OVERVIEW

What is **Power BI**?

"It is Microsoft's Self-Service Business Intelligence tool for processing and analyzing data."

Components

› **Power BI Desktop**—Desktop application
› **Report**—Multi-page canvas visible to end users. It serves for the placement of visuals, buttons, images, slicers, etc.
› **Data**—Preview pane for data loaded into a model.
› **Model**—Editable scheme of relationships between tables in a model. Pages can be used in a model for easier navigation.
› **Power Query**—A tool for connecting, transforming, and combining data.

"Apart from the standard version, there is also a version for Report Server."

› **Power BI Service**—A cloud service enabling access to, and sharing and administration of, output data.
› **Workspace**—There are three types of workspaces: **Personal**, **Team**, and **Develop a template app**. They serve as storage and enable controlled access to output data.
› **Dashboard**—A space consisting of **tiles** in which visuals and report pages are stored.*
› **Report**—A **report** of pages containing visuals.*
› **Worksheet**—A published Excel worksheet. Can be used as a tile on a dashboard.
› **Dataset**—A published sequence for fetching and transforming data from **Power BI Desktop**.
› **Dataflow**—Online Power Query representing a special dataset outside of Power BI Desktop.*
› **Application**—A single location combining one or more reports or dashboards.*
› **Admin portal**—Administration portal that lets you configure capacities, permissions, and capabilities for individual users and workspaces.

**Can be created and edited in the Power BI Service environment.*

› **Data Gateway**—On-premises data gateway that lets you transport data from an internal network or a custom device to the **Power BI Service**.
› **Power BI Mobile**—Mobile app for viewing reports. Mobile view is applied, if it exists, otherwise the desktop view is used.
› **Report Server**—On-premises version of Power BI Service.
› **Report Builder**—A tool for creating page reports.

Built-in and additional languages

Built-in languages

› **M/Query Language**—Lets you transform data in Power Query.
› **DAX** (Data Analysis Expressions)—Lets you define custom calculated tables, columns, and measures in Power BI Desktop.

"Both languages are natively available in Power BI, which eliminates the need to install anything."

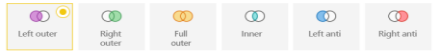
Additional languages

› **Python**—Lets you fetch data and create visuals. Requires installation of the Python language on your computer and enabling Python scripting.
› **R**—Lets you fetch and transform data and create visuals. Requires installation of the R language on your computer and enabling R scripting.

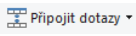
Power Query

Works with data fetched from data sources using connectors. This data is then processed at the Power BI app level and stored to an **in-memory** database in the program background. This means that data is not processed at the source level. The basic unit in Power Query is **query**, which means one sequence consisting of **steps**. A step is a data command that dictates what should happen to the data when it is loaded into Power BI. The basic definition of each step is based on its use:

› **Connecting data**—Each query begins with a function that provides data for the subsequent steps. E.g., data can be loaded from **Excel**, **SQL database**, **SharePoint** etc. Connection steps can also be used later.
› **Transforming data**—Steps that modify the structure of the data. These steps include features such as Pivot Column, converting columns to rows, grouping data, splitting columns, removing columns, etc. Transformation steps are necessary in order to clean data from not entirely clean data sources.
› **Combining data**—Data split into multiple source files needs to be combined so that it can be analyzed in bulk. Functions include merging queries and appending queries.
› **Merge queries**—This function merges queries based on the selected key. The primary query then contains a column which can be used to extract data from a secondary query. Supports typical join types:



› **Append query**—Places the resulting data from one or more selected queries under the primary query. In this case, data is placed in columns with names that are an exact match. Non-matching columns form new columns with a unique name in the primary query.



› **Custom function**—A query intended to apply a pre-defined sequence of steps so that the author does not need to create them repeatedly. The custom function can also accept input data (values, sheets, etc.) to be used in the sequence.
› **Parameter**—Values independent of datasets. These values can then be used in queries. Values enable the quick editing of a model because they can be changed in the Power BI Service environment.

Dataflow

The basic unit is a table or **Entity** consisting of columns or **Fields**. Just like Queries in Power Query, Entities in Dataflows consist of sequences of steps. The result of such steps is stored in native Azure Data Lake Gen 2.

