Функции в языке M Power Query

Это перевод статьи Александра Иванова, опубликованной в блоге 4pbi. В статье рассмотрено, как определять, вызывать, передавать и возвращать функции в языке M. Что такое замыкание (closure), как вызывать рекурсивные функции.

Как правило, функция – это фрагмент программного кода, который имеет имя, принимает значения в качестве параметров и возвращает результат. Удобно определять фрагмент кода как функцию, когда вам нужно выполнять одни и те же вычисления много раз в разных частях программы. Функцию не обязательно писать самому. В языке M масса встроенных функций, и вам не нужно знать, что происходит внутри функции. Вам лишь нужно понимать, какие параметры передать функции и какой результат будет возвращен. Итак, функция – черный ящик: мы передаем параметр(ы) и получаем результат.

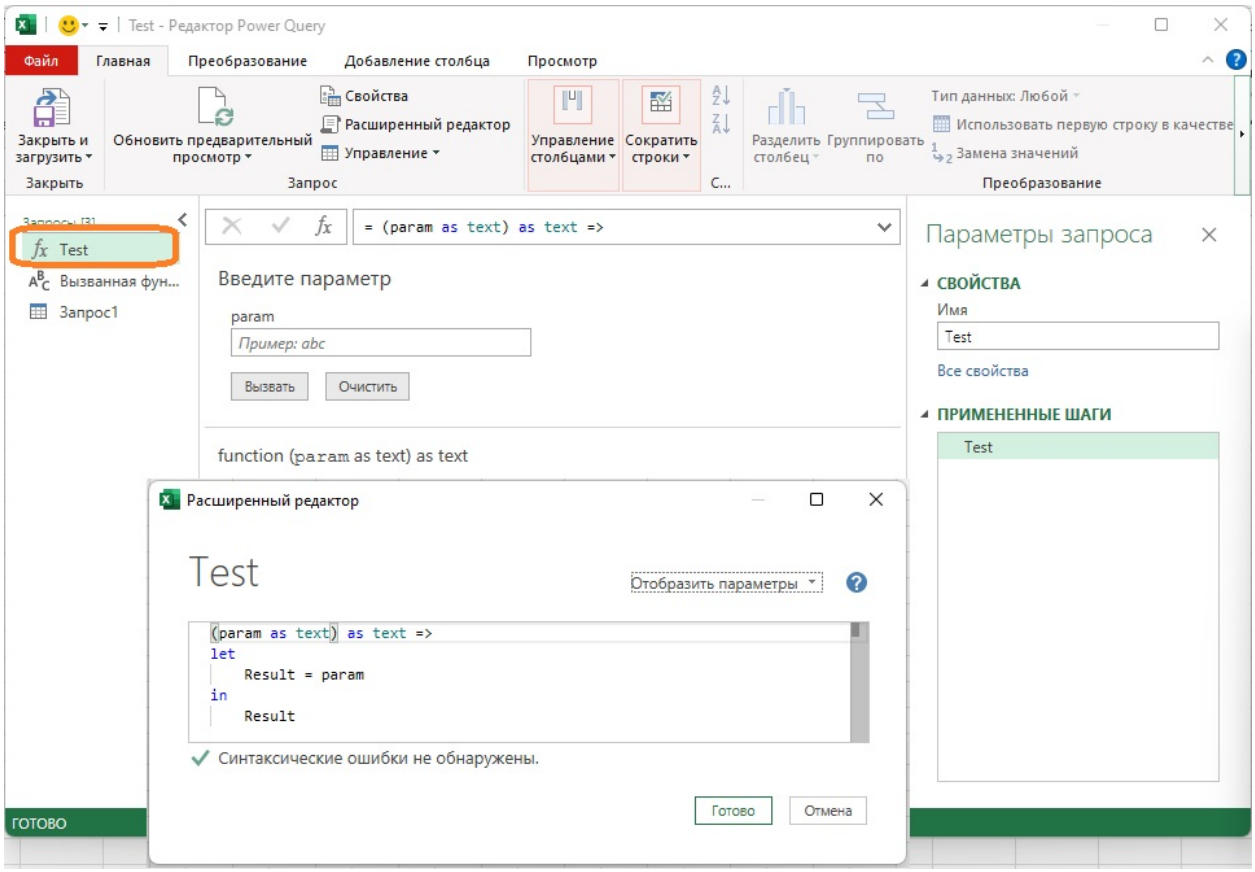

Рис. 1. Код функции Test

Синтаксис функций

Вот пример функции, которая просто возвращает переданный ей параметр (это выглядит как пустой черный ящик):

```
function Test (string param): string {
        return param;
}
```

Это код в стиле Java. На мой взгляд, его легче понять начинающим. Мы видим следующие элементы:

function – объявление функции.

Test – имя функции. Мы можем использовать это имя для вызова функции в других местах кода.

(string param) означает, что функция принимает один параметр строкового типа.

: string означает, что функция возвращает результат строкового типа.

Внутри {} – тело функции (содержимое черного ящика). Поскольку наша функция ничего не делает, мы просто возвращаем параметр, который был передан функции – return param;

Вызов этой функции в Java выглядит следующим образом:

a = Test ("Hello, world!");

После выполнения кода в переменной *a* сохраняется значение Hello, world!

Функция на языке M Power Query

Давайте напишем этот пример на языке M:

```
Test = (param as text) as text =>
let
   Result = param
in
   Result
```

Функция на языке M в целом выглядят так же, как и в других языках, но есть и некоторые синтаксические различия. Выражение = () => означает, что это функция. В скобках мы должны описать тип параметра. В нашем примере это текст. После закрывающей скобки мы указываем тип значения, возвращаемого функцией. Опять же, это текст. Между let и in мы пишем тело функции (содержимое черного ящика). После in мы указываем, какой литерал должен быть возвращен, как результат функции. На самом деле in означает out (Microsoft шутит 😊).

Вызов функции на языке M:

a = Test ("Hello, world!")

Если функция *Test* используется внутри запроса, как его часть, то, приведенный выше код, будет работать. Если функция *Test* создана как функция верхнего уровня, и на нее будут ссылаться в других запросах, следует изменить код на…

```
(param as text) as text =>
let
   Result = param
in
   Result
```

… а сам запрос, содержащий функцию, переименовать в *Test* (см. рис. 1).

> Вы можете экспериментировать с кодом в приложенном Excel-файле. Имена запросов видны на рисунках в верхней строке редактора PQ.

Функциональный и императивный типы языков программирования

Язык M является функциональным языком, поэтому переменная *a* не хранит никакого значения. Более того, это не переменная в классическом понимании, а просто литерал, который ссылается на другую функцию. В общем случае код на языке M ничего не хранит и не использует стек, а просто вычисляет выражение одно за другим по ссылкам. Есть два важных различия с императивными языками:

- Все вычисления выполняются языком M от конца кода. Это означает, что M принимает конечный литерал (указанный после in) и вычисляет его. Если этот литерал ссылается на другой, тот также находится и вычисляется. Таким образом, M идет от литерала к литералу, пока не достигнет простейших выражений, таких как x = 1 или s = "строка", которые не имеют вложенных литералов.
- Исходя из вышесказанного, M – это ленивый язык. Это означает, что если какое-то выражение не упоминается в конечном результате, то оно не будет вычислено, M просто "не видит" никаких выражений, которые не влияют на результат (прямо или косвенно).

