous day, as determined by the current date and time on the system.
Date.IsInPreviousMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous month, as determined by the current date and time on the system.
Date.IsInPreviousNDays	Indicates whether the given datetime value dateTime occurs during the previous number of days, as determined by the current date and time on the system.
Date.IsInPreviousNMonths	Indicates whether the given datetime value dateTime occurs during the previous number of months, as determined by the current date and time on the system.
Date.IsInPreviousNQuarters	Indicates whether the given datetime value dateTime occurs during the previous number of quarters, as determined by the current date and time on the system.
Date.IsInPreviousNWeeks	Indicates whether the given datetime value dateTime occurs during the previous number of weeks, as determined by the current date and time on the system.
Date.IsInPreviousNYears	Indicates whether the given datetime value dateTime occurs during the previous number of years, as determined by the current date and time on the system.
Date.IsInPreviousQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous quarter, as determined by the current date and time on the system.

FUNCTION	DESCRIPTION
Date.IsInPreviousWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous week, as determined by the current date and time on the system.
Date.IsInPreviousYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous year, as determined by the current date and time on the system.
Date.IsInYearToDate	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred in the period starting January 1st of the current year and ending on the current day, as determined by the current date and time on the system.
Date.IsLeapYear	Returns a logical value indicating whether the year portion of a DateTime value is a leap year.
Date.Month	Returns the month from a DateTime value.
Date.MonthName	Returns the name of the month component.
Date.QuarterOfYear	Returns a number between 1 and 4 for the quarter of the year from a DateTime value.
Date.StartOfDay	Returns a DateTime value for the start of the day.
Date.StartOfMonth	Returns a DateTime value representing the start of the month.
Date.StartOfQuarter	Returns a DateTime value representing the start of the quarter.
Date.StartOfWeek	Returns a DateTime value representing the start of the week.
Date.StartOfYear	Returns a DateTime value representing the start of the year.
Date.ToRecord	Returns a record containing parts of a Date value.
Date.ToText	Returns a text value from a Date value.
Date.WeekOfMonth	Returns a number for the count of week in the current month.
Date.WeekOfYear	Returns a number for the count of week in the current year.
Date.Year	Returns the year from a DateTime value.
#date	Creates a date value from year, month, and day.

PARAMETER VALUES	DESCRIPTION
Day.Sunday	Represents Sunday.
Day.Monday	Represents Monday.
Day.Tuesday	Represents Tuesday.
Day.Wednesday	Represents Wednesday.
Day.Thursday	Represents Thursday.
Day.Friday	Represents Friday.
Day.Saturday	Represents Saturday.

Date.AddDays

3/15/2021 • 2 minutes to read

Syntax

```
Date.AddDays(dateTime as any, numberOfDays as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfDays` days to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which days are being added.
- `numberOfDays`: The number of days to add.

Example 1

Add 5 days to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

```
Date.AddDays(#date(2011, 5, 14), 5)
```

```
#date(2011, 5, 19)
```

Date.AddMonths

3/15/2021 • 2 minutes to read

Syntax

```
Date.AddMonths(dateTime as any, numberOfMonths as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfMonths` months to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which months are being added.
- `numberOfMonths`: The number of months to add.

Example 1

Add 5 months to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

```
Date.AddMonths(#date(2011, 5, 14), 5)
```

```
#date(2011, 10, 14)
```

Example 2

Add 18 months to the `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 08:15:22 AM.

```
Date.AddMonths(#datetime(2011, 5, 14, 8, 15, 22), 18)
```

```
#datetime(2012, 11, 14, 8, 15, 22)
```

Date.AddQuarters

3/15/2021 • 2 minutes to read

Syntax

```
Date.AddQuarters(dateTime as any, numberOfQuarters as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfQuarters` quarters to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which quarters are being added.
- `numberOfQuarters`: The number of quarters to add.

Example 1

Add 1 quarter to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

```
Date.AddQuarters(#date(2011, 5, 14), 1)
```

```
#date(2011, 8, 14)
```

Date.AddWeeks

3/15/2021 • 2 minutes to read

Syntax

```
Date.AddWeeks(dateTime as any, numberOfWeeks as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfWeeks` weeks to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which weeks are being added.
- `numberOfWeeks`: The number of weeks to add.

Example 1

Add 2 weeks to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

```
Date.AddWeeks(#date(2011, 5, 14), 2)
```

`#date(2011, 5, 28)`

Date.AddYears

3/15/2021 • 2 minutes to read

Syntax

```
Date.AddYears(dateTime as any, numberOfYears as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result of adding `numberOfYears` to a `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which years are added.
- `numberOfYears`: The number of years to add.

Example 1

Add 4 years to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

```
Date.AddYears(#date(2011, 5, 14), 4)
```

```
#date(2015, 5, 14)
```

Example 2

Add 10 years to the `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 08:15:22 AM.

```
Date.AddYears(#datetime(2011, 5, 14, 8, 15, 22), 10)
```

```
#datetime(2021, 5, 14, 8, 15, 22)
```

Date.Day

3/15/2021 • 2 minutes to read

Syntax

```
Date.Day(dateTime as any) as nullable number
```

About

Returns the day component of a `date`, `datetime`, or `datetimezone` value.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the day component is extracted.

Example 1

Get the day component of a `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 05:00:00 PM.

```
Date.Day(#datetime(2011, 5, 14, 17, 0, 0))
```

`14`

Date.DayOfWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.DayOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number (from 0 to 6) indicating the day of the week of the provided `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value.
- `firstDayOfWeek`: A `Day` value indicating which day should be considered the first day of the week. Allowed values are Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday, Day.Thursday, Day.Friday, or Day.Saturday. If unspecified, a culture-dependent default is used.

Example 1

Get the day of the week represented by Monday, February 21st, 2011, treating Sunday as the first day of the week.

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Sunday)`
```

`1`

Example 2

Get the day of the week represented by Monday, February 21st, 2011, treating Monday as the first day of the week.

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Monday)
```

`0`

Date.DayOfWeekName

3/15/2021 • 2 minutes to read

Syntax

```
Date.DayOfWeekName(date as any, optional culture as nullable text)
```

About

Returns the day of the week name for the provided `date`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the day of the week name.

```
Date.DayOfWeekName(#date(2011, 12, 31), "en-US")
```

`"Saturday"`

Date.DayOfYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.DayOfYear(dateTime as any) as nullable number
```

About

Returns a number representing the day of the year in the provided `date`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

The number of the day March 1st, 2011 (`#date(2011, 03, 01)`).

```
Date.DayOfYear(#date(2011, 03, 01))
```

`60`

Date.DaysInMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.DaysInMonth(dateTime as any) as nullable number
```

About

Returns the number of days in the month in the `date`, `datetime`, or `datetimezone` value `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value for which the number of days in the month is returned.

Example 1

Number of days in the month December as represented by `#date(2011, 12, 01)`.

```
Date.DaysInMonth(#date(2011, 12, 01))
```

`31`

Date.EndOfDay

3/15/2021 • 2 minutes to read

```
Date.EndOfDay(dateTime as any) as any
```

About

Returns a `date`, `datetime`, or `datetimezone` value representing the end of the day in `dateTime`. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from from which the end of the day is calculated.

Example 1

Get the end of the day for 5/14/2011 05:00:00 PM.

```
Date.EndOfDay(#datetime(2011, 5, 14, 17, 0, 0))
```

```
#datetime(2011, 5, 14, 23, 59, 59.9999999)
```

Example 2

Get the end of the day for 5/17/2011 05:00:00 PM -7:00.

```
Date.EndOfDay(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 5, 17, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.EndOfMonth(dateTime as any) as any
```

About

Returns the last day of the month in `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the month is calculated

Example 1

Get the end of the month for 5/14/2011.

```
Date.EndOfMonth(#date(2011, 5, 14))
```

```
#date(2011, 5, 31)
```

Example 2

Get the end of the month for 5/17/2011 05:00:00 PM -7:00.

```
Date.EndOfMonth(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 5, 31, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfQuarter

3/15/2021 • 2 minutes to read

Syntax

```
Date.EndOfQuarter(dateTime as any) as any
```

About

Returns a `date`, `datetime`, or `datetimezone` value representing the end of the quarter in `dateTime`. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the quarter is calculated.

Example 1

Find the end of the quarter for October 10th, 2011, 8:00AM (`#datetime(2011, 10, 10, 8, 0, 0)`).

```
Date.EndOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

```
#datetime(2011, 12, 31, 23, 59, 59.9999999)
```

Date.EndOfWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.EndOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

About

Returns the last day of the week in the provided `date`, `datetime`, or `datetimezone` `dateTime`. This function takes an optional `Day`, `firstDayOfWeek`, to set the first day of the week for this relative calculation. The default value is `Day.Sunday`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the last day of the week is calculated
- `firstDayOfWeek`: *[Optional]* A `Day.Type` value representing the first day of the week. Possible values are `Day.Sunday`, `Day.Monday`, `Day.Tuesday`, `Day.Wednesday`, `Day.Thursday`, `Day.Friday` and `Day.Saturday.`. The default value is `Day.Sunday`.

Example 1

Get the end of the week for 5/14/2011.

```
Date.EndOfWeek(#date(2011, 5, 14))
```

```
#date(2011, 5, 14)
```

Example 2

Get the end of the week for 5/17/2011 05:00:00 PM -7:00, with Sunday as the first day of the week.

```
Date.EndOfWeek(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0), Day.Sunday)
```

```
#datetimezone(2011, 5, 21, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.EndOfYear(dateTime as any) as any
```

About

Returns a value representing the end of the year in `dateTime`, including fractional seconds. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the year is calculated.

Example 1

Get the end of the year for 5/14/2011 05:00:00 PM.

```
Date.EndOfYear(#datetime(2011, 5, 14, 17, 0, 0))
```

```
#datetime(2011, 12, 31, 23, 59, 59.9999999)
```

Example 2

Get the end of hour for 5/17/2011 05:00:00 PM -7:00.

```
Date.EndOfYear(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 12, 31, 23, 59, 59.9999999, -7, 0)
```

Date.From

3/15/2021 • 2 minutes to read

Syntax

```
Date.From(value as any, optional culture as nullable text) as nullable date
```

About

Returns a `date` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, `Date.From` returns `null`. If the given `value` is `date`, `value` is returned. Values of the following types can be converted to a `date` value:

- `text`: A `date` value from textual representation. See `Date.FromText` for details.
- `datetime`: The date component of the `value`.
- `datetimezone`: The date component of the local datetime equivalent of `value`.
- `number`: The date component of the datetime equivalent the OLE Automation Date expressed by `value`.

If `value` is of any other type, an error is returned.

Example 1

Convert `43910` to a `date` value.

```
Date.From(43910)
```

```
#date(2020, 3, 20)
```

Example 2

Convert `#datetime(1899, 12, 30, 06, 45, 12)` to a `date` value.

```
Date.From(#datetime(1899, 12, 30, 06, 45, 12))
```

```
#date(1899, 12, 30)
```

Date.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Date.FromText(text as nullable text, optional culture as nullable text) as nullable date
```

About

Creates a `date` value from a textual representation, `text`, following ISO 8601 format standard. An optional `culture` may also be provided (for example, "en-US").

- `Date.FromText("2010-02-19")` // Date, yyyy-MM-dd

Example 1

Convert `"December 31, 2010"` into a date value.

```
Date.FromText("2010-12-31")
```

`#date(2010, 12, 31)`

Example 2

Convert `"December 31, 2010"` into a date value, with a different format

```
Date.FromText("2010, 12, 31")
```

`#date(2010, 12, 31)`

Example 3

Convert `"December, 2010"` into a date value.

```
Date.FromText("2010, 12")
```

`#date(2010, 12, 1)`

Example 4

Convert `"2010"` into a date value.

```
Date.FromText("2010")
```

`#date(2010, 1, 1)`

Date.IsInCurrentDay

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInCurrentDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current day, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example

Determine if the current system time is in the current day.

```
Date.IsInCurrentDay(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInCurrentMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current month, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current month.

```
Date.IsInCurrentMonth(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentQuarter

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInCurrentQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current quarter, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current quarter.

```
Date.IsInCurrentQuarter(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInCurrentWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current week, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current week.

```
Date.IsInCurrentWeek(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInCurrentYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current year, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current year.

```
Date.IsInCurrentYear(DateTime.FixedLocalNow())
```

`true`

Date.IsInNextDay

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next day, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the day after the current system time is in the next day.

```
Date.IsInNextDay(Date.AddDays(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next month, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the month after the current system time is in the next month.

```
Date.IsInNextMonth(Date.AddMonths(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextNDays

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextNDays(dateTime as any, days as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of days, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `days`: The number of days.

Example 1

Determine if the day after the current system time is in the next two days.

```
Date.IsInNextNDays(Date.AddDays(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNMonths

3/15/2021 • 2 minutes to read

```
Date.IsInNextNMonths(dateTime as any, months as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of months, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `months`: The number of months.

Example 1

Determine if the month after the current system time is in the next two months.

```
Date.IsInNextNMonths(Date.AddMonths(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNQuarters

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextNQuarters(dateTime as any, quarters as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of quarters, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.
- `quarters` : The number of quarters.

Example 1

Determine if the quarter after the current system time is in the next two quarters.

```
Date.IsInNextNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNWeeks

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextNWeeks(dateTime as any, weeks as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of weeks, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `weeks`: The number of weeks.

Example 1

Determine if the week after the current system time is in the next two weeks.

```
Date.IsInNextNWeeks(Date.AddDays(DateTime.FixedLocalNow(), 7), 2)
```

`true`

Date.IsInNextNYears

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextNYears(dateTime as any, years as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of years, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `years`: The number of years.

Example 1

Determine if the year after the current system time is in the next two years.

```
Date.IsInNextNYears(Date.AddYears(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextQuarter

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next quarter, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

####Example 1 Determine if the quarter after the current system time is in the next quarter.

```
Date.IsInNextQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next week, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the week after the current system time is in the next week.

```
Date.IsInNextWeek(Date.AddDays(DateTime.FixedLocalNow(), 7))
```

`true`

Date.IsInNextYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInNextYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next year, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the year after the current system time is in the next year.

```
Date.IsInNextYear(Date.AddYears(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInPreviousDay

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous day, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the day before the current system time is in the previous day.

```
Date.IsInPreviousDay(Date.AddDays(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous month, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the month before the current system time is in the previous month.

```
Date.IsInPreviousMonth(Date.AddMonths(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousNDays

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousNDays(dateTime as any, days as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of days, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `days`: The number of days.

Example 1

Determine if the day before the current system time is in the previous two days.

```
Date.IsInPreviousNDays(Date.AddDays(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNMonths

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousNMonths(dateTime as any, months as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of months, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `months`: The number of months.

Example 1

Determine if the month before the current system time is in the previous two months.

```
Date.IsInPreviousNMonths(Date.AddMonths(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNQuarters

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousNQuarters(dateTime as any, quarters as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of quarters, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.
- `quarters` : The number of quarters.

Example 1

Determine if the quarter before the current system time is in the previous two quarters.

```
Date.IsInPreviousNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNWeeks

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousNWeeks(dateTime as any, weeks as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of weeks, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `weeks`: The number of weeks.

Example 1

Determine if the week before the current system time is in the previous two weeks.

```
Date.IsInPreviousNWeeks(Date.AddDays(DateTime.FixedLocalNow(), -7), 2)
```

`true`

Date.IsInPreviousNYears

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousNYears(dateTime as any, years as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of years, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `years`: The number of years.

Example 1

Determine if the year before the current system time is in the previous two years.

```
Date.IsInPreviousNYears(Date.AddYears(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousQuarter

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous quarter, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the quarter before the current system time is in the previous quarter.

```
Date.IsInPreviousQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous week, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the week before the current system time is in the previous week.

```
Date.IsInPreviousWeek(Date.AddDays(DateTime.FixedLocalNow(), -7))
```

`true`

Date.IsInPreviousYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInPreviousYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous year, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the year before the current system time is in the previous year.

```
Date.IsInPreviousYear(Date.AddYears(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInYearToDate

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsInYearToDate(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current year and is on or before the current day, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the year to date.

```
Date.IsInYearToDate(DateTime.FixedLocalNow())
```

`true`

Date.IsLeapYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.IsLeapYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` falls in is a leap year.

- `dateTime` : A `date` , `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the year 2012, as represented by `#date(2012, 01, 01)` is a leap year.

```
Date.IsLeapYear(#date(2012, 01, 01))
```

`true`

Date.Month

3/15/2021 • 2 minutes to read

Syntax

```
Date.Month(dateTime as any) as nullable number
```

About

Returns the month component of the provided `datetime` value, `dateTime`.

Example 1

Find the month in #datetime(2011, 12, 31, 9, 15, 36).

```
Date.Month(#datetime(2011, 12, 31, 9, 15, 36))
```

`12`

Date.MonthName

3/15/2021 • 2 minutes to read

Syntax

```
Date.MonthName(date as any, optional culture as nullable text) as nullable text
```

About

Returns the name of the month component for the provided `date`. An optional `culture` may also be provided (for example, "en-US").

Example

Get the month name.

```
Date.MonthName(#datetime(2011, 12, 31, 5, 0, 0), "en-US")
```

```
"December"
```

Date.QuarterOfYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.QuarterOfYear(dateTime as any) as nullable number
```

About

Returns a number from 1 to 4 indicating which quarter of the year the date `dateTime` falls in. `dateTime` can be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find which quarter of the year the date #date(2011, 12, 31) falls in.

```
Date.QuarterOfYear(#date(2011, 12, 31))
```

```
4
```

Date.StartOfDay

3/15/2021 • 2 minutes to read

Syntax

```
Date.StartOfDay(dateTime as any) as any
```

About

Returns the first value of the day `dateTime`. `dateTime` must be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the day for October 10th, 2011, 8:00AM (`#datetime(2011, 10, 10, 8, 0, 0)`).

```
Date.StartOfDay(#datetime(2011, 10, 10, 8, 0, 0))
```

`#datetime(2011, 10, 10, 0, 0, 0)`

Date.StartOfMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.StartOfMonth(dateTime as any) as any
```

About

Returns the first value of the month given a `date` or `datetime` type.

Example 1

Find the start of the month for October 10th, 2011, 8:10:32AM (`#datetime(2011, 10, 10, 8, 10, 32)`).

```
Date.StartOfMonth(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 10, 1, 0, 0, 0)`

Date.StartOfQuarter

3/15/2021 • 2 minutes to read

Syntax

```
Date.StartOfQuarter(dateTime as any) as any
```

About

Returns the first value of the quarter < `dateTime` . `dateTime` must be a `date` , `datetime` , or `datetimezone` value.

Example 1

Find the start of the quarter for October 10th, 2011, 8:00AM (`#datetime(2011, 10, 10, 8, 0, 0)`).

```
Date.StartOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

`#datetime(2011, 10, 1, 0, 0, 0)`

Date.StartOfWeek

3/15/2021 • 2 minutes to read

Syntax

```
Date.StartOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

About

Returns the first value of the week given a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the week for October 10th, 2011, 8:10:32AM (`#datetime(2011, 10, 10, 8, 10, 32)`).

```
Date.StartOfWeek(#datetime(2011, 10, 10, 8, 10, 32))
```

```
#datetime(2011, 10, 9, 0, 0, 0)
```

Date.StartOfYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.StartOfYear(dateTime as any) as any
```

About

Returns the first value of the year given a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the year for October 10th, 2011, 8:10:32AM (`#datetime(2011, 10, 10, 8, 10, 32)`).

```
Date.StartOfYear(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 1, 1, 0, 0, 0)`

Date.ToRecord

3/15/2021 • 2 minutes to read

Syntax

```
Date.ToRecord(date as date) as record
```

About

Returns a record containing the parts of the given date value, `date`.

- `date`: A `date` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#date(2011, 12, 31)` value into a record containing parts from the date value.

```
Date.ToRecord(#date(2011, 12, 31))
```

YEAR	2011
MONTH	12
DAY	31

Date.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Date.ToText(date as nullable date, optional format as nullable text, optional culture as
nullable text) as nullable text
```

About

Returns a textual representation of `date`. An optional `format` may be provided to customize the formatting of the text. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get a textual representation of #date(2010, 12, 31).

```
Date.ToText(#date(2010, 12, 31))
```

`"12/31/2010"`

Example 2

Get a textual representation of #date(2010, 12, 31) with format option.

```
Date.ToText(#date(2010, 12, 31), "yyyy/MM/dd")
```

`"2010/12/31"`

Date.WeekOfMonth

3/15/2021 • 2 minutes to read

Syntax

```
Date.WeekOfMonth(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number from 1 to 5 indicating which week of the year month the date `dateTime` falls in.

- `dateTime`: A `datetime` value for which the week-of-the-month is determined.

Example 1

Determine which week of March the 15th falls on in 2011 (`#date(2011, 03, 15)`).

```
Date.WeekOfMonth(#date(2011, 03, 15))
```

`3`

Date.WeekOfYear

3/15/2021 • 2 minutes to read

Syntax

```
Date.WeekOfYear(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number from 1 to 54 indicating which week of the year the date, `dateTime`, falls in.

- `dateTime`: A `datetime` value for which the week-of-the-year is determined.
- `firstDayOfWeek`: An optional `Day.Type` value that indicates which day is considered the start of a new week (for example, `Day.Sunday`. If unspecified, a culture-dependent default is used.

Example 1

Determine which week of the year March 27th, 2011 falls in (`#date(2011, 03, 27)`).

```
Date.WeekOfYear(#date(2011, 03, 27))
```

`14`

Example 2

Determine which week of the year March 27th, 2011 falls in (`#date(2011, 03, 27)`), using Monday as the start of a new week.

```
Date.WeekOfYear(#date(2011, 03, 27), Day.Monday)
```

`13`

Date.Year

3/15/2021 • 2 minutes to read

```
Date.Year(dateTime as any) as nullable number
```

About

Returns the year component of the provided `datetime` value, `dateTime`.

Example 1

Find the year in #datetime(2011, 12, 31, 9, 15, 36).

```
Date.Year(#datetime(2011, 12, 31, 9, 15, 36))
```

`2011`

Day.Friday

3/15/2021 • 2 minutes to read

About

Returns 6, the number representing Friday.

Day.Monday

3/15/2021 • 2 minutes to read

About

Returns 2, the number representing Monday.

Day.Saturday

3/15/2021 • 2 minutes to read

About

Returns 7, the number representing Saturday.

Day.Sunday

3/15/2021 • 2 minutes to read

About

Returns 1, the number representing Sunday.

Day.Thursday

3/15/2021 • 2 minutes to read

About

Returns 5, the number representing Thursday.

Day.Tuesday

3/15/2021 • 2 minutes to read

About

Returns 3, the number representing Tuesday.

Day.Wednesday

3/15/2021 • 2 minutes to read

About

Returns 4, the number representing Wednesday.

#date

6/22/2021 • 2 minutes to read

Syntax

```
#date(year as number, month as number, day as number) as date
```

About

Creates a date value from whole numbers representing the year, month, and day. Raises an error if these conditions are not true:

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31

DateTime functions

3/15/2021 • 2 minutes to read

These functions create and manipulate datetime and datetimezone values.

DateTime

FUNCTION	DESCRIPTION
DateTime.AddZone	Adds the timezonehours as an offset to the input datetime value and returns a new datetimezone value.
DateTime.Date	Returns a date part from a DateTime value
DateTime.FixedLocalNow	Returns a DateTime value set to the current date and time on the system.
DateTime.From	Returns a datetime value from a value.
DateTime.FromFileTime	Returns a DateTime value from the supplied number.
DateTime.FromText	Returns a DateTime value from a set of date formats and culture value.
DateTime.IsInCurrentHour	Indicates whether the given datetime value occurs during the current hour, as determined by the current date and time on the system.
DateTime.IsInCurrentMinute	Indicates whether the given datetime value occurs during the current minute, as determined by the current date and time on the system.
DateTime.IsInCurrentSecond	Indicates whether the given datetime value occurs during the current second, as determined by the current date and time on the system.
DateTime.IsInNextHour	Indicates whether the given datetime value occurs during the next hour, as determined by the current date and time on the system.
DateTime.IsInNextMinute	Indicates whether the given datetime value occurs during the next minute, as determined by the current date and time on the system.
DateTime.IsInNextNHours	Indicates whether the given datetime value occurs during the next number of hours, as determined by the current date and time on the system.
DateTime.IsInNextNMinutes	Indicates whether the given datetime value occurs during the next number of minutes, as determined by the current date and time on the system.

FUNCTION	DESCRIPTION
DateTime.IsInNextNSeconds	Indicates whether the given datetime value occurs during the next number of seconds, as determined by the current date and time on the system.
DateTime.IsInNextSecond	Indicates whether the given datetime value occurs during the next second, as determined by the current date and time on the system.
DateTime.IsInPreviousHour	Indicates whether the given datetime value occurs during the previous hour, as determined by the current date and time on the system.
DateTime.IsInPreviousMinute	Indicates whether the given datetime value occurs during the previous minute, as determined by the current date and time on the system.
DateTime.IsInPreviousNHours	Indicates whether the given datetime value occurs during the previous number of hours, as determined by the current date and time on the system.
DateTime.IsInPreviousNMinutes	Indicates whether the given datetime value occurs during the previous number of minutes, as determined by the current date and time on the system.
DateTime.IsInPreviousNSeconds	Indicates whether the given datetime value occurs during the previous number of seconds, as determined by the current date and time on the system.
DateTime.IsInPreviousSecond	Indicates whether the given datetime value occurs during the previous second, as determined by the current date and time on the system.
DateTime.LocalNow	Returns a datetime value set to the current date and time on the system.
DateTime.Time	Returns a time part from a DateTime value.
DateTime.ToRecord	Returns a record containing parts of a DateTime value.
DateTime.ToText	Returns a text value from a DateTime value.
#datetime	Creates a datetime value from year, month, day, hour, minute, and second.

DateTime.AddZone

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.AddZone(dateTime as nullable datetime, timezoneHours as number, optional
timezoneMinutes as nullable number) as nullable datetimezone
```

About

Sets timezone information to on the datetime value `dateTime`. The timezone information will include `timezoneHours` and optionally `timezoneMinutes`.

Example 1

Set timezone information for #datetime(2010, 12, 31, 11, 56, 02) to 7 hours, 30 minutes.

```
DateTime.AddZone(#datetime(2010, 12, 31, 11, 56, 02), 7, 30)
```

```
#datetimezone(2010, 12, 31, 11, 56, 2, 7, 30)
```

DateTime.Date

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.Date(dateTime as any) as nullable date
```

About

Returns the date component of `dateTime`, the given `date`, `datetime`, or `datetimezone` value.

Example 1

Find date value of #datetime(2010, 12, 31, 11, 56, 02).

```
DateTime.Date(#datetime(2010, 12, 31, 11, 56, 02))
```

```
#date(2010, 12, 31)
```

DateTime.FixedLocalNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.FixedLocalNow() as datetime
```

About

Returns a `datetime` value set to the current date and time on the system. This value is fixed and will not change with successive calls, unlike DateTime.LocalNow, which may return different values over the course of execution of an expression.

DateTime.From

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.From(value as any, optional culture as nullable text) as nullable datetime
```

About

Returns a `datetime` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, `DateTime.From` returns `null`. If the given `value` is `datetime`, `value` is returned. Values of the following types can be converted to a `datetime` value:

- `text`: A `datetime` value from textual representation. See `DateTime.FromText` for details.
- `date`: A `datetime` with `value` as the date component and `12:00:00 AM` as the time component.
- `datetimezone`: The local `datetime` equivalent of `value`.
- `time`: A `datetime` with the date equivalent of the OLE Automation Date of `0` as the date component and `value` as the time component.
- `number`: A `datetime` equivalent the OLE Automation Date expressed by `value`.

If `value` is of any other type, an error is returned.

Example 1

Convert `#time(06, 45, 12)` to a `datetime` value.

```
DateTime.From(#time(06, 45, 12))
```

```
#datetime(1899, 12, 30, 06, 45, 12)
```

Example 2

Convert `#date(1975, 4, 4)` to a `datetime` value.

```
DateTime.From(#date(1975, 4, 4))
```

```
#datetime(1975, 4, 4, 0, 0, 0)
```

DateTime.FromFileTime

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.FromFileTime(fileTime as nullable number) as nullable datetime
```

About

Creates a `datetime` value from the `fileTime` value and converts it to the local time zone. The filetime is a Windows file time value that represents the number of 100-nanosecond intervals that have elapsed since 12:00 midnight, January 1, 1601 A.D. (C.E.) Coordinated Universal Time (UTC).

Example 1

Convert `129876402529842245` into a datetime value.

```
DateTime.FromFileTime(129876402529842245)
```

```
#datetime(2012, 7, 24, 14, 50, 52.9842245)
```

DateTime.FromText

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.FromText(text as nullable text, optional culture as nullable text) as nullable datetime
```

About

Creates a `datetime` value from a textual representation, `text`, following ISO 8601 format standard. An optional `culture` may also be provided (for example, "en-US").

- `DateTime.FromText("2010-12-31T01:30:00")` // yyyy-MM-ddThh:mm:ss

Example 1

Convert `"2010-12-31T01:30:25"` into a datetime value.

```
DateTime.FromText("2010-12-31T01:30:25")
```

```
#datetime(2010, 12, 31, 1, 30, 25)
```

Example 2

Convert `"2010-12-31T01:30"` into a datetime value.

```
DateTime.FromText("2010-12-31T01:30")
```

```
#datetime(2010, 12, 31, 1, 30, 0)
```

Example 3

Convert `"20101231T013025"` into a datetime value.

```
DateTime.FromText("20101231T013025")
```

```
#datetime(2010, 12, 31, 1, 30, 25)
```

Example 4

Convert `"20101231T01:30:25"` into a datetime value.

```
DateTime.FromText("20101231T01:30:25")
```

```
#datetime(2010, 12, 31, 1, 30, 25)
```

Example 5

Convert `"20101231T01:30:25.121212"` into a datetime value.

```
DateTime.FromText("20101231T01:30:25.121212")
```

```
#datetime(2010, 12, 31, 1, 30, 25.121212)
```

DateTime.IsInCurrentHour

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInCurrentHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current hour, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current hour.

```
DateTime.IsInCurrentHour(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInCurrentMinute

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInCurrentMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current minute, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current minute.

```
DateTime.IsInCurrentMinute(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInCurrentSecond

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInCurrentSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current second, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current second.

```
DateTime.IsInCurrentSecond(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInNextHour

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next hour, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime` : A `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the hour after the current system time is in the next hour.

```
DateTime.IsInNextHour(DateTime.FixedLocalNow() + #duration(0, 1, 0, 0))
```

`true`

DateTime.IsInNextMinute

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next minute, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the minute after the current system time is in the next minute.

```
DateTime.IsInNextMinute(DateTime.FixedLocalNow() + #duration(0, 0, 1, 0))
```

`true`

DateTime.IsInNextNHours

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextNHours(dateTime as any, hours as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of hours, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `hours`: The number of hours.

Example 1

Determine if the hour after the current system time is in the next two hours.

```
DateTime.IsInNextNHours(DateTime.FixedLocalNow() + #duration(0, 2, 0, 0), 2)
```

`true`

DateTime.IsInNextNMinutes

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextNMinutes(dateTime as any, minutes as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of minutes, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `minutes`: The number of minutes.

Example 1

Determine if the minute after the current system time is in the next two minutes.

```
DateTime.IsInNextNMinutes(DateTime.FixedLocalNow() + #duration(0, 0, 2, 0), 2)
```

`true`

DateTime.IsInNextNSeconds

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextNSeconds(dateTime as any, seconds as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of seconds, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `seconds`: The number of seconds.

Example 1

Determine if the second after the current system time is in the next two seconds.

```
DateTime.IsInNextNSeconds(DateTime.FixedLocalNow() + #duration(0, 0, 0, 2), 2)
```

`true`

DateTime.IsInNextSecond

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInNextSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next second, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime` : A `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the second after the current system time is in the next second.

```
DateTime.IsInNextSecond(DateTime.FixedLocalNow() + #duration(0, 0, 0, 1))
```

`true`

DateTime.IsInPreviousHour

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous hour, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the hour before the current system time is in the previous hour.

```
DateTime.IsInPreviousHour(DateTime.FixedLocalNow() - #duration(0, 1, 0, 0))
```

`true`

DateTime.IsInPreviousMinute

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous minute, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime` : A `datetime` , or `datetimezone` value to be evaluated.

Example 1

Determine if the minute before the current system time is in the previous minute.

```
DateTime.IsInPreviousMinute(DateTime.FixedLocalNow() - #duration(0, 0, 1, 0))
```

`true`

DateTime.IsInPreviousNHours

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousNHours(dateTime as any, hours as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of hours, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `hours`: The number of hours.

Example 1

Determine if the hour before the current system time is in the previous two hours.

```
DateTime.IsInPreviousNHours(DateTime.FixedLocalNow() - #duration(0, 2, 0, 0), 2)
```

`true`

DateTime.IsInPreviousNMinutes

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousNMinutes(dateTime as any, minutes as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of minutes, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `minutes`: The number of minutes.

Example 1

Determine if the minute before the current system time is in the previous two minutes.

```
DateTime.IsInPreviousNMinutes(DateTime.FixedLocalNow() - #duration(0, 0, 2, 0), 2)
```

`true`

DateTime.IsInPreviousNSeconds

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousNSeconds(dateTime as any, seconds as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of seconds, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `seconds`: The number of seconds.

Example 1

Determine if the second before the current system time is in the previous two seconds.

```
DateTime.IsInPreviousNSeconds(DateTime.FixedLocalNow() - #duration(0, 0, 0, 2), 2)
```

`true`

DateTime.IsInPreviousSecond

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.IsInPreviousSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous second, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the second before the current system time is in the previous second.

```
DateTime.IsInPreviousSecond(DateTime.FixedLocalNow() - #duration(0, 0, 0, 1))
```

`true`

DateTime.LocalNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.LocalNow() as datetime
```

About

Returns a `datetime` value set to the current date and time on the system.

DateTime.Time

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.Time(dateTime as any) as nullable time
```

About

Returns the time part of the given datetime value, `dateTime`.

Example 1

Find the time value of #datetime(2010, 12, 31, 11, 56, 02).

```
DateTime.Time(#datetime(2010, 12, 31, 11, 56, 02))
```

`#time(11, 56, 2)`

DateTime.ToRecord

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.ToRecord(dateTime as datetime) as record
```

About

Returns a record containing the parts of the given datetime value, `dateTime`.

- `dateTime`: A `datetime` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#datetime(2011, 12, 31, 11, 56, 2)` value into a record containing Date and Time values.

```
DateTime.ToRecord(#datetime(2011, 12, 31, 11, 56, 2))
```

YEAR	2011
MONTH	12
DAY	31
HOUR	11
MINUTE	56
SECOND	2

DateTime.ToText

3/15/2021 • 2 minutes to read

Syntax

```
DateTime.ToText(dateTime as nullable datetime, optional format as nullable text, optional
culture as nullable text) as nullable text
```

About

Returns a textual representation of `dateTime`. An optional `format` may be provided to customize the formatting of the text. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get a textual representation of #datetime(2011, 12, 31, 11, 56, 2).

```
DateTime.ToText(#datetime(2010, 12, 31, 11, 56, 2))
```

```
"12/31/2010 11:56:02 AM"
```

Example 2

Get a textual representation of #datetime(2011, 12, 31, 11, 56, 2) with format option.

```
DateTime.ToText(#datetime(2010, 12, 31, 11, 56, 2), "yyyy/MM/ddThh:mm:ss")
```

```
"2010/12/31T11:56:02"
```

#datetime

6/22/2021 • 2 minutes to read

Syntax

```
#datetime(year as number, month as number, day as number, hour as number, minute as number,
second as number) as any
```

About

Creates a datetime value from numbers representing the year, month, day, hour, minute, and (fractional) second. Raises an error if these conditions are not true:

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31
- 0 ≤ hour ≤ 23
- 0 ≤ minute ≤ 59
- 0 ≤ second < 60

DateTimeZone functions

3/15/2021 • 2 minutes to read

These functions create and manipulate datetimezone values.

DateTimeZone

FUNCTION	DESCRIPTION
DateTimeZone.FixedLocalNow	Returns a DateTimeZone value set to the current date, time, and timezone offset on the system.
DateTimeZone.FixedUtcNow	Returns the current date and time in UTC (the GMT timezone).
DateTimeZone.From	Returns a datetimezone value from a value.
DateTimeZone.FromFileTime	Returns a DateTimeZone from a number value.
DateTimeZone.FromText	Returns a DateTimeZone value from a set of date formats and culture value.
DateTimeZone.LocalNow	Returns a DateTime value set to the current system date and time.
DateTimeZone.RemoveZone	Returns a datetime value with the zone information removed from the input datetimezone value.
DateTimeZone.SwitchZone	Changes the timezone information for the input DateTimeZone.
DateTimeZone.ToLocal	Returns a DateTime value from the local time zone.
DateTimeZone.ToRecord	Returns a record containing parts of a DateTime value.
DateTimeZone.ToText	Returns a text value from a DateTime value.
DateTimeZone.ToUtc	Returns a DateTime value to the Utc time zone.
DateTimeZone.UtcNow	Returns a DateTime value set to the current system date and time in the Utc timezone.
DateTimeZone.ZoneHours	Returns a time zone hour value from a DateTime value.
DateTimeZone.ZoneMinutes	Returns a time zone minute value from a DateTime value.
#datetimezone	Creates a datetimezone value from year, month, day, hour, minute, second, offset-hours, and offset-minutes.

DateTimeZone.FixedLocalNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.FixedLocalNow() as datetimezone
```

About

Returns a `datetime` value set to the current date and time on the system. The returned value contains timezone information representing the local timezone. This value is fixed and will not change with successive calls, unlike DateTimeZone.LocalNow, which may return different values over the course of execution of an expression.

DateTimeZone.FixedUtcNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.FixedUtcNow() as datetimezone
```

About

Returns the current date and time in UTC (the GMT timezone). This value is fixed and will not change with successive calls.

DateTimeZone.From

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.From(value as any, optional culture as nullable text) as nullable datetimezone
```

About

Returns a `datetimezone` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, `DateTimeZone.From` returns `null`. If the given `value` is `datetimezone`, `value` is returned. Values of the following types can be converted to a `datetimezone` value:

- `text`: A `datetimezone` value from textual representation. See `DateTimeZone.FromText` for details.
- `date`: A `datetimezone` with `value` as the date component, `12:00:00 AM` as the time component and the offset corresponding the local time zone.
- `datetime`: A `datetimezone` with `value` as the datetime and the offset corresponding the local time zone.
- `time`: A `datetimezone` with the date equivalent of the OLE Automation Date of `0` as the date component, `value` as the time component and the offset corresponding the local time zone.
- `number`: A `datetimezone` with the datetime equivalent the OLE Automation Date expressed by `value` and the offset corresponding the local time zone.

If `value` is of any other type, an error is returned.

