Язык M Power Query. Обработка ошибок

В M можно вызвать и обработать ошибки во время выполнения. Если из других языков программирования вы знакомы с идеей исключения, обработка ошибок Power Query отличается по крайней мере одним существенным моментом.[1]

Предыдущая заметка Следующая заметка

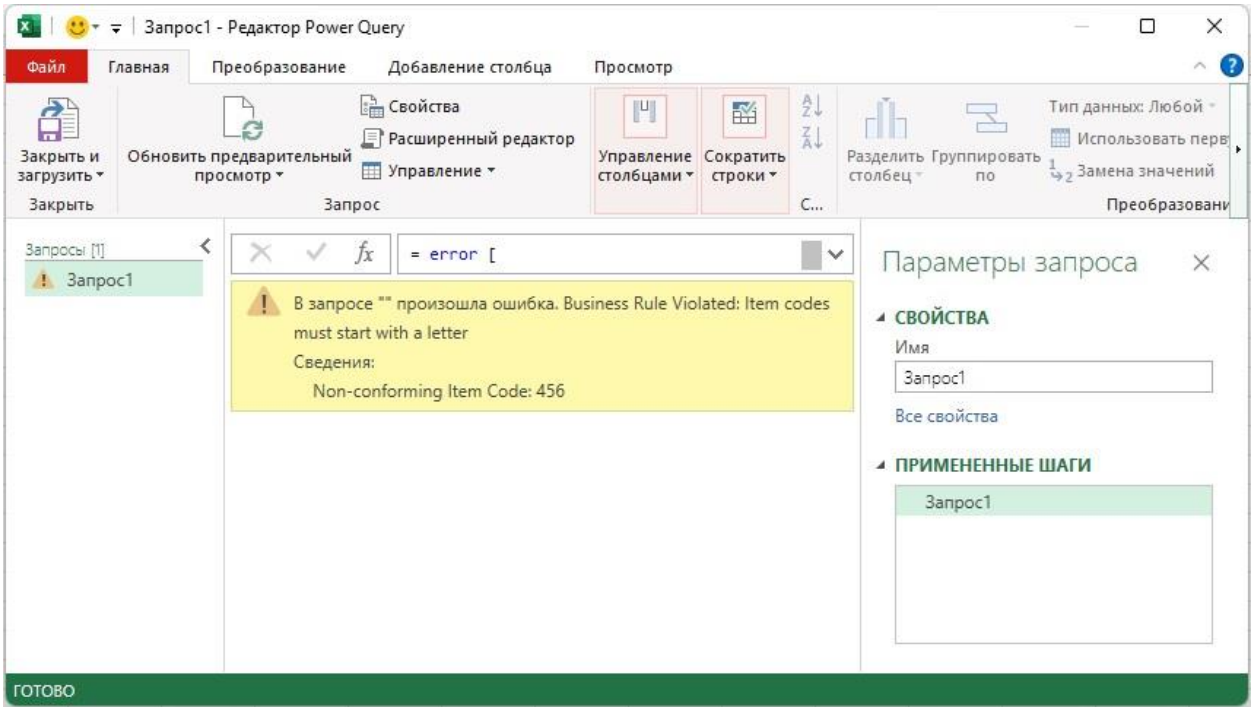

Рис. 1. Три поля записи *error*

Сообщение об ошибке

В Power Query каждое выражение должно что-то возвращать. Как правило, это значение. Но выражение также может вызвать ошибку – особый способ указать, что не получилось вернуть значение. Один из способов вызвать ошибку – создать запись с ключевым словом *error*. Такая запись имеет три поля: *причина*, *сообщение* и *подробности.* Поля с любыми другими именами будут проигнорированы.

Листинг 1[2]

```
= error [
   Reason = "Business Rule Violated",
   Message = "Item codes must start with a letter",
   Detail = "Non-conforming Item Code: 456"
]
```

Все три поля являются необязательными. Если поле *Reason* отсутствует, причина ошибки будет иметь значение по умолчанию – *Expression.Error*. Запись ошибки можно также создать с помощью функции Error.Record. В отличие от описанного выше подхода, в *Error.Record* атрибут *Reason* является обязательным.

