Язык M Power Query. Парадигма

Чтобы лучше понимать, как работает Power Query, давайте отойдем от деталей и рассмотрим парадигму,[1] на которой основан M. Такие детали, как переменные, выражения *let*, функции и идентификаторы, важны. Но, если мы сосредоточимся только на деталях, мы можем упустить общую картину. Сделаем шаг назад и покажем, как устроен M. Не зная этого, мы будем озадачены необычным его поведением. Удивлены, почему M не позволяет делать некоторые вещи, к которые мы привыкли в других языках.[2]

Предыдущая заметка Следующая заметка

Что представляет собой язык M

Power Query (также известный как M) – это язык запросов. На языке M обращаются к источнику данных. При этом M не позволяет изменять данные в источнике. Кроме того, M не является языком программирования общего назначения. Он не предназначен для рисование диалоговых окон, копирования или печати файлов. M служит созданию запросов и объединению данных, но не решает все вычислительные проблемы в мире! Этот специализированный подход означает, что M делает какие-то вещи исключительно хорошо… а другие не делает вовсе.

Как это работает

С технической точки зрения M – функциональный язык, подобный F# и Haskell. M – язык в основном *чистый*, *высокоуровневый* и частично *ленивый*. Ниже мы поясним эти понятия.

Порядок вычислений

Задумайтесь, как работают формулы Excel:

	A
1	=A3+2
2	=A3+1
3	=A4*2
4	10

Рис. 1. Порядок вычислений в Excel

Сначала будет вычислена формулу в ячейке A3. Далее – в A1 и A2. Интерпретатор M работает аналогично. Он начинает с результата (выражения после оператора *in*), и определяет порядок вычислений, необходимый для того, чтобы в конце концов добраться до результата. Если есть шаги, где этот порядок не диктуется зависимостями, M получает право выбора. Можно сказать, что M идет от конца к началу.

Это означает, что вы можете располагать переменные M в любом порядке. Порядок влияет лишь на удобочитаемость кода. Оба приведенных ниже выражения являются допустимыми и выдают один и тот же результат. В обоих случаях M определит правильный порядок выполнения расчетов.

Листинг 1[3]

```
let
  Data = { 1, 2, 3 },
  Result = List.Transform(Data, each _ * 10)
in
  Result
```

Листинг 2

```
let
  Result = List.Transform(Data, each _ * 10),
  Data = { 1, 2, 3 }
```

---

[1] Научная парадигма – безоговорочно принятая научным сообществом модель научной деятельности. Любопытно, что термин *парадигма* ввёл относительно недавно – в середине XX века – Томас Кун, см. Структура научных революций.

[2] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 5): Paradigm. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

