Язык M Power Query. Функции: вложенность, *each*, замыкание и рекурсия

Это очередная заметка, посвященная основам языка M Power Query. Ранее мы рассмотрели, как определять и вызывать функции. Но функции также можно создавать налету. Т.е. использовать функции без явного определения и вызова. Для этого в языке M применяется конструкция с ключевым словом *each*. Это слово часто появляется в коде, сгенерированном интерфейсом редактора запросов. Оказывается, это удобный ярлык для упрощения кода.[1]

Предыдущая заметка   Следующая заметка

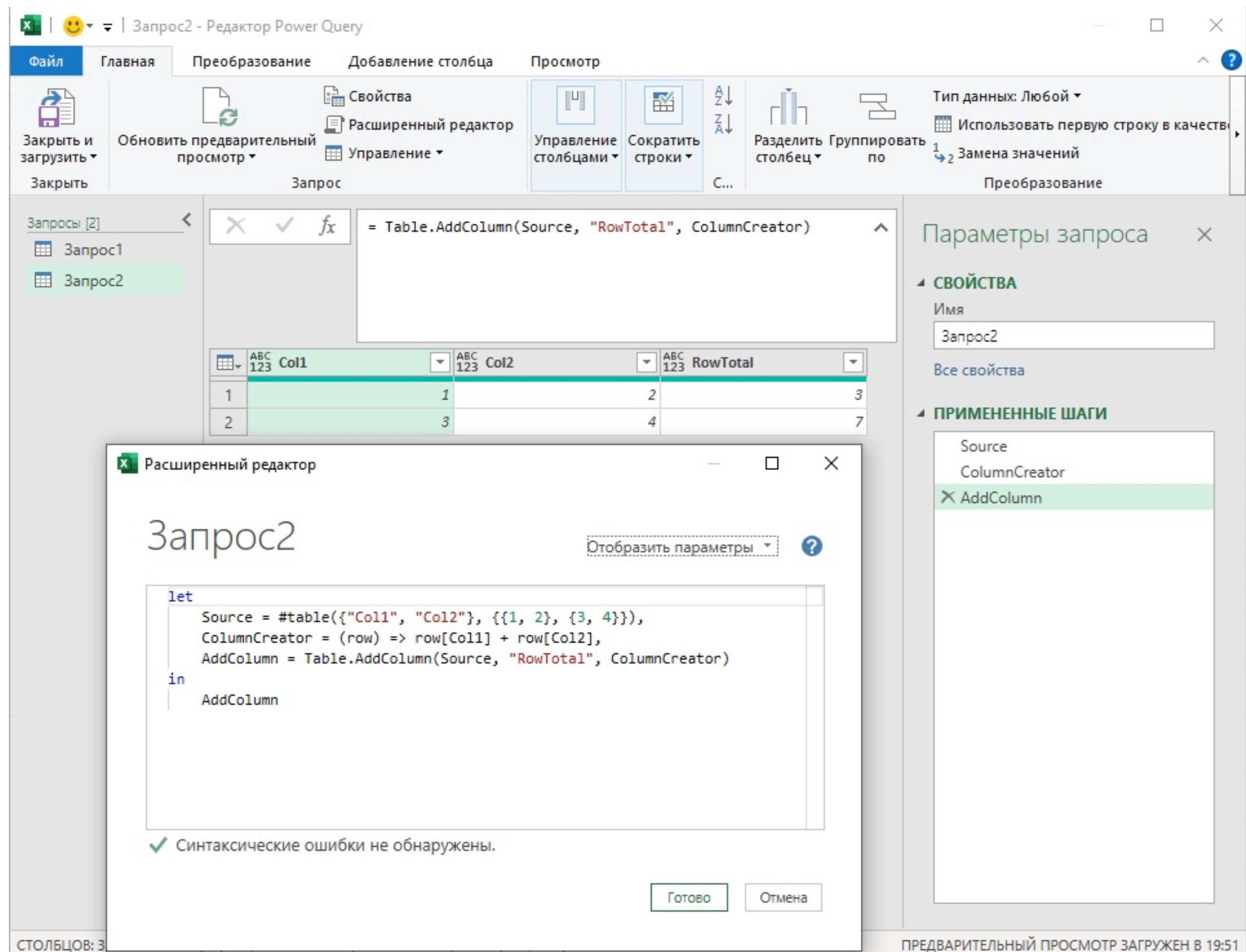

Рис. 1. Функция, как аргумент другой функции

Формула функции

Ранее мы описали функцию как *выражение, которое в итоге возвращает значение*. Это верно, но есть еще один промежуточный шаг. Технически функция вычисляется как нечто, что можно назвать *формулой*.[2] *Формула* сохраняется в переменных и передается в таком виде в ответ на запрос функции. Вызов функции запускает вычисления по формуле, и возвращает конечное значение. Думайте об этом так. Выражение функции – это код. Формула – это результат функции в переменных, но не значениях. Возвращаемое значение – это результат функции с использованием формулы при вызове функции с конкретными параметрами.

Листинг 1[3]

```
let
   Multiply = (x, y) => x * y,
```

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (part 3): Functions: Function Values, Passing, Returning, Defining Inline, Recursion. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

[2] Я решил использовать термин *формула*. В оригинале *function value*. Однако, дословный перевод приведет к смешению с понятием *значение функции*, которое в оригинале звучит как *function return value*.

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

```
    Result = Multiply(10, 20)
in
    Result
```

Выражение (x, y) => x * y в коде стоит до вычислений. Это выражение и есть код функции. Формула x * y вернется в ответ на вызов функции. Никакого значения еще нет. Когда функция вызывается в следующей строке (Multiply(10, 20)), фактически осуществляется обращение к формуле, которая возвращает значение 200. Это значение присваивается переменной *Result*.

Функция, как аргумент другой функции

