POWER QUERY

Traducido Eva Ballester

Qué es **Power Query**?

"Un IDE para el desarrollo de M"

Componentes

› **Cinta de consultas –** una cinta de consultas que contiene la configuración y las características precompilados por la propia Power Query se reescribe en el lenguaje M para comodidad del usuario.

› **Consultas –** simplemente una expresión M con nombre. Las consultas se pueden mover a grupos

› **Primitivo –** un valor primitivo es un valor de una sola parte, como un número, lógico, fecha, texto o **null**. Se puede utilizar un valor nulo para indicar la ausencia de datos.

› **Lista –** la lista es una secuencia ordenada de valores. M soporta listas interminables. Las listas definen el carácter **"{" y "}"** indican el principio y el final de la lista.

› **Registro –** un registro es un conjunto de campos, donde el campo es un par de los cuales forman el nombre y el valor. El nombre es un valor de texto que se encuentra en el registro de campo único.

› **Tabla –** una tabla es un conjunto de valores organizados en columnas y filas. La tabla puede ser operada como si fuera una lista de registros, o como si fuera un registro de listas. La tabla [Campo]' (sintaxis de referencia de campo para registros) devuelve una lista de valores en ese campo.

› **Función –** una función es un valor que cuando se llama mediante argumentos crea un nuevo valor. Las funciones se escriben enumerando la función argumentos entre paréntesis, seguido del símbolo de transición ">" y la expresión que define la función. Esta expresión generalmente se refiere a argumentos por Nombre. También hay funciones sin argumentos.

› **Parámetro –** el parámetro almacena un valor que se puede utilizar para las transformaciones. Además del nombre del parámetro y el valor que almacena, también tiene otras propiedades que proporcionan metadatos. La ventaja innegable del parámetro es que se puede cambiar desde el entorno de **Power BI Service** sin necesidad de intervención directa en el conjunto de datos. La sintaxis del parámetro es como consulta regular que es especial es que los metadatos siguen un formato específico.

› **Barra de fórmulas –** muestra el paso actual y le permite editarlo. Para poder ver la barra de fórmulas, tiene que estar habilitado en el menú de la cinta de opciones dentro de la categoría Ver.

› **Configuración de Consultas –** configuración que incluye la capacidad de editar el nombre y la descripción de la consulta. También contiene una vista de todos los pasos aplicados actualmente. Los pasos aplicados son las variables definidas en una expresión **let** y están representados por nombres variables.

› **Previsualización de datos –** componente que muestra una vista previa de los datos en el paso de transformación seleccionado actualmente

› **Barra de estado** – Esta es la barra situada en la parte inferior de la pantalla. La fila contiene información sobre el estado aproximado de las filas, columnas y Hora de la Última revisión. Además de esta Información hay información de origen para las columnas. Aquí es posible cambiar el Perfiles De 1000 filas para todo el conjunto de datos

Funciones en Power Query

El conocimiento de las funciones es su mejor ayudante cuando se trabaja con un lenguaje funcional como **M**. Las funciones se llaman entre paréntesis.

› **Compartido –** es una palabra clave que carga todas las funciones (incluidas la ayuda y el ejemplo) y los enumeradores en el conjunto de resultados. La llamada de la función se realiza dentro de la consulta vacía usando por **= # shared**

```
= #shared
```

Las funciones se pueden dividir en dos categorías:

› Prefabricado – Ejemplo: Date.From()

› Custom – son funciones que el propio usuario prepara para el modelo mediante la extensión de la notación por **"()=>"**, donde los argumentos que serán necesarios para la evaluación de la función se pueden colocar entre paréntesis. Cuando se utilizan varios argumentos, es necesario separarlos mediante un delimitador.

Valor de los datos

Cada tipo de valor está asociado a una sintaxis literal, un conjunto de valores de ese tipo, un conjunto de operadores definidos por encima de ese conjunto de valores, y un tipo interno atribuido a los valores recién creados.