Листинг 2

```
= error Error.Record(
   "Business Rule Violated",
   "Item codes must start with a letter",
   "Non-conforming Item Code: 456"
)
```

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 15): Error Handling. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

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

Оба приведенных выше примера приводят к эквивалентной ошибке, изображенной на рис. 1 Глядя на рисунок, видно, как три поля/параметра соотносятся с отображаемым сообщением.

Вместо записи *error* также может принимать строку. Результирующее сообщение об ошибке будет иметь значение предоставленной строки, а его причина – значение *Expression.Error*.

Листинг 3

```
= error "help!"
```

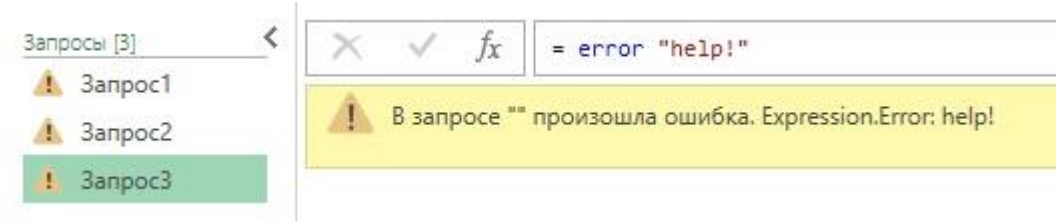

Рис. 2. Строка в *error*

Ярлык с многоточием

Существует также оператор быстрого доступа для создания ошибок, который пригодится во время разработки. Допустим, вы хотите протестировать запрос, часть кода которого еще не написана. Поскольку каждое выражение должно возвращать значение, или вызывать ошибку, вы не можете протестировать свой запрос, не поместив *что-то* в качестве заполнителя в нереализованные участки кода. Используйте оператор многоточия (...). При вызове ... выдает ошибку *Expression.Error: Значение не задано.* Вот фрагмент кода, в котором не реализована ветвь *else*:

Листинг 4

```
let
   a = 6,
   Result = if a = 5 then true else ...
in
   Result
```

Когда условие (a = 5) принимает значение *false*, вызывается "...", что приводит к ошибке. Обратите внимание, ключевое слово *error* не используется. Оператор многоточия как определяет, так и вызывает ошибку.

Особое поведение

Что именно происходит, когда возникает ошибка? Какое поведение возвращает ошибку, а не значение? Рассмотрим выражение:

```
SomeFunction(GetValue())
```

В обычных условиях сначала выполняется функция GetValue(). Затем полученное значение передается в someFunction(), которая возвращает финальный результат. Предположим, GetValue() выдает ошибку. Дальнейшее выполнение выражения прекращается. someFunction() не вызывается. Ошибка GetValue() становится итогом выражения. Такое поведение также известно, как *повышение*. Ошибка передается тому шагу, с которого была вызвана someFunction().

Дальнейшее зависит от того, предусмотрено ли в коде появление ошибки. Если да, запрос продолжит выполнение. Если нет, возникнет ошибка верхнего уровня. Запрос завершит работу и вернет значение ошибки.

Сдерживание ошибок

Если ошибка возникает в выражении, которое *что-то* определяет (поле записи, ячейку таблицы, переменную в выражении *let*, …), ошибка содержится в этом *чём-то*. Последствия ошибки ограничены этим *чем-то* и логикой, которая пытается получить доступ к значению этого *чего-то*. Ниже последствия ошибки *GetValue* содержатся в той части запроса, на которую она повлияла. Ошибка не остановила выполнение запроса. Запрос завершился успешно и вернул запись. Два поля – *FieldB* и *FieldC* – вернули ошибку, потому что они являются *чем-то*, затронутым ошибкой.

Листинг 5

```
let
```

```
  GetValue = () => error "Something bad happened!",
  DoSomething = (input) => input + 1,
  Result = [
    FieldA = 25,
    FieldB = DoSomething(GetValue),
    FieldC = FieldA + FieldB
  ]
in
  Result
```