"You can connect a custom Data Lake where the data will be stored."

There are three types of entities:
Standard entity—It only works with data fetched directly from a data source or with data from non-stored entities within the same dataflow.
Computed entity*—It uses data from another stored entity within the same dataflow.
Linked entity*—Uses data from an entity located in another dataflow. If data in the original entity is updated, the new data is directly passed to all linked entities.

**Can only be used in a dedicated Power BI Premium workspace.*

"It supports custom functions as well as parameters."

DAX

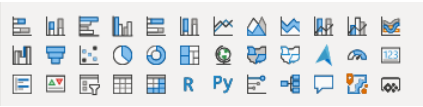
Language developed for data analysis. It enables the creation of the following objects using expressions:
› **Measures**
› **Calculated Columns**
› **Calculated Tables**

Each expression starts with the = sign, followed by links to tables/columns/functions/measures and operators. The following operators are supported:
› **Arithmetic** { + , - , / , * , ^ }
› **Comparison** { = , == , > , < , >= , <= , <> }
› **Text concatenation** { & , && , II , IN }
› **Precedence** { (,) }

Operators and functions require that all values/columns used are of the same data type or of a type that can be freely converted; such as a date or a number.

Visualization

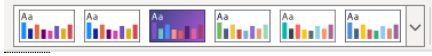
Visualizations or visuals let you present data in various graphical forms, from graphs to tables, maps, and values. Some visuals are linked to other services outside Power BI, such as Power Apps.



In addition to basic visuals, Power BI supports creating custom visuals. Custom visuals can be added using a file import or from a free Marketplace offering certified and non-certified visuals. Certification is optional, but it verifies whether, among other things, a visual accesses external services and resources.

Themes

Serves as a single location for configuring all native graphical settings for visuals and pages.



By default, you can choose from 19 predefined themes. Custom themes can be added.

A custom theme can be applied in two different ways:
› **Modification of an existing theme**—A native window that lets you modify a theme directly in the Power BI environment.
› **Importing a JSON file**—Any file you create only defines the formatting that should change. Everything else remains the same. The advantage of this approach is that you can customize any single visual.

*"The resulting theme **can be exported** in the JSON format and **used** in any report without the need to create a theme from scratch."*

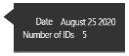
Drill Down

The Visual that supports the embedding of hierarchies enables drilling down to the embedded hierarchy's individual levels using the following symbols:

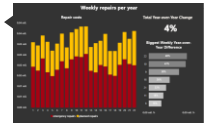
Drill up to a higher-level hierarchy
Drill down to a specific field
Drill down to the next level in the hierarchy
Expand next-level hierarchy

Tooltip/Custom Tooltip

› **Tooltip** —A default detail preview pane which appears above a visual when you hover over its values.

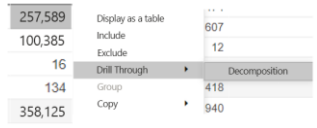


› **Custom Tooltip** —A custom tooltip is a custom-designed report page identified as descriptive. When you hover over visual, a page appears with content filtered based on criteria specified by the value in the visual.



Drill-through

Drill-through lets you pass from a data overview visual to a page with specific details. The target page is displayed with all the applied filters affecting the value from which the drill-through originated.



Bookmarks

Bookmarks capture the currently configured view or a report page visual. Later, you can go back to that state by selecting the saved bookmark. Setting options:
› **Data**—Stores filters, applied sort order in visuals and slicers. By selecting the bookmark, you can re-apply the corresponding settings.
› **Display**—Stores the state of the display for visuals and report elements (buttons, images, etc.). By selecting the bookmark, you can go back to the previously stored state of the display.
› **Current page**—Stores the currently displayed page. By selecting the bookmark, you can go back the to stored page.

License

Per-user License
› **Free**—Can be obtained for any Microsoft work or school email account. Intended for personal use. Users with this license can only use the personal workspace. They cannot share or consume shared content.

