Язык M Power Query. Типы logical, null и binary

Эта заметка завершает обзор примитивных типов языка M: текст, число, дата/время, логический, бинарный и null.[1]

Предыдущая заметка Следующая заметка

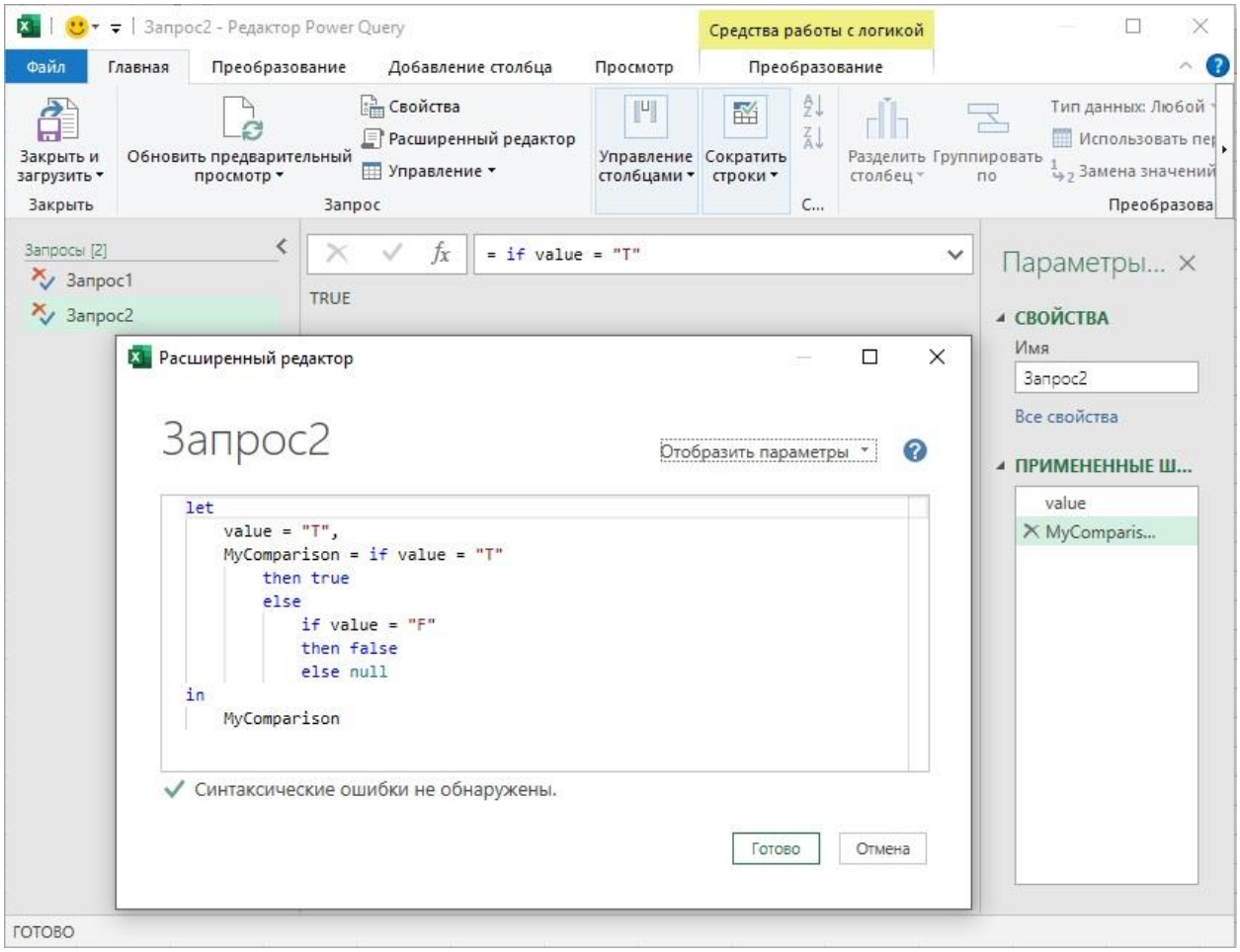

Рис. 1. Выражение *if* возвращает логическое значение

Логический

Тип *logical* хранит логические значения: *true* и *false*. Логический литерал можно получить преобразованием подходящей текстовой строки или числа. "false" преобразуется в *false*, "true" – в *true*. Число 0 – в *false*, любое другое число – в *true*.

