Язык M Power Query. Типы list, record (список и запись)

Возможно, вы склонны пропустить изучение списков и записей, и сразу перейти к таблицам. И действительно, в большинстве запросов вы в основном работаете с таблицами. А таблицы состоят из значений примитивных типов, о которых вы узнали в предыдущих заметках. Так зачем же вам list и record? Оказывается, таблица ведет себя так же, как список и запись. Если вы хотите использовать все возможности, которые предлагает таблица, изучите list и record. Кроме того, существует связь между выражением let и типом record. Так что, полезного вам чтения![1]

Предыдущая заметка Следующая заметка

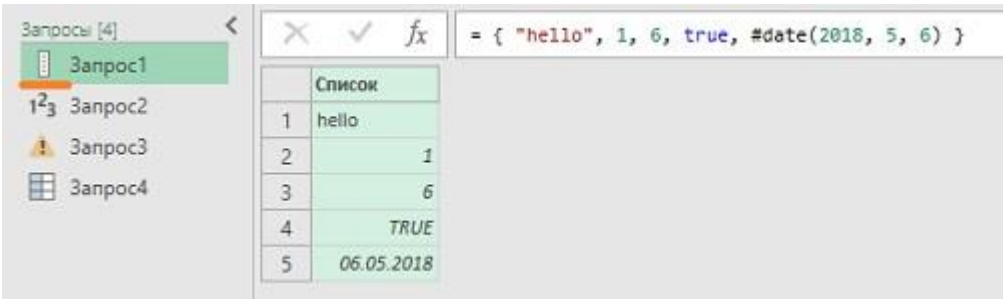

Рис. 1. Список в редакторе Power Query

list (список)

Тип *list* хранит список значений.

{ 1, 2, 5, 10 }

{ "hello", "hi", "good bye" }

Список может быть пустым:

{ }

Значения в списке не обязательно должны быть одного и того же типа.

Листинг 1[2]

{ "hello", 1, 6, true, #date(2018, 5, 6) }

Обратите внимание, на рис. 1 редактор Power Query распознал содержание *Запроса1*, и отразил пиктограмму списка перед именем запроса.

Можно создавать списки, которые ограничены только значениями определенного типа, но это продвинутый сценарий, который требует знания системы типов M, поэтому мы не будем его здесь рассматривать.

Поскольку список может содержать значения, а список сам по себе является значением, список может содержать списки, которые, в свою очередь, могут содержать списки и так далее.

```
{
   { "first name", "last name", "children" },
   { "Sue", "Smith",
      { "Sarah", "Sharon" }
   }
}
```

Если вас интересует увеличивающийся последовательный список целых чисел, две точки .. – это удобный ярлык, избавляющий вас от необходимости вводить весь список.

{ 1..5 } // эквивалентно { 1, 2, 3, 4, 5 }

{ -2..0 } // эквивалентно { -2, -1, 0 }

{1, 6..8, 15 } // эквивалентно {1 , 6, 7, 8, 15 }

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 10): Types—List, Record. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

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

Этот ярлык работает только для создания списков возрастающих значений. Попытка использовать его для создания последовательности, которая ведет обратный отсчет, вернет пустой список.

{ 5..1 } // { } выводит пустой список

Несколько списков могут быть объединены с помощью оператора конкатенации.

{ 1, 2 } & {3, 4 } // { 1, 2, 3, 4 }

{ 1, 2 } & { 2, 3} // { 1, 2, 2, 3 }

Два списка равны, если они содержат одни и те же значения в одном и том же порядке.

{ 2, 4 } = { 2, 4 } // true

{ 1, 2 } = { 1, 2, 3} // false

{ 4, 5 } = { 5, 4 } // false, те же значения, но в другом порядке

{ 2, 4 } <> { 2, 4 } // false

Сравнения больше (>) и меньше (<) в списках не поддерживаются.

Доступ к элементу списка

Доступ к элементу списка можно получить с помощью оператора позиционного индекса. Просто возьмите ссылку на интересующий список и добавьте индекс нужного элемента списка, заключенный в фигурные скобки: someList{someIndex}. В M индексы начинают отсчет от 0.

Если…

Values = { 10, 20, 30 }

… то

Values{0} // 10, значение первого элемента списка

Values{1} // 20, значение второго элемента списка

Values{2} // 30, значение третьего элемента списка

Попытка получить доступ к индексу, превышающему тот, который находится в списке, возвращает ошибку.