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

```
in
  Result
```

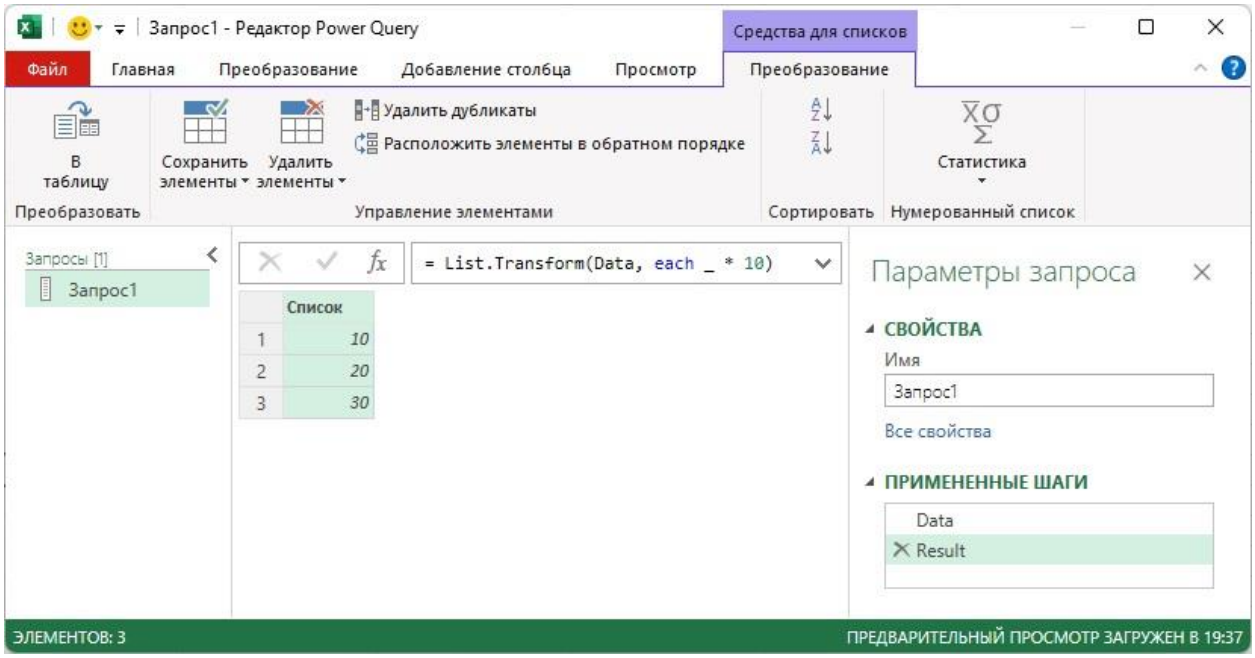

Рис. 2. Результат запроса 1 – список из трех значений

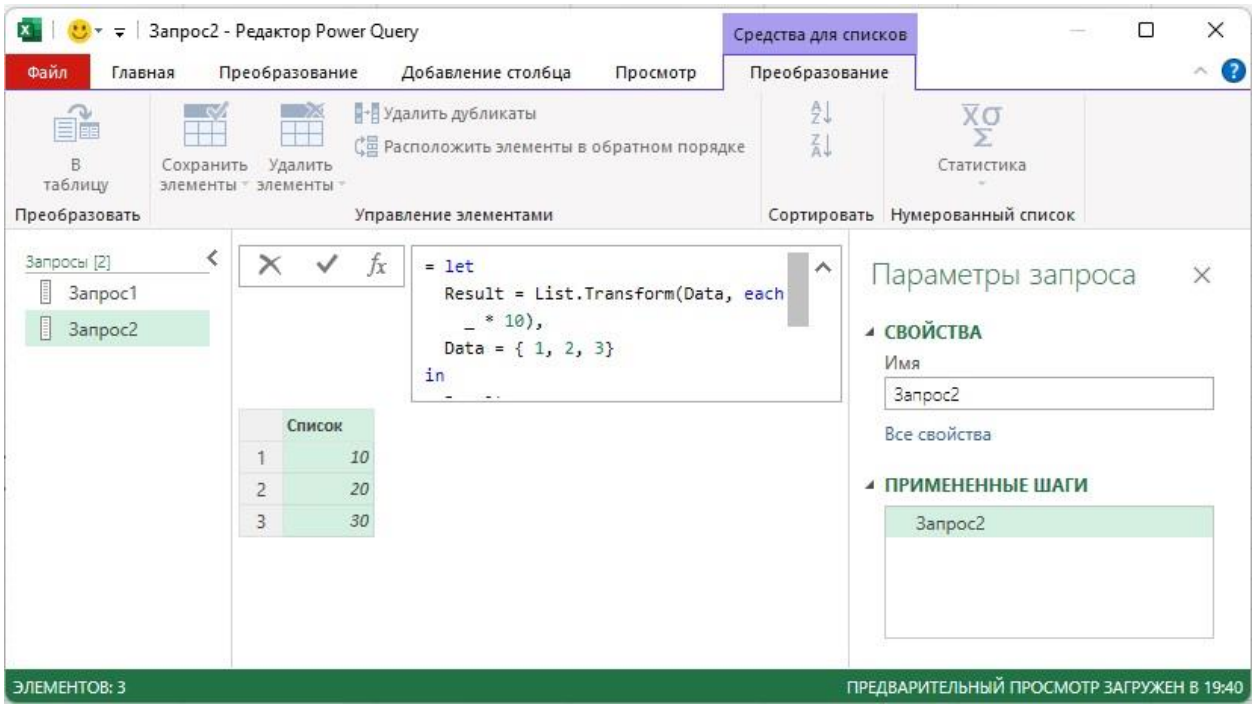

Рис. 3. Результат запроса 2 аналогичен, за исключением одной детали…

… вы так запутали PQ, что он не смог вывести на панель ПРИМЕНЕННЫЕ ШАГИ ни одного шага, и отразил запрос целиком. Это еще один аргумент (кроме удобочитаемости) в пользу некой логики расположения переменных в коде.

Частично ленивый

Когда интерпретатор M определяет порядок вычислений, что произойдет, если он встретит выражение, которое не влияет на результат?