"If it is not available in Premium workspace"

› **Pro**—It is associated with a work/school account priced at €8.40 per month or it is included in the E5 license. Intended for team collaboration. Let's users access team workspaces, consume shared content, and use apps.
› **Premium per User** – Includes all Power BI Pro license capabilities, and adds features such as paginated reports, AI, greater frequency for refresh rate, XMLA endpoint and other capabilities that are only available to Premium subscribers.
Per-tenant License
› **Premium**—Premium is set up for individual workspaces. 0 to N workspaces can be used with a single version of this license. It provides dedicated server computing power based on license type: P1, P2, P3, P4*, P5*. It offers more space for datasets, extended metrics for individual workspaces, managed consumption of dedicated capacity, linking of Azure AI features with datasets, and access for users with **Free** licenses to shared content. Prices start at €4,212.30.

**Only available upon special request. Intended for models larger than 100GB.*

› **Embedded**—Supports embedding dashboards and reports in custom apps.
› **Report Server**—Included in Premium or SQL Server Enterprise licenses.

Administration

› **Use metrics**—Usage metrics let you monitor Power BI usage for your organization.
› **Users**—The Users tab provides a link to the Microsoft 365 admin center.
› **Audit logs**—The Audit logs tab provides a link to the Security & Compliance center.
› **Tenant settings**—Tenant settings enable fine-grained control over features made available to your organization. It controls which features will be enabled or disabled and for which users and groups.
› **Capacity settings**—The Power BI Premium tab enables you to manage any Power BI Premium and Embedded capacities.
› **Embed codes**—You can view the embed codes that are generated for your tenant to share reports publicly. You can also revoke or delete codes.
› **Organization visuals**—You can control which type of Power BI visuals users can access across the organization.
› **Azure connections**—You can control workspace-level storage permissions for Azure Data Lake Gen 2.
› **Workspaces**—You can view the workspaces that exist in your tenant on the Workspaces tab.
› **Custom branding**—You can customize the look of Power BI for your whole organization.
› **Protection metrics**—The report shows how sensitivity labels help protect your content.
› **Featured content**—You can manage all the content promoted in the Featured section.

External Tools

They simplify the use of Power BI and extend the capabilities offered in Power BI. These tools are mostly developed by the community. Recommended external tools:
› Tabular Editor
› DAX studio
› ALM Toolkit
› VertiPaq Analyzer

DAX

What is **DAX**?

“ Data Analysis Expressions (DAX) is a library of functions and operators combined to create formulas and expressions “

Introduction to DAX

› **Where to find**
› Power BI, Power Pivot for Excel, Microsoft Analysis Services
› **Purpose**
› DAX was created to enumerate formulas across the data model, where the data is stored in the form of tables, which can be linked together through the sessions. They may have a cardinality of either 1: 1, 1: N, or M: N and your direction, which decides which table filters which. These sessions are either active or inactive. The active session is automatically and participates in the calculation. The inactive is involved in this when it is activated, for example, by a function **USERELATIONSHIP()**

Basic concepts

› **Constructs and their notation**
› Table – ‘Table‘
› Column – [Column] -> ‘Table‘[Column]
› Measure – [NameOfMeasure]
› **Comments**
› Single-line (CTRL + ´) – // or --
› Multi-line – /* */
› **Data types**
› INTEGER
› DECIMAL
› CURRENCY
› DATETIME
› BOOLEAN
› STRING
› VARIANT (not implemented in Power BI)
› BINARY

› DAX can work very well with some types as well combined as if it were the same type. If so, for example, the DATETIME and INTEGER data types are supported operator "+" then it is possible to use them together.

Example: DATETIME ([Date]) + INTEGER (1) = DATETIME ([Date] + 1)

Operators

› **Arithmetic** { + , - , / , * , ^ }
› **Comparative** { = , == , > , < , >= , <= , <> }
› **Joining text** { & }
› **Logic** { && , II , IN, NOT }
› **Prioritization** { (,) }

Calculated Columns

› They behave like any other column in the table. Instead of coming from a data source, they are created through a DAX expression evaluated based on the current context line, and we cannot get values of another row directly.
› **Import mode.** Their evaluation and storage is in progress when processing the model.
› **DirectQuery mode.** They are evaluated at runtime, which may slow down the model.

