**Джеффри Фридл. Регулярные выражения в Excel**

Когда я работал в издательстве, то очень активно пользовался обработкой текста с помощью шаблонов. Тогда я использовал программу PageMaker (ныне InDesign) и язык скриптов. Позже я перенес этот опыт на обработку текста в Word и Excel, но использовал в макросах возможности, предоставляемые самими программами. Несколько лет назад я открыл для себя язык регулярных выражений. Прочитал книгу Бена Форта [Регулярные выражения за 10 минут](http://baguzin.ru/wp/?p=17394). К сожалению, доступ к регулярным выражениям открывался только через код VBA или специальные программы, что затрудняло понимание прочитанного и дальнейшее практическое использование регэкспов.

И вот совсем недавно я наткнулся на [заметку](https://www.planetaexcel.ru/techniques/7/4844/) Николая Павлова, в которой предлагается пользовательская функция RegExpExtract, переносящая всю работу с регулярными выражениями на листы Excel. В заметке также есть ссылка на два ресурса для проверки регулярных выражений в режиме онлайн: <https://regex101.com/>, <https://regexr.com/>. Рекомендую! Ваши шаблоны будут разобраны на элементы и показана их работа.

Поняв, как упростить работу с регулярными выражениями в Excel, я решил сделать второй подход к теме. Для чего взял фундаментальный труд Джеффри Фридла и изложил его фрагменты на примерах в Excel.

Джеффри Фридл. Регулярные выражения, 3-е издание. – СПб.: Символ-Плюс, 2008. – 608 с.

Изображение выглядит как текст, хищная птица, сова, птица

Автоматически созданное описание

Купить цифровую книгу в [ЛитРес](https://www.ozon.ru/product/regulyarnye-vyrazheniya-141734369/?lfrom=13042861)

### Для начала настроим функцию RegExpExtract

Посмотрите, есть ли на вашей ленте Excel вкладка *Разработчик.* Если нет, пройдите по меню *Файл* –> *Параметры*. Перейдите на вкладку *Настроить ленту*.

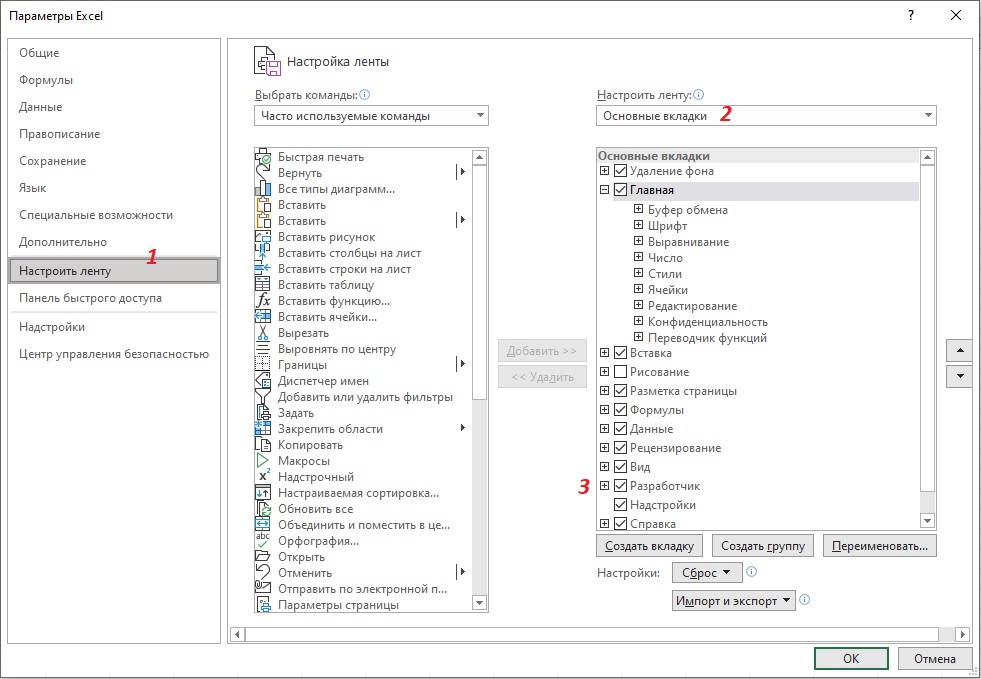


Рис. 1. Добавление вкладки *Разработчик* на ленту Excel

В окне *Настройка ленты* в области *Основные вкладки* установите флажок *Разработчик.* Нажмите Ok. В окне Excel перейдите на вкладку *Разработчик*. Кликните на кнопку *Visual Basic*. В окне редактора VBA кликните на строку *VBAProject [название вашего файла]*. Вставьте новый модуль, пройдя по меню *Insert* –> *Module* (рис. 2). Скопируйте код во вновь созданный модуль (я дополнил функцию Николая Павлова еще одним параметром – учетом регистра):

Public Function RegExpExtract( \_

Text As String, \_

Pattern As String, \_

Optional Item As Integer = 1, \_

Optional IgnoreCase As Boolean = 0 \_

) As String

On Error GoTo ErrHandl

Set regex = CreateObject("VBScript.RegExp")

regex.Pattern = Pattern

regex.Global = True

regex.IgnoreCase = IgnoreCase

If regex.Test(Text) Then

Set matches = regex.Execute(Text)

RegExpExtract = matches.Item(Item - 1)

Exit Function

End If

ErrHandl:

RegExpExtract = CVErr(xlErrValue)

End Function

Изображение выглядит как текст

Автоматически созданное описание

Рис. 2. Код функции RegExpExtract

Поскольку вы поместили код в модуль, относящийся к конкретной книге Excel, он не будет работать в иной книге. Сохраните файл Excel в формате с поддержкой макросов \*.xslm. Функция RegExpExtract имеет четыре параметра:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 3. Синтаксис функции RegExpExtract

Следует заметить, что регулярные выражения в VBA могут инициироваться [двумя способами](https://vremya-ne-zhdet.ru/vba-excel/regulyarnyye-vyrazheniya/): ранним и поздним связыванием. Выше в коде функции RegExpExtract продемонстрировано позднее связывание. Оно заключается в присвоении нового экземпляра объекта RegExp переменной, объявленной с помощью функции CreateObject:

Set regex = CreateObject("VBScript.RegExp")

Открывая файл на другом ПК, где библиотека VBScript явным образом не инициирована, пользовательская функция RegExpExtract всё равно будет работать.

Однако чаще рекомендуется использовать объекты с ранней привязкой, так как у них выше быстродействие, а также при написании и редактировании кода доступны подсказки свойств и методов. Для осуществления ранней привязки необходимо подключить к проекту VBA ссылку на новую библиотеку. В редакторе VBA пройдите по меню Tools –> References, и в окне *References - VBAProject* поставьте флажок в строке *Microsoft VBScript Regular Expressions 5.5*.

Изображение выглядит как текст

Автоматически созданное описание

Рис. 3а. Раннее связывание через подключение библиотеки регулярных выражений

После этого объект RegExp может быть создан, например, так:

Dim myRegExp As RegExp

Set myRegExp = New RegExp

Далее в этой заметке будет использоваться только код с поздней привязкой.

### Глава 1. Знакомство с регулярными выражениями

*Регулярные выражения* – это шаблоны и язык их обработки. Регулярные выражения способны вставлять, удалять, выделять и выполнять самые невероятные операции с текстовыми данными любого вида.

Регулярное выражение состоит из двух типов символов. Специальные символы (вроде \* в файловых шаблонах) называются *метасимволами*. Все остальные символы, т.е. обычный текст, называются *литералами*.

Простейшими метасимволами являются ^ (крышка, циркумфлекс) и $ (доллар), представляющие соответственно начало и конец проверяемой строки. Для выражения *^cat* совпадение происходит лишь в том случае, если символы cat находятся в начале строки.

#### Символьные классы

Допустим, необходимо найти строку *grey*, которая также может быть записана в виде *gray*. При помощи конструкции […], называемой символьным классом, можно перечислить символы, которые могут находиться в данной позиции текста. Таким образом, выражение *gr[ea]y* означает: «Найти символ g, за которым следует r, за которым следует e **или** a, и все это завершается символом y».

В контексте символьного класса метасимвол - (дефис) обозначает интервал символов. Классы [0-9] и [a-z] обычно используются соответственно для поиска цифр и символов нижнего регистра. Символьный класс может содержать несколько интервалов, поэтому класс [0123456789abcdefABCDEF] записывается в виде [0-9a-fA-F] при этом порядок перечисления роли не играет. Такое выражение пригодится при обработке шестнадцатеричных чисел. Интервалы также можно объединять с литералами: выражение [0-9A-Z\_!.?] совпадает с цифрой, символом верхнего регистра, символом подчеркивания, восклицательным знаком, точкой или вопросительным знаком.

Обратите внимание: дефис выполняет функции метасимвола только внутри символьного класса – в остальных случаях он совпадает с обычным дефисом. Более того, даже в символьных классах дефис не всегда интерпретируется как метасимвол. Если дефис является первым символом, указанным в классе, он заведомо не может определять интервал и поэтому интерпретируется как литерал. Аналогично, вопросительный знак и точка считаются метасимволами в контексте обычных регулярных выражений, но не в контексте класса.

Если вместо […] используется запись [^…], класс совпадает с любыми символами, **не входящими** в приведенный список. Символ ^ является метасимволом символьного класса лишь в том случае, если следует сразу же после открывающей скобки (в противном случае он интерпретируется как обычный символ).

#### Один произвольный символ

Метасимвол . (точка) представляет собой сокращенную форму записи для символьного класса, совпадающего с любым символом. Он применяется в тех случаях, когда в некоторых позициях регулярного выражения могут находиться произвольные символы. Допустим, надо найти дату, которая может быть записана в формате 19/03/76, 19-03-76 или даже 19.03.76. Конечно, можно сконструировать регулярное выражение, в котором между числами указываются все допустимые символы-разделители, например 19[-./]03[-./]76. Однако возможен и другой вариант – просто ввести выражение 19.03.76.

В приведенном примере имеется ряд неочевидных аспектов. В выражении 19[-./]03[-./]76 точки не являются метасимволами, поскольку они находятся внутри символьного класса. Дефисы в данном случае тоже интерпретируются как литералы, поскольку они следуют сразу же после [ и [^ Если бы дефисы не стояли на первых местах (например, [.-/]), они интерпретировались бы как интервальные метасимволы, что в данном случае привело бы к ошибке. В выражении 19.03.76 точки являются метасимволами, совпадающими с любым символом.

Очень удобный метасимвол | означает «или». Он позволяет объединить несколько регулярных выражений в одно, совпадающее с любым из выражений-компонентов. Подвыражения, объединенные этим способом, называются альтернативами.

Вернемся к примеру gr[ea]y. Выражение также можно записать в виде grey|gray и даже gr(a|e)y. Круглые скобки отделяют конструкцию выбора от остального выражения (и тоже являются метасимволами). Конструкция вида gr[a|e]y не подойдет – в символьном классе символ | является обычным символом.

Конструкция выбора действует только внутри круглых скобок. Будьте внимательны и не путайте конструкцию выбора с символьными классами. Символьный класс представляет один символ целевого текста. В конструкциях выбора каждая альтернатива может являться полноценным регулярным выражением, совпадающим с произвольным количеством символов. Символьные классы обладают собственным мини-языком (и, в частности, собственными представлениями о метасимволах), тогда как конструкция выбора является частью «основного» языка регулярных выражений.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 4. Сводка упоминавшихся метасимволов

#### Необязательные элементы

Рассмотрим пример поиска слова color или colour. Поскольку эти два слова отличаются только символом u, для их поиска можно использовать выражение colou?r. Метасимвол ? (знак вопроса) означает, что предшествующий ему символ является необязательным.

