Tomasz **Sochacki**

JavaScript

WYRAŻENIA REGULARNE DLA PROGRAMISTÓW

Helion

Opieka redakcyjna: Ewelina Burska
Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wyrjsp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-3892-0

Printed in Poland.

Spis treści

Rozdział 5.

Przyszłość wyrażeń regularnych w JavaScripcie

Wyrażenia regularne w JavaScripcie są obsługiwane w znacznie mniejszym zakresie niż w innych językach, np. w Javie. W ostatnim czasie komitet normalizacyjny TC39 zaczyna jednak coraz bardziej rozszerzać możliwości regexp, a w niedługiej przyszłości planowane są kolejne zmiany.

Ten rozdział został poświęcony omówieniu planowanych nowych funkcjonalności wyrażeń regularnych na podstawie materiałów publikowanych przez komitet normalizacyjny. Można z nich korzystać po włączeniu w przeglądarce Chrome flagi *javascript-harmony* odpowiedzialnej za aktywowanie eksperymentalnych elementów silnika JavaScript:

```
chrome://flags/#enable-javascript-harmony
```

Niniejszy rozdział opisuje planowane zmiany w odniesieniu do ich eksperymentalnej implementacji w przeglądarce Chrome. Omawiane w tym rozdziale elementy są obecnie (stan na listopad 2017 r.) w stage 3, co oznacza, że z bardzo dużym prawdopodobieństwem znajdą się w kolejnym wydaniu standardu ECMAScript 2018.

Jak wygląda proces tworzenia standardów ECMAScript

Na początku należy zaznaczyć, że JavaScript nie jest określeniem tożsamym z ECMAScript. Standard ES jest punktem wyjścia do tworzenia nowych wersji silników JavaScript. Przykładem może być chociażby kwestia obiektów globalnych w JavaScripcie. W silnikach obsługiwanych przez przeglądarki internetowe dostępny jest np. obiekt `window`, którego nie ma jednak w node.js, działającym po stronie serwerowej.

Rozwojem standardu ECMAScript zajmuje się komitet normalizacyjny TC39. W jego skład wchodzą głównie przedstawiciele firm tworzących przeglądarki internetowe. W regularnych odstępach czasu spotykają się i dyskutują o nowych propozycjach zmian w standardzie, które później będą implementować w poszczególnych przeglądarkach.

Cały proces od pomysłu do wdrożenia rozwiązania w standardzie ECMAScript obejmuje pięć etapów.

- **Stage 0** — w tej fazie dyskutuje się o pomyśle bez jakichkolwiek analiz jego implementacji. Najczęściej jeden z członków komitetu zgłasza jakąś propozycję i jeśli zostanie ona przyjęta, to wchodzi do kolejnego etapu.
- **Stage 1** — po wstępnej akceptacji pomysłu w etapie 1. jeden z członków komitetu przygotowuje dokładny opis proponowanej funkcjonalności wraz z przykładowymi algorytmami. Jeśli w tym momencie rozwiązanie zostanie przyjęte, to przejście do etapu 2. jest równoważne ze wstępną zgodą na włączenie danego pomysłu do docelowego standardu ECMAScript. Opisywane rozwiązania mogą się jednak jeszcze zmieniać na kolejnych etapach.
- **Stage 2** — na tym etapie proponowane rozwiązanie musi być już kompletne, to znaczy opisane językiem zgodnym ze stosowanym w standardzie ECMAScript, a wszelkie dalsze zmiany mogą być jedynie zmianami przyrostowymi lub poprawką błędów zgłoszonych na etapie testowania. Aby przejść do etapu 3., muszą zostać opracowane co najmniej dwie eksperymentalne implementacje.
- **Stage 3** — w tym momencie cała proponowana funkcjonalność jest już kompletna i w pełni opisana w standardzie. Ponadto są już gotowe eksperymentalne implementacje. Jest to etap, w którym ponownie dyskutuje się nad daną funkcjonalnością i podejmuje się ostateczną decyzję co do wpisania jej do standardu ECMAScript.
- **Stage 4** — ostatni etap, w którym gotowe już eksperymentalne implementacje poddaje się różnym testom. Jeśli wszystkie testy przechodzą prawidłowo i pomysł ma już pełną dokumentację, to znajdzie się on w najbliższym wydaniu nowego standardu ECMAScript. Rzeczywista implementacja w przeglądarkach i innych środowiskach może się jednak istotnie opóźnić w zależności od stopnia skomplikowania danego rozwiązania. Warto jednak śledzić rozwój transpilatora Babel, który często pozwala korzystać z nowych funkcjonalności, gdy są we wczesnych fazach, długo przed ich wdrożeniem do standardu ES.