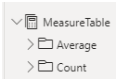
Profit = Trades[Quantity]*Trades[UnitPrice]

Measures

› They do not compare row-based calculations, but they perform aggregation of row-based values input contexts that the environment passes to the calculation. Because of this, there can be no pre-counting result. **It must be evaluated only at the moment when Measure is called.**
› The condition is that they must always be linked to the table to store their code, which is possible at any time alter. Because their calculation is no longer directly dependent, it is common practice to have one separate **Measure Table**, which groups all Measures into myself. For clarity, they are therefore further divided into **folders**.

Example of **Measure**:

SalesVolume = **SUM** (Trades[Quantity])



Variables

› Variables in DAX calculations allow avoiding repeated recalculations of the same procedure. Which might look like this:

```
NumberSort =
VAR _selectedNumber =
  SELECTEDVALUE( Table[Number] )
RETURN
IF( _selectedNumber < 4, _selectedNumber, 5 )
```

› Their declaration uses the word VAR after followed by the **name "="** and the **expression**. The first using the word VAR creates a section for DAX where possible declare such variables 1 to X. Individual variables always require a comment for their declaration VAR before setting the name. To end this section, the word **RETURN** that it defines is a necessary return point for calculations.
› **Variables are local only.**
› If there is a variable in the formula that is not used to get the result, this variable does not evaluate. **(Lazy Evaluation)**
› Evaluation of variables is **performed based on evaluated context instead of the context in which the variable is used directly.** Within one, The expression can be multiple VAR / RETURN sections that always serve to evaluate the currently evaluated context.
› They can store both the value and the whole table

Calculation contexts

› All calculations are evaluated on a base basis some context that the environment brings to the calculation. (**Evaluation context**)
› **Context Filter -**
The following calculation calculates the profit forindividual sales.

```
Revenue =
    SUMX( Trades,
      Trades[Quantity]*
      Trades[UnitPrice]
    )
```

Country	Revenue
Australia	2.838.077.18
Canada	73,959.95
Germany	340.392.76
Japan	1.833.026.16
Mexico	320.833.27
Nigeria	378,202.44
Total	**5,784,491.77**

If I place this calculation in a table without a **Country** column, then the result will be 5,784,491.77. With this column, we get "**Total**" the same as the previous calculation. Still, the individual records provide us with a **FILTER** context that filters in calculating the input the **SUMX** function's input. They behave the same way, for example, **AXES** in the chart.
› The filter context is can be adjusted with various functions, such as **FILTER,ALL, ALLSELECTED**
› **Row context -** Unlike the previous one, this context does not filter the table. It is used to iterate over tables and evaluate values columns. They are typical, but at the same time, specific example calculated columns that are calculated from data that are valid for the table row being evaluated. In particular that, manual creation is not required when creating the line context because **DAX** makes it. Above the mentioned example with the use of **SUMX** also hides in itself line context. Because **SUMX** is the function for that specified, the table in the first argument performs an iterative pass and evaluates the calculation line by line. The line context is possible to use even nested. Or, for each row of the table, evaluates each row of a different table.

Calculate type function

› **CALCULATE,** and **CALCULATETABLE** are functions that can programmatically set the **context filter**. In addition to this feature converts any existing line context to a context filter.
› **Calculate** and **Calculatetable** syntax:

```
CALCULATE / CALCULATETABLE (
                <expression> [, <filter1> [, ... ]]
                )
```

› The section filter within the **Calculate** expression is NOT of type **boolean** but **Table** type. Nevertheless, boolean can be used as an argument.
› Example of using the calculate function in a cumulative calculation the sum of sales for the last 12 months:

```
CALCULATE (
  SUM ( Trades[Quantity] ),
  DATESINPERIOD(
      DateKey[Date],
      MAX ( DateKey[Date] ),
      -1,
      YEAR
  ) )
```

› Syntax Sugar:

```
› [TradeVolume](Trades[Dealer] = 1)
                =
CALCULATE ( [TradeVolume], Trades[Dealer] = 1)
                =
CALCULATE ( [TradeVolume], FILTER (
                ALL (Trades[Dealer]) ,
                Trades[Dealer] = 1) )
```