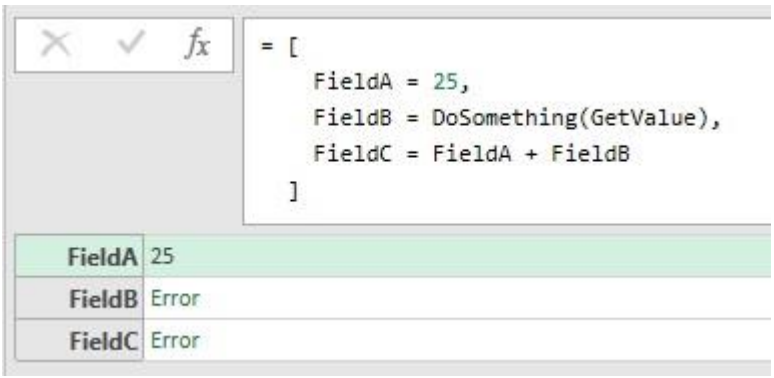

Рис. 3. Результат запроса

Сдерживание ошибок влечет за собой еще одну особенность. Ошибка сохраняется в *чём-то*, что ее содержит. Пока выполняется запрос, любая попытка получить доступ к значению этого *чего-то* приводит к повторному возникновению сохраненной ошибки. Когда происходит попытка доступа, логика, которая первоначально вызвала ошибку, не подвергается повторной оценке. Эта логика при повторном обращении могла бы вернуть допустимое значение. Но логика пропускается, и ранее сохраненная ошибка просто вызывается повторно.

Ниже функция GetDataFromWebService() вычисляется один раз, даже если к самим данным обращаются дважды. Если первое обращение вернуло ошибку, второе обращение тоже вернет ошибку, сохраненную ранее.

```
let
   Data = GetDataFromWebService() // повышенная ошибка
in
   { List.Sum(Data[Amount]), List.Max(Data[TransactionDate]) }
```

Ошибки верхнего уровня

Если ошибку не удалось сдержать, она передается из выражения в верхний уровень, как результат всего запроса. Далее в среду хоста и выполнение запроса останавливается. В следующем выражении *ничто* не содержит ошибку, поэтому её нельзя сдержать, и запрос завершается ошибкой:

Листинг 6

```
let
  GetValue= () => error "Something bad happened!",
  SomeFunction = (input) => input + 1
in
  SomeFunction(GetValue())
```

Сдерживание против исключения

Поведение Power Query по *сдерживанию* ошибок отличает его от большинства языков программирования, основанных на *исключениях*. В мире исключений ошибка автоматически распространяется вплоть до среды хоста, что приводит к завершению работы программы (если только в коде не будет предусмотрена обработка ошибок). В M ошибка локализуется, пока есть *что-то*, что ее содержит. Это позволяет успешно завершить запрос, даже если не удалось вычислить отдельные элементы.

Сдерживание ошибок – отличное поведение, учитывая цель использования M: обработка данных. Предположим, что выражение, определяющее значение столбца таблицы, содержит ошибку для одной ячейки из всей таблицы. В мире, основанном на исключениях, эта ошибка может привести к завершению всей обработки. В мире M ошибка просто влияет на эту единственную ячейку и любой код, который обращается к этой ячейке. Обработка продолжается, и будет получен результат.

На самом деле, из-за лени M, если к ячейке с ошибкой не будет обращений, то ошибка и не возникнет.

Листинг 7

```
let
  Data = #table({"Col1"}, {{"SomeValue"}, { error "bad" }})
in
  Table.RowCount(Data) // возвращает 2
```

Хотя одна ячейка и содержит ошибку, запрошенные данные (количество строк), не требуют вычисления значения ошибочной ячейки, поэтому выражение вернет значение 2.

Хотя сдерживание ошибок – отличное поведение по умолчанию, что, если оно не соответствует вашим потребностям? В частности, что делать с таблицами, если важно различать строки с ошибками и строки без ошибок? Возможно, вы не обращаетесь к содержимому строки напрямую, поэтому не делаете ничего, что могло бы вызвать распространение ошибок, но все же хотите знать, в каких строках есть ошибка, а в каких нет. Функции Table.SelectRowsWithErrors и Table.RemoveRowsWithErrors, то, что вам нужно.

