## Цикл в Power Query на основе функции List.Accumulate

Язык М Power Query в явном виде использует единственную управляющую структуру:

if <…> then <…> else <…>

Привычные нам по другим языкам программирования циклы отсутствуют. Как указывает [Ben Gribaudo](https://baguzin.ru/wp/?p=25206), это ни в коей мере не говорит о незрелости языка М. Просто в функциональных языках (к которым относится М) эти задачи решаются иначе. И всё же в М есть обходные пути, позволяющие организовать циклы. В частности, можно использовать функцию List.Accumulate. Как часто бывает, [официальная документация](https://docs.microsoft.com/ru-ru/powerquery-m/list-accumulate) по функции оставляет желать лучшего:

Изображение выглядит как текст

Автоматически созданное описание

Рис. 1. Сведения о функции List.Accumulate с сайта [Microsoft](https://docs.microsoft.com/ru-ru/powerquery-m/list-accumulate)

Чтобы проверить код примера 1, вставим его в редактор PQ:

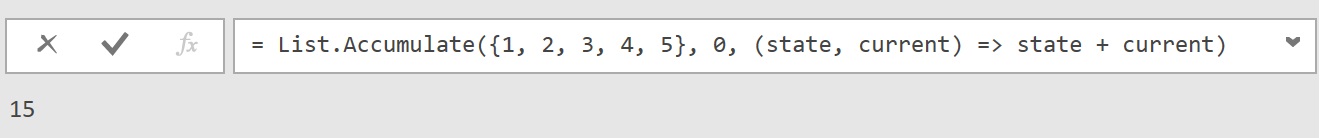


Рис. 2. Пример работает верно

**Листинг 1[[1]](#footnote-1)**

List.Accumulate(

{1, 2, 3, 4, 5},

0,

(state, current) => state + current

)

Для начала разберемся с понятиями *начальное значение* и *аккумулятор* (накопитель). В листинге 1 начальное значение = 0, а функция…

(state, current) => state + current

…является аккумулятором.

Состояние (state) – это накопленное значение функции аккумулятора. В нашем примере накопление начинается с нуля. Накопленное значение меняется после каждой итерации функции аккумулятора. Значение состояния будет увеличиваться на число, которое представлено текущим значением списка (current). Давайте рассмотрим шаги последовательно:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 4. Итерационная работа функции List.Accumulate

На первом шаге состояние равно нулю , а текущее значение = 1 (первый элемент в списке). На этом этапе Power Query реализует функцию аккумулятора и выполняет суммирование state + current. Сумма = 1 и это значение присваивается состоянию перед второй итерацией.

#### Типы данных

В качестве накопителя вы всегда должны использовать функцию с двумя аргументами. Начальное состояние должно иметь тот же тип, что и первый аргумент функции-накопителя, и тип результата.

Продемонстрируем это с помощью псевдокода:

**Листинг 3**

myList = {... items of TYPE\_X ...},

fnMyAccumulator = (state as TYPE\_Y, current as TYPE\_X) as TYPE\_Y =>

... ,

myAccumulatedResult = List.Accumulate(

myList,

some\_initial\_state\_whose\_type\_is\_Y,

fnMyAccumulator)

myList – это список, элементы которого имеют тип TYPE\_X (например, запись, список, число или текст), а наше начальное состояние имеет тип TYPE\_Y, мы должны определить нашу функцию-накопитель как функцию, которая получает аргумент состояния, тип которого TYPE\_Y и текущий аргумент тип которого равен TYPE\_X, и возвращает значение, тип которого равен TYPE\_Y.

Если не всё понятно, не теряйтесь! Продолжим изучение примеров.

#### List.Accumulate как конвейер

**Листинг 4**

= List.Accumulate(

{1, 2, 100, -40, 5},

0,

(max, current) =>

if max < current then current else max

)

Функция вернет число 100, которое является максимальным в списке {1, 2, 100, -40, 5}.

Давайте усложним и вычислим одновременно максимальное и минимальное значение в одном «конвейере».

