Язык M Power Query. Пользовательские типы

В предыдущих заметках мы рассказали об основах системы типов и аспектах типов. В настоящей заметке будут представлены сложные типы (также известные как пользовательские или производные). Основное внимание будет уделено синтаксису и правилам соответствия. А вот обсуждение того, как M работает с этими типами, мы перенесем в следующий пост.[1]

Предыдущая заметка Следующая заметка

Что значит *соответствие*? Утверждение о том, что значение соответствует типу, означает, что значение может быть описано данным типом, или, другими словами, значение совместимо с типом. Число 1 *соответствует* типам *number*, *nullable number*, *anynonnull* и *any*, поскольку каждый из этих типов может быть использован для описания этого значения. Правила соответствия – это правила, используемые для определения того, соответствует ли значение типу.

Список

type list можно записать, как *type { any }*. Фигурные скобки указывают на то, что определяется тип списка. Между этими фигурными скобками находится тип элемента списка, который в данном случае является типом *any*. Тип элемента описывает значения, составляющие список. Все значения в списке должны соответствовать типу элемента списка. Например, список *type { number}* должен содержать только значения *type number*. Никаких *nulls*. Ничего иного. (Интересно, что M не заставляет следовать этому правилу, но подробнее об этом в следующий раз.)

По умолчанию список создается с *type list*, который является эквивалентом *type { any }*. Тип элемента списка можно изменить, приписав списку пользовательский тип списка. Сначала нужно определить пользовательский тип списка, указав тип элемента отличный от *any*…

type { number } // описывает список чисел

type { text} // описывает список строк

type { { function } } // описывает список, содержащий список функций

…затем добавьте новый тип в список:

Листинг 1[2]

```
let
   MyList = { 1, 2, 3 } // список по умолчанию создается с типом { any }
in
   Value.ReplaceType(MyList, type { number }) // возвращает список с типом элементов number
```

Как вы, возможно, заметили, в редакторе запросов, когда тип отображается в окне предварительного просмотра, появляется только имя базового типа. Например, *type { number }* будет отображаться как *type list*. Если вы хотите отобразить пользовательские сведения типа, используйте библиотечные функции. Для *type list* можно использовать функцию Type.ListItem:

Листинг 2

```
let
   SomeType = type { date }
in
   Type.ListItem(SomeType) // возвращает тип date
```

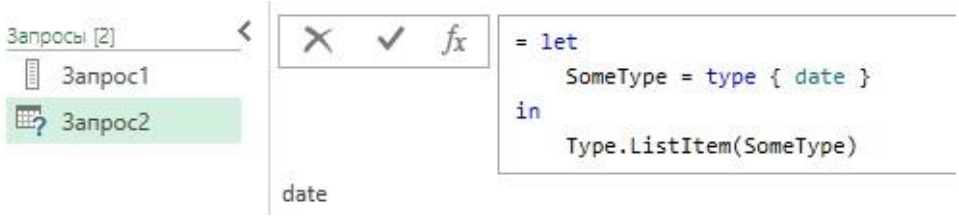

Рис. 1. Функция *Type.ListItem* позволяет извлечь тип элемента списка

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 18): Type System III – Custom Types. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

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

Запись

Пользовательский тип записи определяется с использованием следующего синтаксиса, где имена полей записи и (необязательно) типы указываются в квадратных скобках. Если полю не присвоен тип, его тип по умолчанию = *any*.

type [ItemCode = text, Amount]

Эта запись имеет два поля, *ItemCode* и *Amount*, содержащие значения, соответствующие типу *text* (поскольку он явно указан) и типу *any* (по умолчанию).