Nowa flaga dotAll

Znak kropki często jest definiowany jako dopasowanie dowolnego znaku. Jak może pamiętasz z rozdziału 2., definicja ta nie jest do końca poprawna. Po pierwsze nie dopasowuje ona sekwencji oznaczającej nowy wiersz, a po drugie działa tylko w podstawowym zakresie znaków Unicode, czyli do punktu kodowego `U+FFFF`.

```
/^.$/.test( 'a' );          // true
/^.$/.test( '\n' );         // false
/^.$/.test( '\u{1F31B}' ); // false
```

Zwróć uwagę, że w powyższych przykładach zastosowaliśmy granice początku i końca ciągu, co skutkuje próbą dopasowania dokładnie jednego dowolnego znaku. Bez tych granic ostatnie wywołanie metody `test` zakończyłoby się powodzeniem, gdyż jako dowolny znak zostałby dopasowany pierwszy surogat znaku (jak zwykle użyliśmy naszego symbolu półksiężyca).

Pytanie jednak, jak stworzyć wzorzec, który faktycznie dopasuje dowolny znak z pełnego zakresu Unicode. Po części problem ten udało nam się już rozwiązać, stosując sztuczkę z odpowiednim zakresem znaków:

```
/^[\s\S]$/.test( '\n' );          // true

/^[\s\S]$/.test( '\u{1F31B}' );  // false
/^[\s\S]$/u.test( '\u{1F31B}' ); // true po włączeniu flagi unicode
```

Tworząc zakres znaków `[\s\S]`, w rzeczywistości próbujemy dopasować dowolny znak, ale żeby uwzględnić również znaki spoza podstawowej platformy wielojęzycznej BMP, konieczne jest aktywowanie flagi *unicode*.

Metoda ta jest jednak mało czytelna, szczególnie dla osób nieznających dokładnie zasad tworzenia klas znaków. Ponadto, jeśli we wzorcu będziemy potrzebowali kilka razy dopasować dowolny znak, to powyższy zapis istotnie skomplikuje całe wyrażenie regularne.

Z pomocą przychodzi nowa flaga *dotAll* (s). Po jej włączeniu znak kropki dopasowuje wszystkie znaki Unicode do `U+FFFF`, łącznie ze znakiem nowego wiersza. Jeśli chcemy, aby kropka dopasowała znaki spoza zakresu BMP, to musimy dodatkowo włączyć również flagę *unicode*.

```
/^.$/s.test( '\n' );          // true

/^.$/s.test( '\u{1F31B}' );  // false
/^.$/su.test( '\u{1F31B}' ); // true
```

Należy jednak pamiętać, że omawiane powyżej specjalne znaczenie kropki jest ważne tylko wtedy, gdy znak ten znajduje się poza nawiasami kwadratowymi, określającymi zakres znaków. Wewnątrz zakresu kropka jest dopasowywana w sposób dosłowny.

Po wprowadzeniu flagi *dotAll* zmienia się również zalecana kolejność definiowania flag w wyrażeniach regularnych, która jest następnie stosowana przy odczycie właściwości `flags` dla obiektu `RegExp`:

```
"gimsuy"
```

Nazwane grupy przechwytujące

Dotychczas, aby odwołać się do przechwyconych grup, musieliśmy znać ich dokładne numery, co przy bardziej rozbudowanych wzorcach wymagało dłuższej analizy. Szczególnie ostrożnie należy tworzyć zagnieżdżone grupy przechwytujące, co omawialiśmy dokładnie w rozdziale 2.

Przeanalizujmy prosty przykład dopasowania daty zgodnej ze standardem najczęściej używanym w Polsce, czyli dzień-miesiąc-rok:

```
/(\d{2})-(\d{2})-(\d{4})/
```

Jeśli teraz chcielibyśmy odnieść się do poszczególnych grup, to musielibyśmy dokładnie pamiętać kolejność, w jakiej zostały przechwycone. Problem ten rozwiązują nazwane grupy przechwytujące, które pozwalają na inny zapis powyższego wyrażenia regularnego:

```
/(?<day>\d{2})-(?<month>\d{2})-(?<year>\d{4})/
```

Zapis ten na pewno jest dłuższy, ale jednocześnie zwiększa czytelność wzorca i ułatwia jego późniejszą analizę. Nie musimy w tym momencie tworzyć odpowiednich komentarzy informujących, czym są poszczególne grupy, wystarczy nadać im odpowiednio precyzyjne nazwy. Zauważ, że w tym wypadku grupę definiuje się w nieco odmienny sposób:

```
(?<nazwa>)
```

Pomiędzy znacznikami < i > należy określić nazwę grupy, stosując w tym celu znaki, które są dopuszczone podczas definiowania właściwości obiektów w JavaScripcie. Nie należy tutaj stosować m.in. znaków spacji.