**Листинг 5**

let

Источник = List.Accumulate(

{1, 2, 100, -40, 4},

[min = #infinity, max = -#infinity],

(state, current) =>

[

min = if state[min] > current then current else state[min],

max = if state[max] < current then current else state[max]

]

)

in

Источник

Результат – запись: [min = -40, max = 100]

В этом примере, чтобы передать два числа в качестве состояния (начального значения), мы определили запись с ключами min и max. Функция накопления также вычисляет новую запись min и max в соответствии с предыдущими min и max и текущим значением в списке. Можно еще усложнить пример, чтобы выполнить более двух вычислений за один проход.

Следующая функция использует List.Accumulate для создания отобранного списка имен столбцов из первоначального списка имен столбцов на основе списка индексов столбцов.

**Листинг 6**

let

Источник = #table(3, {}),

SelectionColumn = (tbl as table, columnID as list) as list =>

List.Accumulate(

columnID,

{},

(state, current) => List.Combine({state, {Table.ColumnNames(tbl){current}}})

),

Result = SelectionColumn(

Источник,

{0,2}

)

in

Result

Первая строка кода…

Источник = #table(3, {}),

… задает пустую таблицу из трех столбцов. Имена столбцов по умолчанию образуют список {"Column1", "Column2", "Column3"}.

Функция…

SelectionColumn = (tbl as table, columnID as list) as list =>

List.Accumulate(

columnID,

{},

(state, current) => List.Combine({state, {Table.ColumnNames(tbl){current}}})

),

… принимает таблицу tbl и список columnID.

В соответствии с требованиями к синтаксису функции List.Accumulate начальное состояние…

{}

… имеет тот же тип (список), что и первый аргумент функции-накопителя…

state

… и тип результата…

(…) as list

Функция-накопитель использует функцию [List.Combine](https://docs.microsoft.com/ru-ru/powerquery-m/list-combine)…

List.Combine({state, {Table.ColumnNames(tbl){current}}})

… и объединяет начальное состояние state, представляющее собой пустой список {} с именами таблицы tbl, последовательно отбираемыми по индексам columnID по одному за раз с помощью функции [Table.ColumnNames](https://docs.microsoft.com/ru-ru/powerquery-m/table-columnnames)()

Изображение выглядит как текст

Автоматически созданное описание

Рис. 5. Итерации функции SelectionColumn

Думайте о функции List.Accumulate, как о конвейере. Каждый участок получает данные с предыдущего участка, преобразует их и передает следующему участку. Cила функции List.Accumulate проявляется, когда в качестве результата нужен список, запись, таблица или при построении прогрессий.

#### Рекурсия

Используем List.Accumulate для расчета [чисел Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8). Эти числа задаются линейным рекуррентным соотношением:

*F0 = 0, F1 = 1, Fn = Fn–1 + Fn–2,*

где *n ≥ 2, n ∈ Z*.

**Листинг 7. Функция Fibonacci**

(input) =>

let

Fibonacci = if input = 0 then [PreviousNum = 0, CurrentNum = 0] else

if input = 1 then [PreviousNum = 0, CurrentNum = 1] else

List.Accumulate(

{0..input-2},

[PreviousNum = 0, CurrentNum = 1],

(state, current) =>

[

PreviousNum = state[CurrentNum],

CurrentNum = state[CurrentNum] + state[PreviousNum]

]

)

in

Fibonacci[CurrentNum]

Первые два условия if проверяют входное значение и возвращают нулевой и первый член последовательности Фибоначчи. Если n ≥ 2, начинает работать рекуррентная формула на основе List.Accumulate. Для функции List.Accumulate нельзя определить условие динамического разрыва (выхода из цикла), поэтому нужно задать «правильный» список числа итераций:

{0..input-2},

… а функция List.Accumulate переберет в списке все элементы. Мы инициализируем список числами от 0 до input-2. Обратите внимание, что числа из списка {0..input-2} (они же, текущие значения) не участвуют в расчетах функции List.Accumulate. Список нужен лишь для определения числа итераций до останова. Расчеты же основаны на записи [PreviousNum = 0, CurrentNum = 1].

Эта запись определяет *начальное значение* и состоит из двух числовых полей: *PreviousNum* и *CurrentNum*. Функция-аккумулятор возвращает обновленную версию этой записи. Новое значение *PreviousNum* равно предыдущему значению *CurrentNum*. Новое значение *CurrentNum* равно сумме предыдущих значений *PreviousNum* и *CurrentNum*.