Язык М позволяет передавать функцию в другую функцию. Внешняя функция реализует заложенный в ней алгоритм (формулу), а внутренняя функция рассчитывает один из аргументов. Рассмотрим добавление нового столбца в таблицу:

Листинг 2

```
let
    Source = #table({"Col1", "Col2"}, {{1, 2}, {3, 4}}),
    ColumnCreator = (row) => row[Col1] + row[Col2],
    AddColumn = Table.AddColumn(Source, "RowTotal", ColumnCreator)
in
    AddColumn
```

Стандартная функция M Power Query *Table.AddColumn* (внешняя) реализует добавление столбца. А значения, используемые для нового столбца, формирует функция *ColumnCreator*. В спецификации языка М указано, что функция *Table.AddColumn* имеет три обязательных параметра:

Table.AddColumn(table as table, newColumnName as text, columnGenerator as function, …) as table

В общем случае функция добавляет столбец с именем *newColumnName* в таблицу *table*. Значения для этого столбца вычисляются с помощью функции *columnGenerator*. В нашем примере функция *Table.AddColumn* добавляет столбец с именем *RowTotal* в таблицу *Source*. Значения для этого столбца вычисляются с помощью функции *ColumnCreator* (см. рис. 1).

Функция *Table.AddColumn* вызывает *ColumnCreator* для каждой строки таблицы, передавая ей текущую строку в качестве аргумента, а затем использует возвращаемое значение в качестве значения для нового столбца в текущей строке.

Table.AddColumn делает основную работу по добавлению нового столбца (общий алгоритм). А что именно поместить в новый столбец решает *ColumnCreator*, вызываемая по мере необходимости (в данном случае один раз для каждой строки). Нам не нужно писать функцию, которая делает всю работу сразу. Достаточно простой функции (*ColumnCreator*), которая принимает одну строку и выдает одно значение. Это делает нашу жизнь намного проще.

Определение функции налету

Поскольку функция – это выражение, а выражения могут использоваться в качестве параметров, мы можем определять функции налету (встроенно), непосредственно в списке параметров. Изменим код предыдущего примера, но теперь новую функцию определим в списке аргументов (вместо того, чтобы сначала присвоить функцию переменной):

Листинг 3

```
let
    Source = #table( {"Col1", "Col2"}, { {1, 2}, {3, 4} } ),
    AddColumn = Table.AddColumn(Source, "RowTotal", (row) => row[Col1] + row[Col2])
in
    AddColumn
```

Здесь (row) – это функция без имени с одним параметром – *row*.