= Logical.FromText("true") // true

= Logical.FromText("false") // false

= Logical.FromText("something else") /* возвращает ошибку
Expression.Error: Не удалось преобразовать в логический тип.*/

= Logical.From(0) // false

= Logical.From(1) // true

= Logical.From(-199) // true

Логический литерал также можно получить в результате сравнения или используя оператора *if*.

Листинг 1[2]

```
= 5 = Number.FromText("5") // возвращает true
```

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 9): Types—Logical, Null, Binary. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

[2] Номер листинга соответствует номеру запроса в приложенном Excel файле.

Листинг 2

```
let
   value = "T",
   MyComparison = if value = "T"
      then true
      else
         if value = "F"
         then false
         else null
in
   MyComparison // возвращает true (см. рис. 1)
```

Иногда логические значения не так очевидны, как true/false. Следующие пары 1/0, yes/no, Y/N, is something/is not something по сути, являются логическими значениями. Не позволяйте форме маскировать содержание значений. Если вы видите столбец, состоящий из подобных значений, преобразуйте его в тип *logical*. Как Power Query, так и Excel, работают лучше, когда тип столбца соответствует типу данных значений столбца.

null

Тип *null* – довольно странный тип. Он включает единственное значение – *null*. *null* представляет отсутствие значения (или неизвестное, или неопределенное). Если *null* представляет собой неизвестное значение, является ли *null* в принципе значением? Оставим такие глубокие размышления философам и теоретикам компьютерного языка. Нам важен практический аспект: как операторы должны обрабатывать *null*? Если сравниваются два *null*, должен ли результат быть *true*, ибо сравниваются идентичные значения. Или результат должен быть *null*? Потому что эти значения неизвестные, а сравнение двух неизвестных, тоже является неизвестным.

Итак, возможны, по крайней мере, два способа обработки *null*. Поскольку нельзя поддерживать несколько вариантов поведения, разработчики языка должны выбрать один из них. В M прямые сравнения (= и <>), когда аргумент равен *null*, возвращают *true* / *false*:

= null = null // true

= null <> null // false

= 1 = null // false

= null <> 1 // true

Сравнение с использованием оператора *and* (И), где хотя бы один из операндов *null* возвращает *null*. С одним исключением: если второй операнд *false*, сравнение возвращает *false*. Похоже ведет себя сравнение с использованием оператора *or* (ИЛИ). Если один из операндов *null* возвращается *null*. С одним исключением: если второй операнд *true*, сравнение возвращает *true*:

= null and null // null

= null and true // null

= null and false // false

= null or null // null

= null or true // true

= null or false // null

Если *null* используется с любым другим оператором, результат сравнения возвращает *null*.

= 1 > null // null

= 1 >= null // null

= null < null // null

= null <= null // null

= 10 + null // null

= null - 16.3 // null

= null * 25 // null

= 8 / null // null

= "abc" & null & "def" // null

И здесь есть исключения. Использование *null* с операторами *is* и *meta* (дает информацию о значении, а не работает непосредственно со значением) не всегда возвращают *null*.

Предпочитаете другое поведение?

Иногда работа M с *null* не соответствует вашим намерениям. Но языку присуща гибкость, и есть обходные пути. Рассмотрим последнюю строку из блока примеров выше. Допустим вы хотите объединить строки, даже если некоторые фрагменты возвращают *null*. Во-первых, можно проверить, содержит ли переменная значение *null*. Если это так, то заменить *null* пустой строкой перед конкатенацией.

Листинг 3

```
let
  value = null,
  NullToBlank = (input) => if (input = null) then "" else input
in
  "abc" & NullToBlank(value) & "def" // возвращает "abcdef"
```

Во-вторых, можно использовать функцию Text.Combine, которая игнорирует значения *null*. Код…

Листинг 4

```
= Text.Combine({"abc", null, "def"})
```

… вернет "abcdef".

В-третьих, можно использовать следующий трюк. Допустим, вы импортируете таблицу из Excel, содержащую положительные числа. Пустые ячейки в Excel, PQ импортирует как *null*. Если для последующих вычислений нужно преобразовать *null* в 0, воспользуйтесь следующей идеей:

Листинг 5

```
let
    import = null,
    future = List.Max({import,0})
 in
    future // возвращает 0
```

import может быть положительным числом, импортированным из Excel, или *null*, если ячейка была пустой. В первом случае List.Max() вернет число *import*, которое больше 0. Во втором случае List.Max() вернет 0, так как по логике PQ *null* меньше любого значения не равного *null*.

