Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

Grammaire lexicale

Cette page décrit la grammaire lexicale de JavaScript. Le code source d'un script ECMAScript est analysé de gauche à droite et est converti en une série d'éléments qui sont : des jetons, des caractères de contrôle, des terminateurs de lignes, des commentaires ou des blancs. ECMAScript définit également certains mots-clés et littéraux. ECMAScript possède également des règles pour insérer automatiquement des points-virgules à la fin des instructions.

Caractères de contrôle

Les caractères de contrôle n'ont aucune représentation visuelle mais sont utilisés pour contrôler l'interprétation du texte.

Point de code	Nom	Abréviation	Description
`U+200C`	Antiliant sans chasse (*zero width non-joiner* en anglais)		Placé entre des caractères pour empêcher qu'ils soient connectés par une ligature dans certaines langues (Wikipédia).
`U+200D`	Liant sans chasse (*zero width joiner* en anglais)		Placé entre des caractères qui ne seraient normalement pas connectés pour les afficher comme connectés dans certaines langues (Wikipédia).
`U+FEFF`	Indicateur d'ordre des octets (*byte order mark* en anglais)		Utilisé au début d'un script pour indiquer qu'il est en Unicode et quel est l'ordre des octets (Wikipedia).

Blancs

Les caractères d'espacement (blancs) sont utilisés pour des raisons de lisibilité et permetttent de séparer les différents fragments entre eux. Ces caractères sont généralement inutiles au

code. Les outils de minification sont souvent utilisés pour retirer les blancs afin de réduire le volume de données à transférer.

Point de code	Nom	Abréviation	Description	Séquence d'échappement
U+0009	Tabulation (horizontale)		Tabulation horizontale	\t
U+000B	Tabulation verticale		Tabulation verticale	\v
U+000C	Caractère de saut de page (*form feed* en anglais)		Caractère de contrôle pour le saut de page (Wikipédia).	\f
U+0020	Espace sécable (*space* en anglais)		Espace sécable	
U+00A0	Espace insécable (*no-break space* en anglais)		Espace insécable	
Autres	Autres caractères d'espaces Unicode		Espaces Unicode sur Wikipédia	

Terminateurs de lignes

En plus des blancs, les caractères de fin de ligne (terminateurs de lignes) sont utilisés pour améliorer la lisibilité du texte. Cependant, dans certains cas, les terminateurs de lignes peuvent influencer l'exécution du code JavaScript là où ils sont interdits. Les terminateurs de lignes affectent également le processus d'insertion automatique des points-virgules. Les terminateurs de lignes correspondent à la classe **\s** des expressions rationnelles.

Seuls les points de code Unicode qui suivent sont traités comme des fins de lignes en ECMAScript, les autres caractères sont traités comme des blancs (par exemple : *Next Line*

(nouvelle ligne) : NEL, U+0085 est considéré comme un blanc).

Point de code	Nom	Abréviation	Description	Séquence d'échappement
U+000A	Nouvelle ligne		Caractère de nouvelle ligne pour les systèmes UNIX.	\n
U+000D	Retour chariot		Caractère de nouvelle ligne pour les systèmes Commodore et les premiers Mac.	\r
U+2028	Séparateur de ligne		Wikipédia	
U+2029	Séparateur de paragraphe		Wikipédia	

Commentaires

Les commentaires sont utilisés pour fournir des notes, des suggestions, des indications ou des avertissements sur le code JavaScript. Cela peut en faciliter la lecture et la compréhension. Ils peuvent également être utilisés pour empêcher l'exécution d'un certain code ; cela peut être pratique lors du débogage.

En JavaScript, Il existe actuellement deux façons de former des commentaires (cf. ci-après pour une troisième méthode en cours de discussion).

Commentaire sur une ligne

La première façon est d'utiliser `//` (double barre oblique), pour commenter tout le texte qui suit (sur la même ligne). Par exemple :

```
function comment() {
```

```
  // Voici un commentaire d'une ligne en JavaScript
  console.log("Hello world !");
}
comment();
```

Commentaire sur plusieurs lignes

La seconde façon est d'utiliser `/* */`, qui est plus flexible.

Il est possible d'utiliser cette forme sur une seule ligne :

```
function comment() {
  /* Voici un commentaire d'une ligne en JavaScript */
  console.log("Hello world !");
}
comment();
```

Mais également sur plusieurs lignes, comme ceci :

```
function comment() {
 /* Ce commentaire s'étend sur plusieurs lignes. Il n'y a
    pas besoin de clore le commentaire avant d'avoir
     fini. */
  console.log("Hello world !");
}
comment();
```

Il est également possible d'utiliser un commentaire au milieu d'une ligne. En revanche, cela rend le code plus difficile à lire et devrait être utilisé avec attention :

```
function comment(x) {
  console.log("Hello " + x /* insérer la valeur de x */ + " !");
}
comment("world");
```

On peut également encadrer du code pour l'empêcher d'être exécuté. Par exemple :

```
function comment() {
  /* console.log("Hello world !"); */
}
comment();
```

Ici, l'appel `console.log()` n'a jamais lieu car il fait partie d'un commentaire. On peut ainsi désactiver plusieurs lignes de code d'un coup.