Рассмотрим поиск даты 4 июля (July), где название месяца July может быть записано как July или Jul, а число может быть записано как fourth, 4th или просто 4. Мы могли бы использовать выражение (July|Jul)•(fourth|4th|4). Здесь символ • используется для обозначения пробела в тексте книги; в коде используется обычный пробел.

Часть (July|Jul) можно сократить до July? Ликвидация метасимвола | позволяет убрать круглые скобки, ставшие ненужными. Наличие круглых скобок не повлияет на конечный результат, однако выражение July? без скобок воспринимается проще. Выражение 4th|4 можно сократить до 4(th)? Знак ? может присоединяться и к выражениям в круглых скобках. Подвыражение внутри скобок может быть сколь угодно сложным, но «снаружи» оно воспринимается как единое целое.

После упрощений наше выражение принимает вид July?•(fourth|4(th)?)

#### Квантификаторы повторения

Метасимвол + обозначает *один или несколько экземпляров непосредственно предшествующего элемента*, a \* – *любое количество экземпляров элемента (в том числе и нулевое)*. Иначе говоря, \* означает «найти столько экземпляров, сколько это возможно, но при необходимости обойтись и без них». Конструкция + имеет похожий смысл (она также пытается найти как можно большее число экземпляров указанного элемента), но при отсутствии хотя бы одного экземпляра сопоставление завершается неудачей. Три метасимвола, которые могут совпадать с переменным количеством экземпляров элемента, ?, + и \*, называются квантификаторами, поскольку они определяют количество элементов, на которые воздействуют.

Совпадение для конструкций \* и ?, существует всегда. Вопрос лишь в том, какой текст будет (и будет ли вообще) содержаться в совпадении. Конструкция + требует наличия хотя бы одного экземпляра искомого текста. Например, выражение •? допускает не более одного необязательного пробела, тогда как •\* – произвольное число необязательных пробелов.

Например, шаблон <H[1-6]> позволяет искать заголовков HTML; <Н1>, <Н2>, <Н3> и т.д. Однако в спецификации HTML говорится, что непосредственно перед закрывающей угловой скобкой допускаются пробелы – например, <H3•> или <H4•••>. Вставляя •\* в ту позицию регулярного выражения, где могут находиться (а могут и отсутствовать) пробелы, мы получаем <H[1-6]•\*>. Выражение по-прежнему совпадает с <H1>, поскольку наличие пробелов необязательно, но при этом также подходит и для других вариантов.

Попробуем организовать поиск тегов HTML вида <HК•SIZE=14>, этот тег означает, что на экране рисуется горизонтальная линия толщиной 14 пикселов. Перед закрывающей угловой скобкой могут стоять необязательные пробелы. Кроме того, пробелы могут находиться и по обе стороны знака =. Наконец, минимум один пробел должен разделять HR и SIZE, хотя их может быть и больше. В последнем случае мы воспользуемся •+. Плюс разрешает дополнительные пробелы, но требует обязательного присутствия хотя бы одного пробела. В итоге мы приходим к выражению <HR•+SIZE•\*=•\*14•\*>.

Если вместо поиска тегов с одной конкретной толщиной линии (14) мы хотим найти все варианты, заменим 14 на любое число цифр. Цифра определяется выражением [0-9], а чтобы отыскать «одну или несколько цифр» достаточно добавить символ +. Символьный класс является отдельным элементом, поэтому применение к нему метасимволов + или ? не требует круглых скобок. В итоге получаем <HR•+SIZE•\*=•\*[0-9]+•\*>

К счастью VBA может игнорировать регистр символов, поэтому нам не пришлось использовать [Hh][Rr] и т.д.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 5. Сводка квантификаторов

В VBA также поддерживается метапоследовательность для явного определения минимального и максимального количества совпадений: {min, max}. Эта конструкция называется интервальным квантификатором. Например, выражение [a-zA-Z]{1,5} может использоваться для поиска обозначений биржевых котировок – от одной до пяти букв. В следующем примере шаблон в Е2 позволяет найти три разных вхождения последовательностей букв t, а шаблон в Е8 – только два.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 6. Работа интервального квантификатора: а) t{2,8}, б) t{3,8}

#### Обратные ссылки

До настоящего момента мы встречались с двумя применениями круглых скобок: ограничение области действия в конструкции выбора | и группировка символов для применения квантификаторов ?, \* и +. Еще одно применение круглых скобок – «запоминание» текста внутри скобок. В дальнейшем можно ссылаться на этот фрагмент. Например, при поиске повторяющихся слов. Т.е., слов, значения которых не известны до начала поиска: \b([a-z]+)•+\1\b.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 7. Обратная ссылка \1 позволяет найти любые повторяющиеся слова

\b соответствует границе слова, то есть позиции между словом и пробелом; [a-z]+ любое количество любых букв (чувствительность к регистру отключаем на уровне функции RegExpExtract; •+ любое число пробелов; \1 повтор выражения, найденного в круглых скобках; \b этот повтор должен быть целым словом (это не даст регулярному выражению найти соответствие, скажем для the•theory).

В выражение можно включить несколько пар круглых скобок и ссылаться на совпавший текст при помощи метасимволов \1, \2, \3 и т.д. Пары скобок нумеруются в соответствии с порядковым номером открывающей скобки слева направо, поэтому в выражении ([а-z])([0-9])\1\2 метасимвол \1 ссылается на текст, совпавший с [a-z], a \2 ссылается на текст, совпавший с [0-9].

#### Экранирование

Для включения в регулярное выражение символа, который обычно интерпретируется как метасимвол, перед ним нужно поставить обратный слеш \. Например, для поиска слова в круглых скобках (very) можно воспользоваться регулярным выражением \([a-zA-Z]+\). Символ \ в последовательностях \( и \) отменяет особую интерпретацию символов () и превращает их в литералы, совпадающие с круглыми скобками в тексте. Экранирование работает и в символьных классах:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 8. Экранирование в символьных классах: сравните шаблоны в Е2 и Е7

При построении регулярного выражения приходится постоянно следить за тем, чтобы регулярное выражение: совпадало там, где нужно, и не совпадало там, где не нужно.

При работе с регулярными выражениями, как и с любым языком, чрезвычайно полезен практический опыт, поэтому ниже приведено еще несколько распространенных примеров регулярных выражений.

#### Имена переменных

Такие имена состоят из алфавитно-цифровых символов и знаков подчеркивания, но не могут начинаться с цифры. На языке регулярных выражений эта формулировка записывается в виде [a-zA-Z\_][a-zA-Z\_0-9]\*. Первый класс определяет возможные значения первого символа имени, второй (вместе с суффиксом \*) определяет оставшуюся часть идентификатора. Если длина идентификатора ограничивается, допустим, 32 символами, звездочку можно заменить выражением {0, 31}.

#### Последовательности символов, заключенные в кавычки

Простое выражение, обозначающее строку в кавычках: "[^"]\*". Двойные кавычки, ограничивающие регулярное выражение, совпадают с открывающими и закрывающими кавычками строки. Между ними может находиться все, что угодно... кроме других кавычек! Выражение [^"] совпадает с любым символом, кроме ", а звездочка говорит о том, что количество таких символов может быть любым.

#### Денежные суммы в долларах (с необязательным указанием центов)

Одно из возможных решений: \$[0-9]+(\.[0-9][0-9])? Это регулярное выражение разбивается на три части: \$, …+ и (…)?. Первая – литерал-знак доллара, вторая – неограниченная последовательность цифр, третья – десятичная точка и две цифры. Третья часть является необязательной.

Этот пример наивен по нескольким причинам. Например, это регулярное выражение предполагает, что денежная сумма в долларах будет записываться как $1000, но не как $1,000. Кроме того, приведенное выражение не находит суммы вида $.49. Возникает искушение заменить + на \*, но такое решение не годится. Мы вернемся к этому примеру в главе 5.

#### URL адреса

Адреса могут иметь довольно сложную структуру, поэтому построение регулярного выражения для любых возможных URL довольно сложная задача. Тем не менее небольшое смягчение требований позволяет описать большинство URL. Обычные адреса URL имеют следующую структуру: http://хост/путь.html, хотя также часто встречается расширение htm. Для хоста можно предложить шаблон [-a-z0-9\_.]+. Структура пути может быть еще более разнообразной, поэтому мы обозначим путь выражением [-a-z0-9\_:@&?=+,.!/~+%$]\*. Обратите внимание: перечисление символов в этих классах начинается с дефиса, чтобы он включался в список как литерал, а не интерпретировался как часть интервала. В принципе можно обойтись еще более простым выражением: \<http://[^ •]\*\.html?\>

По мере углубления в изучение темы вы убедитесь, что правильная оценка данных в значительной мере определяет компромисс между сложностью и точностью. Мы вернемся к примеру с поиском URL в следующей главе.

#### Время

Поиск времени тоже может осуществляться с разной степенью точности. Например, выражение [0-9]?[0-9]:[0-9][0-9]•(am|pm) успешно находит 9:17•am и 12:30•pm, но с такой же легкостью обнаруживает время 99:99•pm.

Если час состоит из двух цифр, то первая цифра может быть только единицей. Для поиска часов подойдет (1[012]|[1-9]). Для минут первая цифра определяется выражением [0-5], а для второй цифры можно оставить [0-9]. Объединяя все компоненты, мы получаем (1[012]|[1-9]):[0-5][0-9] •(am|pm).

Попробуйте построить регулярное выражение для поиска времени в 24-часовом формате, с нумерацией часов от 0 до 23. Чтобы задание было посложнее, разрешите использование начального нуля до 09:59. Решение несколькими абзацами ниже.

### Глава 2. Дополнительные примеры

Ранее мы увидели, что круглые скобки могут применяться для «запоминания» текста внутри скобок. Далее в шаблоне можно обратиться к запомненному фрагменту с помощью метасимволов \1, \2, … Если же на фрагмент нужно сослаться за пределами шаблона, используются иные механизмы.

#### Ссылки $1, $2, …

Пусть у нас есть текст, из которого нужно убрать дефис:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 9. Функция RegExpRemoveHyphen, удаляющая дефис из текста

Для обнаружения конструкции используем шаблон ([^-]\*)-(.\*). Здесь ([^-]\*) соответствует любому количеству символов за исключением дефиса. Далее в шаблоне следует дефис, и наконец (.\*) соответствует любому количеству любых символов. (Чтобы продемонстрировать принцип я не стал усложнять шаблон, так что наш удаляет только первый дефис.) Код функции:

Public Function RegExpRemoveHyphen(Text As String, Pattern As String) As String

On Error GoTo ErrHandl

Set regEx = CreateObject("VBScript.RegExp")

regEx.Pattern = Pattern

RegExpRemoveHyphen = regEx.Replace(Text, "$1$2")

Exit Function

ErrHandl:

RegExpRemoveHyphen = CVErr(xlErrValue)

End Function

В строке RegExpRemoveHyphen = regEx.Replace(Text, "$1$2") исходный текст заменяется на два фрагмента, найденных в соответствии с шаблоном. На эти фрагменты ссылаются по порядковому номеру появления круглых скобок. Сначала берется часть ([^-]\*). Ссылка на нее – $1. Затем часть (.\*). Ссылка на нее – $2. Таким образом, просто исключается дефис.