Вообще-то синтаксис List.Max включает четыре параметра:

```
List.Max(
    list as list,
    optional default as any,
    optional comparisonCriteria as any,
    optional includeNulls as nullable logical
) as any
```

В спецификации говорится. Функция List.Max() возвращает максимальный элемент в списке *list* или необязательное значение по умолчанию *default*, если список пуст. Для определения способа сравнения элементов в списке можно указать необязательный параметр *comparisonCriteria*. Если этот параметр равен *null*, используется функция сравнения по умолчанию. Четвертый параметр может быть *true*, тогда список {null} засчитается (хоть он и нулевой), и List.Max() вернет *null*. Если этот параметр *false*, список {null} не засчитается, и List.Max() вернет значение *default.*

В связи с вышесказанным можно использовать…

Листинг 6

```
let
   import = null,
   future = List.Max({import},0)
 in
   future // возвращает 0
```

Здесь 0 в future = List.Max({import},0) не элемент списка (как в листинге 5), а второй аргумент функции List.Max() – *default.*

Другая ситуация, в которой вам может потребоваться иная обработка значений *null*, связана со сравнениями меньше и больше. В M, если значение *null* сравнивается с использованием операторов >, >=, <, <=, результат равен *null*. Это вроде бы логично, поскольку невозможно узнать, является ли неизвестное значение больше или меньше другого значения (известного или неизвестного). Однако, возможен и иной взгляд, если принять ранжирование, в котором значение *null* меньше любого иного значения не равного *null*.

Если вы предпочитаете такое поведение, можете использовать функцию Value.Compare для выполнения сравнения. Эта библиотечная функция возвращает 0, если сравниваемые значения равны, -1, если первое значение меньше второго, и 1, если первое значение больше второго. В этой функции значение *null* оценивается как меньшее всех иных значений. Вот как работает функция Value.Compare:

= Value.Compare(1, 1) // 0, аргументы равны

= Value.Compare(10, 1) // 1, первый аргумент больше второго

= Value.Compare(10, 100) // -1, второй аргумент больше первого

= Value.Compare(null, 1) // -1; сравни с null < 1, которое возвращает null

= Value.Compare(null, null) // 0; сравни с null = null, которое возвращает null

= Value.Compare("a", null) // 1; сравни с "a" > null, которое возвращает null

И последнее замечание на эту тему. По умолчанию сравнение *null* = *null* принимает значение *true*. Если вы предпочитаете, чтобы значение *null* = *null* было равно *null*, используйте функцию Value.NullableEquals. В спецификации функции сказано. Возвращает *null*, если любой из аргументов *value1* и *value2* равен *null*, в противном случае — эквивалент Value.Equals.

= Value.NullableEquals(null, null) // null, сравни с null = null, которое возвращает true

Двоичный тип литерала

Обычно тип *binary* вы видите при работе с файлами. Как правило вы используете библиотечную функцию (или цепочку функций) для преобразования двоичного значения во что-то более удобное для обработки, например в таблицу.

Если по какой-то причине вы хотите использовать *binary* литерал, в M поддерживаются как списки чисел (целых / шестнадцатеричных), так и текстовые значения в кодировке base 64. Ниже мы видим одинаковые два байта, записанные с использованием трех синтаксисов.

= #binary({ 0x00, 0x10 }) // список шестнадцатеричных литералов

= #binary({ 0, 16 }) // список десятичных литералов

= #binary("ABA=") // строка в кодировке base 64

Стандартная библиотека содержит ряд функций для работы с двоичными значениями. Существуют функции, которые преобразуют значения в двоичные файлы и из них. Вы можете сжимать и распаковывать файлы с помощью *gzip* и *deflate*. Существуют также функции, которые пытаются извлечь тип содержимого и, в некоторых случаях, кодировку и информацию о разделителях *csv*. Это может быть полезно, если вы хотите найти все текстовые файлы в папке, когда не все они имеют расширения *.txt*. Существует даже семейство функций, которые можно использовать для анализа пользовательского формата, для нечетного случая, когда вам нужно проанализировать двоичное значение, которое не понимает ни одна библиотечная функция.

Тип type

Вот и все! Мы рассмотрели все примитивные типы, т.е., те, что содержат одно значение, за исключением типа *type*. Этот тип описывает доступные типы значений. Углубляться в *type* (и связанную с ним концепцию системы типов Power Query) – это более сложная тема, которую пока мы оставим в покое.

В следующей заметке

Примитивные типы являются основополагающими для работы с данными. Но часто мы хотим работать со значениями, которые как-то сгруппированы: в виде списка, записи или таблицы. Язык M имеет тип для каждой из этих группировок. В следующий раз мы начнем их изучать.