Все записи имеют пользовательский тип – либо назначенный системой, либо определенный вами. Когда создается новая запись, система автоматически присваивает ей пользовательский тип, в котором перечислены поля в записи с типом каждого поля *any*. Например, записи…

[Name = "Joe", Age = 50]

…автоматически присваивается тип…

type [Name = any, Age = any]

Так же, как и *type list*, вы можете изменить установки по умолчанию, определив новый пользовательский тип, а затем приписав его записи:

Листинг 3

```
let
   Person = [Name = "Joe", Age = 50], // по умолчанию type [Name = any, Age = any]
   NewType = type [FullName = text, Age = number]
in
   Value.ReplaceType(Person, NewType) /* возвращает запись
   [Name = "Joe", Age = 50] с типом type [Name = text, Age = number]*/
```

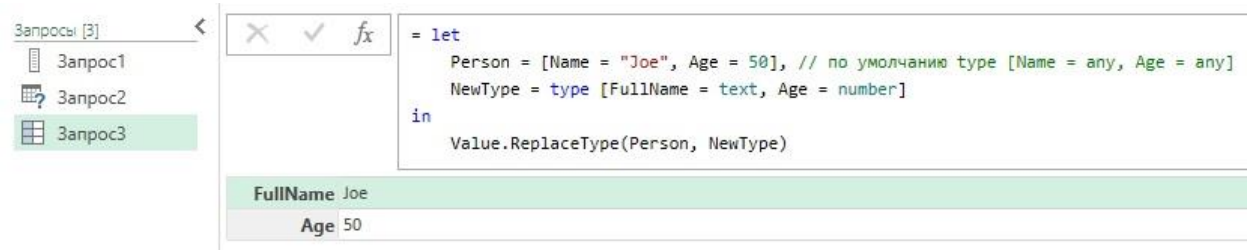

Рис. 2. Пользовательский тип записи

Информация о новом типе применяется к полям записи на основе их позиций. То есть определение первого поля в новом типе заменяет текущее определение типа для первого поля записи, второе поле в типе заменяет тип для второго поля записи и так далее. В отличие от этого, при проверке соответствия значения записи типу записи поля должны совпадать по имени, а позиция игнорируется.

Движок M не проверяет имена полей во время приписывания. Вы можете приписать записи, первое поле которой называется *FieldA*, тип, первое поле которого равно *Field1*. Однако, лучше не делать этого. Важно, чтобы имена полей оставались неизменными при присвоении нового типа; в противном случае может возникнуть непредвиденное поведение, поскольку другие части кода могут ожидать, что имена полей для типа и значения будут синхронизированы.

Подобно *type list*, при присвоении пользовательского типа записи движок не проверяет типы полей на совместимость. Вы могли бы приписать записи, созданной в предыдущем примере, *type [A = logical, B = logical]*, и M с радостью применит новый тип, несмотря на то, что значения полей *Joe* и *50* не относятся к типу *logical*.

Для пользовательских типов записей могут использоваться следующие элементы синтаксиса, создающие абстрактные типы:

type [FirstName = text, optional LastName = text, Age = number]

type [Amount = number, ...]

В первом случае имеется необязательное поле. Во втором – ... (троеточие) в конце списка полей. В этом контексте ... известно как маркер открытой записи. Открытая запись допускает любое количество дополнительных полей (включая ноль) сверх указанных.

Почему абстрактные типы?

Никакое значение не может иметь абстрактный тип. Поскольку типы записей с необязательными полями и те, которые открыты, являются абстрактными, вы никогда не увидите значение записи, тип которой имеет любой из этих типов. Если тип конкретного значения не может быть абстрактным, какой смысл определять тип именно так?

Описание ожиданий или, другими словами, классификация значений.

У вас может быть функция, которая ожидает в качестве своего аргумента запись, содержащую числовое поле *Amount*. Если бы вы определили аргумент функции как *type [Amount = number]*, вы бы сказали, что функции должны передаваться только записи, содержащие одно поле с именем *Amount* типа *number*. Если вы измените тип на *type [Amount = number, …]*, вы бы сказали, что передаваемые записи должны содержать числовое поле *Amount*, но также могут содержать любое количество других полей. В первом типе указывается именно то, что ожидается (не больше и не меньше), в то время как второй тип определяет минимальное требование (по крайней мере, столько, но разрешено больше). Какой подход является подходящим, зависит от вашей ситуации.