Листинг 3

```
let
  Source = { "Bob", "Bill", "Sam" },
  Filtered = List.Select(Source, each Text.StartsWith(_, "B"))
in
```

Source

Здесь шаг *Filtered* не нужен для получения выходных данных (после *in*). Интерпретатор пропустит шаг *Filtered*. В его вычислении нет необходимости. Но зачем кому-то писать ненужный код?

Во-первых, как насчет тестирования? В редакторе Power Query на панели ПРИМЕНЕННЫЕ ШАГИ вы можете выбрать шаг, предшествующий последнему, чтобы проверить его выходные данные. Когда вы выбираете какой-то шаг, выражение *let … in* показывает результат работы кода с первого шага по выбранный.

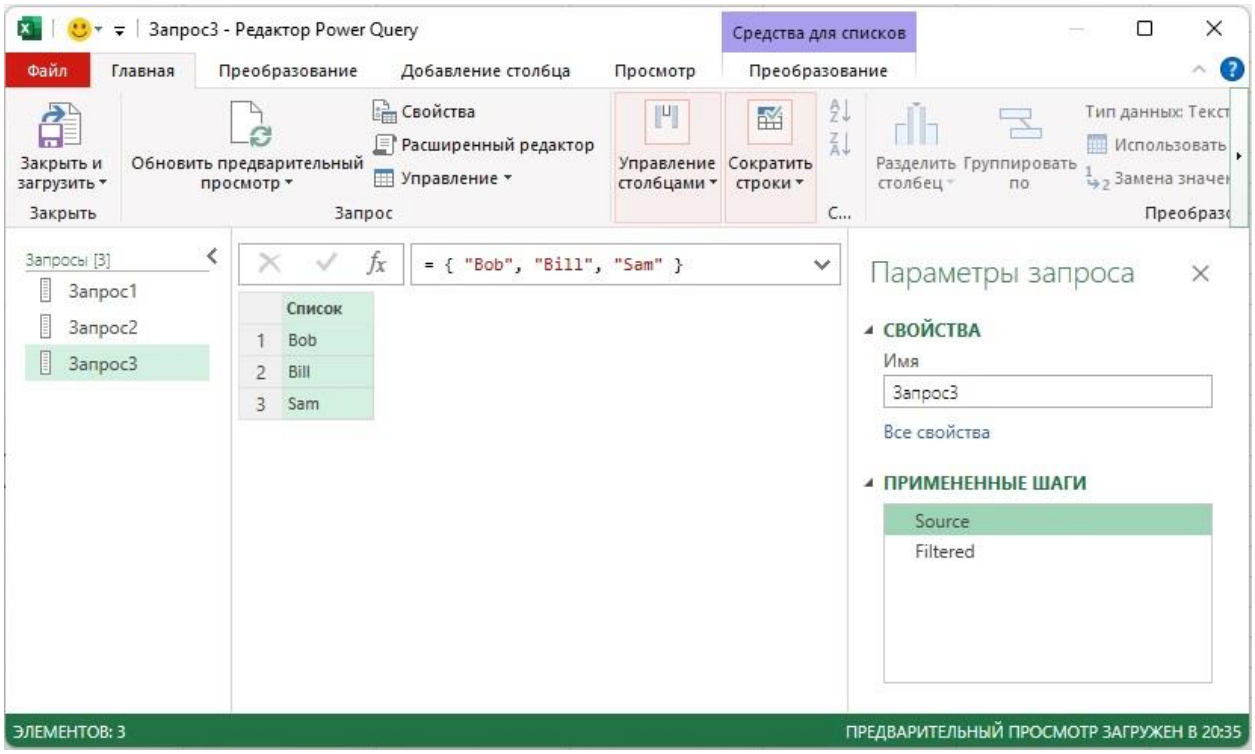

Рис. 4. Панель ПРИМЕНЕННЫЕ ШАГИ

Во-вторых, часть кода может быть востребована лишь в некоторых ситуациях.

Листинг 4

```
(x as logical) =>
let
  Odd = { 1, 3, 5 },
  Even = { 2, 4, 6 }
in
  if x then Even else Odd
```

Каждый раз, когда вызывается описанная выше функция, будет использована либо переменная *Odd*, либо *Even*, но не обе сразу. M вычисляет только ту переменную, которая будет использоваться *in* для получения результата. Если бы M был процедурным или объектно-ориентированным языком, обе переменные подлежали вычислению до *in*. Была бы проделана ненужная работа! К счастью, M не работает таким образом.

