1 Construire des expressions régulières en JavaScript

Il existe deux façons de construire des d'expressions régulières en JavaScript :

☐ La première d'entre elles, vue dans les exemples, consiste à utiliser une variable littérale RegExp contenant le motif. Avec cette méthode, il est possible d'utiliser les étiquettes g, i et m, en les plaçant directement à la fin de la variable RegExp. Le tableau ci-dessous précise comment ces étiquettes sont interprétées.

Etiquette	Interprétation
`g`	Global / glouton – correspondance effectuée plusieurs fois
`i`	Insensible – la casse des caractères est ignorée
`m`	Multi-lignes - ^ et $ permettent d'identifier les caractères de fin de ligne

```
// Expression régulière pour identifier une chaîne de caractères JavaScript.
var mon_regexp = /"(?:\\.|[^\\\"])*"/g;
```

☐ La deuxième méthode repose sur l'utilisation du constructeur RegExp. Ce constructeur prend en entrée deux paramètres, puis les compile en un objet RegExp. Le premier paramètre est une chaîne de caractères. Il convient d'être prudent lors de la construction de la chaîne de caractères, car le caractère « \ » a une signification différente au sein d'un objet RegExp et au sein d'une variable littérale. Il est généralement nécessaire de doubler les backslashes et déspécifier les guillemets. Le deuxième paramètre est une chaîne de caractères précisant l'étiquette à utiliser. Cette méthode est utile lorsque l'expression régulière doit être générée à l'exécution, en utilisant des ressources qui ne sont pas disponibles pour le programmeur. Les propriétés d'un objet RegExp sont données dans le tableau ci-dessous.

Propriété objet RegExp	Utilisation
`global`	Vrai si l'étiquette 'g' a été utilisée.
`ignoreCase`	Vrai si l'étiquette 'i' a été utilisée.
`multiline`	Vrai si l'étiquette 'm' a été utilisée.
`lastIndex`	Valeur de l'index à partir du quel commencer la prochaine correspondance exec. Par défaut, il vaut 0.
`source`	Le texte source de l'expression régulière.

```
// Expression régulière pour identifier une chaîne de caractères JavaScript.
var mon_regexp = new RegExp("\"(?:\\.|[^\\\\\\\"])*\"", 'g');
```

2 Les méthodes

Les expressions régulières sont utilisées avec des méthodes pour chercher, remplacer et extraire des informations à partir de chaînes de caractères. Les méthodes suivantes s'appliquent en JavaScript :

Nom méthode	Description
`regexp.exec(string)`	Ceci est la méthode la plus puissante (mais aussi la plus lente) : si **`regexp`** et **`string`** correspondent, la méthode retourne un tableau. L'élément 0 du tableau contient la sous-chaîne qui correspond à regexp. L'élément 1 représente le texte capturé par le 1er groupe, etc. S'il n'y a pas de correspondance, cela retourne **`null`**.
`regexp.test(string)`	C'est la méthode la plus simple (et la plus rapide). Si **`regexp`** correspond à **`string`**, cela retourne vrai ; sinon faux.

En règle générale, les meilleures expressions régulières sont courtes et simples. Celles qui sont plus compliquées ont plus de chances d'avoir des problèmes de portabilité. Les expressions régulières imbriquées connaissent des performances limitées dans certaines implémentations. La simplicité est la meilleure stratégie.

2.1 Exemple 1 – identification d'URLs (méthode exec)

Soit l'expression régulière suivante qui identifie les URLs :

```
var parse_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)
                (?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/;
```

Nous allons utiliser la méthode exec de cette expression régulière avec la variable suivante :

```
var url = "http://www.ora.com:80/goodparts?q#fragment";
```