Начнем с таблицы

Для начала создадим таблицу:

```
let
   t = #table(
      {"Name", "Salary"},
      {
         {"Alex", 100},
```

```
        {"Steve", 150},
        {"Bill", 200},
        {"John", 250},
        {"Bruce", 300}
      }
  )
in
  t
```

Мы использовали стандартную функцию конструктора таблиц #table. Она принимает 2 параметра: список имен столбцов {"Name", "Salary"}, список записей – строк таблицы. Для каждой строки два значения:

```
      {
        {"Alex", 100},
        {"Steve", 150},
        {"Bill", 200},
        {"John", 250},
        {"Bruce", 300}
      }
```

Внешние фигурные скобки говорят о том, что второй аргумент – список. Он содержит 5 записей, каждая из которых также представляет собой список.

Определение пользовательской функции

Теперь определим нашу собственную функцию:

```
  myFunc = () =>
    let
      res = "test"
    in
      res
```

Добавим ее код после определения таблицы t, и не забудьте поставить запятую после закрывающей скобки),

```
let
  t = #table(
    {"Name", "Salary"},
    {
      {"Alex", 100},
      {"Steve", 150},
      {"Bill", 200},
      {"John", 250},
      {"Bruce", 300}
    }
  ),
  myFunc = () =>
    let
      res = "test"
    in
      res
in
  t
```

Функция, которую мы только что определили, не принимает никаких параметров. Кроме того, мы не указали тип результата, который возвращается функцией (на самом деле это означает, что функция возвращает тип any). Наша функция просто возвращает строку *test*. В расширенном редакторе нажмите *Готово*. В окне предварительного просмотра вы увидите, что ничего не изменилось.

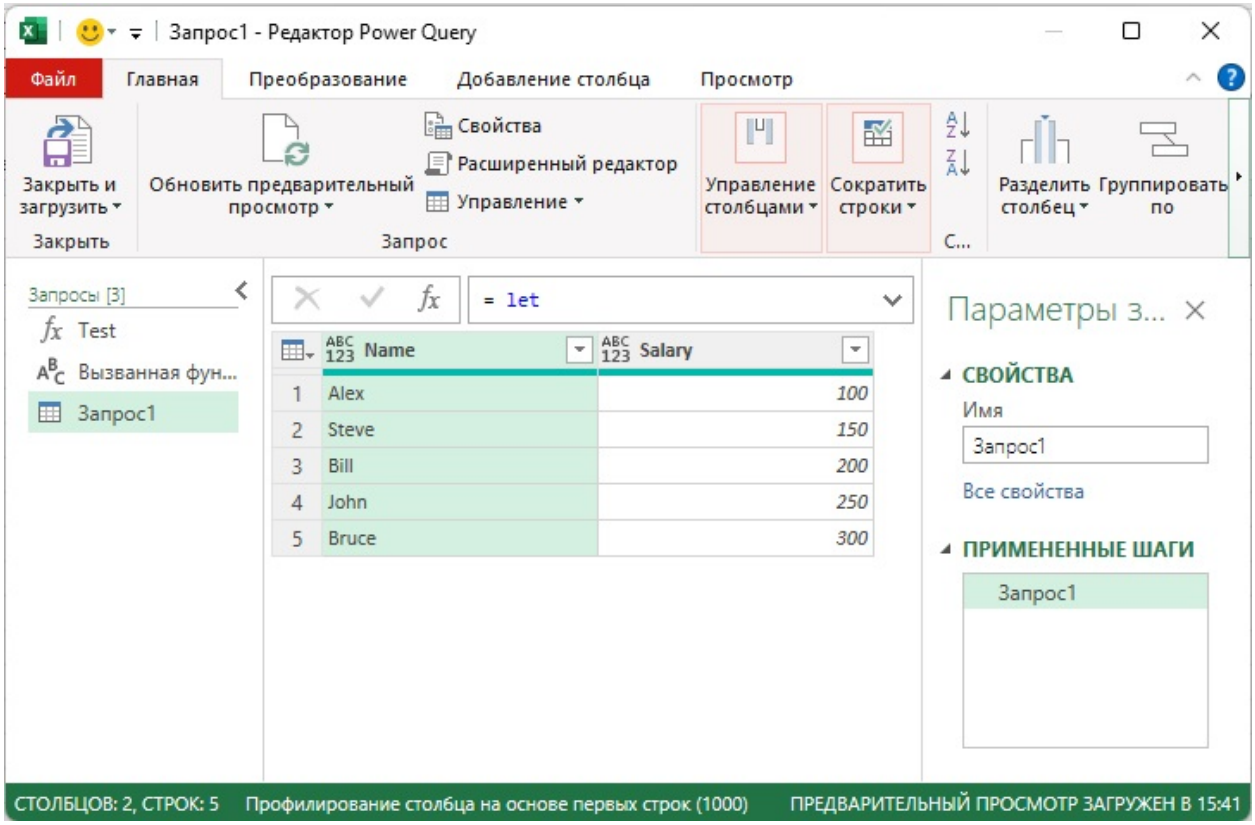

Рис. 2. Добавление функции myFunc не изменило результат

Это происходит потому, что наша функция не влияет на конечный результат, обозначенный литералом t сразу после in. Чтобы показать, как функция работает, нужно обратиться к ней, а то, что она вернет должно стать частью результата (после in), тогда *ленивый* M вычисли её.

Вызов пользовательской функции

Добавим в таблицу столбец, сгенерированный функцией myFunc. Для этого воспользуемся встроенной функцией Table.AddColumn. Она принимает 4 параметра:

Table.AddColumn(table as table, newColumnName as text, columnGenerator as function, optional columnType as nullable type) as table

Функция добавляет столбец с именем newColumnName в таблицу table. Значения для этого столбца вычисляются с помощью заданной функции выбора columnGenerator, при этом каждая строка берется в качестве входных данных. Параметр *Тип нового столбца* является необязательным.

Поставим запятую после литерала res и добавим строку

testColumnAdded = Table.AddColumn(t, "Test column", myFunc)

В итоге получим:

```
let
   t = #table(
      {"Name", "Salary"},
      {
         {"Alex", 100},
         {"Steve", 150},
         {"Bill", 200},
         {"John", 250},
         {"Bruce", 300}
      }
   ),
   myFunc = () =>
      let
```

```
        res = "test"
    in
        res,
  testColumnAdded = Table.AddColumn(t, "Test column", myFunc)
in
  testColumnAdded
```

Новый столбец отображается, но вместо значений мы видим ошибки (рис. 3). Что это значит? Нажмите в свободное пространства любой ячейки с ошибкой (но не на слово *Error*, и вы увидите описание ошибки. Функция ожидает 0 параметров, в то время как в момент вызова ей был передан 1 параметр.

