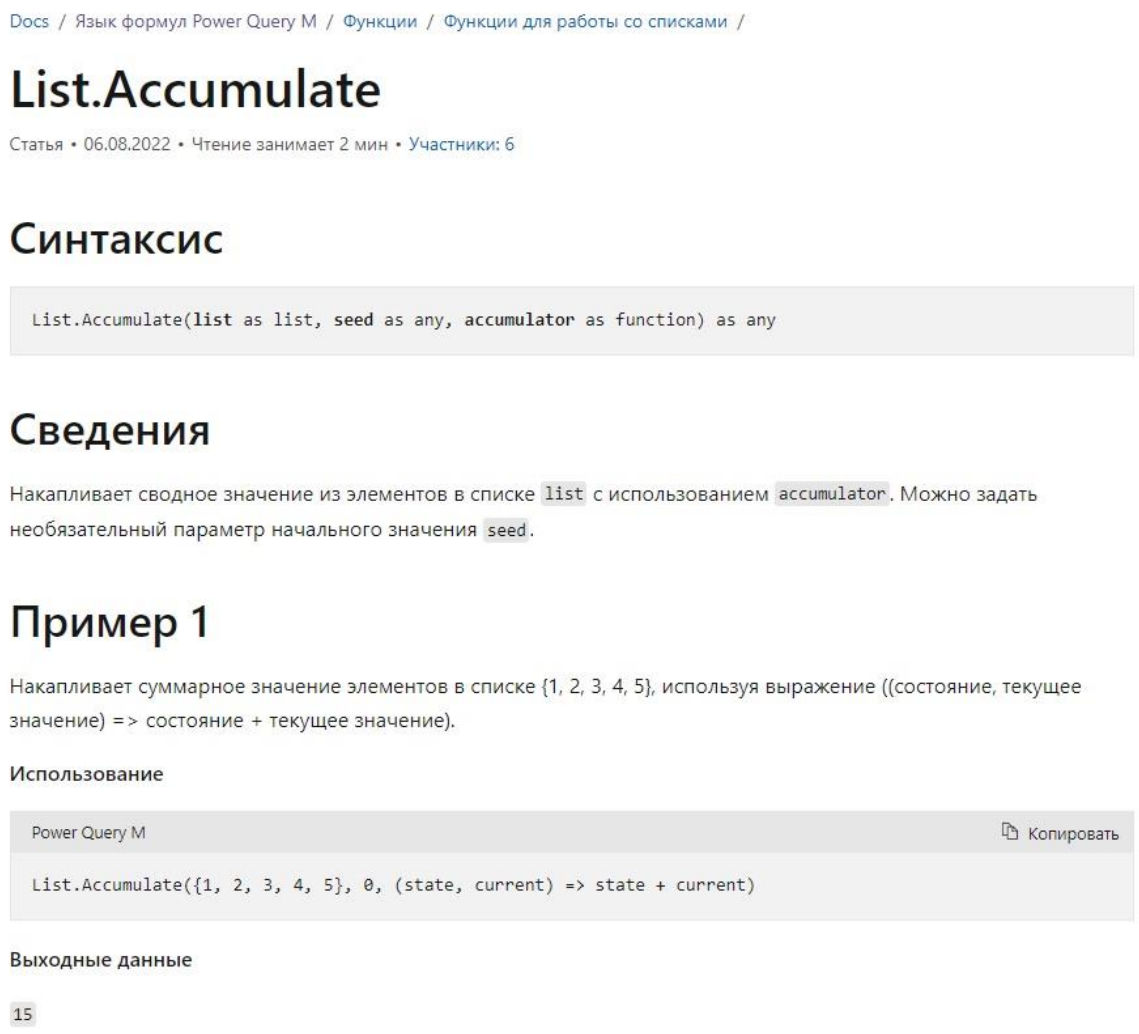
Цикл в Power Query на основе функции List.Accumulate

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

if <…> then <…> else <…>

Привычные нам по другим языкам программирования циклы отсутствуют. Как указывает Ben Gribaudo, это ни в коей мере не говорит о незрелости языка M. Просто в функциональных языках (к которым относится M) эти задачи решаются иначе. И всё же в M есть обходные пути, позволяющие организовать циклы. В частности, можно использовать функцию List.Accumulate. Как часто бывает, официальная документация по функции оставляет желать лучшего:



Docs / Язык формул Power Query M / Функции / Функции для работы со списками /

List.Accumulate

Статья • 06.08.2022 • Чтение занимает 2 мин • Участники: 6

Синтаксис

```
List.Accumulate(list as list, seed as any, accumulator as function) as any
```

Сведения

Накапливает сводное значение из элементов в списке `list` с использованием `accumulator`. Можно задать необязательный параметр начального значения `seed`.