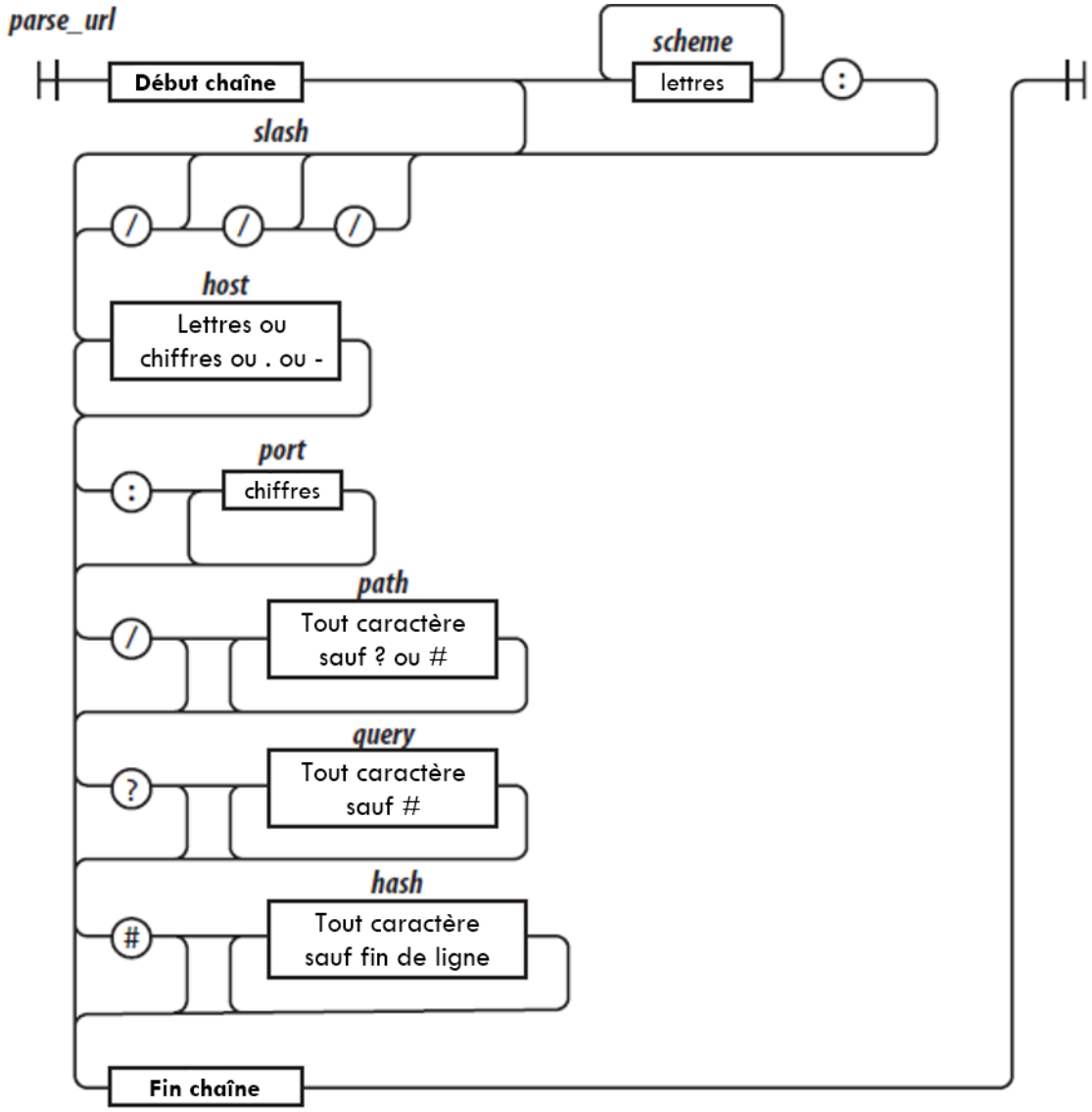
Si la variable **`url`** correspond à l'expression régulière définie dans **`parse_url`**, alors cela retournera un tableau contenant des éléments extraits à partir de la chaîne de caractères initiale (**`url`**).
Considérons le code suivant :

```
var url = "http://www.ora.com:80/goodparts?q#fragment";
var result = parse_url.exec(url);
var names = ['url','scheme','slash','host','port','path','query','hash'];
var blanks = ' ';
var i;
for (i = 0; i < names.length; i += 1) {
      document.writeln(names[i] + ':' +
      blanks.substring(names[i].length), result[i]);
}
```

Ce code produit le résultat suivant:

```
url: http://www.ora.com:80/goodparts?q#fragment
scheme: http
slash: //
host: www.ora.com
port: 80
path: goodparts
query: q
hash: fragment
```

La figure suivante illustre les traitements effectués par le motif défini dans **`parse_url`** :