Например, 11-й член ряда Фибоначчи вычисляется за 10 итераций:



Рис. 6. Итерации функции Fibonacci

Функция Fibonacci в листинге 7 имеет следующий интерфейс:

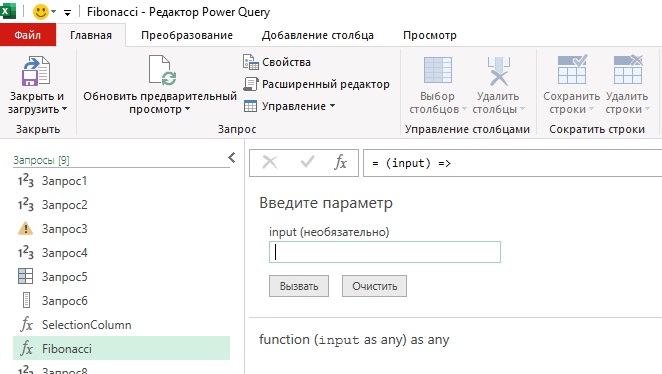


Рис. 7. Вызов функции Fibonacci

Чтобы рассчитать значение любого члена ряда, просто введите его порядковый номер в поле *input*.

**Листинг 8**

let

Источник = Fibonacci(11)

in

Источник

#### Переименование столбцов по их позиции в таблице

В следующем примере я импортирую данные из файла fplreview.csv (приложен), который получаю из Интернета. Это прогноз выступления игроков [Fantasy Premier League](https://fantasy.premierleague.com/):

Изображение выглядит как стол

Автоматически созданное описание

Рис. 8. Исходные данные (показаны не все столбцы)

Проблема в том, что с переходом к следующему туру некоторые заголовки столбцов (подчеркнул на рис. 8) меняются. Поскольку в коде PQ названия заголовков прописаны жестко, приходилось править код. Более того, поскольку я использую прогноз на восемь туров, нужно вносить изменения в код PQ, когда чемпионат подходит к концу и остается менее восьми туров. Я задумался, нельзя ли сделать запрос более робастным, чтобы его не приходилось править от тура к туру!?

И я добавил параметр – *tour*, который считывается из таблицы на листе Excel:

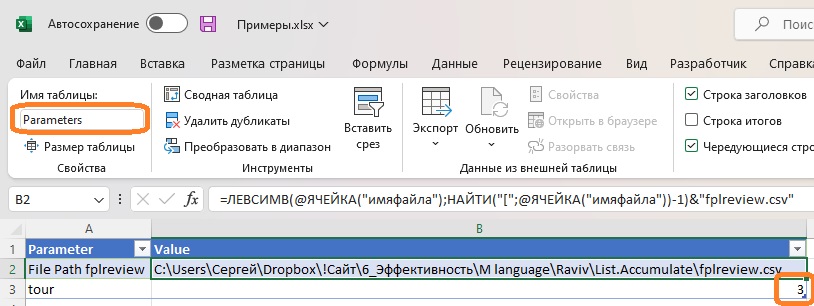


Рис. 9. Маленькая таблица параметров

Обязательно сохраните имя таблицы – *Parameters*. Оно жестко прописано в коде. Я также использую параметр *fullfilepath*, основанный на первой строк таблицы. Благодаря этому достаточно разместить файл fplreview.csv в той же папке, что Excel-файл с запросом, и ссылка на fplreview.csv обновится автоматически.

**Листинг 9**

let

fullfilepath = Excel.CurrentWorkbook(){[Name="Parameters"]}[Content]{0}[Value],

//считывает путь к файлу fplreview.csv из первой строки таблицы Parameters

tour = Excel.CurrentWorkbook(){[Name="Parameters"]}[Content]{1}[Value],

//считывает номер тура из второй строки таблицы Parameters

Источник = Csv.Document(

File.Contents(fullfilepath),

[

Delimiter = ",",

Columns = if (6 + (39 - tour)\*2) >= 22 then 22 else 6 + (39 - tour)\*2,

//определяет число столбцов с данными в зависимости от оставшегося числа туров

Encoding = 65001,

QuoteStyle = QuoteStyle.None

]