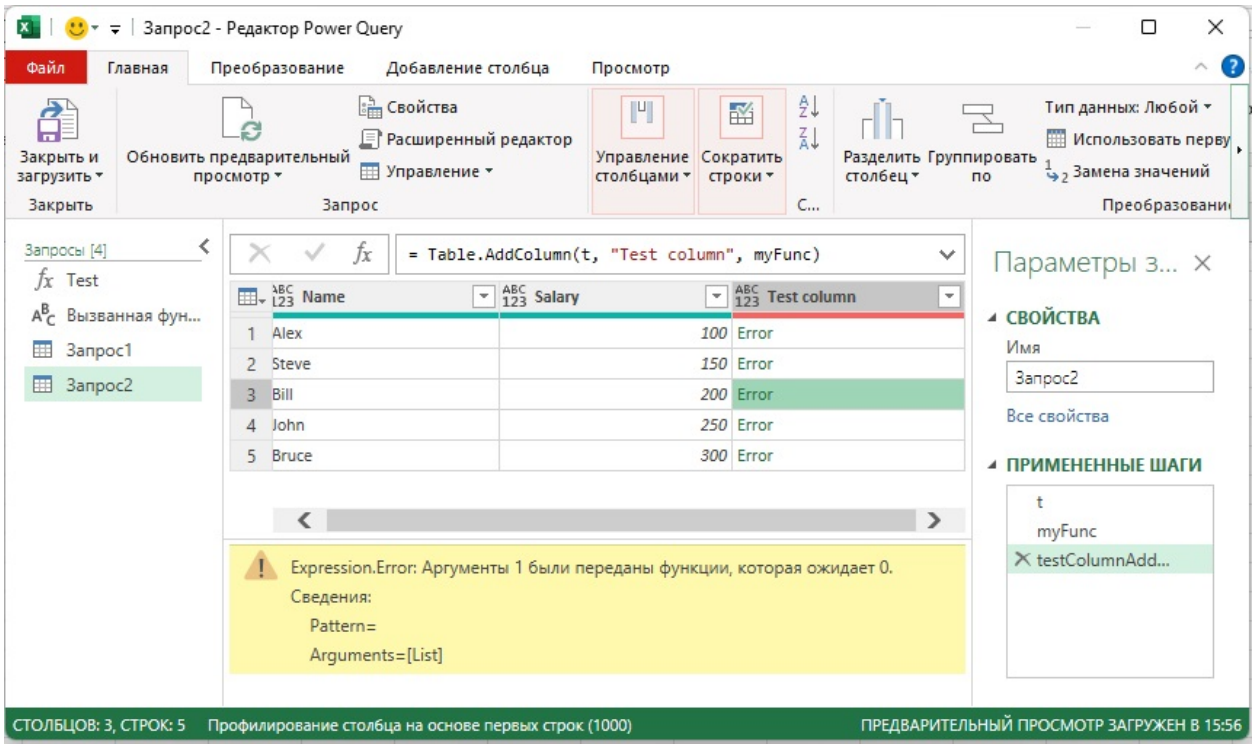

Рис. 3. Запрос вернул ошибку в новом столбце

Функция Table.AddColumn при вызове пользовательской функции myFunc передает параметр

Давайте узнаем, что это за параметр. Чтобы сделать это, мы добавим параметр в функцию *myFunc*, а затем увидим его, как новое значение столбца. Замените код функции…

```
  myFunc = (p as any) =>
    let
        res = p
    in
        res,
```

… и изучите результаты:

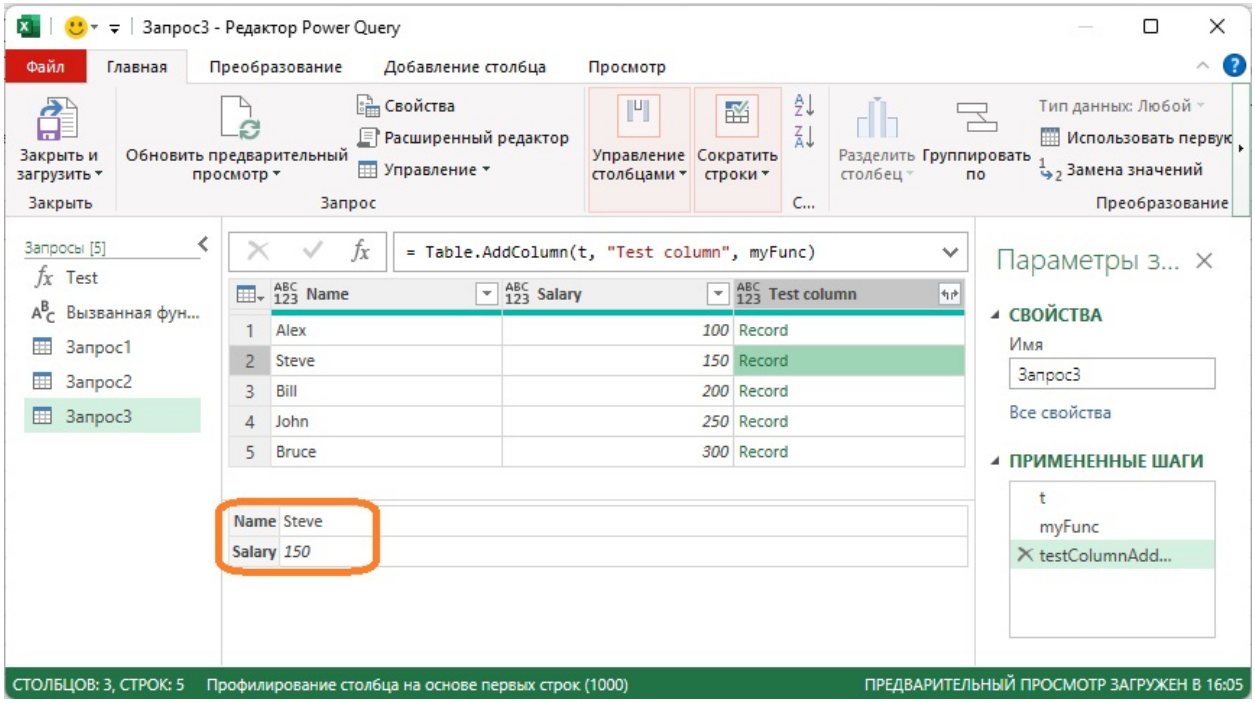

Рис. 4. Функция вернула запись

Power Query передает функции запись, содержащую значения каждого столбца текущей строки. Давайте используем это, чтобы сформировать текст для каждой строки. Что-то типа "Alex's salary is $100" (Зарплата Алекса составляет 100 долларов), "Steve's salary is $150"… Измените код функции:

```
myFunc = (p as record) as text =>
  let
    res = p[Name] & "'s salary is $" & Number.ToText(p[Salary])
  in
    res,
```

Обратите внимание, новый код содержит определение типов: параметр имеет тип record, а результатом является текст. В принципе, вы не обязаны указывать типы, тогда все параметры по умолчанию будут иметь тип any. Тем не менее, я настоятельно рекомендую всегда указывать типы, даже если вы не знаете, каков тип параметра (в этом случае укажите any). Таким образом, код становится более читаемым, и вы получаете дополнительную точку управления вашей программой на языке M.

Итак, теперь наша функция создает предложение, используя поля строки, которые передаются в качестве параметра:

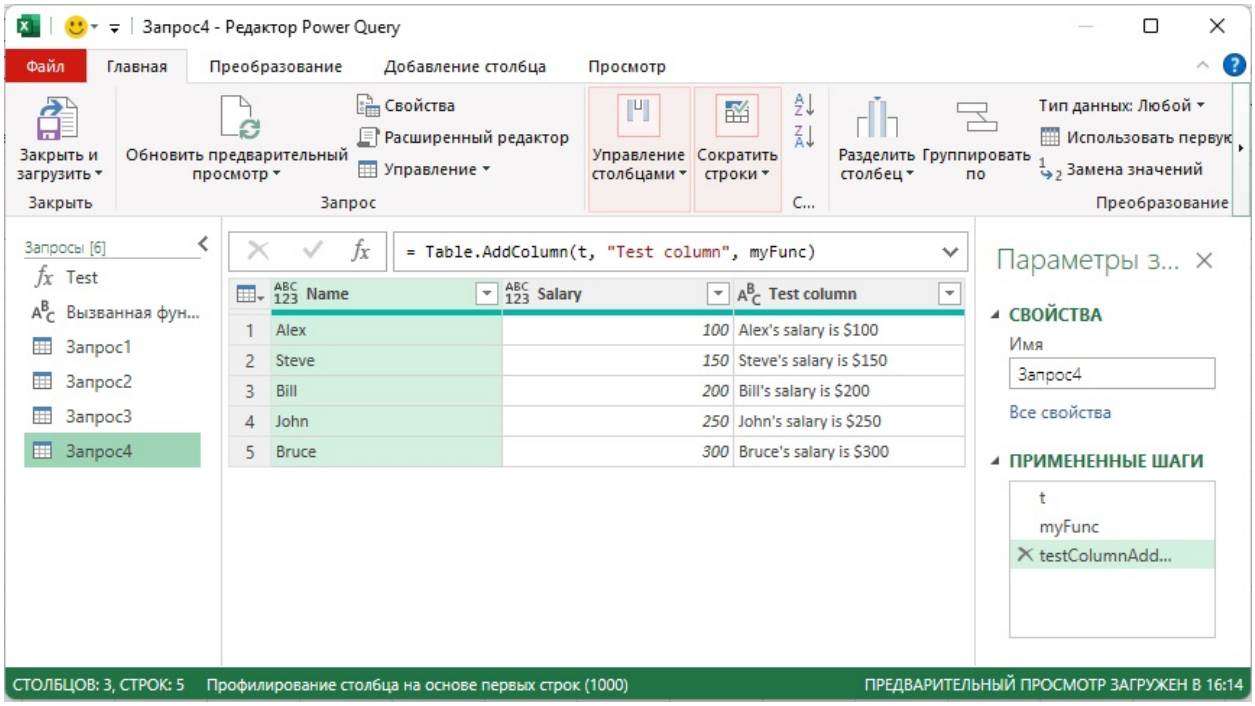

Рис. 5. Функция *myFunc* объединила данные двух столбцов по каждой строке

Функция, как результат запроса

Отлично! Мы выявили интересную возможность работы с функциями на языке M. Функция может возвращаться в качестве результата. Это дает нам мощный инструмент для написания гибкого и читаемого кода. Скажем, мы хотим увеличить заработную плату для каждого сотрудника на 10%:

```
increase = (val as number, percent as number) as number => val * (1 + percent),
```

Упс, а где же let...in? Оказывается, если функция содержит только один оператор, конструкция let...in является необязательной.

Добавим новый столбец *Increased*, который содержит новые значения заработной платы:

```
increasesSalaryAdded = Table.AddColumn(
    testColumnAdded,
    "Increased",
    each increase(_[Salary], 0.1),
    type number)
```

Обратите внимание, что мы использовали четвертый параметр, задав тип столбца *Increased* как number. Я советую вам всегда указывать тип столбца.

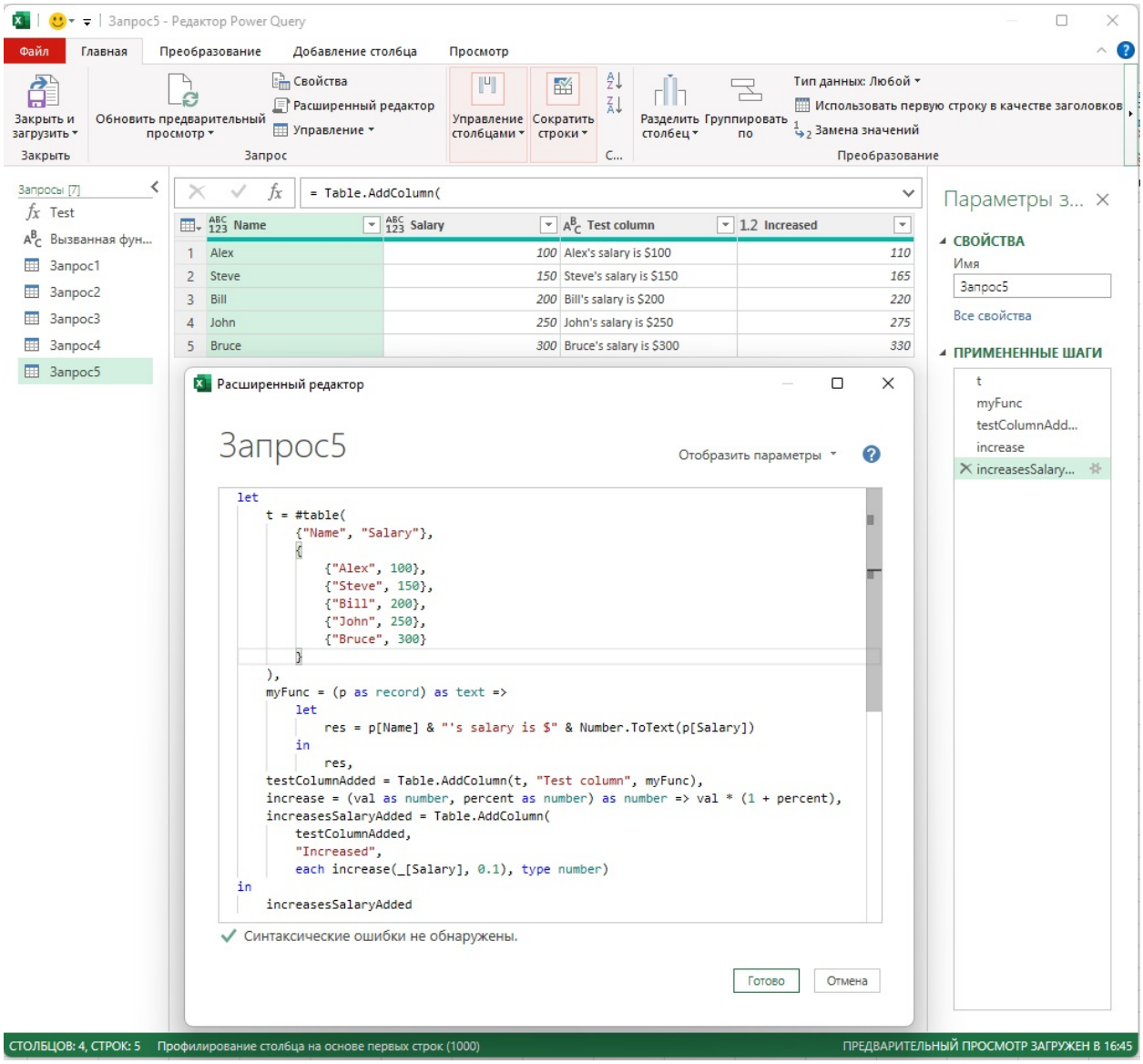

Рис. 6. Функция *increasesSalaryAdded* увеличивает зарплату на 10%

Добавим условную логику

Усложним задачу. Для сотрудников, у которых текущая зарплата меньше 250 долларов, увеличим зарплату на 10%, а для остальных – на 10 долларов. Мы можем написать что-то вроде:

```
increase = (val as number, percent as number) as number =>
    if val < 250 then val * (1 + percent) else val + 10,
```

Но я хочу продемонстрировать, как мы можем возвращать функцию в качестве результата, поэтому создадим две отдельные функции: первая увеличивает зарплату на 10%, а вторая – на 10 долларов. Затем мы напишем функцию, которая будет использовать одну из них в зависимости от текущей зарплаты сотрудника.

```
increasePercentage = (val as number, percent as number) as number => val * (1 + percent),

increaseAbsolutely = (val as number) as number => val + 10,

increase = (val as number, percent as number) => if val < 250 then increasePercentage else
increaseAbsolutely,
```

А также изменим финальную функцию кода:

```
increaseFuncAdded = Table.AddColumn(
    testColumnAdded,
    "Increase Function",
    each increase(_[Salary], 0),
```

type function)