Commentaire d'environnement (*hashbang*)

Une troisième syntaxe, en cours de standardisation par ECMAScript, permet d'indiquer l'environnement dans lequel est exécuté le script via un commentaire *hashbang* . Un tel commentaire commence par `#!` et est **uniquement valide au tout début du script ou du module** (aucun espace/blanc n'est autorisé avant `#!`). Un tel commentaire ne tient que sur une seule ligne et il ne peut y avoir qu'un seul commentaire de ce type.

```
#!/usr/bin/env node

console.log("Coucou le monde");
```

Les commentaires d'environnements sont conçus pour fonctionner comme les *shebangs* qu'on peut trouver sous Unix et indiquent l'interpréteur à utiliser pour exécuter le script ou le module.

> **Attention :** Bien qu'utiliser un BOM avant le *hashbang* fonctionne dans un navigateur, cela n'est pas conseillé. En effet, un BOM empêchera le bon fonctionnement sous Unix/Linux. Utilisez un encodage UTF-8 sans BOM si vous souhaitez exécuter vos scripts depuis une invite de commande.

Si vous souhaitez placer un commentaire en début de fichier sans indiquer d'environnement d'exécution spécifique, on pourra utiliser le commentaire classique avec `//` .

Mots-clés

Mots-clés réservés selon ECMAScript 2015

- `break`
- `case`
- `class`
- `catch`
- `const`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `else`

- `export`
- `extends`
- `finally`
- `for`
- `function`
- `if`
- `import`
- `in`
- `instanceof`
- `new`
- `return`
- `super`
- `switch`
- `this`
- `throw`
- `try`
- `typeof`
- `var`
- `void`
- `while`
- `with`
- `yield`

Mots-clés réservés pour le futur

Les mots-clés qui suivent ont été réservés pour une utilisation future dans la spécification ECMAScript. Ils n'ont actuellement aucune utilité mais pourrait être utilisés par la suite. Ils ne peuvent donc pas être utilisés comme identifiants. Ces mots-clés ne peuvent être utilisés ni en mode strict ni en mode non strict.

- `enum`
- `await` (lorsqu'il est utilisé dans le contexte d'un module)

Les mots-clés suivants sont réservés dans du code en mode strict :

- `implements`
- `let`

- `package`
- `protected`
- `static`
- `interface`
- `private`
- `public`

Mots-clés réservés pour un usage future dans les anciens standards

Les mots-clés suivants sont réservés dans les anciennes spécifications ECMAScript (ECMAScript 1 à 3).

- `abstract`
- `boolean`
- `byte`
- `char`
- `double`
- `final`
- `float`
- `goto`
- `int`
- `long`
- `native`
- `short`
- `synchronized`
- `throws`
- `transient`
- `volatile`

Par ailleurs, les littéraux `null`, `true`, et `false` sont réservés dans ECMAScript pour leur usage normal.

Utilisation des mots-clés réservés

Les mots-clés réservés ne le sont que pour les identifiants (et non pour les `IdentifierNames`). Comme décrit dans es5.github.com/#A.1, dans l'exemple qui suit, on a, légalement, des `IdentifierNames` qui utilisent des `ReservedWords`.

```
a.import
a["import"]
a = { import: "test" }.
```

En revanche, dans ce qui suit, c'est illégal car c'est un identifiant. Un identifiant peut être un `IdentifierName` mais pas un mot-clé réservé. Les identifiants sont utilisés pour les

`FunctionDeclaration` (déclarations de fonction), les `FunctionExpression` (expressions de fonction), les `VariableDeclaration` (déclarations de variable) .

```
function import() {} // Illégal.
```

Littéraux

Littéral `null`

Voir aussi la page `null` pour plus d'informations.

```
null
```

Littéraux booléens

Voir aussi la page `Boolean` pour plus d'informations.

```
true
false
```

Littéraux numériques

Décimaux

```
1234567890
42

// Attention à l'utilisation de zéros en début :

0888 // 888 est compris comme décimal
0777 // est compris comme octal et égale 511 en décimal
```

Les littéraux décimaux peuvent commencer par un zéro (`0`) suivi d'un autre chiffre. Mais si tous les chiffres après le 0 sont (strictement) inférieurs à 8, le nombre sera analysé comme un nombre octal. Cela n'entraînera pas d'erreur JavaScript, voir bug 957513 . Voir aussi la page sur `parseInt()` .

