Лабораторная работа №3. Регулярные выражения и объект Date

Цель работы: освоить основные принципы работы с регулярными выражениями и объектом Date.

Краткие теоретические сведения по теме лабораторной работы:

Регулярное выражение состоит из шаблона и необязательных флагов.

```
regexp = /шаблон/; // без флагов
regexp = /шаблон/gmi; // с флагами gmi (будут описаны далее)
```

Флаги влияют на поиск. В JavaScript их всего шесть: **i** – С этим флагом поиск не зависит от регистра: нет разницы между A и a. **g** – С этим флагом поиск ищет все совпадения, без него – только первое. **m** – Многострочный режим (подробно рассмотрим отдельно). **s** – Включает режим «dotall», при котором точка может соответствовать символу перевода строки **\n**. **u** – Включает полную поддержку юникода. Флаг разрешает корректную обработку суррогатных пар. **y** – Режим поиска на конкретной позиции в тексте.

Метод **str.match(regexp)** для строки **str** возвращает массив совпадений с регулярным выражением **regexp**. Если совпадений нет, то возвращается **null**.

Метод **str.replace(regexp, replacement)** заменяет совпадения с **regexp** в строке **str** на **replacement** (все, если есть флаг **g**, иначе только первое).

Метод **regexp.test(str)** проверяет, есть ли хоть одно совпадение, если да, то возвращает **true**, иначе **false**.

Часто используемые символьные классы:

\d – цифры: символ от 0 до 9. **\s** – пробельные символы: включает в себя символ пробела, табуляции **\t**, перевода строки **\n** и некоторые другие редкие пробельные символы, обозначаемые как **\v**, **\f** и **\r**. **\w** – символ «слова», а именно – буква латинского алфавита или цифра или подчёркивание _. Нелатинские буквы не являются частью класса **\w**, т. е. буква русского алфавита не подходит.

Для каждого символьного класса существует «обратный класс», обозначаемый той же буквой, но в верхнем регистре. «Обратный» означает, что он соответствует всем символам, кроме данного символьного класса.

\D – Не цифра: любой символ, кроме **\d**, например, буква. **\S** – Не пробел: любой символ, кроме \s, например, буква. **\W** – Любой символ, кроме **\w**, т.е. не буквы из латиницы, не знак подчёркивания и не цифра. В частности, русские буквы принадлежат этому классу.

Набор **[eao]** соответсвует любому из 3 символов: **'a'**, **'e'** или **'o'**. Наборы могут использоваться в регулярных выражениях вместе с обычными символами:

```
// найти [т или х], после которых идёт "оп"
alert( "Топ хоп".match(/[тх]оп/gi) ); // "топ", "хоп"
```

Также квадратные скобки могут содержать диапазоны символов. К примеру, **[a-z]** соответствует символу в диапазоне от **a** до **z**, а **[0-5]** – цифре от **0** до **5**.

Простейший квантификатор – это число в фигурных скобках: **{n}**. Он добавляется к символу (или символьному классу, или набору [...] и т.д.) и указывает, сколько раз символ повторяется.

Шаблон **\d{5}** обозначает ровно 5 цифр, он эквивалентен **\d\d\d\d\d**. Диапазон: **{3,5}**, от 3 до 5. Верхнюю границу можно не указывать. Таким образом **\d{3,}** найдёт последовательность чисел длиной 3 и более цифр.

```
alert( "Мне не 12, а 345678 лет".match(/\d{3,}/) ); // "345678"
```

Часть шаблона можно заключить в скобки (…). Это – «скобочная группа».

У такого выделения есть два эффекта: 1) Позволяет поместить часть совпадения в отдельный массив. 2) Если установить квантификатор после скобок, то он будет применяться ко всему содержимому скобки, а не к одному символу.

Без скобок шаблон **go+** означает символ **g** и идущий после него символ **o**, который повторяется один или более раз. Например, **goooo** или **gooooooooo**. Скобки группируют символы вместе. Так **(go)+** будет находить **go**, **gogo**, **gogogo** и т.д.