Afin de mieux comprendre les résultats, étudions le motif **`parse_url`** :

Partie étudiée	Explication
^	Début de chaîne
`(?:([A-Za-z]+):)`	Permet d'identifier un nom de schéma, seulement si suivi de « : ». (?:…) précise qu'il s'agit d'un regroupement non-mémorisant. « ? » en suffixe précise que la présence de ce groupe est optionnelle. Par contre ([A-Za-z]+) est un groupement mémorisant et le texte identifié par ce groupe sera contenu dans result[1].
`(\/{0,3})`	C'est le 2e groupement mémorisant. Il cherche à identifier le caractère « / » (déspécifié avec « \ ») répété 0, 1, 2 ou 3 fois.
`([0-9.\-A-Za-z]+)`	Ce 3e groupement mémorisant permet d'identifier un nom d'hôte, qui est composé d'un ou plusieurs chiffres, lettres, le caractère « . » ou « - » (ici déspécifié).
`(?::(\d+))?`	Ce 4e groupement mémorisant identifie (optionnellementà le numéro de port, cest-à-dire une séquence d'un ou plusieurs chiffres précédés par le caractère « : ».
`(?:\/([^?#]*))?`	C'est un autre groupement optionnel, qui débute avec le caractère « / » et qui contient le groupement mémorisant 5. La classe de caractères [^ ?#] correspond à tous les caractères[1] sauf « ? » et « # ». Ces caractères sont indentifiés 0 ou plusieurs fois (caractère *).

[1] Cette classe contient aussi des caractères tels que la fin de ligne ou des caractères de contrôle qui ne devraient pas être identifiés ici. En pratique, il faudrait mieux spécifier cette classe, afin d'éviter les erreurs. En effet, les expressions régulières mal écrites sont une source fréquente d'erreurs de sécurité.

`(?:\?([^#]*))?`	Il s'agit d'un autre groupement non-mémorisant débutant avec le caractère « ? » et contenant la mémorisation 6 (contient 0 ou plusieurs caractères qui ne sont pas des « # »).
`(?:#(.*))?`	Dernier groupement optionnel, qui débute avec un « # » et qui contient le groupement mémorisant 7 (tout caractère sauf la fin de ligne)
`$`	Fin de la chaîne

2.2 Exemple 2 – identification de nombres (méthode test)

Considérons l'exemple de l'expression régulière suivante qui permet d'identifier des nombres sous différentes formes : entiers signés ou pas, avec partie décimale ou sans, avec exposant ou pas.

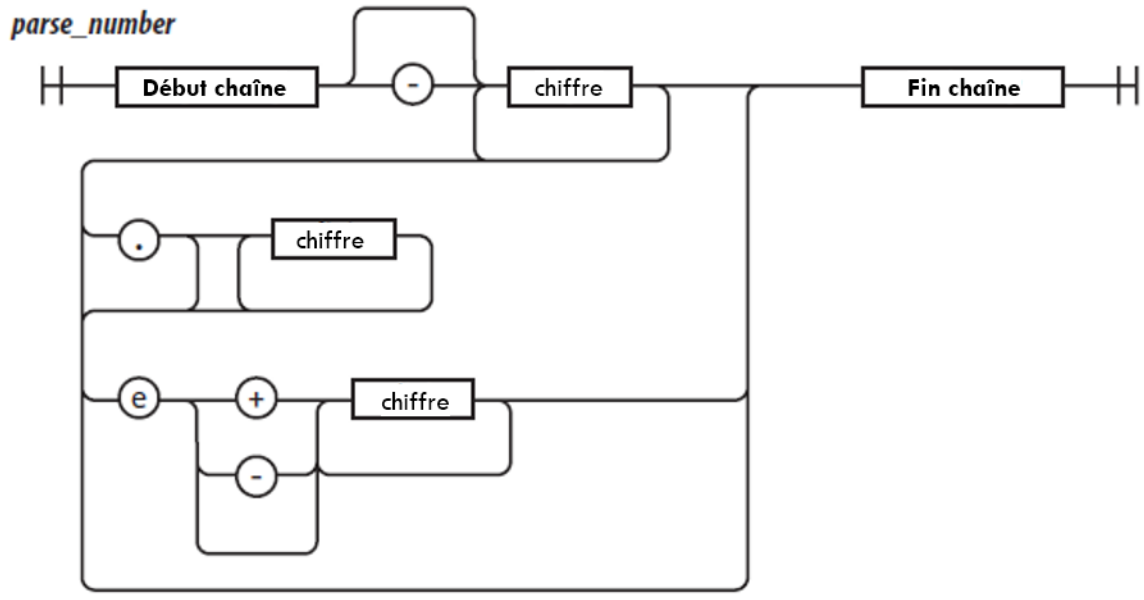
Soit le code suivant :

```
var parse_number = /^-?\d+(?:\.\d*)?(?:e[+\-]?\d+)?$/i;
var test = function (num) {
      document.writeln(parse_number.test(num));
};
```