Binaires

La représentation binaire des nombres peut être utilisée avec une syntaxe qui comporte un zéro (0) suivi par le caractère latin "B" (minuscule ou majuscule) (`0b` ou `0B`). Cette syntaxe est apparue avec ECMAScript 2015 et il faut donc faire attention au tableau de compatibilité pour cette fonctionnalité. Si les chiffres qui composent le nombre ne sont pas 0 ou 1, cela entraînera une erreur `SyntaxError` : "Missing binary digits after 0b".

```
var FLT_SIGNBIT  = 0b10000000000000000000000000000000; // 2147483648
var FLT_EXPONENT = 0b01111111100000000000000000000000; // 2139095040
var FLT_MANTISSA = 0B00000000011111111111111111111111; // 8388607
```

Octaux

La syntaxe pour représenter des nombres sous forme octale est : un zéro (0), suivi par la lettre latine "O" (minuscule ou majuscule) (ce qui donne `0o` ou `0O`) . Cette syntaxe est apparue avec ECMAScript 2015 et il faut donc faire attention au tableau de compatibilité pour cette fonctionnalité. Si les chiffres qui composent le nombre ne sont pas compris entre 0 et 7, cela entraînera une erreur `SyntaxError` : "Missing octal digits after 0o".

```
var n = 0O755; // 493
var m = 0o644; // 420

// Aussi possible en utilisant des zéros en début du nombre (voir la note ci
0755
0644
```

Hexadécimaux

Les littéraux hexadécimaux ont pour syntaxe : un zéro (0), suivi par la lettre latine "X" (minuscule ou majuscule) (ce qui donne `0x` ou `0X`) . Si les chiffres qui composent le nombre sont en dehors des unités hexadécimales (0123456789ABCDEF), cela entraînera une erreur `SyntaxError` : "Identifier starts immediately after numeric literal".

```
0xFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF  // 81985529216486900
0XA                // 10
```

Littéraux `BigInt`

Le type `BigInt` est un type numérique primitif de JavaScript qui permet de représenter des entiers avec une précision arbitraire. De tels littéraux s'écrivent en ajoutant un `n` à la fin d'un entier.

```
123456789123456789n (nombre décimal, en base 10)
0o7777777777777777n (nombre octal, en base 8)
```

```
0x123456789ABCDEF1n (nombre hexadécimal, en base 16)
0b0101010101110101n (nombre binaire, en base 2)
```

Voir aussi le paragraphe sur les grands entiers/BigInt sur les structures de données en JavaScript.

Littéraux objets

Voir aussi les pages `Object` et Initialisateur d'objet pour plus d'informations.

```
var o = { a: "toto", b: "truc", c: 42 };

// notation raccourcie depuis ES6
var a = "toto", b = "truc", c = 42;
var o = {a, b, c};
// plutôt que
var o = { a: a, b: b, c: c };
```

Littéraux de tableaux

Voir aussi la page `Array` pour plus d'informations.

```
[1954, 1974, 1990, 2014]
```

Littéraux de chaînes de caractères

Un littéral de chaîne de caractères correspond à zéro ou plusieurs codets Unicode entourés de simples ou de doubles quotes. Les codets Unicode peuvent également être représentés avec des séquences d'échappements. Tous les codets peuvent apparaître dans un littéral de chaîne de caractères à l'exception de ces trois codets :

- U+005C \ (barre oblique inverse)
- U+000D (retour chariot, *carriage return*, *CR*)
- U+000A (saut de ligne, *line feed*, *LF*)

Avant la proposition consistant à rendre les chaînes JSON valides selon ECMA-262, les caractères U+2028 et U+2029 étaient également interdits.

Tous les codets peuvent être écrits sous la forme d'une séquence d'échappement. Les littéraux de chaînes de caractères sont évalués comme des valeurs `String` ECMAScript. Lorsque ces valeurs `String` sont générées, les codets Unicode sont encodés en UTF-16.

```
'toto'
"truc"
```

Séquence d'échappement hexadécimale

Une séquence d'échappement hexadécimale consiste en la succession de `\x` et de deux chiffres hexadécimaux représentant un codet sur l'intervalle 0x0000 à 0x00FF.

```
'\xA9' // "©"
```

Séquence d'échappement Unicode

La séquence d'échappement Unicode est composée de `\u` suivi de quatre chiffres hexadécimaux. Chacun de ces chiffres définit un caractères sur deux octets selon l'encodage UTF-16. Pour les codes situés entre `U+0000` et `U+FFFF`, les chiffres à utiliser sont identiques au code. Pour les codes supérieurs, il faudra utiliser deux séquences d'échappement dont chacune représentera un demi-codet de la paire de *surrogates*.