Ключевое слово each и оператор подстановки _ (подчеркивание)

Функции, которые принимают один аргумент, настолько распространены в М, что для упрощения их использования придуман ярлык – ключевое слово *each*, которое является сокращением для (_) =>.

each – не единственный ярлык в PQ. Еще один – _ (нижнее подчеркивание). _ – ярлык для обозначения имени, если оно используется однократно, и на него не потребуется ссылаться в других местах кода. Зачем тратить время для придумывания имени!? Например, следующие записи эквивалентны:

```
(x) => x * 3
```

```
(_) => _ * 3
```

Еще один ярлык – обращение к столбцу без упоминания имени таблицы. [FieldName] вместо _[FieldName], где _ – имя таблицы. Каждое из приведенных ниже выражений эквивалентно. Каждое следующее использует более лаконичный синтаксис, что облегчает его чтение.

```
(_) => _[Col1] + _[Col2]
```

```
each _[Col1] + _[Col2]
```

```
each [Col1] + [Col2]
```

Если из контекста понятно, о какой таблице идет речь, PQ не требует упоминания имени таблицы перед именем столбца. Но (внимание!) упоминание имени таблицы перед номером строки всё еще требуется. Подробнее см. Ключевое слово each в языке M Power Query.

Мы можем упростить наш пример встроенного определения функции, используя ярлыки.

Листинг 4

```
let
  Source = #table( {"Col1", "Col2"}, { {1, 2}, {3, 4} } ),
  AddColumn = Table.AddColumn(Source, "RowTotal", each [Col1] + [Col2])
in
  AddColumn
```

Почему выбрали имя *each*? Я предполагаю, что название связано с тем, что *each* используется для упрощения определения функции, где функция будет вызываться один раз для каждого элемента в наборе входных данных. Например, *Table.AddColumn* вызывает функцию с одним аргументом один раз для каждой строки таблицы. Независимо от этимологии, *each* можно использовать всегда, когда вы определяете функцию с одним аргументом, независимо от того, будет ли она вызываться один раз для каждого элемента или нет.

Возвращаемые функции

Функции также могут возвращать функции. Использование возвращаемых функций наиболее интересно в контексте замыкания. Суть замыкания в следующем. Если какая-то функция объявлена внутри другой, то вложенная функция может знать о значениях переменных, с которыми вызвана внешняя функция. Другими словами, если какой-то код вызывает внешнюю функцию, то вложенная функция может видеть все новые значения переменных внешней функции каждый раз, когда внешняя функция получает новые параметры.

Следующий код…

```
(x) => (y) => (x * y)
```

… вызывает внешнюю функцию (x) и передает ей значение для *x*. Внутренняя функция благодаря замыканию будет знать это *x*. Когда мы вызываем внутреннюю функцию (y), нам нужно передать ей только значение для *y*. Затем внутренняя функция умножает *x* на *y*.

Например, если вызвать внешнюю функцию со значением *x* = 5, внутренняя функция ведет себя следующим образом:

```
(y) =>
  let
    x = 5
  in
    x * y
```

Рассмотри менее тривиальный пример. Библиотечная функция List.Transform ожидает два аргумента: исходный список (*list*), и функцию (*transform*), которая будет вызываться один раз для каждого элемента списка, чтобы вычислить для этого элемента новое значение. Функции *transform* будет передано значение текущего элемента списка в качестве аргумента.

```
List.Transform(list as list, transform as function) as list
```

Допустим, мы хотим преобразовать список числовых значений, уменьшив их на фиксированный процент. Один из способов сделать это – определить функцию, которая принимает процент скидки и возвращает функцию, которая в свою очередь примет значение из списка и уменьшит его на запомненный процент скидки. Эта возвращаемая функция будет передана в *List.Transform*:

Листинг 5

```
let
   Source = { 1, 2, 3, 4, 5 },
   CalculatorGenerator = (discountPercentage) =>
      (value) => (1 - discountPercentage) * value,
   Result = List.Transform(Source, CalculatorGenerator(0.5))
in
   Result
```