Типы record & []

type record сам по себе является эквивалентом типа [...], который представляет собой абстрактный тип, описывающий записи, имеющие любое количество полей, включая отсутствие полей. Это означает, что все записи совместимы с *type record*, хотя, поскольку он является абстрактным, ни одно значение записи никогда не может быть непосредственно *type record*.

И таки да, можно определить тип записи, который описывает пустую запись – запись, содержащую нулевые поля:

type []

Вряд ли вы будете определять этот тип или вручную создавать пустые записи. Но, если в коде программно будут удалены поля из записи, получится пустая запись, которая будет иметь именно такой тип. Так что есть причина для существования такого типа.

Таблица

Основываясь на вышеизложенном, вы, вероятно, сможете понять смысл следующего синтаксиса для определения пользовательского типа таблицы:

type table [SomeColumn = text, AnotherColumn = any, YetOneMore]

Часть, заключенная в квадратные скобки, определяет тип строки таблицы. По сути, тип *строки* – это тип *записи* с двумя правилами: типы строк не могут иметь необязательных полей или быть открытыми, за исключением одного встроенного типа (подробнее об этом чуть позже).

Вы могли бы возразить, что возможность сделать что-то вроде *type table [Amount = number, ...]* была бы полезна. Это позволило бы вам сказать: «Я ожидаю таблицу, содержащую столбец *Amount*, типа *number*, и другие столбцы». Но такой синтаксис не поддерживается. Если бы были разрешены таблицы, содержащие различное количество столбцов, а также открытые типы строк, могли бы возникнуть неровные таблицы. Т.е., таблицы, в которых каждая строка имела бы разное количество столбцов. Так, в строке 1 может быть 3 дополнительных столбца, в строке 2 нет дополнительных столбцов, в строке 3 может быть 1 дополнительный столбец и так далее. M не поддерживает неровные таблицы.

Как и в случае с *record*, тип таблицы по умолчанию содержит все имена столбцов, для которых задано значение *any*. Функция…

#table({"EmployeeID", "Type", "Wage"}, {...})

… создает таблицу следующего типа…

type table [EmployeeID = any, Type = any, Wage = any]

Базовый тип всех таблиц – тип *table* – является абстрактным. Его тип строки – пустая открытая запись (исключение, когда тип таблицы может иметь открытый тип строки). Таким образом, *type table* совместим с таблицами, строки которых содержат любое количество столбцов (включая отсутствие столбцов), что делает его совместимым со всеми таблицами.

Ключи таблицы

Типы таблиц включают в себя то, чего нет ни в одном другом типе: определение ключей таблицы. Например, следующий тип таблицы включает первичный ключ для столбца *OrderID*:

Листинг 4

```
let
   BasicTableType = type table [OrderID = number, Total = number, Shipped = logical]
in
   Type.AddTableKey(BasicTableType, {"OrderID"}, true)
```

Тот факт, что тип таблицы содержит ключи, означает, что два типа таблиц с одинаковыми столбцами и типами столбцов могут иметь разные типы, если ключи, определенные для них, различаются.

Стандартная библиотека также содержит функции, позволяющие добавлять и заменять ключи в самих таблицах (а не в табличных типах, как в примере выше).

Листинг 5

```
let
   SomeTable = #table({"Col1", "Col2"}, {{"a", "b"}})
in
   Table.AddKey(SomeTable, {"Col1"}, true)
```

О функциях, которые работают с ключами таблиц, можно думать, как о ярлыках, которые принимают таблицу, добавляют или заменяют в ней ключи, приписывают измененный тип, и возвращают измененную таблицу. Приведенный выше пример более лаконичен, чем кодирование всех этих шагов вручную, что могло бы выглядеть примерно так:

Листинг 6

```
let
   SomeTable = #table({"Col1", "Col2"}, {{"a", "b"}}),
   StartingType = Value.Type(SomeTable),
   UpdatedType = Type.AddTableKey(StartingType, {"Col1"}, true)
in
   Value.ReplaceType(SomeTable, UpdatedType)
```

Подобно *type record*, сведения о типах столбцов в строке таблицы применяются позиционно (а не по именам столбцов). Таким образом, тип для первого столбца, определенного в новом типе, будет применен к первому столбцу в таблице, определение второго столбца типа ко второму столбцу в таблице и так далее.

