Скобочные группы

Часть шаблона можно заключить в скобки `(...)`. Это называется «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.

Примеры

Разберём скобки на примерах.

Пример: gogogo

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 что `(go)+` означает `go`, `gogo`, `gogogo` и т.п.

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

Пример: домен

Сделаем что-то более сложное – регулярное выражение, которое соответствует домену сайта.

Например:

```
mail.com
users.mail.com
smith.users.mail.com
```

Как видно, домен состоит из повторяющихся слов, причём после каждого, кроме последнего, стоит точка.

На языке регулярных выражений `(\w+\.)+\w+`:

```
let regexp = /(\w+\.)+\w+/g;

alert( "site.com my.site.com".match(regexp) ); // site.
```

Поиск работает, но такому шаблону не соответствует домен с дефисом, например, `my-site.com`, так как дефис не входит в класс `\w`.

Можно исправить это, заменим `\w` на `[\w-]` везде, кроме как в конце: `([\w-]+\.)+\w+`.

Пример: email

Предыдущий пример можно расширить, создав регулярное выражение для поиска email.

Формат email: имя@домен. В качестве имени может быть любое слово, разрешены дефисы и точки. На языке регулярных выражений это `[-.\w]+`.

Итоговый шаблон:

```
let regexp = /[-.\w]+@([\w-]+\.)+[\w-]+/g;

alert("my@mail.com @ his@site.com.uk".match(regexp)); /
```

Это регулярное выражение не идеальное, но, как правило, работает и помогает исправлять опечатки. Окончательную проверку правильности email, в любом случае, можно осуществить, лишь послав на него письмо.

Содержимое скобок в match

Скобочные группы нумеруются слева направо. Поисковый 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:

1. На позиции `0` будет всё совпадение целиком.
2. На позиции `1` – содержимое первой скобочной группы.
3. На позиции `2` – содержимое второй скобочной группы.
4. ...и так далее...

Например, мы хотим найти HTML теги `<.*?>` и обработать их. Было бы удобно иметь содержимое тега (то, что внутри уголков) в отдельной переменной.

Давайте 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
```

Вложенные группы

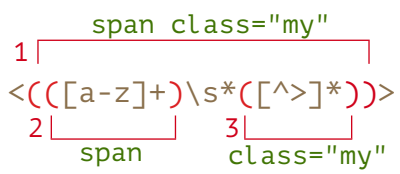
Скобки могут быть и вложенными.

Например, при поиске тега в `<span class="my">` 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):



В действии:

```
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"

```

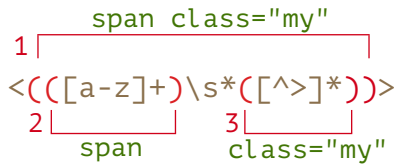
```
9  alert(result[2]); // span
   alert(result[3]); // class="my"
```


По нулевому индексу в `result` всегда идёт 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.

Затем в `result[2]` идёт группа, образованная второй открывающей скобкой `([a-z]+)` – имя тега, далее в `result[3]` будет остальное содержимое тега: `([^>]*)`.

Соответствие для каждой группы в строке:



Необязательные группы

Даже если скобочная группа необязательна (например, стоит квантификатор `(...)?`), соответствующий элемент массива `result` существует и равен `undefined`.

Например, рассмотрим регулярное выражение `a(z)?(c)?`. Оно ищет букву `"a"`, за которой идёт необязательная буква `"z"`, за которой, в свою очередь, идёт необязательная буква `"c"`.

Если применить его к строке из одной буквы `a`, то результат будет такой:

```
1  let match = 'a'.match(/a(z)?(c)?/);
2
3  alert( match.length ); // 3
4  alert( match[0] ); // a (всё совпадение)
5  alert( match[1] ); // undefined
6  alert( match[2] ); // undefined
```

Массив имеет длину `3`, но все скобочные группы пустые.

А теперь более сложная ситуация для строки `ac`:

```
1  let match = 'ac'.match(/a(z)?(c)?/)
2
3  alert( match.length ); // 3
4  alert( match[0] ); // ac (всё совпадение)
5  alert( match[1] ); // undefined, потому что для (z)? ни
6  alert( match[2] ); // c
```

Длина массива всегда равна `3`. Для группы `(z)?` ничего нет, поэтому результат: `["ac", undefined, "c"]`.

Поиск всех совпадений с группами: matchAll

> **`matchAll` является новым, может потребоваться полифил**
>
> Метод не поддерживается в старых браузерах.
>
> Может потребоваться полифил, например https://github.com/ljharb/String.prototype.matchAll.

При поиске всех совпадений (флаг `g`) метод `match` не возвращает скобочные группы.

Например, попробуем найти все теги в строке:

```
let str = '<h1> <h2>';

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

alert( tags ); // <h1>,<h2>
```

Результат – массив совпадений, но без деталей о каждом. Но на практике скобочные группы тоже часто нужны.

Для того, чтобы их получать, мы можем использовать метод `str.matchAll(regexp)`.

Он был добавлен в язык JavaScript гораздо позже чем `str.match`, как его «новая и улучшенная» версия.

Он, как и `str.match(regexp)`, ищет совпадения, но у него есть три отличия:

1. Он возвращает не массив, а перебираемый объект.
2. При поиске с флагом `g`, он возвращает каждое совпадение в виде массива со скобочными группами.
3. Если совпадений нет, он возвращает не `null`, а просто пустой перебираемый объект.

Например:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

// results - не массив, а перебираемый объект
alert(results); // [object RegExp String Iterator]

alert(results[0]); // undefined (*)

results = Array.from(results); // превращаем в массив

alert(results[0]); // <h1>,h1 (первый тег)
alert(results[1]); // <h2>,h2 (второй тег)
```

Как видите, первое отличие – очень важное, это демонстрирует строка `(*)`. Мы не можем получить совпадение как `results[0]`, так как этот объект не является псевдомассивом. Его можно превратить в настоящий массив при помощи `Array.from`. Более подробно о псевдомассивах и перебираемых объектов мы говорили в главе Перебираемые объекты.

В явном преобразовании через `Array.from` нет необходимости, если мы перебираем результаты в цикле, вот так:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

for(let result of results) {
  alert(result);
  // первый вывод: <h1>,h1
  // второй: <h2>,h2
}
```

...Или используем деструктуризацию:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Каждое совпадение, возвращаемое `matchAll`, имеет тот же вид, что и при `match` без флага `g`: это массив с дополнительными свойствами `index` (позиция совпадения) и `input` (исходный текст):

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

```
2
3  let [tag1, tag2] = results;
4
5  alert( tag1[0] ); // <h1>
6  alert( tag1[1] ); // h1
7  alert( tag1.index ); // 0
8  alert( tag1.input ); // <h1> <h2>
```

Почему результат `matchAll` – перебираемый объект, а не обычный массив?

Зачем так сделано? Причина проста – для оптимизации.

При вызове `matchAll` движок JavaScript возвращает перебираемый объект, в котором ещё нет результатов. Поиск осуществляется по мере того, как мы запрашиваем результаты, например, в цикле.

Таким образом, будет найдено ровно столько результатов, сколько нам нужно.

Например, всего в тексте может быть 100 совпадений, а в цикле после 5-го результата мы поняли, что нам их достаточно и сделали `break`. Тогда движок не будет тратить время на поиск остальных 95.

Именованные группы

Запоминать группы по номерам не очень удобно. Для простых шаблонов это допустимо, но в сложных регулярных выражениях считать скобки затруднительно. Гораздо лучше – давать скобкам имена.

Это делается добавлением `?<name>` непосредственно после открытия скобки.

Например, поищем дату в формате «день-месяц-год»:

```
1  let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-
2  let str = "2019-04-30";
3
4  let groups = str.match(dateRegexp).groups;
5
6  alert(groups.year); // 2019
7  alert(groups.month); // 04
8  alert(groups.day); // 30
```

Как вы можете видеть, группы располагаются в свойстве `groups` результата `match`.

Чтобы найти не только первую дату, используем флаг `g`.

Также нам понадобится `matchAll`, чтобы получить скобочные группы:

```
1  let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-
2
3  let str = "2019-10-30 2020-01-01";
4
5  let results = str.matchAll(dateRegexp);
6
7  for(let result of results) {
8    let {year, month, day} = result.groups;
9
10   alert(`${day}.${month}.${year}`);
11   // первый вывод: 30.10.2019
12   // второй: 01.01.2020
13 }
```

Скобочные группы при замене

Метод `str.replace(regexp, replacement)`, осуществляющий замену совпадений с `regexp` в строке `str`, позволяет использовать в строке замены содержимое скобок. Это делается при помощи обозначений вида `$n`, где `n` – номер скобочной группы.

Например:

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert( str.replace(regexp, '$2, $1') ); // Bull, John
```

Для именованных скобок ссылка будет выглядеть как `$<имя>`.

Например, заменим даты в формате «год-месяц-день» на «день.месяц.год»:

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<d

let str = "2019-10-30, 2020-01-01";

alert( str.replace(regexp, '$<day>.$<month>.$<year>') )
// 30.10.2019, 01.01.2020
```

Исключение из запоминания через ?:

Бывает так, что скобки нужны, чтобы квантификатор правильно применился, но мы не хотим, чтобы их содержимое было выделено в результате.

Скобочную группу можно исключить из запоминаемых и нумеруемых, добавив в её начало `?:`.

Например, если мы хотим найти `(go)+`, но не хотим иметь в массиве-результате отдельным элементом содержимое скобок (`go`), то можем написать `(?:go)+`.

В примере ниже мы получим только имя `John` как отдельный элемент совпадения:

```
let str = "Gogogo John!";

// ?: исключает go из запоминания
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert( result[0] ); // Gogogo John (полное совпадение)
alert( result[1] ); // John
alert( result.length ); // 2 (больше в массиве элементо
```

Как видно, содержимое скобок `(?:go)` не стало отдельным элементом массива `result`.

Итого

Круглые скобки группируют вместе часть регулярного выражения, так что квантификатор применяется к ним в целом.

Скобочные группы нумеруются слева направо. Также им можно дать имя с помощью `(?<name>...)`.

Часть совпадения, соответствующую скобочной группе, мы можем получить в результатах поиска.

- Метод `str.match` возвращает скобочные группы только без флага `g`.
- Метод `str.matchAll` возвращает скобочные группы всегда.

Если скобка не имеет имени, то содержимое группы будет по своему номеру в массиве-результате, если имеет, то также в свойстве `groups`.

Содержимое скобочной группы можно также использовать при замене `str.replace(regexp, replacement)`: по номеру `$n` или по имени `$<имя>`.

Можно исключить скобочную группу из запоминания, добавив в её начало `?:`. Это используется, если необходимо применить квантификатор ко всей группе, но не запоминать их содержимое в отдельном элементе массива-результата. Также мы не можем ссылаться на такие скобки в строке замены.

Задачи

Найти цвет в формате #abc или #abcdef

Напишите регулярное выражение, которое соответствует цветам в формате `#abc` или `#abcdef`. То есть: `#` и за ним 3 или 6 шестнадцатеричных цифр.

Пример использования:

```
let regexp = /ваш шаблон/g;

let str = "color: #3f3; background-color: #AA00ef; and:

alert( str.match(regexp) ); // #3f3 #AA00ef
```

P.S. Это должно быть ровно 3 или 6 шестнадцатеричных цифр. При этом значения с 4-мя цифрами типа `#abcd` не должны совпадать в результат.

решение

Найти все числа

Напишите регулярное выражение, которое ищет любые десятичные числа, включая целочисленные, с плавающей точкой и отрицательные.

Пример использования:

```
let regexp = /ваш шаблон/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(regexp) ); // -1.5, 0, 2, -123.4
```

решение

Разобрать выражение

Арифметическое выражение включает два числа и оператор между ними. Например:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

Оператором может быть: `"+"`, `"-"`, `"*"` или `"/"`.

В выражении могут быть пробелы в начале, в конце или между частями выражения.

Создайте функцию `parse(expr)`, которая принимает выражение и возвращает массив из трёх элементов:

1. Первое число.
2. Оператор.
3. Второе число.

Например:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

решение

Проверьте MAC-адрес

MAC-адрес сетевого интерфейса состоит из 6-ти двузначных шестнадцатеричных чисел, разделённых двоеточиями.

Например: `'01:32:54:67:89:AB'`.

Напишите регулярное выражение, которое проверит, является ли строка MAC-адресом.

Использование:

```
let regexp = /ваш regexp/;

alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (нет дво

alert( regexp.test('01:32:54:67:89') ); // false (5 чис

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ
```

решение