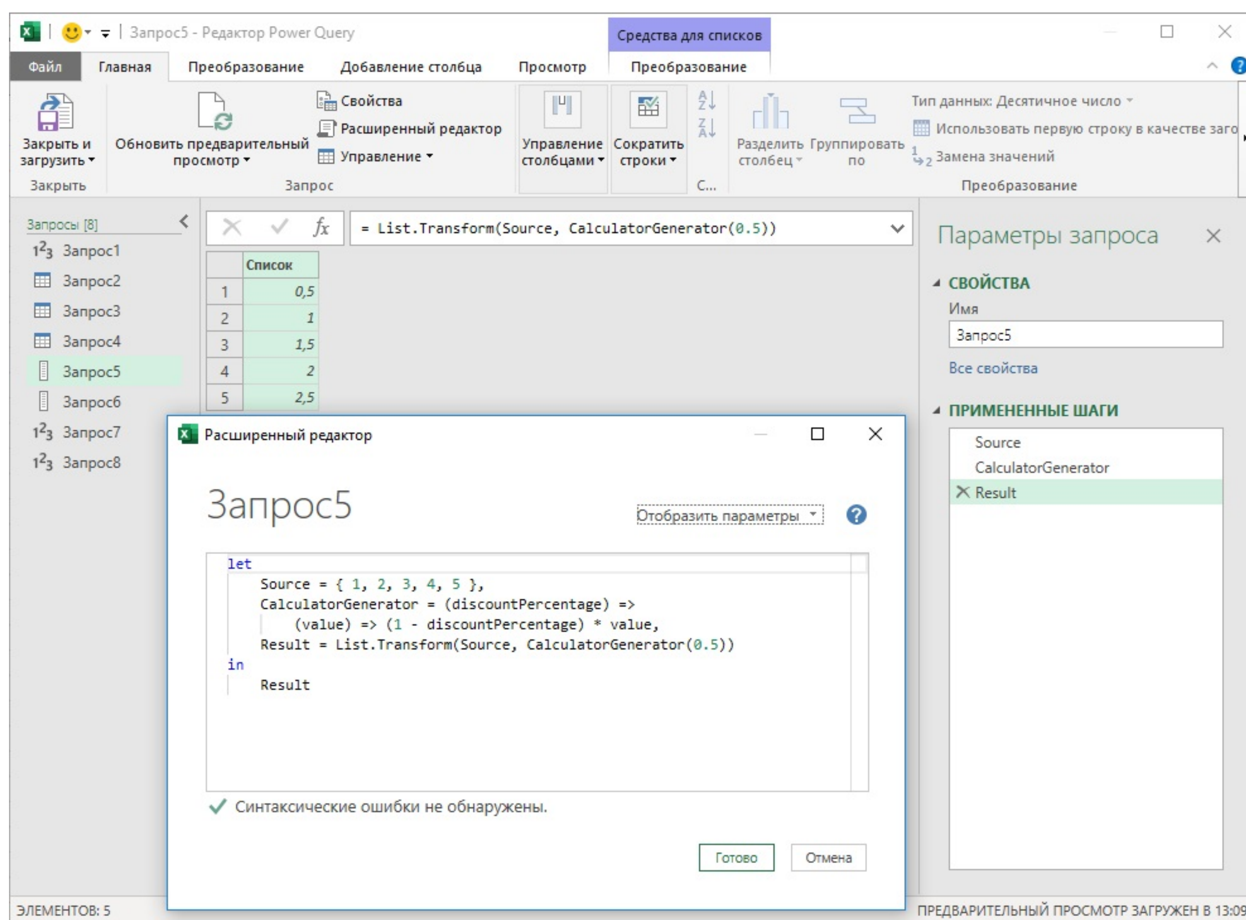

Рис. 2. Замыкание на примере внешней функции *List.Transform* и внутренней – *CalculatorGenerator*