Мы только что показали, что переменные в выражениях *let* вычисляются *лениво*. Так же ведут себя вычисления в списках, записях и таблицах.

```
List.Count({ ComputeValue(), ComputeAnotherValue() })
```

Чтобы подсчитать количество элементов в списке, интерпретатору M не нужно рассчитывать содержимое этих элементов. ComputeValue() и ComputeAnotherValue() никогда не вызываются из-за ленивой оценки. Функция List.Count просто вернет значение 2, поскольку элементов в списке 2.

В следующем коде заработная плата (Salary) не вычисляется, поскольку ее значение не используется для формирования результата – PayDetails[Wage].

Листинг 5

```
let
  ComputeWeeklySalary = (wage) => ...,
  PayDetails =
  [
    Wage = 25.10,
    Salary = ComputeWeeklySalary(Wage)
  ]
in
   PayDetails[Wage]
```

Синтаксис [Wage = 25.10, Salary = ComputeWeeklySalary(Wage)] задает запись, где *Wage* – имя первого поля записи, 25.10 – значение первого поля, *Salary* – имя второго поля записи, *ComputeWeeklySalary(Wage)* – значение второго поля; в данном случае значение представлено функцией, которая вычисляется в другой строке кода. Подробнее мы обсудим это позже.

В M лениво вычисляются выражения *let*, списки, записи и таблицы. Для всего остального используются жадные вычисления. Вот почему мы говорим, что M ленив лишь отчасти.

Например, аргументы для вызова функции рассчитываются жадно – как только они встречаются.

Листинг 6

```
let
  Numbers = { 1, 2, 3 },
  Letters = { "A", "B", "C" },
  Choose = (chooseFirst as logical, first, second) =>
    if chooseFirst then first else second
in
  Choose(true, Numbers, Letters)
```

Поскольку при вызове функции *Choose* в качестве параметров используются как цифры, так и буквы, а параметры не вычисляются лениво, выражения для обеих переменных (*Numbers* и *Letters*) вычисляются, даже если функция будет использовать только одну из них.

Высокоуровневый язык

В листинге 6 оба списка (переменные *Numbers* и *Letters*) рассчитываются, потому что они передаются в качестве аргументов функции. Что, если нам действительно важно избежать расчета обоих списков, один которых не будет использован. Вместо того, чтобы передавать два списка в качестве параметров, передадим внешней функции *Choose* две вложенные функции, которые генерируют списки при их вызове. Функция *Choose* запустит только одну вложенную функцию – ту, что требуется в конкретном случае. Другая функция не вызывается, поэтому список, который она генерирует, не будет создан.

Листинг 7

```
let
  Numbers = () => { 1, 2, 3 },
  Letters = () => { "A", "B", "C" },
  Choose = (chooseFirst as logical, first, second) =>
    if chooseFirst then first() else second()
in
  Choose(true, Numbers, Letters)
```

Технически интерпретатор M по-прежнему рассчитывает оба списка (*Numbers* и *Letters*). Однако, теперь эти имена идентифицируют функции. Конечно, функция без параметров выглядит странно (спрашивается, зачем такая функция?), но синтаксис ничему не противоречит.

Возможность передавать функции в другие функции в качестве аргументов, а также возможность возвращать функцию, как результат работы функции, делает M высокоуровневым языком.

Потоковая семантика

С помощью написания кода в расширенном редакторе или командами в интерфейсе редактора Power Query, M создает цепочку действий. «Вызови функцию, которая извлечет данные из источника, передай ее выходные данные функции, которая применит преобразование, возьми результат и передай его другой функции, которая агрегирует данные» и так далее.

Однако такой способ передачи данных может быть неэффективным. Реализованный буквально, он требует, чтобы полный вывод первого шага был использовался для передачи следующему шагу. Затем второй шаг обрабатывает всё, что ему поступило на вход, чтобы создать полный набор выходных данных. Далее всё повторяется на третьем шаге. Каждый шаг цепочки может включать в себя (дублирующую) передачу всего набора данных от шага к шагу.

Зачем перебирать все данные для каждого преобразования (шага)? Не лучше ли выполнить все преобразования для первого элемента из источника, затем повторить процесс для второго элемента и так далее? Для списков и таблиц M поддерживает что-то подобное автоматически. От пользователя ничего не требуется.

Следующий код в качестве результата выводит первые три элемента огромного списка. Так как данные извлекаются из списка по мере необходимости, только первые три элемента пройдут через всю цепочку шагов. Поскольку извлекаются только три элемента, встроенная функция List.Generate создаст ровно три элемента. И это несмотря на то, что в строке для List.Generate указано создать последовательности чисел от 1 до 100 000 000. Было бы расточительно создавать столь большой список, чтобы использовать только три элемента. К счастью, с потоковой семантикой M этого не произошло.

