Язык M Power Query. Управление глобальной средой, замыкание

В прошлый раз мы узнали, что код M обычно вычисляется в глобальной среде, состоящей из всех общих членов разделов и стандартной библиотеки. Кроме того, обычно мы не можем вводить дополнительные идентификаторы непосредственно в эту глобальную среду. Обычно... но не всегда! Сегодня мы узнаем об исключениях.

Но и это еще не всё. Знаете ли вы, что у M есть механизм для запоминания того, как получить доступ к переменным, которые позже выйдут за рамки видимости? Этот механизм называется *замыканием*. Он особенно полезен при генерации функций. Замыкание позволяет создавать объектоподобную конструкцию, которая поддерживает внутреннее частное состояние (private state) и взаимодействует с ней через общедоступный интерфейс (public interface; что-то вроде объекта из объектно-ориентированного программирования!).[1]

Предыдущая заметка Следующая заметка

Чистая окружающая среда

Вспомните первый пример из предыдущей статьи – запись, которая живет в мире без стандартной библиотеки, без других запросов (т.е. без других членов раздела).

```
[
 a = 1,
 b = 2,
 c = 3
]
```

Оказывается, нам не нужно этого представлять. Благодаря Expression.Evaluate мы можем сделать такую минималистичную среду реальностью. Expression.Evaluate – стандартная библиотечная функция – вычисляет произвольный M-код. Первый аргумент текст кода (в кавычках, как текст), второй опциональный аргумент – окружающая среда в виде записи.

Запрос 1[2]

```
Expression.Evaluate("10 + 20") // возвращает 30
```

То, что передается в Expression.Evaluate определяет всю среду, в которой вычисляется выражение. Указанное выражение вычисляется в безсекционном мире. Никаких разделов или членов разделов нет. Это имеет смысл, так как код M, который вы передаете, является выражением, а не документом раздела. Если вы хотите доказать, что разделы и глобальная среда отсутствуют, попробуйте…

Запрос 2

```
Expression.Evaluate("#sections")
```

…и обратите внимание, что вернется пустая запись.

По умолчанию глобальная среда, используемая при вычислении выражения, пуста. Стандартной библиотеки нет. Попробуйте…

Запрос 3

```
Expression.Evaluate("#shared")
```

… также вернется пустая запись.

Заселение глобальной окружающей среды

Вы можете добавить идентификаторы в глобальную среду. В этом и заключается цель необязательного второго аргумента *Expression.Evaluate*. Он указывается в виде записи, в которой имя каждого поля является именем идентификатора, добавляемого в глобальную среду, созданную Expression.Evaluate, а соответствующее значение поля является значением, связанным

---

[1] Заметка написана на основе статьи Power Query M Primer (Part 22): Identifier Scope II – Controlling the Global Environment, Closures. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query. – Здесь и далее примечания Багузина.

[2] Номер соответствует запросу в приложенном Excel-файле

с этим идентификатором. Подчеркнем: эта запись определяет всю глобальную среду, используемую при вычислении выражения, содержащегося в первом аргументе.

Ниже глобальная среда, используемая в Expression.Valuate, задается в виде идентификаторов *a* (значение 10) и *b* (значение 20).

Запрос 4

```
Expression.Evaluate(
  "a + b",
  [a = 10, b = 20] // определяет глобальную среду, используемую при вычислении предоставленного выражения
) // возвращает 30
```

Содержимое записи внедряется в глобальную среду, используемую Expression.Evaluate, и полностью ее определяет. Можно вывести глобальную среду с помощью выражения:

Запрос 5

```
Expression.Evaluate("#shared", [a = 10, b = 20]) // возвращает [a = 10, b = 20]
```

Рис. 1. Глобальная среда, заданная записью [a = 10, b = 20]

Если вы хотите вычислить выражение, которое ссылается на функции из стандартной библиотеки, необходимо добавить соответствующие функции в эту глобальную среду.