Ссылки в стиле $1, $2 могут использоваться в методе Replace (возможно, где-то еще), но в общем случае используется иной механизм.

#### Ответ на вопрос: поиск времени в 24-часовом формате

Задача разбивается на три временных интервала: утро (с 00 до 09 часов, возможен начальный ноль), день (с 10 до 19 часов) и вечер (с 20 до 23 часов). Прямолинейное решение выглядит так: (0?[1-9]|1[0-9]|2[0-3]):[0-5][0-9]. Но первые два варианта можно объединить: [01]?[0-9]|2[0-3]:[0-5][0-9].

#### Метод SubMatches

Пусть мы хотим найти в текстовой строке температуру в градусах Цельсия или Фаренгейта, и вернуть значения в обеих шкалах. Опять же ради демонстрации метода не будем усложнять шаблон:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 10. Функция RegExpTemp, возвращающая температуру в градусах Цельсия и Фаренгейта

Public Function RegExpTemp(Text As String, Pattern As String) As String

Dim m As Object

On Error GoTo ErrHandl

Set regEx = CreateObject("VBScript.RegExp")

regEx.Pattern = Pattern

Set Matches = regEx.Execute(Text)

For Each m In Matches

If m.SubMatches(1) = "C" Then

RegExpTemp = m.SubMatches(0) & "C = " & Format(m.SubMatches(0) \* 9 / 5 + 32, "#.0") & "F"

Else

RegExpTemp = Format((m.SubMatches(0) - 32) \* 5 / 9, "#.0") & "C = " & m.SubMatches(0) & "F"

End If

Next

Exit Function

ErrHandl:

RegExpTemp = CStr(CVErr(xlErrValue))

End Function

Извлеченная по шаблону подстрока, помещается в коллекцию Matches = regEx.Execute(Text). Нумерация элементов в коллекциях Matches и SubMatches начинается с нуля. Далее проверяется, какую букву возвращает вторая пара круглых скобок (C|F): m.SubMatches(1). Если "C", то помещаем значение из первой группы (\d+) в начало функции RegExpTemp = m.SubMatches(0), а значение для градусов Фаренгейта вычисляем по формуле. Чтобы вычисленные значения смотрелись красиво с помощью функции Format ограничиваем число символов после запятой одним. Если значение m.SubMatches(1) = "F", то вычисляем градусы по шкале Цельсия.

#### Объектная модель VBA

Эксперт с форума Планет Excel [Kuzmich](https://www.planetaexcel.ru/forum/index.php?PAGE_NAME=profile_view&UID=20) предложил более изящный код для функции RegExpTemp.

Function iDegree(cell$)

With CreateObject("VBScript.RegExp")

.Pattern = "(\d+)(C|F)"

If .test(cell) Then

If .Execute(cell)(0).SubMatches(1) = "C" Then

iDegree = .Execute(cell)(0).SubMatches(0) & "C = " & \_

Format(.Execute(cell)(0).SubMatches(0) \* 9 / 5 + 32, "#.0") & "F"

Else

iDegree = Format((.Execute(cell)(0).SubMatches(0) - 32) \* 5 / 9, "#.0") & "C = " & \_

.Execute(cell)(0).SubMatches(0) & "F"

End If

Else

iDegree = "Температура не найдена"

End If

End With

End Function

Изображение выглядит как стол

Автоматически созданное описание

Рис. 11. Альтернативный вариант функции, возвращающей температуру в двух шкалах

Этот код основан на хорошем понимании иерархии объекта RegExp (рис. 12). Разберем, что происходит в строке…

If .Execute(cell)(0).SubMatches(1) = "C" Then

Метод Execute(cell) возвращает коллекцию Matches – все совпадения, найденные по шаблону Pattern в строке cell. Из коллекции берется первый элемент (0) и его свойство – SubMatches. При этом выбирается вторая подстрока SubMatches(1).

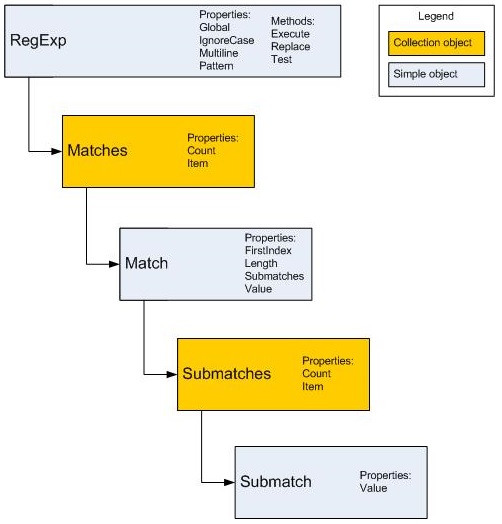


Рис. 12. Объектная [модель](https://filedb.experts-exchange.com/incoming/2009/08_w35/174887/Regular-Expressions-Object-Model.jpg) RegExp

#### Объектная модель .NET

В книге Джеффри Фридл не говорит о VBA, но целую главу посвящает платформе Microsoft .NET Framework, используемой в Visual Basic, C# и C++ (а также в других языках). .NET содержит общую библиотеку регулярных выражений, обеспечивающую единую семантику операций с регулярными выражениями в разных языках. Поддержка регулярных выражений в .NET использует традиционный механизм НКА.

Объектная модель регулярных выражений .NET отличается от модели VBA, представленной на рис. 12, но она настолько наглядна, что будет полезна для изучения. Объектная модель представляет собой совокупность классов, через которую пользователь получает доступ к функциональности регулярных выражений. В .NET эту функциональность обеспечивают семь тесно взаимодействующих классов, но на практике обычно бывает достаточно трех классов, представленных на рис. 12а. На рисунке схематично изображен процесс многократного поиска совпадений \s+(\d+) в строке May•16,•1998

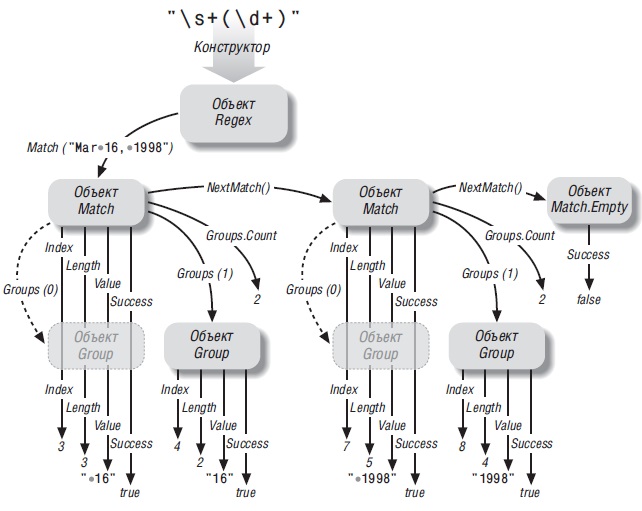


Рис. 12а. Объектная модель библиотеки регулярных выражений .NET

#### Несохраняющие круглые скобки

Допустим нам нужен более тонко настроенный шаблон для поиска температуры. Например [такой](https://regex101.com/r/87hy5m/1): ([-+]?[0-9]+(,[0-9]\*)?)([CF]). Он позволяет искать температуры с необязательным знаком плюс или минус, а также с необязательной дробной частью после запятой.

Обратите внимание: мы добавили необязательную группу (,[0-9]\*)? внутрь первой пары круглых скобок. Поскольку первые скобки используются для сохранения числа, в них также должна быть включена его дробная часть. Однако появление новой пары круглых скобок, пусть даже предназначенных только для применения квантификатора, приводит к побочному эффекту – содержимое этих скобок также сохраняется в переменной. Поскольку открывающая скобка является второй слева в выражении, дробная часть числа сохраняется в $2.



Рис. 13. Вложенные круглые скобки

Появление новых круглых скобок не изменяет смысла выражения [CF] напрямую, но отражается на нем косвенно. Дело в том, что круглые скобки, в которые заключено это подвыражение, становятся третьей парой, а это означает, что содержимое извлекается по ссылке $3 вместо $2.

Чтобы не менять код функции RegExpTemp следует использовать разновидность круглых скобок, которая обеспечивает группировку без сохранения: ([-+]?[0-9]+(**?:**,[0-9]\*)?)([CF]). Здесь открывающая круглая скобка состоит из трех символов (?: Вопросительный знак не имеет отношения к «необязательному» метасимволу. С таким шаблоном подвыражение [CF], хотя и заключено в третий набор круглых скобок, совпадающий с ним текст обозначается переменной $2, поскольку конструкция (**?:**,[0-9]\*)? при подсчете не учитывается. Функция RegExpTemp без каких-либо изменений возвращает корректный результат:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 14. Функция RegExpTemp распознает и десятичные значения

#### Обобщенный пропуск

В VBA используется метасимвол \s. Он обеспечивает сокращенную запись для целого символьного класса, означающего «любой пропуск»: [•\r\t\n\f].

#### Позиционная проверка (lookaround)

Существует класс задач, которые решаются при помощи *позиционной проверкой* (lookaround). Конструкции позиционной проверки обладают определенным сходством с метасимволами границ слов – \b, ^ и $ – они тоже совпадают не с символами, а с позициями в тексте. Однако позиционная проверка используется значительно шире.

Различают четыре типа проверки:

Код Тип

(?= Положительная опережающая (Positive Lookahead)

(?! Отрицательная опережающая (Negative Lookahead)

(?<= Положительная ретроспективная (Positive Lookbehind)

(?<! Отрицательная ретроспективная (Negative Lookbehind)

Подобно конструкции (?: все позиционные проверки не захватывают значения. Опережающая проверка анализирует текст, расположенный справа. Например, подвыражение (?=\d) совпадает в тех позициях, за которыми следует цифра. Ретроспективная проверка анализирует текст в обратном направлении (к левому краю). Например, подвыражение (?<=\d) совпадает в тех позициях, слева от которых находится цифра (т.е., в позиции после цифры).

Позиционные проверки не «поглощают» найденную подстроку. Ниже выделено совпадение регулярного выражения Jeffrey в строке … by Jeffrey Friedl (рис. 15а). Однако в конструкции опережающей проверки то же самое выражение (?=Jeffrey), совпадает только в отмеченной на рис. 15б позиции. Опережающая проверка использует свое подвыражение для поиска, но находит только позицию, с которой начинается совпадение, а не фактически совпадающий текст. Объединение опережающей проверки с выражением, совпадающим с символами текста, открывает дополнительные возможности. Объединенное выражение (?=Jeffrey)Jeff совпадает с текстом Jeff только в том случае, если он является частью слова Jeffrey. Выражение совпадет в строке 15в и не совпадет в 15г.

Выражение (?=Jeffrey)Jeff эквивалентно выражению Jeff(?=rey). Оба выражения совпадают с текстом Jeff только в том случае, если он является частью слова Jeffrey.