Листинг 8

```
let
  Numbers = List.Generate(()=>1, each _ <= 100000000, each _ + 1),
  Filtered = List.Transform(Numbers, each _ * 2),
  Result = List.FirstN(Filtered, 3),
  #"Обращенный список" = List.Reverse(Result)
in
  #"Обращенный список"
```

M позволяет кодировать то, что должно быть сделано (логика вычислений), без необходимости задумываться, как это должно быть сделано (поток управления). M, как и другие функциональные языки, является декларативным, а не императивным. Вы сосредотачиваетесь на объявлении намерения; M заботится о выборе технических шагов для достижения этого намерения.[4]

Свёртывание запросов

Потоковая семантика повышает эффективность кода M. Еще более эффективным может стать перемещение обработки за пределы интерпретатора M – обратно к источнику. Пусть у вас есть база данных, содержащая миллион записей, которые фильтруются запросом. На выходе – одна строка. Чтобы выполнить обработку, Power Query извлечет миллион записей из базы данных, а затем применит фильтр. Если бы только M мог сообщить базе данных: «пришли мне отфильтрованные результаты».

Такое возможно (по крайней мере, иногда)! При свёртывании запросов интерпретатор M преобразует цепочку выражений в запрос, написанный на родном языке источника данных. Например, приведенное ниже выражение преобразуется в следующий за ним SQL. В итоге сервер базы данных отправляет в PQ только одну строку.

Листинг 9

```
let
   Source = Sql.Databases("some-server"),
   MillionRowTable = Source{[Schema="dbo",Item="MillionRowTable"]}[Data],
   Filtered = Table.SelectRows(MillionRowTable, each [ID] = 123)
in
```

[4] Подробнее см. Основные принципы программирования: императивное и декларативное программирование.

```
    Filtered
```

Листинг 10

```
select [_].[ID],
    [_].[Value]
from [dbo].[MillionRowTable] as [_]
where [_].[ID] = 123 and [_].[ID] is not null
```

Не каждый источник данных и функция M поддерживают свёртывание запросов. Уровень поддержки определяется кодом и редакцией M. Последовательность свёртываемых шагов будет свёрнута, но, как только в коде встретится шаг, который не может быть свёрнут, он и все шаги за ним будут обрабатываться на стороне интерпретатора M.

Вам ничего не нужно делать, чтобы включить свёртывание запросов. Однако при работе с большими наборами данных желательно так писать код, чтобы в начале шли шаги, поддерживающие свёртывание. Если Power Query увидит, что ваше выражение может быть изменено для повышения эффективности без изменения результата, он автоматически оптимизирует код. Например, изменения могут заключаться в перестановке шагов, чтобы собрать все шаги, поддерживающие свёртывание, в начале кода.

Чтобы увидеть, работает ли свёртывание для всего запроса, на панели ПРИМЕНЕННЫЕ ШАГИ щелкните правой кнопкой мыши на последнем шаге, и найдите опцию Просмотреть машинный запрос. Если пункт меню активен, значит запрос свёртывается. Если пункт не активен, можно выбрать более ранний шаг, и повторить процедуру. Так вы можете найти шаг, последний в череде свёртываемых.