Values{3} // Expression.Error: Элементов в перечислении было недостаточно для выполнения операции.

Если вы предпочитаете получать значение *null* при использовании слишком большого индекса, добавьте ? к фигурным скобкам. Это называется: *выполнить необязательный выбор поля*.

Values{3}? // null

Нельзя извлечь более одного элемента списка:

= { 10, 20, 30 }{1,2} /* редактор PQ укажет на запятую внутри второй скобки,
и вернет ошибку Expression.SyntaxError: Ожидался токен RightBrace */

Ленивая оценка

Списки рассчитываются лениво.

Листинг 2

```
let
   Data = { 1, 2, error "help", 10, 20 }
in
   List.Count(Data) // 5
```

Приведенный код возвращает значение 5, даже несмотря на то, что один из элементов определен как выражение, вызывающее ошибку. Мы не запрашивали значение этого элемента, поэтому PQ его и не рассчитывал. Выражение, вызывающее ошибку, не вызывалось. Ошибка не возникала. Мы только спросили: «сколько элементов в списке?» Являются ли все они действительными или нет, – это другой вопрос, который мы не задавали, и на который движок M не пытался ответить.

Более того, если требуются некие значения списка, ленивая оценка означает, что движок рассчитает столько элементов списка, сколько необходимо для получения запрошенных выходных данных. Используя данные из приведенного выше примера, следующие выражения не вернут ошибок. Ни для того, ни для другого не требуется значение третьего элемента.

List.Sum(List.FirstN(Data, 2)) // возвращает 3, сумма двух первых элементов списка

List.Sum(List.LastN(Data, 2)) // возвращает 30, сумма двух последних элементов списка

Попытка запросить сумму первых трех элементов вернет ошибку.

Листинг 3

```
let
   Data = { 1, 2, error "help", 10, 20 }
in
   List.Sum(List.FirstN(Data, 3))
```