Запрос 6

```
Expression.Evaluate(""hi"")
```

…возвращает ошибку, так как Text.Upper отсутствует в используемой глобальной среде:

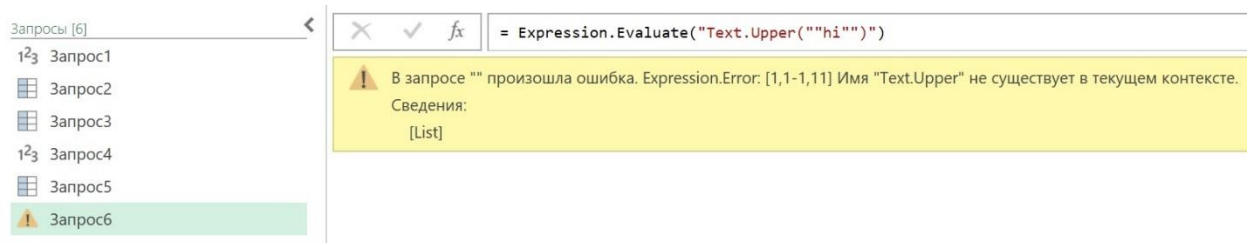

Рис. 2. Функции стандартной библиотеки не представлены в нашей глобальной среде

Запрос 7

```
Expression.Evaluate(""hi"")", [Text.Upper = Text.Upper]) // возвращает "HI"
```

Второй аргумент Expression.Evaluate дает точный контроль над средой, используемой при вычислении указанного выражения. Целью Expression.Evaluate является обработка произвольного M-кода, который в некоторых случаях может исходить из ненадежных источников (скажем, пользовательского ввода, хранящегося в базе данных). В подобных случаях ограничение функций, которые может использовать выражение, только теми, которые вы ожидаете от него, является важной мерой безопасности, помогающей защититься от атаки с внедрением кода.

Если вы хотите, чтобы выражение вычислялось в «нормальной» глобальной среде, то есть в той же глобальной среде, которую вы получили бы за пределами Expression.Evaluate, присвойте второму аргументу Expression.Evaluate значение #shared.

Запрос 8

```
Expression.Evaluate(""hi"")", #shared) // возвращает "HI"
```

#shared возвращает запись, содержащую по одному полю на каждый идентификатор (функцию, константу, тип) в глобальной среде. Это именно тот формат, который ожидает Expression.Evaluate, чтобы внедрить эти идентификаторы в глобальную среду.

Замыкания

Чтобы проиллюстрировать эту важную концепцию, давайте создадим глобальную среду, в которой Text.Lower фактически определяется как Text.Upper))

Запрос 9

```
Expression.Evaluate("Text.Lower(""Hello"")", [Text.Lower = Text.Upper]) // выводит "HELLO"
```

Продолжим в том же духе. Во внешней среде определим собственную функцию, которая форматирует текст. Эта функция использует Text.Lower и мы передадим её в Expression.Evaluate.

Запрос 10

```
let
   FormatText = (input as text) as text => Text.Lower(input),
   EvaluatedExpression = Expression.Evaluate(
      "FormatText(""Hello"")",
      [
         FormatText = FormatText,
         Text.Lower = Text.Upper
      ]
   )
in
   EvaluatedExpression
```

Итак, глобальная среда, созданная Expression.Evaluate содержит два идентификатора: FormatText, использующий Text.Lower и Text.Lower, сопоставляемый с Text.Upper стандартной библиотеки. Если внутри Expression.Evaluate мы вызовем FormatText("Hello"), как это сделано в приведенном выше примере, что будет на выходе?

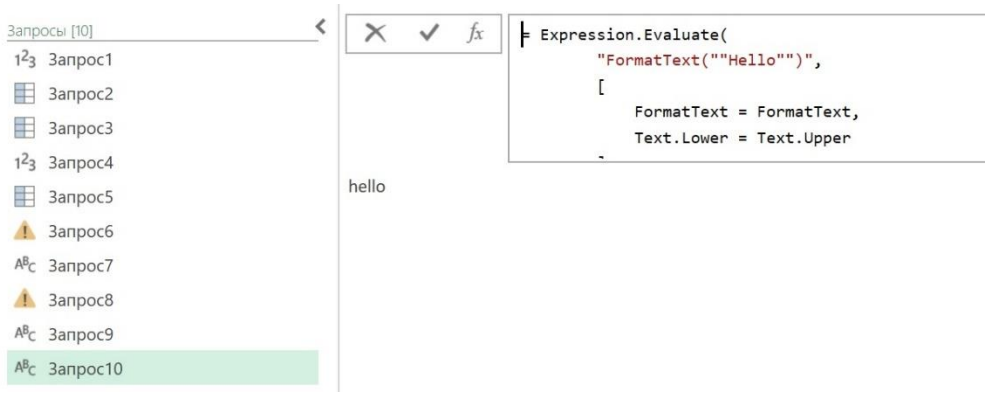

Рис. 3. Неожиданный результат

Странно… Разве не должно быть "HELLO"? Ведь внутри Expression.Evaluate Text.Lower ссылается на функцию Text.Upper, переводящую текст в верхний регистр, а FormatText использует Text.Lower.