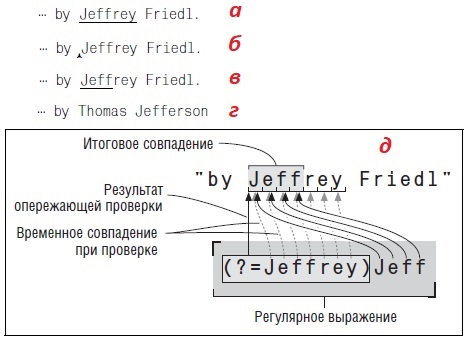


Рис. 15. Поиск совпадения для выражения (?=Jeffrey)Jeff

Обратите внимание и на важность порядка перечисления объединяемых подвыражений. Выражение Jeff(?=Jeffrey) не совпадает ни в одной из приведенных строк – оно совпадет только с текстом Jeff, сразу же после которого следует Jeffrey.

#### Пример ретроспективной проверки

Пусть вы хотите заменить вхождения Jeff на Jeff’s. Задача легко решается без применения проверки, но будет полезна в качестве учебного [примера](https://regex101.com/r/bLn8Om/1):

Изображение выглядит как текст

Автоматически созданное описание

Рис. 16. Ретроспективная и опережающая проверки определяют позицию для вставки апострофа

Первая часть шаблона (?<=\bJeff) ищет позицию слева от которой будут символы Jeff в начале слова. Вторая часть шаблона (?=s\b) ищет позицию за которой идет символ s в конце слова. Т.е., шаблон возвращает не подстроку, а позицию в искомом тексте. В примере она имеет номер 24. Остается заменить «ничто» в позиции 24 на апостроф '.

К сожалению, ретроспективная проверка [не поддерживается](https://www.experts-exchange.com/articles/1336/Using-Regular-Expressions-in-Visual-Basic-for-Applications-and-Visual-Basic-6.html) библиотекой Microsoft VBScript Regular Expressions 5.5. Т.е., в коде VBA можно использовать только опережающую проверку.

#### Разделение групп разрядов в больших числах

Такое разделение часто улучшает внешний вид отчетов, так что весьма полезно. Формально постановка задачи выглядит так: нужно вставить пробелы во всех позициях, у которых количество цифр справа кратно трем, а слева есть хотя бы одна цифра. Группа из трех цифр определяется выражением \d{3}. Заключим его в конструкцию (…)+ чтобы совпадение могло состоять из нескольких групп, и завершим метасимволом $, чтобы гарантировать отсутствие символов после совпадения. Само по себе выражение (\d{3})+$ совпадает с группами из трех цифр, следующими до конца строки, но в конструкции опережающей проверки (?=…) оно совпадает с позицией, справа от которой до конца строки следуют группы из трех цифр – например, в отмеченных позициях текста ’123’456’789. Однако перед первой цифрой пробел не ставится, поэтому совпадение дополнительно ограничим ретроспективной проверкой (?<=\d), гарантирующей, что слева есть хотя бы одна цифра.

Итак, шаблон (?<=\d)(?=(\d{3})+$) в строке «Население России составляет 146247381» выделит две позиции, в которых мы вставим пробел:

Изображение выглядит как текст

Автоматически созданное описание

Рис. 17. [Пробелы](https://regex101.com/r/IhUnMh/1), разделяющие группы цифр

#### Разделение разрядов без ретроспективной проверки

Поскольку VBA не поддерживает ретроспективную проверку, покажем, как можно разделить разряды без нее. Для этой цели подойдет [шаблон](https://regex101.com/r/aGzWaj/1) поиска (\d)(?=(\d{3})+(?!\d)), а в строку замены добавим ссылку на переменную $1 и знак пробела:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 18. Добавление пробелов между разрядами без ретроспективной проверки

Public Function RegExpFindRepl(Text$, Pattern$, Repl$)

On Error GoTo ErrHandl

With CreateObject("VBScript.RegExp")

.Global = True

.Pattern = Pattern

RegExpFindRepl = .Replace(Text, Repl)

Exit Function

End With

ErrHandl:

RegExpFindRepl = CVErr(xlErrValue)

End Function

Добавлена строка .Global = True для поиска всех совпадений (а не только первого). Поиск найдет все цифры (\d), за которыми следует одна или несколько групп из трех цифр (?=(\d{3})+ после которых идет не цифра (?!\d). Вхождение (\d) будет заменено на само себя и пробел $1•. Опережающая проверка имеет еще одно важное свойство по сравнению с обычной группой. Опережающая проверка не влияет на позицию, с которой начинается поиск следующего совпадения. Сравним работу двух шаблонов:

Изображение выглядит как текст

Автоматически созданное описание

Рис. 19. Точка возобновления обычной и опережающей проверки

Первый шаблон требует одну цифру + максимальное число групп по три цифры, и возобновит поиск после нахождения 7 цифр в точке, указанной стрелкой. Второй шаблон требует одну цифру, проверяя, что за ней следует максимальное число групп по три цифры. Но эти группы не входят в возвращаемое значение. Второй шаблон возобновляет поиск после найденной одной цифры.

### Глава 4. Механика обработки регулярных выражений

Существует два разных типа механизмов регулярных выражений: ДКА и НКА. Чтобы определить тип механизма примените регулярное выражение nfa|nfa•not к строке nfa•not; если совпадет только nfa, значит, это традиционный механизм НКА:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 20. Движок VBA реализует механизм НКА

Было бы удобно свести алгоритмы выполнения поиска к нескольким простым правилам. К сожалению, это не удастся. Я могу выделить лишь два универсальных правила:

1. Предпочтение отдается более раннему совпадению.
2. Стандартные квантификаторы (\*, +, ? и {m,n}) работают максимально.

#### Правило 1

Правило 1 работает следующим образом: сначала механизм пытается найти совпадение от самого начала строки, в которой осуществляется поиск (с позиции перед первым символом). Слово «пытается» означает, что с указанной позиции ищутся все возможные комбинации всего (иногда довольно сложного) регулярного выражения. Если после перебора всех возможностей совпадение так и не будет найдено, вторая попытка делается с позиции, предшествующей второму символу. Эта процедура повторяется для каждой позиции в строке. Результат «совпадение не найдено» возвращается лишь в том случае, если совпадение не находится после перебора всех позиций до конца строки.

Например, при поиске cat в строке The dragging belly indicates your cat is too fat совпадение находится в слове indicates… Где в строке The dragging belly indicates your cat is too fat совпадет выражение fat|cat|belly|your? При каждой попытке осуществляется полный поиск регулярного выражения, поэтому fat|cat|belly|your совпадет belly раньше, чем с fat, хотя fat в списке альтернатив находится в более ранней позиции.

#### Правило 2

Правило 2 гласит, если некоторый элемент может совпадать переменное количество раз, механизм всегда пытается найти максимальное количество повторений. Максимальные квантификаторы всегда захватывают больше повторений, чем им требуется по минимуму, поэтому их иногда называют *жадными*.

Например, шаблон ^.\*([0-9][0-9]) находит две последних цифры в строке, где бы они ни располагались, и сохраняет их в переменной $1. Это происходит следующим образом: сначала .\* сопоставляется со всей строкой. Следующее подвыражение ([0-9][0-9]) является обязательным, и при отсутствии совпадения для него «говорит»: «Эй, .\*, ты взял лишнее! Верни мне что-нибудь, чтобы и для меня нашлось совпадение». «Жадные» компоненты всегда сначала стараются захватить побольше, но потом всегда отдают излишки, если при этом достигается общее совпадение.

Применим ^.\*([0-9][0-9]) к строке about 24 characters long. После того как r.\* совпадет со всей строкой, необходимость совпадения для первого класса [0-9] заставит .\* уступить символ g (последний из совпавших). Однако при этом совпадение для [0-9] по-прежнему не находится, поэтому .\* уступает n в слове long. Цикл повторяется 15 раз, пока .\* не доберется до цифры 4. К сожалению, совпадение появляется лишь для первого класса [0-9]. Второму классу, как и прежде, совпадать не с чем. Поэтому .\* уступает 2, с которой может совпасть первый класс [0-9]. Цифра 4 освобождается и совпадает со вторым классом, а все выражение – с *about 24*. Переменной $1 присваивается строка 24.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 21. Жадный квантификатор отдает по минимуму

Если вы думаете, что ([0-9][0-9]) можно заменить на ([0-9]+), то ошибаетесь. Какой текст совпадет, если применить выражение ^.\*([0-9]+) к строке 'Copyright 2003.'? Что запишется в переменную $1? Подвыражение .\* как и в первом примере идет на частичные уступки. Сначала уступит завершающую точку, а затем 3, после чего появляется совпадение для класса [0-9]. К классу применен квантификатор +, для которого один экземпляр соответствует минимальным требованиям; дальше в строке идет точка, поэтому второго совпадения нет.

Но в отличие от предыдущего случая никаких обязательных элементов дальше нет, поэтому .\* не приходится уступать 0 и другие цифры. В принципе подвыражение [0-9]+ с радостью согласилось бы на такой подарок, но нет: кто первым приходит, того первым и обслуживают. Жадные квантификаторы по-настоящему скупы: если что-то попало им в лапы, они отдадут это «что-то» только по принуждению. В итоге переменная $1 содержит только символ 3.

Фразы типа *.\* уступает...* или *[0-9] заставляет...* на самом деле неточны. Я воспользовался этими оборотами, потому что они достаточно наглядны и при этом приводят к правильному результату. Однако события, которые в действительности происходят за кулисами, зависят от базового типа механизма, ДКА или НКА.

#### Механизмы регулярных выражений

Говорят, что механизм НКА «управляется регулярным выражением», а механизм ДКА «управляется текстом».

Рассмотрим алгоритм поиска совпадения выражения to(nite|knight|night) в тексте …tonight… Механизм просматривает регулярное выражение по одному компоненту, начиная с t, и проверяет, совпадает ли компонент с «текущим текстом». В случае совпадения проверяется следующий компонент. Процедура повторяется до тех пор, пока не будет найдено совпадение для всех компонентов регулярного выражения. Управление передается внутри регулярного выражения от компонента к компоненту, поэтому я говорю, что такой механизм «управляется регулярным выражением».

Механизму ДКА сканирует строку и следит за всеми «потенциальными совпадениями». В описанном выше примере механизм узнает о начале потенциального совпадения, как только в строке встречается символ t:



Рис. 22. Работа механизма ДКА после первого совпадения

Каждый следующий сканируемый символ обновляет список потенциальных совпадений. Через несколько символов мы приходим к следующей ситуации с двумя потенциальными совпадениями (одна альтернатива, knight, к этому моменту уже отвергается):



Рис. 23. Механизм ДКА после 4 совпадений

Проверка следующего символа, g, исключает и первую альтернативу. После сканирования h и t механизм понимает, что он нашел полное совпадение, и успешно завершает свою работу.

Механизм ДКА «управляется текстом», поскольку его работа зависит от каждого просканированного символа строки. В приведенном выше примере частичное совпадение может быть началом любого количества разных (но потенциально возможных) совпадений. Отвергаемые совпадения исключаются из дальнейшего рассмотрения по мере сканирования последующих символов.

Механизм ДКА работает быстрее: каждый символ в строке проверяется не более одного раза. Если символ совпал, вы еще не знаете, войдет ли он в итоговое совпадение (символ может быть частью потенциального совпадения, которое позднее будет отвергнуто), но, поскольку механизм отслеживает все потенциальные совпадения одновременно, символ достаточно проверить только один раз.

Расшифровка аббревиатур: НКА – недетерминированный конечный автомат и ДКА – детерминированный конечный автомат.