Пример 1

Накапливает суммарное значение элементов в списке {1, 2, 3, 4, 5}, используя выражение ((состояние, текущее значение) => состояние + текущее значение).

Использование

Power Query M — Копировать

```
List.Accumulate({1, 2, 3, 4, 5}, 0, (state, current) => state + current)
```

Выходные данные

```
15
```

Рис. 1. Сведения о функции List.Accumulate с сайта Microsoft

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

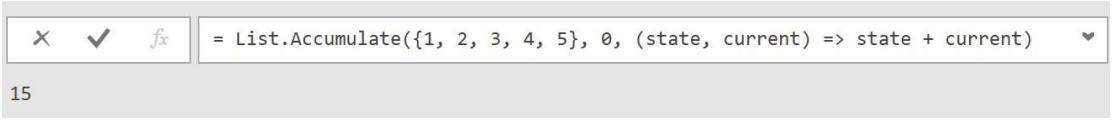


Рис. 2. Пример работает верно

Листинг 1[1]

```
List.Accumulate(
        {1, 2, 3, 4, 5},
        0,
        (state, current) => state + current
)
```

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

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:

Итерация	Состояние	Текущее значение	Функция аккумулятор
1	0	1	0+1
2	1	2	1+2
3	3	3	3+3
4	6	4	6+4
5	10	5	10+5
Результат	15		

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…

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()

Заголовки таблицы tbl {"Column1", "Column2", "Column3"}				
Отбираемые номера столбцов columnID {0,2}				
Итерация	Состояние, state	Текущее значение списка columnID	Функция-аккумулятор List.Combine	Результат
1	{}	0	List.Combine({{}, {Table.ColumnNames(tbl){0}}})	{"Column1"}
2	{"Column1"}	2	List.Combine({{"Column1"}, {Table.ColumnNames(tbl){2}}})	{"Column1", "Column3"}

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. Сила функции List.Accumulate проявляется, когда в качестве результата нужен список, запись, таблица или при построении прогрессий.

Рекурсия

Используем List.Accumulate для расчета чисел Фибоначчи. Эти числа задаются линейным рекуррентным соотношением:

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2},$

где $n \geq 2, n \in Z$.

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:

Итерация	Состояние	Текущее значение	Функция-аккумулятор	Результат
1	[0,1]	0	[1,1+0]	[1,1]
2	[1,1]	1	[1,1+1]	[1,2]
3	[1,2]	2	[2,2+1]	[2,3]
4	[2,3]	3	[3,3+2]	[3,5]
5	[3,5]	4	[5,5+3]	[5,8]
6	[5,8]	5	[8,8+5]	[8,13]
7	[8,13]	6	[13,13+8]	[13,21]
8	[13,21]	7	[21,21+13]	[21,34]
9	[21,34]	8	[34,34+21]	[34,55]
10	[34,55]	9	[55,55+34]	[55,89]

Рис. 6. Итерации функции Fibonacci

Функция Fibonacci в листинге 7 имеет следующий интерфейс:

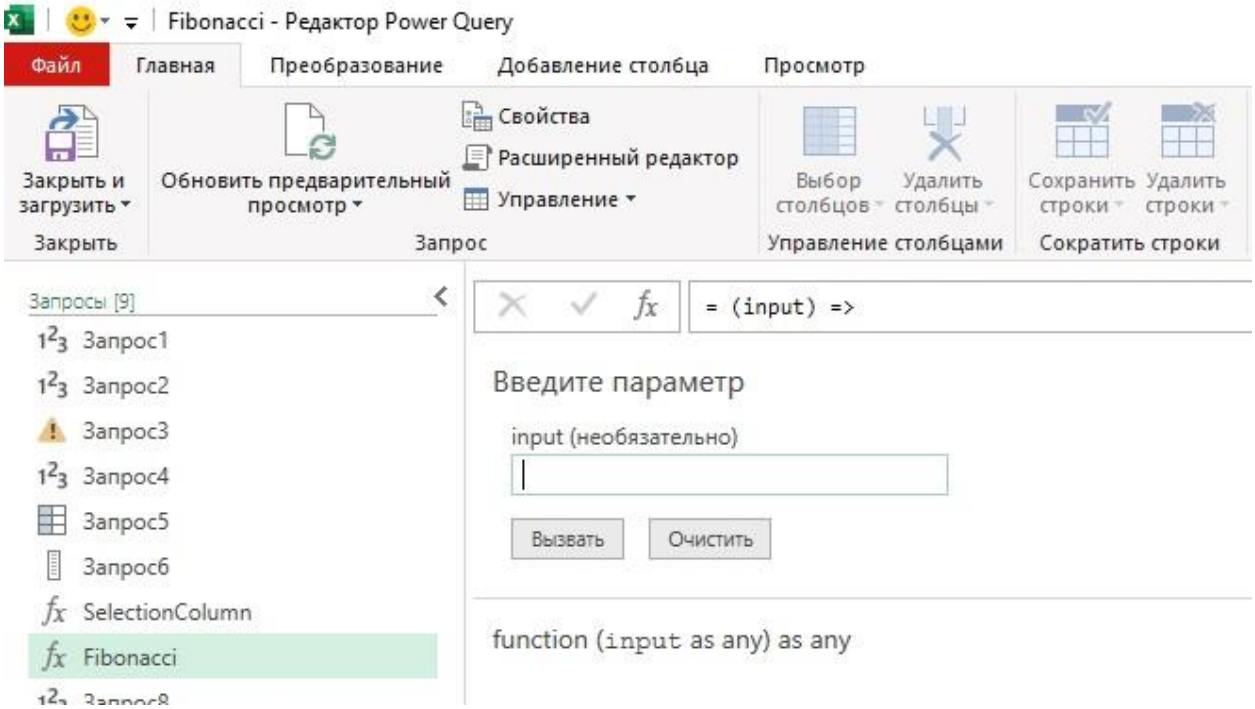


Рис. 7. Вызов функции Fibonacci

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:

	Pos	ID	Name	BV	SV	Team	3_xMins	3_Pts	4_xMins	4_Pts	5_xMins	5_Pts	6_xMins	6_Pts	7_xMins	7_Pts
1	D	1	Cédric	4.4	4.4	Arsenal	3	0.14	6	0.27	7	0.28	8	0.27	7	0.32
2	G	2	Leno	4.5	4.5	Fulham	36	1.47	50	1.79	67	2.72	80	2.83	80	2.91
3	M	3	Xhaka	5.0	5.0	Arsenal	79	2.80	72	2.67	70	2.40	68	2.06	66	2.39
4	M	4	Elneny	4.4	4.4	Arsenal	9	0.36	11	0.45	9	0.36	13	0.43	15	0.56
5	D	5	Holding	4.4	4.4	Arsenal	6	0.23	7	0.27	6	0.22	9	0.27	10	0.44
6	M	6	Partey	5.0	5.0	Arsenal	83	3.12	79	3.12	78	2.84	75	2.38	73	2.84
7	M	7	Ødegaard	6.5	6.5	Arsenal	74	4.08	74	4.39	72	3.93	69	3.17	68	3.95
8	D	8	Tierney	4.9	4.9	Arsenal	27	1.48	41	2.30	55	2.72	56	2.05	53	2.93
9	M	9	Pépé	5.4	5.4	Arsenal	9	0.75	10	0.78	7	0.55	11	0.73	12	0.97
10	D	10	White	4.5	4.5	Arsenal	75	3.54	67	3.23	61	2.62	59	1.89	60	2.92
11	F	11	Nketiah	6.9	6.9	Arsenal	28	1.82	32	2.26	30	1.93	28	1.60	29	1.97
12	M	12	Smith Rowe	5.9	5.9	Arsenal	12	0.84	17	1.22	22	1.44	28	1.56	30	1.96
13	M	13	Saka	8.0	8.0	Arsenal	78	5.22	74	5.37	73	4.78	72	3.93	72	5.03

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:

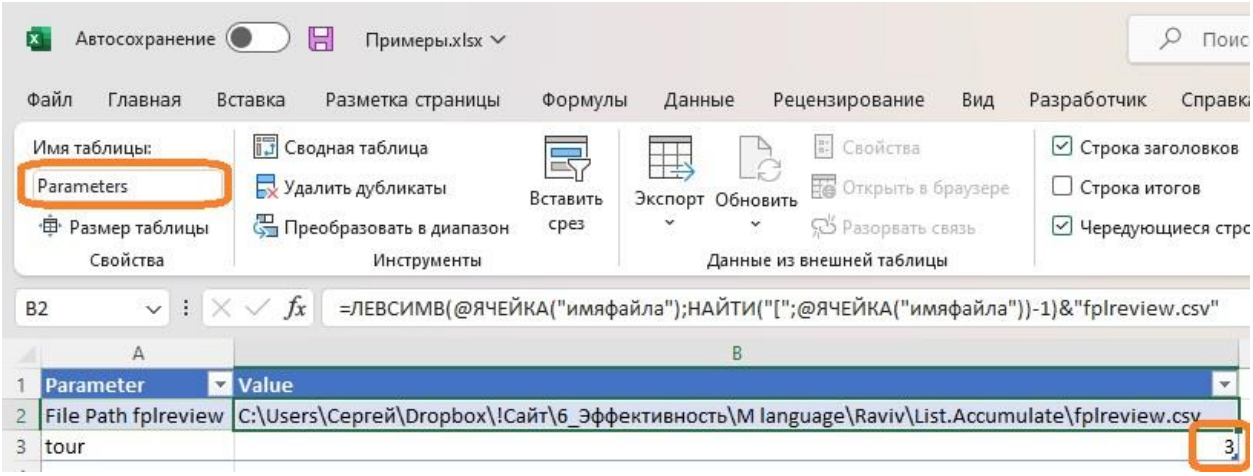


Рис. 9. Маленькая таблица параметров

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

	A	B	C	D	E
1	минимальное значение		tour = 3		
2	максимальное значение		tour + count - 1 = 10		
3					
4	Итерация	Состояние	Текущее значение	Функция аккумулятор	Результат
5	1	{}	3	{} & { "Min" & Text.From (3) } & { "Pts" & Text.From (3) }	{"Min3", "Pts3"}
6	2	{"Min3", "Pts3"}	4	{"Min3", "Pts3"} & { "Min" & Text.From (4) } & { "Pts" & Text.From (4) }	{"Min3", "Pts3", "Min4", "Pts4"}
7	3	{"Min3", "Pts3", "Min4", "Pts4"}	5	{"Min3", "Pts3", "Min4", "Pts4"} & { "Min" & Text.From (5) } & { "Pts" & Text.From (5) }	{"Min3", "Pts3", "Min4", "Pts4", "Min5", "Pts5"}
8	4	…	6		
9	5		7		
10	6		8		
11	7		9		
12	8		10		