Это то, что мы хотели продемонстрировать: FormatText не использует Text.Lower, присутствующий в глобальной среде Expression.Evaluate (которая использует текст в верхнем регистре); FormatText использует Text.Lower, который существовал в среде, где был определен FormatText.

Запоминание определяющего контекста

В M функции «запоминают» контекст, в котором они были первоначально определены, даже если позже они вызываются из другого контекста. В теории языков программирования этот прием называется замыканием (closure). Когда FormatText определен, «захватывается» контекст, который позволяет FormatText использовать Text.Lower, существовавший в его глобальной среде (текст в нижнем регистре). Это справедливо, несмотря на то, что позже, когда вызывается FormatText, Text.Lower, который находится в глобальной среде в этот момент времени, является тем, который выводит прописные буквы. Этот последний Text.Lower не имеет отношения к

FormatText, он вообще ему не нужен. Text.Lower, используемый FormatText, появился ранее, и используется благодаря замыканию.

Убедитесь в этом, удалив второй Text.Lower из кода:

Запрос 11

```
let
   FormatText = (input as text) as text => Text.Lower(input),
   EvaluatedExpression = Expression.Evaluate(
      "FormatText(""Hello"")",
      [
         FormatText = FormatText
      ]
   )
in
   EvaluatedExpression // вернет "hello"
```

Ничего не изменилось.

Замыкания не ограничиваются случаями использования Expression.Evaluate. Ниже создана функция Generator, которая принимает почасовую ставку заработной платы и возвращает функцию, которая берет отработанные часы, умножает ее на почасовую ставку (которая была «захвачена» замыканием), и затем возвращает результат.

Запрос 12

```
let
  Generator = (hourlyRate as number) as function =>
    (hoursWorked as number) as number => hourlyRate * hoursWorked,
  FifteenPerHour = Generator(15)
in
  FifteenPerHour(40) // возвращает 600
```

Чтобы понять, что происходит, давайте пойдем в обратном направлении. FifteenPerHour после in – это переменная, указывающая на функцию. Мы можем вызвать FifteenPerHour, передав ей количество отработанных часов, и функция вернет сумму заработной платы.

Откуда взялась функция FifteenPerHour? Вместо того, чтобы определить её как мы привыкли, например…

FifteenPerHour = (hoursWorked) = > hoursWorked * 15

…она определяется как результат функции. Как бы странно это ни звучало: функция может возвращать функцию, а возвращаемая функция может храниться в переменной.[3]

---

[3] Мне проще это осознать в следующей логике. Есть исходная функция двух переменных. Задавая одну переменную, мы получаем функцию одной (второй, незаданной) переменной. На этом этапе функция вернула формулу. Задавая вторую переменную, мы получим результат. Вот как поясняет замыкание ChatGPT: Замыкание (closure) в M Power Query представляет собой функцию, которая сохраняет ссылки на переменные в своей окружающей области видимости, даже после того, как эта окружающая область видимости уже завершилась. Это позволяет замыканию обращаться к этим переменным и продолжать использовать их значения. Например,

Запрос 13

```
let
   multiplyBy = (factor) =>
      let
         multiply = (x) => x * factor
      in
         multiply,
   multiplyByTwo = multiplyBy(2),
   multiplyByThree = multiplyBy(3)
in
   {
```

Generator – функция вызываемая для создания функции FifteenPerHour, принимает аргумент, указывающий почасовую ставку заработной платы. Когда эта функция генерирует функцию, которая затем вычислит общую заработную плату (со ставкой пятнадцать в час), эта ставка «фиксируется» замыканием. Это позволяет сгенерированной функции FifteenPerHour ссылаться на Generator(15) на протяжении всего времени жизни сгенерированной функции FifteenPerHour.

При последующем вызове FifteenPerHour(40) почасовая ставка нигде не видна. Она не находится в контексте в момент вызова. Тем не менее, FifteenPerHour помнит, что ставка = 15, благодаря замыканию.

Функции преобразователи

Замыкания могут быть особенно удобны при преобразовании значений. Допустим, у вас есть список, который вы хотите преобразовать с помощью функции List.Transform. Вторым её аргументом является функция-преобразователь. Она будет вызываться один раз для каждого элемента списка. Каждый раз функции-преобразователю будет передаваться текущий элемент списка в качестве единственного аргумента, который нужно преобразовать и вернуть.