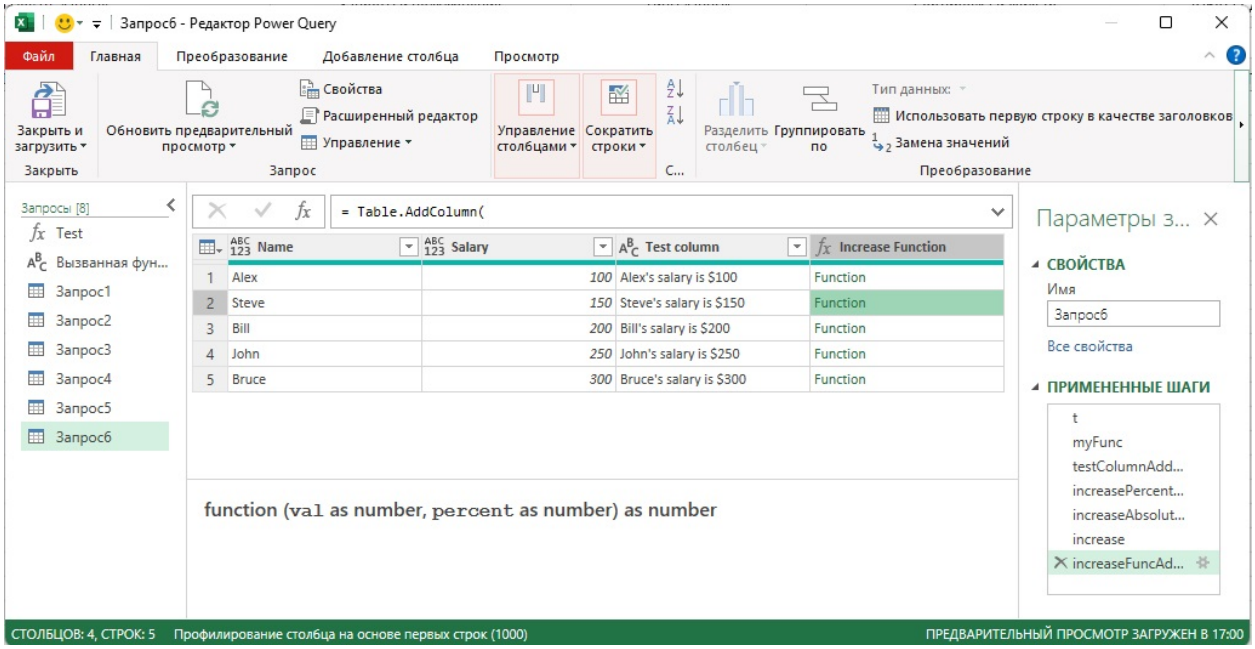

Рис. 7. Новый столбец возвращает функции

Обратите внимание, тип нового столбца – *функция*. Если вы нажмете на разные ячейки этого столбца, вы увидите, что функция отличается для сотрудников в зависимости от их заработной платы.

Осталось вызвать эти функции. Добавим следующий фрагмент в конце кода:

increasedSalaryAdded = Table.AddColumn(increaseFuncAdded, "Increased Salary",
each [Increase Function](_[Salary], 0.1), type number)

Не забывайте менять результат запроса – имя литерала, возвращаемого после строки in.

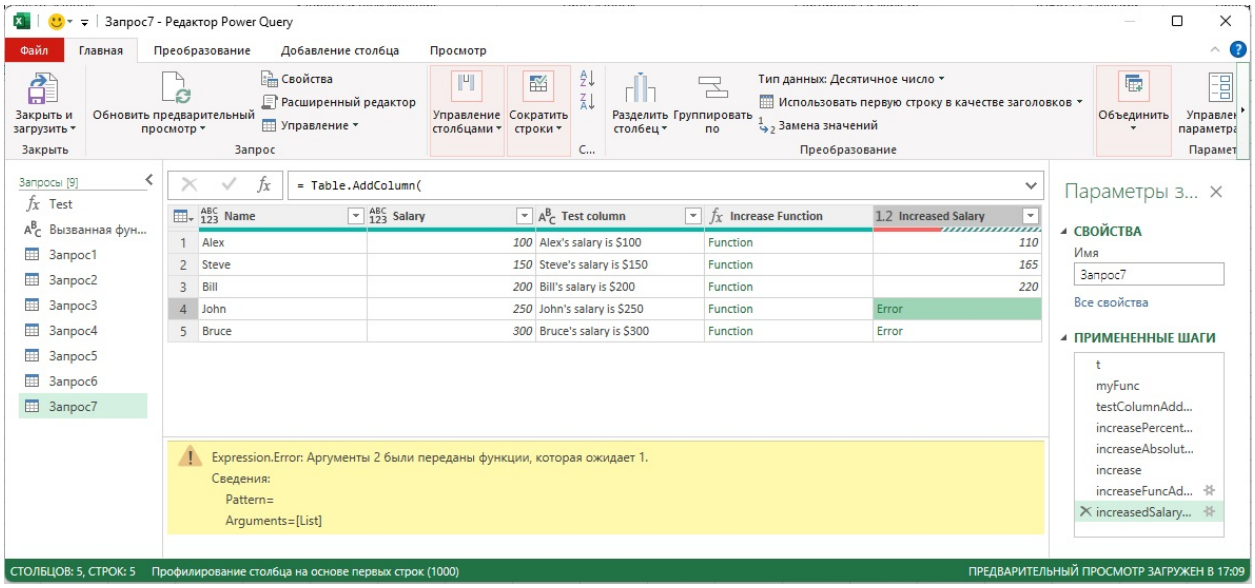

Рис. 8. Для значений зарплаты более 250 функция возвращает ошибку

Кликните в пустое пространство ячейки, содержащей ошибку. Функция *increaseAbsolutely* ожидает 1 параметр, в то время как при вызове функции передаются 2 параметра. Исправим это: добавим второй параметр в функцию:

increaseAbsolutely = (val as number, percent as number) as number => val + 10,

Итоговый код запроса

```
let
   t = #table(
```

```
        {"Name", "Salary"},
        {
            {"Alex", 100},
            {"Steve", 150},
            {"Bill", 200},
            {"John", 250},
            {"Bruce", 300}
        }
    ),
    myFunc = (p as record) as text =>
        let
            res = p[Name] & "'s salary is $" & Number.ToText(p[Salary])
        in
            res,
    testColumnAdded = Table.AddColumn(t, "Test column", myFunc),
    increasePercentage = (val as number, percent as number) as number => val * (1 + percent),
    increaseAbsolutely = (val as number, percent as number) as number => val + 10,
    increase = (val as number, percent as number) =>
        if val < 250
        then increasePercentage
        else increaseAbsolutely,
    increaseFuncAdded = Table.AddColumn(
        testColumnAdded,
        "Increase Function",
        each increase(_[Salary], 0),
        type function
    ),
    increasedSalaryAdded = Table.AddColumn(
        increaseFuncAdded,
        "Increased Salary",
        each [Increase Function](_[Salary], 0.1),
        type number
    )
in
    increasedSalaryAdded
```

Результат:

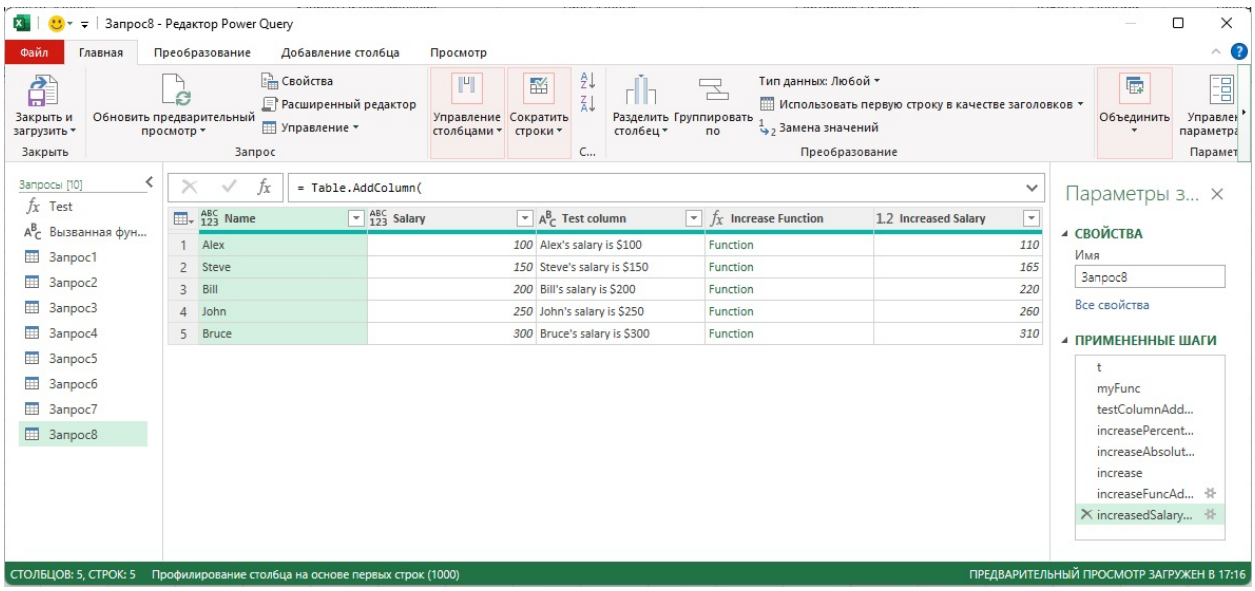

Рис. 9. Запрос отработал без ошибок

Приведенный выше пример предназначен для изучения. На самом деле, нам не нужен такой сложный код. В реальной жизни вы можете столкнуться со сложной логикой вычисления столбцов, основанной на множестве правил. Тогда, зная вещи, описанные выше, вы сможете писать гибкий и читаемый код.

Есть еще несколько важных вещей, о которых я хочу поговорить.

Определение функции налету

Если функция короткая и простая, вы можете написать ее как встроенный код, а не как отдельный фрагмент кода с именем. Это особенно полезно, если вы знаете, что в программе функция не будет использоваться несколько раз. Если использовать стандартный способ обращения к функции, то код будет подобен следующему:

```
newSalaryAdded = Table.AddColumn(increasedSalaryAdded, "New salary", newSalaryFunc, type text)
```

Где *newSalaryFunc* – функция, определенная в другом месте кода. Поскольку она относительно простая, мы можем вписать ее код внутрь функции Table.AddColumn. Это избавит нас от использования имени функции newSalaryFunc:

```
newSalaryAdded = Table.AddColumn(
    increasedSalaryAdded,
    "New salary",
    (row) =>
      let
        res = "New " &  row[Name] & "'s salary is $" & Number.ToText(row[Increased Salary])
      in
        res,
    type text
)
```

И, поскольку функция состоит из одного выражения, мы можем пропустить let...in и сделать код еще проще:

```
newSalaryAdded = Table.AddColumn(
    increasedSalaryAdded,
    "New salary",
    (row) => "New " &  row[Name] & "'s salary is $" & Number.ToText(row[Increased Salary]),
    type text
)
```

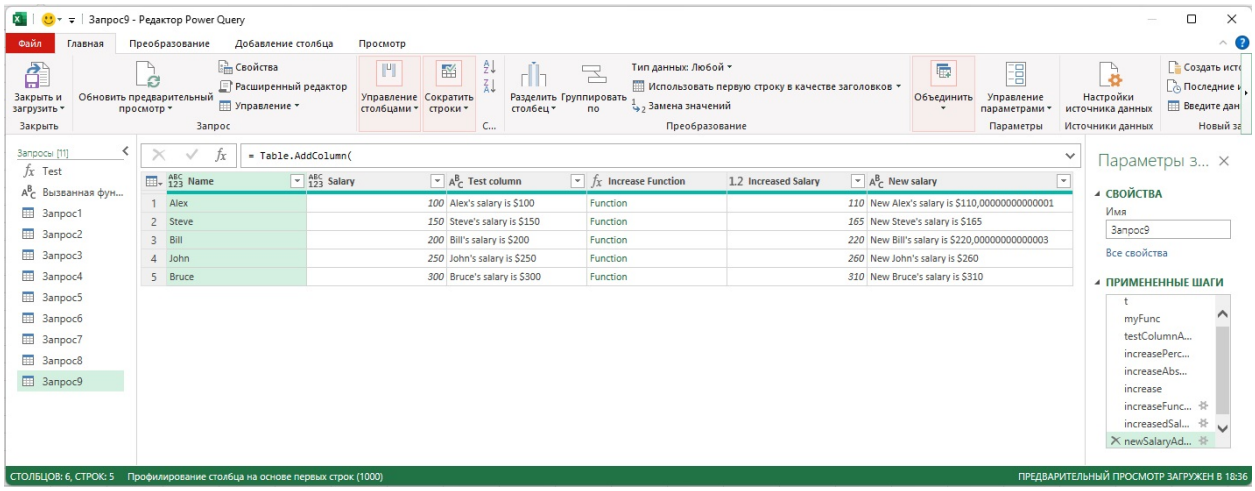

Рис. 10. Добавлен столбец *New salary*

Упс, у Alex и Bill много чисел после десятичного разделителя. Округлим новые значения заработной платы с помощью функции Number.Round. Изменим код функции, которая вычисляет процентное увеличение:

```
increasePercentage = (val as number, percent as number) as number =>
```

Number.Round(val * (1 + percent)),

Теперь результат выглядит лучше:

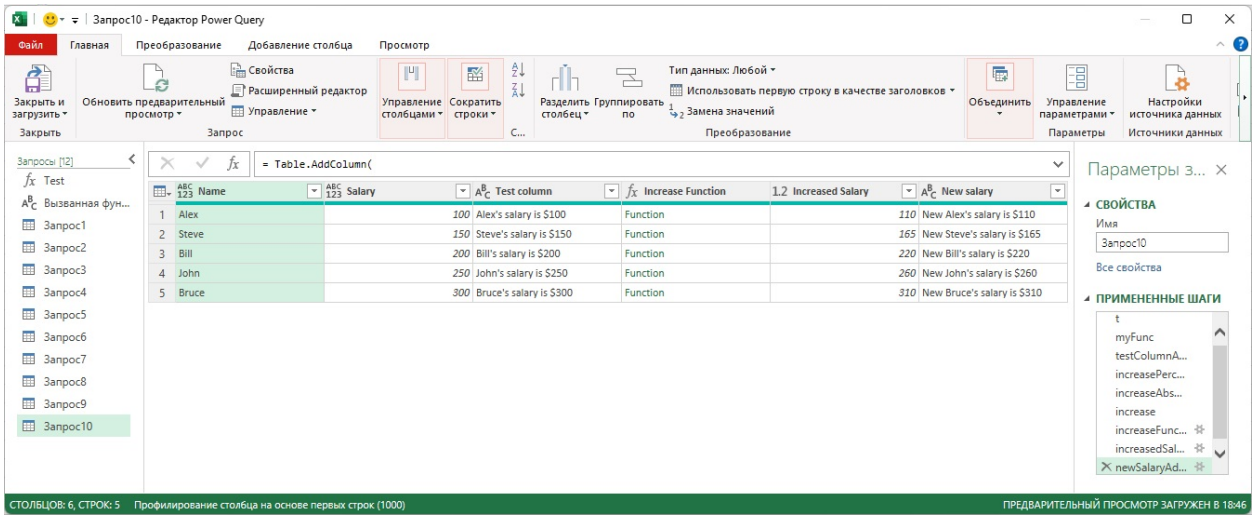

Рис. 11. Добавлено округление числовых значений в столбце *New salary*

Ключевое слово each

Мы можем написать нашу функцию еще короче. Для этого в языке M есть специальное ключевое слово *each*, которое на самом деле является синтаксическим сахаром для встроенного определения функции. Использование *each* не изменяет работу кода, но делает его более читаемым и лаконичным. Вместо…

```
(row) => "New " &  row[Name] & "'s salary is $" & Number.ToText(row[Increased Salary]),
```