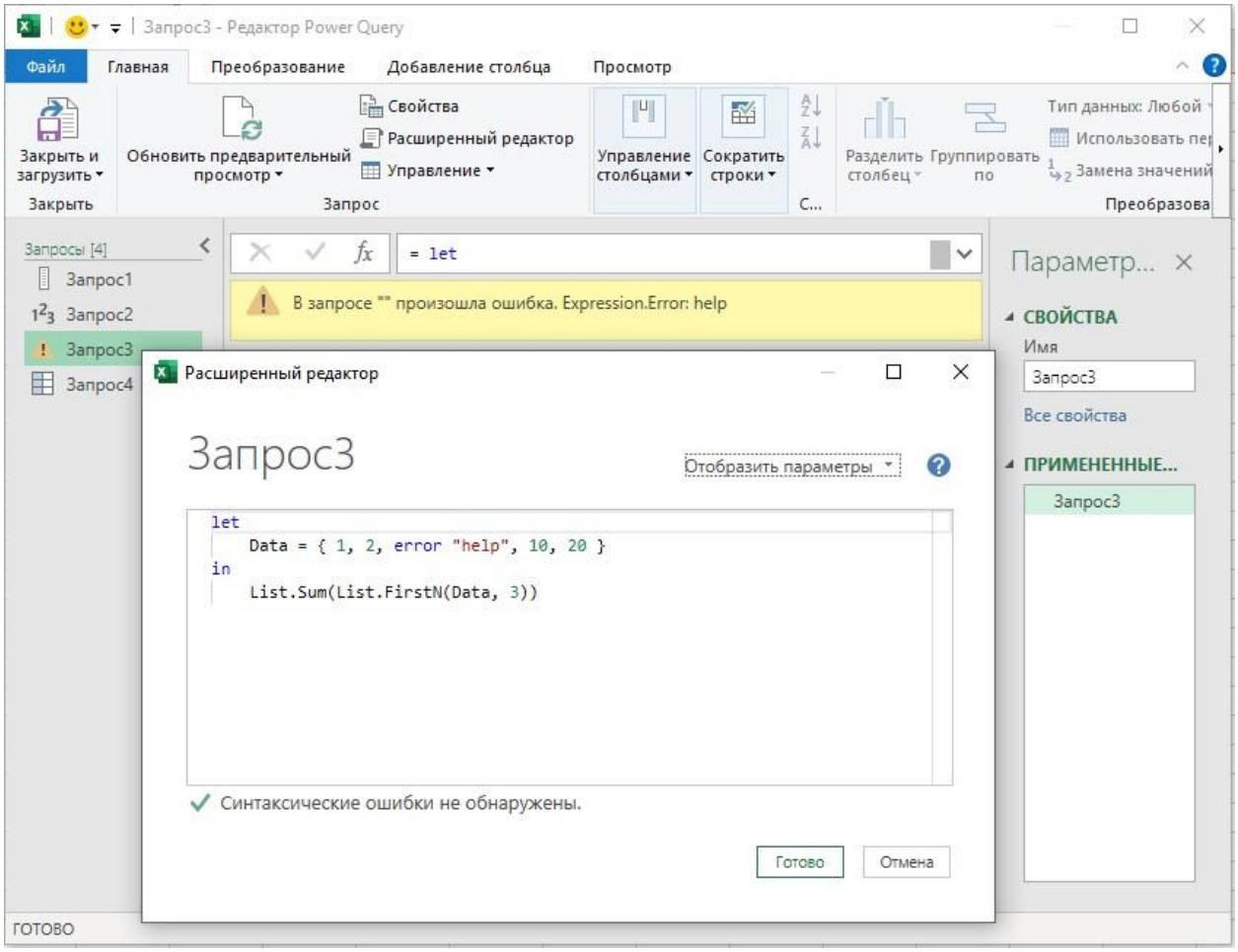

Рис. 2. Ленивая оценка M вернет ошибку только, если выражение требует рассчитать третий элемент списка

Библиотечные функции для работы со списками

Их много (я насчитал 75). Функции позволяют подсчитать число элементов, найти текст, агрегировать числовые элементы (найти сумму, произведение, …), преобразовать список (удалить или заменить элементы, расположить элементы в обратном порядке, …), вывести статистики (среднее, максимальное значение, стандартное отклонение, …), проверить принадлежность (содержит ли список некое значение, все ли значения в списке вернут *true* при обработке некой функцией), обработать несколько списков (найти объединение, пересечение, различие, ..), отфильтровать и отсортировать элементы списка, ... Существует семейство функций для генерации списков из элементов примитивных типов. Это удобно, если вам нужен список последовательных дат, чисел или список случайных чисел.

record (запись)

Запись позволяет сгруппировать набор именованных полей.

Листинг 4
= [FirstName = "Joe", LastName = "Smith", Birthdate = #date(2010, 1, 2)]

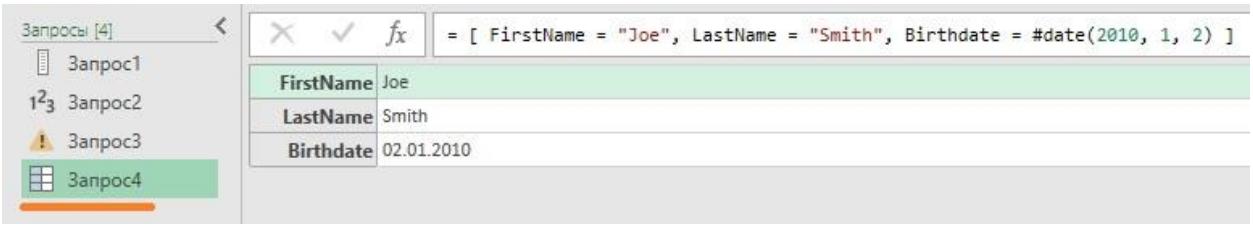

Рис. 3. Запись в редакторе Power Query

И здесь редактор PQ распознал запись и пометил пиктограммой *Запрос4*.

Технически запись сохраняет порядок своих полей. Однако, при сравнении записей порядок полей не учитывается. Поэтому порядок полей – элемент удобства пользовательского интерфейса. Например, поля будут выводиться на экран в том же порядке, в котором вы их определили, что облегчает визуальное определение интересующих вас элементов.

Пустая запись не содержит полей.

[ ]

Равенство определяется именем и значением поля. Позиция поля не учитывается.

= [a = 1, b = 2] = [a = 1, b = 2] // true

= [a = 1, b = 2] = [b = 2, a = 1] /* true, те же имена
и значения полей, даже если порядок отличается*/

= [a = 1] = [A = 1] // false, разные имена полей

= [a = 1] = [a = 2] // false, те же имена полей, но разные значения

= [a = 1] <> [A = 1] // true

Записи могут быть объединены с помощью оператора конкатенации.