Листинг 8

```
let
   Data = #table({"Col1"}, {{"SomeValue"}, { error "bad" }})
in
   [
      RowsWithErrors = Table.RowCount(Table.SelectRowsWithErrors(Data)),
      RowsWithoutErrors = Table.RowCount(Table.RemoveRowsWithErrors(Data))
   ]
```

Рис. 4. Список, содержащий количество строк с ошибками и без

Обработка ошибок

Понимая, какие операции способны вызывать ошибку, вы можете использовать ключевое слово *try*, чтобы проверить, не возвращает ли выражение ошибку. *try* используется в двух вариантах…

try ExpressionToTry otherwise FallbackExpression

try ExpressionToTry

… первый, *try with otherwise*, пытается выполнить *ExpressionToTry*. Если это выражение возвращает значение, всё Ok, переходим к следующему шагу запроса. Если выражение выдает ошибку, вычисляется выражение *otherwise* и возвращается его значение.

try Number.FromText(input) otherwise 0

Если *Number.FromText* возвращает значение, оно и будет результатом выражения. Если *Number.FromText* выдает ошибку, *try* обращается к части *otherwise*, и возвращает 0. Другими словами, если входные данные могут быть преобразованы в число, возвращается это число; в противном случае возвращается значение по умолчанию – 0.

Имейте в виду, что ошибки будут обработаны только в выражении, расположенном непосредственно справа от *try*. Если ошибку возвращает выражение *otherwise*, эта ошибка не

будет обработана предшествующим *try*. Но… поскольку *otherwise* само по себе является выражением, *try* можно поместить внутрь него, чтобы обработать ошибку, вызванную *otherwise*.

```
try GetFromPrimary()
   otherwise try GetFromSecondary()
      otherwise "Возникли проблемы с обеими серверами. Возьми отгул на остаток дня :)"
```

Проблема с конструкцией *try with otherwise* в том, что она неразборчива: любая ошибка возвращает альтернативное значение. Иногда последующие действия зависят от типа ошибки. Для этих ситуаций подойдет второй вариант – простое выражение *try*.

```
try ExpressionToTry
```

Эта форма *всегда* возвращает запись. Если выражение завершилось успешно, эта запись имеет вид:

```
[
   HasError = false,
   Value = (значение выражения ExpressionToTry)
]
```

Если выражение вызвало ошибку, возвращаемая запись выглядит следующим образом:

```
[
   HasError = true,
   Error = (запись, описывающая возникшую ошибку)
]
```

Например:

Листинг 9

```
let
  DoSomething = () => 45,
  Result = try DoSomething()
in
  Result // [HasError = false, Value = 45]
```

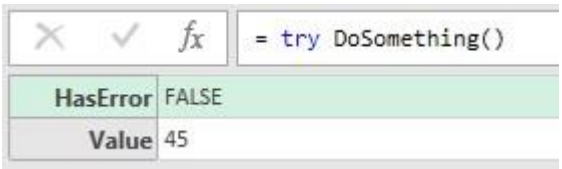

Рис. 5. Запись, возвращаемая *try*, если нет ошибки

Листинг 10

```
let
  DoSomething = () => error "bad",
  Result = try DoSomething()
in
  Result // [HasError = true, Error = [Reason = "Expression.Error", Message = "bad", Details = null]
```