Example 1

Convert `"2020-10-30T01:30:00-08:00"` to a `datetimezone` value.

```
DateTimeZone.From("2020-10-30T01:30:00-08:00")
```

```
#datetimezone(2020, 10, 30, 01, 30, 00, -8, 00)
```

DateTimeZone.FromFileTime

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.FromFileTime(fileTime as nullable number) as nullable datetimezone
```

About

Creates a `datetimezone` value from the `fileTime` value and converts it to the local time zone. The filetime is a Windows file time value that represents the number of 100-nanosecond intervals that have elapsed since 12:00 midnight, January 1, 1601 A.D. (C.E.) Coordinated Universal Time (UTC).

Example 1

Convert `129876402529842245` into a datetimezone value.

```
DateTimeZone.FromFileTime(129876402529842245)
```

```
#datetimezone(2012, 7, 24, 14, 50, 52.9842245, -7, 0)
```

DateTimeZone.FromText

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.FromText(text as nullable text, optional culture as nullable text) as nullable
datetimezone
```

About

Creates a `datetimezone` value from a textual representation, `text`, following ISO 8601 format standard. An optional `culture` may also be provided (for example, "en-US").

- `DateTimeZone.FromText("2010-12-31T01:30:00-08:00")` // yyyy-MM-ddThh:mm:ssZ

Example 1

Convert `"2010-12-31T01:30:00-08:00"` into a datetimezone value.

```
DateTimeZone.FromText("2010-12-31T01:30:00-08:00")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, -8, 0)
```

Example 2

Convert `"2010-12-31T01:30:00.121212-08:00"` into a datetimezone value.

```
DateTimeZone.FromText("2010-12-31T01:30:00.121212-08:00")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0.121212, -8, 0)
```

Example 3

Convert `"2010-12-31T01:30:00Z"` into a datetimezone value.

```
DateTimeZone.FromText("2010-12-31T01:30:00Z")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, 0, 0)
```

Example 4

Convert `"20101231T013000+0800"` into a datetimezone value.

```
DateTimeZone.FromText("20101231T013000+0800")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, 8, 0)
```

DateTimeZone.LocalNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.LocalNow() as datetimezone
```

About

Returns a `datetimezone` value set to the current date and time on the system. The returned value contains timezone information representing the local timezone.

DateTimeZone.RemoveZone

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.RemoveZone(dateTimeZone as nullable datetimezone) as nullable datetime
```

About

Returns a #datetime value from `dateTimeZone` with timezone information removed.

Example 1

Remove timezone information from the value #datetimezone(2011, 12, 31, 9, 15, 36, -7, 0).

```
DateTimeZone.RemoveZone(#datetimezone(2011, 12, 31, 9, 15, 36, -7, 0))
```

```
#datetime(2011, 12, 31, 9, 15, 36)
```

DateTimeZone.SwitchZone

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.SwitchZone(dateTimeZone as nullable datetimezone, timezoneHours as number, optional
timezoneMinutes as nullable number) as nullable datetimezone
```

About

Changes timezone information to on the datetimezone value `dateTimeZone` to the new timezone information provided by `timezoneHours` and optionally `timezoneMinutes`. If `dateTimeZone` does not have a timezone component, an exception is thrown.

Example 1

Change timezone information for #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) to 8 hours.

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 8)
```

```
#datetimezone(2010, 12, 31, 12, 26, 2, 8, 0)
```

Example 2

Change timezone information for #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) to -30 minutes.

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 0, -30)
```

```
#datetimezone(2010, 12, 31, 3, 56, 2, 0, -30)
```

DateTimeZone.ToLocal

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ToLocal(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

About

Changes timezone information of the datetimezone value `dateTimeZone` to the local timezone information. If `dateTimeZone` does not have a timezone component, the local timezone information is added.

Example 1

Change timezone information for #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) to local timezone (assuming PST).

```
DateTimeZone.ToLocal(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

```
#datetimezone(2010, 12, 31, 12, 26, 2, -8, 0)
```

DateTimeZone.ToRecord

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ToRecord(dateTimeZone as datetimezone) as record
```

About

Returns a record containing the parts of the given datetimezone value, `dateTimeZone`.

- `dateTimeZone`: A `datetimezone` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0)` value into a record containing Date, Time, and Zone values.

```
DateTimeZone.ToRecord(#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0))
```

YEAR	2011
MONTH	12
DAY	31
HOUR	11
MINUTE	56
SECOND	2
ZONEHOURS	8
ZONEMINUTES	0

DateTimeZone.ToText

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ToText(dateTimeZone as nullable datetimezone, optional format as nullable text,
optional culture as nullable text) as nullable text
```

About

Returns a textual representation of `dateTimeZone`. An optional `format` may be provided to customize the formatting of the text. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get a textual representation of #datetimezone(2011, 12, 31, 11, 56, 2, 8, 0).

```
DateTimeZone.ToText(#datetimezone(2010, 12, 31, 11, 56, 2, 8, 0))
```

```
"12/31/2010 11:56:02 AM +08:00"
```

Example 2

Get a textual representation of #datetimezone(2010, 12, 31, 11, 56, 2, 10, 12) with format option.

```
DateTimeZone.ToText(#datetimezone(2010, 12, 31, 11, 56, 2, 10, 12), "yyyy/MM/ddThh:mm:sszzz")
```

```
"2010/12/31T11:56:02+10:12"
```

DateTimeZone.ToUtc

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ToUtc(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

About

Changes timezone information of the datetime value `dateTimeZone` to the UTC or Universal Time timezone information. If `dateTimeZone` does not have a timezone component, the UTC timezone information is added.

Example 1

Change timezone information for #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) to UTC timezone.

```
DateTimeZone.ToUtc(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

```
#datetimezone(2010, 12, 31, 4, 26, 2, 0, 0)
```

DateTimeZone.UtcNow

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.UtcNow() as datetimezone
```

About

Returns the current date and time in UTC (the GMT timezone).

Example 1

Get the current date & time in UTC.

```
DateTimeZone.UtcNow()
```

```
#datetimezone(2011, 8, 16, 23, 34, 37.745, 0, 0)
```

DateTimeZone.ZoneHours

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ZoneHours(dateTimeZone as nullable datetimezone) as nullable number
```

About

Changes the timezone of the value.

DateTimeZone.ZoneMinutes

3/15/2021 • 2 minutes to read

Syntax

```
DateTimeZone.ZoneMinutes(dateTimeZone as nullable datetimezone) as nullable number
```

About

Changes the timezone of the value.

#datetimezone

6/22/2021 • 2 minutes to read

Syntax

```
#datetimezone(year as number, month as number, day as number, hour as number, minute as number,
second as number, offsetHours as number, offsetMinutes as number) as any
```

About

Creates a datetimezone value from numbers representing the year, month, day, hour, minute, (fractional) second, (fractional) offset-hours, and offset-minutes. Raises an error if these conditions are not true:

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31
- 0 ≤ hour ≤ 23
- 0 ≤ minute ≤ 59
- 0 ≤ second < 60
- -14 ≤ offset-hours + offset-minutes / 60 ≤ 14

Duration functions

6/22/2021 • 2 minutes to read

These functions create and manipulate duration values.

Duration

FUNCTION	DESCRIPTION
Duration.Days	Returns the day component of a Duration value.
Duration.From	Returns a duration value from a value.
Duration.FromText	Returns a Duration value from a text value.
Duration.Hours	Returns an hour component of a Duration value.
Duration.Minutes	Returns a minute component of a Duration value.
Duration.Seconds	Returns a second component of a Duration value.
Duration.ToRecord	Returns a record with parts of a Duration value.
Duration.TotalDays	Returns the total magnitude of days from a Duration value.
Duration.TotalHours	Returns the total magnitude of hours from a Duration value.
Duration.TotalMinutes	Returns the total magnitude of minutes from a Duration value.
Duration.TotalSeconds	Returns the total magnitude of seconds from a duration value.
Duration.ToText	Returns a text value from a Duration value.
#duration	Creates a duration value from days, hours, minutes, and seconds.

Duration.Days

3/15/2021 • 2 minutes to read

Syntax

```
Duration.Days(duration as nullable duration) as nullable number
```

About

Returns the day component of the provided `duration` value, `duration`.

Example 1

Find the day in #duration(5, 4, 3, 2).

```
Duration.Days(#duration(5, 4, 3, 2))
```

`5`

Duration.From

3/15/2021 • 2 minutes to read

Syntax

```
Duration.From(value as any) as nullable duration
```

About

Returns a `duration` value from the given `value`. If the given `value` is `null`, `Duration.From` returns `null`. If the given `value` is `duration`, `value` is returned. Values of the following types can be converted to a `duration` value:

- `text`: A `duration` value from textual elapsed time forms (d.h:m:s). See `Duration.FromText` for details.
- `number`: A `duration` equivalent to the number of whole and fractional days expressed by `value`.

If `value` is of any other type, an error is returned.

Example 1

Convert `2.525` into a `duration` value.

```
Duration.From(2.525)
```

```
#duration(2, 12, 36, 0)
```

Duration.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Duration.FromText(text as nullable text) as nullable duration
```

About

Returns a duration value from the specified text, `text`. The following formats can be parsed by this function:

- (-)hh:mm(:ss(.ff))
- (-)ddd(.hh:mm(:ss(.ff)))

(All ranges are inclusive)
ddd: Number of days.
hh: Number of hours, between 0 and 23.
mm: Number of minutes, between 0 and 59.
ss: Number of seconds, between 0 and 59.
ff: Fraction of seconds, between 0 and 9999999.

Example 1

Convert `"2.05:55:20"` into a `duration` value.

```
Duration.FromText("2.05:55:20")
```

`#duration(2, 5, 55, 20)`

Duration.Hours

3/15/2021 • 2 minutes to read

Syntax

```
Duration.Hours(duration as nullable duration) as nullable number
```

About

Returns the hour component of the provided `duration` value, `duration`.

Example 1

Find the hours in #duration(5, 4, 3, 2).

```
Duration.Hours(#duration(5, 4, 3, 2))
```

`4`

Duration.Minutes

3/15/2021 • 2 minutes to read

Syntax

```
Duration.Minutes(duration as nullable duration) as nullable number
```

About

Returns the minutes component of the provided `duration` value, `duration`.

Example 1

Find the minutes in #duration(5, 4, 3, 2).

```
Duration.Minutes(#duration(5, 4, 3, 2))
```

`3`

Duration.Seconds

3/15/2021 • 2 minutes to read

Syntax

```
Duration.Seconds(duration as nullable duration) as nullable number
```

About

Returns the seconds component of the provided `duration` value, `duration`.

Example 1

Find the seconds in #duration(5, 4, 3, 2).

```
Duration.Seconds(#duration(5, 4, 3, 2))
```

`2`

Duration.ToRecord

3/15/2021 • 2 minutes to read

Syntax

```
Duration.ToRecord(duration as duration) as record
```

About

Returns a record containing the parts the duration value, `duration`.

- `duration`: A `duration` from which the record is created.

Example 1

Convert `#duration(2, 5, 55, 20)` into a record of its parts including days, hours, minutes and seconds if applicable.

```
Duration.ToRecord(#duration(2, 5, 55, 20))
```

DAYS	2
HOURS	5
MINUTES	55
SECONDS	20

Duration.TotalDays

3/15/2021 • 2 minutes to read

Syntax

```
Duration.TotalDays(duration as nullable duration) as nullable number
```

About

Returns the total days spanned by the provided `duration` value, `duration`.

Example 1

Find the total days spanned in #duration(5, 4, 3, 2).

```
Duration.TotalDays(#duration(5, 4, 3, 2))
```

```
5.1687731481481478
```

Duration.TotalHours

3/15/2021 • 2 minutes to read

Syntax

```
Duration.TotalHours(duration as nullable duration) as nullable number
```

About

Returns the total hours spanned by the provided `duration` value, `duration`.

Example 1

Find the total hours spanned in #duration(5, 4, 3, 2).

```
Duration.TotalHours(#duration(5, 4, 3, 2))
```

```
124.05055555555555
```

Duration.TotalMinutes

3/15/2021 • 2 minutes to read

Syntax

```
Duration.TotalMinutes(duration as nullable duration) as nullable number
```

About

Returns the total minutes spanned by the provided `duration` value, `duration`.

Example 1

Find the total minutes spanned in #duration(5, 4, 3, 2).

```
Duration.TotalMinutes(#duration(5, 4, 3, 2))
```

`7443.0333333333338`

Duration.TotalSeconds

3/15/2021 • 2 minutes to read

Syntax

```
Duration.TotalSeconds(duration as nullable duration) as nullable number
```

About

Returns the total seconds spanned by the provided `duration` value, `duration`.

Example 1

Find the total seconds spanned in #duration(5, 4, 3, 2).

```
Duration.TotalSeconds(#duration(5, 4, 3, 2))
```

```
446582
```

Duration.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Duration.ToText(duration as nullable duration, optional format as nullable text) as nullable text
```

About

Returns a textual representation in the form "day.hour:mins:sec" of the given duration value, `duration`.

- `duration`: A `duration` from which the textual representation is calculated.
- `format`: *[Optional]* Deprecated, will throw an error if not null.

Example 1

Convert `#duration(2, 5, 55, 20)` into a text value.

```
Duration.ToText(#duration(2, 5, 55, 20))
```

`"2.05:55:20"`

#duration

6/22/2021 • 2 minutes to read

Syntax

```
#duration(days as number, hours as number, minutes as number, seconds as number) as duration
```

About

Creates a duration value from numbers representing days, hours, minutes, and (fractional) seconds.

Error handling

3/15/2021 • 2 minutes to read

These functions return diagnostic traces at different levels of verbosity, as well as throw error records.

Error

FUNCTION	DESCRIPTION
Diagnostics.ActivityId	Returns an opaque identifier for the currently-running evaluation.
Diagnostics.Trace	Writes a trace message, if tracing is enabled, and returns value.
Error.Record	Returns a record containing fields "Reason", "Message", and "Detail" set to the provided values. The record can be used to raise or throw an error.
TraceLevel.Critical	Returns 1, the value for Critical trace level.
TraceLevel.Error	Returns 2, the value for Error trace level.
TraceLevel.Information	Returns 4, the value for Information trace level.
TraceLevel.Verbose	Returns 5, the value for Verbose trace level.
TraceLevel.Warning	Returns 3, the value for Warning trace level.

Diagnostics.ActivityId

3/15/2021 • 2 minutes to read

`

Syntax

```
Diagnostics.ActivityId() as nullable text
```

About

Returns an opaque identifier for the currently-running evaluation.

Diagnostics.Trace

3/15/2021 • 2 minutes to read

Syntax

```
Diagnostics.Trace(traceLevel as number, message as anynonnull, value as any, optional delayed as nullable logical) as any
```

About

Writes a trace `message`, if tracing is enabled, and returns `value`. An optional parameter `delayed` specifies whether to delay the evaluation of `value` until the message is traced. `traceLevel` can take one of the following values:

- `TraceLevel.Critical`
- `TraceLevel.Error`
- `TraceLevel.Warning`
- `TraceLevel.Information`
- `TraceLevel.Verbose`

Example 1

Trace the message before invoking Text.From function and return the result.

```
Diagnostics.Trace(TraceLevel.Information, "TextValueFromNumber", () => Text.From(123), true)
```

`"123"`

Error.Record

3/15/2021 • 2 minutes to read

Syntax

```
Error.Record(reason as text, optional message as nullable text, optional detail as any) as
record
```

About

Returns an error record from the provided text values for reason, message and detail.

TraceLevel.Critical

3/15/2021 • 2 minutes to read

About

Returns 1, the value for Critical trace level.

TraceLevel.Error

3/15/2021 • 2 minutes to read

About

Returns 2, the value for Error trace level.

TraceLevel.Information

3/15/2021 • 2 minutes to read

About

Returns 4, the value for Information trace level.

TraceLevel.Verbose

3/15/2021 • 2 minutes to read

About

Returns 5, the value for Verbose trace level.

TraceLevel.Warning

3/15/2021 • 2 minutes to read

About

Returns 3, the value for Warning trace level.

Expression functions

3/15/2021 • 2 minutes to read

These functions allow the construction and evaluation of M code.

Expression

FUNCTION	DESCRIPTION
Expression.Constant	Returns the M source code representation of a constant value.
Expression.Evaluate	Returns the result of evaluating an M expression.
Expression.Identifier	Returns the M source code representation of an identifier.

Expression.Constant

3/15/2021 • 2 minutes to read

Syntax

```
Expression.Constant(value as any) as text
```

About

Returns the M source code representation of a constant value.

Example 1

Get the M source code representation of a number value.

```
Expression.Constant(123)
```

```
"123"
```

Example 2

Get the M source code representation of a date value.

```
Expression.Constant(#date(2035, 01, 02))
```

```
"#date(2035, 1, 2)"
```

Example 3

Get the M source code representation of a text value.

```
Expression.Constant("abc")
```

```
"""abc"""
```

Expression.Evaluate

3/15/2021 • 2 minutes to read

Syntax

```
Expression.Evaluate(document as text, optional environment as nullable record) as any
```

About

Returns the result of evaluating an M expression `document`, with the available identifiers that can be referenced defined by `environment`.

Example 1

Evaluate a simple sum.

```
Expression.Evaluate("1 + 1")
```

```
2
```

Example 2

Evaluate a more complex sum.

```
Expression.Evaluate("List.Sum({1, 2, 3})", [List.Sum = List.Sum])
```

```
6
```

Example 3

Evaluate the concatenation of a text value with an identifier.

```
Expression.Evaluate(Expression.Constant("""abc") & " & " & Expression.Identifier("x"), [x = "def"""])
```

```
"""abcdef"""
```

Expression.Identifier

3/15/2021 • 2 minutes to read

Syntax

```
Expression.Identifier(name as text) as text
```

About

Returns the M source code representation of an identifier `name`.

Example 1

Get the M source code representation of an identifier.

```
Expression.Identifier("MyIdentifier")
```

```
"MyIdentifier"
```

Example 2

Get the M source code representation of an identifier that contains a space.

```
Expression.Identifier("My Identifier")
```

```
"#""My Identifier"""
```

Function values

3/15/2021 • 2 minutes to read

These functions create and invoke other M functions.

Function

FUNCTION	DESCRIPTION
Function.From	Takes a unary function `function` and creates a new function with the type `functionType` that constructs a list out of its arguments and passes it to `function`.
Function.Invoke	Invokes the given function using the specified and returns the result.
Function.InvokeAfter	Returns the result of invoking function after duration delay has passed.
Function.IsDataSource	Returns whether or not function is considered a data source.
Function.ScalarVector	Returns a scalar function of type scalarFunctionType that invokes vectorFunction with a single row of arguments and returns its single output.

Function.From

3/15/2021 • 2 minutes to read

Syntax

```
Function.From(functionType as type, function as function) as function
```

About

Takes a unary function `function` and creates a new function with the type `functionType` that constructs a list out of its arguments and passes it to `function`.

Example 1

Converts List.Sum into a two-argument function whose arguments are added together.

```
Function.From(type function (a as number, b as number) as number, List.Sum)(2, 1)
```

`3`

Example 2

Converts a function taking a list into a two-argument function.

```
Function.From(type function (a as text, b as text) as text, (list) => list{0} & list{1})("2", "1")
```

`"21"`

Function.Invoke

3/15/2021 • 2 minutes to read

Syntax

```
Function.Invoke(function as function, args as list) as any
```

About

Invokes the given function using the specified list of arguments and returns the result.

Example 1

Invokes Record.FieldNames with one argument [A=1,B=2]

```
Function.Invoke(Record.FieldNames, {[A = 1, B = 2]})
```

A
B

Function.InvokeAfter

3/15/2021 • 2 minutes to read

Syntax

```
Function.InvokeAfter(function as function, delay as duration) as any
```

About

Returns the result of invoking `function` after duration `delay` has passed.

Function.IsDataSource

3/15/2021 • 2 minutes to read

Syntax

```
Function.IsDataSource(function as function) as logical
```

About

Returns whether or not `function` is considered a data source.

Function.ScalarVector

3/15/2021 • 2 minutes to read

Syntax

```
Function.ScalarVector(scalarFunctionType as type, vectorFunction as function) as function
```

About

Returns a scalar function of type `scalarFunctionType` that invokes `vectorFunction` with a single row of arguments and returns its single output. Additionally, when the scalar function is repeatedly applied for each row of a table of inputs, such as in Table.AddColumn, instead `vectorFunction` will be applied once for all inputs.

`vectorFunction` will be passed a table whose columns match in name and position the parameters of `scalarFunctionType`. Each row of this table contains the arguments for one call to the scalar function, with the columns corresponding to the parameters of `scalarFunctionType`.

`vectorFunction` must return a list of the same length as the input table, whose item at each position must be the same result as evaluating the scalar function on the input row of the same position.

The input table is expected to be streamed in, so `vectorFunction` is expected to stream its output as input comes in, only working with one chunk of input at a time. In particular, `vectorFunction` must not enumerate its input table more than once.

Lines functions

3/15/2021 • 2 minutes to read

These functions convert lists of text to and from binary and single text values.

Lines

FUNCTION	DESCRIPTION
Lines.FromBinary	Converts a binary value to a list of text values split at lines breaks.
Lines.FromText	Converts a text value to a list of text values split at lines breaks.
Lines.ToBinary	Converts a list of text into a binary value using the specified encoding and lineSeparator.The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.
Lines.ToText	Converts a list of text into a single text. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

Lines.FromBinary

3/15/2021 • 2 minutes to read

Syntax

```
Lines.FromBinary(binary as binary, optional quoteStyle as nullable number, optional
includeLineSeparators as nullable logical, optional encoding as nullable number) as list
```

About

Converts a binary value to a list of text values split at lines breaks. If a quote style is specified, then line breaks may appear within quotes. If includeLineSeparators is true, then the line break characters are included in the text.

Lines.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Lines.FromText(text as text, optional quoteStyle as nullable number, optional
includeLineSeparators as nullable logical) as list
```

About

Converts a text value to a list of text values split at lines breaks. If includeLineSeparators is true, then the line break characters are included in the text.

- `QuoteStyle.None:` (default) No quoting behavior is needed.
- `QuoteStyle.Csv:` Quoting is as per Csv. A double quote character is used to demarcate such regions, and a pair of double quote characters is used to indicate a single double quote character within such a region.

Lines.ToBinary

3/15/2021 • 2 minutes to read

Syntax

```
Lines.ToBinary(lines as list, optional lineSeparator as nullable text, optional encoding as
nullable number, optional includeByteOrderMark as nullable logical) as binary
```

About

Converts a list of text into a binary value using the specified encoding and lineSeparator.The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

Lines.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Lines.ToText(lines as list, optional lineSeparator as nullable text) as text
```

About

Converts a list of text into a single text. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

List functions

3/15/2021 • 8 minutes to read

These functions create and manipulate list values.

Information

FUNCTION	DESCRIPTION
List.Count	Returns the number of items in a list.
List.NonNullCount	Returns the number of items in a list excluding null values
List.IsEmpty	Returns whether a list is empty.

Selection

FUNCTION	DESCRIPTION
List.Alternate	Returns a list with the items alternated from the original list based on a count, optional repeatInterval, and an optional offset.
List.Buffer	Buffers the list in memory. The result of this call is a stable list, which means it will have a determinimic count, and order of items.
List.Distinct	Filters a list down by removing duplicates. An optional equation criteria value can be specified to control equality comparison. The first value from each equality group is chosen.
List.FindText	Searches a list of values, including record fields, for a text value.
List.First	Returns the first value of the list or the specified default if empty. Returns the first item in the list, or the optional default value, if the list is empty. If the list is empty and a default value is not specified, the function returns.
List.FirstN	Returns the first set of items in the list by specifying how many items to return or a qualifying condition provided by **countOrCondition**.
List.InsertRange	Inserts items from values at the given index in the input list.
List.IsDistinct	Returns whether a list is distinct.
List.Last	Returns the last set of items in the list by specifying how many items to return or a qualifying condition provided by **countOrCondition**.

FUNCTION	DESCRIPTION
List.LastN	Returns the last set of items in a list by specifying how many items to return or a qualifying condition.
List.MatchesAll	Returns true if all items in a list meet a condition.
List.MatchesAny	Returns true if any item in a list meets a condition.
List.Positions	Returns a list of positions for an input list.
List.Range	Returns a count items starting at an offset.
List.Select	Selects the items that match a condition.
List.Single	Returns the single item of the list or throws an Expression.Error if the list has more than one item.
List.SingleOrDefault	Returns a single item from a list.
List.Skip	Skips the first item of the list. Given an empty list, it returns an empty list. This function takes an optional parameter countOrCondition to support skipping multiple values.

Transformation functions

FUNCTION	DESCRIPTION
List.Accumulate	Accumulates a result from the list. Starting from the initial value seed this function applies the accumulator function and returns the final result.
List.Combine	Merges a list of lists into single list.
List.ConformToPageReader	This function is intended for internal use only.
List.RemoveRange	Returns a list that removes count items starting at offset. The default count is 1.
List.RemoveFirstN	Returns a list with the specified number of elements removed from the list starting at the first element. The number of elements removed depends on the optional countOrCondition parameter.
List.RemoveItems	Removes items from list1 that are present in list2, and returns a new list.
List.RemoveLastN	Returns a list with the specified number of elements removed from the list starting at the last element. The number of elements removed depends on the optional countOrCondition parameter.
List.Repeat	Returns a list that repeats the contents of an input list count times.

FUNCTION	DESCRIPTION
List.ReplaceRange	Returns a list that replaces count values in a list with a replaceWith list starting at an index.
List.RemoveMatchingItems	Removes all occurrences of the given values in the list.
List.RemoveNulls	Removes null values from a list.
List.ReplaceMatchingItems	Replaces occurrences of existing values in the list with new values using the provided equationCriteria. Old and new values are provided by the replacements parameters. An optional equation criteria value can be specified to control equality comparisons. For details of replacement operations and equation criteria, see Parameter Values.
List.ReplaceValue	Searches a list of values for the value and replaces each occurrence with the replacement value.
List.Reverse	Returns a list that reverses the items in a list.
List.Split	Splits the specified list into a list of lists using the specified page size.
List.Transform	Performs the function on each item in the list and returns the new list.
List.TransformMany	Returns a list whose elements are projected from the input list.

Membership functions

Since all values can be tested for equality, these functions can operate over heterogeneous lists.

FUNCTION	DESCRIPTION
List.AllTrue	Returns true if all expressions in a list are true
List.AnyTrue	Returns true if any expression in a list in true
List.Contains	Returns true if a value is found in a list.
List.ContainsAll	Returns true if all items in values are found in a list.
List.ContainsAny	Returns true if any item in values is found in a list.
List.PositionOf	Finds the first occurrence of a value in a list and returns its position.
List.PositionOfAny	Finds the first occurrence of any value in values and returns its position.

Set operations

FUNCTION	DESCRIPTION
List.Difference	Returns the items in list 1 that do not appear in list 2. Duplicate values are supported.
List.Intersect	Returns a list from a list of lists and intersects common items in individual lists. Duplicate values are supported.
List.Union	Returns a list from a list of lists and unions the items in the individual lists. The returned list contains all items in any input lists. Duplicate values are matched as part of the Union.
List.Zip	Returns a list of lists combining items at the same position.

Ordering

Ordering functions perform comparisons. All values that are compared must be comparable with each other. This means they must all come from the same datatype (or include null, which always compares smallest). Otherwise, an Expression.Error is thrown.

Comparable data types

- Number
- Duration
- DateTime
- Text
- Logical
- Null

FUNCTION	DESCRIPTION
List.Max	Returns the maximum item in a list, or the optional default value if the list is empty.
List.MaxN	Returns the maximum values in the list. After the rows are sorted, optional parameters may be specified to further filter the result
List.Median	Returns the median item from a list.
List.Min	Returns the minimum item in a list, or the optional default value if the list is empty.
List.MinN	Returns the minimum values in a list.
List.Sort	Returns a sorted list using comparison criterion.
List.Percentile	Returns one or more sample percentiles corresponding to the given probabilities.
PercentileMode.ExcelExc	When interpolating values for `List.Percentile`, use a method compatible with Excel's `PERCENTILE.EXC`.

FUNCTION	DESCRIPTION
PercentileMode.ExcelInc	When interpolating values for `List.Percentile`, use a method compatible with Excel's `PERCENTILE.INC`.
PercentileMode.SqlCont	When interpolating values for `List.Percentile`, use a method compatible with SQL Server's `PERCENTILE_CONT`.
PercentileMode.SqlDisc	When interpolating values for `List.Percentile`, use a method compatible with SQL Server's `PERCENTILE_DISC`.

Averages

These functions operate over homogeneous lists of Numbers, DateTimes, and Durations.

FUNCTION	DESCRIPTION
List.Average	Returns an average value from a list in the datatype of the values in the list.
List.Mode	Returns an item that appears most commonly in a list.
List.Modes	Returns all items that appear with the same maximum frequency.
List.StandardDeviation	Returns the standard deviation from a list of values. List.StandardDeviation performs a sample based estimate. The result is a number for numbers, and a duration for DateTimes and Durations.

Addition

These functions work over homogeneous lists of Numbers or Durations.

FUNCTION	DESCRIPTION
List.Sum	Returns the sum from a list.

Numerics

These functions only work over numbers.

FUNCTION	DESCRIPTION
List.Covariance	Returns the covariance from two lists as a number.
List.Product	Returns the product from a list of numbers.

Generators

These functions generate list of values.

FUNCTION	DESCRIPTION
List.Dates	Returns a list of date values from size count, starting at start and adds an increment to every value.
List.DateTimes	Returns a list of datetime values from size count, starting at start and adds an increment to every value.
List.DateTimeZones	Returns a list of of datetimezone values from size count, starting at start and adds an increment to every value.
List.Durations	Returns a list of durations values from size count, starting at start and adds an increment to every value.
List.Generate	Generates a list from a value function, a condition function, a next function, and an optional transformation function on the values.
List.Numbers	Returns a list of numbers from size count starting at initial, and adds an increment. The increment defaults to 1.
List.Random	Returns a list of count random numbers, with an optional seed parameter.
List.Times	Returns a list of time values of size count, starting at start.

Parameter values

Occurrence specification

- Occurrence.First = 0;
- Occurrence.Last = 1;
- Occurrence.All = 2;

Sort order

- Order.Ascending = 0;
- Order.Descending = 1;

Equation criteria

Equation criteria for list values can be specified as either a

- A function value that is either
  - A key selector that determines the value in the list to apply the equality criteria, or
  - A comparer function that is used to specify the kind of comparison to apply. Built in comparer functions can be specified, see section for Comparer functions.
- A list value which has
  - Exactly two items
  - The first element is the key selector as specified above
  - The second element is a comparer as specified above.

For more information and examples, see List.Distinct.

Comparison criteria

Comparison criterion can be provided as either of the following values:

- A number value to specify a sort order. For more inforarmtion, see sort order in Parameter values.
- To compute a key to be used for sorting, a function of 1 argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order.
- To completely control the comparison, a function of 2 arguments can be used that returns -1, 0, or 1 given the relationship between the left and right inputs. Value.Compare is a method that can be used to delegate this logic.

For more information and examples, see List.Sort.

Replacement operations

Replacement operations are specified by a list value, each item of this list must be

- A list value of exactly two items
- Fist item is the old value in the list, to be replaced
- Second item is the new which should replace all occurrences of the old value in the list

List.Accumulate

3/15/2021 • 2 minutes to read

Syntax

```
List.Accumulate(list as list, seed as any, accumulator as function) as any
```

About

Accumulates a summary value from the items in the list `list`, using `accumulator`. An optional seed parameter, `seed`, may be set.

Example 1

Accumulates the summary value from the items in the list {1, 2, 3, 4, 5} using ((state, current) => state + current).

```
List.Accumulate({1, 2, 3, 4, 5}, 0, (state, current) => state + current)
```

```
15
```

List.AllTrue

3/15/2021 • 2 minutes to read

Syntax

```
List.AllTrue(list as list) as logical
```

About

Returns true if all expressions in the list `list` are true.

Example 1

Determine if all the expressions in the list {true, true, 2 > 0} are true.

```
List.AllTrue({true, true, 2 > 0})
```

`true`

Example 2

Determine if all the expressions in the list {true, true, 2 < 0} are true.

```
List.AllTrue({true, false, 2 < 0})
```

`false`

List.Alternate

3/15/2021 • 2 minutes to read

Syntax

```
List.Alternate(list as list, count as number, optional repeatInterval as nullable number,
optional offset as nullable number) as list
```

About

Returns a list comprised of all the odd numbered offset elements in a list. Alternates between taking and skipping values from the list `list` depending on the parameters.

- `count`: Specifies number of values that are skipped each time.
- `repeatInterval`: An optional repeat interval to indicate how many values are added in between the skipped values.
- `offset`: An option offset parameter to begin skipping the values at the initial offset.

Example 1

Create a list from {1..10} that skips the first number.

```
List.Alternate({1..10}, 1)
```

2
3
4
5
6
7
8
9
10

Example 2

Create a list from {1..10} that skips the every other number.

```
List.Alternate({1..10}, 1, 1)
```

2
4
6
8
10

Example 3

Create a list from {1..10} that starts at 1 and skips every other number.

```
List.Alternate({1..10}, 1, 1, 1)
```

1
3
5
7
9

Example 4

Create a list from {1..10} that starts at 1, skips one value, keeps two values and so on.

```
List.Alternate({1..10}, 1, 2, 1)
```

1
3
4
6
7
9
10

List.AnyTrue

3/15/2021 • 2 minutes to read

Syntax

```
List.AnyTrue(list as list) as logical
```

About

Returns true if any expression in the list `list` is true.

Example 1

Determine if any of the expressions in the list {true, false, 2 > 0} are true.

```
List.AnyTrue({true, false, 2>0})
```

`true`

Example 2

Determine if any of the expressions in the list {2 = 0, false, 2 < 0} are true.

```
List.AnyTrue({2 = 0, false, 2 < 0})
```

`false`

List.Average

3/15/2021 • 2 minutes to read

Syntax

```
List.Average(list as list, optional precision as nullable number) as any
```

About

Returns the average value for the items in the list, `list`. The result is given in the same datatype as the values in the list. Only works with number, date, time, datetime, datetimezone and duration values. If the list is empty null is returned.

Example 1

Find the average of the list of numbers, `{3, 4, 6}`.

```
List.Average({3, 4, 6})
```

```
4.333333333333333
```

Example 2

Find the average of the date values January 1, 2011, January 2, 2011 and January 3, 2011.

```
List.Average({#date(2011, 1, 1), #date(2011, 1, 2), #date(2011, 1, 3)})
```

```
#date(2011, 1, 2)
```

List.Buffer

3/15/2021 • 2 minutes to read

Syntax

```
List.Buffer(list as list) as list
```

About

Buffers the list `list` in memory. The result of this call is a stable list.

Example 1

Create a stable copy of the list {1..10}.

```
List.Buffer({1..10})
```

1
2
3
4
5
6
7
8
9
10

List.Combine

3/15/2021 • 2 minutes to read

Syntax

```
List.Combine(lists as list) as list
```

About

Takes a list of lists, `lists`, and merges them into a single new list.

Example 1

Combine the two simple lists {1, 2} and {3, 4}.

```
List.Combine({{1, 2}, {3, 4}})
```

1
2
3
4

Example 2

Combine the two lists, {1, 2} and {3, {4, 5}}, one of which contains a nested list.

```
List.Combine({{1, 2}, {3, {4, 5}}})
```

1
2
3
[List]

List.ConformToPageReader

3/15/2021 • 2 minutes to read

Syntax

```
List.ConformToPageReader(list as list, optional options as nullable record) as table
```

About

This function is intended for internal use only.

List.Contains

3/15/2021 • 2 minutes to read

Syntax

```
List.Contains(list as list, value as any, optional equationCriteria as any) as logical
```

About

Indicates whether the list `list` contains the value `value`. Returns true if value is found in the list, false otherwise. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find if the list {1, 2, 3, 4, 5} contains 3.

```
List.Contains({1, 2, 3, 4, 5}, 3)
```

`true`

Example 2

Find if the list {1, 2, 3, 4, 5} contains 6.

```
List.Contains({1, 2, 3, 4, 5}, 6)
```

`false`

List.ContainsAll

3/15/2021 • 2 minutes to read

Syntax

```
List.ContainsAll(list as list, values as list, optional equationCriteria as any) as logical
```

About

Indicates whether the list `list` includes all the values in another list, `values`. Returns true if value is found in the list, false otherwise. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find out if the list {1, 2, 3, 4, 5} contains 3 and 4.

```
List.ContainsAll({1, 2, 3, 4, 5}, {3, 4})
```

`true`

Example 2

Find out if the list {1, 2, 3, 4, 5} contains 5 and 6.

```
List.ContainsAll({1, 2, 3, 4, 5}, {5, 6})
```

`false`

List.ContainsAny

3/15/2021 • 2 minutes to read

Syntax

```
List.ContainsAny(list as list, values as list, optional equationCriteria as any) as logical
```

About

Indicates whether the list `list` includes any of the values in another list, `values`. Returns true if value is found in the list, false otherwise. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find out if the list {1, 2, 3, 4, 5} contains 3 or 9.

```
List.ContainsAny({1, 2, 3, 4, 5}, {3, 9})
```

`true`

Example 2

Find out if the list {1, 2, 3, 4, 5} contains 6 or 7.

```
List.ContainsAny({1, 2, 3, 4, 5}, {6, 7})
```

`false`

List.Count

3/15/2021 • 2 minutes to read

Syntax

```
List.Count(list as list) as number
```

About

Returns the number of items in the list `list`.

Example 1

Find the number of values in the list {1, 2, 3}.

```
List.Count({1, 2, 3})
```

`3`

List.Covariance

3/15/2021 • 2 minutes to read

Syntax

```
List.Covariance(numberList1 as list, numberList2 as list) as nullable number
```

About

Returns the covariance between two lists, `numberList1` and `numberList2`. `numberList1` and `numberList2` must contain the same number of `number` values.

Example 1

Calculate the covariance between two lists.

```
List.Covariance({1, 2, 3}, {1, 2, 3})
```

`0.66666666666666607`

List.Dates

3/15/2021 • 2 minutes to read

Syntax

```
List.Dates(start as date, count as number, step as duration) as list
```

About

Returns a list of `date` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 5 values starting from New Year's Eve (#date(2011, 12, 31)) incrementing by 1 day(#duration(1, 0, 0, 0)).

```
List.Dates(#date(2011, 12, 31), 5, #duration(1, 0, 0, 0))
```

12/31/2011 12:00:00 AM
1/1/2012 12:00:00 AM
1/2/2012 12:00:00 AM
1/3/2012 12:00:00 AM
1/4/2012 12:00:00 AM

List.DateTimes

3/15/2021 • 2 minutes to read

Syntax

```
List.DateTimes(start as datetime, count as number, step as duration) as list
```

About

Returns a list of `datetime` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example

Create a list of 10 values starting from 5 minutes before New Year's Day (#datetime(2011, 12, 31, 23, 55, 0)) incrementing by 1 minute (#duration(0, 0, 1, 0)).