Давайте подробнее обсудим, как работает *CalculatorGenerator* в этом коде. Чтобы прояснить, что именно передает *List.Transform* на вход функции *CalculatorGenerator*, изменим логику работы *CalculatorGenerator*. Пусть *CalculatorGenerator* просто вернет параметр, полученный на вход от *List.Transform*:

Листинг 6

```
let
   Source = { 1, 2, 3, 4, 5 },
   CalculatorGenerator = (parameter) => parameter,
   Result = List.Transform(Source, CalculatorGenerator)
```

in
  Result

Вот, что мы видим:

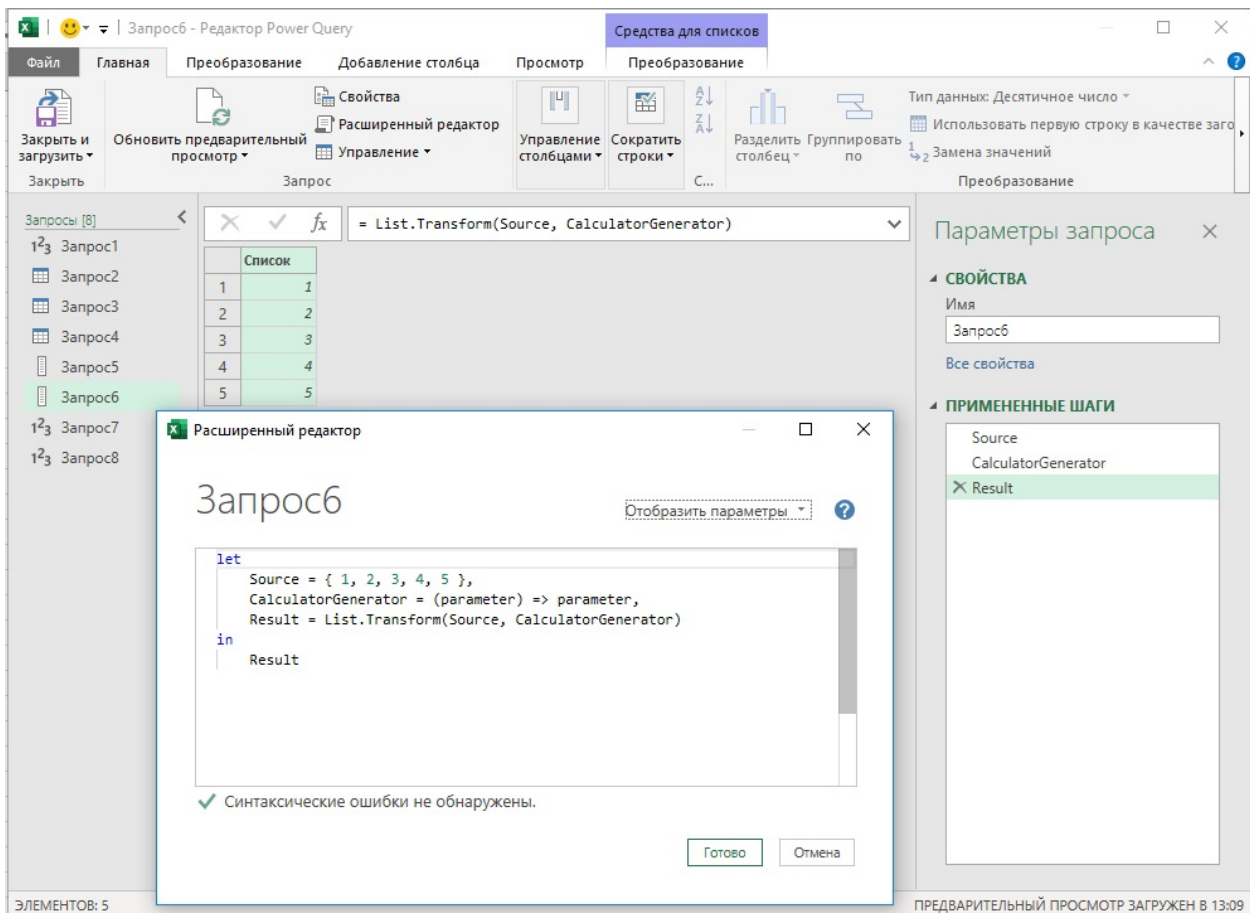

Рис. 3. Что подается на вход *CalculatorGenerator*

На вход *CalculatorGenerator* передается столбец { 1, 2, 3, 4, 5 }

Итак, в листинге 5 мы явно вызвали *CalculatorGenerator* с аргументом 0.5, а *List.Transform* вдобавок неявно подал на вход *CalculatorGenerator* список чисел от 1 до 5, который нужен для работы внутренней функции (value).