Metoda match i exec

Przeanalizujmy stworzone przed chwilą wyrażenie regularne użyte wraz z metodami `String.prototype.match` oraz `RegExp.prototype.exec`:

```
const reg = /(?<day>\d{2})-(?<month>\d{2})-(?<year>\d{4})/

const match = '12-05-2017'.match( reg );
match;
// ["12-05-2017", "12", "05", "2017", index: 0, input: "12-05-2017", groups: {...}]

const exec = reg.exec( '12-05-2017' );
exec;
// ["12-05-2017", "12", "05", "2017", index: 0, input: "12-05-2017", groups: {...}]
```

Zwróć uwagę, że w obu przypadkach korzystamy z wyrażenia regularnego, w którym nie ma aktywnej flagi *global*. Jeśli zostałaby ona włączona, to metoda `match` zwróciłaby tablicę z jednym elementem, którym byłby ciąg znakowy reprezentujący znalezione dopasowanie (w tym wypadku byłby on identyczny z ciągiem wejściowym).

Tablice zwrotne mają elementy identyczne jak omawiane w rozdziale 3. Zauważ jednak, że tablica jako obiekt posiada dodatkową właściwość `groups`, w której zapisane są wszystkie dopasowane grupy nazwane. Do grup tych możemy więc odnieść się na dwa sposoby:

```
match[1];          // "12"
match.groups.day; // "12"

exec[1];           // "12"
exec.groups.day;  // "12"
```

Nie należy więc obawiać się, że wprowadzenie grup nazwanych może zaburzyć działanie istniejących aplikacji. W języku JavaScript jednym z ważniejszych problemów jest konieczność zachowania zgodności wstecz, w związku z czym zdecydowano się na dodanie kolejnego elementu, bez usuwania dotychczasowych.

Tworząc wyrażenia regularne z grupami przechwytującymi, należy zdecydować się, czy w danym przypadku stosujemy grupy nazwane, a jeśli tak, to używać ich dla wszystkich grup. Zwrotny obiekt `groups` zawiera bowiem tylko grupy, którym nadano nazwę:

```
const reg = /(?<digit>\d)([A-Z])/;
const match = '5A'.match( reg );

match.groups; // {digit: "5"}
// ale:
match[1]; // "5"
match[2]; // "A"
```

Stosując grupę nazwaną, możemy mieć jednak pewność, że obiekt `groups` zawsze będzie zawierał taką właściwość, nawet jeśli grupie zostanie przypisany kwantyfikator wystąpienia warunkowego (przyjrzyj się dokładnie położeniu kwantyfikatora `?`):

```
const reg = /(?<digit>\d?)([A-Z])/;
const match = 'A'.match( reg );
match.groups;  // {digit: ""}

const reg1 = /(?<digit>\d)?([A-Z])/;
const match1 = 'A'.match( reg1 );
match1.groups; // {digit: undefined}
```

Bardzo podobnie zachowuje się metoda `RegExp.prototype.exec`:

```
const reg = /(?<digit>\d?)([A-Z])/;
const exec = reg.exec( 'A' );
exec.groups;  // {digit: ""}

const reg1 = /(?<digit>\d)?([A-Z])/;
const exec1 = reg1.exec( 'A' );
exec1.groups; // {digit: undefined}
```

Zmodyfikowana metoda String.prototype.replace

Podobnie jak metody `match` i `exec`, również i metoda `String.prototype.replace` może dysponować dodatkowym obiektem `groups`, jeśli w wyrażeniu regularnym znajdą się grupy nazwane. Obiekt ten jest dostępny w funkcji zwrotnej (użytej jako drugi argument metody `replace`) jako ostatni jej parametr:

```
const reg = /(?<month>\d{2})-(?<year>\d{4})/

'12-2017'.replace( reg, ( match, p1, p2, offset, string, groups ) => {
    // Odwracamy kolejność:
    return `${groups.year}-${groups.month}`;
} ); // "2017-12"
```

Można oczywiście zastosować również zapis wykorzystujący destrukturyzację obiektu `groups`:

```
'12-2017'.replace( reg, ( match, p1, p2, offset, string, { month, year } ) => {
    // Odwracamy kolejność:
    return `${year}-${month}`;
} ); // "2017-12"
```

Zauważ, że w obu przypadkach dostępne są również dopasowania do grup przechwytujących zapisane w zmiennych `p1` i `p2`. Jest to konieczne dla zachowania zgodności wstecz. Wewnątrz funkcji zwrotnej warto jednak trzymać się jednej konwencji, co zapewni czytelność kodu, łatwość jego utrzymania i wprowadzania zmian. Należy również pamiętać, że obiekt z grupami nazwanymi dostępny jest za zmiennymi `offset` oraz `string`.