› **Null –** null
› **Lógico –** verdadero, falso
› **Número –** 1, 2, 3, ...
› **Tiempo –** #time(HH,MM,SS)
› **Fecha –** #date(yyyy,mm,ss)
› **DateTime –** #datetime(yyyy,mm,dd,HH,MM,SS)
› **DateTimeZone –** #datetimezone(yyyy,mm,dd,HH,MM,SS, 9,00)
› **Duración –** #duration(DD,HH,MM,SS)
› **Texto –** "texto"
› **Binario –** #binary("link")
› **Lista –** {1, 2, 3}
› **Registro –** [A = 1, B = 2]
› **Tabla –** #table({columns},{{first row contenct},{}...})*
› **Función –** (x) => x + 1
› **Tipo –** type { number }, type table [A = any, B = text]

* El índice de la primera fila de la tabla es el mismo que para los registros de la hoja 0

Operadores

Hay varios operadores dentro del lenguaje M, pero no todos los operadores se pueden utilizar para todos los tipos de valores.

› **Operadores primarios**
› **(x)** – Expresión entre paréntesis
› **x[i]** – Referencia de campo. Valor devuelto del registro, lista de valores de la tabla.
› **x{i}** – Acceso a los artículos. Devolver el valor de la lista, registro de la tabla.
"Colocar el "**?**" Carácter después de que el operador devuelve **null** si el índice no está en la lista "
› **x(...)** – Invocación de funciones
› **{1 .. 10}** – Creación automática de la lista del 1 al 10
› **...** – No implementado

› **Operadores matemáticos** – **+, -, *, /**

› **Operadores comparativos**
› **>** , **>=** – Mayor que, mayor o igual que
› **<** , **<=** – Menor que, menor o igual que
› **=** , **<>** – es igual, no es igual. Equal devuelve true incluso para null = null

› **Operadores lógicos**
› **and** – conjunción
› **or** – disyunción
› **not** – negación lógica

› **Tipo de operadores**
› **as** – Es compatible con tipo o error primitivo que acepta valores NULL
› **is** – pruebe si es compatible con un tipo primitivo que acepta valores NULL

› **Metadatos -** la palabra **meta** asigna metadatos a un valor. Ejemplo de asignación de metadatos a la variable **x:** **"x meta y" / "x meta [name = x, value = 123,...]"**

Dentro de Power Query, se aplica la prioridad de los operadores, por lo que por ejemplo "**X + Y * Z**" se evaluará como "**X + (Y * Z)**"

Comentarios

El lenguaje M admite dos versiones de comentarios:
› Comentarios de una sola línea – puede ser creado por **//** antes de código
› Acceso directo: **CTRL + ´**
› Comentarios multilínea (puede ser creado por **/*** antes de código y ***/** después del código
› Acceso directo : **ALT + SHIFT + A**

Expresión let

La expresión **let** se utiliza para capturar el valor de un cálculo intermedio en una variable con nombre. Estas variables con nombre son locales en el ámbito de la expresión '**let**'. La construcción del término es de la siguiente forma:

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

Cuando se evalúa, siempre se aplica lo siguiente:

› Las expresiones en variables definen un nuevo rango que contiene identificadores de la producción de la lista de variables y deben estar presentes cuando se evalúa los términos dentro de una lista de variables. Las expresiones en la lista de variables son que pueden hacer referencia entre sí
› Todas las variables deben evaluarse antes de que se evalúe el término let.
› Si las expresiones en variables no están disponibles, no se evaluará.
› Los errores que se producen durante la evaluación de consultas se propagan como un error a otras consultas vinculadas.

Condicionales

Incluso en Power Query, hay una expresión "**If**", que, en función de la condición insertada, decide si el resultado será una expresión true o una expresión falsa.

Forma sintáctica de expresión If:
if <predicate> **then** < true-expression > **else** < false-expression >
"**else** es necesario en la expresión condicional de M"

Entrada de condición:
If x > 2 **then** 1 **else** 0
If [Month] > [Fiscal_Month] **then** true **else** false

La expresión if es el único condicional en M. Si tiene varios predicados para probar, debe encadenar juntos como:

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

Al evaluar las condiciones, se aplica lo siguiente:

› Si el valor creado mediante la evaluación de la condición if no es un valor lógico, entonces se genera un error con el código de motivo "**Expression.Error**„
› Una expresión verdadera se evalúa solo si la condición if se evalúa como true. De lo contrario, se evalúara expresión falsa.
› Si las expresiones en variables no están disponibles, no deben evaluarse
› El error que se produjo durante la evaluación de la condición se extenderá aún más en forma de un error de toda la consulta o "**Error**" valor en el registro.