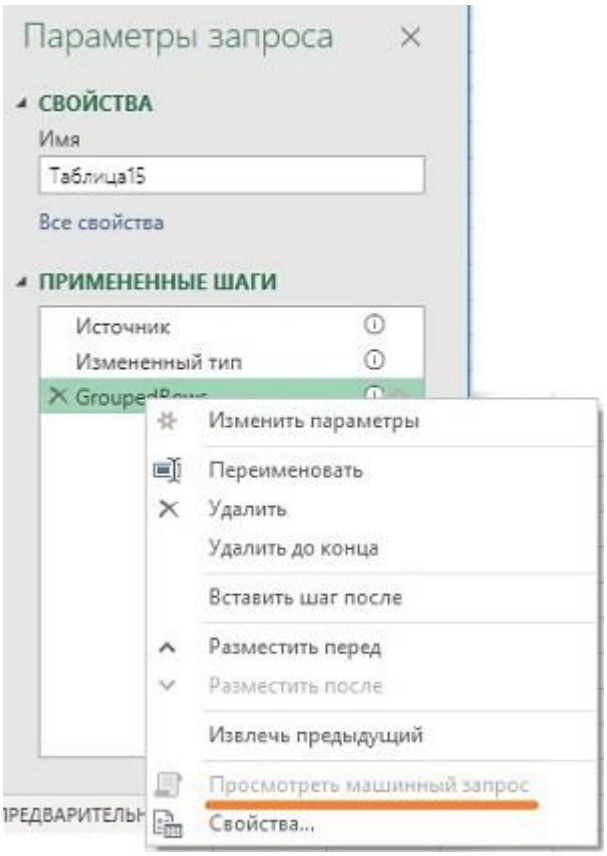

Рис. 5. Если опция *Просмотреть машинный запрос* активна, значит шаг участвует в свёртывании

К сожалению, не всё так гладко. Если пункт *Просмотреть машинный запрос* не активен, не факт, что свёртывание не работает. Для проверки свёртывания надежнее использовать трассировку и иные механизмы диагностики запросов.

Неизменяемые переменные

В предыдущей заметке мы упомянули, что переменная, однажды определенная в языке M, не может быть изменена. В M, когда переменная вычисляется в первый раз, ее значение фиксируется на всю жизнь. Переменные M неизменяемы.

Язык M в основном чистый

Функция считается чистой, если она:

1) Выдает один и тот же результат при одних и тех же входных данных. Например, addTwo(5), где addTwo() определяется как (x) => x + 2, будет возвращать 7 при каждом вызове.
2) Не имеет побочных эффектов. То есть функция лишь возвращает результат. Никакая глобальная переменная не меняется, никакие изменения не сохраняются в файле.

Природа языка M делает любое выражение чистым, если только оно не обращается к нечистой библиотечной функции или, возможно, не сталкивается с ошибкой на одном из своих подэтапов.

Библиотечные функции могут получить доступ к внешнему миру. Внешний мир не всегда детерминирован. Если вы используете библиотечную функцию для считывания данных из файла или базы данных, а затем повторно запускаете это выражение некоторое время спустя, вы можете получить разные результаты, даже если вы вызвали функцию с теми же аргументами. Почему? Возможно, файл или база данных изменились. Это изменение в выходных данных означает, что функция не всегда выдает один и тот же результат при одних и тех же входных данных, поэтому мы называем ее нечистой функцией.

Ошибки в подэтапе выражения также могут привести к тому, что оно будет недетерминированным и, следовательно, нарушит правило *один и тот же ввод приводит к одному и тому же результату*. Например, вы вычисляете выражение *a + b*. Интерпретатор M может вычислять *a* и *b* в любом порядке, потому что выражение не диктует конкретный порядок вычисления. Если однажды M сначала выполнит *a*, именно его ошибка станет ошибкой выражения *a + b*. В следующий раз, если b будет вычислен первым, ошибка b станет ошибкой выражения. Тот факт, что при одних и тех же входных данных возвращаемая ошибка может быть ошибкой из *a* или из *b*, означает, что выражение не всегда выдает один и тот же результат при одних и тех же входных данных.

Итак, M в основном чистый язык. Тот факт, что некоторые библиотечные функции не являются чистыми, – это нормально. Более того, если бы язык был на 100% чистым, его полезность была бы ограничена, потому что он не смог бы обращаться к данным из внешнего мира. Случайное отсутствие чистоты, возникающее из-за ошибок, обычно не является проблемой.

Заключение

Если вы привыкли к программированию в процедурной или объектно-ориентированной среде, некоторые из изученных моделей поведения M могут показаться чрезмерно ограничивающими. Почему бы не разрешить изменять переменные? Что такого особенного в чистых функциях?

Условия, которые мы обсуждали, позволяют M работать так, как он работает. Если, скажем, функции, которые вы пишете, могут иметь побочные эффекты, M может быть не в состоянии использовать отложенное вычисление, потому что, вроде бы, ненужные выражения могут иметь побочные эффекты.

Разрешение изменять переменные потребовало бы определенного порядка шагов, поскольку отклонение от этого порядка может привести к тому, что предполагаемая манипуляция переменными произойдет в неподходящее время. Это всего лишь пара примеров преимуществ, которые дает способ ведения дел в M.

По сути, если бы M работал как процедурный или объектно-ориентированный язык, вам пришлось бы делать то, что вы делаете на этих языках: императивно кодировать поток управления, вместо того, чтобы просто декларативно указывать, что вы сделали. Именно акцент на «что» делает функциональную природу M столь привлекательной.