Мы можем эту логику работы увидеть в явном виде, если используем *CalculatorGenerator* для обработки одного значения (а не списка):

Листинг 7

```
let
  Source = { 1, 2, 3, 4, 5 },
  CalculatorGenerator = (discountPercentage) =>
    (value) => (1 - discountPercentage) * value,
  result = CalculatorGenerator(0.75)(5)
in
  result
```

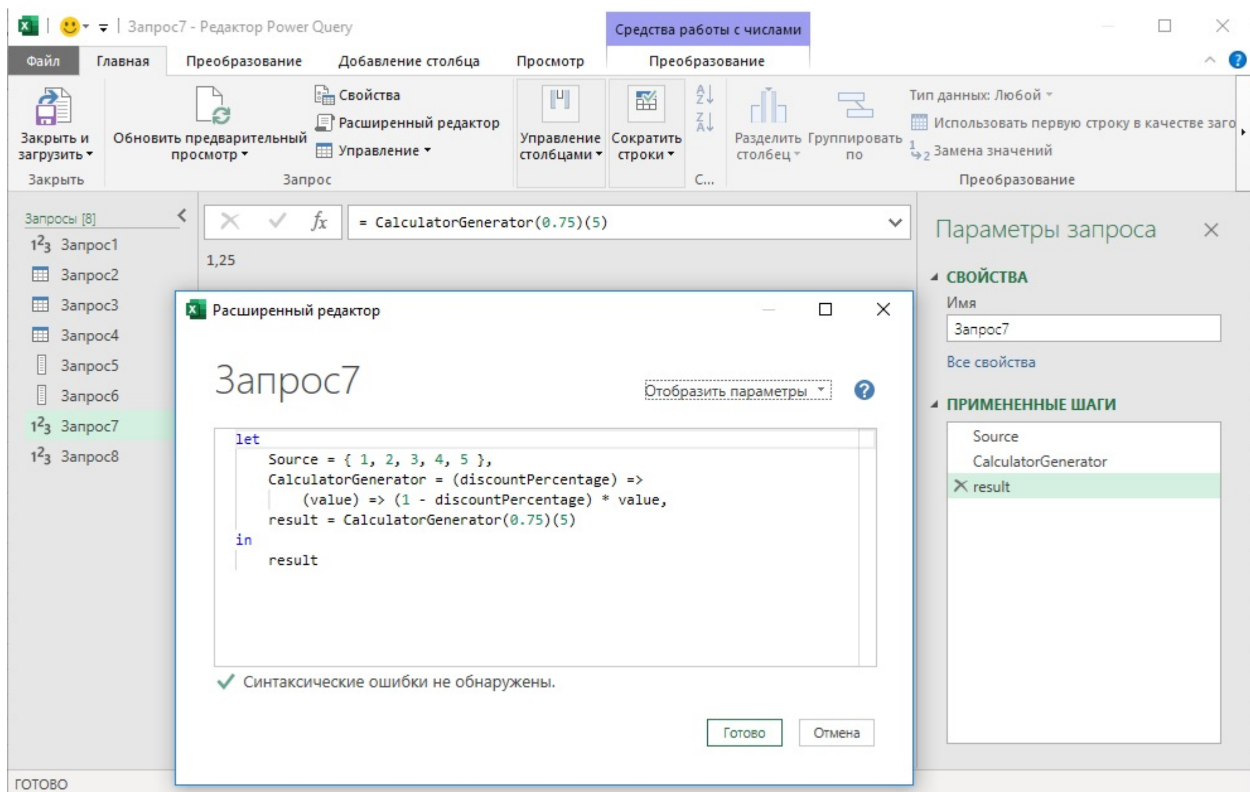

Рис. 4. Синтаксис при явной передаче двух параметров внешней и внутренней функциям

Обратите внимание на синтаксис: CalculatorGenerator(0.75)(5). Функции подаются два параметра, но не как обычно, через запятую внутри скобок, а двумя парами скобок.