… можно использовать…

```
each "New " & _[Name] & "'s salary is $" & Number.ToText(_[Increased Salary]),
```

each означает «применить следующую функцию к каждому элементу». До этого элемента можно добраться с помощью символа _. В нашем примере мы добавляем столбец в таблицу и функция Table.AddColumn передает строку в качестве параметра встроенной функции. До использования *each* эта встроенная функция имела имя (row). Итак, в нашем случае слово *each* означает «применить следующую за словом *each* функцию к каждой строке». Поскольку строка на самом деле является записью, мы можем получить любое поле строки, используя конструкцию _[FieldName]. Исходя из сказанного, _ [Name] – это имя сотрудника, а _ [Increased Salary] – его новая зарплата.

Мы можем сделать наше встроенное определение функции еще короче! Выше мы говорили, что символ _ не является обязательным, поэтому наша функция может выглядеть так:

```
each "New " & [Name] & "'s salary is $" & Number.ToText([Increased Salary]),
```

Подробнее см. Ключевое слово *each* в языке M Power Query.

Замыкание в языке M

Если какая-то функция объявлена внутри другой, то вложенная функция может знать о значениях переменных и определениях во внешней функции. 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. Поясним на примере.

Допустим, мы хотим выплатить премию сотрудникам, но расчета бонуса довольно сложен:

- если новая зарплата сотрудника < 150 долларов, бонус равен половине старой зарплаты;
- если новая зарплата сотрудника ≥ 150 долларов, то нам нужно оценить разницу между новой зарплатой и старой зарплатой, и, если разница ≥ 15 долларов, то сотрудник получает 5 долларов, в противном случае он получает 10 долларов.

Чтобы реализовать приведенную выше логику, мы пишем что-то вроде:

```
bonus = (newSalary as number, oldSalary as number) as number =>
    if newSalary < 150 then Number.Round(oldSalary / 2)
    else
      if newSalary - oldSalary >= 15 then 5
      else 10,
```

И добавляем столбец с бонусом:

```
  bonusAdded = Table.AddColumn(
    newSalaryAdded,
    "Bonus",
    each bonus([Increased Salary], [Salary]),
    type number
  )
```

Результат должен быть следующим:

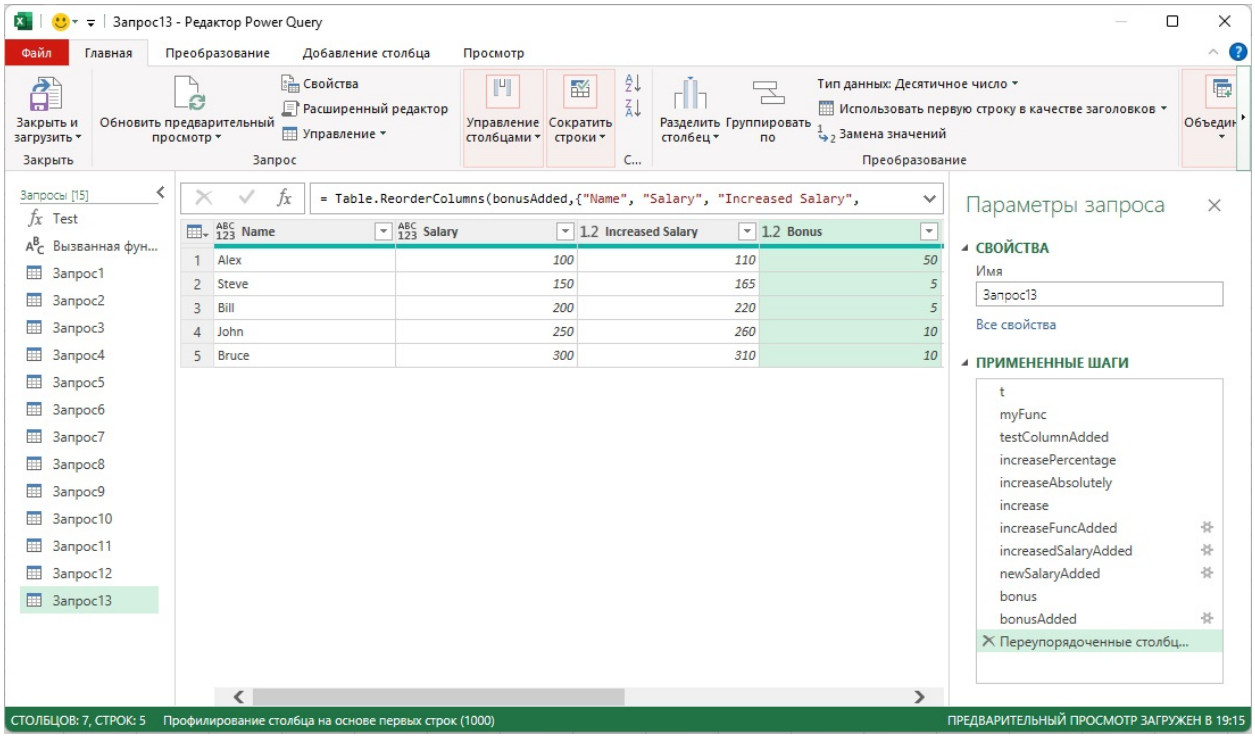

Рис. 12. Расчет бонуса

Значения в столбце *Bonus* отвечают нашей логике начисления бонуса.

Вот как можно реализовать эту логику, используя замыкание на языке M:

```
  bonus = (newSalary as number) as function =>
    if newSalary < 150 then
      (oldSalary as number) as number => Number.Round(oldSalary / 2)
    else (oldSalary as number) as number =>
      if newSalary - oldSalary >= 15 then 5 else 10,
```

Обратите внимание, приведенная выше функция *bonus* возвращает функцию, зависящую от значения новой зарплаты, которая ей передается. Наиболее интересна следующая часть кода:

```
    else (oldSalary as number) as number =>
      if newSalary - oldSalary >= 15 then 5 else 10,
```

Внутренняя функция => if newSalary – oldSalary >= 15, то 5 else 10 *видит* текущее значение переменной *newSalary*, которая на самом деле является параметром внешней функции. Это называется замыканием и дает гибкость для написания функций на языке M.

Добавим вызов функции *bonus*:

```
  bonusAdded = Table.AddColumn(
```

```
    newSalaryAdded,
    "Bonus",
    each bonus([Increased Salary])([Salary]),
    type number
  ),
```

Обратите внимание…

```
    each bonus2([Increased Salary])([Salary]),
```

… сначала передается новая заплата, а затем старая.