Ce code produit le résultat suivant:

```
test('1'); // true
test('number'); // false
test('98.6'); // true
test('132.21.86.100'); // false
test('123.45E-67'); // true
test('123.45D-67'); // false
```

La figure suivante illustre les traitements effectués par le motif défini dans **`parse_number`** :



Afin de mieux comprendre les résultats, étudions le motif **`parse_number`** :

Partie étudiée	Explication
`/^...$/i`	Nous utilisons ^ et $ pour ancrer notre motif. Cela permet de considérer tous les caractères dans le texte pour l'opération de correspondance. Sans ces ancres, l'expression régulière nous indiquera seulement si le texte contient un nombre. Avec les ancres, elle indique si le texte contient seulement un nombre. L'étiquette i à la fin

	permet d'être insensible à la casse, et donc d'identifier à la fois le caractère « e » et « E ». Cela nous permet d'éviter les syntaxes suivantes [Ee] ou (?:E|e)
-?	Spécifie que le signe « -« est optionnel.
\d+	Un ou plusieurs chiffres.
(?:\.\d*)?	On utilise un groupement non-mémorisant afin d'améliorer les performances. Ceci permet d'identifier un séparateur décimal (ici le caractète « . ») suivi par 0 ou plusieurs chiffres.
(?::(\d+))?	Ce 4e groupement mémorisant identifie (optionnellementà le numéro de port, cest-à-dire une séquence d'un ou plusieurs chiffres précédés par le caractère « : ».
(?:e[+\-]?\d+)?	C'est un autre groupement optionnel, qui identifie le caractère « e » ou « E », un signe optionnel et un ou plusieurs chiffres.

3 Autres exemples

3.1 Les objets RegExp créés par des expressions régulières partagent une même instance

```
function test() {
      return /a/gi;
}
var x = test();
var y = test(); // x et y sont le même objet
x.lastIndex = 10;
document.writeln(y.lastIndex); // 10
```

3.2 Méthode exec - manipulation du tableau de résultats (1)

```
var re = new RegExp("JS*", "ig");
var str = "cfdsJS *(&YJSjs 888JS";
var resultArray = re.exec(str);
while (resultArray) {
   document.writeln(resultArray[0]);
   document.writeln(" prochaine correspondance débute à "+re.lastIndex+"<br />");
   resultArray = re.exec(str);
}
```

Le motif recherché ici est formé du caractère « J » suivi de 0 ou plusieurs fois le caractère « S ». En utilisant l'étiquette 'i', la casse est ignorée. En utilisant l'étiquette 'g', la propriété **lastIndex** de l'objet **RegExp** est définie sur la valeur de l'index où a été trouvé le dernier motif, pour chaque appel successif. De cette manière, chaque nouvel appel à la méthode exec permet de trouver le prochain motif. On affiche donc les 4 éléments trouvés, puis lorsqu'on n'en trouve plus, la valeur **null** est définie pour le tableau de résultats. Ceci permet de sortir de la boucle.

Voici la sortie du programme :

```
JS prochaine correspondance débute à 6
JS prochaine correspondance débute à 13
js prochaine correspondance débute à 15
JS prochaine correspondance débute à 21
```

La méthode **exec** retourne donc un tableau, mais les éléments du tableau ne sont pas tous des correspondances ; il s'agit de la correspondance en cours et les éventuels groupements mémorisants.

3.3 Méthode exec - manipulation du tableau de résultats (2)

En effet, si vous utilisez des parenthèses pour identifier des portions de la chaine de caractères globale, ces sélections sont incluses en tant qu'entrées succéssives du tableau résultat, à la suite de la chaîne identifiée (contenue à la position 0 du tableau).