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo", "go"
```

Скобочные группы нумеруются слева направо. Поисковой движок запоминает содержимое, которое соответствует каждой скобочной группе, и позволяет получить его в результате.

Метод **str.match(regexp)**, если у регулярного выражения **regexp** нет флага **g**, ищет первое совпадение и возвращает его в виде массива: на позиции **0** будет всё совпадение целиком. На позиции **1** – содержимое первой скобочной группы. На позиции **2** – содержимое второй скобочной группы и т.д.

Пример – нужно найти HTML теги **<.*?>** и обработать их. Было бы удобно иметь содержимое тега в отдельной переменной. Заключим внутреннее содержимое в круглые скобки: **<(.*?)>**. Теперь получим как тег целиком **<h1>**, так и его содержимое **h1** в виде массива:

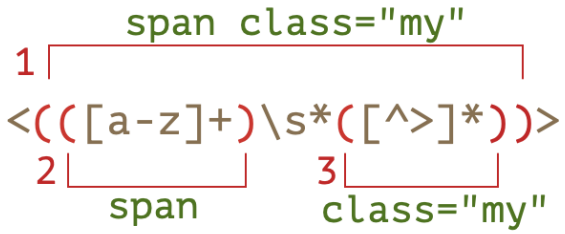
```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Скобки могут быть и вложенными. Например, при поиске тега в **** нас может интересовать: 1) Содержимое тега целиком: **span class="my"**. 2) Название тега: **span**. 3) Атрибуты тега: **class="my"**.

Заключим их в скобки в шаблоне: <(([a-z]+)\s*([^>]*))>. Вот их номера (слева направо, по открывающей скобке):




```
let str = '<span class="my">';

let regexp = /<(([a-z]+)\s*([^>]*))>/;

let result = str.match(regexp);
alert(result[0]); // <span class="my">
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

По нулевому индексу в **result** хранится полное совпадение. Затем следуют группы, нумеруемые слева направо, по открывающим скобкам. Группа, открывающая скобка которой идёт первой, получает первый индекс в результате – **result[1]**. Там находится всё содержимое тега. В **result[2]** хранится группа, образованная второй открывающей скобкой **([a-z]+)**, – имя тега, в **result[3]** будет находиться содержимое тега: **([^>]*)**.

К группе можно обратиться в шаблоне, используя \N, где N – это номер группы. Рассмотрим пример. Необходимо найти строки в кавычках – либо одинарных **'...'**, либо двойных **"..."** – оба варианта должны подходить. Как найти такие строки?

Можно попытаться добавить оба вида кавычек в квадратные скобки: ['"](.*?)['"], но в таком случае будут находиться строки со смешанными

кавычками, например "**...**' и '**...**". Это приведёт к ошибке, когда одна кавычка окажется внутри других, как в строке "**She's the one!**":

```
let str = `He said: "She's the one!".`;

let regexp = /['"](.*?)['"]/g;

// Результат – не тот, который хотелось бы
alert( str.match(regexp) ); // "She'
```

Как видно, шаблон нашёл открывающую кавычку ", нашёл текст вплоть до следующей кавычки ', после чего поиск завершился. Как этого избежать?

Для того, чтобы шаблон искал закрывающую кавычку такую же, как открывающую, обернём открывающие кавычки в скобочную группу и используем обратную ссылку на неё: (['"])(.*?)\1. Реализация:

```
let str = `He said: "She's the one!".`;

let regexp = /(['"])(.*?)\1/g;

alert( str.match(regexp) ); // "She's the one!"
```

Движок регулярных выражений находит первую кавычку из шаблона (['"]) и запоминает её содержимое. Это первая скобочная группа. В шаблоне **\1** означает «найти то же самое, что в первой скобочной группе», а именно – ту же самую кавычку для нашего случая.

Альтернация – термин в регулярных выражениях, которому в русском языке соответствует слово «ИЛИ». В регулярных выражениях она обозначается символом вертикальной черты |. Например, нам нужно найти языки: HTML, CSS, Java и JavaScript. Соответствующее регулярное выражение: html|css|java(script)?.

```
let regexp = /html|css|java(script)?/gi;

let str = "Сначала появился язык Java, затем HTML, потом JavaScript";

alert( str.match(regexp) ); // Java,HTML,JavaScript
```

Квадратные скобки позволяют выбирать между несколькими символами, например, gr[ae]y найдёт **gray** либо **grey**.

Квадратные скобки работают только с символами или наборами символов. Альтернация мощнее, она работает с любыми выражениями. Регулярное выражение A|B|C обозначает поиск одного из выражений: A, B или C.

Например: gr(a|e)y означает точно то же, что и gr[ae]y.

gra|ey означает **gra** или **ey**.

Чтобы применить альтернацию только к части шаблона, следует заключить её в скобки:

Люблю HTML|CSS найдёт **Люблю HTML** или **CSS**.

Люблю (HTML|CSS) найдёт **Люблю HTML** или **Люблю CSS**.

Для создания нового объекта **Date** нужно вызвать конструктор **new Date()** с одним из следующих аргументов: **new Date()** – без аргументов – создать объект **Date** с текущими датой и временем:

```
let now = new Date();
alert( now ); // показывает текущие дату и время
```

new Date(milliseconds) – Создать объект **Date** с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// теперь добавим 24 часа и получим 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

new Date(datestring) – Если аргумент всего один и это строка, то из неё «прочитывается» дата.

new Date(year, month, date, hours, minutes, seconds, ms) – Создать объект **Date** с заданными компонентами в местном часовом поясе. Обязательны только первые два аргумента. **year** должен состоять из четырёх цифр: значение **2013** корректно, **98** – нет. **month** начинается с **0** (январь) по **11** (декабрь). Параметр **date** здесь представляет собой день месяца. Если параметр не задан, то принимается значение **1**. Если параметры **hours**/**minutes**/**seconds**/**ms** отсутствуют, их значением становится 0.

```
new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // то же самое, так как часы и проч. равны 0
```

Существуют методы получения года, месяца и т.д. из объекта **Date**:

getFullYear() – получить год (4 цифры); **getMonth()** – получить месяц (от 0 до 11); **getDate()** – получить день месяца (от 1 до 31, что несколько противоречит названию метода); **getHours()**, **getMinutes()**, **getSeconds()**, **getMilliseconds()** – получить, соответственно, часы, минуты, секунды или миллисекунды; **getDay()** – вернуть день недели от 0 (воскресенье) до 6 (суббота).

Следующие методы позволяют установить компоненты даты и времени:

setFullYear(year, [month], [date]); **setMonth(month, [date])**; **setDate(date)**; **setHours(hour, [min], [sec], [ms])**; **setMinutes(min, [sec], [ms])**; **setSeconds(sec, [ms])**; **setMilliseconds(ms)**; **setTime(milliseconds)** (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC).

Метод **Date.parse(str)** считывает дату из строки. Формат строки должен быть следующим: **YYYY-MM-DDTHH:mm:ss.sssZ**, где:

YYYY-MM-DD – это дата: год-месяц-день.

Символ **"T"** используется в качестве разделителя.

HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.

Необязательная часть **'Z'** обозначает часовой пояс в формате +–hh:mm. Если указать просто букву **Z**, то получим UTC+0.

Возможны и более короткие варианты, например, YYYY-MM-DD или YYYY-MM, или даже YYYY.

Задания для самостоятельной работы:

Регулярные выражения

1. С использованием регулярного выражения написать функцию, проверяющую то, является ли строка из чисел номером банковской карты. Решить задачу, исходя из предположения, что номер может состоять только из 16 цифр, которые а) сгруппированы по 4 цифры и отделены друг от друга пробелом или дефисом; б) записаны в одну строку.

Пример работы программы:

func("1234-5678-9012-3456") // true

func("1234 5678 9012 3456") // true

func("1234567890123456") // true

func("12345678901234567 89") // false

func("1234-5678 9012 3456") // false

2. С использованием регулярного выражения написать функцию, которая возвращает количество гласных букв в строке.

Пример работы программы:

func("The Russian Federation") // 9

3. С использованием регулярного выражения написать функцию, которая ищет даты в строке. Даты могут быть записаны только в формате дд/мм/гггг. В качестве разделителя / могут выступать дефис (-), точка (.) или косая черта (/).

4. С использованием регулярного выражения написать функцию **func1(str)**, заменяющую все гласные буквы в строке согласно следующему правилу:

a $\rightarrow$ 1

e $\rightarrow$ 2

i $\rightarrow$ 3

o $\rightarrow$ 4

u $\rightarrow$ 5

Таким образом **func1('Hello World')** должна вернуть '**H2ll4 W4rld**'.

Создать функцию **func2(str)** для обратного преобразования. Таким образом **func2('H2ll4 W4rld')** должна вернуть '**Hello World**'.

5. С использованием регулярного выражения реализовать функцию, которая расширяет строку из латинских букв и цифр по следующим правилам:

I) Первое вхождение числового значения должно соответствовать количеству повторений каждого символа после него, пока не появится следующее числовое значение.

II) Если несколько цифровых символов идут друг за другом, используется только последний, а предыдущие игнорируются.

Таким образом функция должна работать следующим образом:

func('3a2b5c2d') // 'aaabbcccccdd'

func('3xyz') // 'xxxyyyzzz'

func('3a443c1d') // 'aaacccd'

func('1111') // пустая строка

Объект **Date**

1. Создать функцию, которая возвращает все даты, в которых 13-ый день месяца приходится на пятницу. Функция должна принимать 2 параметра, определяющие диапазон лет, внутри которого выполняется поиск “Пятниц-

тринадцатых”. Если задается только 1 параметр, функция должна вернуть даты “Пятниц-тринадцатых” только для этого года.

Пример работы программы:

func(1999, 2000) // 8/13/1999, 10/13/2000

func(2000) // 10/13/2000