Естественно, результат не отличается от предыдущего:

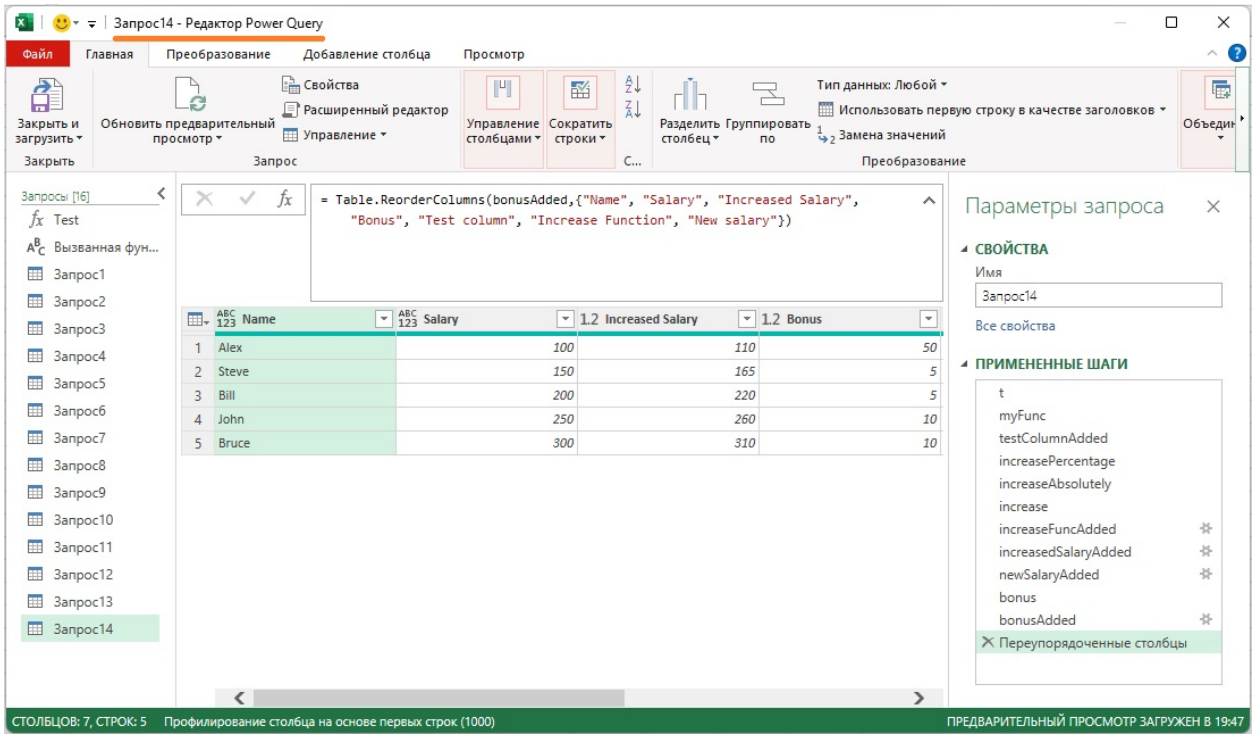

Рис. 13. Расчет бонуса с использованием замыкания

Рекурсивный вызов функции на языке M

Язык M позволяет вызывать функцию рекурсивно – функция может вызывать саму себя. Это может быть полезно, когда вам нужно вычислить факториал или просмотреть иерархические данные, а количество уровней иерархии неизвестно. Чтобы вызвать функцию рекурсивно, вы должны поместить символ @ перед именем функции. Вычислим факториал:

```
Factorial = (N as number) as number => if N <= 1 then 1 else N * @Factorial(N - 1),
```

Эта функция умножает переданное значение N на значение функции для N -1, пока N не достигнет 1. Следующий код сначала создает таблицу с цифрами от 1 до 5, далее определяет функцию Factorial, и, наконец, добавляет столбец со значениями факториала:

```
let
  t = #table(
    {"Number"},
    {{1},{2},{3},{4},{5}}
  ),
  Factorial = (N as number) as number =>
    if N <= 1 then 1
    else N * @Factorial(N - 1),
  result = Table.AddColumn(
    t,
    "Factorial",
    each Factorial([Number]),
```

```
        type number
    )
in
    result
```

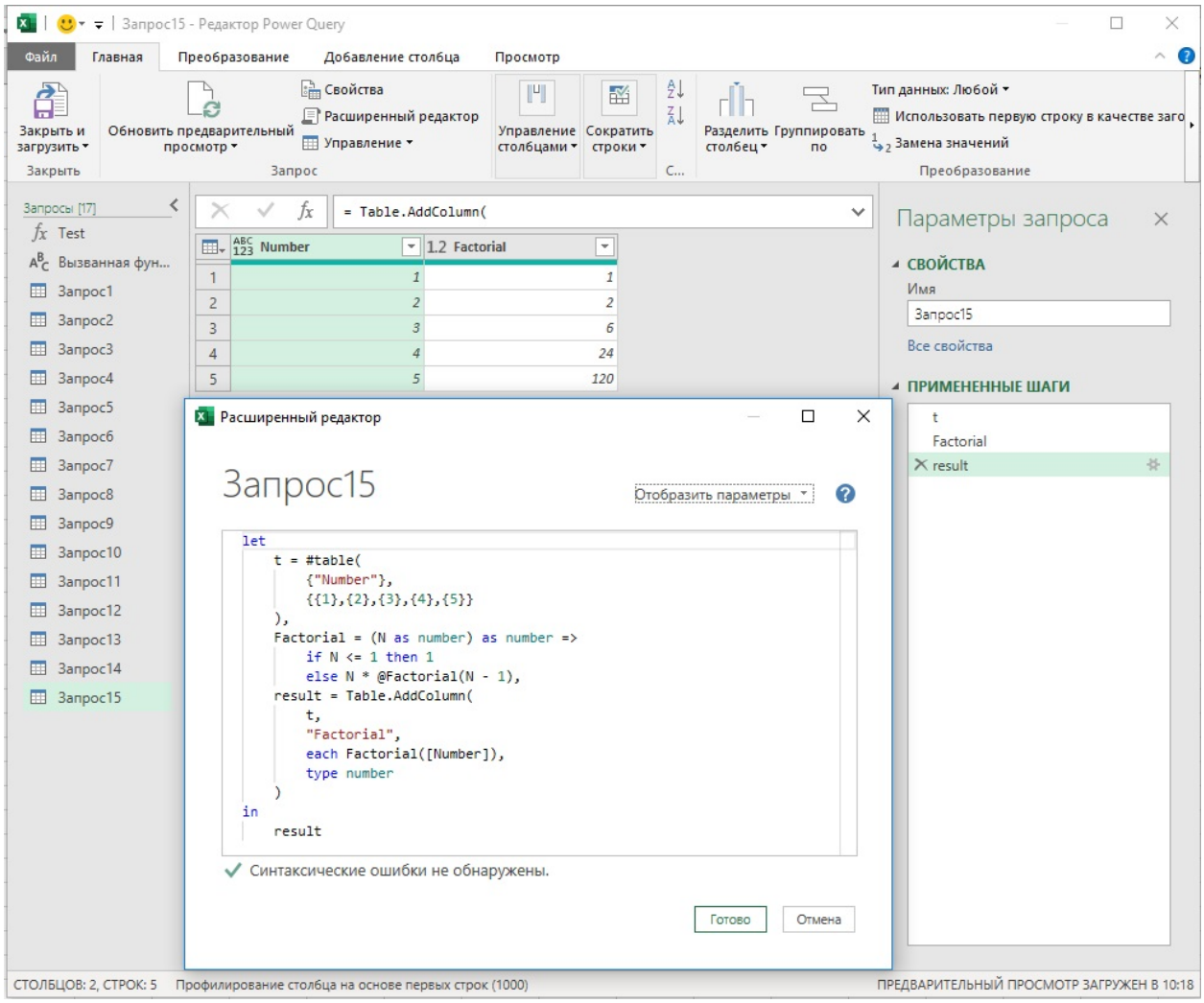

Рис. 14. Вычисление факториала через рекурсивный вызов функции