```
List.DateTimes(#datetime(2011, 12, 31, 23, 55, 0), 10, #duration(0, 0, 1, 0))
```

12/31/2011 11:55:00 PM
12/31/2011 11:56:00 PM
12/31/2011 11:57:00 PM
12/31/2011 11:58:00 PM
12/31/2011 11:59:00 PM
1/1/2012 12:00:00 AM
1/1/2012 12:01:00 AM
1/1/2012 12:02:00 AM
1/1/2012 12:03:00 AM
1/1/2012 12:04:00 AM

List.DateTimeZones

3/15/2021 • 2 minutes to read

Syntax

```
List.DateTimeZones(start as datetimezone, count as number, step as duration) as list
```

About

Returns a list of `datetimezone` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 10 values starting from 5 minutes before New Year's Day (#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0)) incrementing by 1 minute (#duration(0, 0, 1, 0)).

```
List.DateTimeZones(#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0), 10, #duration(0, 0, 1, 0))
```

12/31/2011 11:55:00 PM -08:00
12/31/2011 11:56:00 PM -08:00
12/31/2011 11:57:00 PM -08:00
12/31/2011 11:58:00 PM -08:00
12/31/2011 11:59:00 PM -08:00
1/1/2012 12:00:00 AM -08:00
1/1/2012 12:01:00 AM -08:00
1/1/2012 12:02:00 AM -08:00
1/1/2012 12:03:00 AM -08:00
1/1/2012 12:04:00 AM -08:00

List.Difference

3/15/2021 • 2 minutes to read

```
List.Difference(list1 as list, list2 as list, optional equationCriteria as any) as list
```

About

Returns the items in list `list1` that do not appear in list `list2`. Duplicate values are supported. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the items in list {1, 2, 3, 4, 5}that do not appear in {4, 5, 3}.

```
List.Difference({1, 2, 3, 4, 5}, {4, 5, 3})
```

```
1
```

```
2
```

Example 2

Find the items in the list {1, 2} that do not appear in {1, 2, 3}.

```
List.Difference({1, 2}, {1, 2, 3})
```

List.Distinct

3/15/2021 • 2 minutes to read

Syntax

```
List.Distinct(list as list, optional equationCriteria as any) as list
```

About

Returns a list that contains all the values in list `list` with duplicates removed. If the list is empty, the result is an empty list.

Example 1

Remove the duplicates from the list {1, 1, 2, 3, 3, 3}.

```
List.Distinct({1, 1, 2, 3, 3, 3})
```

1
2
3

List.Durations

3/15/2021 • 2 minutes to read

Syntax

```
List.Durations(start as duration, count as number, step as duration) as list
```

About

Returns a list of `count` `duration` values, starting at `start` and incremented by the given `duration` `step`.

Example

Create a list of 5 values starting 1 hour and incrementing by an hour.

```
List.Durations(#duration(0, 1, 0, 0), 5, #duration(0, 1, 0, 0))
```

01:00:00
02:00:00
03:00:00
04:00:00
05:00:00

List.FindText

3/15/2021 • 2 minutes to read

Syntax

```
List.FindText(list as list, text as text) as list
```

About

Returns a list of the values from the list `list` which contained the value `text`.

Example 1

Find the text values in the list {"a", "b", "ab"} that match "a".

```
List.FindText({"a", "b", "ab"}, "a")
```

a
ab

List.First

3/15/2021 • 2 minutes to read

Syntax

```
List.First(list as list, optional defaultValue as any) as any
```

About

Returns the first item in the list `list`, or the optional default value, `defaultValue`, if the list is empty. If the list is empty and a default value is not specified, the function returns `null`.

Example 1

Find the first value in the list {1, 2, 3}.

```
List.First({1, 2, 3})
```

`1`

Example 2

Find the first value in the list {}. If the list is empty, return -1.

```
List.First({}, -1)
```

`-1`

List.FirstN

3/15/2021 • 2 minutes to read

Syntax

```
List.FirstN(list as list, countOrCondition as any) as any
```

About

- If a number is specified, up to that many items are returned.
- If a condition is specified, all items are returned that initially meet the condition. Once an item fails the condition, no further items are considered.

Example 1

Find the intial values in the list {3, 4, 5, -1, 7, 8, 2} that are greater than 0.

```
List.FirstN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

3
4
5

List.Generate

3/15/2021 • 2 minutes to read

Syntax

```
List.Generate(initial as function, condition as function, next as function, optional selector as nullable function) as list
```

About

Generates a list of values given four functions that generate the initial value `initial`, test against a condition `condition`, and if successful select the result and generate the next value `next`. An optional parameter, `selector`, may also be specified.

Example 1

Create a list that starts at 10, remains greater than 0 and decrements by 1.

```
List.Generate(() => 10, each _ > 0, each _ - 1)
```

10
9
8
7
6
5
4
3
2
1

Example 2

Generate a list of records containing x and y, where x is a value and y is a list. x should remain less than 10 and represent the number of items in the list y. After the list is generated, return only the x values.

```
List.Generate(
    () => [x = 1, y = {}],
    each [x] < 10,
    each [x = List.Count([y]), y = [y] & {x}],
    each [x]
)
```

1
0
1
2
3
4
5
6
7
8
9

List.InsertRange

3/15/2021 • 2 minutes to read

Syntax

```
List.InsertRange(list as list, index as number, values as list) as list
```

About

Returns a new list produced by inserting the values in `values` into `list` at `index`. The first position in the list is at index 0.

- `list`: The target list where values are to be inserted.
- `index`: The index of the target list(`list`) where the values are to be inserted. The first position in the list is at index 0.
- `values`: The list of values which are to be inserted into `list`.

Example 1

Insert the list ({3, 4}) into the target list ({1, 2, 5}) at index 2.

```
List.InsertRange({1, 2, 5}, 2, {3, 4})
```

1
2
3
4
5

Example 2

Insert a list with a nested list ({1, {1.1, 1.2}}) into a target list ({2, 3, 4}) at index 0.

```
List.InsertRange({2, 3, 4}, 0, {1, {1.1, 1.2}})
```

1
[List]
2
3

List.Intersect

6/14/2021 • 2 minutes to read

Syntax

```
List.Intersect(lists as list, optional equationCriteria as any) as list
```

About

Returns the intersection of the list values found in the input list `lists`. An optional parameter, `equationCriteria`, can be specified.

Example 1

Find the intersection of the lists {1..5}, {2..6}, {3..7}.

```
List.Intersect({{1..5}, {2..6}, {3..7}})
```

3
4
5

List.IsDistinct

3/15/2021 • 2 minutes to read

Syntax

```
List.IsDistinct(list as list, optional equationCriteria as any) as logical
```

About

Returns a logical value whether there are duplicates in the list `list`; `true` if the list is distinct, `false` if there are duplicate values.

Example 1

Find if the list {1, 2, 3} is distinct (i.e. no duplicates).

```
List.IsDistinct({1, 2, 3})
```

`true`

Example 2

Find if the list {1, 2, 3, 3} is distinct (i.e. no duplicates).

```
List.IsDistinct({1, 2, 3, 3})
```

`false`

List.IsEmpty

3/15/2021 • 2 minutes to read

Syntax

```
List.IsEmpty(list as list) as logical
```

About

Returns `true` if the list, `list`, contains no values (length 0). If the list contains values (length > 0), returns `false`.

Example 1

Find if the list {} is empty.

```
List.IsEmpty({})
```

`true`

Example 2

Find if the list {1, 2} is empty.

```
List.IsEmpty({1, 2})
```

`false`

List.Last

3/15/2021 • 2 minutes to read

Syntax

```
List.Last(list as list, optional defaultValue as any) as any
```

About

Returns the last item in the list `list`, or the optional default value, `defaultValue`, if the list is empty. If the list is empty and a default value is not specified, the function returns `null`.

Example 1

Find the last value in the list {1, 2, 3}.

```
List.Last({1, 2, 3})
```

`3`

Example 2

Find the last value in the list {} or -1 if it empty.

```
List.Last({}, -1)
```

`-1`

List.LastN

3/15/2021 • 2 minutes to read

Syntax

```
List.LastN(list as list, optional countOrCondition as any) as any
```

About

Returns the last item of the list `list`. If the list is empty, an exception is thrown. This function takes an optional parameter, `countOrCondition`, to support gathering multiple items or filtering items. `countOrCondition` can be specified in three ways:

- If a number is specified, up to that many items are returned.
- If a condition is specified, all items are returned that initially meet the condition, starting at the end of the list. Once an item fails the condition, no further items are considered.
- If this parameter is null the last item in the list is returned.

Example 1

Find the last value in the list {3, 4, 5, -1, 7, 8, 2}.

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, 1)
```

```
2
```

Example 2

Find the last values in the list {3, 4, 5, -1, 7, 8, 2} that are greater than 0.

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

```
7
```

```
8
```

```
2
```

List.MatchesAll

3/15/2021 • 2 minutes to read

Syntax

```
List.MatchesAll(list as list, condition as function) as logical
```

About

Returns `true` if the condition function, `condition`, is satisfied by all values in the list `list`, otherwise returns `false`.

Example 1

Determine if all the values in the list {11, 12, 13} are greater than 10.

```
List.MatchesAll({11, 12, 13}, each _  > 10)
```

`true`

Example 2

Determine if all the values in the list {1, 2, 3} are greater than 10.

```
List.MatchesAll({1, 2, 3}, each _  > 10)
```

`false`

List.MatchesAny

3/15/2021 • 2 minutes to read

Syntax

```
List.MatchesAny(list as list, condition as function) as logical
```

About

Returns `true` if the condition function, `condition`, is satisfied by any of values in the list `list`, otherwise returns `false`.

Example 1

Find if any of the values in the list {9, 10, 11} are greater than 10.

```
List.MatchesAny({9, 10, 11}, each _ > 10)
```

`true`

Example 2

Find if any of the values in the list {1, 2, 3} are greater than 10.

```
List.MatchesAny({1, 2, 3}, each _ > 10)
```

`false`

List.Max

3/15/2021 • 2 minutes to read

Syntax

```
List.Max(list as list, optional default as any, optional comparisonCriteria as any, optional
includeNulls as nullable logical) as any
```

About

Returns the maximum item in the list `list`, or the optional default value `default` if the list is empty. An optional comparisonCriteria value, `comparisonCriteria`, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the max in the list {1, 4, 7, 3, -2, 5}.

```
List.Max({1, 4, 7, 3, -2, 5}, 1)
```

`7`

Example 2

Find the max in the list {} or return -1 if it is empty.

```
List.Max({}, -1)
```

`-1`

List.MaxN

3/15/2021 • 2 minutes to read

Syntax

```
List.MaxN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as list
```

About

Returns the maximum value(s) in the list, `list`. After the rows are sorted, optional parameters may be specified to further filter the result. The optional parameter, `countOrCondition`, specifies the number of values to return or a filtering condition. The optional parameter, `comparisonCriteria`, specifies how to compare values in the list.

- `list`: The list of values.
- `countOrCondition`: If a number is specified, a list of up to `countOrCondition` items in ascending order is returned. If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.
- `comparisonCriteria`: *[Opional]* An optional `comparisonCriteria` value, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

List.Median

3/15/2021 • 2 minutes to read

Syntax

```
List.Median(list as list, optional comparisonCriteria as any) as any
```

About

Returns the median item of the list `list`. This function returns `null` if the list contains no non-`null` values. If there is an even number of items, the function chooses the smaller of the two median items unless the list is comprised entirely of datetimes, durations, numbers or times, in which case it returns the average of the two items.

Example 1

Find the median of the list `{5, 3, 1, 7, 9}`.

```
powerquery-mList.Median({5, 3, 1, 7, 9})
```

`5`

List.Min

3/15/2021 • 2 minutes to read

Syntax

```
List.Min(list as list, optional default as any, optional comparisonCriteria as any, optional
includeNulls as nullable logical) as any
```

About

Returns the minimum item in the list `list`, or the optional default value `default` if the list is empty. An optional comparisonCriteria value, `comparisonCriteria`, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the min in the list {1, 4, 7, 3, -2, 5}.

```
List.Min({1, 4, 7, 3, -2, 5})
```

`-2`

Example 2

Find the min in the list {} or return -1 if it is empty.

```
List.Min({}, -1)
```

`-1`

List.MinN

3/15/2021 • 2 minutes to read

Syntax

```
List.MinN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as list
```

About

Returns the minimum value(s) in the list, `list`. The parameter, `countOrCondition`, specifies the number of values to return or a filtering condition. The optional parameter, `comparisonCriteria`, specifies how to compare values in the list.

- `list`: The list of values.
- `countOrCondition`: If a number is specified, a list of up to `countOrCondition` items in ascending order is returned. If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered. If this parameter is null the single smallest value in the list is returned.
- `comparisonCriteria`: *[Opional]* An optional `comparisonCriteria` value, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the 5 smallest values in the list `{3, 4, 5, -1, 7, 8, 2}`.

```
List.MinN({3, 4, 5, -1, 7, 8, 2}, 5)
```

-1
2
3
4
5

List.Mode

3/15/2021 • 2 minutes to read

Syntax

```
List.Mode(list as list, optional equationCriteria as any) as any
```

About

Returns the item that appears most frequently in the list, `list`. If the list is empty an exception is thrown. If multiple items appear with the same maximum frequency, the last one is chosen. An optional `comparisonCriteria` value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the item that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5}`.

```
List.Mode({"A", 1, 2, 3, 3, 4, 5})
```

`3`

Example 2

Find the item that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5, 5}`.

```
List.Mode({"A", 1, 2, 3, 3, 4, 5, 5})
```

`5`

List.Modes

3/15/2021 • 2 minutes to read

Syntax

```
List.Modes(list as list, optional equationCriteria as any) as list
```

About

Returns the item that appears most frequently in the list, `list`. If the list is empty an exception is thrown. If multiple items appear with the same maximum frequency, the last one is chosen. An optional `comparisonCriteria` value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the items that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5, 5}`.

```
List.Modes({"A", 1, 2, 3, 3, 4, 5, 5})
```

```
3
```

```
5
```

List.NonNullCount

3/15/2021 • 2 minutes to read

Syntax

```
List.NonNullCount(list as list) as number
```

About

Returns the number of non-null items in the list `list`.

List.Numbers

3/15/2021 • 2 minutes to read

Syntax

```
List.Numbers(start as number, count as number, optional increment as nullable number) as list
```

About

Returns a list of numbers given an initial value, count, and optional increment value. The default increment value is 1.

- `start`: The initial value in the list.
- `count`: The number of values to create.
- `increment`: *[Optional]* The value to increment by. If omitted values are incremented by 1.

Example 1

Generate a list of 10 consecutive numbers starting at 1.

```
List.Numbers(1, 10)
```

1
2
3
4
5
6
7
8
9
10

Example 2

Generate a list of 10 numbers starting at 1, with an increment of 2 for each subsequent number.

```
List.Numbers(1, 10, 2)
```

1
3
5
7
9
11
13
15
17
19

List.Percentile

3/15/2021 • 2 minutes to read

Syntax

```
List.Percentile(list as list, percentiles as any, optional options as nullable record) as any
```

About

Returns one or more sample percentiles of the list `list`. If the value `percentiles` is a number between 0.0 and 1.0, it will be treated as a percentile and the result will be a single value corresponding to that probability. If the value `percentiles` is a list of numbers with values between 0.0 and 1.0, the result will be a list of percentiles corresponding to the input probability.

The PercentileMode option in `options` can be used by advanced users to pick a more-specific interpolation method but is not recommended for most uses. Predefined symbols `PercentileMode.ExcelInc` and `PercentileMode.ExcelExc` match the interpolation methods used by the Excel functions `PERCENTILE.INC` and `PERCENTILE.EXC`. The default behavior matches `PercentileMode.ExcelInc`. The symbols `PercentileMode.SqlCont` and `PercentileMode.SqlDisc` match the SQL Server behavior for `PERCENTILE_CONT` and `PERCENTILE_DISC`, respectively.

Example 1

Find the first quartile of the list `{5, 3, 1, 7, 9}`.

```
List.Percentile({5, 3, 1, 7, 9}, 0.25)
```

`3`

Example 2

Find the quartiles of the list `{5, 3, 1, 7, 9}` using an interpolation method matching Excel's `PERCENTILE.EXC`.

```
List.Percentile({5, 3, 1, 7, 9}, {0.25, 0.5, 0.75}, [PercentileMode=PercentileMode.ExcelExc])
```

`{2, 5, 8}`

List.PositionOf

3/15/2021 • 2 minutes to read

Syntax

```
List.PositionOf(list as list, value as any, optional occurrence as nullable number, optional equationCriteria as any) as any
```

About

Returns the offset at which the value `value` appears in the list `list`. Returns -1 if the value doesn't appear. An optional occurrence parameter `occurrence` can be specified.

- `occurrence`: The maximum number of occurrences to report.

Example 1

Find the position in the list {1, 2, 3} at which the value 3 appears.

```
List.PositionOf({1, 2, 3}, 3)
```

`2`

List.PositionOfAny

3/15/2021 • 2 minutes to read

Syntax

```
List.PositionOfAny(list as list, values as list, optional occurrence as nullable number,
optional equationCriteria as any) as any
```

About

Returns the offset in list `list` of the first occurrence of a value in a list `values`. Returns -1 if no occurrence is found. An optional occurrence parameter `occurrence` can be specified.

- `occurrence`: The maximum number of occurrences that can be returned.

Example 1

Find the first position in the list {1, 2, 3} at which the value 2 or 3 appears.

```
List.PositionOfAny({1, 2, 3}, {2, 3})
```

`1`

List.Positions

3/15/2021 • 2 minutes to read

Syntax

```
List.Positions(list as list) as list
```

About

Returns a list of offsets for the input list `list`. When using List.Transform to change a list, the list of positions can be used to give the transform access to the position.

Example 1

Find the offsets of values in the list {1, 2, 3, 4, null, 5}.

```
List.Positions({1, 2, 3, 4, null, 5})
```

0
1
2
3
4
5

List.Product

3/15/2021 • 2 minutes to read

Syntax

```
List.Product(numbersList as list, optional precision as nullable number) as nullable number
```

About

Returns the product of the non-null numbers in the list, `numbersList`. Returns null if there are no non-null values in the list.

Example 1

Find the product of the numbers in the list `{1, 2, 3, 3, 4, 5, 5}`.

```
List.Product({1, 2, 3, 3, 4, 5, 5})
```

```
1800
```

List.Random

3/15/2021 • 2 minutes to read

Syntax

```
List.Random(count as number, optional seed as nullable number) as list
```

About

Returns a list of random numbers between 0 and 1, given the number of values to generate and an optional seed value.

- `count`: The number of random values to generate.
- `seed`: *[Optional]* A numeric value used to seed the random number generator. If omitted a unique list of random numbers is generated each time you call the function. If you specify the seed value with a number every call to the function generates the same list of random numbers.

Example 1

Create a list of 3 random numbers.

```
List.Random(3)
```

0.992332
0.132334
0.023592

Example 2

Create a list of 3 random numbers, specifying seed value.

```
List.Random(3, 2)
```

0.883002
0.245344
0.723212

List.Range

3/15/2021 • 2 minutes to read

Syntax

```
List.Range(list as list, offset as number, optional count as nullable number) as list
```

About

Returns a subset of the list beginning at the offset `list`. An optional parameter, `offset`, sets the maximum number of items in the subset.

Example 1

Find the subset starting at offset 6 of the list of numbers 1 through 10.

```
List.Range({1..10}, 6)
```

7
8
9
10

Example 2

Find the subset of length 2 from offset 6, from the list of numbers 1 through 10.

```
List.Range({1..10}, 6, 2)
```

7
8

List.RemoveFirstN

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveFirstN(list as list, optional countOrCondition as any) as list
```

About

Returns a list that removes the first element of list `list`. If `list` is an empty list an empty list is returned. This function takes an optional parameter, `countOrCondition`, to support removing multiple values as listed below.

- If a number is specified, up to that many items are removed.
- If a condition is specified, the returned list begins with the first element in `list` that meets the criteria. Once an item fails the condition, no further items are considered.
- If this parameter is null, the default behavior is observed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the first 3 numbers.

```
List.RemoveFirstN({1, 2, 3, 4, 5}, 3)
```

4
5

Example 2

Create a list from {5, 4, 2, 6, 1} that starts with a number less than 3.

```
List.RemoveFirstN({5, 4, 2, 6, 1}, each _ > 3)
```

2
6
1

List.RemoveItems

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveItems(list1 as list, list2 as list) as list
```

About

Removes all occurrences of the given values in the `list2` from `list1`. If the values in `list2` don't exist in `list1`, the original list is returned.

Example 1

Remove the items in the list {2, 4, 6} from the list {1, 2, 3, 4, 2, 5, 5}.

```
List.RemoveItems({1, 2, 3, 4, 2, 5, 5}, {2, 4, 6})
```

1
3
5
5

List.RemoveLastN

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveLastN(list as list, optional countOrCondition as any) as list
```

About

Returns a list that removes the last `countOrCondition` elements from the end of list `list`. If `list` has less than `countOrCondition` elements, an empty list is returned.

- If a number is specified, up to that many items are removed.
- If a condition is specified, the returned list ends with the first element from the bottom in `list` that meets the criteria. Once an item fails the condition, no further items are considered.
- If this parameter is null, only one item is removed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the last 3 numbers.

```
List.RemoveLastN({1, 2, 3, 4, 5}, 3)
```

1
2

Example 2

Create a list from {5, 4, 2, 6, 4} that ends with a number less than 3.

```
List.RemoveLastN({5, 4, 2, 6, 4}, each _ > 3)
```

5
4
2

List.RemoveMatchingItems

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveMatchingItems(list1 as list, list2 as list, optional equationCriteria as any) as list
```

About

Removes all occurrences of the given values in `list2` from the list `list1`. If the values in `list2` don't exist in `list1`, the original list is returned. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a list from {1, 2, 3, 4, 5, 5} without {1, 5}.

```
List.RemoveMatchingItems({1, 2, 3, 4, 5, 5}, {1, 5})
```

2
3
4

List.RemoveNulls

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveNulls(list as list) as list
```

About

Removes all occurrences of "null" values in the `list`. If there are no 'null' values in the list, the original list is returned.

Example 1

Remove the "null" values from the list {1, 2, 3, null, 4, 5, null, 6}.

```
List.RemoveNulls({1, 2, 3, null, 4, 5, null, 6})
```

1
2
3
4
5
6

List.RemoveRange

3/15/2021 • 2 minutes to read

Syntax

```
List.RemoveRange(list as list, index as number, optional count as nullable number) as list
```

About

Removes `count` values in the `list` starting at the specified position, `index`.

Example 1

Remove 3 values in the list {1, 2, 3, 4, -6, -2, -1, 5} starting at index 4.

```
List.RemoveRange({1, 2, 3, 4, -6, -2, -1, 5}, 4, 3)
```

1
2
3
4
5

List.Repeat

3/15/2021 • 2 minutes to read

Syntax

```
List.Repeat(list as list, count as number) as list
```

About

Returns a list that is `count` repetitions of the original list, `list`.

Example 1

Create a list that has {1, 2} repeated 3 times.

```
List.Repeat({1, 2}, 3)
```

1
2
1
2
1
2

List.ReplaceMatchingItems

3/15/2021 • 2 minutes to read

Syntax

```
List.ReplaceMatchingItems(list as list, replacements as list, optional equationCriteria as any)
as list
```

About

Performs the given replacements to the list `list`. A replacement operation `replacements` consists of a list of two values, the old value and new value, provided in a list. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a list from {1, 2, 3, 4, 5} replacing the value 5 with -5, and the value 1 with -1.

```
List.ReplaceMatchingItems({1, 2, 3, 4, 5}, {{5, -5}, {1, -1}})
```

-1
2
3
4
-5

List.ReplaceRange

3/15/2021 • 2 minutes to read

Syntax

```
List.ReplaceRange(list as list, index as number, count as number, replaceWith as list) as list
```

About

Replaces `count` values in the `list` with the list `replaceWith`, starting at specified position, `index`.

Example 1

Replace {7, 8, 9} in the list {1, 2, 7, 8, 9, 5} with {3, 4}.

```
List.ReplaceRange({1, 2, 7, 8, 9, 5}, 2, 3, {3, 4})
```

1
2
3
4
5

List.ReplaceValue

3/15/2021 • 2 minutes to read

Syntax

```
List.ReplaceValue(list as list, oldValue as any, newValue as any, replacer as function) as list
```

About

Searches a list of values, `list`, for the value `oldValue` and replaces each occurrence with the replacement value `newValue`.

Example 1

Replace all the "a" values in the list {"a", "B", "a", "a"} with "A".

```
List.ReplaceValue({"a", "B", "a", "a"}, "a", "A", Replacer.ReplaceText)
```

A
B
A
A

List.Reverse

3/15/2021 • 2 minutes to read

Syntax

```
List.Reverse(list as list) as list
```

About

Returns a list with the values in the list `list` in reversed order.

Example 1

Create a list from {1..10} in reverse order.

```
List.Reverse({1..10})
```

10
9
8
7
6
5
4
3
2
1

List.Select

3/15/2021 • 2 minutes to read

Syntax

```
List.Select(list as list, selection as function) as list
```

About

Returns a list of values from the list `list`, that match the selection condition `selection`.

Example 1

Find the values in the list {1, -3, 4, 9, -2} that are greater than 0.

```
List.Select({1, -3, 4, 9, -2}, each _ > 0)
```

1
4
9

List.Single

3/15/2021 • 2 minutes to read

Syntax

```
List.Single(list as list) as any
```

About

If there is only one item in the list `list`, returns that item. If there is more than one item or the list is empty, the function throws an exception.

Example 1

Find the single value in the list {1}.

```
List.Single({1})
```

```
1
```

Example 2

Find the single value in the list {1, 2, 3}.

```
List.Single({1, 2, 3})
```

```
[Expression.Error] There were too many elements in the enumeration to complete the operation.
```

List.SingleOrDefault

3/15/2021 • 2 minutes to read

Syntax

```
List.SingleOrDefault(list as list, optional default as any) as any
```

About

If there is only one item in the list `list`, returns that item. If the list is empty, the function returns null unless an optional `default` is specified. If there is more than one item in the list, the function returns an error.

Example 1

Find the single value in the list {1}.

```
List.SingleOrDefault({1})
```

```
1
```

Example 2

Find the single value in the list {}.

```
List.SingleOrDefault({})
```

```
null
```

Example 3

Find the single value in the list {}. If is empty, return -1.

```
List.SingleOrDefault({}, -1)
```

```
-1
```

List.Skip

3/15/2021 • 2 minutes to read

Syntax

```
List.Skip(list as list, optional countOrCondition as any) as list
```

About

Returns a list that skips the first element of list `list`. If `list` is an empty list an empty list is returned. This function takes an optional parameter, `countOrCondition`, to support skipping multiple values as listed below.

- If a number is specified, up to that many items are skipped.
- If a condition is specified, the returned list begins with the first element in `list` that meets the criteria. Once an item fails the condition, no further items are considered.
- If this parameter is null, the default behavior is observed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the first 3 numbers.

```
List.Skip({1, 2, 3, 4, 5}, 3)
```

4
5

Example 2

Create a list from {5, 4, 2, 6, 1} that starts with a number less than 3.

```
List.Skip({5, 4, 2, 6, 1}, each _ > 3)
```

2
6
1

List.Sort

3/15/2021 • 2 minutes to read

Syntax

```
List.Sort(list as list, optional comparisonCriteria as any) as list
```

About

Sorts a list of data, `list`, according to the optional criteria specified. An optional parameter, `comparisonCriteria`, can be specified as the comparison criterion. This can take the following values:

- To control the order, the comparison criterion can be an Order enum value. (`Order.Descending`, `Order.Ascending`).
- To compute a key to be used for sorting, a function of 1 argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order (`{each 1 / _, Order.Descending}`).
- To completely control the comparison, a function of 2 arguments can be used that returns -1, 0, or 1 given the relationship between the left and right inputs. Value.Compare is a method that can be used to delegate this logic.

Example 1

Sort the list {2, 3, 1}.

```
List.Sort({2, 3, 1})
```

1
2
3

Example 2

Sort the list {2, 3, 1} in descending order.

```
List.Sort({2, 3, 1}, Order.Descending)
```

3
2
1

Example 3

Sort the list {2, 3, 1} in descending order using the Value.Compare method.

```
List.Sort({2, 3, 1}, (x, y) => Value.Compare(1/x, 1/y))
```

3
2
1

List.Split

3/15/2021 • 2 minutes to read

Syntax

```
List.Split(list as list, pageSize as number) as list
```

About

Splits `list` into a list of lists where the first element of the output list is a list containing the first `pageSize` elements from the source list, the next element of the output list is a list containing the next `pageSize` elements from the source list, etc.

List.StandardDeviation

3/15/2021 • 2 minutes to read

Syntax

```
List.StandardDeviation(numbersList as list) as nullable number
```

About

Returns a sample based estimate of the standard deviation of the values in the list, `numbersList`. If `numbersList` is a list of numbers, a number is returned. An exception is thrown on an empty list or a list of items that is not type `number`.

Example 1

Find the standard deviation of the numbers 1 through 5.

```
List.StandardDeviation({1..5})
```

```
1.5811388300841898
```

List.Sum

3/15/2021 • 2 minutes to read

Syntax

```
List.Sum(list as list, optional precision as nullable number) as any
```

About

Returns the sum of the non-null values in the list, `list`. Returns null if there are no non-null values in the list.

Example 1

Find the sum of the numbers in the list `{1, 2, 3}`.

```
List.Sum({1, 2, 3})
```

`6`

List.Times

3/15/2021 • 2 minutes to read

Syntax

```
List.Times(start as time, count as number, step as duration) as list
```

About

Returns a list of `time` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 4 values starting from noon (#time(12, 0, 0)) incrementing by one hour (#duration(0, 1, 0, 0)).

```
List.Times(#time(12, 0, 0), 4, #duration(0, 1, 0, 0))
```

12:00:00
13:00:00
14:00:00
15:00:00

List.Transform

3/15/2021 • 2 minutes to read

Syntax

```
List.Transform(list as list, transform as function) as list
```

About

Returns a new list of values by applying the transform function `transform` to the list, `list`.

Example 1

Add 1 to each value in the list {1, 2}.

```
List.Transform({1, 2}, each _ + 1)
```

```
2
```

```
3
```

List.TransformMany

3/15/2021 • 2 minutes to read

Syntax

```
List.TransformMany(list as list, collectionTransform as function, resultTransform as function)
as list
```

About

Returns a list whose elements are projected from the input list. The `collectionTransform` function is applied to each element, and the `resultTransform` function is invoked to construct the resulting list. The `collectionSelector` has the signature (x as Any) => ... where x is an element in list. The `resultTransform` projects the shape of the result and has the signature (x as Any, y as Any) => ... where x is the element in list and y is the element obtained by applying the `collectionTransform` to that element.

List.Union

3/15/2021 • 2 minutes to read

Syntax

```
List.Union(lists as list, optional equationCriteria as any) as list
```

About

Takes a list of lists `lists`, unions the items in the individual lists and returns them in the output list. As a result, the returned list contains all items in any input lists. This operation maintains traditional bag semantics, so duplicate values are matched as part of the Union. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a union of the list {1..5}, {2..6}, {3..7}.

```
List.Union({{1..5}, {2..6}, {3..7}})
```

1
2
3
4
5
6
7

List.Zip

3/15/2021 • 2 minutes to read

Syntax

```
List.Zip(lists as list) as list
```

About

Takes a list of lists, `lists`, and returns a list of lists combining items at the same position.

Example 1

Zips the two simple lists {1, 2} and {3, 4}.

```
List.Zip({{1, 2}, {3, 4}})
```

[List]

[List]

Example 2

Zips the two simple lists of different lengths {1, 2} and {3}.

```
List.Zip({{1, 2}, {3}})
```

[List]

[List]

PercentileMode.ExcelExc

3/15/2021 • 2 minutes to read

About

When interpolating values for `List.Percentile`, use a method compatible with Excel's `PERCENTILE.EXC`.

PercentileMode.ExcelInc

3/15/2021 • 2 minutes to read

About

When interpolating values for `List.Percentile`, use a method compatible with Excel's `PERCENTILE.INC`.

PercentileMode.SqlCont

3/15/2021 • 2 minutes to read

About

When interpolating values for `List.Percentile`, use a method compatible with SQL Server's `PERCENTILE_CONT`.

PercentileMode.SqlDisc

3/15/2021 • 2 minutes to read

About

When interpolating values for `List.Percentile`, use a method compatible with SQL Server's `PERCENTILE_DISC`.

Logical functions

3/15/2021 • 2 minutes to read

These functions create and manipulate logical (that is, true/false) values.

Logical

FUNCTION	DESCRIPTION
Logical.From	Returns a logical value from a value.
Logical.FromText	Returns a logical value of true or false from a text value.
Logical.ToText	Returns a text value from a logical value.

Logical.From

3/15/2021 • 2 minutes to read

Syntax

```
Logical.From(value as any) as nullable logical
```

About

Returns a `logical` value from the given `value`. If the given `value` is `null`, `Logical.From` returns `null`. If the given `value` is `logical`, `value` is returned.

Values of the following types can be converted to a `logical` value:

- `text`: A `logical` value from the text value, either `"true"` or `"false"`. See `Logical.FromText` for details.
- `number`: `false` if `value` equals `0`, `true` otherwise.

If `value` is of any other type, an error is returned.

Example 1

Convert `2` to a `logical` value.

```
Logical.From(2)
```

`true`

Logical.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Logical.FromText(text as nullable text) as nullable logical
```

About

Creates a logical value from the text value `text`, either "true" or "false". If `text` contains a different string, an exception is thrown. The text value `text` is case insensitive.

Example 1

Create a logical value from the text string "true".

```
Logical.FromText("true")
```

```
true
```

Example 2

Create a logical value from the text string "a".

```
Logical.FromText("a")
```

```
[Expression.Error] Could not convert to a logical.
```

Logical.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Logical.ToText(logicalValue as nullable logical) as nullable text
```

About

Creates a text value from the logical value `logicalValue`, either `true` or `false`. If `logicalValue` is not a logical value, an exception is thrown.

Example 1

Create a text value from the logical `true`.

```
Logical.ToText(true)
```

`"true"`

Number functions

3/15/2021 • 3 minutes to read

These functions create and manipulate number values.

Number

Constants

FUNCTION	DESCRIPTION
Number.E	Returns 2.7182818284590451, the value of e up to 16 decimal digits.
Number.Epsilon	Returns the smallest possible number.
Number.NaN	Represents 0/0.
Number.NegativeInfinity	Represents -1/0.
Number.PI	Returns 3.1415926535897931, the value for Pi up to 16 decimal digits.
Number.PositiveInfinity	Represents 1/0.

Information

FUNCTION	DESCRIPTION
Number.IsEven	Returns true if a value is an even number.
Number.IsNaN	Returns true if a value is Number.NaN.
Number.IsOdd	Returns true if a value is an odd number.

Conversion and formatting

FUNCTION	DESCRIPTION
Byte.From	Returns a 8-bit integer number value from the given value.
Currency.From	Returns a currency value from the given value.
Decimal.From	Returns a decimal number value from the given value.
Double.From	Returns a Double number value from the given value.
Int8.From	Returns a signed 8-bit integer number value from the given value.

FUNCTION	DESCRIPTION
Int16.From	Returns a 16-bit integer number value from the given value.
Int32.From	Returns a 32-bit integer number value from the given value.
Int64.From	Returns a 64-bit integer number value from the given value.
Number.From	Returns a number value from a value.
Number.FromText	Returns a number value from a text value.
Number.ToText	Returns a text value from a number value.
Percentage.From	Returns a percentage value from the given value.
Single.From	Returns a Single number value from the given value.

Rounding

FUNCTION	DESCRIPTION
Number.Round	Returns a nullable number (n) if value is an integer.
Number.RoundAwayFromZero	Returns Number.RoundUp(value) when value >= 0 and Number.RoundDown(value) when value < 0.
Number.RoundDown	Returns the largest integer less than or equal to a number value.
Number.RoundTowardZero	Returns Number.RoundDown(x) when x >= 0 and Number.RoundUp(x) when x < 0.
Number.RoundUp	Returns the larger integer greater than or equal to a number value.

Operations

FUNCTION	DESCRIPTION
Number.Abs	Returns the absolute value of a number.
Number.Combinations	Returns the number of combinations of a given number of items for the optional combination size.
Number.Exp	Returns a number representing *e* raised to a power.
Number.Factorial	Returns the factorial of a number.
Number.IntegerDivide	Divides two numbers and returns the whole part of the resulting number.
Number.Ln	Returns the natural logarithm of a number.

FUNCTION	DESCRIPTION
Number.Log	Returns the logarithm of a number to the base.
Number.Log10	Returns the base-10 logarithm of a number.
Number.Mod	Divides two numbers and returns the remainder of the resulting number.
Number.Permutations	Returns the number of total permutatons of a given number of items for the optional permutation size.
Number.Power	Returns a number raised by a power.
Number.Sign	Returns 1 for positive numbers, -1 for negative numbers or 0 for zero.
Number.Sqrt	Returns the square root of a number.

Random

FUNCTION	DESCRIPTION
Number.Random	Returns a random fractional number between 0 and 1.
Number.RandomBetween	Returns a random number between the two given number values.

Trigonometry

FUNCTION	DESCRIPTION
Number.Acos	Returns the arccosine of a number.
Number.Asin	Returns the arcsine of a number.
Number.Atan	Returns the arctangent of a number.
Number.Atan2	Returns the arctangent of the division of two numbers.
Number.Cos	Returns the cosine of a number.
Number.Cosh	Returns the hyperbolic cosine of a number.
Number.Sin	Returns the sine of a number.
Number.Sinh	Returns the hyperbolic sine of a number.
Number.Tan	Returns the tangent of a number.
Number.Tanh	Returns the hyperbolic tangent of a number.

Bytes

FUNCTION	DESCRIPTION
Number.BitwiseAnd	Returns the result of a bitwise AND operation on the provided operands.
Number.BitwiseNot	Returns the result of a bitwise NOT operation on the provided operands.
Number.BitwiseOr	Returns the result of a bitwise OR operation on the provided operands.
Number.BitwiseShiftLeft	Returns the result of a bitwise shift left operation on the operands.
Number.BitwiseShiftRight	Returns the result of a bitwise shift right operation on the operands.
Number.BitwiseXor	Returns the result of a bitwise XOR operation on the provided operands.

PARAMETER VALUES	DESCRIPTION
RoundingMode.AwayFromZero	RoundingMode.AwayFromZero
RoundingMode.Down	RoundingMode.Down
RoundingMode.ToEven	RoundingMode.ToEven
RoundingMode.TowardZero	RoundingMode.TowardZero
RoundingMode.Up	RoundingMode.Up

Byte.From

3/15/2021 • 2 minutes to read

Syntax

```
Byte.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

About

Returns a 8-bit integer `number` > value from the given `value`. If the given `value` > is `null`, `Byte.From` returns `null`. If the given `value` is `number` within the range of 8-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is `RoundingMode.ToEven`. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to 8-bit integer `number` value applies.See `Number.Round` for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 8-bit integer `number` value of `"4"`.