Odwołania wsteczne wewnątrz wyrażenia regularnego

Tworząc nazwane grupy przechwytujące, można się do nich odwoływać również w samych wyrażeniach regularnych. W tym celu należy zastosować składnię z sekwencją `\k`, a po niej należy wskazać grupę, do której chcemy się odnieść:

```
(?<nazwa>) ... \k<nazwa>
```

Przeanalizujmy poniższy przykład, w którym chcemy sprawdzić, czy ciąg znakowy zawiera dowolną cyfrę, następnie co najmniej jedną małą literę i ponownie tę samą cyfrę. Co prawda w tak prostym przykładzie w praktyce nie ma sensu stosowanie grup nazwanych, ale w bardziej złożonych wzorcach jest to dobra praktyka, ułatwiająca analizę wyrażenia regularnego.

```
const reg = /(?<digit>\d)[a-z]+\k<digit>/

reg.test( '5abc5' ); // true
reg.test( '5abc3' ); // false
```

Wszystkie dopasowane grupy nadal są dostępne również poprzez odwołania numeryczne, czyli sekwencję `\N`. Grupom nadawane są kolejne numery zgodnie z występowaniem kolejnych nawiasów grupujących od lewej do prawej. Oczywiście pomijane są przy tym grupy nieprzechwytujące, ale każda grupa nazwana zawsze stanowi grupę przechwytującą, zapamiętaną pod wskazaną nazwą oraz kolejnym numerem.

```
const reg = /(?<digit>\d)\1\k<digit>/

reg.test( '555' ); // true
reg.test( '543' ); // false
```

Stosując nazwane grupy przechwytujące, trzeba pamiętać, że istotna jest wielkość znaków użytych do opisania grupy:

```
const reg = /(?<digit>\d)\1\k<DIGIT>/
// SyntaxError: Invalid named capture referenced
```

Cofnięcia pozytywne i negatywne

Do tej pory mogliśmy w JavaScripcie korzystać z wyprzedzeń zarówno pozytywnych, jak i negatywnych, ale nie można było w sposób bezpośredni używać cofnięć, które od dawna dostępne są w wielu innych językach programowania.

Teraz ma się to zmienić i mamy otrzymać nowy element składniowy, tzw. *lookbehind assertions*. Definiowanie cofnięć wygląda następująco:

```
/(?<=...)/ <- dla cofnięć pozytywnych
/(?<!...)/ <- dla cofnięć negatywnych
```

Co istotne, podobnie jak jest w przypadku wyprzedzeń, cofnięcia także nie są zwracane w dopasowanym ciągu znakowym. Daje nam to duże możliwości precyzyjnej analizy ciągów i zwracania od razu tylko dokładnie wyselekcjonowanych fragmentów, ograniczając tym samym konieczność ich dalszego przetwarzania (np. metodą `String.prototype.replace`).

Przeanalizujmy prosty przykład cofnięcia pozytywnego i negatywnego:

```
// $ === \u0024
// € === \u20AC

const price = '$100 €200';

price.match( /(?<=\u0024)\d{3}/g ); // ["100"]
price.match( /(?<=\u20AC)\d{3}/g ); // ["200"]

price.match( /(?<!\u0024)\d{3}/g ); // ["200"]
price.match( /(?<!\u20AC)\d{3}/g ); // ["100"]
```

W powyższych przykładach próbujemy dopasować liczbę trzycyfrową, przed którą występuje lub nie występuje odpowiednio symbol dolara lub euro.

Jako cofnięcie, podobnie jak przy wyprzedzeniach, można wskazać dowolne wyrażenie regularne, a więc również wyrażenie zawierające grupy przechwytujące:

```
const price = '$100 €200';

// Grupa wewnątrz cofnięcia:
/(?<=(\u20AC))\d{3}/.exec( price );
```

```
// ["200", "€", index: 6, input: "$100 €200"]

// Dodatkowa grupa poza cofnięciem:
/(?<=(\u20AC))(\d{3})/.exec( price );
// ["200", "€", "200", index: 6, input: "$100 €200"]
```

Oczywiście również w tym przypadku można stosować grupy nazwane:

```
const p = /(?<=(?<currency>\u20AC))(?<price>\d{3})/.exec( price );

p.groups.currency; // "€"
p.groups.price;    // "200"
```