),

#"Повышенные заголовки" = Table.PromoteHeaders (Источник, [ PromoteAllScalars = true ] ),

clmnNames = Table.ColumnNames ( #"Повышенные заголовки" ),

//формирует список заголовков столбцов

count = ( List.Count ( clmnNames ) - 6 ) / 2,

//определяет число пар столбцов данных (пара - это Pts + Min)

//для туров с 1 по 31 count = 8

//для туров с 32 по 38 count = 39 - tour

newClmnNames = List.FirstN ( clmnNames, 6 )

//названия первых шести столбцов остаются без изменений

& List.Accumulate (

{ tour .. tour + count - 1 },

{},

( state, current ) => state

& { "Min" & Text.From ( current ) } & { "Pts" & Text.From ( current ) }

),

rename = Table.FromColumns ( Table.ToColumns ( #"Повышенные заголовки" ), newClmnNames )

in

rename

Фрагмент…

List.Accumulate (

{ tour .. tour + count - 1 },

{},

( state, current ) => state

& { "Min" & Text.From ( current ) } & { "Pts" & Text.From ( current ) }

),

… формирует названия переменных столбцов. Рассмотрим пример для tour = 3

Изображение выглядит как текст

Автоматически созданное описание

Рис. 10. Итерации функции List.Accumulate

Код…

rename = Table.FromColumns ( Table.ToColumns ( #"Повышенные заголовки" ), newClmnNames )

… сначала превращает таблицу в список столбцов (функция Table.ToColumns); при этом теряются заголовки столбцов, а затем (функция Table.FromColumns) – обратно в таблицу, давая столбцам новые заголовки (newClmnNames).

#### Множественный поиск и замена

Если у вас есть текст, и нужно заменить несколько слов (подстрок), поместим текст в умную таблицу на лист Excel (*mytext*, рис. 11а), а слова для поиска и замены в таблицу *replacements* (рис. 11б).

Изображение выглядит как текст

Автоматически созданное описание

Рис. 11. Исходные данные для множественного поиска и замены: а) текст для поиска; б) таблица подстановки

**Листинг 10**

let

Source = Excel.CurrentWorkbook(){[Name="mytext"]}[Content],

#"Changed Type" = Table.TransformColumnTypes(Source,{{"Text", type text}}),

Source1 = Excel.CurrentWorkbook(){[Name="replacements"]}[Content],

#"Changed Type1" = Table.TransformColumnTypes(Source1,{{"Find", type text}, {"Replace", type text}}),

#"Added Custom" = Table.AddColumn(

#"Changed Type",

"After replacement",

each List.Accumulate(

List.Numbers(0, Table.RowCount(#"Changed Type1")),

[Text],

(state, current) => Text.Replace(

state,

#"Changed Type1"[Find]{current},

#"Changed Type1"[Replace]{current}

)

)

)

in

#"Added Custom"

Сначала мы загрузили две таблицы и изменили тип их столбцов на *text*. Далее добавили пользовательский столбец *After replacement*, заполнив его текстом на основе функции *List.Accumulate*. Внутри этой функции создали список чисел, начиная с 0. Размер списка равен количеству строк в таблице подстановки – {0,1,2,3,4}. В качестве начального состояния взяли столбец *Text* таблицы *mytext*. Итерации начинаются со значения *current* = 0. Берется значение из первой строки столбца *Find* таблицы *replacements* – *four* и заменяется на значение из первой строки столбца *Replace* – *six*. Перед второй итерацией новое состояние аргумента *state* содержит исходный текст с заменными значениями *four* –> *six*. Аргумент *current* получает значение 1, и происходит замена *normal* –> *casual*. После пяти замен работа *List.Accumulate* завершается. Текст после этих замен помещается в соответствующие строки столбца *After replacement*.