```
Byte.From("4")
```

`4`

Example 2

Get the 8-bit integer `number` value of `"4.5"` using `RoundingMode.AwayFromZero`.

```
Byte.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

=

Currency.From

3/15/2021 • 2 minutes to read

Syntax

```
Currency.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a `currency` value from the given `value`. If the given `value` is `null`, `Currency.From` returns `null`. If the given `value` is `number` within the range of currency, fractional part of the `value` is rounded to 4 decimal digits and returned. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to `currency` value applies. Valid range for currency is `-922,337,203,685,477.5808` to `922,337,203,685,477.5807`. See `Number.Round` for the available rounding modes. The default is `RoundingMode.ToEven`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the `currency` value of `"1.23455"`.

```
Currency.From("1.23455")
```

`1.2346`

Example 2

Get the `currency` value of `"1.23455"` using `RoundingMode.Down`.

```
Currency.From("1.23455", "en-Us", RoundingMode.Down)
```

`1.2345`

Decimal.From

3/15/2021 • 2 minutes to read

Syntax

```
Decimal.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Decimal `number` value from the given `value`. If the given `value` is `null`, `Decimal.From` returns `null`. If the given `value` is `number` within the range of Decimal, `value` is returned, otherwise an error is returned. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to Decimal `number` value applies. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the Decimal `number` value of `"4.5"`.

```
Decimal.From("4.5")
```

`4.5`

Double.From

3/15/2021 • 2 minutes to read

Syntax

```
Double.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Double `number` value from the given `value`. If the given `value` is `null`, `Double.From` returns `null`. If the given `value` is `number` within the range of Double, `value` is returned, otherwise an error is returned. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to Double `number` value applies. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the Double `number` value of `"4"`.

```
Double.From("4.5")
```

`4.5`

Int8.From

3/15/2021 • 2 minutes to read

Syntax

```
Int8.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

About

Returns a signed 8-bit integer `number` value from the given `value`. If the given `value` is `null`, `Int8.From` returns `null`. If the given `value` is `number` within the range of signed 8-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is `RoundingMode.ToEven`. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to signed 8-bit integer `number` value applies. See `Number.Round` for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the signed 8-bit integer `number` value of `"4"`.

```
Int8.From("4")
```

`4`

Example 2

Get the signed 8-bit integer `number` value of `"4.5"` using `RoundingMode.AwayFromZero`.

```
Int8.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int16.From

3/15/2021 • 2 minutes to read

Syntax

```
Int16.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

About

Returns a 16-bit integer `number` value from the given `value`. If the given `value` is `null`, `Int16.From` returns `null`. If the given `value` is `number` within the range of 16-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is `RoundingMode.ToEven`. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to 16-bit integer `number` value applies. See `Number.Round` for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 16-bit integer `number` value of `"4"`.

```
Int64.From("4")
```

`4`

Example 2

Get the 16-bit integer `number` value of `"4.5"` using `RoundingMode.AwayFromZero`.

```
Int16.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int32.From

3/15/2021 • 2 minutes to read

Syntax

```
Int32.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

About

Returns a 32-bit integer `number` value from the given `value`. If the given `value` is `null`, `Int32.From` returns `null`. If the given `value` is `number` within the range of 32-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is `RoundingMode.ToEven`. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to 32-bit integer `number` value applies. See `Number.Round` for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 32-bit integer `number` value of `"4"`.

```
Int32.From("4")
```

`4`

Example 2

Get the 32-bit integer `number` value of `"4.5"` using `RoundingMode.AwayFromZero`.

```
Int32.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int64.From

3/15/2021 • 2 minutes to read

Syntax

```
Int64.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

About

Returns a 64-bit integer `number` value from the given `value`. If the given `value` is `null`, `Int64.From` returns `null`. If the given `value` is `number` within the range of 64-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is `RoundingMode.ToEven`. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to 64-bit integer `number` value applies. See `Number.Round` for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 64-bit integer `number` value of `"4"`.

```
Int64.From("4")
```

`4`

Example 2

Get the 64-bit integer `number` value of `"4.5"` using `RoundingMode.AwayFromZero`.

```
Int64.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Number.Abs

3/15/2021 • 2 minutes to read

Syntax

```
Number.Abs(number as nullable number) as nullable number
```

About

Returns the absolute value of `number`. If `number` is null, `Number.Abs` returns null.

- `number`: A `number` for which the absolute value is to be calculated.

Example 1

Absolute value of -3.

```
Number.Abs(-3)
```

`3`

Number.Acos

3/15/2021 • 2 minutes to read

Syntax

```
Number.Acos(number as nullable number) as nullable number
```

About

Returns the arccosine of `number`.

Number.Asin

3/15/2021 • 2 minutes to read

Syntax

```
Number.Asin(number as nullable number) as nullable number
```

About

Returns the arcsine of `number`.

Number.Atan

3/15/2021 • 2 minutes to read

Syntax

```
Number.Atan(number as nullable number) as nullable number
```

About

Returns the arctangent of `number`.

Number.Atan2

3/15/2021 • 2 minutes to read

Syntax

```
Number.Atan2(y as nullable number, x as nullable number) as nullable number
```

About

Returns the arctangent of the division of the two numbers, `y` and `x`. The division will be constructed as `y / x`.

Number.BitwiseAnd

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseAnd(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise "And" operation between `number1` and `number2`.

Number.BitwiseNot

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseNot(number as any) as any
```

About

Returns the result of performing a bitwise "Not" operation on `number`.

Number.BitwiseOr

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseOr(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise "Or" between `number1` and `number2`.

Number.BitwiseShiftLeft

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseShiftLeft(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise shift to the left on `number1`, by the specified number of bits `number2`.

Number.BitwiseShiftRight

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseShiftRight(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise shift to the right on `number1`, by the specified number of bits `number2`.

Number.BitwiseXor

3/15/2021 • 2 minutes to read

Syntax

```
Number.BitwiseXor(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise "XOR" (Exclusive-OR) between `number1` and `number2`.

Number.Combinations

3/15/2021 • 2 minutes to read

Syntax

```
Number.Combinations(setSize as nullable number, combinationSize as nullable number) as nullable number
```

About

Returns the number of unique combinations from a list of items, `setSize` with specified combination size, `combinationSize`.

- `setSize`: The number of items in the list.
- `combinationSize`: The number of items in each combination.

Example 1

Find the number of combinations from a total of 5 items when each combination is a group of 3.

```
Number.Combinations(5, 3)
```

`10`

Number.Cos

3/15/2021 • 2 minutes to read

Syntax

```
Number.Cos(number as nullable number) as nullable number
```

About

Returns the cosine of `number`.

Example 1

Find the cosine of the angle 0.

```
Number.Cos(0)
```

```
1
```

Number.Cosh

3/15/2021 • 2 minutes to read

Syntax

```
Number.Cosh(number as nullable number) as nullable number
```

About

Returns the hyperbolic cosine of `number`.

Number.E

3/15/2021 • 2 minutes to read

About

A constant that represents 2.7182818284590451, the value for e up to 16 decimal digits.

Number.Epsilon

3/15/2021 • 2 minutes to read

About

A constant value that represents the smallest positive number a floating-point number can hold.

Number.Exp

3/15/2021 • 2 minutes to read

Syntax

```
Number.Exp(number as nullable number) as nullable number
```

About

Returns the result of raising e to the power of `number` (exponential function).

- `number`: A `number` for which the exponential function is to be calculated. If `number` is null, `Number.Exp` returns null.

Example 1

Raise e to the power of 3.

```
Number.Exp(3)
```

```
20.085536923187668
```

Number.Factorial

3/15/2021 • 2 minutes to read

Syntax

```
Number.Factorial(number as nullable number) as nullable number
```

About

Returns the factorial of the number `number`.

Example 1

Find the factorial of 10.

```
Number.Factorial(10)
```

```
3628800
```

Number.From

3/15/2021 • 2 minutes to read

Syntax

```
Number.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a `number` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, `Number.From` returns `null`. If the given `value` is `number`, `value` is returned. Values of the following types can be converted to a `number` value:

- `text`: A `number` value from textual representation. Common text formats are handled ("15", "3,423.10", "5.0E-10"). See `Number.FromText` for details.
- `logical`: 1 for `true`, 0 for `false`.
- `datetime`: A double-precision floating-point number that contains an OLE Automation date equivalent.
- `datetimezone`: A double-precision floating-point number that contains an OLE Automation date equivalent of the local date and time of `value`.
- `date`: A double-precision floating-point number that contains an OLE Automation date equivalent.
- `time`: Expressed in fractional days.
- `duration`: Expressed in whole and fractional days.

If `value` is of any other type, an error is returned.

Example 1

Get the `number` value of `"4"`.

```
powerquery-mNumber.From("4")
```

```
4
```

Example 2

Get the `number` value of `#datetime(2020, 3, 20, 6, 0, 0)`.

```
Number.From(#datetime(2020, 3, 20, 6, 0, 0))
```

```
43910.25
```

Example 3

Get the `number` value of `"12.3%"`.

```
Number.From("12.3%")
```

```
0.123
```

Number.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Number.FromText(text as nullable text, optional culture as nullable text) as nullable number
```

About

Returns a `number` value from the given text value, `text`.

- `text`: The textual representation of a number value. The representation must be in a common number format, such as "15", "3,423.10", or "5.0E-10".
- `culture`: An optional culture that controls how `text` is interpreted (for example, "en-US").

Example 1

Get the number value of `"4"`.

```
Number.FromText("4")
```

`4`

Example 2

Get the number value of `"5.0e-10"`.

```
Number.FromText("5.0e-10")
```

`5E-10`

Number.IntegerDivide

3/15/2021 • 2 minutes to read

Syntax

```
Number.IntegerDivide(number1 as nullable number, number2 as nullable number, optional precision
as nullable number) as nullable number
```

About

Returns the integer portion of the result from dividing a number, `number1`, by another number, `number2`. If `number1` or `number2` are null, `Number.IntegerDivide` returns null.

- `number1`: The dividend.
- `number2`: The divisor.

Example 1

Divide 6 by 4.

```
Number.IntegerDivide(6, 4)
```

```
1
```

Example 2

Divide 8.3 by 3.

```
Number.IntegerDivide(8.3, 3)
```

```
2
```

Number.IsEven

3/15/2021 • 2 minutes to read

Syntax

```
Number.IsEven(number as number) as logical
```

About

Indicates if the value, `number`, is even by returning `true` if it is even, `false` otherwise.

Example 1

Check if 625 is an even number.

```
Number.IsEven(625)
```

`false`

Example 2

Check if 82 is an even number.

```
Number.IsEven(82)
```

`true`

Number.IsNaN

3/15/2021 • 2 minutes to read

Syntax

```
Number.IsNaN(number as number) as logical
```

About

Indicates if the value is NaN (Not a number). Returns `true` if `number` is equivalent to `Number.IsNaN`, `false` otherwise.

Example 1

Check if 0 divided by 0 is NaN.

```
Number.IsNaN(0/0)
```

`true`

Example 2

Check if 1 divided by 0 is NaN.

```
Number.IsNaN(1/0)
```

`false`

Number.IsOdd

3/15/2021 • 2 minutes to read

Syntax

```
Number.IsOdd(number as number) as logical
```

About

Indicates if the value is odd. Returns `true` if `number` is an odd number, `false` otherwise.

Example 1

Check if 625 is an odd number.

```
Number.IsOdd(625)
```

`true`

Example 2

Check if 82 is an odd number.

```
Number.IsOdd(82)
```

`false`

Number.Ln

3/15/2021 • 2 minutes to read

Syntax

```
Number.Ln(number as nullable number) as nullable number
```

About

Returns the natural logarithm of a number, `number`. If `number` is null `Number.Ln` returns null.

####Example 1 Get the natural logarithm of 15.

```
Number.Ln(15)
```

```
2.70805020110221
```

Number.Log

3/15/2021 • 2 minutes to read

Syntax

```
Number.Log(number as nullable number, optional base as nullable number) as nullable number
```

About

Returns the logarithm of a number, `number`, to the specified `base` base. If `base` is not specified, the default value is Number.E. If `number` is null `Number.Log` returns null.

Example 1

Get the base 10 logarithm of 2.

```
Number.Log(2, 10)
```

```
0.3010299956639812
```

Example 2

Get the base e logarithm of 2.

```
Number.Log(2)
```

```
0.69314718055994529
```

Number.Log10

3/15/2021 • 2 minutes to read

Syntax

```
Number.Log10(number as nullable number) as nullable number
```

About

Returns the base 10 logarithm of a number, `number`. If `number` is null `Number.Log10` returns null.

Example 1

Get the base 10 logarithm of 2.

```
Number.Log10(2)
```

```
0.3010299956639812
```

Number.Mod

3/15/2021 • 2 minutes to read

Syntax

```
Number.Mod(number as nullable number, divisor as nullable number, optional precision as nullable number) as nullable number
```

About

Returns the remainder resulting from the integer division of `number` by `divisor`. If `number` or `divisor` are null, `Number.Mod` returns null.

- `number`: The dividend.
- `divisor`: The divisor.

Example 1

Find the remainder when you divide 5 by 3.

```
Number.Mod(5, 3)
```

`2`

Number.NaN

3/15/2021 • 2 minutes to read

About

A constant value that represents 0 divided by 0.

Number.NegativeInfinity

3/15/2021 • 2 minutes to read

About

A constant value that represents -1 divided by 0.

Number.Permutations

3/15/2021 • 2 minutes to read

Syntax

```
Number.Permutations(setSize as nullable number, permutationSize as nullable number) as nullable number
```

About

Returns the number of permutations that can be generated from a number of items, `setSize`, with a specified permutation size, `permutationSize`.

Example 1

Find the number of permutations from a total of 5 items in groups of 3.

```
Number.Permutations(5, 3)
```

`60`

Number.PI

3/15/2021 • 2 minutes to read

About

A constant that represents 3.1415926535897932, the value for pi up to 16 decimal digits.

Number.PositiveInfinity

3/15/2021 • 2 minutes to read

About

A constant value that represents 1 divided by 0.

Number.Power

3/15/2021 • 2 minutes to read

Syntax

```
Number.Power(number as nullable number, power as nullable number) as nullable number
```

About

Returns the result of raising `number` to the power of `power`. If `number` or `power` are null, `Number.Power` returns null.

- `number`: The base.
- `power`: The exponent.

Example 1

Find the value of 5 raised to the power of 3 (5 cubed).

```
Number.Power(5, 3)
```

```
125
```

Number.Random

3/15/2021 • 2 minutes to read

Syntax

```
Number.Random() as number
```

About

Returns a random number between 0 and 1.

Example 1

Get a random number.

```
Number.Random()
```

```
0.919303
```

Number.RandomBetween

3/15/2021 • 2 minutes to read

Syntax

```
Number.RandomBetween(bottom as number, top as number) as number
```

About

Returns a random number between `bottom` and `top`.

Example 1

Get a random number between 1 and 5.

```
Number.RandomBetween(1, 5)
```

`2.546797`

Number.Round

6/22/2021 • 2 minutes to read

Syntax

```
Number.Round(number as nullable number, optional digits as nullable number, optional
roundingMode as nullable number) as nullable number
```

About

Returns the result of rounding `number` to the nearest number. If `number` is null, `Number.Round` returns null.

By default, `number` is rounded to the nearest integer, and ties are broken by rounding to the nearest even number (using `RoundingMode.ToEven`, also known as "banker's rounding").

However, these defaults can be overridden via the following optional parameters.

- `digits`: Causes `number` to be rounded to the specified number of decimal digits.
- `roundingMode`: Overrides the default tie-breaking behavior when `number` is at the midpoint between two potential rounded values (see `RoundingMode.Type` for possible values).

Example 1

Round 1.234 to the nearest integer.

```
Number.Round(1.234)
```

`1`

Example 2

Round 1.56 to the nearest integer.

```
Number.Round(1.56)
```

`2`

Example 3

Round 1.2345 to two decimal places.

```
Number.Round(1.2345, 2)
```

`1.23`

Example 4

Round 1.2345 to three decimal places (Rounding up).

```
Number.Round(1.2345, 3, RoundingMode.Up)
```

```
1.235
```

Example 5

Round 1.2345 to three decimal places (Rounding down).

```
Number.Round(1.2345, 3, RoundingMode.Down)
```

```
1.234
```

Number.RoundAwayFromZero

3/15/2021 • 2 minutes to read

Syntax

```
Number.RoundAwayFromZero(number as nullable number, optional digits as nullable number) as
nullable number
```

About

Returns the result of rounding `number` based on the sign of the number. This function will round positive numbers up and negative numbers down. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Example 1

Round the number -1.2 away from zero.

```
Number.RoundAwayFromZero(-1.2)
```

`-2`

Example 2

Round the number 1.2 away from zero.

```
Number.RoundAwayFromZero(1.2)
```

`2`

Example 3

Round the number -1.234 to two decimal places away from zero.

```
Number.RoundAwayFromZero(-1.234, 2)
```

`-1.24`

Number.RoundDown

3/15/2021 • 2 minutes to read

Syntax

```
Number.RoundDown(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` down to the previous highest integer. If `number` is null, `Number.RoundDown` returns null. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Example 1

Round down 1.234 to integer.

```
Number.RoundDown(1.234)
```

```
1
```

Example 2

Round down 1.999 to integer.

```
Number.RoundDown(1.999)
```

```
1
```

Example 3

Round down 1.999 to two decimal places.

```
Number.RoundDown(1.999, 2)
```

```
1.99
```

Number.RoundTowardZero

3/15/2021 • 2 minutes to read

Syntax

```
Number.RoundTowardZero(number as nullable number, optional digits as nullable number) as
nullable number
```

About

Returns the result of rounding `number` based on the sign of the number. This function will round positive numbers down and negative numbers up. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Number.RoundUp

3/15/2021 • 2 minutes to read

Syntax

```
Number.RoundUp(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` down to the previous highest integer. If `number` is null, `Number.RoundDown` returns null. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Example 1

Round up 1.234 to integer.

```
Number.RoundUp(1.234)
```

```
2
```

Example 2

Round up 1.999 to integer.

```
Number.RoundUp(1.999)
```

```
2
```

Example 3

Round up 1.234 to two decimal places.

```
Number.RoundUp(1.234, 2)
```

```
1.24
```

Number.Sign

3/15/2021 • 2 minutes to read

Syntax

```
Number.Sign(number as nullable number) as nullable number
```

About

Returns 1 for if `number` is a positive number, -1 if it is a negative number, and 0 if it is zero. If `number` is null, `Number.Sign` returns null.

Example 1

Determine the sign of 182.

```
Number.Sign(182)
```

`1`

Example 2

Determine the sign of -182.

```
Number.Sign(-182)
```

`-1`

Example 3

Determine the sign of 0.

```
Number.Sign(0)
```

`0`

Number.Sin

3/15/2021 • 2 minutes to read

Syntax

```
Number.Sin(number as nullable number) as nullable number
```

About

Returns the sine of `number`.

Example 1

Find the sine of the angle 0.

```
Number.Sin(0)
```

`0`

Number.Sinh

3/15/2021 • 2 minutes to read

Syntax

```
Number.Sinh(number as nullable number) as nullable number
```

About

Returns the hyperbolic sine of `number`.

Number.Sqrt

3/15/2021 • 2 minutes to read

Syntax

```
Number.Sqrt(number as nullable number) as nullable number
```

About

Returns the square root of `number`. If `number` is null, `Number.Sqrt` returns null. If it is a negative value, `Number.NaN` is returned (Not a number).

Example 1

Find the square root of 625.

```
Number.Sqrt(625)
```

```
25
```

Example 2

Find the square root of 85.

```
Number.Sqrt(85)
```

```
9.2195444572928871
```

Number.Tan

3/15/2021 • 2 minutes to read

Syntax

```
Number.Tan(number as nullable number) as nullable number
```

About

Returns the tangent of `number`.

Example 1

Find the tangent of the angle 1.

```
Number.Tan(1)
```

```
1.5574077246549023
```

Number.Tanh

3/15/2021 • 2 minutes to read

Syntax

```
Number.Tanh(number as nullable number) as nullable number
```

About

Returns the hyperbolic tangent of `number`.

Number.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Number.ToText(number as nullable number, optional format as nullable text, optional culture as
nullable text) as nullable text
```

About

Formats the numeric value `number` to a text value according to the format specified by `format`. The format is a single character code optionally followed by a number precision specifier. The following character codes may be used for `format`.

- "D" or "d": (Decimal) Formats the result as integer digits. The precision specifier controls the number of digits in the output.
- "E" or "e": (Exponential/scientific) Exponential notation. The precision specifier controls the maximum number of decimal digits (default is 6).
- "F" or "f": (Fixed-point) Integral and decimal digits.
- "G" or "g": (General) Most compact form of either fixed-point or scientific.
- "N" or "n": (Number) Integral and decimal digits with group separators and a decimal separator.
- "P" or "p": (Percent) Number multiplied by 100 and displayed with a percent symbol.
- "R" or "r": (Round-trip) A text value that can round-trip an identical number. The precision specifier is ignored.
- "X" or "x": (Hexadecimal) A hexadecimal text value.

An optional `culture` may also be provided (for example, "en-US").

Example 1

Format a number as text without format specified.

```
Number.ToText(4)
```

`"4"`

Example 2

Format a number as text in Exponential format.

```
Number.ToText(4, "e")
```

`"4.000000e+000"`

Example 3

Format a number as text in Decimal format with limited precision.

```
Number.ToText(-0.1234, "P1")
```

```
"-12.3 %"
```

Percentage.From

3/15/2021 • 2 minutes to read

Syntax

```
Percentage.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a `percentage` value from the given `value`. If the given `value` is `null`, `Percentage.From` returns `null`. If the given `value` is `text` with a trailing percent symbol, then the converted decimal number will be returned. Otherwise, see `Number.From` for converting it to `number` value. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the `percentage` value of `"12.3%"`.

```
Percentage.From("12.3%")
```

`0.123`

RoundingMode.AwayFromZero

3/15/2021 • 2 minutes to read

About

RoundingMode.AwayFromZero

RoundingMode.Down

3/15/2021 • 2 minutes to read

About

RoundingMode.Down

RoundingMode.ToEven

3/15/2021 • 2 minutes to read

About

RoundingMode.ToEven

RoundingMode.TowardZero

3/15/2021 • 2 minutes to read

About

RoundingMode.TowardZero

RoundingMode.Up

3/15/2021 • 2 minutes to read

About

RoundingMode.Up

Single.From

3/15/2021 • 2 minutes to read

Syntax

```
Single.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Single `number` value from the given `value`. If the given `value` is `null`, `Single.From` returns `null`. If the given `value` is `number` within the range of Single, `value` is returned, otherwise an error is returned. If the given `value` is of any other type, see `Number.FromText` for converting it to `number` value, then the previous statement about converting `number` value to Single `number` value applies. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the Single `number` value of `"1.5"`.

```
Single.From("1.5")
```

`1.5`

Record functions

3/15/2021 • 2 minutes to read

These functions create and manipulate record values.

Record

Information

FUNCTION	DESCRIPTION
Record.FieldCount	Returns the number of fields in a record.
Record.HasFields	Returns true if the field name or field names are present in a record.

Transformations

FUNCTION	DESCRIPTION
Geography.FromWellKnownText	Translates text representing a geographic value in Well-Known Text (WKT) format into a structured record.
Geography.ToWellKnownText	Translates a structured geographic point value into its Well-Known Text (WKT) representation.
GeographyPoint.From	Creates a record representing a geographic point from parts.
Geometry.FromWellKnownText	Translates text representing a geometric value in Well-Known Text (WKT) format into a structured record.
Geometry.ToWellKnownText	Translates a structured geometric point value into its Well-Known Text (WKT) representation.
GeometryPoint.From	Creates a record representing a geometric point from parts.
Record.AddField	Adds a field from a field name and value.
Record.Combine	Combines the records in a list.
Record.RemoveFields	Returns a new record that reorders the given fields with respect to each other. Any fields not specified remain in their original locations.
Record.RenameFields	Returns a new record that renames the fields specified. The resultant fields will retain their original order. This function supports swapping and chaining field names. However, all target names plus remaining field names must constitute a unique set or an error will occur.

FUNCTION	DESCRIPTION
Record.ReorderFields	Returns a new record that reorders fields relative to each other. Any fields not specified remain in their original locations. Requires two or more fields.
Record.TransformFields	Transforms fields by applying transformOperations. For more more information about values supported by transformOperations, see Parameter Values.

Selection

FUNCTION	DESCRIPTION
Record.Field	Returns the value of the given field. This function can be used to dynamically create field lookup syntax for a given record. In that way it is a dynamic verison of the record[field] syntax.
Record.FieldNames	Returns a list of field names in order of the record's fields.
Record.FieldOrDefault	Returns the value of a field from a record, or the default value if the field does not exist.
Record.FieldValues	Returns a list of field values in order of the record's fields.
Record.SelectFields	Returns a new record that contains the fields selected from the input record. The original order of the fields is maintained.

Serialization

FUNCTION	DESCRIPTION
Record.FromList	Returns a record given a list of field values and a set of fields.
Record.FromTable	Returns a record from a table of records containing field names and values.
Record.ToList	Returns a list of values containing the field values of the input record.
Record.ToTable	Returns a table of records containing field names and values from an input record.

Parameter Values

The following type definitions are used to describe the parameter values that are referenced in Record functions above.

TYPE DEFINITION	DESCRIPTION
MissingField option	MissingField.Error = 0; MissingField.Ignore = 1; MissingField.UseNull = 2;

TYPE DEFINITION	DESCRIPTION
Transform operations	Transform operations can be specified by either of the following values: A list value of two items, first item being the field name and the second item being the transformation function applied to that field to produce a new value. A list of transformations can be provided by providing a list value, and each item being the list value of 2 items as described above. For examples, see description of Record.TransformFields
Rename operations	Rename operations for a record can be specified as either of: A single rename operation, which is represented by a list of two field names, old and new. For examples, see description of Record.RenameFields.

Geography.FromWellKnownText

3/15/2021 • 2 minutes to read

Syntax

```
Geography.FromWellKnownText(input as nullable text) as nullable record
```

About

Translates text representing a geographic value in Well-Known Text (WKT) format into a structured record. WKT is a standard format defined by the Open Geospatial Consortium (OGC) and is the typical serialization format used by databases including SQL Server.

Geography.ToWellKnownText

3/15/2021 • 2 minutes to read

Syntax

```
Geography.ToWellKnownText(input as nullable record, optional omitSRID as nullable logical) as
nullable text
```

About

Translates a structured geographic point value into its Well-Known Text (WKT) representation as defined by the Open Geospatial Consortium (OGC), also the serialization format used by many databases including SQL Server.

GeographyPoint.From

3/15/2021 • 2 minutes to read

Syntax

```
GeographyPoint.From(longitude as number, latitude as number, optional z as nullable number,
optional m as nullable number, optional srid as nullable number) as record
```

About

Creates a record representing a geographic point from its constituent parts, such as longitude, latitude, and if present, elevation (Z) and measure (M). An optional spatial reference identifier (SRID) can be given if different from the default value (4326).

Geometry.FromWellKnownText

3/15/2021 • 2 minutes to read

Syntax

```
Geometry.FromWellKnownText(input as nullable text) as nullable record
```

About

Translates text representing a geometric value in Well-Known Text (WKT) format into a structured record. WKT is a standard format defined by the Open Geospatial Consortium (OGC) and is the typical serialization format used by databases including SQL Server.

Geometry.ToWellKnownText

3/15/2021 • 2 minutes to read

Syntax

```
Geometry.ToWellKnownText(input as nullable record, optional omitSRID as nullable logical) as
nullable text
```

About

Translates a structured geometric point value into its Well-Known Text (WKT) representation as defined by the Open Geospatial Consortium (OGC), also the serialization format used by many databases including SQL Server.

GeometryPoint.From

3/15/2021 • 2 minutes to read

Syntax

```
GeometryPoint.From(x as number, y as number, optional z as nullable number, optional m as
nullable number, optional srid as nullable number) as record
```

About

Creates a record representing a geometric point from its constituent parts, such as X coordinate, Y coordinate, and if present, Z coordinate and measure (M). An optional spatial reference identifier (SRID) can be given if different from the default value (0).

MissingField.Error

3/15/2021 • 2 minutes to read

About

An optional parameter in record and table functions indicating that missing fields should result in an error. (This is the default parameter value.)

MissingField.Ignore

3/15/2021 • 2 minutes to read

About

An optional parameter in record and table functions indicating that missing fields should be ignored.

MissingField.UseNull

3/15/2021 • 2 minutes to read

About

An optional parameter in record and table functions indicating that missing fields should be included as null values.

Record.AddField

3/15/2021 • 2 minutes to read

Syntax

```
Record.AddField(record as record, fieldName as text, value as any, optional delayed as nullable
logical) as record
```

About

Adds a field to a record `record`, given the name of the field `fieldName` and the value `value`.

Example 1

Add the field Address to the record.

```
Record.AddField([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "Address", "123 Main St.")
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567
ADDRESS	123 Main St.

Record.Combine

3/15/2021 • 2 minutes to read

Syntax

```
Record.Combine(records as list) as record
```

About

Combines the records in the given `records`. If the `records` contains non-record values, an error is returned.

Example 1

Create a combined record from the records.

```
Record.Combine({
    [CustomerID = 1, Name = "Bob"],
    [Phone = "123-4567"]
})
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Record.Field

3/15/2021 • 2 minutes to read

Syntax

```
Record.Field(record as record, field as text) as any
```

About

Returns the value of the specified `field` in the `record`. If the field is not found, an exception is thrown.

Example 1

Find the value of field "CustomerID" in the record.

```
Record.Field([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

```
1
```

Record.FieldCount

3/15/2021 • 2 minutes to read

Syntax

```
Record.FieldCount(record as record) as number
```

About

Returns the number of fields in the record `record`.

Example 1

Find the number of fields in the record.

```
Record.FieldCount([CustomerID = 1, Name = "Bob"])
```

`2`

Record.FieldNames

3/15/2021 • 2 minutes to read

Syntax

```
Record.FieldNames(record as record) as list
```

About

Returns the names of the fields in the record `record` as text.

Example 1

Find the names of the fields in the record.

```
Record.FieldNames([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

OrderID
CustomerID
Item
Price

Record.FieldOrDefault

3/15/2021 • 2 minutes to read

Syntax

```
Record.FieldOrDefault(record as nullable record, field as text, optional defaultValue as any) as any
```

About

Returns the value of the specified field `field` in the record `record`. If the field is not found, the optional `defaultValue` is returned.

Example 1

Find the value of field "Phone" in the record, or return null if it doesn't exist.

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone")
```

`null`

Example 2

Find the value of field "Phone" in the record, or return the default if it doesn't exist.

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone", "123-4567")
```

`"123-4567"`

Record.FieldValues

3/15/2021 • 2 minutes to read

Syntax

```
Record.FieldValues(record as record) as list
```

About

Returns a list of the field values in record `record`.

Example 1

Find the field values in the record.

```
Record.FieldValues([CustomerID = 1, Name = "Bob", Phone = "123-4567"])
```

1
Bob
123-4567

Record.FromList

3/15/2021 • 2 minutes to read

Syntax

```
Record.FromList(list as list, fields as any) as record
```

About

Returns a record given a `list` of field values and a set of fields. The `fields` can be specified either by a list of text values, or a record type. An error is thrown if the fields are not unique.

Example 1

Build a record from a list of field values and a list of field names.

```
Record.FromList({1, "Bob", "123-4567"}, {"CustomerID", "Name", "Phone"})
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Example 2

Build a record from a list of field values and a record type.

```
Record.FromList({1, "Bob", "123-4567"}, type [CustomerID = number, Name = text, Phone = number])
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Record.FromTable

3/15/2021 • 2 minutes to read

Syntax

```
Record.FromTable(table as table) as record
```

About

Returns a record from a table of records `table` containing field names and value names `{[Name = name, Value = value]}`. An exception is thrown if the field names are not unique.

Example 1

Create a record from the table of the form Table.FromRecords({[Name = "CustomerID", Value = 1], [Name = "Name", Value = "Bob"], [Name = "Phone", Value = "123-4567"]}).

```
Record.FromTable(
    Table.FromRecords({
        [Name = "CustomerID", Value = 1],
        [Name = "Name", Value = "Bob"],
        [Name = "Phone", Value = "123-4567"]
    })
)
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Record.HasFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.HasFields(record as record, fields as any) as logical
```

About

Indicates whether the record `record` has the fields specified in `fields`, by returning a logical value (true or false). Multiple field values can be specified using a list.

Example 1

Check if the record has the field "CustomerID".

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

`true`

Example 2

Check if the record has the field "CustomerID" and "Address".

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"], {"CustomerID", "Address"})
```

`false`

Record.RemoveFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.RemoveFields(record as record, fields as any, optional missingField as nullable number)
as record
```

About

Returns a record that removes all the fields specified in list `fields` from the input `record`. If the field specified does not exist, an exception is thrown.

Example 1

Remove the field "Price" from the record.

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00], "Price")
```

CUSTOMERID	1
ITEM	Fishing rod

Example 2

Remove the fields "Price" and "Item" from the record.

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00], {"Price", "Item"})
```

CUSTOMERID	1

Record.RenameFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.RenameFields(record as record, renames as list, optional missingField as nullable number)
as record
```

About

Returns a record after renaming fields in the input `record` to the new field names specified in list `renames`. For multiple renames, a nested list can be used ({ {old1, new1}, {old2, new2} }.

Example 1

Rename the field "UnitPrice" to "Price" from the record.

```
Record.RenameFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],
    {"UnitPrice", "Price"}
)
```

ORDERID	1
CUSTOMERID	1
ITEM	Fishing rod
PRICE	100

Example 2

Rename the fields "UnitPrice" to "Price" and "OrderNum" to "OrderID" from the record.

```
Record.RenameFields(
    [OrderNum = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],
    {
        {"UnitPrice", "Price"},
        {"OrderNum", "OrderID"}
    }
)
```

ORDERID	1
CUSTOMERID	1
ITEM	Fishing rod

PRICE	100

Record.ReorderFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.ReorderFields(record as record, fieldOrder as list, optional missingField as nullable
number) as record
```

About

Returns a record after reordering the fields in `record` in the order of fields specified in list `fieldOrder`. Field values are maintained and fields not listed in `fieldOrder` are left in their original position.

Example 1

Reorder some of the fields in the record.

```
Record.ReorderFields(
    [CustomerID = 1, OrderID = 1, Item = "Fishing rod", Price = 100.0],
    {"OrderID", "CustomerID"}
)
```

ORDERID	1
CUSTOMERID	1
ITEM	Fishing rod
PRICE	100

Record.SelectFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.SelectFields(record as record, fields as any, optional missingField as nullable number)
as record
```

About

Returns a record which includes only the fields specified in list `fields` from the input `record`.

Example 1

Select the fields "Item" and "Price" in the record.

```
Record.SelectFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
    {"Item", "Price"}
)
```

ITEM	Fishing rod
PRICE	100

Record.ToList

3/15/2021 • 2 minutes to read

Syntax

```
Record.ToList(record as record) as list
```

About

Returns a list of values containing the field values from the input `record`.

Example 1

Extract the field values from a record.

```
Record.ToList([A = 1, B = 2, C = 3])
```

1
2
3

Record.ToTable

3/15/2021 • 2 minutes to read

Syntax

```
Record.ToTable(record as record) as table
```

About

Returns a table containing the columns `Name` and `Value` with a row for each field in `record`.

Example 1

Return the table from the record.

```
Record.ToTable([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

NAME	VALUE
OrderID	1
CustomerID	1
Item	Fishing rod
Price	100

Record.TransformFields

3/15/2021 • 2 minutes to read

Syntax

```
Record.TransformFields(record as record, transformOperations as list, optional missingField as
nullable number) as record
```

About

Returns a record after applying transformations specified in list `transformOperations` to `record`. One or more fields may be transformed at a given time.

In the case of a single field being transformed, `transformOperations` is expected to be a list with two items. The first item in `transformOperations` specifies a field name, and the second item in `transformOperations` specifies the function to be used for transformation. For example, `{"Quantity", Number.FromText}`
In the case of a multiple fields being transformed, `transformOperations` is expected to be a list of lists, where each inner list is a pair of field name and transformation operation. For example,
`{{"Quantity",Number.FromText},{"UnitPrice", Number.FromText}}`

Example 1

Convert "Price" field to number.

```
Record.TransformFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = "100.0"],
    {"Price", Number.FromText}
)
```

ORDERID	1
CUSTOMERID	1
ITEM	Fishing rod
PRICE	100

Example 2

Convert "OrderID" and "Price" fields to numbers.

```
Record.TransformFields(
    [OrderID = "1", CustomerID = 1, Item = "Fishing rod", Price = "100.0"],
    {{"OrderID", Number.FromText}, {"Price", Number.FromText}}
)
```

ORDERID	1

CUSTOMERID	1
ITEM	Fishing rod
PRICE	100

Replacer functions

3/15/2021 • 2 minutes to read

These functions are used by other functions in the library to replace a given value.

Replacer

FUNCTION	DESCRIPTION
Replacer.ReplaceText	This function be provided to List.ReplaceValue or Table.ReplaceValue to do replace of text values in list and table values respectively.
Replacer.ReplaceValue	This function be provided to List.ReplaceValue or Table.ReplaceValue to do replace values in list and table values respectively.

Replacer.ReplaceText

3/15/2021 • 2 minutes to read

Syntax

```
Replacer.ReplaceText(text as nullable text, old as text, new as text) as nullable text
```

About

Replaces the `old` text in the original `text` with the `new` text. This replacer function can be used in `List.ReplaceValue` and `Table.ReplaceValue`.

Example 1

Replace the text "hE" with "He" in the string "hEllo world".

```
Replacer.ReplaceText("hEllo world", "hE", "He")
```

`"Hello world"`

Replacer.ReplaceValue

3/15/2021 • 2 minutes to read

Syntax

```
Replacer.ReplaceValue(value as any, old as any, new as any) as any
```

About

Replaces the `old` value in the original `value` with the `new` value. This replacer function can be used in `List.ReplaceValue` and `Table.ReplaceValue`.

Example 1

Replace the value 11 with the value 10.

```
Replacer.ReplaceValue(11, 11, 10)
```

```
10
```

Splitter functions

3/15/2021 • 2 minutes to read

These functions split text.

Splitter

FUNCTION	DESCRIPTION
Splitter.SplitByNothing	Returns a function that does no splitting, returning its argument as a single element list.
Splitter.SplitTextByCharacterTransition	Returns a function that splits text into a list of text according to a transition from one kind of character to another.
Splitter.SplitTextByAnyDelimiter	Returns a function that splits text by any supported delimiter.
Splitter.SplitTextByDelimiter	Returns a function that will split text according to a delimiter.
Splitter.SplitTextByEachDelimiter	Returns a function that splits text by each delimiter in turn.
Splitter.SplitTextByLengths	Returns a function that splits text according to the specified lengths.
Splitter.SplitTextByPositions	Returns a function that splits text according to the specified positions.
Splitter.SplitTextByRanges	Returns a function that splits text according to the specified ranges.
Splitter.SplitTextByRepeatedLengths	Returns a function that splits text into a list of text after the specified length repeatedly.
Splitter.SplitTextByWhitespace	Returns a function that splits text according to whitespace.