Как и в случае с *record*, важно, чтобы имена столбцов оставались неизменными в описании типов, даже если движок M к именам не чувствителен. В противном случае может возникнуть непредвиденное поведение. И, пожалуйста, не используйте приписывание для переименования столбцов таблицы!

Ниже приведены примеры проблем при несоблюдении этих правил. В некоторых случаях M видит столбец со своим старым именем (листинг 7), в других – с новым (листинг 8). Вы же не хотите исследовать, что именно произойдет в вашем случае!?

Листинг 7

```
let
   MyTable = #table({"Col1"}, {{"Joe"}}),
   TypeChanged = Value.ReplaceType(MyTable, type table [Name = any])
   // возвращает таблицу, в которой, как думает разработчик, есть столбец с именем Name
in
   Table.SelectRows(TypeChanged, each [Name] = "Joe")
```

```
/* попытка использовать новое имя столбца приводит к ошибке:
Expression.Error: Поле "Name" записи не найдено.*/
// Table.SelectRows(TypeChanged, each [Col1] = "Joe") // старое имя столбца – работает
```

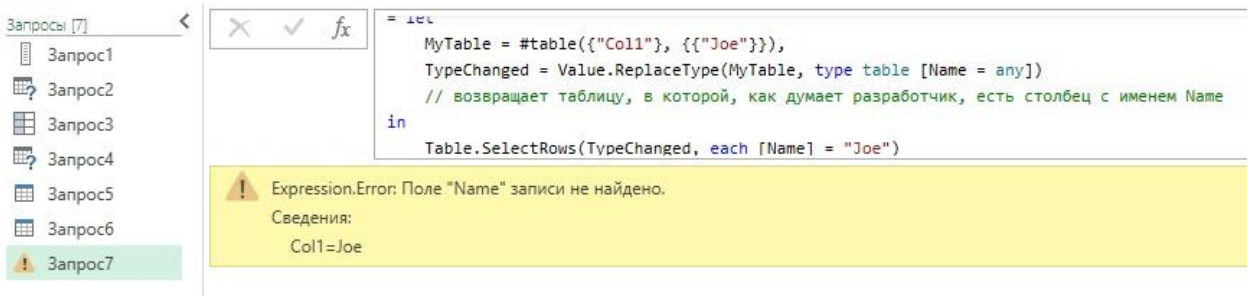

Рис. 3. Приписывание типа не переименовывает столбец

Листинг 8

```
let
   MyTable = #table({"Col1"}, {{"Joe"}}),
   TypeChanged = Value.ReplaceType(MyTable, type table [Name = any])
in
   TypeChanged[Name] // новое имя работает
   // TypeChanged[Col1] // старое имя возвращает ошибку:
   //Expression.Error: Столбец "Col1" таблицы не найден.
```

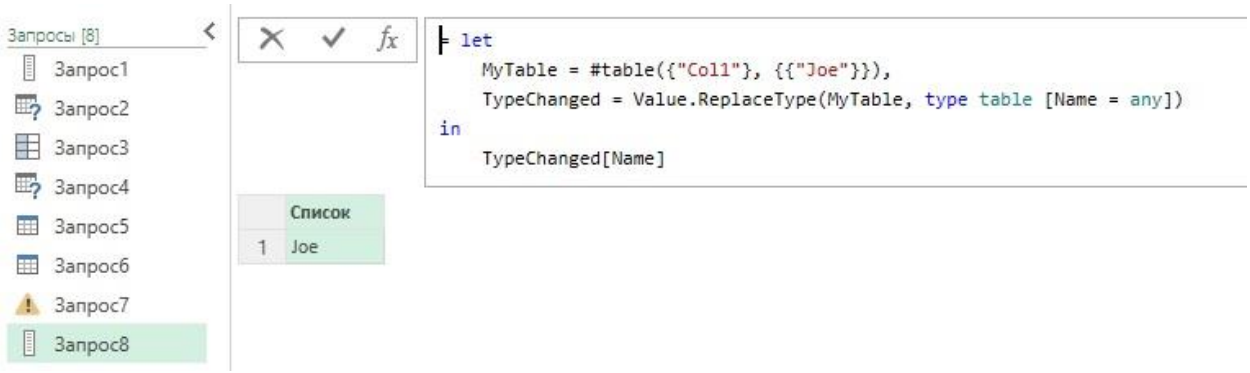

Рис. 4. А здесь приписывание типа переименовывает столбец

Как и для *list* и *record*, движок M не проверяет, совместимы ли новые типы столбцов таблицы со значениями, фактически в них содержащимися. Это выглядит странно, но подробнее об этом в следующей заметке.

Функция

Синтаксис для определения пользовательских типов функций:

```
type function(name as text, age as number) as record

type function(amount as number, optional tax as number) as number
```

При определении *функции* указание типа параметра и типа возвращаемого значения являются необязательными. Однако, при определении *типа* функции должны быть указаны оба типа:

Листинг 9

```
/* допустимое определение функции – не указаны тип аргумента
и тип возвращаемого значения функции */
(total) => total * 0.1
```

Листинг 10

```
/* недопустимое определение типа функции – должны быть
определены тип аргументов и тип возвращаемого значения */
type function(total)
```

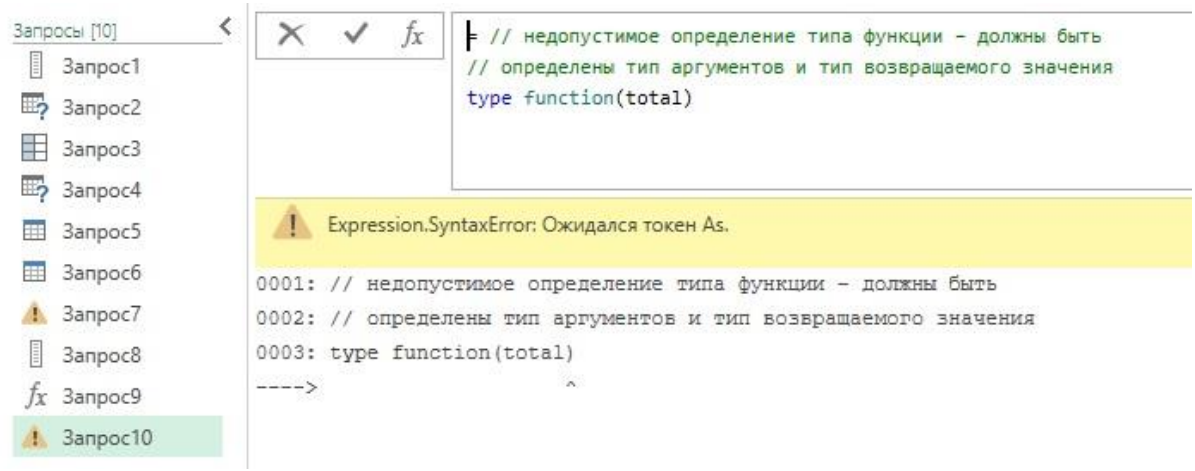

Рис. 5. При определении *типа* функции должны быть указаны, и типы аргументов, и тип значения

Листинг 11

```
/* допустимое определение типа функции
указаны, как тип аргумента, так и тип функции */
type function(total as number) as number
```

Когда функция определена, пользовательский тип, автоматически созданный для нее, по умолчанию для всех пропущенных утверждений типа устанавливает значение *any*. Например, для функции…

(total) => total * 0.1

… установлены типы…

type function (total as *any*) as *any*

При присвоении функции нового типа количество определяемых параметров должно совпадать с числом исходных параметров. Кроме того, новый тип не может изменить обязательные параметры на необязательные или наоборот.

Вы можете подумать, что изменение необязательного параметра на обязательный – безопасно, но это не так. Присвоение функции нового типа изменяет информацию о ней, но не изменяет ее поведение, а изменение необязательного/обязательного статуса аргумента — это изменение поведения.

Аналогично записи и таблице, параметры нового типа функции сопоставляются с параметрами на основе позиции. Даже если движок M не проверяет, что имена параметров остаются неизменными, не используйте приписывание для переименования аргументов. Иначе может возникнуть непредвиденное поведение.

Поскольку приписывание типа может изменять только информацию о функции, а не поведение функции на уровне языка, приписывание типа функции, которая задает другие аргумента или является возвратной, не влияет на поведение функции. Движок M всегда использует утверждения типа, указанные при первоначальном определении функции, даже если позже ей приписывается другой тип.

Ниже определяется *someFunction* как ожидающая один аргумент, совместимый с типом *text*. Затем функции присваивается новый тип с типом аргумента *number*. Этот новый тип влияет на информацию о функции (например, в документации тип аргумента будет указан как число), но не изменяет её поведение. Функция была определена как ожидающая текст и по-прежнему ожидает текст, несмотря на то, что утверждает новый тип. Если передано нетекстовое значение, функция отклонит его как недопустимое.

Листинг 12

```
let
   SomeFunction = (input as text) => "hi",
   NewType = type function (input as number) as any,
```

```
    Ascribed = Value.ReplaceType(SomeFunction, NewType)
in
    Ascribed(1) // Expression.Error: Не удается преобразовать значение 1 в тип Text.
```

Синтаксис

Выше мы описали различные типы, определив типы их компонентов (типы элементов, типы строк, типы полей и типы аргументов/возвращаемых значений), используя имена литеральных типов. При указании типов компонентов также могут использоваться выражения. Это позволяет создавать сложные типы с использованием переменных и даже выходных данных функций.

Столбец в типе строки таблицы может определяться с использованием литерального типа *number* или *Currency.Type*:

```
type table [Amount = number]
```

```
type table [Amount = Currency.Type]
```

Но вместо ссылки на идентификатор (литерал имени типа) может быть вызвана функция. Также могут использоваться переменные:

Листинг 13

```
let
    ColumnType = type number,
    RowType = type [SomeColumn = ColumnType],
    TableType = type table RowType,
    ListType = type { TableType }
in
    type function (list as ListType) as any
```

Приведенное выше вычисление относится к типу функции с одним аргументом типа *list*, у которой элементы имеет тип *table [SomeColumn = number]*. Другими словами, этот тип описывает функцию, которая ожидает передачи списка таблиц (ноль или более таблиц), каждая из которых имеет один столбец с именем *SomeColumn*, заполненный значениями типа *number*.

Контекст типа

При создании значений типа ключевое слово *type* – это то, что придает именам литеральных типов их особое значение. *type* переключает интерпретацию синтаксиса M в так называемый *контекст типа*. Это просто означает, что ключевые слова типа приобретают свое особое значение. Например, само по себе выражение *any* ссылается на переменную с именем *any*; в то время как в сочетание *type any* переводит выражение в контекст типа, где *any* интерпретируется как имя типа.

Обратите внимание, как *type* переводит следующее выражение в контекст типа. *type* в начале строки приводит к тому, что, и *table*, и *any* интерпретируются как типы.

```
type table [Col1 = any]
```

В контексте типа идентификаторы, имена которых не совпадают с именами типов, по-прежнему интерпретируются как обычные ссылки на идентификаторы.