Calcuation Groups

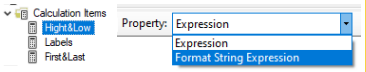
› They are very similar to **Calculated members** from MDX. In Power BI, it is not possible to create them directly in the **Desktop** application environment, but an External Tool **Tabular Editor** is required.
› This is a set of **Calculation Items** grouped according to their purpose and whose purpose is to prepare an expression, which can be used for different input measures, so it doesn’t have to write the same expression multiple times. To where she would be, but the input measure is placed **SELECTEDMEASURE()**.
Example:

```
CALCULATE ( SELECTEDMEASURE(),
                Trades[Dealer] = 1)
```

Name	Ordinal
Hight&Low	0
Labels	1
First&Last	2

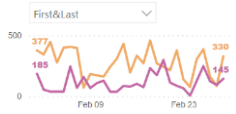
› From a visual point of view, the Calculation Group looks like a table with just two columns, "**Name**," "**Ordinal,**" and rows that indicate the individual Calculation Items.
› In addition to facilitating the reusability of the prepared expressions also provide the ability to modify the output format of individual calculations. Within this section, “**Format String Expression** ”often uses the **DAX** function **SELECTEDMEASUREFORMATSTRING**(), which returns a format string associated with the **Measures** being evaluated.

Example:




```
VAR _selectedCurrency = SELECTEDVALUE( Trades[Currency] )
RETURN
  SELECTEDMEASUREFORMATSTRING() & „ “ & _selectedCurrency
```

› In Power BI, they can all be evaluated pre-prepared items, or it is possible, for example, to use the cross-section to define items that are currently being evaluated
› Sometimes, however, it is necessary to enable the evaluation of Calculation Items only for Specific Measures. In that case, it is possible to use the **ISSELECTEDMEASURE**() function, whose output is a value of type boolean or the **SELECTEDMEASURENAME**() function that returns the name of the currently inserted measure as a string.



Conditions

› Like most languages, DAX uses the IF function. Within this language, it is defined by syntax:

```
IF ( <logical_test>, <value_if_true>[, <value_if_false>])
```

Where false, the branch is optional. The IF function explicitly evaluates only a branch that is based on the result of a logical test relevant.
› If both branches need to be evaluated, then there is a function **IF.EAGER()** whose syntax is the same as IF itself but evaluates as:

```
VAR _value_if_true = <value_if_true>
VAR _value_if_false = <value_if_false>
RETURN
IF (<logical_test>, _value_if_true, _value_if_false)
```

› **IF** has an alternative as **IFERROR**. Evaluates the expression and return the output from the <value_if_error> branch only if the expression returns an error. Otherwise, it returns the value of the expression itself.

› **DAX** supports concatenation of conditions, both using submerged ones **IF**, so thanks to the **SWITCH** function. It evaluates the expression against the list values and returns one of several possible result expressions.

Hierarchy

› DAX itself has no capability within the hierarchy to automatically convert your calculations to parent or child levels. Therefore, each level must Prepare Your Measures, which are then displayed based on the **ISINSCOPE** function. She tests which level to go just evaluating. Evaluation takes place from the bottom to the top level.
› The native data model used by DAX does not directly support its **parent/child** hierarchy. On the other hand, DAX contains functions that can convert this hierarchy to separate columns.
› **PATH** - It accepts two parameters, where the first parameter is the key ID column tables. The second parameter is the column that holds the parent ID of the row. The result of this function then looks like this: **1|2|3|4**
Syntax: **PATH**(<ID_columnName>, <parent_columnName>)
› **PATHITEM** – Returns a specific item based on the specified position from the string, resulting from the PATH function. Positions are counted from left to right. The inverted view uses the PATHITEMREVERSE function.
Syntax: **PATHITEM**(<path>, <position>[, <type>])
› **PATHLENGTH** – Returns the number of parent elements to the specified item in given the **PATH** result, including itself.
Syntax: **PATHLENGTH**(<path>)
› **PATHCONTAINS** – Returns **true** if the specified item is specified exists in the specified PATH path.
Syntax: **PATHCONTAINS**(<path>, <item>)