PARAMETER VALUES	DESCRIPTION
QuoteStyle.Csv	Quote characters indicate the start of a quoted string. Nested quotes are indicated by two quote characters.
QuoteStyle.None	Quote characters have no significance.

QuoteStyle.Csv

3/15/2021 • 2 minutes to read

About

Quote characters indicate the start of a quoted string. Nested quotes are indicated by two quote characters.

QuoteStyle.None

3/15/2021 • 2 minutes to read

About

Quote characters have no significance.

Splitter.SplitByNothing

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitByNothing() as function
```

About

Returns a function that does no splitting, returning its argument as a single element list.

Splitter.SplitTextByAnyDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByAnyDelimiter(delimiters as list, optional quoteStyle as nullable number,
optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text at any of the specified delimiters.

Splitter.SplitTextByCharacterTransition

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByCharacterTransition(before as anynonnull, after as anynonnull) as function
```

About

Returns a function that splits text into a list of text according to a transition from one kind of character to another. The `before` and `after` parameters can either be a list of characters, or a function that takes a character and returns true/false.

Splitter.SplitTextByDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByDelimiter(delimiter as text, optional quoteStyle as nullable number) as
function
```

About

Returns a function that splits text into a list of text according to the specified delimiter.

Splitter.SplitTextByEachDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number,
optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text at each specified delimiter in sequence.

Splitter.SplitTextByLengths

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByLengths(lengths as list, optional startAtEnd as nullable logical) as
function
```

About

Returns a function that splits text into a list of text by each specified length.

Splitter.SplitTextByPositions

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByPositions(positions as list, optional startAtEnd as nullable logical) as
function
```

About

Returns a function that splits text into a list of text at each specified position.

Splitter.SplitTextByRanges

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByRanges(ranges as list, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text according to the specified offsets and lengths.

Splitter.SplitTextByRepeatedLengths

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByRepeatedLengths(length as number, optional startAtEnd as nullable logical)
as function
```

About

Returns a function that splits text into a list of text after the specified length repeatedly.

Splitter.SplitTextByWhitespace

3/15/2021 • 2 minutes to read

Syntax

```
Splitter.SplitTextByWhitespace(optional quoteStyle as nullable number) as function
```

About

Returns a function that splits text into a list of text at whitespace.

Table functions

6/22/2021 • 14 minutes to read

These functions create and manipulate table values.

Table construction

FUNCTION	DESCRIPTION
ItemExpression.From	Returns the AST for the body of a function.
ItemExpression.Item	An AST node representing the item in an item expression.
RowExpression.Column	Returns an AST that represents access to a column within a row expression.
RowExpression.From	Returns the AST for the body of a function.
RowExpression.Row	An AST node representing the row in a row expression.
Table.FromColumns	Returns a table from a list containing nested lists with the column names and values.
Table.FromList	Converts a list into a table by applying the specified splitting function to each item in the list.
Table.FromRecords	Returns a table from a list of records.
Table.FromRows	Creates a table from the list where each element of the list is a list that contains the column values for a single row.
Table.FromValue	Returns a table with a column containing the provided value or list of values.
Table.FuzzyGroup	Groups the rows of a table by fuzzily matching values in the specified column for each row.
Table.FuzzyJoin	Joins the rows from the two tables that fuzzy match based on the given keys.
Table.FuzzyNestedJoin	Performs a fuzzy join between tables on supplied columns and produces the join result in a new column.
Table.Split	Splits the specified table into a list of tables using the specified page size.
Table.View	Creates or extends a table with user-defined handlers for query and action operations.
Table.ViewFunction	Creates a function that can be intercepted by a handler defined on a view (via `Table.View`).

FUNCTION	DESCRIPTION

Conversions

FUNCTION	DESCRIPTION
Table.ToColumns	Returns a list of nested lists each representing a column of values in the input table.
Table.ToList	Returns a table into a list by applying the specified combining function to each row of values in a table.
Table.ToRecords	Returns a list of records from an input table.
Table.ToRows	Returns a nested list of row values from an input table.

Information

FUNCTION	DESCRIPTION
Table.ApproximateRowCount	Returns the approximate number of rows in the table.
Table.ColumnCount	Returns the number of columns in a table.
Table.IsEmpty	Returns true if the table does not contain any rows.
Table.Profile	Returns a profile of the columns of a table.
Table.RowCount	Returns the number of rows in a table.
Table.Schema	Returns a table containing a description of the columns (i.e. the schema) of the specified table.
Tables.GetRelationships	Returns the relationships among a set of tables.

Row operations

FUNCTION	DESCRIPTION
Table.AlternateRows	Returns a table containing an alternating pattern of the rows from a table.
Table.Combine	Returns a table that is the result of merging a list of tables. The tables must all have the same row type structure.
Table.FindText	Returns a table containing only the rows that have the specified text within one of their cells or any part thereof.

FUNCTION	DESCRIPTION
Table.First	Returns the first row from a table.
Table.FirstN	Returns the first row(s) of a table, depending on the countOrCondition parameter.
Table.FirstValue	Returns the first column of the first row of the table or a specified default value.
Table.FromPartitions	Returns a table that is the result of combining a set of partitioned tables into new columns. The type of the column can optionally be specified, the default is any.
Table.InsertRows	Returns a table with the list of rows inserted into the table at an index. Each row to insert must match the row type of the table..
Table.Last	Returns the last row of a table.
Table.LastN	Returns the last row(s) from a table, depending on the countOrCondition parameter.
Table.MatchesAllRows	Returns true if all of the rows in a table meet a condition.
Table.MatchesAnyRows	Returns true if any of the rows in a table meet a condition.
Table.Partition	Partitions the table into a list of groups number of tables, based on the value of the column of each row and a hash function. The hash function is applied to the value of the column of a row to obtain a hash value for the row. The hash value modulo groups determines in which of the returned tables the row will be placed.
Table.Range	Returns the specified number of rows from a table starting at an offset.
Table.RemoveFirstN	Returns a table with the specified number of rows removed from the table starting at the first row. The number of rows removed depends on the optional countOrCondition parameter.
Table.RemoveLastN	Returns a table with the specified number of rows removed from the table starting at the last row. The number of rows removed depends on the optional countOrCondition parameter.
Table.RemoveRows	Returns a table with the specified number of rows removed from the table starting at an offset.
Table.RemoveRowsWithErrors	Returns a table with all rows removed from the table that contain an error in at least one of the cells in a row.
Table.Repeat	Returns a table containing the rows of the table repeated the count number of times.

FUNCTION	DESCRIPTION
Table.ReplaceRows	Returns a table where the rows beginning at an offset and continuing for count are replaced with the provided rows.
Table.ReverseRows	Returns a table with the rows in reverse order.
Table.SelectRows	Returns a table containing only the rows that match a condition.
Table.SelectRowsWithErrors	Returns a table with only the rows from table that contain an error in at least one of the cells in a row.
Table.SingleRow	Returns a single row from a table.
Table.Skip	Returns a table that does not contain the first row or rows of the table.
Table.SplitAt	Returns a list containing the first count rows specified and the remaining rows.

Column operations

FUNCTION	DESCRIPTION
Table.Column	Returns the values from a column in a table.
Table.ColumnNames	Returns the names of columns from a table.
Table.ColumnsOfType	Returns a list with the names of the columns that match the specified types.
Table.DemoteHeaders	Demotes the header row down into the first row of a table.
Table.DuplicateColumn	Duplicates a column with the specified name. Values and type are copied from the source column.
Table.HasColumns	Returns true if a table has the specified column or columns.
Table.Pivot	Given a table and attribute column containing pivotValues, creates new columns for each of the pivot values and assigns them values from the valueColumn. An optional aggregationFunction can be provided to handle multiple occurrence of the same key value in the attribute column.
Table.PrefixColumns	Returns a table where the columns have all been prefixed with a text value.
Table.PromoteHeaders	Promotes the first row of the table into its header or column names.
Table.RemoveColumns	Returns a table without a specific column or columns.

FUNCTION	DESCRIPTION
Table.ReorderColumns	Returns a table with specific columns in an order relative to one another.
Table.RenameColumns	Returns a table with the columns renamed as specified.
Table.SelectColumns	Returns a table that contains only specific columns.
Table.TransformColumnNames	Transforms column names by using the given function.
Table.Unpivot	Given a list of table columns, transforms those columns into attribute-value pairs.
Table.UnpivotOtherColumns	Translates all columns other than a specified set into attribute-value pairs, combined with the rest of the values in each row.

Parameters

PARAMETER VALUES	DESCRIPTION
JoinKind.Inner	A possible value for the optional `JoinKind` parameter in `Table.Join`. The table resulting from an inner join contains a row for each pair of rows from the specified tables that were determined to match based on the specified key columns.
JoinKind.LeftOuter	A possible value for the optional `JoinKind` parameter in `Table.Join`. A left outer join ensures that all rows of the first table appear in the result.
JoinKind.RightOuter	A possible value for the optional `JoinKind` parameter in `Table.Join`. A right outer join ensures that all rows of the second table appear in the result.
JoinKind.FullOuter	A possible value for the optional `JoinKind` parameter in `Table.Join`. A full outer join ensures that all rows of both tables appear in the result. Rows that did not have a match in the other table are joined with a default row containing null values for all of its columns.
JoinKind.LeftAnti	A possible value for the optional `JoinKind` parameter in `Table.Join`. A left anti join returns that all rows from the first table which do not have a match in the second table.
JoinKind.RightAnti	A possible value for the optional `JoinKind` parameter in `Table.Join`. A right anti join returns that all rows from the second table which do not have a match in the first table.
MissingField.Error	An optional parameter in record and table functions indicating that missing fields should result in an error. (This is the default parameter value.)

PARAMETER VALUES	DESCRIPTION
MissingField.Ignore	An optional parameter in record and table functions indicating that missing fields should be ignored.
MissingField.UseNull	An optional parameter in record and table functions indicating that missing fields should be included as null values.
GroupKind.Global	GroupKind.Global
GroupKind.Local	GroupKind.Local
ExtraValues.List	If the splitter function returns more columns than the table expects, they should be collected into a list.
ExtraValues.Ignore	If the splitter function returns more columns than the table expects, they should be ignored.
ExtraValues.Error	If the splitter function returns more columns than the table expects, an error should be raised.
JoinAlgorithm.Dynamic	JoinAlgorithm.Dynamic
JoinAlgorithm.PairwiseHash	JoinAlgorithm.PairwiseHash
JoinAlgorithm.SortMerge	JoinAlgorithm.SortMerge
JoinAlgorithm.LeftHash	JoinAlgorithm.LeftHash
JoinAlgorithm.RightHash	JoinAlgorithm.RightHash
JoinAlgorithm.LeftIndex	JoinAlgorithm.LeftIndex
JoinAlgorithm.RightIndex	JoinAlgorithm.RightIndex
JoinSide.Left	Specifies the left table of a join.
JoinSide.Right	Specifies the right table of a join.

Transformation

Parameters for Group options

- GroupKind.Global = 0;
- GroupKind.Local = 1;

Parameters for Join kinds

- JoinKind.Inner = 0;
- JoinKind.LeftOuter = 1;
- JoinKind.RightOuter = 2;

- JoinKind.FullOuter = 3;
- JoinKind.LeftAnti = 4;
- JoinKind.RightAnti = 5

Join Algorithm

The following JoinAlgorithm values can be specified to Table.Join

```
JoinAlgorithm.Dynamic        0,
```

```
JoinAlgorithm.PairwiseHash   1,
```

```
JoinAlgorithm.SortMerge      2,
```

```
JoinAlgorithm.LeftHash       3,
```

```
JoinAlgorithm.RightHash      4,
```

```
JoinAlgorithm.LeftIndex      5,
```

```
JoinAlgorithm.RightIndex     6,
```

PARAMETER VALUES	DESCRIPTION
JoinSide.Left	Specifies the left table of a join.
JoinSide.Right	Specifies the right table of a join.

Example data

The following tables are used by the examples in this section.

Customers table

```
Customers = Table.FromRecords({
  [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
  [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
  [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
  [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
}
```

Orders table

```
Orders = Table.FromRecords({
  [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
  [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
  [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
  [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
  [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
  [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
  [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
  [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
  [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
})
```

FUNCTION	DESCRIPTION
Table.AddColumn	Adds a column named newColumnName to a table.
Table.AddFuzzyClusterColumn	Adds a new column with representative values obtained by fuzzy grouping values of the specified column in the table.
Table.AddIndexColumn	Returns a table with a new column with a specific name that, for each row, contains an index of the row in the table.
Table.AddJoinColumn	Performs a nested join between table1 and table2 from specific columns and produces the join result as a newColumnName column for each row of table1.
Table.AddKey	Add a key to table.
Table.AggregateTableColumn	Aggregates tables nested in a specific column into multiple columns containing aggregate values for those tables.
Table.CombineColumns	Table.CombineColumns merges columns using a combiner function to produce a new column. Table.CombineColumns is the inverse of Table.SplitColumns.
Table.CombineColumnsToRecord	Combines the specified columns into a new record-valued column where each record has field names and values corresponding to the column names and values of the columns that were combined.
Table.ConformToPageReader	This function is intended for internal use only.
Table.ExpandListColumn	Given a column of lists in a table, create a copy of a row for each value in its list.
Table.ExpandRecordColumn	Expands a column of records into columns with each of the values.

FUNCTION	DESCRIPTION
Table.ExpandTableColumn	Expands a column of records or a column of tables into multiple columns in the containing table.
Table.FillDown	Replaces null values in the specified column or columns of the table with the most recent non-null value in the column.
Table.FillUp	Returns a table from the table specified where the value of the next cell is propagated to the null values cells above in the column specified.
Table.FilterWithDataTable	
Table.Group	Groups table rows by the values of key columns for each row.
Table.Join	Joins the rows of table1 with the rows of table2 based on the equality of the values of the key columns selected by table1, key1 and table2, key2.
Table.Keys	Returns a list of key column names from a table.
Table.NestedJoin	Joins the rows of the tables based on the equality of the keys. The results are entered into a new column.
Table.ReplaceErrorValues	Replaces the error values in the specified columns with the corresponding specified value.
Table.ReplaceKeys	Returns a new table with new key information set in the keys argument.
Table.ReplaceRelationshipIdentity	
Table.ReplaceValue	Replaces oldValue with newValue in specific columns of a table, using the provided replacer function, such as text.Replace or Value.Replace.
Table.SplitColumn	Returns a new set of columns from a single column applying a splitter function to each value.
Table.TransformColumns	Transforms columns from a table using a function.
Table.TransformColumnTypes	Transforms the column types from a table using a type.
Table.TransformRows	Transforms the rows from a table using a transform function.
Table.Transpose	Returns a table with columns converted to rows and rows converted to columns from the input table.

Membership

Parameters for membership checks

Occurrence specification

```
Occurrence.First  = 0
```

```
Occurrence.Last   = 1
```

```
Occurrence.All    = 2
```

FUNCTION	DESCRIPTION
Table.Contains	Determines whether the a record appears as a row in the table.
Table.ContainsAll	Determines whether all of the specified records appear as rows in the table.
Table.ContainsAny	Determines whether any of the specified records appear as rows in the table.
Table.Distinct	Removes duplicate rows from a table, ensuring that all remaining rows are distinct.
Table.IsDistinct	Determines whether a table contains only distinct rows.
Table.PositionOf	Determines the position or positions of a row within a table.
Table.PositionOfAny	Determines the position or positions of any of the specified rows within the table.
Table.RemoveMatchingRows	Removes all occurrences of rows from a table.
Table.ReplaceMatchingRows	Replaces specific rows from a table with the new rows.

Ordering

Example data

The following tables are used by the examples in this section.

Employees table

```
Employees = Table.FromRecords(
    {[Name="Bill",   Level=7,  Salary=100000],
     [Name="Barb",   Level=8,  Salary=150000],
     [Name="Andrew", Level=6,  Salary=85000],
     [Name="Nikki",  Level=5,  Salary=75000],
     [Name="Margo",  Level=3,  Salary=45000],
     [Name="Jeff",   Level=10, Salary=200000]},
type table [
    Name = text,
    Level = number,
    Salary = number
])
```

FUNCTION	DESCRIPTION
Table.Max	Returns the largest row or rows from a table using a comparisonCriteria.
Table.MaxN	Returns the largest N rows from a table. After the rows are sorted, the countOrCondition parameter must be specified to further filter the result.
Table.Min	Returns the smallest row or rows from a table using a comparisonCriteria.
Table.MinN	Returns the smallest N rows in the given table. After the rows are sorted, the countOrCondition parameter must be specified to further filter the result.
Table.Sort	Sorts the rows in a table using a comparisonCriteria or a default ordering if one is not specified.

Other

FUNCTION	DESCRIPTION
Table.Buffer	Buffers a table into memory, isolating it from external changes during evaluation.

Parameter Values

Naming output columns

This parameter is a list of text values specifying the column names of the resulting table. This parameter is generally used in the Table construction functions, such as Table.FromRows and Table.FromList.

Comparison criteria

Comparison criterion can be provided as either of the following values:

- A number value to specify a sort order. See sort order in the parameter values section above.
- To compute a key to be used for sorting, a function of 1 argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order.
- To completely control the comparison, a function of 2 arguments can be used that returns -1, 0, or 1 given the relationship between the left and right inputs. Value.Compare is a method that can be used to delegate this logic.

For examples, see description of Table.Sort.

Count or Condition critieria

This criteria is generally used in ordering or row operations. It determines the number of rows returned in the table and can take two forms, a number or a condition:

- A number indicates how many values to return inline with the appropriate function
- If a condition is specified, the rows containing values that initially meet the condition is returned. Once a value fails the condition, no further values are considered.

See Table.FirstN or Table.MaxN.

Handling of extra values

This is used to indicate how the function should handle extra values in a row. This parameter is specified as a number, which maps to the options below.

```
ExtraValues.List = 0

ExtraValues.Error = 1

ExtraValues.Ignore = 2
```

For more information, see Table.FromList.

Missing column handling

This is used to indicate how the function should handle missing columns. This parameter is specified as a number, which maps to the options below.

```
MissingField.Error = 0;

MissingField.Ignore = 1;

MissingField.UseNull = 2;
```

This is used in column or transformation operations. For Examples, see Table.TransformColumns.

Sort Order

This is used to indicate how the results should be sorted. This parameter is specified as a number, which maps to the options below.

```
Order.Ascending = 0

    Order.Descending = 1
```

Equation criteria

Equation criteria for tables can be specified as either a

- A function value that is either
  - A key selector that determines the column in the table to apply the equality criteria, or
  - A comparer function that is used to specify the kind of comparison to apply. Built in comparer functions can be specified, see section for Comparer functions.
- A list of the columns in the table to apply the equality criteria

For examples, look at description for Table.Distinct.

ExtraValues.Error

3/15/2021 • 2 minutes to read

About

If the splitter function returns more columns than the table expects, an error should be raised.

ExtraValues.Ignore

3/15/2021 • 2 minutes to read

About

If the splitter function returns more columns than the table expects, they should be ignored.

ExtraValues.List

3/15/2021 • 2 minutes to read

About

If the splitter function returns more columns than the table expects, they should be collected into a list.

GroupKind.Global

3/15/2021 • 2 minutes to read

About

Syntax

```
GroupKind.Global
```

GroupKind.Local

3/15/2021 • 2 minutes to read

Syntax

```
GroupKind.Local
```

About

GroupKind.Local

ItemExpression.From

3/15/2021 • 2 minutes to read

Syntax

```
ItemExpression.From(function as function) as record
```

About

Returns the AST for the body of `function`, normalized into an *item expression*:

- The function must be a 1-argument lambda.
- All references to the function parameter are replaced with `ItemExpression.Item`.
- The AST will be simplified to contain only nodes of the kinds:
  - `Constant`
  - `Invocation`
  - `Unary`
  - `Binary`
  - `If`
  - `FieldAccess`
  - `NotImplemented`

An error is raised if an item expression AST cannot be returned for the body of `function`.

Example 1

Returns the AST for the body of the function `each _ <> null`

```
ItemExpression.From(each _ <> null)
```

KIND	Binary
OPERATOR	NotEquals
LEFT	[Record]
RIGHT	[Record]

ItemExpression.Item

3/15/2021 • 2 minutes to read

About

An AST node representing the item in an item expression.

JoinAlgorithm.Dynamic

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.Dynamic

JoinAlgorithm.LeftHash

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.LeftHash

JoinAlgorithm.LeftIndex

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.LeftIndex

JoinAlgorithm.PairwiseHash

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.PairwiseHash

JoinAlgorithm.RightHash

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.RightHash

JoinAlgorithm.RightIndex

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.RightIndex

JoinAlgorithm.SortMerge

3/15/2021 • 2 minutes to read

About

JoinAlgorithm.SortMerge

JoinKind.FullOuter

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. A full outer join ensures that all rows of both tables appear in the result. Rows that did not have a match in the other table are joined with a default row containing null values for all of its columns.

JoinKind.Inner

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. The table resulting from an inner join contains a row for each pair of rows from the specified tables that were determined to match based on the specified key columns.

JoinKind.LeftAnti

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. A left anti join returns that all rows from the first table which do not have a match in the second table.

JoinKind.LeftOuter

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. A left outer join ensures that all rows of the first table appear in the result.

JoinKind.RightAnti

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. A right anti join returns that all rows from the second table which do not have a match in the first table.

JoinKind.RightOuter

3/15/2021 • 2 minutes to read

About

A possible value for the optional `JoinKind` parameter in `Table.Join`. A right outer join ensures that all rows of the second table appear in the result.

JoinSide.Left

3/15/2021 • 2 minutes to read

About

Specifies the left table of a join.

JoinSide.Right

3/15/2021 • 2 minutes to read

About

Specifies the right table of a join.

Occurrence.All

3/15/2021 • 2 minutes to read

About

A list of positions of all occurrences of the found values is returned.

Occurrence.First

3/15/2021 • 2 minutes to read

About

The position of the first occurrence of the found value is returned.

Occurrence.Last

3/15/2021 • 2 minutes to read

About

The position of the last occurrence of the found value is returned.

Order.Ascending

3/15/2021 • 2 minutes to read

About

Function type which sorts the list in ascending order.

Order.Descending

3/15/2021 • 2 minutes to read

About

Function type which sorts the list in descending order.

RowExpression.Column

3/15/2021 • 2 minutes to read

Syntax

```
RowExpression.Column(columnName as text) as record
```

About

Returns an AST that represents access to column columnName of the row within a row expression.

Example 1

Creates an AST representing access of column "CustomerName".

```
RowExpression.Column("CustomerName")
```

KIND	FieldAccess
EXPRESSION	[Record]
MEMBERNAME	CustomerName

RowExpression.From

3/15/2021 • 2 minutes to read

Syntax

```
RowExpression.From(function as function) as record
```

About

Returns the AST for the body of `function`, normalized into a *row expression*.

- The function must be a 1-argument lambda.
- All references to the function parameter are replaced with `RowExpression.Row`.
- All references to columns are replaced with `RowExpression.Column(columnName)`.
- The AST will be simplified to contain only nodes of the kinds:
  - `Constant`
  - `Invocation`
  - `Unary`
  - `Binary`
  - `If`
  - `FieldAccess`
  - `NotImplemented`

An error is raised if a row expression AST cannot be returned for the body of `function`.

Example 1

Returns the AST for the body of the function each [CustomerID] = "ALFKI"

```
RowExpression.From(each [CustomerName] = "ALFKI")
```

KIND	Binary
OPERATOR	Equals
LEFT	[Record]
RIGHT	[Record]

RowExpression.Row

3/15/2021 • 2 minutes to read

About

An AST node representing the row in a row expression.

Table.AddColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.AddColumn(table as table, newColumnName as text, columnGenerator as function, optional
columnType as nullable type) as table
```

About

Adds a column named `newColumnName` to the table `table`. The values for the column are computed using the specified selection function `columnGenerator` with each row taken as an input.

Example 1

Add a column named "TotalPrice" to the table with each value being the sum of column [Price] and column [Shipping].

```
Table.AddColumn(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0, Shipping = 10.00],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0, Shipping = 15.00],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0, Shipping = 10.00]
    }),
    "TotalPrice",
    each [Price] + [Shipping]
)
```

ORDERID	CUSTOMERID	ITEM	PRICE	SHIPPING	TOTALPRICE
1	1	Fishing rod	100	10	110
2	1	1 lb. worms	5	15	20
3	2	Fishing net	25	10	35

Table.AddFuzzyClusterColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.AddFuzzyClusterColumn(table as table, columnName as text, newColumnName as text, optional
options as nullable record) as table
```

About

Adds a new column `newColumnName` to `table` with representative values of `columnName`. The representatives are obtained by fuzzily matching values in `columnName`, for each row.

An optional set of `options` may be included to specify how to compare the key columns. Options include:

- `Culture`: Allows grouping records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" groups records based on the Japanese culture. The default value is "", which groups based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key grouping. For example, when true, "Grapes" is grouped with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find groups. For example, when true, "Gra pes" is grouped with "Grapes". The default value is true.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be grouped. For example, "Grapes" and "Graes" (missing "p") are grouped together only if this option is set to less than 0.90. A threshold of 1.00 is the same as specifying an exact match criteria while grouping. The default value is 0.80.
- `TransformationTable`: A table that allows grouping records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is grouped with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be grouped with "Raisins are sweet".

Example 1

Find the representative values for the location of the employees.

```
Table.AddFuzzyClusterColumn(
    Table.FromRecords(
        {
            [EmployeeID = 1, Location = "Seattle"],
            [EmployeeID = 2, Location = "seattl"],
            [EmployeeID = 3, Location = "Vancouver"],
            [EmployeeID = 4, Location = "Seatle"],
            [EmployeeID = 5, Location = "vancover"],
            [EmployeeID = 6, Location = "Seattle"],
            [EmployeeID = 7, Location = "Vancouver"]
        },
        type table [EmployeeID = nullable number, Location = nullable text]
    ),
    "Location",
    "Location_Cleaned",
    [IgnoreCase = true, IgnoreSpace = true]
)
```

```
Table.FromRecords(
    {
        [EmployeeID = 1, Location = "Seattle", Location_Cleaned = "Seattle"],
        [EmployeeID = 2, Location = "seattl", Location_Cleaned = "Seattle"],
        [EmployeeID = 3, Location = "Vancouver", Location_Cleaned = "Vancouver"],
        [EmployeeID = 4, Location = "Seatle", Location_Cleaned = "Seattle"],
        [EmployeeID = 5, Location = "vancover", Location_Cleaned = "Vancouver"],
        [EmployeeID = 6, Location = "Seattle", Location_Cleaned = "Seattle"],
        [EmployeeID = 7, Location = "Vancouver", Location_Cleaned = "Vancouver"]
    },
    type table [EmployeeID = nullable number, Location = nullable text, Location_Cleaned = nullable text]
)
```

Table.AddIndexColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.AddIndexColumn(table as table, newColumnName as text, optional initialValue as nullable
number, optional increment as nullable number, optional columnType as nullable type) as table
```

About

Appends a column named `newColumnName` to the `table` with explicit position values. An optional value, `initialValue`, the initial index value. An optional value, `increment`, specifies how much to increment each index value.

Example 1

Add an index column named "Index" to the table.

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index"
)
```

CUSTOMERID	NAME	PHONE	INDEX
1	Bob	123-4567	0
2	Jim	987-6543	1
3	Paul	543-7890	2
4	Ringo	232-1550	3

Example 2

Add an index column named "index", starting at value 10 and incrementing by 5, to the table.

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index",
    10,
    5
)
```

CUSTOMERID	NAME	PHONE	INDEX
1	Bob	123-4567	10
2	Jim	987-6543	15
3	Paul	543-7890	20
4	Ringo	232-1550	25

Table.AddJoinColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.AddJoinColumn(table1 as table, key1 as any, table2 as function, key2 as any, newColumnName as text) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are entered into the column named `newColumnName`. This function behaves similarly to Table.Join with a JoinKind of LeftOuter except that the join results are presented in a nested rather than flattened fashion.

Example 1

Add a join column to ({[saleID = 1, item = "Shirt"], [saleID = 2, item = "Hat"]}) named "price/stock" from the table ({[saleID = 1, price = 20], [saleID = 2, price = 10]}) joined on [saleID].

```
Table.AddJoinColumn(
    Table.FromRecords({
        [saleID = 1, item = "Shirt"],
        [saleID = 2, item = "Hat"]
    }),
    "saleID",
    () => Table.FromRecords({
        [saleID = 1, price = 20, stock = 1234],
        [saleID = 2, price = 10, stock = 5643]
    }),
    "saleID",
    "price"
)
```

SALEID	ITEM	PRICE
1	Shirt	[Table]
2	Hat	[Table]

Table.AddKey

3/15/2021 • 2 minutes to read

Syntax

```
Table.AddKey(table as table, columns as list, isPrimary as logical) as table
```

About

Add a key to `table`, given `columns` is the subset of `table`'s column names that defines the key, and `isPrimary` specifies whether the key is primary.

Example 1

Add a key to {[Id = 1, Name = "Hello There"], [Id = 2, Name = "Good Bye"]} that comprise of {"Id"} and make it a primary.

```
let
    tableType = type table [Id = Int32.Type, Name = text],
    table = Table.FromRecords({
        [Id = 1, Name = "Hello There"],
        [Id = 2, Name = "Good Bye"]
    }),
    resultTable = Table.AddKey(table, {"Id"}, true)
in
    resultTable
```

ID	NAME
1	Hello There
2	Good Bye

Table.AggregateTableColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.AggregateTableColumn(table as table, column as text, aggregations as list) as table
```

About

Aggregates tables in `table` [`column`] into multiple columns containing aggregate values for the tables. `aggregations` is used to specify the columns containing the tables to aggregate, the aggregation functions to apply to the tables to generate their values, and the names of the aggregate columns to create.

Example 1

Aggregate table columns in `[t]` in the table `{[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2]}` into the sum of `[t.a]`, the min and max of `[t.b]`, and the count of values in `[t.a]`.

```
Table.AggregateTableColumn(
    Table.FromRecords(
        {
            [
                t = Table.FromRecords({
                    [a = 1, b = 2, c = 3],
                    [a = 2, b = 4, c = 6]
                }),
                b = 2
            ]
        },
        type table [t = table [a = number, b = number, c = number], b = number]
    ),
    "t",
    {
        {"a", List.Sum, "sum of t.a"},
        {"b", List.Min, "min of t.b"},
        {"b", List.Max, "max of t.b"},
        {"a", List.Count, "count of t.a"}
    }
)
```

SUM OF T.A	MIN OF T.B	MAX OF T.B	COUNT OF T.A	B
3	2	4	2	2

Table.AlternateRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.AlternateRows(table as table, offset as number, skip as number, take as number) as table
```

About

Keeps the initial offset then alternates taking and skipping the following rows.

- `table`: The input table.
- `offset`: The number of rows to keep before starting iterations.
- `skip`: The number of rows to remove in each iteration.
- `take`: The number of rows to keep in each iteration.

Example 1

Return a table from the table that, starting at the first row, skips 1 value and then keeps 1 value.

```
Table.AlternateRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    1,
    1,
    1
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
3	Paul	543-7890

Table.ApproximateRowCount

6/22/2021 • 2 minutes to read

Syntax

```
Table.ApproximateRowCount(table as table) as number
```

About

Returns the approximate number of rows in the `table`.

Table.Buffer

3/15/2021 • 2 minutes to read

Syntax

```
Table.Buffer(table as table) as table
```

About

Buffers a table in memory, isolating it from external changes during evaluation.

Table.Column

3/15/2021 • 2 minutes to read

Syntax

```
Table.Column(table as table, column as text) as list
```

About

Returns the column of data specified by `column` from the table `table` as a list.

Example 1

Returns the values from the [Name] column in the table.

```
Table.Column(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

Bob
Jim
Paul
Ringo

Table.ColumnCount

3/15/2021 • 2 minutes to read

Syntax

```
Table.ColumnCount(table as table) as number
```

About

Returns the number of columns in the table `table`.

Example 1

Find the number of columns in the table.

```
Table.ColumnCount(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

`3`

Table.ColumnNames

3/15/2021 • 2 minutes to read

Syntax

```
Table.ColumnNames(table as table) as list
```

About

Returns the column names in the table `table` as a list of text.

Example 1

Find the column names of the table.

```
Table.ColumnNames(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

CustomerID
Name
Phone

Table.ColumnsOfType

3/15/2021 • 2 minutes to read

Syntax

```
Table.ColumnsOfType(table as table, listOfTypes as list) as list
```

About

Returns a list with the names of the columns from table `table` that match the types specified in `listOfTypes`.

Example 1

Return the names of columns of type Number.Type from the table.

```
Table.ColumnsOfType(
    Table.FromRecords(
        {[a = 1, b = "hello"]},
        type table[a = Number.Type, b = Text.Type]
    ),
    {type number}
)
```

```
a
```

Table.Combine

3/15/2021 • 2 minutes to read

Syntax

```
Table.Combine(tables as list, optional columns as any) as table
```

About

Returns a table that is the result of merging a list of tables, `tables`. The resulting table will have a row type structure defined by `columns` or by a union of the input types if `columns` is not specified.

Example 1

Merge the three tables together.

```
Table.Combine({
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    Table.FromRecords({[CustomerID = 2, Name = "Jim", Phone = "987-6543"]}),
    Table.FromRecords({[CustomerID = 3, Name = "Paul", Phone = "543-7890"]})
})
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

Example 2

Merge three tables with different structures.

```
Table.Combine({
    Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
    Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
    Table.FromRecords({[Cell = "543-7890"]})
})
```

NAME	PHONE	FAX	CELL
Bob	123-4567		
	838-7171	987-6543	
			543-7890

Example 3

Merge two tables and project onto the given type.

```
Table.Combine(
    {
        Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
        Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
        Table.FromRecords({[Cell = "543-7890"]})
    },
    {"CustomerID", "Name"}
)
```

CUSTOMERID	NAME
	Bob

Table.CombineColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.CombineColumns(table as table, sourceColumns as list, combiner as function, column as
text) as table
```

About

Combines the specified columns into a new column using the specified combiner function.

Table.CombineColumnsToRecord

3/15/2021 • 2 minutes to read

Syntax

```
Table.CombineColumnsToRecord(table as table, newColumnName as text, sourceColumns as list,
optional options as nullable record) as table
```

About

Combines the specified columns of `table` into a new record-valued column named `newColumnName` where each record has field names and values corresponding to the column names and values of the columns that were combined. If a record is specified for `options`, the following options may be provided:

- `DisplayNameColumn`: When specified as text, indicates that the given column name should be treated as the display name of the record. This need not be one of the columns in the record itself.
- `TypeName`: When specified as text, supplies a logical type name for the resulting record which can be used during data load to drive behavior by the loading environment.

Table.ConformToPageReader

3/15/2021 • 2 minutes to read

Syntax

```
Table.ConformToPageReader(table as table, shapingFunction as function) as table
```

About

This function is intended for internal use only.

Table.Contains

3/15/2021 • 2 minutes to read

Syntax

```
Table.Contains(table as table, row as record, optional equationCriteria as any) as logical
```

About

Indicates whether the specified record, `row`, appears as a row in the `table`. An optional parameter `equationCriteria` may be specified to control comparison between the rows of the table.

Example 1

Determine if the table contains the row.

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [Name = "Bob"]
)
```

`true`

Example 2

Determine if the table contains the row.

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [Name = "Ted"]
)
```

`false`

Example 3

Determine if the table contains the row comparing only the column [Name].

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [CustomerID = 4, Name = "Bob"],
    "Name"
)
```

`true`

Table.ContainsAll

3/15/2021 • 2 minutes to read

Syntax

```
Table.ContainsAll(table as table, rows as list, optional equationCriteria as any) as logical
```

About

Indicates whether all the specified records in the list of records `rows`, appear as rows in the `table`. An optional parameter `equationCriteria` may be specified to control comparison between the rows of the table.

Example 1

Determine if the table contains all the rows, comparing only the column [CustomerID].

```
Table.ContainsAll(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {
        [CustomerID = 1, Name = "Bill"],
        [CustomerID = 2, Name = "Fred"]
    },
    "CustomerID"
)
```

`true`

Example 2

Determine if the table contains all the rows.

```
Table.ContainsAll(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {
        [CustomerID = 1, Name = "Bill"],
        [CustomerID = 2, Name = "Fred"]
    }
)
```

`false`

Table.ContainsAny

3/15/2021 • 2 minutes to read

Syntax

```
Table.ContainsAny(table as table, rows as list, optional equationCriteria as any) as logical
```

About

Indicates whether any the specified records in the list of records `rows`, appear as rows in the `table`. An optional parameter `equationCriteria` may be specified to control comparison between the rows of the table.

Example 1

Determine if the table `({[a = 1, b = 2], [a = 3, b = 4]})` contains the rows `[a = 1, b = 2]` or `[a = 3, b = 5]`.

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 2],
        [a = 3, b = 5]
    }
)
```

`true`

Example 2

Determine if the table `({[a = 1, b = 2], [a = 3, b = 4]})` contains the rows `[a = 1, b = 3]` or `[a = 3, b = 5]`.

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 3],
        [a = 3, b = 5]
    }
)
```

`false`

Example 3

Determine if the table `(Table.FromRecords({[a = 1, b = 2], [a = 3, b = 4]}))` contains the rows

`[a = 1, b = 3]` or `[a = 3, b = 5]` comparing only the column [a].

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 3],
        [a = 3, b = 5]
    },
    "a"
)
```

`true`

Table.DemoteHeaders

3/15/2021 • 2 minutes to read

Syntax

```
Table.DemoteHeaders(table as table) as table
```

About

Demotes the column headers (i.e. column names) to the first row of values. The default column names are "Column1", "Column2" and so on.

Example 1

Demote the first row of values in the table.

```
Table.DemoteHeaders(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    })
)
```

COLUMN1	COLUMN2	COLUMN3
CustomerID	Name	Phone
1	Bob	123-4567
2	Jim	987-6543

Table.Distinct

3/15/2021 • 2 minutes to read

Syntax

```
Table.Distinct(table as table, optional equationCriteria as any) as table
```

About

Removes duplicate rows from the table `table`. An optional parameter, `equationCriteria`, specifies which columns of the table are tested for duplication. If `equationCriteria` is not specified, all columns are tested.

Example 1

Remove the duplicate rows from the table.

```
Table.Distinct(
    Table.FromRecords({
        [a = "A", b = "a"],
        [a = "B", b = "b"],
        [a = "A", b = "a"]
    })
)
```

A	B
A	a
B	b

Example 2

Remove the duplicate rows from column [b] in the table `({[a = "A", b = "a"], [a = "B", b = "a"], [a = "A", b = "b"]})`.