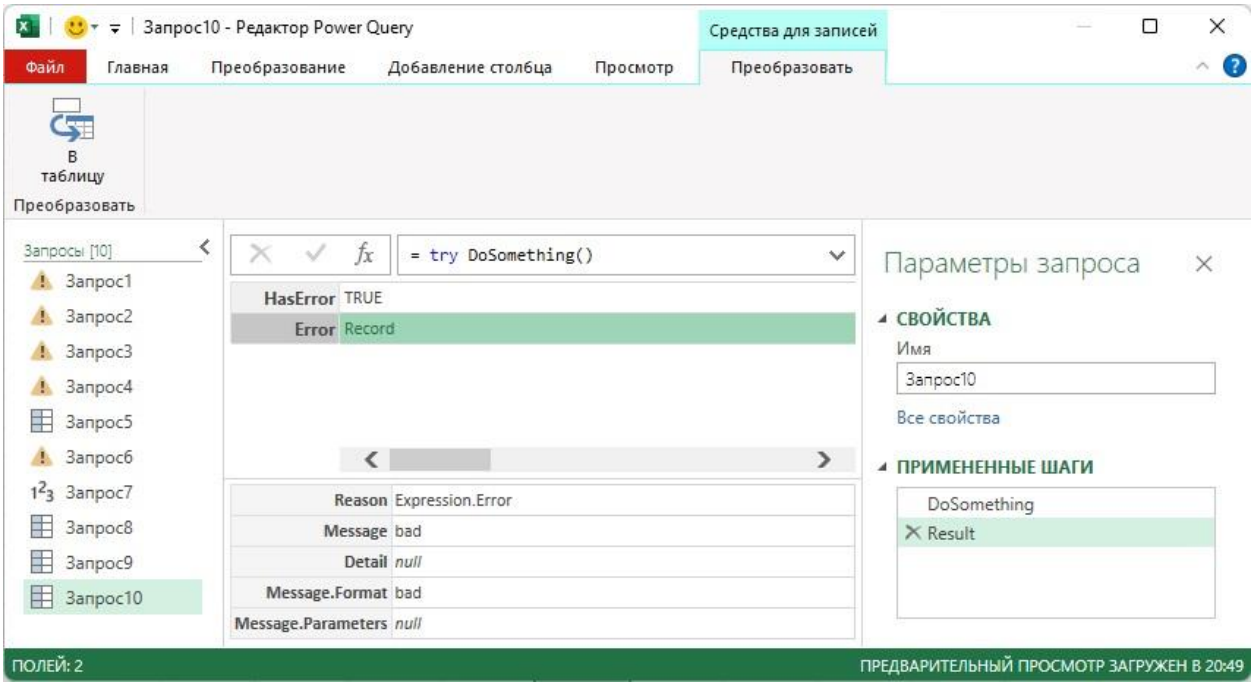

Рис. 6. Запись, возвращаемая *try*, если есть ошибка

Запись, помещенная в поле *Error*, содержит ровно три поля: *Reason*, *Message* и *Details*.[3] Это верно, даже если в записи, первоначально использовавшейся для определения ошибки, отсутствовало одно или несколько из этих полей (помните, они необязательны при определении ошибки), или если она включала дополнительные поля.

Запись, возвращаемая *try*, может быть использована для реализации логики условного исправления. Следующий код обращается к вторичному источнику данных, если первичный выдает ошибку из-за недоступности сервера.

```
let
   Primary = try GetDataFromPrimary(),
   Source =
      if Primary[HasError] = false then Primary[Value]
         /* если Primary возвращает значение, используй его */
      else if Primary[Error][Reason] = "External Source Error"
         and Primary[Error][Message] = "Server is unreachable"
         then GetDataFromSecondary()
         /* если ошибка Primary вызвана тем, что его источник недоступен,
         запроси данные с сервера Secondary */
      else error Primary[Error]
         /* если Primary вернул иную ошибку, верни её
         в качестве результата запроса */
in
   Source
```

Используя *try with otherwise*, мы бы запросили Secondary, если Primary выдаст *любую* ошибку, а не только когда основной сервер недоступен:

```
try GetDataFromPrimary()
   otherwise GetDataFromSecondary()
```

Масштаб (область действия)

Обработка ошибок должна происходить на уровне, на котором они возникают. Нельзя обработать ошибки, содержащиеся на другом уровне.

```
let
  Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}})
```

[3] Судя по рис. 6 современная реализация M создает пять полей.

```
in
  try Data otherwise 0
```

try – бесполезный оператор в этом запросе. Возможно, разработчик надеялся с помощью *try* заменить ошибки нулями. Но в этом случае *Data* возвращает допустимую таблицу. Правда, в ней самой есть ячейки с ошибками, но эти ошибки содержатся на уровне ячеек. Поскольку они не влияют на выражение данных на уровне таблицы, *try* не дает эффекта.

try будет полезен в следующем случае, но его эффект может быть не таким, как предполагал разработчик.

Листинг 11

```
let
   Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}})
in
   try List.Sum(Data[Amount]) otherwise 0
```

List.Sum суммирует значения в столбце *[Amount]* таблицы *Data.* Если выражение, определяющее значение элемента, вызывает ошибку, она повышается. *List.Sum* прекращает суммирование и возвращает ошибку. *try* обрабатывает эту ошибку, возвращая 0 вместо суммы элементов списка. Скорее всего, цель у разработчика была иная. Он хотел заменить элементы с ошибками на 0, и суммировать числовые значения. Необходимо применить *try* так, чтобы обработка ошибок велась на уровне ячеек таблицы. Кажется, что можно сделать так:

```
Table.TransformColumns(Data, (input) => try input otherwise 0)
```

Однако, эта логика не улавливает ошибки, вызванные выражениями значений ячеек. Дело в том, что аргументы вычисляются до того, как их значения будут переданы в функцию. Если оценка приводит к ошибке, функция не вызывается. Вместо этого ошибка передается шагу, который вызвал функцию. В нашем случае, если выражение значения столбца выдает ошибку, функция преобразования *(input) => ...* не вызывается, поэтому *try* не может обработать ошибку. Вместо этого ошибка передается обратно в *Table.TransformColumns*.