DAX Queries

› The basic building block of DAX queries is the expression EVALUATE followed by any expression whose output is a table.
Example:

```
EVALUATE
ALL (Trades[Dealer] )
```

› The **EVALUATE** statement can be divided into three primary sections. Each section has its specific purpose and its introductory word.
› **Definition –** It always starts with the word **DEFINE**. This section defines local entities such as tables, columns, variables, and measures. There can be one section definition for an entire query, although a query can contain multiple EVALUATEs
› **Query –** It always starts with the word **EVALUATE**. This section contains the table expression to evaluate and return as a result.
› **Result –** This is a section that is optional and starts with the word ORDER BY. It contains the possibility to sort the result based on the inserted inputs.
Example:

```
DEFINE
    VAR _tax = 0.79
EVALUATE
    ADDCOLUMNS(
    Trades,
    „AdjustedpProfit“,
        ( Trades[Quantity] * Trades[UnitPrice] ) * _tax
    )
ORDER BY [AdjustedpProfit]
```

› This type of notation is used, for example, in DAX Studio (**daxstudio.org**). It is a publicly available tool that provides free access to query validation, code debugging, and query performance measurement.
› **DAX studio** has the ability to connect directly to **Analysis Services**, **Power BI** a **Power Pivot for Excel**

Recommended sources

› **Marco Russo & Alberto Ferrari**
› Daxpatterns.com
› dax.guide
› The Definitive Guide to DAX

POWER QUERY

What is **Power Query**?

"An IDE for M development"

Components

› **Ribbon –** A ribbon containing settings and pre-built features by Power Query itself rewrites in M language for user convenience.
› **Queries –** simply a named M expression. Queries can be moved into groups
› **Primitive –** A primitive value is a single-part value, such as a number, logical, date, text, or null. A **null** value can be used to indicate the absence of any data.
› **List –** The list is an ordered sequence of values. M supports endless lists. Lists define the characters **"{"** and **"}"** indicate the beginning and the end of the list.
› **Record –** A record is a set of fields, where the field is a pair of which form the name and value. The name is a text value that is in the field record unique.
› **Table –** A table is a set of values arranged in named columns and rows.
Table can be operated on as if it is a list of records, or as if it is a record of lists. Table[Field]` (field reference syntax for records) returns a list of values in that field. `Table{i}` (list index access syntax) returns a record representing a row of the table.
› **Function –** A function is a value that when called using arguments creates a new value. Functions are written by listing the function argumets in parentheses, followed by the transition symbol "=>" and the expression defining the function. This expression usually refers to argumets by name. There are also functions without argumets.
› **Parameter –** The parameter stores a value that can be used for transformations. In addition to the name of the parameter and the value it stores, it also has other properties that provide metadata. The undeniable advantage of the parameter is that it can be changed from the **Power BI Service** environment without the need for direct intervention in the data set. Syntax of parameter is as regular query only thing that is special is that the metadata follows a specific format.
› **Formula Bar –** Displays the currently loaded step and allows you to edit it.To be able to see formula bar, It has to be enabled in the ribbon menu inside **View** category.
› **Query settings –** Settings that include the ability to edit the name and description of the query. It also contains an overview of all currently applied steps. Applied Steps are the variables defined in a let expression and they are represented by varaibles names.
› **Data preview –** A component that displays a preview of the data in the currently selected transformation step.
› **Status bar –** This is the bar located at the bottom of the screen. The row contains information about the approximate state of the rows, columns, and time the data was last reviewed. In addition to this information, there is profiling source information for the columns. Here it is possible to switch the profiling from 1000 rows to the entire data set.

Functions in Power Query

Knowledge of functions is your best helper when working with a functional language such as **M**. Functions are called with parentheses.
› **Shared –** Is a keyword that loads all functions (including help and example) and enumerators in result set. The call of function is made inside empty query using by **= # shared**

```
= #shared
```

Functions can be divided into two categories:
› Prefabricated – Example: Date.From()
› Custom – these are functions that the user himself prepares for the model by means of the extension of the notation by „()=> ", where the argumets that will be required for the evaluation of the function can be placed in parentheses. When using multiple argumets, it is necessary to separate them using a delimiter.