```
Table.Distinct(
    Table.FromRecords({
        [a = "A", b = "a"],
        [a = "B", b = "a"],
        [a = "A", b = "b"]
    }),
    "b"
)
```

A	B
A	a
A	b

Table.DuplicateColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.DuplicateColumn(table as table, columnName as text, newColumnName as text, optional
columnType as nullable type) as table
```

About

Duplicate the column named `columnName` to the table `table`. The values and type for the column `newColumnName` are copied from column `columnName`.

Example

Duplicate the column "a" to a column named "copied column" in the table `({[a = 1, b = 2], [a = 3, b = 4]})`.

```
Table.DuplicateColumn(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    "a",
    "copied column"
)
```

A	B	COPIED COLUMN
1	2	1
3	4	3

Table.ExpandListColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.ExpandListColumn(table as table, column as text) as table
```

About

Given a `table`, where a `column` is a list of values, splits the list into a row for each value. Values in the other columns are duplicated in each new row created.

Example 1

Split the list column [Name] in the table.

```
Table.ExpandListColumn(
    Table.FromRecords({[Name = {"Bob", "Jim", "Paul"}, Discount = .15]}),
    "Name"
)
```

NAME	DISCOUNT
Bob	0.15
Jim	0.15
Paul	0.15

Table.ExpandRecordColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.ExpandRecordColumn(table as table, column as text, fieldNames as list, optional
newColumnNames as nullable list) as table
```

About

Given the `column` of records in the input `table`, creates a table with a column for each field in the record. Optionally, `newColumnNames` may be specified to ensure unique names for the columns in the new table.

- `table`: The original table with the record column to expand.
- `column`: The column to expand.
- `fieldNames`: The list of fields to expand into columns in the table.
- `newColumnNames`: The list of column names to give the new columns. The new column names cannot duplicate any column in the new table.

Example 1

Expand column [a] in the table `({[a = [aa = 1, bb = 2, cc = 3], b = 2]})` into 3 columns "aa", "bb" and "cc".

```
Table.ExpandRecordColumn(
    Table.FromRecords({
        [
            a = [aa = 1, bb = 2, cc = 3],
            b = 2
        ]
    }),
    "a",
    {"aa", "bb", "cc"}
)
```

AA	BB	CC	B
1	2	3	2

Table.ExpandTableColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.ExpandTableColumn(table as table, column as text, columnNames as list, optional
newColumnNames as nullable list) as table
```

About

Expands tables in `table` [`column`] into multiple rows and columns. `columnNames` is used to select the columns to expand from the inner table. Specify `newColumnNames` to avoid conflicts between existing columns and new columns.

Example 1

Expand table columns in `[a]` in the table `({[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2]})` into 3 columns `[t.a]`, `[t.b]` and `[t.c]`.

```
Table.ExpandTableColumn(
    Table.FromRecords({
        [
            t = Table.FromRecords({
                [a = 1, b = 2, c = 3],
                [a = 2, b = 4, c = 6]
            }),
            b = 2
        ]
    }),
    "t",
    {"a", "b", "c"},
    {"t.a", "t.b", "t.c"}
)
```

T.A	T.B	T.C	B
1	2	3	2
2	4	6	2

Table.FillDown

3/15/2021 • 2 minutes to read

Syntax

```
Table.FillDown(table as table, columns as list) as table
```

About

Returns a table from the `table` specified where the value of a previous cell is propagated to the null-valued cells below in the `columns` specified.

Example 1

Return a table with the null values in column [Place] filled with the value above them from the table.

```
Table.FillDown(
    Table.FromRecords({
        [Place = 1, Name = "Bob"],
        [Place = null, Name = "John"],
        [Place = 2, Name = "Brad"],
        [Place = 3, Name = "Mark"],
        [Place = null, Name = "Tom"],
        [Place = null, Name = "Adam"]
    }),
    {"Place"}
)
```

PLACE	NAME
1	Bob
1	John
2	Brad
3	Mark
3	Tom
3	Adam

Table.FillUp

3/15/2021 • 2 minutes to read

Syntax

```
Table.FillUp(table as table, columns as list) as table
```

About

Returns a table from the `table` specified where the value of the next cell is propagated to the null-valued cells above in the `columns` specified.

Example 1

Return a table with the null values in column [Column2] filled with the value below them from the table.

```
Table.FillUp(
    Table.FromRecords({
        [Column1 = 1, Column2 = 2],
        [Column1 = 3, Column2 = null],
        [Column1 = 5, Column2 = 3]
    }),
    {"Column2"}
)
```

COLUMN1	COLUMN2
1	2
3	3
5	3

Table.FilterWithDataTable

3/15/2021 • 2 minutes to read

Syntax

```
Table.FilterWithDataTable(**table** as table, **dataTableIdentifier** as text) as any
```

About

Table.FilterWithDataTable

Table.FindText

3/15/2021 • 2 minutes to read

Syntax

```
Table.FindText(table as table, text as text) as table
```

About

Returns the rows in the table `table` that contain the text `text`. If the text is not found, an empty table is returned.

Example 1

Find the rows in the table that contain "Bob".

```
Table.FindText(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Bob"
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Table.First

3/15/2021 • 2 minutes to read

Syntax

```
Table.First(table as table, optional default as any) as any
```

About

Returns the first row of the `table` or an optional default value, `default`, if the table is empty.

Example 1

Find the first row of the table.

```
Table.First(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Example 2

Find the first row of the table `({})` or return [a = 0, b = 0] if empty.

```
Table.First(Table.FromRecords({}), [a = 0, b = 0])
```

A	0
B	0

Table.FirstN

3/15/2021 • 2 minutes to read

Syntax

```
Table.FirstN(table as table, countOrCondition as any) as table
```

About

Returns the first row(s) of the table `table`, depending on the value of `countOrCondition`:

- If `countOrCondition` is a number, that many rows (starting at the top) will be returned.
- If `countOrCondition` is a condition, the rows that meet the condition will be returned until a row does not meet the condition.

Example 1

Find the first two rows of the table.

```
Table.FirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    2
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
2	Jim	987-6543

Example 2

Find the first rows where [a] > 0 in the table.

```
Table.FirstN(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4],
        [a = -5, b = -6]
    }),
    each [a] > 0
)
```

A	B
1	2

3	4

Table.FirstValue

3/15/2021 • 2 minutes to read

Syntax

```
Table.FirstValue(table as table, optional default as any) as any
```

About

Returns the first column of the first row of the table `table` or a specified default value.

Table.FromColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromColumns(lists as list, optional columns as any) as table
```

About

Creates a table of type `columns` from a list `lists` containing nested lists with the column names and values. If some columns have more values then others, the missing values will be filled with the default value, 'null', if the columns are nullable.

Example 1

Return a table from a list of customer names in a list. Each value in the customer list item becomes a row value, and each list becomes a column.

```
Table.FromColumns({
    {1, "Bob", "123-4567"},
    {2, "Jim", "987-6543"},
    {3, "Paul", "543-7890"}
})
```

COLUMN1	COLUMN2	COLUMN3
1	2	3
Bob	Jim	Paul
123-4567	987-6543	543-7890

Example 2

Create a table from a given list of columns and a list of column names.

```
Table.FromColumns(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"},
        {3, "Paul", "543-7890"}
    },
    {"CustomerID", "Name", "Phone"}
)
```

CUSTOMERID	NAME	PHONE
1	2	3

Bob	Jim	Paul
123-4567	987-6543	543-7890

Example 3

Create a table with different number of columns per row. The missing row value is null.

```
Table.FromColumns(
    {
        {1, 2, 3},
        {4, 5},
        {6, 7, 8, 9}
    },
    {"column1", "column2", "column3"}
)
```

COLUMN1	COLUMN2	COLUMN3
1	4	6
2	5	7
3		8
		9

Table.FromList

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromList(list as list, optional splitter as nullable function, optional columns as any,
optional default as any, optional extraValues as nullable number) as table
```

About

Converts a list, `list` into a table by applying the optional splitting function, `splitter`, to each item in the list. By default, the list is assumed to be a list of text values that is split by commas. Optional `columns` may be the number of columns, a list of columns or a TableType. Optional `default` and `extraValues` may also be specified.

Example 1

Create a table from the list with the column named "Letters" using the default splitter.

```
Table.FromList({"a", "b", "c", "d"}, null, {"Letters"})
```

LETTERS
a
b
c
d

Example 2

Create a table from the list using the Record.FieldValues splitter with the resulting table having "CustomerID" and "Name" as column names.

```
Table.FromList(
    {
        [CustomerID = 1, Name = "Bob"],
        [CustomerID = 2, Name = "Jim"]
    },
    Record.FieldValues,
    {"CustomerID", "Name"}
)
```

CUSTOMERID	NAME
1	Bob

2	Jim

Table.FromPartitions

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromPartitions(partitionColumn as text, partitions as list, optional partitionColumnType
as nullable type) as table
```

About

Returns a table that is the result of combining a set of partitioned tables, `partitions`. `partitionColumn` is the name of the column to add. The type of the column defaults to `any`, but can be specified by `partitionColumnType`.

Example 1

Find item type from the list `{number}`.

```
Table.FromPartitions(
    "Year",
    {
        {
            1994,
            Table.FromPartitions(
                "Month",
                {
                    {
                        "Jan",
                        Table.FromPartitions(
                            "Day",
                            {
                                {1, #table({"Foo"}, {{"Bar"}})},
                                {2, #table({"Foo"}, {{"Bar"}})}
                            }
                        )
                    },
                    {
                        "Feb",
                        Table.FromPartitions(
                            "Day",
                            {
                                {3, #table({"Foo"}, {{"Bar"}})},
                                {4, #table({"Foo"}, {{"Bar"}})}
                            }
                        )
                    }
                }
            )
        }
    }
)
```

FOO	DAY	MONTH	YEAR
Bar	1	Jan	1994

Bar	2	Jan	1994
Bar	3	Feb	1994
Bar	4	Feb	1994

Table.FromRecords

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromRecords(records as list, optional columns as any, optional missingField as nullable
number) as table
```

About

Converts `records`, a list of records, into a table.

Example 1

Create a table from records, using record field names as column names.

```
Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
})
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

Example 2

Create a table from records with typed columns and select the number columns.

```
Table.ColumnsOfType(
    Table.FromRecords(
        {[CustomerID = 1, Name = "Bob"]},
        type table[CustomerID = Number.Type, Name = Text.Type]
    ),
    {type number}
)
```

CustomerID

Table.FromRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromRows(rows as list, optional columns as any) as table
```

About

Creates a table from the list `rows` where each element of the list is an inner list that contains the column values for a single row. An optional list of column names, a table type, or a number of columns could be provided for `columns`.

Example 1

Return a table with column [CustomerID] with values {1, 2}, column [Name] with values {"Bob", "Jim"}, and column [Phone] with values {"123-4567", "987-6543"}.

```
Table.FromRows(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"}
    },
    {"CustomerID", "Name", "Phone"}
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
2	Jim	987-6543

Example 2

Return a table with column [CustomerID] with values {1, 2}, column [Name] with values {"Bob", "Jim"}, and column [Phone] with values {"123-4567", "987-6543"}, where [CustomerID] is number type, and [Name] and [Phone] are text types.

```
Table.FromRows(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"}
    },
    type table [CustomerID = number, Name = text, Phone = text]
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

2	Jim	987-6543

Table.FromValue

3/15/2021 • 2 minutes to read

Syntax

```
Table.FromValue(value as any, optional options as nullable record) as table
```

About

Creates a table with a column containing the provided value or list of values, `value`. An optional record parameter, `options`, may be specified to control the following options:

- `DefaultColumnName` : The column name used when constructing a table from a list or scalar value.

Example 1

Create a table from the value 1.

```
Table.FromValue(1)
```

VALUE
1

Example 2

Create a table from the list.

```
Table.FromValue({1, "Bob", "123-4567"})
```

VALUE
1
Bob
123-4567

Example 3

Create a table from the value 1, with a custom column name.

```
Table.FromValue(1, [DefaultColumnName = "MyValue"])
```

MYVALUE

1

Table.FuzzyGroup

3/15/2021 • 2 minutes to read

Syntax

```
Table.FuzzyGroup(table as table, key as any, aggregatedColumns as list, optional options as
nullable record) as table
```

About

Groups the rows of `table` by fuzzily matching values in the specified column, `key`, for each row. For each group, a record is constructed containing the key columns (and their values) along with any aggregated columns specified by `aggregatedColumns`. This function cannot guarantee to return a fixed order of rows.

An optional set of `options` may be included to specify how to compare the key columns. Options include:

- `Culture`: Allows grouping records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" groups records based on the Japanese culture. The default value is "", which groups based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key grouping. For example, when true, "Grapes" is grouped with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find groups. For example, when true, "Gra pes" is grouped with "Grapes". The default value is true.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be grouped. For example, "Grapes" and "Graes" (missing "p") are grouped together only if this option is set to less than 0.90. A threshold of 1.00 is the same as specifying an exact match criteria while grouping. The default value is 0.80.
- `TransformationTable`: A table that allows grouping records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is grouped with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be grouped with "Raisins are sweet".

Example

Group the table adding an aggregate column [Count] that contains the number of employees in each location (`each Table.RowCount(_)`).

```
Table.FuzzyGroup(
    Table.FromRecords(
        {
            [EmployeeID = 1, Location = "Seattle"],
            [EmployeeID = 2, Location = "seattl"],
            [EmployeeID = 3, Location = "Vancouver"],
            [EmployeeID = 4, Location = "Seatle"],
            [EmployeeID = 5, Location = "vancover"],
            [EmployeeID = 6, Location = "Seattle"],
            [EmployeeID = 7, Location = "Vancouver"]
        },
        type table [EmployeeID = nullable number, Location = nullable text]
    ),
    "Location",
    {"Count", each Table.RowCount(_)},
    [IgnoreCase = true, IgnoreSpace = true]
)
```

LOCATION	COUNT
Seattle	4
Vancouver	3

Table.FuzzyJoin

3/15/2021 • 2 minutes to read

Syntax

```
Table.FuzzyJoin(table1 as table, key1 as any, table2 as table, key2 as any, optional joinKind as nullable number, optional joinOptions as nullable record) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on a fuzzy matching of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`).

Fuzzy matching is a comparison based on similarity of text rather than equality of text.

By default, an inner join is performed, however an optional `joinKind` may be included to specify the type of join. Options include:

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

An optional set of `joinOptions` may be included to specify how to compare the key columns. Options include:

- `ConcurrentRequests`: A number between 1 and 8 that specifies the number of parallel threads to use for fuzzy matching. The default value is 1.
- `Culture`: Allows matching records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" matches records based on the Japanese culture. The default value is "", which matches based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key matching. For example, when true, "Grapes" is matched with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find matches. For example, when true, "Gra pes" is matched with "Grapes". The default value is true.
- `NumberOfMatches`: A whole number that specifies the maximum number of matching rows that can be returned for every input row. For example, a value of 1 will return at most one matching row for each input row. If this option is not provided, all matching rows are returned.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be matched. For example, "Grapes" and "Graes" (missing "p") are matched only if this option is set to less than 0.90. A threshold of 1.00 is the same as specifying an exact match criteria. The default value is 0.80.
- `TransformationTable`: A table that allows matching records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is matched with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that

the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be matched with "Raisins are sweet".

Example

Left inner fuzzy join of two tables based on [FirstName]

```
Table.FuzzyJoin(
   Table.FromRecords(
        {
          [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
          [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
       },
        type table [CustomerID = nullable number, FirstName1 = nullable text, Phone = nullable text]
    ),
   {"FirstName1"},
   Table.FromRecords(
        {
          [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
          [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
       },
        type table [CustomerStateID = nullable number, FirstName2 = nullable text, State = nullable text]
    ),
   {"FirstName2"},
   JoinKind.LeftOuter,
   [IgnoreCase = true, IgnoreSpace = false]
)
```

CUSTOMERID	FIRSTNAME1	PHONE	CUSTOMERSTATEID	FIRSTNAME2	STATE
1	Bob	555-1234	1	Bob	TX
1	Bob	555-1234	2	bOB	CA
2	Robert	555-4567			

Table.FuzzyNestedJoin

3/15/2021 • 2 minutes to read

Syntax

```
Table.FuzzyNestedJoin(table1 as table, key1 as any, table2 as table, key2 as any, newColumnName
as text, optional joinKind as nullable number, optional joinOptions as nullable record) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on a fuzzy matching of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are returned in a new column named `newColumnName`.

Fuzzy matching is a comparison based on similarity of text rather than equality of text.

The optional `joinKind` specifies the kind of join to perform. By default, a left outer join is performed if a `joinKind` is not specified. Options include:

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

An optional set of `joinOptions` may be included to specify how to compare the key columns. Options include:

- `ConcurrentRequests`: A number between 1 and 8 that specifies the number of parallel threads to use for fuzzy matching. The default value is 1.
- `Culture`: Allows matching records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" matches records based on the Japanese culture. The default value is "", which matches based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key matching. For example, when true, "Grapes" is matched with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find matches. For example, when true, "Gra pes" is matched with "Grapes". The default value is true.
- `NumberOfMatches`: A whole number that specifies the maximum number of matching rows that can be returned for every input row. For example, a value of 1 will return at most one matching row for each input row. If this option is not provided, all matching rows are returned.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be matched. For example, "Grapes" and "Graes" (missing "p") are matched only if this option is set to less than 0.90. A threshold of 1.00 is the same as specifying an exact match criteria. The default value is 0.80.
- `TransformationTable`: A table that allows matching records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is matched with "Raisins" if a transformation table is

provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be matched with "Raisins are sweet".

Example

Left inner fuzzy join of two tables based on [FirstName]

```
Table.FuzzyNestedJoin(
   Table.FromRecords(
        {
          [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
          [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
       },
        type table [CustomerID = nullable number, FirstName1 = nullable text, Phone = nullable text]
    ),
   {"FirstName1"},
   Table.FromRecords(
        {
          [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
          [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
       },
        type table [CustomerStateID = nullable number, FirstName2 = nullable text, State = nullable text]
    ),
   {"FirstName2"},
   "NestedTable",
   JoinKind.LeftOuter,
   [IgnoreCase = true, IgnoreSpace = false]
)
```

CUSTOMERID	FIRSTNAME1	PHONE	NESTEDTABLE
1	Bob	555-1234	[Table]
2	Robert	555-4567	[Table]

Table.Group

6/14/2021 • 2 minutes to read

Syntax

```
Table.Group(table as table, key as any, aggregatedColumns as list, optional groupKind as
nullable number, optional comparer as nullable function) as table
```

About

Groups the rows of `table` by the key columns defined by `key`. The `key` can either be a single column name, or a list of column names. For each group, a record is constructed containing the key columns (and their values), along with any aggregated columns specified by `aggregatedColumns`. Optionally, `groupKind` and `comparer` may also be specified.

If the data is already sorted by the key columns, then a `groupKind` of GroupKind.Local can be provided. This may improve the performance of grouping in certain cases, since all the rows with a given set of key values are assumed to be contiguous.

When passing a `comparer`, note that if it treats differing keys as equal, a row may be placed in a group whose keys differ from its own.

This function does not guarantee the ordering of the rows it returns.

Example 1

Group the table adding an aggregate column [total] which contains the sum of prices ("each List.Sum([price])").

```
Table.Group(
    Table.FromRecords({
        [CustomerID = 1, price = 20],
        [CustomerID = 2, price = 10],
        [CustomerID = 2, price = 20],
        [CustomerID = 1, price = 10],
        [CustomerID = 3, price = 20],
        [CustomerID = 3, price = 5]
    }),
    "CustomerID",
    {"total", each List.Sum([price])}
)
```

CUSTOMERID	TOTAL
1	30
2	30
3	25

Table.HasColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.HasColumns(table as table, columns as any) as logical
```

About

indicates whether the `table` contains the specified column(s), `columns`. Returns `true` if the table contains the column(s), `false` otherwise.

Example 1

Determine if the table has the column [Name].

```
TTable.HasColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

`true`

Example 2

Find if the table has the column [Name] and [PhoneNumber].

```
Table.HasColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {"Name", "PhoneNumber"}
)
```

`false`

Table.InsertRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.InsertRows(table as table, offset as number, rows as list) as table
```

About

Returns a table with the list of rows, `rows`, inserted into the `table` at the given position, `offset`. Each column in the row to insert much match the column types of the table.

Example 1

Insert the row into the table at position 1.

```
Table.InsertRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    }),
    1,
    {[CustomerID = 3, Name = "Paul", Phone = "543-7890"]}
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
3	Paul	543-7890
2	Jim	987-6543

Example 2

Insert two rows into the table at position 1.

```
Table.InsertRows(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    1,
    {
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

2	Jim	987-6543
3	Paul	543-7890

Table.IsDistinct

3/15/2021 • 2 minutes to read

Syntax

```
Table.IsDistinct(table as table, optional comparisonCriteria as any) as logical
```

About

Indicates whether the `table` contains only distinct rows (no duplicates). Returns `true` if the rows are distinct, `false` otherwise. An optional parameter, `comparisonCriteria`, specifies which columns of the table are tested for duplication. If `comparisonCriteria` is not specified, all columns are tested.

Example 1

Determine if the table is distinct.

```
Table.IsDistinct(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

`true`

Example 2

Determine if the table is distinct in column.

```
Table.IsDistinct(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 5, Name = "Bob", Phone = "232-1550"]
    }),
    "Name"
)
```

`false`

Table.IsEmpty

3/15/2021 • 2 minutes to read

Syntax

```
Table.IsEmpty(table as table) as logical
```

About

Indicates whether the `table` contains any rows. Returns `true` if there are no rows (i.e. the table is empty), `false` otherwise.

Example 1

Determine if the table is empty.

```
Table.IsEmpty(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

`false`

Example 2

Determine if the table `({})` is empty.

```
Table.IsEmpty(Table.FromRecords({}))
```

`true`

Table.Join

3/15/2021 • 2 minutes to read

Syntax

```
Table.Join(table1 as table, key1 as any, table2 as table, key2 as any, optional joinKind as nullable number, optional joinAlgorithm as nullable number, optional keyEqualityComparers as nullable list) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`).

By default, an inner join is performed, however an optional `joinKind` may be included to specify the type of join. Options include:

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

An optional set of `keyEqualityComparers` may be included to specify how to compare the key columns. This feature is currently intended for internal use only.

Example 1

Inner join the two tables on [CustomerID]

```
Table.Join(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "CustomerID",
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25]
    }),
    "CustomerID"
)
```

CUSTOMERID	NAME	PHONE	ORDERID	ITEM	PRICE

1	Bob	123-4567	1	Fishing rod	100
1	Bob	123-4567	2	1 lb. worms	5
2	Jim	987-6543	3	Fishing net	25
3	Paul	543-7890	4	Fish tazer	200
3	Paul	543-7890	5	Bandaids	2
1	Bob	123-4567	6	Tackle box	20

Table.Keys

3/15/2021 • 2 minutes to read

Syntax

```
Table.Keys(table as table) as list
```

About

Table.Keys

Table.Last

3/15/2021 • 2 minutes to read

Syntax

```
Table.Last(table as table, optional default as any) as any
```

About

Returns the last row of the `table` or an optional default value, `default`, if the table is empty.

Example 1

Find the last row of the table.

```
Table.Last(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

CUSTOMERID	3
NAME	Paul
PHONE	543-7890

Example 2

Find the last row of the table `({})` or return [a = 0, b = 0] if empty.

```
Table.Last(Table.FromRecords({}), [a = 0, b = 0])
```

A	0
B	0

Table.LastN

3/15/2021 • 2 minutes to read

Syntax

```
Table.LastN(table as table, countOrCondition as any) as table
```

About

Returns the last row(s) from the table, `table`, depending on the value of `countOrCondition`:

- If `countOrCondition` is a number, that many rows will be returned starting from position (end - `countOrCondition`).
- If `countOrCondition` is a condition, the rows that meet the condition will be returned in ascending position until a row does not meet the condition.

Example 1

Find the last two rows of the table.

```
Table.LastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    2
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890

Example 2

Find the last rows where [a] > 0 in the table.

```
Table.LastN(
    Table.FromRecords({
        [a = -1, b = -2],
        [a = 3, b = 4],
        [a = 5, b = 6]
    }),
    each _ [a] > 0
```

A	B

3	4
5	6

Table.MatchesAllRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.MatchesAllRows(table as table, condition as function) as logical
```

About

Indicates whether all the rows in the `table` match the given `condition`. Returns `true` if all of the rows match, `false` otherwise.

Example 1

Determine whether all of the row values in column [a] are even in the table.

```
Table.MatchesAllRows(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    each Number.Mod([a], 2) = 0
)
```

`true`

Example 2

Find if all of the row values are [a = 1, b = 2], in the table `({[a = 1, b = 2], [a = 3, b = 4]})`.

```
Table.MatchesAllRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = -3, b = 4]
    }),
    each _ = [a = 1, b = 2]
)
```

`false`

Table.MatchesAnyRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.MatchesAnyRows(table as table, condition as function) as logical
```

About

Indicates whether any the rows in the `table` match the given `condition`. Returns `true` if any of the rows match, `false` otherwise.

Example 1

Determine whether any of the row values in column [a] are even in the table `({[a = 2, b = 4], [a = 6, b = 8]})`.

```
Table.MatchesAnyRows(
    Table.FromRecords({
        [a = 1, b = 4],
        [a = 3, b = 8]
    }),
    each Number.Mod([a], 2) = 0
)
```

`false`

Example 2

Determine whether any of the row values are [a = 1, b = 2], in the table `({[a = 1, b = 2], [a = 3, b = 4]})`.

```
Table.MatchesAnyRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = -3, b = 4]
    }),
    each _ = [a = 1, b = 2]
)
```

`true`

Table.Max

3/15/2021 • 2 minutes to read

Syntax

```
Table.Max(table as table, comparisonCriteria as any, optional default as any) as any
```

About

Returns the largest row in the `table`, given the `comparisonCriteria`. If the table is empty, the optional `default` value is returned.

Example 1

Find the row with the largest value in column [a] in the table `({[a = 2, b = 4], [a = 6, b = 8]})`.

```
Table.Max(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    "a"
)
```

A	6
B	8

Example 2

Find the row with the largest value in column [a] in the table `({})`. Return -1 if empty.

```
Table.Max(#table({"a"}, {}), "a", -1)
```

`-1`

Table.MaxN

3/15/2021 • 2 minutes to read

Syntax

```
Table.MaxN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

About

Returns the largest row(s) in the `table`, given the `comparisonCriteria`. After the rows are sorted, the `countOrCondition` parameter must be specified to further filter the result. Note the sorting algorithm cannot guarantee a fixed sorted result. The `countOrCondition` parameter can take multiple forms:

- If a number is specified, a list of up to `countOrCondition` items in ascending order is returned.
- If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.

Example 1

Find the row with the largest value in column [a] with the condition [a] > 0, in the table. The rows are sorted before the filter is applied.

```
Table.MaxN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 0, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [a] > 0
)
```

A	B
6	2
2	4

Example 2

Find the row with the largest value in column [a] with the condition [b] > 0, in the table. The rows are sorted before the filter is applied.

```
Table.MaxN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 8, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [b] > 0
)
```

Table.Min

3/15/2021 • 2 minutes to read

Syntax

```
Table.Min(table as table, comparisonCriteria as any, optional default as any) as any
```

About

Returns the smallest row in the `table`, given the `comparisonCriteria`. If the table is empty, the optional `default` value is returned.

Example 1

Find the row with the smallest value in column [a] in the table.

```
Table.Min(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    "a"
)
```

A	2
B	4

Example 2

Find the row with the smallest value in column [a] in the table. Return -1 if empty.

```
Table.Min(#table({"a"}, {}), "a", -1)
```

`-1`

Table.MinN

3/15/2021 • 2 minutes to read

Syntax

```
Table.MinN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

About

Returns the smallest row(s) in the `table`, given the `comparisonCriteria`. After the rows are sorted, the `countOrCondition` parameter must be specified to further filter the result. Note the sorting algorithm cannot guarantee a fixed sorted result. The `countOrCondition` parameter can take multiple forms:

- If a number is specified, a list of up to `countOrCondition` items in ascending order is returned.
- If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.

Example 1

Find the row with the smallest value in column [a] with the condition [a] < 3, in the table. The rows are sorted before the filter is applied.

```
Table.MinN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 0, b = 0],
        [a = 6, b = 4]
    }),
    "a",
    each [a] < 3
)
```

A	B
0	0
2	4

Example 2

Find the row with the smallest value in column [a] with the condition [b] < 0, in the table. The rows are sorted before the filter is applied.

```
Table.MinN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 8, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [b] < 0
)
```

Table.NestedJoin

3/15/2021 • 2 minutes to read

Syntax

```
Table.NestedJoin(table1 as table, key1 as any, table2 as any, key2 as any, newColumnName as
text, optional joinKind as nullable number, optional keyEqualityComparers as nullable list) as
table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are entered into the column named `newColumnName`.

The optional `joinKind` specifies the kind of join to perform. By default, a left outer join is performed if a `joinKind` is not specified.

An optional set of `keyEqualityComparers` may be included to specify how to compare the key columns. This feature is currently intended for internal use only.

Table.Partition

3/15/2021 • 2 minutes to read

Syntax

```
Table.Partition(table as table, column as text, groups as number, hash as function) as list
```

About

Partitions the `table` into a list of `groups` number of tables, based on the value of the `column` and a `hash` function. The `hash` function is applied to the value of the `column` row to obtain a hash value for the row. The hash value modulo `groups` determines in which of the returned tables the row will be placed.

- `table`: The table to partition.
- `column`: The column to hash to determine which returned table the row is in.
- `groups`: The number of tables the input table will be partitioned into.
- `hash`: The function applied to obtain a hash value.

Example

Partition the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` into 2 tables on column [a], using the value of the columns as the hash function.

```
Table.Partition(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    "a",
    2,
    each _
)
```

```
{
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 2, b = 4]
    }),
    Table.FromRecords({
        [a = 1, b = 4],
        [a = 1, b = 4]
    })
}
```

Table.PartitionValues

3/15/2021 • 2 minutes to read

Syntax

```
Table.Partition(table as table, column as text, groups as number, hash as function) as list
```

About

Partitions the `table` into a list of `groups` number of tables, based on the value of the `column` and a `hash` function. The `hash` function is applied to the value of the `column` row to obtain a hash value for the row. The hash value modulo `groups` determines in which of the returned tables the row will be placed.

- `table`: The table to partition.
- `column`: The column to hash to determine which returned table the row is in.
- `groups`: The number of tables the input table will be partitioned into.
- `hash`: The function applied to obtain a hash value.

Example 1

Partition the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` into 2 tables on column [a], using the value of the columns as the hash function.

```
Table.Partition(Table.FromRecords({[a = 2, b = 4], [a = 1, b = 4], [a = 2, b = 4], [a = 1, b = 4]}), "a", 2,
each _)
```

[Table]
[Table]

Table.Pivot

3/15/2021 • 2 minutes to read

Syntax

```
Table.Pivot(table as table, pivotValues as list, attributeColumn as text, valueColumn as text,
optional aggregationFunction as nullable function) as table
```

About

Given a pair of columns representing attribute-value pairs, rotates the data in the attribute column into a column headings.

Example 1

Take the values "a", "b", and "c" in the attribute column of table

```
({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "y",
attribute = "a", value = 2 ], [ key = "y", attribute = "b", value = 4 ] })
```

and pivot them into their own column.

```
Table.Pivot(
    Table.FromRecords({
        [key = "x", attribute = "a", value = 1],
        [key = "x", attribute = "c", value = 3],
        [key = "y", attribute = "a", value = 2],
        [key = "y", attribute = "b", value = 4]
    }),
    {"a", "b", "c"},
    "attribute",
    "value"
)
```

KEY	A	B	C
x	1		3
y	2	4	

Example 2

Take the values "a", "b", and "c" in the attribute column of table

```
({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "x",
attribute = "c", value = 5 ], [ key = "y", attribute = "a", value = 2 ], [ key = "y", attribute = "b", value
= 4 ] })
```

and pivot them into their own column. The attribute "c" for key "x" has multiple values associated with it, so use the function List.Max to resolve the conflict.

```
Table.Pivot(
    Table.FromRecords({
        [key = "x", attribute = "a", value = 1],
        [key = "x", attribute = "c", value = 3],
        [key = "x", attribute = "c", value = 5],
        [key = "y", attribute = "a", value = 2],
        [key = "y", attribute = "b", value = 4]
    }),
    {"a", "b", "c"},
    "attribute",
    "value",
    List.Max
)
```

KEY	A	B	C
x	1		5
y	2	4	

Table.PositionOf

3/15/2021 • 2 minutes to read

Syntax

```
Table.PositionOf(table as table, row as record, optional occurrence as any, optional
equationCriteria as any) as any
```

About

Returns the row position of the first occurrence of the `row` in the `table` specified. Returns -1 if no occurrence is found.

- `table`: The input table.
- `row`: The row in the table to find the position of.
- `occurrence`: *[Optional]* Specifies which occurrences of the row to return.
- `equationCriteria`: *[Optional]* Controls the comparison between the table rows.

Example 1

Find the position of the first occurrence of [a = 2, b = 4] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4]
)
```

`0`

Example 2

Find the position of the second occurrence of [a = 2, b = 4] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4],
    1
)
```

2

Example 3

Find the position of all the occurrences of [a = 2, b = 4] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4],
    Occurrence.All
)
```

0

2

Table.PositionOfAny

3/15/2021 • 2 minutes to read

Syntax

```
Table.PositionOfAny(table as table, rows as list, optional occurrence as nullable number,
optional equationCriteria as any) as any
```

About

Returns the row(s) position(s) from the `table` of the first occurrence of the list of `rows`. Returns -1 if no occurrence is found.

- `table`: The input table.
- `rows`: The list of rows in the table to find the positions of.
- `occurrence`: *[Optional]* Specifies which occurrences of the row to return.
- `equationCriteria`: *[Optional]* Controls the comparison between the table rows.

Example 1

Find the position of the first occurrence of [a = 2, b = 4] or [a = 6, b = 8] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

```
Table.PositionOfAny(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    {
        [a = 2, b = 4],
        [a = 6, b = 8]
    }
)
```

`0`

Example 2

Find the position of all the occurrences of [a = 2, b = 4] or [a = 6, b = 8] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

```
Table.PositionOfAny(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    {
        [a = 2, b = 4],
        [a = 6, b = 8]
    },
    Occurrence.All
)
```

0
1
2

Table.PrefixColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.PrefixColumns(table as table, prefix as text) as table
```

About

Returns a table where all the column names from the `table` provided are prefixed with the given text, `prefix`, plus a period in the form `prefix` `.ColumnName`.

Example 1

Prefix the columns with "MyTable" in the table.

```
Table.PrefixColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "MyTable"
)
```

MYTABLE.CUSTOMERID	MYTABLE.NAME	MYTABLE.PHONE
1	Bob	123-4567

Table.Profile

3/15/2021 • 2 minutes to read

Syntax

```
Table.Profile(table as table, optional additionalAggregates as nullable list) as table
```

About

Returns a profile for the columns in `table`.

The following information is returned for each column (when applicable):

- minimum
- maximum
- average
- standard deviation
- count
- null count
- distinct count

Table.PromoteHeaders

3/15/2021 • 2 minutes to read

Syntax

```
Table.PromoteHeaders(table as table, optional options as nullable record) as table
```

About

Promotes the first row of values as the new column headers (i.e. column names). By default, only text or number values are promoted to headers. Valid options:

`PromoteAllScalars` : If set to `true`, all the scalar values in the first row are promoted to headers using the `Culture`, if specified (or current document locale). For values that cannot be converted to text, a default column name will be used.
`Culture` : A culture name specifying the culture for the data.

Example 1

Promote the first row of values in the table.

```
Table.PromoteHeaders(
    Table.FromRecords({
        [Column1 = "CustomerID", Column2 = "Name", Column3 = #date(1980, 1, 1)],
        [Column1 = 1, Column2 = "Bob", Column3 = #date(1980, 1, 1)]
    })
)
```

CUSTOMERID	NAME	COLUMN3
1	Bob	1/1/1980 12:00:00 AM

Example 2

Promote all the scalars in the first row of the table to headers.

```
Table.PromoteHeaders(
    Table.FromRecords({
        [Rank = 1, Name = "Name", Date = #date(1980, 1, 1)],
        [Rank = 1, Name = "Bob", Date = #date(1980, 1, 1)]}
    ),
    [PromoteAllScalars = true, Culture = "en-US"]
)
```

1	NAME	1/1/1980
1	Bob	1/1/1980 12:00:00 AM

Table.Range

3/15/2021 • 2 minutes to read

Syntax

```
Table.Range(table as table, offset as number, optional count as nullable number) as table
```

About

Returns the rows from the `table` starting at the specified `offset`. An optional parameter, `count`, specifies how many rows to return. By default, all the rows after the offset are returned.

Example 1

Return all the rows starting at offset 1 in the table.

```
Table.Range(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

Example 2

Return one row starting at offset 1 in the table.

```
Table.Range(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1,
    1
)
```

CUSTOMERID	NAME	PHONE

2	Jim	987-6543

Table.RemoveColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveColumns(table as table, columns as any, optional missingField as nullable number) as table
```

About

Removes the specified `columns` from the `table` provided. If the column doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (eg. `MissingField.UseNull` or `MissingField.Ignore`).

Example 1

Remove column [Phone] from the table.

```
Table.RemoveColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "Phone"
)
```

CUSTOMERID	NAME
1	Bob

Example 2

Remove column [Address] from the table. Throws an error if it doesn't exist.

```
Table.RemoveColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "Address"
)
```

```
[Expression.Error] The field 'Address' of the record was not found.
```

Table.RemoveFirstN

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveFirstN(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the first specified number of rows, `countOrCondition`, of the table `table`. The number of rows removed depends on the optional parameter `countOrCondition`.

- If `countOrCondition` is omitted only the first row is removed.
- If `countOrCondition` is a number, that many rows (starting at the top) will be removed.
- If `countOrCondition` is a condition, the rows that meet the condition will be removed until a row does not meet the condition.

Example 1

Remove the first row of the table.

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

Example 2

Remove the first two rows of the table.

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

CUSTOMERID	NAME	PHONE
3	Paul	543-7890
4	Ringo	232-1550

Example 3

Remove the first rows where [CustomerID] <=2 of the table.

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] <= 2
)
```

CUSTOMERID	NAME	PHONE
3	Paul	543-7890
4	Ringo	232-1550

Table.RemoveLastN

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveLastN(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the last `countOrCondition` rows of the table `table`. The number of rows removed depends on the optional parameter `countOrCondition`.

- If `countOrCondition` is omitted only the last row is removed.
- If `countOrCondition` is a number, that many rows (starting at the bottom) will be removed.
- If `countOrCondition` is a condition, the rows that meet the condition will be removed until a row does not meet the condition.

Example 1

Remove the last row of the table.

```
Table.RemoveLastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

Example 2

Remove the last rows where [CustomerID] > 2 of the table.