#### Последствия для пользователей

Поскольку механизм НКА управляется регулярным выражением, при его использовании необходимо очень хорошо разбираться в том, как происходит поиск совпадений. В примере с tonight для повышения эффективности поиска можно было бы записать регулярное выражение в одном из следующих вариантов: to(ni(ght|te)|knight), tonite|toknight|tonight, to(k?night|nite). В ДКА все наоборот: поскольку механизм отслеживает все совпадения одновременно, любые различия в представлении несущественны, если они в конечном счете определяют один и тот же совпадающий текст.

Чтобы досконально разобраться во всех тонкостях НКА, необходимо рассмотреть одну из важнейших концепций этого механизма – возврат.

#### Возврат

Иногда механизму НКА приходится выбирать дальнейший ход действий. Такие ситуации связаны с использованием квантификаторов (нужно ли пытаться найти следующее совпадение) и конструкций выбора (какую альтернативу искать сейчас, а какую оставить на будущее). В этом случае механизма НКА выбирает один вариант и запоминает другой, чтобы позднее вернуться к нему в случае необходимости.

Если выбранный вариант и все выражение успешно совпадают, процесс поиска завершается. Если какая-либо из оставшихся частей регулярного выражения приводит к неудаче, механизм регулярных выражений возвращается к развилке, где было принято решение, и продолжает поиск с другим вариантом.

С концепцией возврата связано два вопроса:

* какой вариант из нескольких возможных должен проверяться в первую очередь?
* какой из сохраненных вариантов должен использоваться механизмом при возврате?

В тех ситуациях, где механизм выбирает между попытками найти совпадение или отказаться от его поиска (например, при использовании квантификаторов ?, \* и других), механизм всегда сначала пытается найти совпадение для максимальных квантификаторов и пропустить совпадение для минимальных квантификаторов.

При локальной неудаче происходит возврат к последнему из сохраненных вариантов. Перебор вариантов осуществляется по правилу LIFO (последним пришел – первым обслужен). При описании LIFO часто используется аналогия со стопкой тарелок: первой со стопки снимается та тарелка, которая была поставлена в нее последней.

Сохраненное состояние определяет точку, с которой при необходимости можно начать очередную проверку. В нем сохраняется как текущая позиция в регулярном выражении, так и позиция в строке, с которой начинается проверка.

Рассмотрим простой пример – поиск выражения *ab?c* в строке *abc*. После совпадения c *a* текущее состояние показано на рис. 24а. Перед поиском b? механизм должен принять решение – пытаться найти b или нет? Из-за своей жадности квантификатор ? начинает искать совпадение. Но для того, чтобы в случае неудачи он мог вернуться к этой точке, в пустой на этот момент список сохраненных состояний добавляется запись на рис. 24б.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 24. Механика поиска: а) текущее состояние после совпадения буквы а; б) сохраненное состояние; в) текущее состояние после совпадения буквы b

Это означает, что механизм позднее продолжит поиск с компонента, следующего в регулярном выражении после b?, и сопоставит его с текстом, находящимся до b (т.е. в текущей позиции). В сущности, это означает, что литерал b пропускается, что допускает квантификатор ?. В нашем примере для *b* находится совпадение, поэтому новое текущее состояние выглядит как на рис. 24в. Последний компонент *с* тоже совпадает. Следовательно, механизм нашел общее совпадение для всего регулярного выражения. Единственное сохраненное состояние становится ненужным, и механизм попросту забывает о нем.

Если бы поиск производился в строке ac, все происходило бы точно так же до попытки найти совпадение для b. Конечно, на этот раз совпадение не обнаруживается. Это означает, что путь, пройденный при попытке найти совпадение для b?, оказался тупиковым. Но поскольку у нас имеется сохраненное состояние, к которому можно вернуться, «локальная неудача» вовсе не означает общей неудачи. Механизм возвращается к последнему сохраненному состоянию и превращает его в новое текущее состояние. В нашем примере восстанавливается состояние сохраненное перед поиском b. На этот раз *c* совпадают, что обеспечивает общее совпадение.

#### Минимальный поиск

Ранее (см. рис. 5) мы изучили так называемые жадные квантификаторы: +, ?, \* и {min, max}. В VBA также поддерживаются минимальные (нежадные, ленивые) квантификаторы: +?, ??, \*? и {min, max}?. Такие квантификаторы покрывают минимальный набор символов и расширяют его, если последующие сцепленные выражения не выполняются (подробнее см. [Квантификаторы в регулярных выражениях](https://habr.com/ru/post/68345/)). Например, шаблон \(.+\) вернет весь текст от первой открывающей до последней закрывающей скобки в тексте. В то же время \(.+?\) остановится на первой закрывающей скобке:

Изображение выглядит как стол

Автоматически созданное описание

Рис. 25. Нежадный квантификатор +? остановится при первом совпадении

#### Подробнее о максимализме и о возврате

Элемент шаблона .\* всегда распространяет совпадение до конца строки. Позднее часть захваченного может быть возвращена, если это необходимо для общего совпадения. Например, какое совпадение шаблон ".\*" найдет в строке:

The name "McDonald's" is said "makudonarudo" in Japanese

После совпадения первого символа " управление передается конструкции .\*, которая немедленно захватывает все символы до конца строки. Она начинает нехотя отступать (под нажимом механизма регулярных выражений), но только до тех пор, пока не будет найдено совпадение для последней кавычки. Найденное совпадение будет выглядеть так:

"McDonald's" is said "makudonarudo"

Как же ограничить совпадение строкой "McDonald's"? Главное – понять, что между кавычками должно находиться не «все, что угодно», а «все, что угодно, кроме кавычек». Если вместо .\* воспользоваться выражением [^"]\*, совпадение не пройдет дальше закрывающей кавычки. После совпадения первой кавычки [^"]\* стремится захватить как можно большее потенциальное совпадение. В данном случае это совпадение распространяется до кавычки, следующей после McDonald's. На этом месте поиск прекращается, поскольку [^"] не совпадает с кавычкой, после чего управление передается закрывающей кавычке в регулярном выражении. Она благополучно совпадает, обеспечивая общее совпадение:

"McDonald's"

С проблемой также справится минимальный квантификатор ".\*?"

Изображение выглядит как стол

Автоматически созданное описание

Рис. 26. Варианты обхода жадности квантификатора \*

#### Минимальный квантификатор не панацея

Усложним пример. Допустим нам надо найти в HTML тексте слова, выделенные полужирным шрифтом, т.е. ограниченные тегами <B>…</B>. Шаблон <B>.\*?</B> справится со строкой (рис. 27а)…

…<B>Billions</B> and <B>Zillions</B> of suns…

… но не справится со строкой (рис. 27б)…

…<В>Вillions and <B>Zillions</B> of suns…

Ничто не помешало фрагменту шаблона .\*? пройти мимо <B> в начале слова Zillions к завершающему тегу </B>.

Задачу можно решить с помощью опережающей негативной проверки (рис. 27в). Нужно проверять не любой знак, а лишь тот, что не совпадает с <B>. Перед точкой появляется блок опережающей проверки (?!<B>) А весь шаблона примет вид <B>((?!<B>).)\*?</B> Его можно прочитать так: после тега <B> взять любое число символов в таких позициях, что далее не следует <B>, до ближайшего </B>

Изображение выглядит как стол

Автоматически созданное описание

Рис. 27. Опережающая проверка

Далее подробно рассмотрены захватывающие (или сверхжадные) квантификаторы ?+, ++, \*+, … и атомарная группировка (?>…Подробнее см. [Атомарная группировка, сверхжадный квантификатор](https://ru.stackoverflow.com/questions/525991/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F-%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0-%D1%81%D0%B2%D0%B5%D1%80%D1%85%D0%B6%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9-%D0%BA%D0%B2%D0%B0%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80). К сожалению VBA не поддерживает этот функционал.

#### Конструкции выбора

При передаче управления конструкции выбора в строке может совпасть любое количество из перечисленных альтернатив, но какой из них будет отдано предпочтение? Конструкция выбора упорядочена. Она предоставляет автору регулярного выражения больше возможностей для управления поиском совпадения – он может действовать по принципу «попробовать сначала одно, затем другое и, наконец, третье, пока не будет найдено совпадение». Мораль: если при упорядоченной конструкции выбора один и тот же текст может совпасть с несколькими альтернативами, будьте особенно внимательны при выборе порядка альтернатив (как правило первым нужно указать самый длинный шаблон).

### Глава 5. Практические приемы построения регулярных выражений

В хорошо написанном регулярном выражении должны быть сбалансированы несколько факторов:

* Регулярное выражение должно совпадать там, где нужно, и нигде больше
* Регулярное выражение должно быть понятным и управляемым
* При использовании механизма НКА выражение должно быть эффективным (т.е. быстро приводить к совпадению или несовпадению, в зависимости от результата поиска)

#### Поиск строк с продолжением

Одна логическая строка может распространяться на несколько физических строк. Признаком продолжения предыдущей строки, например, является символ \, стоящий перед символом новой строки. Рассмотрим следующий пример:

*SRC=array.c builtin.c eval.c field.с gawkmisc.c iо.с main.с \*

*missing.с msg.c node.с re.с version.с*

Для поиска строк в формате «переменная=значение» обычно используется выражение ^\w+=.\*, но в нем не учитываются возможные строки продолжения (предполагается, что в используемой программе точка не совпадает с символом новой строки). Чтобы выражение находило строки продолжения, можно попытаться присоединить к нему (\\\n.\*)\*, в результате чего получится ^\w+=.\*(\\\n.\*)\*. Однако в НКА такое решение не работает. Дело в том, что первое подвыражение .\* захватывает символ \ и «отнимает» его у подвыражения (\\\n.\*)\*, в котором оно должно совпасть. *Первый практический урок:* если мы не хотим, чтобы совпадение распространялось за \, необходимо сообщить об этом в регулярном выражении, заменив каждую точку на [^\n\\] (обратите внимание на включение символа \n в инвертированный класс; предполагается, что точка не совпадает с символом новой строки, поэтому ее замена тоже не должна с ним совпадать.

Выполнив необходимые изменения, мы получаем: \w+=[^\n\\]\*(\\\n[^\n\\]\*)\* (рис. 28а).

Такое решение работает, но решив одну проблему, мы сами создали другую: теперь в строках не допускаются символы \, кроме тех, которые завершают строку (рис. 28б).

До настоящего момента наше решение работало по принципу: «сначала найти совпадение для строки, а затем попытаться найти совпадение для строки продолжения, если она есть». Давайте воспользуемся другим принципом, который чаще подходит для общих случаев: сосредоточим внимание на том, что же может совпадать в каждой конкретной позиции. При поиске совпадения для строки нам нужны либо обычные символы (отличные от \ и \n), либо комбинации \ + любой символ. Мы приходим к трансформации \w+=[^\n\\]\* в ^\w+=([^\n\\]|\\.)\*, а весь шаблон приобретает вид (рис. 28в) \w+=([^\n\\]|\\.)\*(\\\n[^\n\\]\*)\*

Изображение выглядит как текст

Автоматически созданное описание

Рис. 28. Поиск строк с продолжением

#### Поиск IP-адреса

Задача – поиск адресов IP, четырех чисел, разделенных точками. Например, 1.2.3.4. Числа нередко дополняются нулями до трех цифр – 001.002.003.004. Если вы хотите проверить строку на присутствие IP-адреса, можно воспользоваться выражением [0-9]\*\.[0-9]\*\.[0-9]\*\.[0-9]\*, но это решение настолько неопределенное, что оно совпадает даже в строке

and then . . . . . . ?