Grupy przechwytujące stworzone wewnątrz cofnięcia są również dostępne poza nim w pozostałej części wyrażenia regularnego. Pozwala to na przykład stworzyć poniższy wzorzec, który z wejściowego ciągu zwraca liczby, które na początku i na końcu ograniczone są taką samą wielką literą alfabetu łacińskiego, lecz bez zwracania tych granicznych znaków:

```
'X123X A456A B234C'.match( /(?<=([A-Z]))\d+(?=\1)/g ); // ["123", "456"]
```

Pewien problem pojawia się jednak, jeśli wewnątrz cofnięcia chcemy stosować odwołania do przechwyconych grup. Przeanalizujmy dwa poniższe przykłady:

```
'X_Xaaa Y_Ybbb X_Zccc'.match( /(?<=([A-Z])_\1)[a-z]+/g );
// null

'X_Xaaa Y_Ybbb X_Zccc'.match( /(?<=\1_([A-Z]))[a-z]+/g );
// ["aaa", "bbb"]
```

Naszym celem było stworzenia wzorca zwracającego ciąg małych liter (`aaa`, `bbb`), przed którymi znajduje się ciąg składający się z wielkiej litery, znaku podkreślenia i ponownie tej samej wielkiej litery (np. `X_X`, ale nie `X_Z`).

Prawdopodobnie w pierwszym przypadku oczekiwaliśmy, że nasz wzorzec zadziała prawidłowo, gdyż wewnątrz cofnięcia stworzyliśmy podwzorzec przechwytujący wielką literę, a następnie sprawdzający, czy występuje ona ponownie. Otrzymaliśmy jednak zwrotnie wartość `null` wskazującą na brak możliwego do znalezienia dopasowania.

Otóż błąd tkwi w tym, że wewnątrz cofnięć do przechwytywanych grup należy odwoływać się w odwróconej kolejności, czyli odwołanie do grupy (`\1`) musi znaleźć się przed (z lewej strony) wyrażeniem definiującym tę grupę.

Z tego powodu z bardzo dużą ostrożnością trzeba stosować odwołania wsteczne wewnątrz cofnięć, szczególnie jeśli tworzymy bardziej skomplikowany wzorzec. Wielu programistów niemających doświadczenia z cofnięciami może źle zinterpretować sens wyrażenia regularnego. Jako ćwiczenie własne zostawiam Cię, Czytelniku, z ostatnim przykładem łączącym w sobie cofnięcia, wyprzedzenia oraz grupowanie z przechwytem:

```
'X_XaaaX Y_YbbbY'.match( /(?<=\1_(?<START>[A-Z]))[a-z]+(?=\k<START>)/g );
// ["aaa", "bbb"]
```

Skorowidz

E

F

G

I

K

M

W

Z

JavaScript

Świadomie stosuj wzorce — to ułatwi Ci pracę!

Wyrażenia regularne w JavaScripcie, podobnie jak we wszystkich innych językach programowania, wymyślono po to, żeby ułatwić programistom pracę. Są niezastąpione chociażby przy walidacji informacji wprowadzanych przez użytkowników, przeszukiwaniu zbiorów danych czy automatyzacji wielu zadań. Niestety, duża część programistów — i to nie tylko początkujących — omija wyrażenia regularne szerokim łukiem w przekonaniu, że zbyt trudno je zrozumieć, a pomyłka może sporo kosztować. Z tej książki dowiesz się, jak należy czytać oraz samodzielnie konstruować i testować wyrażenia regularne, żeby służyły Twoim celom. Szybko przekonasz się, jak wygodne jest to rozwiązanie.

Niezależnie od tego, czy jesteś nowicjuszem, czy programujesz od lat, znajdziesz tu coś dla siebie. Poznasz metody i zasady pracy z regexami, nauczysz się testować wzorce i dopasowywać je do swoich zamierzeń. Dogłębna analiza składni wyrażeń regularnych oraz ich zastosowania została tu poszerzona o omówienie potencjalnych problemów i częstych błędów popełnianych przez osoby stawiające pierwsze kroki w tym obszarze. Przeczytaj, wypróbuj wyrażenia regularne i zacznij używać ich na co dzień, a Twoja praca w JavaScripcie stanie się znacznie bardziej efektywna!

- Na początek nieco teorii
- Składnia wyrażeń regularnych w JavaScripcie
- Praca z wyrażeniami regularnymi w JavaScripcie
- Przykładowe wyrażenia regularne z omówieniem
- Przyszłość wyrażeń regularnych w JavaScripcie

Regexy — używaj ich regularnie!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-3892-0

9 788328 338920

Cena: 49,00 zł