= [a = 1] & [b = 2] // возвращает [a = 1, b = 2]

Если одно и то же имя поля присутствует в обоих входных данных слияния, используется значение, указанное последним. Фактически, это способ изменить/перезаписать значение поля.

= [a = 1] & [a = 10] // возвращает [a = 10]

Обращение к полю записи

Если списки используют {index} для доступа к элементам списка, то для записи указывается оператор просмотра (lookup) – имя поля в квадратных скобках: someList[someField]. Например, если запись Value определена так…

= Value = [Part = 1355, Description = "Widget", Price = 10.29, Internal Cost = 8.50]

… то выражения вернут…

= Value[Part] // 1355

= Value[Description] // "Widget"

= Value[Price] // 10.29

= Value[Internal Cost] // 8.50

Попытка извлечь содержимое несуществующего поля вернет ошибку:

= Value[NonExistentField]
//Expression.Error: Поле "NonExistentField" в записи не найдено

Здесь также можно добавить ? к оператору просмотра. Вместо ошибки вернется значение *null*.

= Value[NonExistentField]? // возвращает *null*

Внутри записи выражение для значения поля может ссылаться на другие поля.

```
[
 FirstName = "Sarah",
 LastName = "Smith",
 FullName = FirstName & " " & LastName
]
```

Выражение для значения поля даже может ссылаться на само себя (рекурсивно), для этого перед именем следует поставить оператор @, называемый оператором определения области действия (scoping operator). Такой функционал пригодится, когда вы захотите внутри поля использовать рекурсивную функцию.

Листинг 5

```
let
   Value =
   [
      AddOne = (x) => if x > 0 then 1 + @AddOne(x - 1) else 0,
      AddOneThreeTimes = AddOne(3)
   ]
in
   Value[AddOneThreeTimes] // возвращает 3
```

Здесь на шаге *Value* определяется запись из двух полей с именами [AddOne, AddOneThreeTimes]. Первое поле содержит рекурсивную функцию, второе поле – вызов этой функции с параметром 3.

Проекция

Квадратные скобки, используемые для выбора полей записи, также можно применить для извлечения *проекции* записи. Проекция содержит меньше полей, чем исходная запись.

Листинг 6

```
let
   Source = [ FieldA = 10, FieldB = 20, FieldC = 30 ]
in
   Source[[FieldA], [FieldB]]
```

Код возвращает усеченную запись, содержащую поля *FieldA* и *FieldB*:

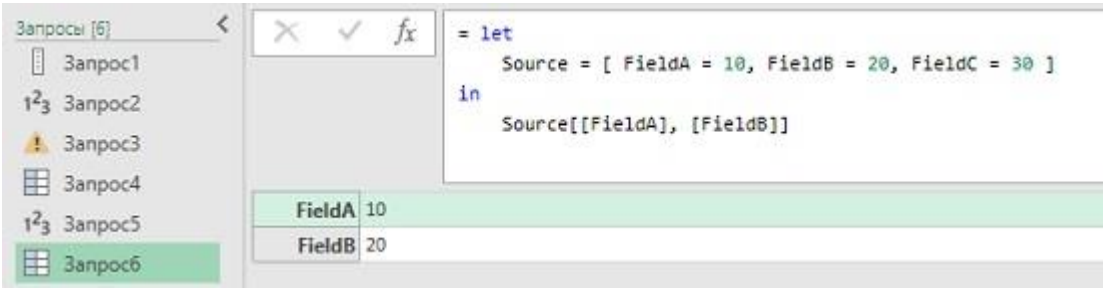

Рис. 4. Проекция записи

Обратите внимание, когда мы извлекаем значение поля записи, используем синтаксис:

Source = [FieldA = 10, FieldB = 20, FieldC = 30] [FieldA]

А когда формируем новый список, как проекцию исходного списка, скобки удваиваются:

Source = [FieldA = 10, FieldB = 20, FieldC = 30] [[FieldA]]

Попытка указать несуществующее поле при извлечении проекции, вернет ошибку:

Листинг 7

```
let
   Source = [ FieldA = 10, FieldB = 20, FieldC = 30 ]
in
   Source[[FieldA], [FieldD]] // Expression.Error: Поле "FieldD" записи не найдено.
```

Но, если поставить знак ? в конце выражения, несуществующее поле будут добавлено в проекцию с значением *null*.

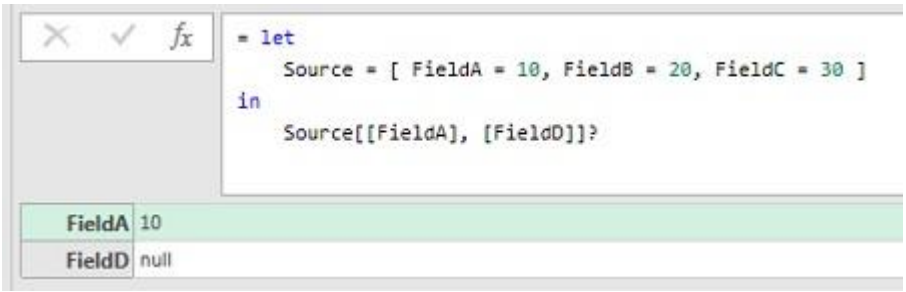

Рис. 5. Использование знака ? необязательного выбора поля

Синтаксис идентификаторов внутри квадратных скобок

В синтаксисе определения, просмотра и проекции записей в квадратных скобках имена полей имеют смягченные правила кавычек. Вне квадратных скобок в языке M допускаются обычные идентификаторы и идентификаторы в кавычках. Обычные идентификаторы могут начинаться с буквы или знака подчеркиваться, и не могут содержать пробелы, специальные символы и ключевые слова M. Идентификаторы в кавычках начинаются со знака #, а далее идет идентификатор в кавычках с двух сторон. Такие идентификаторы могу содержать в своем имени любые символы и ключевые слова M.

Квадратные скобки сами по себе определяют начало и окончание идентификатора, поэтому кавычки в большинстве случаев не требуются. Например, за пределами квадратных скобок идентификатор *Street Address* нуждается в кавычках (и #), потому что он содержит пробел, а *try* – потому что это ключевое слово. В квадратных скобках эти идентификаторы не требуют кавычек (и #):

[#"try" = true, #"Street Address" = "123 Main St."]

[try = true, Street Address = "123 Main St."] // идентичен предыдущему

SomeRecord[#"Street Address"]

SomeRecord[Street Address]

SomeRecord[#"try"]

SomeRecord[try]

Однако, пробелы в начале и конце имени поля без кавычек будут проигнорированы, и поэтому они исключаются из имени поля. Если же такие пробелы, например, присутствуют в имени столбцов, импортированных из внешнего источника, вам нужно заключить имя в кавычки:

SomeRecord[Street Address] // M будет считать, что имя "Street Address"

SomeRecord[#" Street Address "] // M будет работать с именем " Street Address "

Подобное замечание относится и к специальным символам. Они всё еще не допускаются в именах внутри квадратных скобок без кавычек:

SomeRecord[Country/City] // Expression.SyntaxError: Недопустимый идентификатор.

SomeRecord[#"Country/City"] // Ok

Ленивая оценка и фиксация значения

Как и для списков, в записях правит ленивая оценка. Если значение не требуется, оно не вычисляется.

[Price = 10, Quantity = error "help"][Price] // возвращает 10

Поле Quantity не было затребовано, поэтому его значение не вычислялось, и ошибка не возникла.

Когда поле вычисляется в первый раз, результирующее значение или ошибка фиксируются как значение для этого поля. Расчет для каждого поля выполняется только один раз. Затем его

значение кэшируется. Кэшированное значение или ошибка возвращаются при каждом последующем обращении к полю.

[Price = GetValueFromRemoteServer()]

Представьте, что при первом обращении к *Price* удаленный сервер возвращает 10. Позже в коде запроса есть повторное обращение к *Price.* Возможно, к этому моменту вызов GetValueFromRemoteServer() вернул бы значение 11. Однако этот вызов не выполняется повторно. Вместо этого возвращается значение, кэшированное при первом обращении к полю – 10.

Если же при первом обращении к *Price* функция GetValueFromRemoteServer() вернула ошибку из-за временного сбоя связи, эта же ошибка будет повторно возникать при каждом последующем обращении к *Price*, даже если к моменту последующего обращения ошибка устранена и функция GetValueFromRemoteServer() могла бы вернуть что-то осмысленное.

Такая фиксация значений (или кэширование) обеспечивает согласованность. Благодаря этому вы знаете, что значение поля всегда будет одинаковым на протяжении выполнения запроса.

Кэширование значений не используется для разных записей, даже если внутри записи есть идентичные поля и выражения значений для этих полей. Если ваш код приводит к тому, что запись [Price = GetValueFromRemoteServer()] генерируется дважды, и доступ к *Price* осуществляется в обоих экземплярах, каждый из них будет отдельно вызывать функцию GetValueFromRemoteServer(). Если возвращаемое значение изменилось между двумя вызовами, две записи будут иметь разные значения для *Price*.