Soit l'exemple suivant :

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <title>RegExp string matching</title>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <script type="text/javascript">
      //<![CDATA[
         window.onload = function() {
         var re = /(ds)+(j+s)/ig;
         var str = "cfdsJS *(&dsjjjsYJSjs 888dsdsJS";
         var resultArray = re.exec(str);
         while (resultArray) {
            document.writeln(resultArray[0]);
            document.writeln(" corresp. suiv. débute à "+re.lastIndex+"<br />");
            for (var i = 1; i < resultArray.length; i++) {
               document.writeln("sous-chaine de " + resultArray[i] + "<br />");
            }
            document.writeln("<br />");
            resultArray = re.exec(str);
         }
      }
      //]]>
      </script>
   </head>
<body>
   <p></p>
</body>
</html>
```

Ce code produit le résultat suivant :

```
dsJS corresp. suiv. débute à 6
sous-chaine de ds
sous-chaine de JS
dsjjjs corresp. suiv. débute à 16
sous-chaine de ds
sous-chaine de jjjs
dsdsJS corresp. suiv. débute à 31
sous-chaine de ds
sous-chaine de JS
```

4 Exercice – remplacer tous les espaces

a) Ecrivez le code JavaScript permettant, dans une chaîne de caractères, de :

- Trouver le motif composé d'un caractère d'espacement suivi du caractère « * »
- Remplacer ce motif par le caractère « - »
- Appliquer ce motif à la chaîne de caractères « Eh ça va la vache ! ».
- Afficher le résultat à l'utilisateur.

b) Modifiez le motif afin qu'il permette d'identifier tous les caractères entre guillemets.

5 Exercice – date

Ecrivez le code JavaScript, qui, pour une chaîne de caractères donnée, vérifie si elle contient une date au format « JJ Mois AAAA ». Nous faisons l'hypothèse qu'une telle date débute après le caractère « : ».

6 Exercice – inverser l'ordre des mots

Ecrivez une fonction JavaScript qui permet d'identifier des mots séparés par un ou plusieurs caractères "-", puis qui échange l'ordre des mots.

Exemple : (Entrée : Mot1--------Mot2) → (Sortie : Mot2-Mot1)

7 Exercice – expressions régulières pour formulaires

a) Ecrivez les motifs permettant de tester la validité des éléments suivants :

- Un identifiant de certificat : contient 13 caractères alphanumériques, avec les lettres « BUS » entre la 6e et la 8e position
- Une date aux formats suivants : JJ/MM/AAAA ou JJ/MM/AAAA HH:MM:SS
- Une adresse postale française, en retournant les différentes parties de l'adresse, à savoir : 0 = Adresse complète, 1 = Numéro de rue + BIS, TER ou QUATER, 2 = Numéro de rue, 3 et 4 = BIS, TER ou QUATER, 5 à 9 = Rien, 10 = Nom de rue 11 = Rien, 12 = code postal, 13 = Ville.
  - Exemple reconnu : 48 bis Avenue Des Champs Élysée 75001 Paris
  - Exemple non-reconnu : Rue De La Liberté 10 1000 Bruxelles
- Un numéro de TVA: 0 ou un "FR" + 0 ou un espace + 11 caractères (caractères alphabétiques compris - sauf O ou I (en 1e ou 2e positions ou en 1e et 2e position))
  - Exemples reconnus : 12345678901 - X1234567890 - 1X123456789 - XX123456789
  - Exemples non-reconnus : FR I123456789, FR O123456789, FR 1O23456789
- Un numéro de sécurité sociale français, avec ou sans la clé
  - Exemples reconnus : 181049520156962, 1 81 04 95 201 569 62, 1 81 04 95 201 569
  - Exemples non-reconnus : 1 81049520156962, 181049520156962fds, 1810495201569ds

b) Ecrivez le code JavaScript permettant de :

- Prendre en entrée une expression régulière saisie dans un champ de formulaire et une chaîne de caractères saisie dans un autre champ de formulaire
- Teste la correspondance entre les deux lorsque le formulaire est soumis.

- Les résultats du test sont affichés dans un troisième champ du même formulaire.

8 Ressources utiles

http://www.asciitable.com/

http://regexlib.com/