#### Удаление пустых строк и столбцов из таблицы

Допустим, у вас есть неказистая таблица:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 12. Исходная таблица, содержащая пустые строки и столбцы

Задача – удалить полностью пустые строки и столбцы. На рис. 12 они выделены желтым. Если загрузить таблицу в PQ, то пустые строки можно удалить командой *Главная* –> *Удалить стоки* –> *Удалить пустые стоки*. К сожалению в PQ нет команды удалить пустые столбцы… Но, можно транспонировать таблицу, выполнить команду *Удалить пустые стоки* и повторно транспонировать в исходный вид. Эту же задачу можно решить с помощью функции List.Accumulate:

**Листинг 11**

let

Источник = Excel.CurrentWorkbook(){[Name="Таблица7"]}[Content], // загружаем таблицу

FnRemoveEmptyColumns = (tbl) =>

// функция, которая принимает таблицу, удаляет пустые столбцы и возвращает таблицу

let

Headers = Table.ColumnNames(tbl),

//извлекаем заголовки таблицы в виде списка {"Столбец1", "Столбец2", … "Столбец10"}

fnMyAccumulator =

//функция-аккумулятор; будет использоваться внутри List.Accumulate; принимает

//таблицу и название столбца (текст), и удаляет столбец, если все его значения равны null

(tbl as table, columnName as text) as table =>

if List.MatchesAll(Table.Column(tbl, columnName), each \_ is null) then

Table.RemoveColumns(tbl, {columnName})

else

tbl,

myAccumulatedResult = List.Accumulate(

Headers, //в качестве списка значений используется список заголовков таблицы

tbl, // аргумент state – таблица

(tbl, columnName) => fnMyAccumulator(tbl, columnName)

//для каждого заголовка таблицы решается удалять столбец (если он пустой) или нет

)

in

myAccumulatedResult,

Пользовательский = FnRemoveEmptyColumns(Источник),

//передача таблицы, загруженной на первом шаге кода, в качестве аргумента функции

#"Транспонированная таблица" = Table.Transpose(Пользовательский),

// транспонирование столбцов в строки

Пользовательский1 = FnRemoveEmptyColumns(#"Транспонированная таблица"),

//повторный вызов функции для удаления пустых столбцов (то есть, строк исходной таблицы)

#"Транспонированная таблица1" = Table.Transpose(Пользовательский1),

//обратной транспонирование

#"Измененный тип" = Table.TransformColumnTypes(

#"Транспонированная таблица1",

{

{"Column2", Int64.Type},

{"Column3", Int64.Type},

{"Column4", type date}

}

)

in

#"Измененный тип"

#### Цикл на основе функции List.Accumulate

Итак, функция List.Accumulate позволяет организовать цикл. Функция List.Accumulate имеет три аргумента: *list* (список), *seed* (начальное значение) и *accumulator* (функция-аккумулятор).

list может содержать:

* элементы, которые участвуют в работе функции-аккумулятора (листинг 1, 4, 5, 6, 11),
* индексы, которые лишь задают число итераций функции-аккумулятора (листинг 7, 9, 10),

*seed* может быть:

* числом (листинг 1, 4), текстом, иным значением
* записью (листинг 5, 7), списком (листинг 6, 9), таблицей (листинг 10, 11).

*accumulator* – функция с двумя аргументами.

#### Использованные материалы

[Gil Raviv. Power Query List.Accumulate – Unleashed](https://datachant.com/2016/06/02/power-query-list-accumulate-unleashed/)

[Gil Raviv. Automatically Remove Empty Columns And Rows From A Table In Power BI](https://datachant.com/2021/07/21/remove-empty-columns-rows-power-bi/)

[Gil Raviv. Book Excerpt: Loops, Recursions, Fibonacci And More From Chapter 9: M #PowerQuery](https://datachant.com/2018/04/09/book-excerpt-chapter-9-m-powerquery/)

[Chris Webb. Converting Lists Of Numbers To Text Ranges In Power Query](https://blog.crossjoin.co.uk/2016/11/24/converting-lists-of-numbers-to-text-ranges-in-power-query/)

[Purna Duggirala. Multiple Find Replace with Power Query List.Accumulate()](https://chandoo.org/wp/multiple-find-replace-list-accumulate/)

[Ken Puls. Running Totals using the List.Accumulate() Function](https://excelguru.ca/running-totals-using-the-list-accumulate-function/)

1. Номер листинга соответствует номеру запроса в приложенном Excel файле. [↑](#footnote-ref-1)