Запрос 15

List.Transform({ "-abc-", " def "}, Text.Trim) // возвращает { "-abc-", "def" } (второй элемент списка не содержит пробелов)

Но что, если вы хотите настроить поведение функции-преобразователя? Например, в Text.Trim задать необязательный второй аргумент, чтобы метод обрезал дефисы? Одним из вариантов является построение функции с одним аргументом, которая возвращает функцию, вызывающую Text.Trim, со вторым жестко закодированным аргументом.

Запрос 16

```
let
  TrimHyphens = (textToTrim as text) as text => Text.Trim(textToTrim , "-")
in
  List.Transform({ "-abc-", " def "}, TrimHyphens ) // возвращает { "abc", " def " }
```

Это работает, но новый код жестко запрограммирован на обрезку одного конкретного символа. Благодаря замыканию можно сделать функцию-преобразователь настраиваемой:

Функция TrimCharacter()

```
(characterToTrim as text) as function =>
   (textToTrim as text) as text =>
      Text.Trim(textToTrim , characterToTrim)
```

При вызове TrimCharacter("-") возвращается функция с одним аргументом – текстом для обрезания. Если теперь передать ей текст, то указанный символ ("-") в начале и конце строки будет удален.

Запрос 18

```
let
```

```
    multiplyByTwo(5),
    multiplyByThree(5)
  }
```

Этот код даже можно сократить, заметив, что функция multiply используется лишь в одном месте. Станет ли код от этого проще для понимания!? Не уверен…

Запрос 14

```
let
  multiplyBy = (factor) => (x) => x * factor,
  multiplyByTwo = multiplyBy(2),
  multiplyByThree = multiplyBy(3)
in
  {
    multiplyByTwo(5),
    multiplyByThree(5)
  }
```

```
  TrimHyphens = TrimCharacter("-") // генерирует функцию, обрезающую дефисы
in
  TrimHyphens("-abc-") // вызывает сгенерированную функцию, возвращая "abc"
```

То же самое можно сделать и в одну строку:

Запрос 19

```
TrimCharacter("-")("-abc-") // генерирует функцию, обрезающую дефисы, затем вызывает ее, передавая текст, который нужно обрезать, и возвращает "abc"
```

Подключив новую функцию TrimCharacter к нашему сценарию преобразования списка, мы можем использовать ее для создания функции с одним аргументом, которая отсекает любой символ, который мы укажем. Ниже мы используем функцию TrimCharacter, чтобы обрезать дефисы. List.Transform довольна: ей передается функция с одним аргументом, которая принимает текущий элемент списка в качестве входных данных и возвращает преобразованный результат. Мы счастливы, потому что используем универсальную функцию для создания функции-преобразователя для List.Transform))

Запрос 20

```
let
   TrimCharacter = (characterToTrim as text) as function =>
      (textToTrim as text) as text => Text.Trim(textToTrim , characterToTrim)
in
   List.Transform(
      { "-abc-", " def "},
      TrimCharacter("-") // генерирует функцию с одним аргументом, которая обрезает дефисы, а затем вызывается List.Transform для каждого элемента списка
   ) // outputs { "abc", " def " }
```

Все это благодаря замыканиям, которые позволяют сгенерированной функции «запоминать» значение characterToTrim, которое было передано в функцию генератора.

Своего рода объектно-подобное поведение

Поддержка замыканий в языке M обеспечивает тип объектно-подобного поведения: способность создавать и взаимодействовать с программными структурами, которые предоставляют общедоступные интерфейсы и поддерживают частное внутреннее состояние (private internal state).

Поскольку значения M не могут быть изменены, каждое взаимодействие создает новую структуру, которая должна быть использована для следующего взаимодействия. (В отличие от этого, объектно-ориентированный язык позволяет напрямую манипулировать одним и тем же экземпляром объекта, причем несколько раз.)

Давайте представим себе более сложный сценарий, связанный с расчетом заработной платы. Нам нужна объекто-подобная сущность «табель учета рабочего времени», которая позволяет нам устанавливать и изменять почасовую ставку, записывать отработанные часы и получать общую сумму заработной платы.