Рис. 10. Итерации функции List.Accumulate

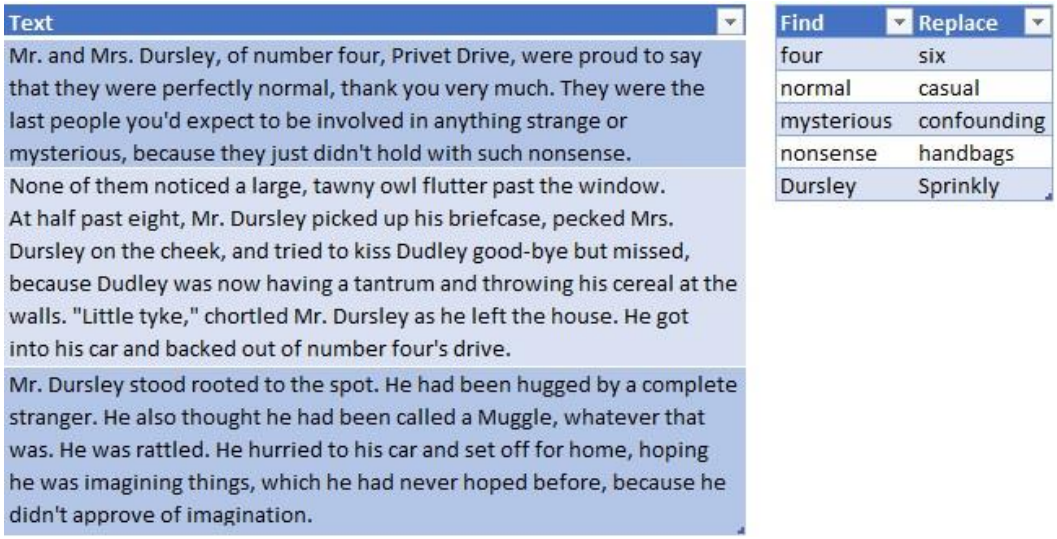
Код…

```
 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).



Text
Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much. They were the last people you'd expect to be involved in anything strange or mysterious, because they just didn't hold with such nonsense.
None of them noticed a large, tawny owl flutter past the window. At half past eight, Mr. Dursley picked up his briefcase, pecked Mrs. Dursley on the cheek, and tried to kiss Dudley good-bye but missed, because Dudley was now having a tantrum and throwing his cereal at the walls. "Little tyke," chortled Mr. Dursley as he left the house. He got into his car and backed out of number four's drive.
Mr. Dursley stood rooted to the spot. He had been hugged by a complete stranger. He also thought he had been called a Muggle, whatever that was. He was rattled. He hurried to his car and set off for home, hoping he was imagining things, which he had never hoped before, because he didn't approve of imagination.

Find	Replace
four	six
normal	casual
mysterious	confounding
nonsense	handbags
Dursley	Sprinkly

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

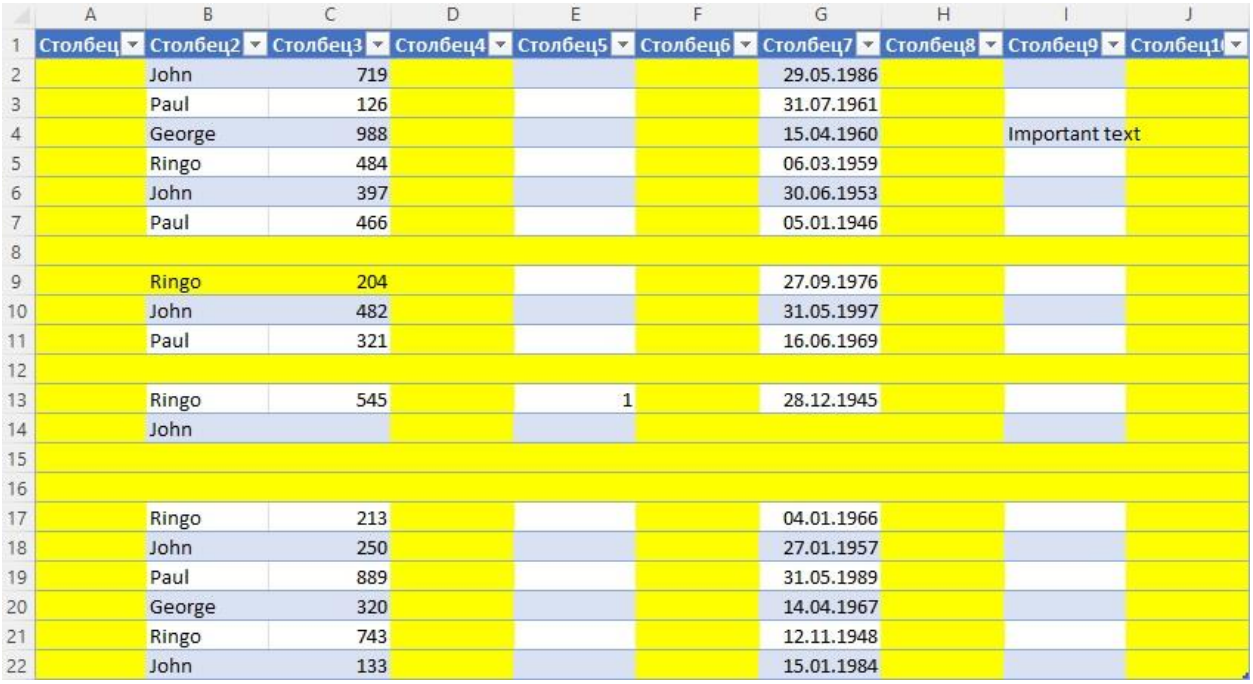
Листинг 10

```
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:



	A	B	C	D	E	F	G	H	I	J
1	Столбец	Столбец2	Столбец3	Столбец4	Столбец5	Столбец6	Столбец7	Столбец8	Столбец9	Столбец10
2		John	719				29.05.1986			
3		Paul	126				31.07.1961			
4		George	988				15.04.1960		Important text	
5		Ringo	484				06.03.1959			
6		John	397				30.06.1953			
7		Paul	466				05.01.1946			
8										
9		Ringo	204				27.09.1976			
10		John	482				31.05.1997			
11		Paul	321				16.06.1969			
12										
13		Ringo	545		1		28.12.1945			
14		John								
15										
16										
17		Ringo	213				04.01.1966			
18		John	250				27.01.1957			
19		Paul	889				31.05.1989			
20		George	320				14.04.1967			
21		Ringo	743				12.11.1948			
22		John	133				15.01.1984			

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

Gil Raviv. Automatically Remove Empty Columns And Rows From A Table In Power BI

Gil Raviv. Book Excerpt: Loops, Recursions, Fibonacci And More From Chapter 9: M #PowerQuery

Chris Webb. Converting Lists Of Numbers To Text Ranges In Power Query

Purna Duggirala. Multiple Find Replace with Power Query List.Accumulate()

Ken Puls. Running Totals using the List.Accumulate() Function