La expresión try... otherwise

Capturar errores es posible, por ejemplo, mediante la expresión try. Se intenta evaluar la expresión después de la palabra try. Si se produce un error durante la evaluación, se aplica una expresión después de la palabra otherwise

Ejemplo de sintaxis:
try Date.From([textDate]) **otherwise** null

Función personalizada

Ejemplo de entradas de función personalizadas:
(x, y) => Number.From(**x**) + Number.From(**y**)

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

Los argumentos de entrada a las funciones son de dos tipos:
› **Requerido –** Todos los argumentos comúnmente escritos en (). Sin estos argumentos, no se puede llamar a la función.
› **Opcional –** tal parámetro puede o no funcionar para entrar. Marque el parámetro como opcional colocando el texto antes del nombre del argumento "**Optional**". Por ejemplo **(optional x).** Si no se produce el cumplimiento de un argumento opcional, así sea el mismo para fines de cálculo, pero su valor será null. **Los argumentos opcionales deben venir después de los argumentos requeridos**

Los argumentos se pueden anotar con **'as <type>'** para indicar el tipo necesario del argumento. La función producirá un error de tipo si se llama con argumentos del tipo incorrecto. Las funciones también pueden tener un retorno anotado de ellas. Esta anotación se proporciona como:
(x as number, y as text) as logical => <expression>

El retorno de las funciones es muy diferente. La salida puede ser una hoja, una tabla, un valor, pero también otras funciones. Esto significa que una función puede producir otra función. Tal función se escribe como sigue:

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

En la evaluación de funciones, se mantiene que:

› Los errores causados por la evaluación de expresiones en una lista de expresiones o en una expresión de función se propagarán aún más como un error o como un valor "Error"
› El número de argumentos creados a partir de la lista de argumentos debe ser compatible con el argumento formal de la función, de lo contrario se producirá un error con el código de motivo "**Expression.Error**"

Funciones recursivas

Para las funciones recursivas es necesario utilizar el carácter "**@**" que hace referencia a la función dentro de su cálculo. Una función recursiva típica es el factorial. La función para el factorial se puede escribir de la siguiente manera:

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

Each

Se puede llamar a funciones en argumentos específicos. Sin embargo, si la función necesita ejecutarse para cada registro, una hoja completa o una columna completa de una tabla, es necesario anexar la palabra **each** al código. Como su nombre indica, para cada contexto record, aplica el procedimiento detrás de él. ¡**Each** nunca es necesario! Simplemente facilita la definición de una función en línea para funciones que requieren una función como argumento.

Syntax Sugar

› Syntax Sugar es esencialmente una abreviatura sintáctica para declarar funciones que no son de tipo, utilizando un único parámetro formal. Por lo tanto, las siguientes notaciones son semánticamente equivalentes:

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

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

La segunda pieza del Syntax Sugar es que los corchetes desnudos son las abreviaturas para el acceso de campo de un registro denominado '_'.

Plegado de Consultas

Capacidad para traducir la consulta la lenguaje de origen. En concreto, los pasos de Power Query se componen en una sola consulta, que se implementa en el origen de datos. Los orígenes de datos que admiten el plegado de consultas son recursos que admiten el concepto de lenguajes de consulta como orígenes de base de datos relacional.
Esto significa que, por ejemplo, un archivo CSV o XML como un archivo plano con datos no serán compatibles con el plegado de consultas. Por lo tanto, la transformación no tiene que tener lugar hasta después de cargar los datos, pero es posible preparar los datos inmediatamente. Desafortunadamente, no todas las fuentes admiten esta característica.

› Funciones válidas
› Eliminar, cambiar el nombre de las columnas
› Filtrado de filas
› Agrupación, resumen, pivote y despivo
› Combinar y extraer datos de consultas
› Conectar consultas basadas en el mismo origen de datos
› Añadir columnas personalizadas con lógica simple
› Funciones no válidas
› Combinar consultas basadas en diferentes orígenes de datos
› Adición de columnas con índice
› Cambiar el tipo de datos de una columna

DEMO

› Los operadores se pueden combinar. Por ejemplo, como sigue:
› LastStep[Year]{[ID]}

*Esto significa que usted puede obtener el valor de otro paso basado en el índice de la columna

› La producción de una dimensión **DateKey** es la iguiente:

```
#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(_)}))
```

Palabras Clave

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