Voir aussi `String.fromCharCode()` et `String.prototype.charCodeAt()`.

```
'\u00A9' // "©" (U+A9)
```

Échappement de points de code Unicode

Apparu avec ECMAScript 2015, l'échappement de points de code Unicode permet d'échapper n'importe quel caractère en utilisant une notation hexadécimale. Il est possible de le faire pour échapper les points de code Unicode dont la représentation va jusqu'à `0x10FFFF`. Avec la séquence « simple » d'échappement Unicode, il était nécessaire d'échapper respectivement les deux demi-codets d'une paire si on voulait échapper le caractère correspondant, avec cette nouvelle méthode, ce n'est plus nécessaire de faire la distinction.

Voir également `String.fromCodePoint()` et `String.prototype.codePointAt()`.

```
'\u{2F804}' // CJK COMPATIBILITY IDEOGRAPH-2F804 (U+2F804)

// avec l'ancienne méthode d'échappement, cela aurait été écrit
// avec une paire de surrogates
'\uD87E\uDC04'
```

Littéraux d'expressions rationnelles

Voir la page `RegExp` pour plus d'informations.

```
/ab+c/g

// Un littéral pour une expression rationnelle
// vide. Le groupe non-capturant est utilisé pour
// lever l'ambigüité avec les commentaires
```

```
/(?:)/
```

Littéraux modèles (gabarits ou *templates*)

Voir également la page sur les gabarits de chaînes de caractères pour plus d'informations.

```
`chaîne de caractères`

`chaîne de caractères ligne 1
 chaîne de caractères ligne 2`

`chaîne1 ${expression} chaîne2`

tag `chaîne1 ${expression} chaîne2`
```

Insertion automatique de points-virgules

Certaines instructions JavaScript doivent finir par un point-virgule et sont donc concernées par l'insertion automatique de points-virgules (ASI pour *automatic semicolon insertion* en anglais) :

- Instruction vide
- instruction de variable, `let`, `const`
- `import`, `export`, déclaration de module
- Instruction d'expression
- `debugger`
- `continue`, `break`, `throw`
- `return`

La spécification ECMAScript mentionne trois règles quant à l'insertion de points-virgules :

1. Un point-vrigule est inséré avant un terminateur de ligne ou une accolade ("}") quand celui ou celle-ci n'est pas autorisé par la grammaire

```
{ 1 2 } 3
// est donc transformé, après ASI, en :
{ 1 2 ;} 3;
```

2. Un point-virgule est inséré à la fin lorsqu'on détecte la fin d'une série de jetons en flux d'entrée et que le parseur est incapable d'analyser le flux d'entrée comme un programme complet.

Ici `++` n'est pas traité comme opérateur postfixe (en-US) s'appliquant à la variable `b` car il y a

un terminateur de ligne entre `b` et `++`.

```
a = b
++c

// devient, après ASI :



a = b;
++c;
```

3. Un point-virgule est inséré à la fin, lorsqu'une instruction, à production limitée pour la grammaire, est suivie par un terminateur de ligne. Les instructions concernées par cette règle sont :

- Expressions postfixes (`++` et `--`)
- `continue`
- `break`
- `return`
- `yield`, `yield*`
- `module`

```
return
a + b

// est transformé, après ASI, en :

return;
a + b;
```

Spécifications

Spécification	État	Commentaires
ECMAScript 1st Edition (ECMA-262)	Standard	Définition initiale.
ECMAScript 5.1 (ECMA-262) La définition de 'Lexical Conventions' dans cette spécification.	Standard	
ECMAScript 2015 (6th Edition, ECMA-262)		Ajout : littéraux binaires et octaux, échappements

Spécification	État	Commentaires
ECMA-262) La définition de 'Lexical Grammar' dans cette spécification.	Standard	Ajout : littéraux binaires et octaux, échappements de points de code Unicode, modèles
ECMAScript (ECMA-262) La définition de 'Lexical Grammar' dans cette spécification.	Standard évolutif	

Compatibilité des navigateurs

Report problems with this compatibility data on GitHub

Array literals (`[1, 2, 3]`)	
Chrome	1
Edge	12
Firefox	1
Internet Explorer	4
Opera	4
Safari	1
WebView Android	1
Chrome Android	18
Firefox for Android	4
Opera Android	10.1
Safari on iOS	1
Samsung Internet	1.0
Deno	1.0
Node.js	0.10.0
Binary numeric literals (`0b`)	
Chrome	41
Edge	12
Firefox	25
Internet Explorer	**No**
Opera	28

Voir aussi

- Jeff Walden : Nombres binaires et forme octale (en anglais)
- Mathias Bynens : Séquences d'échappements de caractères (en anglais)
- `Boolean`
- `Number`
- `RegExp`
- `String`

Last modified: 28 janv. 2022, by MDN contributors