Регулярное выражение даже не требует существования чисел – единственным требованием является наличие трех точек (между которыми нет ничего, кроме цифр, которых тоже может не быть).

Заменим звездочки плюсами, поскольку каждое число должно содержать хотя бы одну цифру. Чтобы гарантировать, что вся строка состоит только из IP-адреса, мы заключаем регулярное выражение в ^…$ Получается ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$

Можно заменить [0-9] метасимволом \d. Считается, что так будет более наглядно ^\d+\.\d+\.\d+\.\d+$ Но это выражение по-прежнему совпадает с конструкциями, которые не являются IP-адресами, например 1234.5678.9101112.131415. (Каждое число в IP-адресе находится в интервале 0–255.) Можно потребовать, чтобы каждое число состояло ровно из трех цифр, ^\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d$ Однако это требование слишком жесткое. Необходимо поддерживать числа, состоящие из одной и двух цифр (например, 1.234.5.67). Можно воспользоваться интервальным квантификатором {min, max} ^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$ (рис. 29а).

Если вы стремитесь к соблюдению всех правил, придется учесть, что \d{1,3} может совпасть с числом 999, которое больше 255 и поэтому не является допустимым компонентом IP-адреса (рис. 29б).

Если число состоит всего из одной или двух цифр, принадлежность числа нужному интервалу проверять не нужно, поэтому для этих случаев вполне достаточно \d|\d\d. Проверка не нужна и в том случае, если число из трех цифр начинается с 0 или 1 , поскольку оно заведомо принадлежит интервалу 000–199. Следовательно, в конструкцию выбора можно добавить [01]\d\d; получается \d|\d\d|[01 ]\d\d. Число из трех цифр, начинающееся с 2, допустимо лишь в том случае, если оно равно 255 и меньше. Следовательно, если вторая цифра меньше 5, значит, число правильное. Если вторая цифра равна пяти, третья цифра должна быть меньше 6. Все сказанное выражается в форме 2[0-4]\d|25[0-5].

В результате получается выражение \d|\d\d|[01]\d\d|2[0-4]\d|25[0-5]. Первые три альтернативы можно объединить [01]?\d\d?|2[0-4]\d |25[0-5]. Обратите внимание: использование \d\d? в первой альтернативе вместо \d?\d немного ускоряет выявление неудачи при полном отсутствии цифр. Окончательный результат выглядит так (рис. 29в):

^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.

([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$

Изображение выглядит как текст, стол

Автоматически созданное описание

Рис. 29. Поиск IP адреса

Это выражение все еще допускает строку 0.0.0.0, которая семантически неверна, поскольку в ней все цифры равны нулю. Используя опережающую проверку, можно запретить этот конкретный случай, разместив конструкцию (?!0+\.0+\.0+\.0+$) после начального ^ (рис. 29г), но в какой-то момент в зависимости от конкретной ситуации вы должны решить, стоит ли стремиться к дальнейшей точности – хлопоты начинают приносить все меньше пользы. Иногда бывает проще лишить регулярное выражение каких-то второстепенных возможностей. Например, можно вернуться к выражению ^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$, заключить каждый компонент в круглые скобки, чтобы числа сохранились в переменных $1, $2, $3 и $4, и проверить их в программном коде за пределами регулярного выражения.

#### Удаление пути из полного имени файла

Например, превратим \usr\local\bin\gcc в gcc. Мы хотим удалить все символы до последнего \ включительно. Если в выражении нет ни одного символа \, значит, все нормально и делать ничего не нужно. Конструкцией .\* часто злоупотребляют, но ее максимализм в данном случае уместен. В выражении ^.\*\\ подвыражение .\* поглощает всю строку, но затем отступает назад к последнему символу \, чтобы обеспечить совпадение (рис. 30а).

Изображение выглядит как стол

Автоматически созданное описание

Рис. 30. Извлечение имени файла: а) через удаление пути; б) через прямое извлечение имени

Что произойдет, если забыть о начальном метасимволе ^ (это типичная ошибка), и применить выражение к строке, не содержащей ни одного символа \. Как обычно, механизм регулярных выражений начинает поиск от начала строки. Конструкция .\* распространяется до конца текста, но затем постепенно отступает в поисках символа \. Рано или поздно будут возвращены все символы, захваченные предварительным совпадением .\*, но совпадение так и не найдется. Механизм регулярных выражений приходит к выводу, что от начала строки совпадения не существует, и … проверка возобновится во второй позиции. Что только снизит эффективность работы программы: если шаблон ^.\*\\ не найден с первой позиции, то заведомо не будет найдет и в других позициях проверяемого текста. Метасимвол ^ позволит ограничиться проверкой только с первой позиции.

Возможен и другой подход: напрямую извлечь имя файла, которое состоит из всех символов в конце строки, отличных от \. На этот раз якорный метасимвол $ предназначен не только для оптимизации, нам действительно нужна привязка к концу строки (рис. 30б).

#### Путь и имя файла

Можно воспользоваться выражением ^(.\*)\(.\*)$, чтобы присвоить соответствующие компоненты переменным $1 и $2. Выражение выглядит симметрично, но, зная, как работают максимальные квантификаторы, можно быть уверенным в том, что первое подвыражение .\* никогда не оставит для $2 текст, в котором присутствует символ \ Первоначально .\* захватывает все символы, после чего происходит возврат для нахождения самого правого символа \ Таким образом, $1 будет содержать путь к файлу, а $2 – завершающее имя файла (рис. 31а, 31б).

Изображение выглядит как стол

Автоматически созданное описание

Рис. 31. Путь и имя файла

Чтобы регулярное выражение точнее передавало, что мы ищем, для имени файла лучше использовать [^\\]\*. Получается ^(.\*)\\([^\\]\*)$ (рис. 31в, 31г). Это выражение более наглядно, и кроме того выполняет документирующие функции.

#### Исключение нежелательных совпадений

Допустим, вам нужно найти число: целое или десятичное. Простейшее решение -?[0-9]\*,?[0-9]\* не выдерживает критики. Хотя это выражение успешно совпадает с такими примерами, как 1, 12923, 31, -272,37, и даже -,0, оно также совпадет с любой текстовой строкой. Внимательно взгляните на регулярное выражение – все его элементы являются необязательными. Если там присутствует число и если оно находится в начале строки, то это число будет успешно найдено. В общем случае выражение совпадет всегда. Оно совпадет с числом или «ничем» в начале строки. Более того, оно совпадет с «ничем» даже в начале такой строки, как ’номер 123’, поскольку совпадение с «ничем» обнаруживается раньше, чем с числом!

Поэтому необходимо точно сформулировать, что такое число. Например, число должно содержать как минимум одну цифру. Сначала предположим, что хотя бы одна цифра находится перед десятичной запятой. В этом случае для цифр целой части используется квантификатор +: -?[0-9 ]+ При написании подвыражения для поиска необязательной десятичной запятой и последующих цифр необходимо понять, что наличие дробной части полностью зависит от наличия самой десятичной запятой. Если воспользоваться наивным выражением типа ,?[0-9]\*, то [0-9]\* сможет совпасть независимо от наличия десятичной запятой.

(,[0-9]\*)? Здесь вопросительный знак квантифицирует не одну десятичную запятую, а комбинацию десятичной запятой с последующими цифрами. Внутри этой комбинации десятичная запятая должна присутствовать обязательно: если ее нет, [0-9]\* вообще не получит шанса на совпадение.

Объединяя все сказанное, мы получаем -?[0-9]+(,[0-9]\*)? Это выражение все еще не находит числа вида ,007, поскольку оно требует наличия хотя бы одной цифры перед десятичной запятой. Если изменить левую часть и разрешить нулевое количество цифр, придется изменять и правую часть, потому что нельзя допустить, чтобы все цифры выражения были необязательными.

Одно из возможных решений – добавить для таких ситуаций специальную альтернативу: -?[0-9]+(,[0-9]\*)?|-?,[0-9]+ Вы обратили внимание на то, что во второй альтернативе также предусмотрено наличие необязательного минуса? Об этом легко забыть. Этот минус можно вынести за скобки: -?([0-9]+(,[0-9]\*)?|,[0-9]+)

Хотя такое решение работает лучше первоначального предложения, оно все равно найдет совпадение в строке 2003,04,12. Хорошее знание контекста, в котором будет применяться регулярное выражение, помогает выдержать основное требование – обеспечить совпадения там, где нужно, и исключить их там, где они излишни.

#### Синхронизация

Предположим, данные представляют собой сплошную последовательность почтовых индексов США, каждый из которых состоит из 5 цифр. Вы хотите выбрать из потока все индексы, начинающиеся, допустим, с цифр 44. Ниже приведен пример данных, в котором нужные индексы выделены жирным шрифтом:

03824’53144’94116’15213’**44182’**95035’**44272’**75201’02174’43235

Многократное применение выражения \d\d\d\d\d выделит отдельные индексы во входном потоке данных. Но выражение 44\d\d\d для нахождения индексов, начинающихся с 44, не пройдет. После того как попытка совпадения завершится неудачей, механизм смещается на один символ, в результате чего теряется синхронизация 44… с началом очередного индекса. При использовании 44\d\d\d будут найдены все последовательности, начинающиеся с 44…

Изображение выглядит как стол

Автоматически созданное описание

Рис. 32. Без синхронизации теряется привязка к 5-цифровым группам

Нам нужна возможность синхронизации механизма регулярных выражений, при которой регулярное выражение пропускало бы отвергнутые индексы. Важно, чтобы индекс пропускался полностью, а не по одному символу, как при автоматическом смещении текущей позиции поиска.

Ниже приведено два варианта пропуска ненужных индексов в регулярном выражении. Чтобы добиться желаемого эффекта, достаточно вставить их перед конструкцией (44\d\d\d) Совпадения для неподходящих индексов заключены в несохраняющие круглые скобки (?:…), что позволяет игнорировать их и сохранить подходящий индекс в переменной $1:

(?:[^4]\d\d\d\d|\d[^4]\d\d\d)\*

Это решение можно назвать «методом грубой силы»: мы активно пропускаем все индексы, начинающиеся с чего-либо, кроме 44. Мы не можем использовать конструкцию (?:[^4][^4]\d\d\d)\*, поскольку она не пропускает ненужные индексы типа 43210.

Для демонстрации примера в Excel я модифицировал функцию RegExpFindRepl, добавив возможность множественной замены

Public Function RegExpSyncing( \_

Text As String, \_

Pattern As String, \_

Repl As String, \_

Optional Item As Integer = 1 \_

) As String

Dim m As Object

On Error GoTo ErrHandl

Set regEx = CreateObject("VBScript.RegExp")

regEx.Global = True

regEx.Pattern = Pattern

If regEx.test(Text) Then

Set Matches = regEx.Execute(Text)

TextS = Matches.Item(Item - 1)

RegExpSyncing = regEx.Replace(TextS, Repl)

Exit Function

End If

ErrHandl:

RegExpSyncing = CStr(CVErr(xlErrValue))

End Function

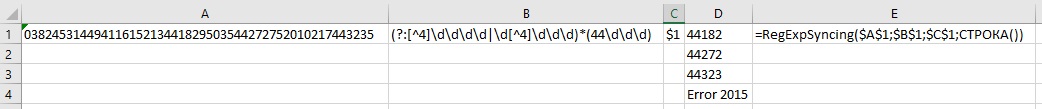


Рис. 33. Первая попытка синхронизации

К сожалению, наше регулярное выражение не справилось с задачей. Как только в конце текста не останется ни одного нужного индекса, поиск совпадения завершится неудачей. Как всегда, при этом вступает в действие механизм смещения текущей позиции, и поиск продолжится с позиции внутри очередного индекса. После четвертого смещения регулярное выражение пропускает 10217 и ошибочно находит «индекс» 44323.

