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 content},{}...})*
› **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]}

`1899`

*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