Data values

Each value type is associated with a literal syntax, a set of values of that type, a set of operators defined above that set of values, and an internal type attributed to the newly created values.
› **Null –** null
› **Logical –** true, false
› **Number –** 1, 2, 3, ...
› **Time –** #time(HH,MM,SS)
› **Date –** #date(yyyy,mm,ss)
› **DateTime –** #datetime(yyyy,mm,dd,HH,MM,SS)
› **DateTimeZone –** #datetimezone(yyyy,mm,dd,HH,MM,SS, 9,00)
› **Duration –** #duration(DD,HH,MM,SS)
› **Text –** "text"
› **Binary –** #binary("link")
› **List –** {1, 2, 3}
› **Record –** [A = 1, B = 2]
› **Table –** #table({columns},{{first row contenct},{}...})*
› **Function –** (x) => x + 1
› **Type –** type { number }, type table [A = any, B = text]
* The index of the first row of the table is the same as for the records in sheet 0

Operators

There are several operators within the M language, but not every operator can be used for all types of values.
› **Primary operators**
› **(x)** – Parenthesized expression
› **x[i]** – Field Reference. Return value from record, list of values from table.
› **x{i}** – Item access. Return value from list, record from table. "Placing the "**?**" Character after the operator returns null if the index is not in the list "
› **x(...)** – Function invocation
› **{1 .. 10}** – Automatic list creation from 1 to 10
› **...** – Not implemented
› **Mathematical operators** – **+, -, *, /**
› **Comparative operators**
› **>** , **>=** – Greater than, greater than or equal to
› **<** , **<=** – Less than, less than or equal to
› **=** , **<>** – is equal, is not equal. Equal returns true even for null = null
› **Logical operators**
› **and** – short-circuiting conjunction
› **or** – short-circuiting disjunction
› **not** – logical negation
› **Type operators**
› **as** – Is compatible nullable-primitive type or error
› **is** – Test if compatible nullable-primitive type
› **Metadata** - The word **meta** assigns metadata to a value.
Example of assigning metadata to variable **x:**
"x meta y" or **"x meta [name = x, value = 123,...]"**
Within Power Query, the priority of the operators applies, so for example "X + Y * Z" will be evaluated as "X + (Y * Z)"

Comments

M language supports **two** versions of comments:
› Single-line comments – can be created by **//** before code
› Shortcut: **CTRL + ´**
› Multi-line comments – can be created by **/*** before code and ***/** after code
› Shortcut: **ALT + SHIFT + A**

let expression

The expression let is used to capture the value from an intermediate calculation in a named variable. These named variables are local in scope to the `let` expression. The construction of the term let looks like this:

```
let
    name_of_variable = <expression>,
   returnVariable = <function>(name_of_variable)
in
returnVariable
```

When it is evaluated, the following always applies:

› Expressions in variables define a new range containing identifiers from the production of the list of variables and must be present when evaluating terms within a list variables. The expressions in the list of variables are they can refer to each other
› All variables must be evaluated before the term let is evaluated.
› If expressions in variables are not available, let will not be evaluated
› Errors that occur during query evaluation propagate as an error to other linked queries.

Conditions

Even in Power Query, there is an "**If**" expression, which, based on the inserted condition, decides whether the result will be a true-expression or a false-expression.

Syntactic form of If expression:
if <predicate> **then** < true-expression > **else** < false-expression >
"**else** is required in M's conditional expression "

Condition entry:
If x > 2 **then** 1 **else** 0
If [Month] > [Fiscal_Month] **then** true **else** false

If expression is the only conditional in M. If you have multiple predicates to test, you must chain together like:

```
if <predicate>
  then < true-expression >
  else if <predicate>
    then < false-true-expression >
    else  < false-false-expression >
```

When evaluating the conditions, the following applies:

› If the value created by evaluating the if a condition is not a logical value, then an error with the reason code "**Expression.Error**„ is raised
› A true-expression is evaluated only if the **if** condition evaluates to **true. Otherwise**, false-expression is evaluated**.**
› If expressions in variables are not available, they must not be evaluated
› The error that occurred during the evaluation of the condition will spread further either in the form of a failure of the entire query or "**Error**" value in the record.