Регулярное выражение прекрасно работает, пока оно применяется от начала индекса, но смещение текущей позиции нарушает все планы. Эта проблема может быть решена за счет предотвращения возврата или за счет создания регулярного выражения, в котором возврат не вызывает проблем.

Один из способов предотвращения возврата основан на том, чтобы сделать подвыражение (44\d\d\d) необязательным, присоединив к нему квантификатор ? Мы знаем, что предшествующее подвыражение (?:[^4]\d\d\d\d|\d[^4]\d\d\d)\* завершается только на границе действительного индекса или если индексов больше нет. В итоге (44\d\d\d)? совпадает с искомым индексом, если он обнаружен, но не приводит к возврату при его отсутствии.

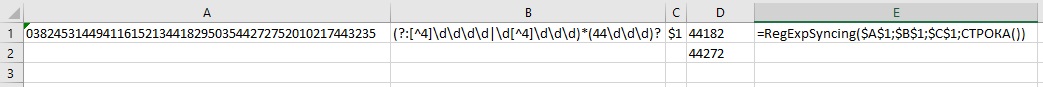


Рис. 34. Необязательный квантификатор ? в конце регулярного выражения решает проблему

Другой [способ](https://regex101.com/r/gdjBbM/1) – поставить перед выражением метасимвол \G. Поскольку выражение строилось таким образом, чтобы совпадение заканчивалось на границе индекса, все последующие совпадения при отсутствии промежуточных смещений текущей позиции также будут начинаться с границы индекса. А если промежуточное смещение все-таки произошло, начальный метасимвол \G немедленно приводит к неудаче, потому что в большинстве диалектов он успешно совпадает лишь при отсутствии промежуточных смещений. К сожалению VBA не поддерживает \G.

Второе решение основано на негативной опережающей проверке (?:(?!44)\d\d\d\d\d)\* которая также активно пропускает индексы, не начинающиеся с 44:

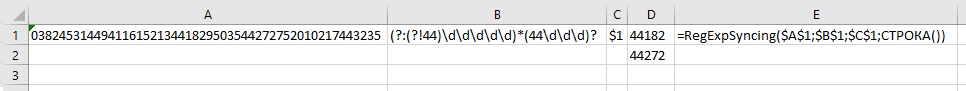


Рис. 35. Решение на основе негативной опережающей проверки

### Глава 6. Построение эффективных регулярных выражений

Наша цель – правильность и эффективность. Это означает, что выражение должно находить все нужное, не находить ничего лишнего, и притом быстро. Главное, что для этого нужно, – доскональное понимание возврата и умение избегать его там, где это возможно.

Начнем с примера, который продемонстрирует, какими важными могут быть проблемы возврата и эффективности. Допустим, вы ищите строки, заключенные в кавычки с помощью регулярного выражения "(\\.|[^\\"])\*" Это выражение найдет текст, заключенный в кавычки (рис. 36а). Но пропустит экранированные кавычки (рис. 36б). При этом распознает, если экранированы не кавычки, а сам слеш, например, в последовательности \\" (рис. 36в):

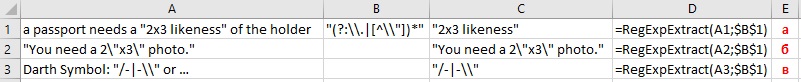


Рис. 36. Поиск текста в кавычках

Регулярное выражение работает, но в механизме НКА конструкция выбора, применяемая к каждому символу, работает крайне неэффективно. Для каждого «обычного» символа в строке (не кавычки и не экранированного символа) механизм должен проверить \\. обнаружить неудачу и вернуться, чтобы в результате найти совпадение для [^\\"] Если выражение используется в ситуации, когда важна эффективность, конечно, хотелось бы немного ускорить обработку этого выражения.

#### Простое изменение – начинаем с более вероятного случая

Поскольку обычных символов больше, чем экранированных, напрашивается изменение порядка альтернатив. Если [^\\"] стоит на первом месте, то возврат происходит лишь при обнаружении экранированного символа в строке. Рис. 37 показывает отличия между этими двумя выражениями. Уменьшение количества стрелок означает, что для первой альтернативы соответствующего выражения совпадения находятся чаще. Это приводит к уменьшению количества возвратов.



Рис. 37. Изменение порядка альтернатив

Оценивая последствия такого изменения для эффективности, необходимо задать себе вопрос: «Когда изменение приносит наибольшую пользу – когда текст совпадает, когда текст не совпадает или в любом случае?».

Изменение приводит к ускорению поиска лишь при наличии совпадения. НКА может сделать вывод о неудаче только после того, как будут проверены все возможные комбинации. Следовательно, если попытка окажется неудачной, значит, были опробованы все комбинации, поэтому порядок не важен. В следующей таблице перечислено количество проверок и возвратов для некоторых случаев (чем меньше число, тем лучше). Как видите, для нового выражения быстродействие возрастает (уменьшается количество возвратов). В ситуации без совпадения (последняя строка в таблице) оба механизма проверяют все возможные комбинации, поэтому и результаты оказываются одинаковыми.

Изображение выглядит как стол

Автоматически созданное описание

Рис. 38. Последовательность альтернатив влияет на скорость лишь при наличии совпадений

Помните, что изменение порядка альтернатив допустимо лишь в том случае, если порядок не влияет на успешность совпадения.

#### «Экспоненциальный» поиск

На рис. 37 видно, что в обоих случаях квантификатор \* должен последовательно перебрать все нормальные символы, при этом он снова и снова входит в конструкцию выбора (и круглые скобки) и выходит из нее. Все эти действия сопряжены с лишней работой, от которой хотелось бы избавиться. Выражение можно оптимизировать, если учесть, что когда [^\\"] относится к «нормальным» (не кавычки и не экранированные символы) случаям, то при его замене на [^\\"]+ одна итерация (…)\* прочитает все последовательно стоящие обычные символы. При отсутствии экранированных символов будет прочитана вся строка. Это позволяет найти совпадение практически без возвратов и сокращает многократное повторение \* до абсолютного минимума.

Однако, прежде чем в выражении появился +, [^\\"] относилось только к \*, и количество вариантов совпадения в тексте выражения ([^\\"])\* было ограниченным. Выражение могло совпасть с одним символом, двумя символами и т.д. до тех пор, пока каждый символ целевого текста не будет проверен максимум один раз. Конструкция с квантификатором могла и не совпадать со всей целевой строкой, но в худшем случае количество совпавших символов было прямо пропорционально длине целевого текста. Потенциальный объем работы возрастал теми же темпами, что и длина целевого текста.

У «эффективного» выражения ([^\\"]+)\* количество вариантов, которыми + и \* могут поделить между собой строку, растет с экспоненциальной скоростью. Возьмем строку makudonarudo. Следует ли рассматривать ее как 12 итераций \*, когда каждое внутреннее выражение [^\\"]+ совпадает лишь с одним символом m•a•k•u•d•o•n•a•r•u•d•о А может, одну итерацию \*, при которой внутреннее выражение [^\\"]+ совпадает со всей строкой makudonarudo А может, три итерации \*, при которых внутреннее выражение [^\\"]+ совпадает соответственно с 5, 3 и 4 символами makud•ona•rudo Или 2, 2, 5 и 3 символами ma•ku•do•narudo Или...

Возможностей очень много. Для каждого нового символа в строке количество возможных комбинаций удваивается, и механизм НКА должен перепробовать все варианты, прежде чем вернуть ответ, поэтому подобная ситуация называется «экспоненциальным поиском» (также иногда встречается термин «суперлинейный поиск»). Для 12 символов 4096 комбинаций обрабатываются быстро, но обработка миллиона с лишним комбинаций для 20 символов занимает уже несколько секунд. Для 30 символов миллиард с лишним комбинаций обрабатывается несколько часов, а для 40 символов обработка займет уже больше года. Конечно, это неприемлемо.

В процессе выполнения экспоненциального поиска может показаться, что программа «зависает». Впервые столкнувшись с проблемой, я решил, что в программе обнаружилась какая-то ошибка. Оказывается, она всего лишь занималась бесконечным перебором комбинаций. Теперь, когда я это понял, подобные выражения вошли в набор тестов для проверки типа механизма:

* Если выражение обрабатывается быстро даже в случае несовпадения, это, вероятно, ДКА.
* Если выражение обрабатывается быстро только при наличии совпадения, это традиционный НКА.

Примените выражение X(.\*)\*X к строке вида =XX====================== Если команда выполняется быстро, это ДКА. Если выполнение команды занимает очень много времени или на экране появилось предупреждение о переполнении стека или прерывании затянувшейся операции, работает НКА.

Изображение выглядит как текст

Автоматически созданное описание

Рис. 39. Двойной квантификатор [подвешивает](https://regex101.com/r/Q4f6ne/1) НКА

#### Возврат с глобальной точки зрения

На локальном уровне возврат – это обратный переход к непроверенному варианту. На глобальном уровне дело обстоит сложнее. Начнем с примера поиска совпадения ".\*" в строке

The name "McDonald's" is said "makudonarudo" in Japanese

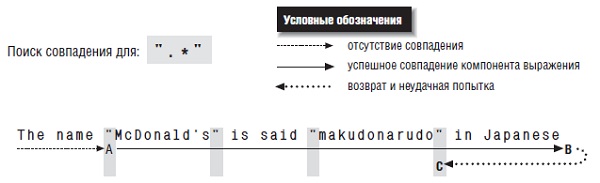


Рис. 40. Успешный поиск

Поиск регулярного выражения осуществляется в каждой позиции строки, начиная с первой. Поскольку несовпадение обнаруживается при проверке первого элемента (кавычка), ничего интересного не происходит до того, когда поиск начнется с позиции A. В этот момент проверяется оставшаяся часть выражения, однако подсистема смещения текущей позиции знает, что если попытка приведет к тупику, все регулярное выражение будет проверено заново со следующей за А позиции.

.\* распространяется до самого конца строки В. Ни один из 46 символов, совпавших с .\*, не является обязательным, поэтому в процессе поиска механизм накапливает 46 сохраненных состояний, к которым он может вернуться в случае необходимости. Находясь в точке В механизм пытается найти совпадение для завершающей кавычки. Не найдя ее в точке В, механизм отступает и пытается найти совпадение для завершающей кавычки в позиции …Japanes'e. Попытка снова оказывается неудачной. Сохраненные состояния, накопленные при поиске совпадения от A до B, последовательно проверяются в обратном порядке при перемещении от B к C. После дюжины возвратов проверяется наличие кавычки в позиции С. На этот раз совпадение найдено. Остальные непроверенные состояния попросту игнорируются, и механизм возвращает найденное совпадение:

"McDonald's" is said "makudonarudo"

#### Работа механизма при отсутствии совпадения

Что происходит при отсутствии совпадений. Рассмотрим выражение ".\*"! для которого в нашем примере не существует совпадения. Однако в процессе поиска оно довольно близко подходит к совпадению, что приводит к существенному возрастанию объема работы.