Если запись, с которой вы работаете, назначена переменной, каждый раз, когда вы обращаетесь к этой переменной, вы будете обращаться к одному и тому же экземпляру записи. Однако, если, обращаясь к записи несколько раз, вы вызываете выражение, которое извлекает данные из внешнего источника (например, базы данных или веб-службы), каждое извлечение может возвращать другой экземпляр записи. Если вам важно работать с одним и тем же экземпляром записи, извлеките его один раз, а затем сохраните в переменной или, в случае списка записей, буферизуйте список.

Библиотечные функции для работы с записями

В стандартной библиотеке таких функций меньше, чем для работы со списками. Я насчитал «всего» 23. В том числе функции для добавления, переименования, изменения порядка и удаления полей, а также преобразования значений полей. Существует также метод, возвращающий список имен полей записи (с сохранением порядка полей), и аналогичный метод, возвращающий значения полей.

Динамические операции

Выше мы использовали оператор просмотра для доступа к значению поля по имени. Если вы хотите использовать программную логику для выбора поля записи, следующий подход не работает.

Листинг 10

```
let
  Item = [Name = "Widget", Wholesale Price = 5, Retail Price = 10],
  PriceToUse = "Wholesale Price"
in
  Item[PriceToUse] // Expression.Error: Поле "PriceToUse" записи не найдено.
```

Это связано с тем, что имена полей в квадратных скобках должны быть строками; ссылки на переменные не допускаются.

Обойти эту проблему позволит функция Record.Field. Это динамический эквивалент оператора просмотра. Еще одна функция Record.FieldOrDefault – аналог оператора просмотра, за которым следует знак вопроса. В дополнительном аргументе можно указать значение, возвращаемое если имя поля не существует.

Листинг 11

```
let
  Item = [Name = "Widget", Wholesale Price = 5, Retail Price = 10],
```

```
  PriceToUse = "Wholesale Price"
in
  Record.Field(Item, PriceToUse) // возвращает 5
```

Листинг 12

```
= let
  Item = [Name = "Widget", Wholesale Price = 5, Retail Price = 10],
  PriceToUse = "My Price"
in
  Record.FieldOrDefault(Item, PriceToUse, 0)
// возвращает 0, так как поле с именем My Price не найдено
```

Чтобы извлекать проекцию по программной (динамической) ссылке, используйте Record.SelectFields.

Библиотечная функция также есть для удаления полей Record.RemoveFields (вместо проекции, где перечисляются оставляемые поля, укажите нежелательные поля, и будет возвращена новая запись, содержащая все остальные поля) и для изменения порядка полей Record.ReorderFields (удобно в тех редких случаях, когда порядок полей имеет значение).

let сахар[3]

Готовы к сюрпризу? Выражение *let* – это, по сути, синтаксический сахар для записи. Следующее выражение…

```
let
  A = 1,
  B = 2,
  Result = A + B
in
  Result
```

…эквивалентно…

```
[
  A = 1,
  B = 2,
  Result = A + B
][Result]
```

Из этого следует: то, что мы знаем о записях, также применимо к выражениям *let* и наоборот. Например, мы знаем, что значение поля записи вычисляется при первом обращении, а затем кэшируется. Поскольку *let* по сути является выражением записи, к нему применяется то же самое правило неизменяемости: выражение переменной *let* будет вычислено при первом обращении, а затем его значение будет кэшировано. С другой стороны для выражений *let* мы знаем, что есть одно исключение из неизменяемости, которое возникает, когда в игру вступает потоковая передача. Это же исключение должно применяться к записям… поскольку *let* и записи ведут себя одинаково.

В следующей заметке

Вы заметили, что *запись* отлично подходит для хранения строки данных!? Если вы хотите сохранить несколько записей, каждая из которых представляет строку данных, в одной переменной, вы можете поместить эти записи в *список*. Хм... похоже, мы приближаемся к понятию *таблицы*!

В следующий раз мы обсудим *таблицы.* Мы узнаем, как *таблицы* наследуют поведение *списков* и *записей*. В то же время мы увидим, что *таблицы* предлагают гораздо больше возможностей, чем *список записей*.

---

[3] Синтаксическим сахаром в языках программирования называются элементы, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.