Рекурсивные функции

Чтобы функция могла ссылаться на свое собственное имя изнутри самой себя, просто добавьте к ссылке префикс @:

Листинг 8

```
let
   SumConsecutive = (x) => if x <= 0 then 0 else x + @SumConsecutive(x - 1),
   Result = SumConsecutive(4)
in
   Result
```

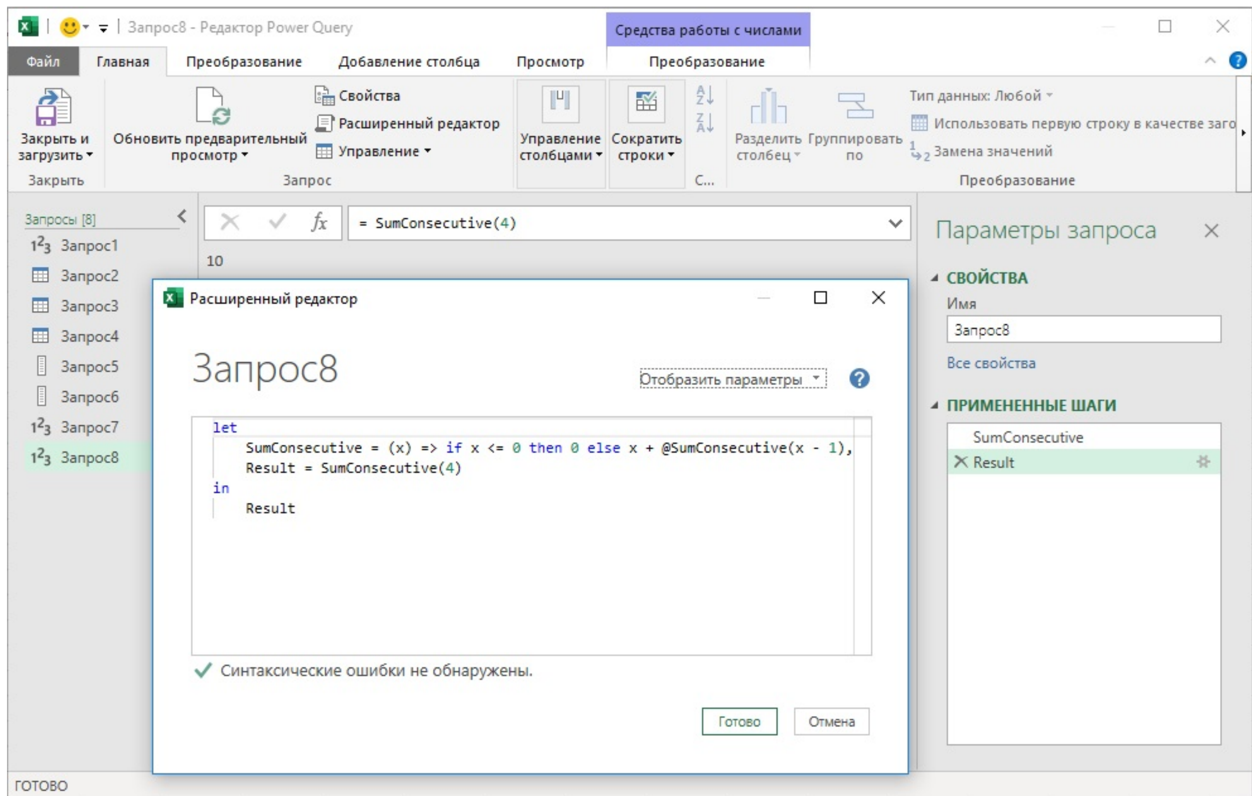

Рис. 5. Рекурсивный вызов функции