Листинг 14

```
let
    ColumnType = type any
in
    type table [Col1 = ColumnType]
```

В некоторых случаях, находясь в контексте типа, вы можете захотеть ссылаться на переменную с тем же именем, что и имя типа. Возьмем в качестве примера *type { record }*. Поскольку *type* помещает выражение в контекст типа, *record* интерпретируется как тип записи. Что, если вместо этого вы пытаетесь сослаться на переменную с именем *record*? Конечно вы можете просто переименовать переменную. Но также можете используйте круглые скобки, чтобы переключить часть выражения, которую они окружают, обратно в обычный контекст. Теперь *record* будет интерпретироваться как ссылка на переменную.

Листинг 15

```
let
   record = type [A = text, B = logical]
in
   type { (record) } // определяет список, тип элемента которого type [A = text, B = logical]
   // type { record } // определяет список, тип элемента которого = type record
```

Более сложный пример показывает переключение в контекст типа, затем обратно, затем снова в контекст типа:

```
type { (type number) }
```

Конечно, можно просто написать *type { number }*, зато теперь вы знаете, что возможно вложенное включение и выключение контекста типа.

Извлечение литерала типа из строки

Можно ли динамически описать тип, используя строку? Например, если у вас строка "number", как присвоить типу значение *number*? А если строка "text", то тип *text*? Можно ли это сделать на основе идеи переключения контекста? Похоже, ни одно из следующих действий не работает:

```
type (number)
```

```
type ("number")
```

Листинг 16

```
let
   TextNumber = "number",
   NewType = type (TextNumber)
in
   NewType
```

Переключение из контекста типа позволяет ссылаться на переменные, имена которых в противном случае интерпретировались бы как литералы типа; однако переключение не приводит к тому, что значение в строке обрабатывается как значение типа.

Чтобы извлечь тип из строковой переменной, можно использовать запись для преобразования из ожидаемых строковых значений в значения типа:

Листинг 17

```
let
   TextNumber = "number",
   Transform = (typeName as text) as type =>
   let
      Mappings = [
         number = type number,
         text = type text,
         logical = type logical
      ]
   in
      Record.Field(Mappings, typeName),
   NewType = Transform(TextNumber)
in
   NewType
```

Только для значений типа

Приведенное выше обсуждение синтаксиса о контексте типа и использовании выражений при составлении типов применимо только к выражениям, возвращающим значения типов, которые можно сохранить в переменной. При определении функций (не типов функций, а самих функций), а также с операторами *as* и *is* могут использоваться только литеральные обнуляемые примитивные типы. Ни ключевое слово *type,* ни круглые скобки, не будут переключать контекст. Никаких выражений. Никаких пользовательских типов.

```
1 as nullable number // допустимый синтаксис
```

1 as type nullable number // недопустимый синтаксис, число не должно иметь префикс *type*

SomeRecord is record // допустимый синтаксис

SomeRecord is [Name = text] // недопустимый синтаксис, нельзя применить пользовательский тип

(input as table) => ... // допустимый синтаксис

(input as [Col1 = any]) => ... // недопустимый синтаксис, нельзя применить пользовательский тип

(input as SomeType) => ... // недопустимый синтаксис, нельзя использовать выражение для типа

Последние два объявления функций синтаксически недопустимы, поскольку они пытаются определить функцию с утверждением аргумента сложного типа и выражением. Ни то, ни другое не допускается в определениях функций. Такие объявления разрешены в типах функций, которые могут быть приписаны существующей функции. Однако помните, что приписывание типов функциям влияет только на информацию, а не на поведение.

```
let
   Function = (input as table) => …,
   SomeType = type table [Col1 = any],
   NewType = type function (input as SomeType) as any
in
   Value.ReplaceType(Function, NewType)
```

В следующей заметке

Наверное вы озадачены, пытаясь понять назначение пользовательских типов и то, почему M не обрабатывает их так, как интуитивно ожидается. В следующей заметке мы постараемся внести ясность в эти вопросы. На данный момент сосредоточьтесь на изучении синтаксиса пользовательских типов; тогда в следующий раз мы сможем сосредоточиться на том, как они себя ведут, не отвлекаясь на их определение.