```
Table.RemoveLastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] >= 2
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Table.RemoveMatchingRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveMatchingRows(table as table, rows as list, optional equationCriteria as any) as table
```

About

Removes all occurrences of the specified `rows` from the `table`. An optional parameter `equationCriteria` may be specified to control the comparison between the rows of the table.

Example 1

Remove any rows where [a = 1] from the table `({[a = 1, b = 2], [a = 3, b = 4], [a = 1, b = 6]})`.

```
Table.RemoveMatchingRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4],
        [a = 1, b = 6]
    }),
    {[a = 1]},
    "a"
)
```

A	B
3	4

Table.RemoveRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveRows(table as table, offset as number, optional count as nullable number) as table
```

About

Removes `count` of rows from the beginning of the `table`, starting at the `offset` specified. A default count of 1 is used if the `count` parameter isn't provided.

Example 1

Remove the first row from the table.

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    0
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

Example 2

Remove the row at position 1 from the table.

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	NAME	PHONE

1	Bob	123-4567
3	Paul	543-7890
4	Ringo	232-1550

Example 3

Remove two rows starting at position 1 from the table.

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1,
    2
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567
4	Ringo	232-1550

Table.RemoveRowsWithErrors

3/15/2021 • 2 minutes to read

Syntax

```
Table.RemoveRowsWithErrors(table as table, optional columns as nullable list) as table
```

About

Returns a table with the rows removed from the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.

Example 1

Remove error value from first row.

```
Table.RemoveRowsWithErrors(
    Table.FromRecords({
        [Column1 = ...],
        [Column1 = 2],
        [Column1 = 3]
    })
)
```

COLUMN1
2
3

Table.RenameColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.RenameColumns(table as table, renames as list, optional missingField as nullable number)
as table
```

About

Performs the given renames to the columns in table `table`. A replacement operation `renames` consists of a list of two values, the old column name and new column name, provided in a list. If the column doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (eg. `MissingField.UseNull` or `MissingField.Ignore`).

Example 1

Replace the column name "CustomerNum" with "CustomerID" in the table.

```
Table.RenameColumns(
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerNum", "CustomerID"}
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Example 2

Replace the column name "CustomerNum" with "CustomerID" and "PhoneNum" with "Phone" in the table.

```
Table.RenameColumns(
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", PhoneNum = "123-4567"]}),
    {
        {"CustomerNum", "CustomerID"},
        {"PhoneNum", "Phone"}
    }
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Example 3

Replace the column name "NewCol" with "NewColumn" in the table, and ignore if the column doesn't exist.

```
Table.RenameColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"NewCol", "NewColumn"},
    MissingField.Ignore
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Table.ReorderColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReorderColumns(table as table, columnOrder as list, optional missingField as nullable
number) as table
```

About

Returns a table from the input `table`, with the columns in the order specified by `columnOrder`. Columns that are not specified in the list will not be reordered. If the column doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (eg. `MissingField.UseNull` or `MissingField.Ignore`).

Example 1

Switch the order of the columns [Phone] and [Name] in the table.

```
Table.ReorderColumns(
    Table.FromRecords({[CustomerID = 1, Phone = "123-4567", Name = "Bob"]}),
    {"Name", "Phone"}
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Example 2

Switch the order of the columns [Phone] and [Address] or use "MissingField.Ignore" in the table. It doesn't change the table because column [Address] doesn't exist.

```
Table.ReorderColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"Phone", "Address"},
    MissingField.Ignore
)
```

CUSTOMERID	NAME	PHONE
1	Bob	123-4567

Table.Repeat

3/15/2021 • 2 minutes to read

Syntax

```
Table.Repeat(table as table, count as number) as table
```

About

Returns a table with the rows from the input `table` repeated the specified `count` times.

Example 1

Repeat the rows in the table two times.

```
Table.Repeat(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "world"]
    }),
    2
)
```

A	B
1	hello
3	world
1	hello
3	world

Table.ReplaceErrorValues

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceErrorValues(table as table, errorReplacement as list) as table
```

About

Replaces the error values in the specified columns of the `table` with the new values in the `errorReplacement` list. The format of the list is {{column1, value1}, ...}. There may only be one replacement value per column, specifying the column more than once will result in an error.

Example 1

Replace the error value with the text "world" in the table.

```
Table.ReplaceErrorValues(
    Table.FromRows({{1, "hello"}, {3, ...}}, {"A", "B"}),
    {"B", "world"}
)
```

A	B
1	hello
3	world

Example 2

Replace the error value in column A with the text "hello" and in column B with the text "world" in the table.

```
Table.ReplaceErrorValues(
    Table.FromRows({{..., ...}, {1, 2}}, {"A", "B"}),
    {{"A", "hello"}, {"B", "world"}}
)
```

A	B
hello	world
1	2

Table.ReplaceKeys

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceKeys(table as table, keys as list) as table
```

About

Table.ReplaceKeys

Table.ReplaceMatchingRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceMatchingRows(table as table, replacements as list, optional equationCriteria as
any) as table
```

About

Replaces all the specified rows in the `table` with the provided ones. The rows to replace and the replacements are specified in `replacements`, using {old, new} formatting. An optional `equationCriteria` parameter may be specified to control comparison between the rows of the table.

Example 1

Replace the rows [a = 1, b = 2] and [a = 2, b = 3] with [a = -1, b = -2],[a = -2, b = -3] in the table.

```
Table.ReplaceMatchingRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 2, b = 3],
        [a = 3, b = 4],
        [a = 1, b = 2]
    }),
    {
        {[a = 1, b = 2], [a = -1, b = -2]},
        {[a = 2, b = 3], [a = -2, b = -3]}
    }
)
```

A	B
-1	-2
-2	-3
3	4
-1	-2

Table.ReplaceRelationshipIdentity

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceRelationshipIdentity(value as any, identity as text) as any
```

About

Table.ReplaceRelationshipIdentity

Table.ReplaceRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceRows(table as table, offset as number, count as number, rows as list) as table
```

About

Replaces a specified number of rows, `count`, in the input `table` with the specified `rows`, beginning after the `offset`. The `rows` parameter is a list of records.

- `table`: The table where the replacement is performed.
- `offset`: The number of rows to skip before making the replacement.
- `count`: The number of rows to replace.
- `rows`: The list of row records to insert into the `table` at the location specified by the `offset`.

Example 1

Starting at position 1, replace 3 rows.

```
Table.ReplaceRows(
    Table.FromRecords({
        [Column1 = 1],
        [Column1 = 2],
        [Column1 = 3],
        [Column1 = 4],
        [Column1 = 5]
    }),
    1,
    3,
    {[Column1 = 6], [Column1 = 7]}
)
```

COLUMN1
1
6
7
5

Table.ReplaceValue

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReplaceValue(table as table, oldValue as any, newValue as any, replacer as function,
columnsToSearch as list) as table
```

About

Replaces `oldValue` with `newValue` in the specified columns of the `table`.

Example 1

Replace the text "goodbye" with the text "world" in the table.

```
Table.ReplaceValue(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "goodbye"]
    }),
    "goodbye",
    "world",
    Replacer.ReplaceText,
    {"b"}
)
```

A	B
1	hello
3	world

Example 2

Replace the text "ur" with the text "or" in the table.

```
Table.ReplaceValue(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "wurld"]
    }),
    "ur",
    "or",
    Replacer.ReplaceText,
    {"b"}
)
```

A	B
1	hello

3	world

Table.ReverseRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.ReverseRows(table as table) as table
```

About

Returns a table with the rows from the input `table` in reverse order.

Example 1

Reverse the rows in the table.

```
Table.ReverseRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

CUSTOMERID	NAME	PHONE
4	Ringo	232-1550
3	Paul	543-7890
2	Jim	987-6543
1	Bob	123-4567

Table.RowCount

3/15/2021 • 2 minutes to read

Syntax

```
Table.RowCount(table as table) as number
```

About

Returns the number of rows in the `table`.

Example 1

Find the number of rows in the table.

```
Table.RowCount(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

```
3
```

Table.Schema

3/15/2021 • 2 minutes to read

Syntax

```
Table.Schema(table as table) as table
```

About

Returns a table describing the columns of `table`.

Each row in the table describes the properties of a column of `table`:

Column Name	Description
`Name`	The name of the column.
`Position`	The 0-based position of the column in `table`.
`TypeName`	The name of the type of the column.
`Kind`	The kind of the type of the column.
`IsNullable`	Whether the column can contain `null` values.
`NumericPrecisionBase`	The numeric base (e.g. base-2, base-10) of the `NumericPrecision` and `NumericScale` fields.
`NumericPrecision`	The precision of a numeric column in the base specified by `NumericPrecisionBase`. This is the maximum number of digits that can be represented by a value of this type (including fractional digits).
`NumericScale`	The scale of a numeric column in the base specified by `NumericPrecisionBase`. This is the number of digits in the fractional part of a value of this type. A value of `0` indicates a fixed scale with no fractional digits. A value of `null` indicates the scale is not known (either because it is floating or not defined).
`DateTimePrecision`	The maximum number of fractional digits supported in the seconds portion of a date or time value.
`MaxLength`	The maximum number of characters permitted in a `text` column, or the maximum number of bytes permitted in a `binary` column.
`IsVariableLength`	Indicates whether this column can vary in length (up to `MaxLength`) or if it is of fixed size.

`NativeTypeName`	The name of the type of the column in the native type system of the source (e.g. `nvarchar` for SQL Server).
`NativeDefaultExpression`	The default expression for a value of this column in the native expression language of the source (e.g. `42` or `newid()` for SQL Server).
`Description`	The description of the column.

Table.SelectColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.SelectColumns(table as table, columns as any, optional missingField as nullable number) as table
```

About

Returns the `table` with only the specified `columns`.

- `table`: The provided table.
- `columns`: The list of columns from the table `table` to return. Columns in the returned table are in the order listed in `columns`.
- `missingField`: *(Optional)* What to do if the columnn does not exist. Example: `MissingField.UseNull` or `MissingField.Ignore`.

Example 1

Only include column [Name].

```
Table.SelectColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

NAME
Bob
Jim
Paul
Ringo

Example 2

Only include columns [CustomerID] and [Name].

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerID", "Name"}
)
```

CUSTOMERID	NAME
1	Bob

Example 3

If the included column does not exit, the default result is an error.

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "NewColumn"
)
```

```
[Expression.Error] The field 'NewColumn' of the record wasn't found.
```

Example 4

If the included column does not exit, option `MissingField.UseNull` creates a column of null values.

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerID", "NewColumn"},
    MissingField.UseNull
)
```

CUSTOMERID	NEWCOLUMN
1	

Table.SelectRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.SelectRows(table as table, condition as function) as table
```

About

Returns a table of rows from the `table`, that matches the selection `condition`.

Example 1

Select the rows in the table where the values in [CustomerID] column are greater than 2.

```
Table.SelectRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] > 2
)
```

CUSTOMERID	NAME	PHONE
3	Paul	543-7890
4	Ringo	232-1550

Example 2

Select the rows in the table where the names do not contain a "B".

```
Table.SelectRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each not Text.Contains([Name], "B")
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890

4	Ringo	232-1550

Table.SelectRowsWithErrors

3/15/2021 • 2 minutes to read

Syntax

```
Table.SelectRowsWithErrors(table as table, optional columns as nullable list) as table
```

About

Returns a table with only those rows of the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.

Example 1

Select names of customers with errors in their rows.

```
Table.SelectRowsWithErrors(
    Table.FromRecords({
        [CustomerID = ..., Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)[Name]
```

Bob

Table.SingleRow

3/15/2021 • 2 minutes to read

Syntax

```
Table.SingleRow(table as table) as record
```

About

Returns the single row in the one row `table`. If the `table` has more than one row, an exception is thrown.

Example 1

Return the single row in the table.

```
Table.SingleRow(Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}))
```

CUSTOMERID	1
NAME	Bob
PHONE	123-4567

Table.Skip

3/15/2021 • 2 minutes to read

Syntax

```
Table.Skip(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the first specified number of rows, `countOrCondition`, of the table `table`. The number of rows skipped depends on the optional parameter `countOrCondition`.

- If `countOrCondition` is omitted only the first row is skipped.
- If `countOrCondition` is a number, that many rows (starting at the top) will be skipped.
- If `countOrCondition` is a condition, the rows that meet the condition will be skipped until a row does not meet the condition.

Example 1

Skip the first row of the table.

```
Table.Skip(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	NAME	PHONE
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

Example 2

Skip the first two rows of the table.

```
Table.Skip(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

CUSTOMERID	NAME	PHONE
3	Paul	543-7890
4	Ringo	232-1550

Example 3

Skip the first rows where [Price] > 25 of the table.

```
Table.Skip(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    each [Price] > 25
)
```

ORDERID	CUSTOMERID	ITEM	PRICE
2	1	1 lb. worms	5
3	2	Fishing net	25
4	3	Fish tazer	200
5	3	Bandaids	2
6	1	Tackle box	20
7	5	Bait	3.25
8	5	Fishing Rod	100
9	6	Bait	3.25

Table.Sort

3/15/2021 • 2 minutes to read

Syntax

```
Table.Sort(table as table, comparisonCriteria as any) as table
```

About

Sorts the `table` using the list of one or more column names and optional `comparisonCriteria` in the form { { col1, comparisonCriteria }, {col2} }.

Example 1

Sort the table on column "OrderID".

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {"OrderID"}
)
```

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	Fishing rod	100
2	1	1 lb. worms	5
3	2	Fishing net	25
4	3	Fish tazer	200
5	3	Bandaids	2
6	1	Tackle box	20
7	5	Bait	3.25
8	5	Fishing Rod	100
9	6	Bait	3.25

Example 2

Sort the table on column "OrderID" in descending order.

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {"OrderID", Order.Descending}
)
```

ORDERID	CUSTOMERID	ITEM	PRICE
9	6	Bait	3.25
8	5	Fishing Rod	100
7	5	Bait	3.25
6	1	Tackle box	20
5	3	Bandaids	2
4	3	Fish tazer	200
3	2	Fishing net	25
2	1	1 lb. worms	5
1	1	Fishing rod	100

Example 3

Sort the table on column "CustomerID" then "OrderID", with "CustomerID" being in ascending order.

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {
        {"CustomerID", Order.Ascending},
        "OrderID"
    }
)
```

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	Fishing rod	100
2	1	1 lb. worms	5
6	1	Tackle box	20
3	2	Fishing net	25
4	3	Fish tazer	200
5	3	Bandaids	2
7	5	Bait	3.25
8	5	Fishing Rod	100
9	6	Bait	3.25

Table.Split

3/15/2021 • 2 minutes to read

Syntax

```
Table.Split(table as table, pageSize as number) as list
```

About

Splits `table` into a list of tables where the first element of the list is a table containing the first `pageSize` rows from the source table, the next element of the list is a table containing the next `pageSize` rows from the source table, etc.

Example 1

Split a table of five records into tables with two records each.

```
let
    Customers = Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"],
        [CustomerID = 5, Name = "Anita", Phone = "530-1459"]
    })
in
    Table.Split(Customers, 2)
```

[Table]
[Table]
[Table]

Table.SplitAt

6/14/2021 • 2 minutes to read

```
Table.SplitAt(table as table, count as number) as list
```

About

Returns a list containing two tables: a table with the first N rows of `table` (as specified by `count`) and a table containing the remaining rows of `table`. If the tables of the resulting list are enumerated exactly once and in order, the function will enumerate `table` only once.

Example 1

Return the first two rows of the table and the remaining rows of the table.

```
Table.SplitAt(#table({"a", "b", "c"}, {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}), 2)</code>
```

```
{
    #table({"a", "b", "c"}, {{1, 2, 3}, {4, 5, 6}}),
    #table({"a", "b", "c"}, {{7, 8, 9}})
}
```

Table.SplitColumn

3/15/2021 • 2 minutes to read

Syntax

```
Table.SplitColumn(table as table, sourceColumn as text, splitter as function, optional
columnNamesOrNumber as any, optional default as any, optional extraColumns as any) as table
```

About

Splits the specified columns into a set of additional columns using the specified splitter function.

Example 1

Split the [Name] column at position of "i" into two columns

```
let
    Customers = Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"]
    })
in
    Table.SplitColumn(Customers, "Name", Splitter.SplitTextByDelimiter("i"), 2
```

CUSTOMERID	NAME.1	NAME.2	PHONE
1	Bob		123-4567
2	J	m	987-6543
3	Paul		543-7890
4	Cr	st	232-1550

Table.ToColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.ToColumns(table as table) as list
```

About

Creates a list of nested lists from the table, `table`. Each list item is an inner list that contains the column values.

Example

Create a list of the column values from the table.

```
Table.ToColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    })
)
```

[List]
[List]
[List]

Table.ToList

3/15/2021 • 2 minutes to read

Syntax

```
Table.ToList(table as table, optional combiner as nullable function) as list
```

About

Converts a table into a list by applying the specified combining function to each row of values in the table.

Example 1

Combine the text of each row with a comma.

```
Table.ToList(
    Table.FromRows({
        {Number.ToText(1), "Bob", "123-4567"},
        {Number.ToText(2), "Jim", "987-6543"},
        {Number.ToText(3), "Paul", "543-7890"}
    }),
    Combiner.CombineTextByDelimiter(",")
)
```

1,Bob,123-4567
2,Jim,987-6543
3,Paul,543-7890

Table.ToRecords

3/15/2021 • 2 minutes to read

Syntax

```
Table.ToRecords(table as table) as list
```

About

Converts a table, `table`, to a list of records.

Example

Convert the table to a list of records.

```
Table.ToRecords(
    Table.FromRows(
        {
            {1, "Bob", "123-4567"},
            {2, "Jim", "987-6543"},
            {3, "Paul", "543-7890"}
        },
        {"CustomerID", "Name", "Phone"}
    )
)
```

[Record]
[Record]
[Record]

Table.ToRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.ToRows(table as table) as list
```

About

Creates a list of nested lists from the table, `table`. Each list item is an inner list that contains the row values.

Example

Create a list of the row values from the table.

```
Table.ToRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

[List]
[List]
[List]

Table.TransformColumnNames

3/15/2021 • 2 minutes to read

Syntax

```
Table.TransformColumnNames(table as table, nameGenerator as function, optional options as
nullable record) as table
```

About

Transforms column names by using the given `nameGenerator` function. Valid options:

`MaxLength` specifies the maximum length of new column names. If the given function results with a longer column name, the long name will be trimmed.
`Comparer` is used to control the comparison while generating new column names. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Remove the `#(tab)` character from column names

```
Table.TransformColumnNames(Table.FromRecords({[#"Col#(tab)umn" = 1]}), Text.Clean)
```

COLUMN
1

Example 2

Transform column names to generate case-insensitive names of length 6.

```
Table.TransformColumnNames(
    Table.FromRecords({[ColumnNum = 1, cOlumnnum = 2, coLumnNUM = 3]}),
    Text.Clean,
    [MaxLength = 6, Comparer = Comparer.OrdinalIgnoreCase]
)
```

COLUMN	COLUM1	COLUM2
1	2	3

Table.TransformColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.TransformColumns(table as table, transformOperations as list, optional
defaultTransformation as nullable function, optional missingField as nullable number) as table
```

About

Returns a table from the input `table` by applying the transform operation to the column specified in the parameter `transformOperations` (where format is { column name, transformation }). If the column doesn't exist, an exception is thrown unless the optional parameter `defaultTransformation` specifies an alternative (eg. `MissingField.UseNull` or `MissingField.Ignore`).

Example 1

Transform the number values in column [A] to number values.

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"A", Number.FromText}
)
```

A	B
1	2
5	10

Example 2

Transform the number values in missing column [X] to text values, ignoring columns which don't exist.

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText},
    null,
    MissingField.Ignore
)
```

A	B
1	2

5	10

Example 3

Transform the number values in missing column [X] to text values, defaulting to null on columns which don't exist.

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText},
    null,
    MissingField.UseNull
)
```

A	B	X
1	2	
5	10	

Example 4

Transform the number values in missing column [X] to text values, giving an error on columns which don't exist.

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText}
)
```

```
[Expression.Error] The column 'X' of the table wasn't found.
```

Table.TransformColumnTypes

3/15/2021 • 2 minutes to read

Syntax

```
Table.TransformColumnTypes(table as table, typeTransformations as list, optional culture as nullable text) as table
```

About

Returns a table from the input `table` by applying the transform operation to the columns specified in the parameter `typeTransformations` (where format is { column name, type name}), using the specified culture in the optional parameter `culture` (for example, "en-US"). If the column doesn't exist, an exception is thrown.

Example 1

Transform the number values in column [a] to text values from the table `({[a = 1, b = 2], [a = 3, b = 4]})`.

```
Table.TransformColumnTypes(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {"a", type text},
    "en-US"
)
```

A	B
1	2
3	4

Table.TransformRows

3/15/2021 • 2 minutes to read

Syntax

```
Table.TransformRows(table as table, transform as function) as list
```

About

Creates a table from `table` by applying the `transform` operation to the rows. If the return type of the `transform` function is specified, then the result will be a table with that row type. In all other cases, the result of this function will be a list with an item type of the return type of the transform function.

Example 1

Transform the rows into a list of numbers from the table `({[A = 1], [A = 2], [A = 3], [A = 4], [A = 5]})`.

```
Table.TransformRows(
    Table.FromRecords({
        [a = 1],
        [a = 2],
        [a = 3],
        [a = 4],
        [a = 5]
    }),
    each [a]
)
```

1
2
3
4
5

Example 2

Transform the rows in column [A] into text values in a column [B] from the table `({[A = 1], [A = 2], [A = 3], [A = 4], [A = 5])`.

```
Table.TransformRows(
    Table.FromRecords({
        [a = 1],
        [a = 2],
        [a = 3],
        [a = 4],
        [a = 5]
    }),
    (row) as record => [B = Number.ToText(row[a])]
)
```

[Record]
[Record]
[Record]
[Record]
[Record]

Table.Transpose

3/15/2021 • 2 minutes to read

Syntax

```
Table.Transpose(table as table, optional columns as any) as table
```

About

Makes columns into rows and rows into columns.

Example 1

Make the rows of the table of name-value pairs into columns.

```
Table.Transpose(
    Table.FromRecords({
        [Name = "Full Name", Value = "Fred"],
        [Name = "Age", Value = 42],
        [Name = "Country", Value = "UK"]
    })
)
```

COLUMN1	COLUMN2	COLUMN3
Full Name	Age	Country
Fred	42	UK

Table.Unpivot

3/15/2021 • 2 minutes to read

Syntax

```
Table.Unpivot(table as table, pivotColumns as list, attributeColumn as text, valueColumn as
text) as table
```

About

Translates a set of columns in a table into attribute-value pairs, combined with the rest of the values in each row.

Example 1

Take the columns "a", "b", and "c" in the table `({[ key = "x", a = 1, b = null, c = 3 ], [ key = "y", a = 2, b = 4, c = null ]})` and unpivot them into attribute-value pairs.

```
Table.Unpivot(
    Table.FromRecords({
        [key = "x", a = 1, b = null, c = 3],
        [key = "y", a = 2, b = 4, c = null]
    }),
    {"a", "b", "c"},
    "attribute",
    "value"
)
```

KEY	ATTRIBUTE	VALUE
x	a	1
x	c	3
y	a	2
y	b	4

Table.UnpivotOtherColumns

3/15/2021 • 2 minutes to read

Syntax

```
Table.UnpivotOtherColumns(table as table, pivotColumns as list, attributeColumn as text,
valueColumn as text) as table
```

About

Translates all columns other than a specified set into attribute-value pairs, combined with the rest of the values in each row.

Example 1

Translates all columns other than a specified set into attribute-value pairs, combined with the rest of the values in each row.

```
Table.UnpivotOtherColumns(
    Table.FromRecords({
        [key = "key1", attribute1 = 1, attribute2 = 2, attribute3 = 3],
        [key = "key2", attribute1 = 4, attribute2 = 5, attribute3 = 6]
    }),
    {"key"},
    "column1",
    "column2"
)
```

KEY	COLUMN1	COLUMN2
key1	attribute1	1
key1	attribute2	2
key1	attribute3	3
key2	attribute1	4
key2	attribute2	5
key2	attribute3	6

Table.View

3/15/2021 • 2 minutes to read

Syntax

```
Table.View(table as nullable table, handlers as record) as table
```

About

Returns a view of `table` where the functions specified in `handlers` are used in lieu of the default behavior of an operation when the operation is applied to the view. Handler functions are optional. If a handler function is not specified for an operation, the default behavior of the operation is applied to `table` instead (except in the case of `GetExpression`).

Handler functions must return a value that is semantically equivalent to the result of applying the operation against `table` (or the resulting view in the case of `GetExpression`).

If a handler function raises an error, the default behavior of the operation is applied to the view.

`Table.View` can be used to implement folding to a data source – the translation of M queries into source-specific queries (e.g. to create T-SQL statements from M queries).

Please see the published documentation for a more complete description of `Table.View`.

Table.ViewFunction

3/15/2021 • 2 minutes to read

Syntax

```
Table.ViewFunction(function as function) as function
```

About

Creates a view function based on `function` that can be handled in a view created by `Table.View`.

The `OnInvoke` handler of `Table.View` can be used to defined a handler for the view function.

As with the handlers for built-in operations, if no `OnInvoke` handler is specified, or if it does not handle the view function, or if an error is raised by the handler, `function` is applied on top of the view.

Please see the published documentation for a more complete description of `Table.View` and custom view functions.

Tables.GetRelationships

3/15/2021 • 2 minutes to read

Syntax

```
Tables.GetRelationships(tables as table, optional dataColumn as nullable text) as table
```

About

Gets the relationships among a set of tables. The set `tables` is assumed to have a structure similar to that of a navigation table. The column defined by `dataColumn` contains the actual data tables.

#table

3/15/2021 • 2 minutes to read

Syntax

```
#table(columns as any, rows as any) as any
```

About

Creates a table value from columns `columns` and the list `rows` where each element of the list is an inner list that contains the column values for a single row. `columns` may be a list of column names, a table type, a number of columns, or null.

Text functions

3/15/2021 • 4 minutes to read

These functions create and manipulate text values.

Text

Information

FUNCTION	DESCRIPTION
Text.InferNumberType	Infers granular number type (Int64.Type, Double.Type, etc.) of `text` using `culture`.
Text.Length	Returns the number of characters in a text value.

Text Comparisons

FUNCTION	DESCRIPTION
Character.FromNumber	Returns a number to its character value.
Character.ToNumber	Returns a character to its number value.
Guid.From	Returns a `Guid.Type` value from the given `value`.
Json.FromValue	Produces a JSON representation of a given value.
Text.From	Returns the text representation of a number, date, time, datetime, datetimezone, logical, duration or binary value. If a value is null, Text.From returns null. The optional culture parameter is used to format the text value according to the given culture.
Text.FromBinary	Decodes data from a binary value in to a text value using an encoding.
Text.NewGuid	Returns a Guid value as a text value.
Text.ToBinary	Encodes a text value into binary value using an encoding.
Text.ToList	Returns a list of characters from a text value.
Value.FromText	Decodes a value from a textual representation, value, and interprets it as a value with an appropriate type. Value.FromText takes a text value and returns a number, a logical value, a null value, a DateTime value, a Duration value, or a text value. The empty text value is interpreted as a null value.

Extraction

FUNCTION	DESCRIPTION
Text.At	Returns a character starting at a zero-based offset.
Text.Middle	Returns the substring up to a specific length.
Text.Range	Returns a number of characters from a text value starting at a zero-based offset and for count number of characters.
Text.Start	Returns the count of characters from the start of a text value.

FUNCTION	DESCRIPTION
Text.End	Returns the number of characters from the end of a text value.

Modification

FUNCTION	DESCRIPTION
Text.Insert	Returns a text value with newValue inserted into a text value starting at a zero-based offset.
Text.Remove	Removes all occurrences of a character or list of characters from a text value. The removeChars parameter can be a character value or a list of character values.
Text.RemoveRange	Removes count characters at a zero-based offset from a text value.
Text.Replace	Replaces all occurrences of a substring with a new text value.
Text.ReplaceRange	Replaces length characters in a text value starting at a zero-based offset with the new text value.
Text.Select	Selects all occurrences of the given character or list of characters from the input text value.

Membership

FUNCTION	DESCRIPTION
Text.Contains	Returns true if a text value substring was found within a text value string; otherwise, false.
Text.EndsWith	Returns a logical value indicating whether a text value substring was found at the end of a string.
Text.PositionOf	Returns the first occurrence of substring in a string and returns its position starting at startOffset.
Text.PositionOfAny	Returns the first occurrence of a text value in list and returns its position starting at startOffset.

FUNCTION	DESCRIPTION
Text.StartsWith	Returns a logical value indicating whether a text value substring was found at the beginning of a string.

Transformations

FUNCTION	DESCRIPTION
Text.AfterDelimiter	Returns the portion of text after the specified delimiter.
Text.BeforeDelimiter	Returns the portion of text before the specified delimiter.
Text.BetweenDelimiters	Returns the portion of text between the specified startDelimiter and endDelimiter.
Text.Clean	Returns the original text value with non-printable characters removed.
Text.Combine	Returns a text value that is the result of joining all text values with each value separated by a separator.
Text.Lower	Returns the lowercase of a text value.
Text.PadEnd	Returns a text value padded at the end with pad to make it at least length characters.
Text.PadStart	Returns a text value padded at the beginning with pad to make it at least length characters. If pad is not specified, whitespace is used as pad.
Text.Proper	Returns a text value with first letters of all words converted to uppercase.
Text.Repeat	Returns a text value composed of the input text value repeated a number of times.
Text.Reverse	Reverses the provided text.
Text.Split	Returns a list containing parts of a text value that are delimited by a separator text value.
Text.SplitAny	Returns a list containing parts of a text value that are delimited by any separator text values.
Text.Trim	Removes any occurrences of characters in trimChars from text.
Text.TrimEnd	Removes any occurrences of the characters specified in trimChars from the end of the original text value.
Text.TrimStart	Removes any occurrences of the characters in trimChars from the start of the original text value.
Text.Upper	Returns the uppercase of a text value.

Parameters

PARAMETER VALUES	DESCRIPTION
Occurrence.All	A list of positions of all occurrences of the found values is returned.
Occurrence.First	The position of the first occurrence of the found value is returned.
Occurrence.Last	The position of the last occurrence of the found value is returned.
RelativePosition.FromEnd	Indicates indexing should be done from the end of the input.
RelativePosition.FromStart	Indicates indexing should be done from the start of the input.
TextEncoding.Ascii	Use to choose the ASCII binary form.
TextEncoding.BigEndianUnicode	Use to choose the UTF16 big endian binary form.
TextEncoding.Unicode	Use to choose the UTF16 little endian binary form.
TextEncoding.Utf8	Use to choose the UTF8 binary form.
TextEncoding.Utf16	Use to choose the UTF16 little endian binary form.
TextEncoding.Windows	Use to choose the Windows binary form.

Character.FromNumber

3/15/2021 • 2 minutes to read

Syntax

```
Character.FromNumber(number as nullable number) as nullable text
```

About

Returns the character equivalent of the number.

Example 1

Given the number 9, find the character value.

```
Character.FromNumber(9)
```

```
"#(tab)"
```

Character.ToNumber

3/15/2021 • 2 minutes to read

Syntax

```
Character.ToNumber(character as nullable text) as nullable number
```

About

Returns the number equivalent of the character, `character`.

Example 1

Given the character "#(tab)" 9, find the number value.

```
Character.ToNumber("#(tab)")
```

```
9
```

Guid.From

3/15/2021 • 2 minutes to read

Syntax

```
Guid.From(value as nullable text) as nullable text
```

About

Returns a `Guid.Type` value from the given `value`. If the given `value` is `null`, `Guid.From` returns `null`. A check will be performed to see if the given `value` is in an acceptable format. Acceptable formats provided in the examples.

Example 1

The Guid can be provided as 32 contiguous hexadecimal digits.

```
Guid.From("05FE1DADC8C24F3BA4C2D194116B4967")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

Example 2

The Guid can be provided as 32 hexadecimal digits separated by hyphens into blocks of 8-4-4-4-12.

```
Guid.From("05FE1DAD-C8C2-4F3B-A4C2-D194116B4967")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

Example 3

The Guid can be provided as 32 hexadecimal digits separated by hyphens and enclosed in braces.

```
Guid.From("{05FE1DAD-C8C2-4F3B-A4C2-D194116B4967}")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

Example 4

The Guid can be provided as 32 hexadecimal digits separated by hyphens and enclosed by parentheses.

```
Guid.From("(05FE1DAD-C8C2-4F3B-A4C2-D194116B4967)")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

Json.FromValue

3/15/2021 • 2 minutes to read

Syntax

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

About

Produces a JSON representation of a given value `value` with a text encoding specified by `encoding`. If `encoding` is omitted, UTF8 is used. Values are represented as follows:

- Null, text and logical values are represented as the corresponding JSON types
- Numbers are represented as numbers in JSON, except that `#infinity`, `-#infinity` and `#nan` are converted to null
- Lists are represented as JSON arrays
- Records are represnted as JSON objects
- Tables are represented as an array of objects
- Dates, times, datetimes, datetimezones and durations are represented as ISO-8601 text
- Binary values are represented as base-64 encoded text
- Types and functions produce an error

Example 1

Convert a complex value to JSON.

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

```
"{""A"":[1,true,""3""],""B"":""2012-03-25""}"
```

RelativePosition.FromEnd

3/15/2021 • 2 minutes to read

About

Indicates indexing should be done from the end of the input.

RelativePosition.FromStart

3/15/2021 • 2 minutes to read

About

Indicates indexing should be done from the start of the input.

Text.AfterDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Text.AfterDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

About

Returns the portion of `text` after the specified `delimiter`. An optional numeric `index` indicates which occurrence of the `delimiter` should be considered. An optional list `index` indicates which occurrence of the `delimiter` should be considered, as well as whether indexing should be done from the start or end of the input.

Example 1

Get the portion of "111-222-333" after the (first) hyphen.

```
Text.AfterDelimiter("111-222-333", "-")
```

`"222-333"`

Example 2

Get the portion of "111-222-333" after the second hyphen.

```
Text.AfterDelimiter("111-222-333", "-", 1)
```

`"333"`

Example 3

Get the portion of "111-222-333" after the second hyphen from the end.

```
Text.AfterDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

`"222-333"`

Text.At

3/15/2021 • 2 minutes to read

Syntax

```
Text.At(text as nullable text, index as number) as nullable text
```

About

Returns the character in the text value, `text` at position `index`. The first character in the text is at position 0.

Example 1

Find the character at position 4 in string "Hello, World".

```
Text.At("Hello, World", 4)
```

`"o"`

Text.BeforeDelimiter

3/15/2021 • 2 minutes to read

Syntax

```
Text.BeforeDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

About

Returns the portion of `text` before the specified `delimiter`. An optional numeric `index` indicates which occurrence of the `delimiter` should be considered. An optional list `index` indicates which occurrence of the `delimiter` should be considered, as well as whether indexing should be done from the start or end of the input.

Example 1

Get the portion of "111-222-333" before the (first) hyphen.

```
Text.BeforeDelimiter("111-222-333", "-")
```

`"111"`

Example 2

Get the portion of "111-222-333" before the second hyphen.

```
Text.BeforeDelimiter("111-222-333", "-", 1)
```

`"111-222"`

Example 3

Get the portion of "111-222-333" before the second hyphen from the end.

```
Text.BeforeDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

`"111"`

Text.BetweenDelimiters

3/15/2021 • 2 minutes to read

Syntax

```
Text.BetweenDelimiters(text as nullable text, startDelimiter as text, endDelimiter as text,
optional startIndex as any, optional endIndex as any) as any
```

About

Returns the portion of `text` between the specified `startDelimiter` and `endDelimiter`. An optional numeric `startIndex` indicates which occurrence of the `startDelimiter` should be considered. An optional list `startIndex` indicates which occurrence of the `startDelimiter` should be considered, as well as whether indexing should be done from the start or end of the input. The `endIndex` is similar, except that indexing is done relative to the `startIndex`.

Example 1

Get the portion of "111 (222) 333 (444)" between the (first) open parenthesis and the (first) closed parenthesis that follows it.

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")")
```

`"222"`

Example 2

Get the portion of "111 (222) 333 (444)" between the second open parenthesis and the first closed parenthesis that follows it.

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", 1, 0)
```

`"444"`

Example 3

Get the portion of "111 (222) 333 (444)" between the second open parenthesis from the end and the second closed parenthesis that follows it.

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", {1, RelativePosition.FromEnd}, {1,
RelativePosition.FromStart})
```

`"222) 333 (444"`

Text.Clean

3/15/2021 • 2 minutes to read

Syntax

```
Text.Clean(text as nullable text) as nullable text
```

About

Returns a text value with all control characters of `text` removed.

Example 1

Remove line feeds and other control characters from a text value.

```
Text.Clean("ABC#(lf)D")
```

`"ABCD"`

Text.Combine

3/15/2021 • 2 minutes to read

Syntax

```
Text.Combine(texts as list, optional separator as nullable text) as text
```

About

Returns the result of combining the list of text values, `texts`, into a single text value. An optional separator used in the final combined text may be specified, `separator`.

Example 1

Combine text values "Seattle" and "WA".

```
Text.Combine({"Seattle", "WA"})
```

`"SeattleWA"`

Example 2

Combine text values "Seattle" and "WA" separated by a comma and a space, ", ".

```
Text.Combine({"Seattle", "WA"}, ", ")
```

`"Seattle, WA"`

Text.Contains

3/15/2021 • 2 minutes to read

Syntax

```
Text.Contains(text as nullable text, substring as text, optional comparer as nullable function)
as nullable logical
```

About

Detects whether the text `text` contains the text `substring`. Returns true if the text is found.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Find if the text "Hello World" contains "Hello".

```
Text.Contains("Hello World", "Hello")
```

`true`

Example 2

Find if the text "Hello World" contains "hello".

```
Text.Contains("Hello World", "hello")
```

`false`

Text.End

3/15/2021 • 2 minutes to read

Syntax

```
Text.End(text as nullable text, count as number) as nullable text
```

About

Returns a `text` value that is the last `count` characters of the `text` value `text`.

Example 1

Get the last 5 characters of the text "Hello, World".

```
Text.End("Hello, World", 5)
```

`"World"`

Text.EndsWith

3/15/2021 • 2 minutes to read

Syntax

```
Text.EndsWith(text as nullable text, substring as text, optional comparer as nullable function)
as nullable logical
```

About

Indicates whether the given text, `text`, ends with the specified value, `substring`. The indication is case-sensitive.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Check if "Hello, World" ends with "world".

```
Text.EndsWith("Hello, World", "world")
```

`false`

Example 2

Check if "Hello, World" ends with "World".

```
Text.EndsWith("Hello, World", "World")
```

`true`

Text.Format

3/15/2021 • 2 minutes to read

Syntax

```
Text.Format(formatString as text, arguments as any, optional culture as nullable text) as text
```

About

Returns formatted text that is created by applying `arguments` from a list or record to a format string `formatString`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Format a list of numbers.

```
Text.Format("#{0}, #{1}, and #{2}.", {17, 7, 22})
```

`"17, 7, and 22."`

Example 2

Format different data types from a record according to United States English culture.

```
Text.Format(
    "The time for the #[distance] km run held in #[city] on #[date] was #[duration].",
    [
        city = "Seattle",
        date = #date(2015, 3, 10),
        duration = #duration(0, 0, 54, 40),
        distance = 10
    ],
    "en-US"
)
```

`"The time for the 10 km run held in Seattle on 3/10/2015 was 00:54:40."`

Text.From

3/15/2021 • 2 minutes to read

Syntax

```
Text.From(value as any, optional culture as nullable text) as nullable text
```

About

Returns the text representation of `value`. The `value` can be a `number`, `date`, `time`, `datetime`, `datetimezone`, `logical`, `duration` or `binary` value. If the given value is null, `Text.From` returns null. An optional `culture` may also be provided (for example, "en-US").