Проблема заключается в том, что выражение значения ячейки должно быть вычислено внутри *try*. Чтобы добиться этого, надо вернуться на уровень строки, и использовать функцию, которая получает строку. Затем внутри функции использовать ссылку на значение столбца строки и вот его подставить в *try*. Лишь тогда *try* сможет обработать ошибку. Чтобы реализовать это, нужно создать новый столбец, значения которого формировать путем проверки *try*. Затем можно удалить исходный столбец, а новому столбцу дать старое имя.

Листинг 12

```
let
   Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}}),
   ErrorsReplacedWithZero = Table.AddColumn(
      Data,
      "NewAmount",
      (row) => try row[Amount] otherwise 0
   ),
   RemoveOldAmount = Table.RemoveColumns(ErrorsReplacedWithZero, {"Amount"}),
   RenameNewAmount = Table.RenameColumns(RemoveOldAmount, {"NewAmount", "Amount"})
in
   List.Sum(RenameNewAmount[Amount]) // возвращает 10
```

Это довольно сложно. Но пример хорошо иллюстрирует *общий подход* к использованию *try* на уровне ячеек. Если же, как в нашем пример, вы просто хотите заменить любую ошибку значением по умолчанию, используйте *Table.ReplaceErrorValues*.

Листинг 13

```
let
   Data = #table({"Amount"}, {{10}, {error "help!"}, {error "save me!"}}),
   ErrorsReplacedWithZero = Table.ReplaceErrorValues(
```

```
    Data,
    {{"Amount", 0}}
  ) // заменяет ошибки в столбце Amount нулями
in
  List.Sum(ErrorsReplacedWithZero[Amount]) // возвращает 10
```

Применить *try* к элементам списка сложнее. Для списков нет функции *List.ReplaceErrorValues*. Самым простым решением может быть преобразование списка в таблицу, обработка ошибки, а затем обратное преобразование таблицы в список.

Листинг 14

```
let
  Data = {10, error "help!", error "save me!"},
  #"Преобразовано в таблицу" = Table.FromValue(Data),
  #"Замененные ошибки" = Table.ReplaceErrorValues(#"Преобразовано в таблицу", {{"Value", 0}}),
  Value = #"Замененные ошибки"[Value],
  #"Вычисленная сумма" = List.Sum(Value)
in
  #"Вычисленная сумма"
```