The expression try... otherwise

Capturing errors is possible, for example, using the **try** expression. An attempt is made to evaluate the expression after the word **try**. If an error occurs during the evaluation, the expression after the word **otherwise** is applied

Syntax example:
try Date.From([textDate]) **otherwise** null

Custom function

Example of custom function entries:
(x, y) => Number.From(**x**) + Number.From(**y**)

```
(x) =>
let
   out = Number.From(x) +
   Number.From(Date.From(DateTime.LocalNow()))
in
   out
```

The input argumets to the functions are of two types:
› **Required –** All commonly written argumets in (). Without these argumets, the function cannot be called.
› **Optional –** Such a parameter may or may not be to function to enter. Mark the parameter as **optional** by placing text before the argument name "**Optional**". For example **(optional x)**. If it does not happen fulfillment of an optional argument, so be the same for for calculation purposes, but its value will be null.
Optional arguments must come after required arguments.

Arguments can be annotated with `as <type>` to indicate required type of the argument. The function will throw a type error if called with arguments of the wrong type. Functions can also have annotated return of them. This annotation is provided as:
(x as number, y as text) as logical => <expression>

The return of the functions is very different. The output can be a sheet, a table, one value but also other functions. This means that one function can produce another function. Such a function is written as follows:

let first = **(x)=> () =>** let out = {1..x} in out in first

When evaluating functions, it holds that:

› Errors caused by evaluating expressions in a list of expressions or in a function expression will propagate further either as a failure or as an "**Error**" value
› The number of arguments created from the argument list must be compatible with the formal argumets of the function, otherwise an error will occur with reason code "**Expression.Error**"

Recursive functions

For recursive functions is necessary to use the character "**@**" which refers to the function within its calculation. A typical recursive function is the factorial. The function for the factorial can be written as follows:

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

Each

Functions can be called against specific arguments. However, if the function needs to be executed for each record, an entire sheet, or an entire column in a table, it is necessary to append the word **each** to the code. As the name implies, for each context record, it applies the procedure behind it. **Each** is never required! It simply makes it easier to define a function in-line for functions which require a function as their argument.

Syntax Sugar

› Each is essentially a syntactic abbreviation for declaring non-type functions, using a single formal parameter named. Therefore, the following notations are semantically equivalent:

```
let
  Source = ...,
  addColumn = Table.AddColumn(Source, „NewName", each [field1] + 1)
in
  addColumn
```

```
let
  Source = ...,
  add1ToField1 = (_) => [field1] + 1,
  addColumn(Source,"NewName",add1ToField1)
in
```

The second piece of syntax sugar is that bare square brackets are syntax sugar for field access of a Record named `_`.

Query Folding

As the name implies, it is about composing. Specifically, the steps in Power Query are composed into a single query, which is then implemented against the data source. Data sources that supports Query folding are resources that support the concept of query languages as relational database sources. This means that, for example, a CSV or XML file as a flat file with data will definitely not be supported by Query Folding. Therefore, the transformation does not have to take place until after the data is loaded, but it is possible to get the data ready immediately. Unfortunately, not every source supports this feature.
› Valid functions
› Remove, Rename columns
› Row filtering
› Grouping, summarizing, pivot and unpivot
› Merge and extract data from queries
› Connect queries based on the same data source
› Add custom columns with simple logic
› Invalid functions
› Merge queries based on different data sources
› Adding columns with Index
› Change the data type of a column

DEMO

› Operators can be combined. For example, as follows:
› LastStep[Year]{[ID]}

*This means that you can get the value from another step based on the index of the column

› Production of a DateKey dimension goes like this:

```
#table(
   type table [Date=date, Day=Int64.Type, Month=Int64.Type,
MonthName=text, Year=Int64.Type,Quarter=Int64.Type],
   List.Transform(
      List.Dates(start_date, (start_date-endd_ate),
                  #duration(1, 0, 0 ,0)),
      each {_, Date.Day(_), Date.Month(_),
Date.MonthName(_), Date.Year(_), Date.QuarterOfYear(_)}
))
```

Keywords

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