Example 1

Create a text value from the number 3.

```
Text.From(3)
```

`"3"`

Text.FromBinary

3/15/2021 • 2 minutes to read

Syntax

```
Text.FromBinary(binary as nullable binary, optional encoding as nullable number) as nullable
text
```

About

Decodes data, `binary`, from a binary value in to a text value, using `encoding` type.

Text.InferNumberType

3/15/2021 • 2 minutes to read

Syntax

```
Text.InferNumberType(text as text, optional culture as nullable text) as type
```

About

Infers the granular number type (Int64.Type, Double.Type, etc.) of `text`. An error is raised if `text` is not a number. An optional `culture` may also be provided (for example, "en-US").

Text.Insert

3/15/2021 • 2 minutes to read

Syntax

```
Text.Insert(text as nullable text, offset as number, newText as text) as nullable text
```

About

Returns the result of inserting text value `newText` into the text value `text` at position `offset`. Positions start at number 0.

Example 1

Insert "C" between "B" and "D" in "ABD".

```
Text.Insert("ABD", 2, "C")
```

`"ABCD"`

Text.Length

3/15/2021 • 2 minutes to read

Syntax

```
Text.Length(text as nullable text) as nullable number
```

About

Returns the number of characters in the text `text`.

Example 1

Find how many characters are in the text "Hello World".

```
Text.Length("Hello World")
```

```
11
```

Text.Lower

3/15/2021 • 2 minutes to read

Syntax

```
Text.Lower(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of converting all characters in `text` to lowercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the lowercase version of "AbCd".

```
Text.Lower("AbCd")
```

`"abcd"`

Text.Middle

3/15/2021 • 2 minutes to read

Syntax

```
Text.Middle(text as nullable text, start as number, optional count as nullable number) as
nullable text
```

About

Returns `count` characters, or through the end of `text`; at the offset `start`.

Example 1

Find the substring from the text "Hello World" starting at index 6 spanning 5 characters.

```
Text.Middle("Hello World", 6, 5)
```

`"World"`

Example 2

Find the substring from the text "Hello World" starting at index 6 through the end.

```
Text.Middle("Hello World", 6, 20)
```

`"World"`

Text.NewGuid

3/15/2021 • 2 minutes to read

Syntax

```
Text.NewGuid() as text
```

About

Returns a new, random globally unique identifier (GUID).

Text.PadEnd

3/15/2021 • 2 minutes to read

Syntax

```
Text.PadEnd(text as nullable text, count as number, optional character as nullable text) as
nullable text
```

About

Returns a `text` value padded to length `count` by inserting spaces at the end of the text value `text`. An optional character `character` can be used to specify the character used for padding. The default pad character is a space.

Example 1

Pad the end of a text value so it is 10 characters long.

```
Text.PadEnd("Name", 10)
```

```
"Name      "
```

Example 2

Pad the end of a text value with "|" so it is 10 characters long.

```
Text.PadEnd("Name", 10, "|")
```

```
"Name||||||"
```

Text.PadStart

3/15/2021 • 2 minutes to read

Syntax

```
Text.PadStart(text as nullable text, count as number, optional character as nullable text) as
nullable text
```

About

Returns a `text` value padded to length `count` by inserting spaces at the start of the text value `text`. An optional character `character` can be used to specify the character used for padding. The default pad character is a space.

Example 1

Pad the start of a text value so it is 10 characters long.

```
Text.PadStart("Name", 10)
```

```
"      Name"
```

Example 2

Pad the start of a text value with "|" so it is 10 characters long.

```
Text.PadStart("Name", 10, "|")
```

```
"||||||Name"
```

Text.PositionOf

3/15/2021 • 2 minutes to read

Syntax

```
Text.PositionOf(text as text, substring as text, optional occurrence as nullable number,
optional comparer as nullable function) as any
```

About

Returns the position of the specified occurrence of the text value `substring` found in `text`. An optional parameter `occurrence` may be used to specify which occurrence position to return (first occurrence by default). Returns -1 if `substring` was not found.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Get the position of the first occurrence of "World" in the text "Hello, World! Hello, World!".

```
Text.PositionOf("Hello, World! Hello, World!", "World")
```

`7`

Example 2

Get the position of last occurrence of "World" in "Hello, World! Hello, World!".

```
Text.PositionOf("Hello, World! Hello, World!", "World", Occurrence.Last)
```

`21`

Text.PositionOfAny

3/15/2021 • 2 minutes to read

Syntax

```
Text.PositionOfAny(text as text, characters as list, optional occurrence as nullable number) as
any
```

About

Returns the position of the first occurrence of any of the characters in the character list `text` found in the text value `characters`. An optional parameter `occurrence` may be used to specify which occurrence position to return.

Example 1

Find the position of "W" in text "Hello, World!".

```
Text.PositionOfAny("Hello, World!", {"W"})
```

`7`

Example 2

Find the position of "W" or "H" in text "Hello, World!".

```
Text.PositionOfAny("Hello, World!", {"H", "W"})
```

`0`

Text.Proper

3/15/2021 • 2 minutes to read

Syntax

```
Text.Proper(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of capitalizing only the first letter of each word in text value `text`. All other letters are returned in lowercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Use `Text.Proper` on a simple sentence.

```
Text.Proper("the QUICK BrOWn fOx jUmPs oVER tHe LAzy DoG")
```

```
"The Quick Brown Fox Jumps Over The Lazy Dog"
```

Text.Range

3/15/2021 • 2 minutes to read

Syntax

```
Text.Range(text as nullable text, offset as number, optional count as nullable number) as
nullable text
```

About

Returns the substring from the text `text` found at the offset `offset`. An optional parameter, `count`, can be included to specify how many characters to return. Throws an error if there aren't enough characters.

Example 1

Find the substring from the text "Hello World" starting at index 6.

```
Text.Range("Hello World", 6)
```

`"World"`

Example 2

Find the substring from the text "Hello World Hello" starting at index 6 spanning 5 characters.

```
Text.Range("Hello World Hello", 6, 5)
```

`"World"`

Text.Remove

3/15/2021 • 2 minutes to read

Syntax

```
Text.Remove(text as nullable text, removeChars as any) as nullable text
```

About

Returns a copy of the text value `text` with all the characters from `removeChars` removed.

Example 1

Remove characters , and ; from the text value.

```
Text.Remove("a,b;c", {",",";"})
```

`"abc"`

Text.RemoveRange

3/15/2021 • 2 minutes to read

Syntax

```
Text.RemoveRange(text as nullable text, offset as number, optional count as nullable number) as
nullable text
```

About

Returns a copy of the text value `text` with all the characters from position `offset` removed. An optional parameter, `count` can by used to specify the number of characters to remove. The default value of `count` is 1. Position values start at 0.

Example 1

Remove 1 character from the text value "ABEFC" at position 2.

```
Text.RemoveRange("ABEFC", 2)
```

`"ABFC"`

Example 2

Remove two characters from the text value "ABEFC" starting at position 2.

```
Text.RemoveRange("ABEFC", 2, 2)
```

`"ABC"`

Text.Repeat

3/15/2021 • 2 minutes to read

Syntax

```
Text.Repeat(text as nullable text, count as number) as nullable text
```

About

Returns a text value composed of the input text `text` repeated `count` times.

Example 1

Repeat the text "a" five times.

```
Text.Repeat("a", 5)
```

`"aaaaa"`

Example 2

Repeat the text "helloworld" three times.

```
Text.Repeat("helloworld.", 3)
```

`"helloworld.helloworld.helloworld."`

Text.Replace

3/15/2021 • 2 minutes to read

Syntax

```
Text.Replace(text as nullable text, old as text, new as text) as nullable text
```

About

Returns the result of replacing all occurrences of text value `old` in text value `text` with text value `new`. This function is case sensitive.

Example 1

Replace every occurrence of "the" in a sentence with "a".

```
Text.Replace("the quick brown fox jumps over the lazy dog", "the", "a")
```

```
"a quick brown fox jumps over a lazy dog"
```

Text.ReplaceRange

3/15/2021 • 2 minutes to read

Syntax

```
Text.ReplaceRange(text as nullable text, offset as number, count as number, newText as text) as
nullable text
```

About

Returns the result of removing a number of characters, `count`, from text value `text` beginning at position `offset` and then inserting the text value `newText` at the same position in `text`.

Example 1

Replace a single character at position 2 in text value "ABGF" with new text value "CDE".

```
Text.ReplaceRange("ABGF", 2, 1, "CDE")
```

`"ABCDEF"`

Text.Reverse

3/15/2021 • 2 minutes to read

Syntax

```
Text.Reverse(text as nullable text) as nullable text
```

About

Reverses the provided `text`.

Example 1

Reverse the text "123".

```
Text.Reverse("123")
```

`"321"`

Text.Select

3/15/2021 • 2 minutes to read

Syntax

```
Text.Select(text as nullable text, selectChars as any) as nullable text
```

About

Returns a copy of the text value `text` with all the characters not in `selectChars` removed.

Example 1

Select all characters in the range of 'a' to 'z' from the text value.

```
Text.Select("a,b;c", {"a".."z"})
```

`"abc"`

Text.Split

3/15/2021 • 2 minutes to read

Syntax

```
Text.Split(text as text, separator as text) as list
```

About

Returns a list of text values resulting from the splitting a text value `text` based on the specified delimiter, `separator`.

Example 1

Create a list from the "|" delimited text value "Name|Address|PhoneNumber".

```
Text.Split("Name|Address|PhoneNumber", "|")
```

Name
Address
PhoneNumber

Text.SplitAny

3/15/2021 • 2 minutes to read

Syntax

```
Text.SplitAny(text as text, separators as text) as list
```

About

Returns a list of text values resulting from the splitting a text value `text` based on any character in the specified delimiter, `separators`.

Example 1

Create a list from the text value "Jamie|Campbell|Admin|Adventure Works|www.adventure-works.com".

```
Text.SplitAny("Jamie|Campbell|Admin|Adventure Works|www.adventure-works.com", "|")
```

Jamie
Campbell
Admin
Adventure Works
www.adventure-works.com

Text.Start

3/15/2021 • 2 minutes to read

Syntax

```
Text.Start(text as nullable text, count as number) as nullable text
```

About

Returns the first `count` characters of `text` as a text value.

Example 1

Get the first 5 characters of "Hello, World".

```
Text.Start("Hello, World", 5)
```

`"Hello"`

Text.StartsWith

3/15/2021 • 2 minutes to read

Syntax

```
Text.StartsWith(text as nullable text, substring as text, optional comparer as nullable
function) as nullable logical
```

About

Returns true if text value `text` starts with text value `substring`.

- `text`: A `text` value which is to be searched
- `substring`: A `text` value which is the substring to be searched for in `substring`
- `comparer`: *[Optional]* A `Comparer` used for controlling the comparison. For example, `Comparer.OrdinalIgnoreCase` may be used to perform case insensitive searches

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case insensitive or culture and locale aware comparisons.
The following built in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture aware comparison

Example 1

Check if the text "Hello, World" starts with the text "hello".

```
Text.StartsWith("Hello, World", "hello")
```

`false`

Example 2

Check if the text "Hello, World" starts with the text "Hello".

```
Text.StartsWith("Hello, World", "Hello")
```

`true`

Text.ToBinary

3/15/2021 • 2 minutes to read

Syntax

```
Text.ToBinary(text as nullable text, optional encoding as nullable number, optional
includeByteOrderMark as nullable logical) as nullable binary
```

About

Encodes the given text value, `text`, into a binary value using the specified `encoding`.

Text.ToList

3/15/2021 • 2 minutes to read

Syntax

```
Text.ToList(text as text) as list
```

About

Returns a list of character values from the given text value `text`.

Example 1

Create a list of character values from the text "Hello World".

```
Text.ToList("Hello World")
```

H
e
l
l
o
W
o
r
l
d

Text.Trim

3/15/2021 • 2 minutes to read

Syntax

```
Text.Trim(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all leading and trailing whitespace from text value `text`.

Example 1

Remove leading and trailing whitespace from " a b c d ".

```
Text.Trim("    a b c d   ")
```

`"a b c d"`

Text.TrimEnd

3/15/2021 • 2 minutes to read

Syntax

```
Text.TrimEnd(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all trailing whitespace from text value `text`.

Example 1

Remove trailing whitespace from " a b c d ".

```
Text.TrimEnd("    a b c d    ")
```

```
"    a b c d"
```

Text.TrimStart

3/15/2021 • 2 minutes to read

Syntax

```
Text.TrimStart(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all leading whitespace from text value `text`.

Example 1

Remove leading whitespace from " a b c d ".

```
Text.TrimStart("   a b c d    ")
```

```
"a b c d    "
```

Text.Upper

3/15/2021 • 2 minutes to read

Syntax

```
Text.Upper(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of converting all characters in `text` to uppercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the uppercase version of "aBcD".

```
Text.Upper("aBcD")
```

`"ABCD"`

TextEncoding.Ascii

3/15/2021 • 2 minutes to read

About

Use to choose the ASCII binary form.

TextEncoding.BigEndianUnicode

3/15/2021 • 2 minutes to read

About

Use to choose the UTF16 big endian binary form.

TextEncoding.Unicode

3/15/2021 • 2 minutes to read

About

Use to choose the UTF16 little endian binary form.

TextEncoding.Utf8

3/15/2021 • 2 minutes to read

About

Use to choose the UTF8 binary form.

TextEncoding.Utf16

3/15/2021 • 2 minutes to read

About

Use to choose the UTF16 little endian binary form.

TextEncoding.Windows

3/15/2021 • 2 minutes to read

About

Use to choose the Windows binary form.

Time functions

3/15/2021 • 2 minutes to read

These functions create and manipulate time values.

Time

FUNCTION	DESCRIPTION
Time.EndOfHour	Returns a DateTime value from the end of the hour.
Time.From	Returns a time value from a value.
Time.FromText	Returns a Time value from a set of date formats.
Time.Hour	Returns an hour value from a DateTime value.
Time.Minute	Returns a minute value from a DateTime value.
Time.Second	Returns a second value from a DateTime value
Time.StartOfHour	Returns the first value of the hour from a time value.
Time.ToRecord	Returns a record containing parts of a Date value.
Time.ToText	Returns a text value from a Time value.
#time	Creates a time value from hour, minute, and second.

Time.EndOfHour

3/15/2021 • 2 minutes to read

Syntax

```
Time.EndOfHour(dateTime as any) as any
```

About

Returns a `time`, `datetime`, or `datetimezone` value representing the end of the hour in `dateTime`, including fractional seconds. Time zone information is preserved.

- `dateTime`: A `time`, `datetime`, or `datetimezone` value from which the end of the hour is calculated.

Example 1

Get the end of the hour for 5/14/2011 05:00:00 PM.

```
Time.EndOfHour(#datetime(2011, 5, 14, 17, 0, 0))
```

```
#datetime(2011, 5, 14, 17, 59, 59.9999999)
```

Example 2

Get the end of the hour for 5/17/2011 05:00:00 PM -7:00.

```
Time.EndOfHour(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 5, 17, 5, 59, 59.9999999, -7, 0)
```

Time.From

3/15/2021 • 2 minutes to read

Syntax

```
Time.From(value as any, optional culture as nullable text) as nullable time
```

About

Returns a `time` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, `Time.From` returns `null`. If the given `value` is `time`, `value` is returned. Values of the following types can be converted to a `time` value:

- `text`: A `time` value from textual representation. See `Time.FromText` for details.
- `datetime`: The time component of the `value`.
- `datetimezone`: The time component of the local datetime equivalent of `value`.
- `number`: A `time` equivalent to the number of fractional days expressed by `value`. If `value` is negative or greater or equal to 1, an error is returned.

If `value` is of any other type, an error is returned.

Example 1

Convert `0.7575` to a `time` value.

```
Time.From(0.7575)
```

```
#time(18, 10, 48)
```

Example 2

Convert `#datetime(1899, 12, 30, 06, 45, 12)` to a `time` value.

```
Time.From(#datetime(1899, 12, 30, 06, 45, 12))
```

```
#time(06, 45, 12)
```

Time.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Time.FromText(text as nullable text, optional culture as nullable text) as nullable time
```

About

Creates a `time` value from a textual representation, `text`, following ISO 8601 format standard. An optional `culture` may also be provided (for example, "en-US").

- `Time.FromText("12:34:12")` // Time, hh:mm:ss
- `Time.FromText("12:34:12.1254425")` // hh:mm:ss.nnnnnnn

Example 1

Convert `"10:12:31am"` into a Time value.

```
Time.FromText("10:12:31am")
```

```
#time(10, 12, 31)
```

Example 2

Convert `"1012"` into a Time value.

```
Time.FromText("1012")
```

```
#time(10, 12, 00)
```

Example 3

Convert `"10"` into a Time value.

```
Time.FromText("10")
```

```
#time(10, 00, 00)
```

Time.Hour

3/15/2021 • 2 minutes to read

Syntax

```
Time.Hour(dateTime as any) as nullable number
```

About

Returns the hour component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the hour in #datetime(2011, 12, 31, 9, 15, 36).

```
Time.Hour(#datetime(2011, 12, 31, 9, 15, 36))
```

`9`

Time.Minute

3/15/2021 • 2 minutes to read

Syntax

```
Time.Minute(dateTime as any) as nullable number
```

About

Returns the minute component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the minute in #datetime(2011, 12, 31, 9, 15, 36).

```
Time.Minute(#datetime(2011, 12, 31, 9, 15, 36))
```

`15`

Time.Second

3/15/2021 • 2 minutes to read

Syntax

```
Time.Second(dateTime as any) as nullable number`
```

About

Returns the second component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the second value from a datetime value.

```
Time.Second(#datetime(2011, 12, 31, 9, 15, 36.5))
```

`36.5`

Time.StartOfHour

3/15/2021 • 2 minutes to read

Syntax

```
Time.StartOfHour(dateTime as any) as any
```

About

Returns the first value of the hour given a `time`, `datetime` or `datetimezone` type.

Example 1

Find the start of the hour for October 10th, 2011, 8:10:32AM (`#datetime(2011, 10, 10, 8, 10, 32)`).

```
Time.StartOfHour(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 10, 10, 8, 0, 0)`

Time.ToRecord

3/15/2021 • 2 minutes to read

Syntax

```
Time.ToRecord(time as time) as record
```

About

Returns a record containing the parts of the given Time value, `time`.

- `time`: A `time` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#time(11, 56, 2)` value into a record containing Time values.

```
Time.ToRecord(#time(11, 56, 2))
```

HOUR	11
MINUTE	56
SECOND	2

Time.ToText

3/15/2021 • 2 minutes to read

Syntax

```
Time.ToText(time as nullable time, optional format as nullable text, optional culture as
nullable text) as nullable text
```

About

Returns a textual representation of `time`. An optional `format` may be provided to customize the formatting of the text. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get a textual representation of #time(11, 56, 2).

```
Time.ToText(#time(11, 56, 2))
```

`"11:56 AM"`

Example 2

Get a textual representation of #time(11, 56, 2) with format option.

```
Time.ToText(#time(11, 56, 2), "hh:mm")
```

`"11:56"`

#time

6/22/2021 • 2 minutes to read

Syntax

```
#time(hour as number, minute as number, second as number) as time
```

About

Creates a time value from numbers representing the hour, minute, and (fractional) second. Raises an error if these conditions are not true:

- 0 ≤ hour ≤ 24
- 0 ≤ minute ≤ 59
- 0 ≤ second < 60
- if hour is 24, then minute and second must be 0

Type functions

3/15/2021 • 2 minutes to read

These functions create and manipulate type values.

Type

FUNCTION	DESCRIPTION
Type.AddTableKey	Add a key to a table type.
Type.ClosedRecord	The given type must be a record type returns a closed version of the given record type (or the same type, if it is already closed)
Type.Facets	Returns the facets of a type.
Type.ForFunction	Creates a function type from the given .
Type.ForRecord	Returns a Record type from a fields record.
Type.FunctionParameters	Returns a record with field values set to the name of the parameters of a function type, and their values set to their corresponding types.
Type.FunctionRequiredParameters	Returns a number indicating the minimum number of parameters required to invoke the a type of function.
Type.FunctionReturn	Returns a type returned by a function type.
Type.Is	Type.Is
Type.IsNullable	Returns true if a type is a nullable type; otherwise, false.
Type.IsOpenRecord	Returns whether a record type is open.
Type.ListItem	Returns an item type from a list type.
Type.NonNullable	Returns the non nullable type from a type.
Type.OpenRecord	Returns an opened version of a record type, or the same type, if it is already open.
Type.RecordFields	Returns a record describing the fields of a record type with each field of the returned record type having a corresponding name and a value that is a record of the form [Type = type, Opional = logical].
Type.ReplaceFacets	Replaces the facets of a type.

FUNCTION	DESCRIPTION
Type.ReplaceTableKeys	Replaces the keys in a table type.
Type.TableColumn	Returns the type of a column in a table.
Type.TableKeys	Returns keys from a table type.
Type.TableRow	Returns a row type from a table type.
Type.TableSchema	Returns a table containing a description of the columns (i.e. the schema) of the specified table type.
Type.Union	Returns the union of a list of types.

Type.AddTableKey

3/15/2021 • 2 minutes to read

Syntax

```
Type.AddTableKey(table as type, columns as list, isPrimary as logical) as type
```

About

Adds a key to the given table type.

Type.ClosedRecord

3/15/2021 • 2 minutes to read

Syntax

```
Type.ClosedRecord(type as type) as type
```

About

Returns a closed version of the given `record` `type` (or the same type, if it is already closed).

Example 1

Create a closed version of `type [ A = number,…]`.

```
Type.ClosedRecord(type [A = number, ...])
```

`type [A = number]`

Type.Facets

3/15/2021 • 2 minutes to read

Syntax

```
Type.Facets(type as type) as record
```

About

Returns a record containing the facets of `type`

Type.ForFunction

3/15/2021 • 2 minutes to read

Syntax

```
Type.ForFunction(signature as record, min as number) as type
```

About

Creates a `function type` from `signature`, a record of `ReturnType` and `Parameters`, and `min`, the minimum number of arguments required to invoke the function.

Example 1

Creates the type for a function that takes a number parameter named X and returns a number.

```
Type.ForFunction([ReturnType = type number, Parameters = [X = type number]], 1)
```

```
type function (X as number) as number
```

Type.ForRecord

3/15/2021 • 2 minutes to read

Syntax

```
Type.ForRecord(fields as record, open as logical) as type
```

About

Returns a type that represents records with specific type constraints on fields.

Type.FunctionParameters

3/15/2021 • 2 minutes to read

Syntax

```
Type.FunctionParameters(type as type) as record
```

About

Returns a record with field values set to the name of the parameters of `type`, and their values set to their corresponding types.

Example

Find the types of the parameters to the function `(x as number, y as text)`.

```
Type.FunctionParameters(type function (x as number, y as text) as any)
```

x	[Type]
y	[Type]

Type.FunctionRequiredParameters

3/15/2021 • 2 minutes to read

Syntax

```
Type.FunctionRequiredParameters(type as type) as number
```

About

Returns a number indicating the minimum number of parameters required to invoke the input `type` of function.

Example 1

Find the number of required parameters to the function `(x as number, optional y as text)`.

```
Type.FunctionRequiredParameters(type function (x as number, optional y as text) as any)
```

```
1
```

Type.FunctionReturn

3/15/2021 • 2 minutes to read

Syntax

```
Type.FunctionReturn(type as type) as type
```

About

Returns a type returned by a function `type`.

Example 1

Find the return type of `() as any)`.

```
Type.FunctionReturn(type function () as any)
```

`type any`

Type.Is

3/15/2021 • 2 minutes to read

Syntax

```
Type.Is(type1 as type, type2 as type) as logical
```

About

Type.Is

Type.IsNullable

3/15/2021 • 2 minutes to read

Syntax

```
Type.IsNullable(type as type) as logical
```

About

Returns `true` if a type is a `nullable` type; otherwise, `false`.

Example 1

Determine if `number` is nullable.

```
Type.IsNullable(type number)
```

`false`

Example 2

Determine if `type nullable number` is nullable.

```
Type.IsNullable(type nullable number)
```

`true`

Type.IsOpenRecord

3/15/2021 • 2 minutes to read

Syntax

```
Type.IsOpenRecord(type as type) as logical
```

About

Returns a `logical` indicating whether a record `type` is open.

Example 1

Determine if the record `type [ A = number, ...]` is open.

```
Type.IsOpenRecord(type [A = number, ...])
```

`true`

Type.ListItem

3/15/2021 • 2 minutes to read

Syntax

```
Type.ListItem(type as type) as type
```

About

Returns an item type from a list `type`.

Example 1

Find item type from the list `{number}`.

```
Type.ListItem(type {number})
```

`type number`

Type.NonNullable

3/15/2021 • 2 minutes to read

Syntax

```
Type.NonNullable(type as type) as type
```

About

Returns the non `nullable` type from the `type`.

Example 1

Return the non nullable type of `type nullable number`.

```
Type.NonNullable(type nullable number)
```

`type number`

Type.OpenRecord

3/15/2021 • 2 minutes to read

Syntax

```
Type.OpenRecord(type as type) as type
```

About

Returns an opened version of the given `record` `type` (or the same type, if it is already opened).

Example 1

Create an opened version of `type [ A = number]`.

```
Type.OpenRecord(type [A = number])
```

`type [A = number, ...]`

Type.RecordFields

3/15/2021 • 2 minutes to read

Syntax

```
Type.RecordFields(type as type) as record
```

About

Returns a record describing the fields of a record `type`. Each field of the returned record type has a corresponding name and a value, in the form of a record `[ Type = type, Optional = logical ]`.

Example

Find the name and value of the record `[ A = number, optional B = any]`.

```
Type.RecordFields(type [A = number, optional B = any])
```

A	[Record]
B	[Record]

Type.ReplaceFacets

3/15/2021 • 2 minutes to read

Syntax

```
Type.ReplaceFacets(type as type, facets as record) as type
```

About

Replaces the facets of `type` with the facets contained in the record `facets`.

Type.ReplaceTableKeys

3/15/2021 • 2 minutes to read

Syntax

```
Type.ReplaceTableKeys(tableType as type, keys as list) as type
```

About

Returns a new table type with all keys replaced by the specified list of keys.

Type.TableColumn

3/15/2021 • 2 minutes to read

Syntax

```
Type.TableColumn(tableType as type, column as text) as type
```

About

Returns the type of the column `column` in the table type `tableType`.

Type.TableKeys

3/15/2021 • 2 minutes to read

Syntax

```
Type.TableKeys(tableType as type) as list
```

About

Returns the possibly empty list of keys for the given table type.

Type.TableRow

3/15/2021 • 2 minutes to read

Syntax

```
Type.TableRow(table as type) as type
```

About

Type.TableRow

Type.TableSchema

3/15/2021 • 2 minutes to read

Syntax

```
Type.TableSchema(tableType as type) as table
```

About

Returns a table describing the columns of `tableType`.

Type.Union

3/15/2021 • 2 minutes to read

Syntax

```
Type.Union(types as list) as type
```

About

Returns the union of the types in `types`.

Uri functions

3/15/2021 • 2 minutes to read

These functions create and manipulate URI query strings.

Uri

FUNCTION	DESCRIPTION
Uri.BuildQueryString	Assemble a record into a URI query string.
Uri.Combine	Returns a Uri based on the combination of the base and relative parts.
Uri.EscapeDataString	Encodes special characters in accordance with RFC 3986.
Uri.Parts	Returns a record value with the fields set to the parts of a Uri text value.

Uri.BuildQueryString

3/15/2021 • 2 minutes to read

Syntax

```
Uri.BuildQueryString(query as record) as text
```

About

Assemble the record `query` into a URI query string, escaping characters as necessary.

Example

Encode a query string which contains some special characters.

```
Uri.BuildQueryString([a = "1", b = "+$"])
```

```
"a=1&b=%2B%24"
```

Uri.Combine

3/15/2021 • 2 minutes to read

Syntax

```
Uri.Combine(baseUri as text, relativeUri as text) as text
```

About

Returns an absolute URI that is the combination of the input `baseUri` and `relativeUri`.

Uri.EscapeDataString

3/15/2021 • 2 minutes to read

Syntax

```
Uri.EscapeDataString(data as text) as text
```

About

Encodes special characters in the input `data` according to the rules of RFC 3986.

Example

Encode the special characters in "+money$".

```
Uri.EscapeDataString("+money$")
```

```
"%2Bmoney%24"
```

Uri.Parts

3/15/2021 • 2 minutes to read

Syntax

```
Uri.Parts(absoluteUri as text) as record
```

About

Returns the parts of the input `absoluteUri` as a record, containing values such as Scheme, Host, Port, Path, Query, Fragment, UserName and Password.

Example 1

Find the parts of the absolute URI "www.adventure-works.com".

```
Uri.Parts("www.adventure-works.com")
```

SCHEME	http
HOST	www.adventure-works.com
PORT	80
PATH	/
QUERY	[Record]
FRAGMENT	
USERNAME	
PASSWORD	

Example 2

Decode a percent-encoded string.

```
let
    UriUnescapeDataString = (data as text) as text => Uri.Parts("http://contoso?a=" & data)[Query][a]
in
    UriUnescapeDataString("%2Bmoney%24")
```

`"+money$"`

Value functions

3/15/2021 • 2 minutes to read

These functions evaluate and perform operations on values.

Values

FUNCTION	DESCRIPTION
Value.Alternate	Expresses alternate query plans.
Value.Compare	Returns 1, 0, or -1 based on value1 being greater than, equal to, or less than the value2. An optional comparer function can be provided.
Value.Equals	Returns whether two values are equal.
Value.Expression	Returns an AST that represents the value's expression.
Value.NativeQuery	Evaluates a query against a target.
Value.NullableEquals	Returns a logical value or null based on two values .
Value.Optimize	If value represents a query that can be optimized, returns the optimized query. Otherwise returns value.
Value.Type	Returns the type of the given value.

Arithmetic operations

FUNCTION	DESCRIPTION
Value.Add	Returns the sum of the two values.
Value.Divide	Returns the result of dividing the first value by the second.
Value.Multiply	Returns the product of the two values.
Value.Subtract	Returns the difference of the two values.

Arithmetic parameters

FUNCTION	DESCRIPTION
Precision.Double	An optional parameter for the built-in arthimetic operators to specify double precision.
Precision.Decimal	An optional parameter for the built-in arthimetic operators to specify decimal precision.

Parameter types

TYPE	DESCRIPTION
Value.As	Value.As is the function corresponding to the as operator in the formula language. The expression value as type asserts that the value of a value argument is compatible with type as per the is operator. If it is not compatible, an error is raised.
Value.Is	Value.Is is the function corresponding to the is operator in the formula language. The expression value is type returns true if the ascribed type of vlaue is compatible with type, and returns false if the ascribed type of value is incompatible with type.
Value.ReplaceType	A value may be ascribed a type using Value.ReplaceType. Value.ReplaceType either returns a new value with the type ascribed or raises an error if the new type is incompatible with the value's native primitive type. In particular, the function raises an error when an attempt is made to ascribe an abstract type, such as any. When replacing a the type of a record, the new type must have the same number of fields, and the new fields replace the old fields by ordinal position, not by name. Similarly, when replacing the type of a table, the new type must have the same number of columns, and the new columns replace the old columns by ordinal position.

IMPLEMENTATION	DESCRIPTION
DirectQueryCapabilities.From	DirectQueryCapabilities.From
Embedded.Value	Accesses a value by name in an embedded mashup.
Value.Firewall	Value.Firewall
Variable.Value	Variable.Value
SqlExpression.SchemaFrom	SqlExpression.SchemaFrom
SqlExpression.ToExpression	SqlExpression.ToExpression

Metadata

FUNCTION	DESCRIPTION
Value.Metadata	Returns a record containing the input's metadata.
Value.RemoveMetadata	Removes the metadata on the value and returns the original value.
Value.ReplaceMetadata	Replaces the metadata on a value with the new metadata record provided and returns the original value with the new metadata attached.

Lineage

FUNCTION	DESCRIPTION
Graph.Nodes	This function is intended for internal use only.
Value.Lineage	This function is intended for internal use only.
Value.Traits	This function is intended for internal use only.

DirectQueryCapabilities.From

3/15/2021 • 2 minutes to read

Syntax

```
DirectQueryCapabilities.From(value as any) as table
```

About

DirectQueryCapabilities.From

Embedded.Value

3/15/2021 • 2 minutes to read

Syntax

```
Embedded.Value(value as any, path as text) as any
```

About

Accesses a value by name in an embedded mashup.

Graph.Nodes

3/15/2021 • 2 minutes to read

Syntax

```
Graph.Nodes(graph as record) as list
```

About

This function is intended for internal use only.

Precision.Decimal

3/15/2021 • 2 minutes to read

About

An optional parameter for the built-in arithmetic operators to specify decimal precision.

Precision.Double

3/15/2021 • 2 minutes to read

About

An optional parameter for the built-in arithmetic operators to specify double precision.

SqlExpression.SchemaFrom

3/15/2021 • 2 minutes to read

Syntax

```
SqlExpression.SchemaFrom(schema as any) as any
```

About

SqlExpression.SchemaFrom

SqlExpression.ToExpression

3/15/2021 • 2 minutes to read

Syntax

```
SqlExpression.ToExpression(sql as text, environment as record) as text
```

About

SqlExpression.ToExpression

Value.Add

3/15/2021 • 2 minutes to read

Syntax

```
Value.Add(value1 as any, value2 as any, optional precision as nullable number) as any
```

About

Returns the sum of `value1` and `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Alternate

3/15/2021 • 2 minutes to read

Syntax

```
Value.Alternates(alternates as list) as any
```

About

Expresses alternate query plans within a query plan expression obtained through `Value.Expression(Value.Optimize(...))`. Not intended for other uses.

Value.As

3/15/2021 • 2 minutes to read

Syntax

```
Value.As(value as any, type as type) as any
```

About

Value.As

Value.Compare

3/15/2021 • 2 minutes to read

Syntax

```
Value.Compare(value1 as any, value2 as any, optional precision as nullable number) as number
```

About

Returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second one.

Value.Divide

3/15/2021 • 2 minutes to read

Syntax

```
Value.Divide(value1 as any, value2 as any, optional precision as nullable number) as any
```

About

Returns the result of dividing `value1` by `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Equals

3/15/2021 • 2 minutes to read

Syntax

```
Value.Equals(value1 as any, value2 as any, optional precision as nullable number) as logical
```

About

Returns true if value `value1` is equal to value `value2`, false otherwise.

Value.Expression

3/15/2021 • 2 minutes to read

Syntax

```
Value.Expression(value as any) as nullable record
```

About

Returns an AST that represents the value's expression.

Value.Firewall

3/15/2021 • 2 minutes to read

Syntax

```
Value.Firewall(key as text) as any
```

About

Value.Firewall

Value.FromText

3/15/2021 • 2 minutes to read

Syntax

```
Value.FromText(text as any, optional culture as nullable text) as any
```

About

Decodes a value from a textual representation, `text`, and interprets it as a value with an appropriate type. `Value.FromText` takes a text value and returns a number, a logical value, a null value, a datetime value, a duration value, or a text value. The empty text value is interpreted as a null value. An optional `culture` may also be provided (for example, "en-US").

Value.Is

3/15/2021 • 2 minutes to read

Syntax

```
Value.Is(value as any, type as type) as logical
```

About

Value.Is

Value.Lineage

3/15/2021 • 2 minutes to read

Syntax

```
Value.Lineage(value as any) as any
```

About

This function is intended for internal use only.

Value.Metadata

3/15/2021 • 2 minutes to read

Syntax

```
Value.Metadata(value as any) as any
```

About

Returns a record containing the input's metadata.

Value.Multiply

3/15/2021 • 2 minutes to read

Syntax

```
Value.Multiply(value1 as any, value2 as any, optional precision as nullable number) as any
```

About

Returns the product of multiplying `value1` by `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.NativeQuery

3/15/2021 • 2 minutes to read

Syntax

```
Value.NativeQuery(target as any, query as text, optional parameters as any, optional options as nullable record) as any
```

About

Evaluates `query` against `target` using the parameters specified in `parameters` and the options specified in `options`.

The output of the query is defined by `target`.

`target` provides the context for the operation described by `query`.

`query` describes the query to be executed against `target`. `query` is expressed in a manner specific to `target` (e.g. a T-SQL statement).

The optional `parameters` value may contain either a list or record as appropriate to supply the parameter values expected by `query`.

The optional `options` record may contain options that affect the evaluation behavior of `query` against `target`. These options are specific to `target`.

Value.NullableEquals

3/15/2021 • 2 minutes to read

Syntax

```
Value.NullableEquals(value1 as any, value2 as any, optional precision as nullable number) as
nullable logical
```

About

Returns null if either argument `value1`, `value2` is null, otherwise equivalent to Value.Equals.

Value.Optimize

3/15/2021 • 2 minutes to read

Syntax

```
Value.Optimize(value as any) as any
```

About

When used within Value.Expression, if `value` represents a query that can be optimized, this function indicates that the optimized expression should be returned. Otherwise, `value` will be passed through with no effect.

Value.RemoveMetadata

3/15/2021 • 2 minutes to read

Syntax

```
Value.RemoveMetadata(value as any, optional metaValue as any) as any
```

About

Strips the input of metadata.

Value.ReplaceMetadata

3/15/2021 • 2 minutes to read

Syntax

```
Value.ReplaceMetadata(value as any, metaValue as any) as any
```

About

Replaces the input's metadata information.

Value.ReplaceType

3/15/2021 • 2 minutes to read

Syntax

```
Value.ReplaceType(value as any, type as type) as any
```

About

Value.ReplaceType

Value.Subtract

3/15/2021 • 2 minutes to read

Syntax

```
Value.Subtract(value1 as any, value2 as any, optional precision as nullable number) as any
```

About

Returns the difference of `value1` and `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Traits

3/15/2021 • 2 minutes to read

Syntax

```
Value.Traits(value as any) as table
```

About

This function is intended for internal use only.

Value.Type

3/15/2021 • 2 minutes to read

Syntax

```
Value.Type(value as any) as type
```

About

Returns the type of the given value.

Variable.Value

3/15/2021 • 2 minutes to read

Syntax

```
Variable.Value(identifier as text) as any
```

About

Variable.Value