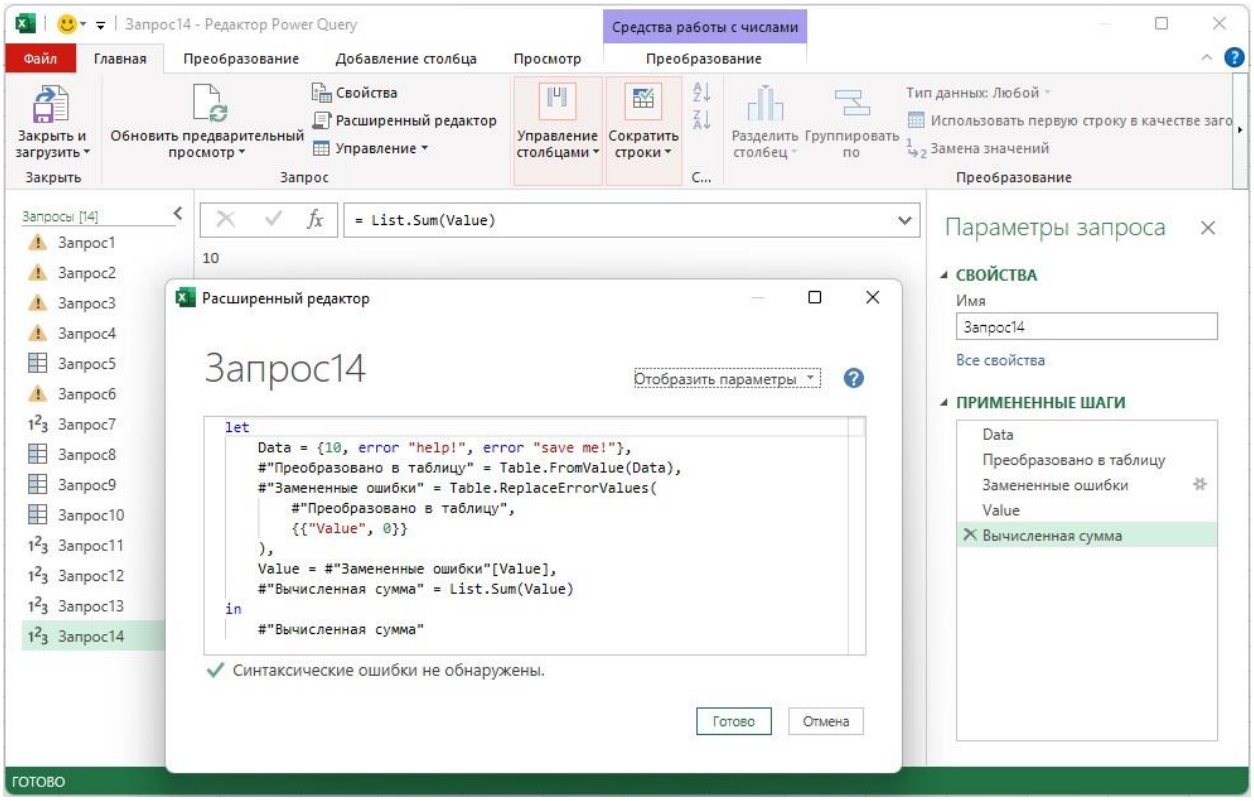

Рис. 7. Сумма элементов списка, содержащего ошибки

Нарушения правил

Вы можете использовать ошибки, как сигнал о несоответствиях. Допустим, вы обрабатываете CSV-файл, где значения в столбце *ItemCode* должны начинаться с буквы *A*. В начале запроса вы проверяете это, заменяя несоответствующие значения ошибками. Последующие этапы обработки, которые обращаются к столбцу, будут предупреждены, если они попытаются работать со значениями, нарушающими правила.

```
let
  Data = GetData(), // for testing use: #table({"ItemCode"}, {{"1"}, {"A2"}})
  Validated = Table.TransformColumns(
    Data,
    {
      "ItemCode",
      each if Text.StartsWith(_, "A") then _ else error Error.Record(
```

```
            "Invalid Data",
            "ItemCode does not start with expected letter",
            _
        )
    }
  )
in
  Validated
```

Такую проверку полезно применить в базовом запросе, на который будут ссылаться несколько других запросов. Это позволит вам выполнить проверку один раз (вспомните принцип *Не повторяйся*, Don’t repeat yourself), гарантируя, что пользователи, пытающиеся использовать ошибочные данные, будут предупреждены о наличии аномалий.

Другой вариант – добавить столбец со значениями *true* и *false*, в зависимости от того, соблюдается ли правило:

```
let
  Data = GetData(), // for testing use: #table({"ItemCode"}, {{"1"}, {"A2"}})
  Validated = Table.AddColumn(
    Data,
    "ValidItemCode",
    each Text.StartsWith(_[ItemCode], "A"),
    type logical
  )
in
  Validated
```

В этом примере логика заботится о том, является ли *ItemCode* допустимым. Если разработчик забудет выполнить проверку, неверные данные могут рассматриваться как действительные. В отличие от этого, подход замены несоответствующих данных ошибками гарантирует, что попытка получить доступ к недопустимому значению, закончится ошибкой. Пользователь будет вынужден поправить данные в источнике. Какой вариант выбрать, зависит от вашего контекста.

В следующей заметке

Я планирую рассказать о том, что остается за кулисами: организации разделов кода и о том, как M предоставляет возможность аннотировать значения дополнительной информацией (метаданными). Однако перед этим обсудим, как работает система типов в Power Query.