Запрос21

```
let
  InitialWage = Timecard(10), // создаёт новую карточку учета рабочего времени и устанавливает начальную почасовую ставку заработной платы = 10
  FirstWeek = InitialWage[RecordHoursWorked](40), // запись работы за первую неделю = 40 часов
  PayRaise = FirstWeek[SetRate](25), // повышает ставку заработной платы за последующую работу
  SecondWeek = PayRaise[RecordHoursWorked](40), // запись работы за вторую неделю = 40 часов
  ThirdWeek = SecondWeek[RecordHoursWorked](35), // запись работы за третью неделю = 35 часов
  TotalWages = ThirdWeek[TotalWages]() // возвращает общую заработанную заработную плату
in
  TotalWages // возвращает 2275
```

Код начинается с инициализации табеля учета рабочего времени со ставкой заработной платы, равной 10. Возвращается запись, состоящая из нескольких полей, каждое из которых содержит функцию. Эти функции образуют публичный интерфейс для взаимодействия с табелем учета рабочего времени:

SetRate	Function
RecordHoursWorked	Function
TotalWages	Function

Рис. 4. Запись публичного интерфейса

Этот новый табель учета рабочего времени, сохраненный как InitialWage, используется для записи отработанных часов FirstWeek с помощью функции RecordHoursWork. Этот метод возвращает новый табель учета рабочего времени с тем же набором полей, что и предыдущая карточка, только с внутренним состоянием карты, обновленным для отражения отработанных часов.

Далее идет PayRaise, которая записывается путем вызова SetRate из записи табеля учета рабочего времени FirstWeek. Результатом этой строки является еще один табель учета рабочего времени, с обновленной (повышенной) ставкой заработной платы. Затем отработанные часы SecondWeek и ThirdWeek записываются в табеле учета рабочего времени путем вызова RecordHoursWork из предыдущего шага.

Наконец, шаг TotalWages вызывает функцию TotalWages в табеле учета рабочего времени ThirdWeek. В отличие от других методов карты, этот не возвращает новый табель учета рабочего времени. Он напрямую возвращает сумму общей заработной платы в виде числа.

Табель учета рабочего времени поддерживает необходимое внутреннее состояние для расчета общей заработной платы. То, как он это делает, скрыто от нас, потребителей, и поэтому мы не можем им манипулировать. Наша способность взаимодействовать с табелем учета рабочего времени определяется интерфейсом общедоступных методов, представленных в его записи.

Внутренняя работа

Рассмотрим подробнее, как работает табель учета рабочего времени.

Функция Timecard

```
(initialRate as number) =>
 let
   NewTimecard = (state as record) =>
    [
     SetRate = (rate as number) =>
       @NewTimecard(state & [Rate = rate]),
     RecordHoursWorked = (hours as number) =>
       @NewTimecard(state & [TotalWages = state[TotalWages] + state[Rate] * hours]),
     TotalWages = () =>
       state[TotalWages]
    ]
in
 NewTimecard([Rate = initialRate, TotalWages = 0])
```

Функция Timecard принимает начальную ставку заработной платы, которую она хранит в записи *state*. Поля этой записи, невидимые для внешнего мира, являются приблизительным эквивалентом частных полей объекта.

Затем запись состояния передается в функцию, определенную в Timecard с именем NewTimecard. Эта функция возвращает запись, содержащую «публичный интерфейс» карты времени, состоящий из трех функций, которые определяют то, как внешний код может взаимодействовать с картой.

Две из этих функций, SetRate и RecordHoursWorked вызывают один и тот же метод NewTimecard, передавая ему запись состояния, которая была соответствующим образом обновлена для отражения новой ставки заработной платы или отработанных часов. Это стало возможным благодаря замыканиям: извне, когда вы вызываете SetRate или RecordHoursWorked, у вас нет

знаний или доступа к этому методу NewTimecard или существующей записи состояния, но, к счастью, эти две функции делают это благодаря замыканиям.

Хотя это интересно, обычно мы не пишем M-код таким образом. Нам это не нужно. Обычно наш интерес заключается в создании набора данных, что достигается путем передачи данных через цепочку вызовов функций. Напротив, постепенное «манипулирование» состоянием путем выполнения серии вызовов функций (как в последнем примере) более применимо к случаям, когда что-то необходимо программно настроить для последующего использования.

Что именно? Во-первых, свертывание запросов. При реализации поддержки свертывания запросов Power Query требуется интерфейс для интерактивного взаимодействия с реализацией, подготавливая ее к созданию данных. Кроме того, ваша реализация, вероятно, имеет внутреннее состояние, которое инфраструктура свертывания Power Query не должна касаться или о котором не должна знать. Это именно та ситуация, для которой этот стиль кодирования M хорошо подходит.

В следующий раз

Стиль объектно-подобного M-кода является основополагающим для реализации свертывания запросов... И освещение основных концепций реализации свёртывания – тема следующего поста. Возможно, вы не планируете когда-либо реализовывать свертывание запросов. Однако, знание основ его реализации поможет вам лучше понять, как он работает, и, учитывая важность свертывания запросов в мире Power Query, расширение знаний в этой области является хорошей инвестицией.