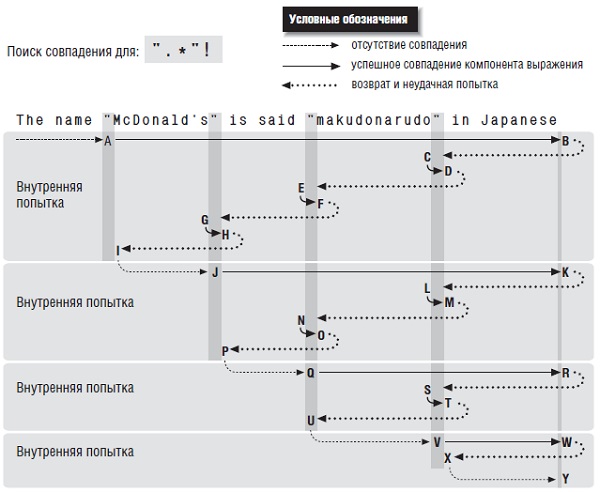


Рис. 41. Неудачный поиск совпадения

Последовательность A—С похожа на предыдущую. Отличие начинается в точке D, в которой отсутствует совпадение для завершающего восклицательного знака. И механизм символ за символом отступает к точке Е. И в F снова не находит ! И так до точки I. Механизм переходит к следующей позиции в строке. Попытки, начинающиеся в позициях J, Q и V, выглядят перспективными, но все они завершаются неудачей. Наконец, в точке Y завершается перебор всех начальных позиций в строке, поэтому вся попытка завершается неудачей. Для получения этого результата пришлось выполнить довольно большую работу.

Для сравнения давайте заменим .\* выражением [^"] Выражение не только становится более точным, но и работает эффективнее. В выражении "[^"]\*"! конструкция [^"]\* не проходит мимо завершающей кавычки, что устраняет многие попытки и последующие возвраты. Ниже показано, что происходит при неудачной попытке. Количество возвратов значительно уменьшилось. Если новый результат вам подходит, то сокращение возвратов можно считать положительным побочным эффектом.

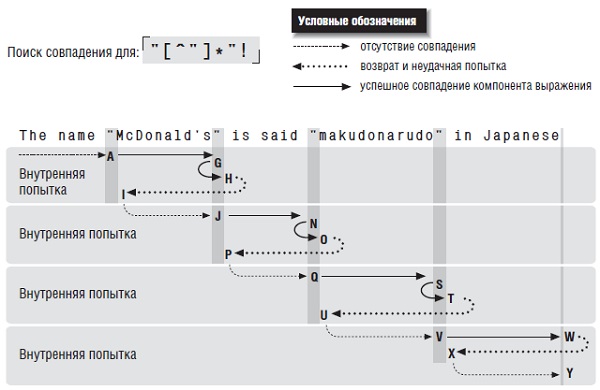


Рис. 42. Неудачный поиск совпадения для выражения "[^"]\*"!

#### Конструкция выбора может дорого обойтись

Конструкция выбора является одной из главных причин, порождающих возвраты. В качестве примера воспользуемся строкой

The name "McDonald's" is said "makudonarudo" in Japanese

… и сравним, как организуется поиск совпадений для выражений ru|v|w|x|y|z и [uvwxyz]. Символьный класс проверяется простым сравнением, поэтому [uvwxyz] «страдает» только от возвратов перехода к следующей позиции в строке (всего 34) до обнаружения совпадения:

The name "McDonald's" is said "mak**u**donarudo" in Japanese

Использование выражения u|v|w|x|y|z требует шести возвратов для каждой начальной позиции, поэтому то же совпадение будет найдено только после 204 возвратов.

Хорошее понимание возврата можно считать важнейшим аспектом эффективности.

#### Хронометраж

Хронометраж сводится к измерению времени, потраченного на выполнение некоторых действий. Мы запрашиваем системное время, выполняем нужные действия, снова получаем системное время и вычисляем разность. Для примера давайте сравним время обработки выражений ^(a|b|c|d|e|f|g)+$ и ^[a-g]+$. Автор приводит пример на языке Perl, я же использую код VBA. Сначала формирую переменную Text, включающую 16 млн. символов. Затем для двух шаблонов делаю по 5 тестов, замеряя время исполнения тестов.

Public Sub RegExpTime()

Set regEx = CreateObject("VBScript.RegExp")

' Задание длинного текста для проверки

m = 2000000

Text = String(m, "a") & String(m, "b") & String(m, "c") & String(m, "d") & String(m, "e") \_

& String(m, "f") & String(m, "g") & String(m, "h")

regEx.Global = True

' Шаблон на основе альтернатив

regEx.Pattern = "^(a|b|c|d|e|f|g|h)+$"

StartTime = Time()

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

EndTime = Time()

MsgBox Format(86400 \* (EndTime - StartTime), "0 сек"), , "Альтернативы"

' Шаблон на основе символьного класса

regEx.Pattern = "^[a-h]+$"

StartTime = Time()

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

n = regEx.Test(Text)

EndTime = Time()

MsgBox Format(86400 \* (EndTime - StartTime), "0 сек"), , "Симв. класс"

End Sub

Изображение выглядит как стол

Автоматически созданное описание

Рис. 43. Хронометраж обработки регулярных выражений на основе альтернатив и символьного класса

Видно, что символьный класс работает на порядок быстрее.

#### Стандартные оптимизации

Оптимизации обычно делятся на две категории:

* Ускорение операций. Некоторые конструкции (такие как \d+) встречаются так часто, что для них может быть предусмотрена особая обработка, превосходящая по скорости общую обработку регулярных выражений;
* Предотвращение лишних операций. Если механизм решает, что некоторая операция не нужна для достижения правильного результата или может применяться к меньшему объему текста, чем указано в исходном выражении, он может отказаться от выполнения этих операций и ускорить обработку выражения. Например, регулярное выражение, начинающееся с \A (начало текста), может совпадать только от начала строки, поэтому при отсутствии совпадения в этой позиции механизму смещения не нужно проверять другие позиции.

#### Приемы построения быстрых выражений

Хорошее понимание принципов оптимизаций, заложенных в механизм НКА поможет вам создавать более эффективные выражения. Это понимание может применяться по трем направлениям:

* Выражение записывается таким образом, чтобы при его обработке использовались существующие оптимизации. Например, замена x+ на xx\* обеспечивает возможность применения многочисленных оптимизаций, в том числе проверки обязательного символа или строки и исключения по первому символу.
* Имитация оптимизаций. Иногда вы знаете, что некая оптимизация не поддерживается программой, но можете самостоятельно имитировать оптимизацию и добиться выигрыша. Например, включение конструкции (?=t) в начало выражения this|that имитирует исключение по первому символу в системах, которые не могут самостоятельно определить, что любое совпадение должно начинаться с символа t.
* Ускоренное достижение совпадения. Например, в this|that обе альтернативы начинаются с th, если у первой альтернативы не найдется совпадение для th, то th второй альтернативы заведомо не совпадет. Чтобы время не тратилось даром, можно сформулировать то же выражение в виде th(?:is|at). В этом случае th проверяется всего один раз, а относительно затратная конструкция выбора откладывается до момента, когда она становится действительно необходимой. Кроме того, в выражении th(?:is|at) появляется литерал th, что позволяет задействовать ряд других оптимизаций.

Вы должны хорошо понимать, что эффективность и оптимизации – дело деликатное. Даже однозначно полезные изменения иногда замедляют работу программы, потому что они мешают применению других оптимизаций, о которых вы не подозревали. Преобразование выражения для достижения максимальной эффективности может усложнить выражение, затруднить его понимание и сопровождение. Выигрыш (или потеря) от конкретного изменения почти всегда сильно зависит от данных, с которыми работает выражение. Изменение, благоприятное для одного набора данных, может оказаться вредным для других данных.

#### Приемы, основанные на здравом смысле

**Избегайте повторной компиляции.** При объектно-ориентированном интерфейсе программист в полной мере управляет компиляцией. Например, если регулярное выражение должно применяться в цикле, создайте объект выражения за пределами цикла и затем многократно используйте его внутри цикла.

**Используйте несохраняющие круглые скобки.** Если вы не используете текст, совпадающий с подвыражениями в круглых скобках, используйте несохраняющие скобки (?:…). Помимо прямого выигрыша из-за отсутствия затрат на сохранение появляется побочная экономия – состояния, используемые при возврате, становятся менее сложными и поэтому быстрее восстанавливаются. Также открывается возможность для дополнительных оптимизаций, таких как исключение лишних скобок.

**Исключите из выражения лишние круглые скобки.** Присутствие круглых скобок может предотвратить применение некоторых оптимизаций. Если вам не нужно знать последний символ, совпавший с .\*, не используйте (. )\*.

**Исключите из выражения лишние символьные классы**

**Используйте привязку к началу строки.** Выражение, начинающееся с .\*, практически всегда должно начинаться с ^ или \A Если такое выражение не совпадет от начала строки, вряд ли поиск даст лучшие результаты после смещения ко второму, третьему символу и т.д. Добавление якорных метасимволов позволяет задействовать стандартную оптимизацию привязки к началу строки и сэкономить массу времени.

#### Выделение литерального текста

Многие виды оптимизации помогают механизму регулярных выражений выделить литеральный текст, который должен входить в любое успешное совпадение:

**Выделение обязательных компонентов в квантификаторах.** Использование xx\* вместо x+ подчеркивает обязательное присутствие хотя бы одного экземпляра x. Аналогично -{5,7} лучше записать в виде -----{0,2}.

**Выделение обязательных компонентов в начале конструкции выбора.** Выражение th(?:is|at) вместо (?:this|that) указывает на то, что элемент th является обязательным. Литеральный текст может выделяться и в правой части, когда общий текст следует за вариативным: (?:optim|standard)ization. Подобные приемы особенно эффективны в том случае, если выделяемая часть содержит якорные метасимволы.

**Выделение якорей.** К числу наиболее эффективных внутренних оптимизаций относятся те, в которых используются якорные метасимволы, привязывающие выражение к одному из концов целевой строки: ^, $ и \G (последний не работает в VBA). Выражения ^(?:abc|123) и ^abc|^123 эквивалентны, но оптимизация привязки к началу строки легче распознается в первом случае. Или сравните (^abc) и ^(abc). Первое выражение лишено возможности применения многих оптимизаций, так как оно «прячет» якорный элемент и вынуждает механизм регулярных выражений входить в сохраняющие круглые скобки до того, как появится возможность проверить соответствие якорному элементу. Аналогично выражения (…|…)$ лучше, чем (…$|…$).

#### Руководство процессом поиска

**Ставьте наиболее вероятную альтернативу на первое место**

**Распределите общий элемент по альтернативам.** Сравним два выражения: (?:com|edu|…|[a-z][a-z])\b и com\b|edu\b|…\b|[a-z][a-z]\b. Во втором случае метасимвол \b, следующий после конструкции выбора, распределяется по всем альтернативам. Потенциальный выигрыш от такого решения связан с тем, что совпавшая альтернатива, которая будет отвергнута из-за отсутствия совпадения для \b после конструкции выбора, во втором варианте отвергается чуть быстрее. Неудача распознается без затрат, связанных с выходом из конструкции выбора.

**Учтите, что эта оптимизация бывает рискованной.** Если «распределяемое» подвыражение содержит литеральный текст, то оптимизации не будет. Например, (?:this|that): работает быстее, чем this:|that: