NOTE: I am using colours in this document to ensure that character styles are applied consistently. They can be removed by changing Word's character styles and will be removed for the final draft.

1 Scope

This Standard defines the ECMAScript Edition 4 scripting language.

2 Conformance

3 Normative References

4 Overview

5 Notational Conventions

5.1 Characters

Throughout this document, the phrase *code point* and the word *character* is used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of Unicode text in the UTF-16 transformation format. The phrase *Unicode character* is used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code point). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual Unicode characters, even though a user might think of the whole sequence as a single character.

When denoted in this specification, characters with values between 20 and 7E hexadecimal inclusive are in a `fixed width font`. Other characters are denoted by enclosing their four-digit hexadecimal Unicode value between «u and ». For example, the non-breakable space character would be denoted in this document as «u00A0». A few of the common control characters are represented by name:

Abbreviation	Unicode Value
«NUL»	«u0000»
«BS»	«u0008»
«TAB»	«u0009»
«LF»	«u000A»
«VT»	«u000B»
«FF»	«u000C»
«CR»	«u000D»
«SP»	«u0020»

A space character is denoted in this document either by a blank space where it's obvious from the context or by «SP» where the space might be confused with some other notation.

5.2 Notation

This specification uses the notation below to represent algorithms and concepts. These concepts are used as notation and are not necessarily represented or visible in the ECMAScript language.

5.2.1 Symbols

This specification uses *symbols* as computational tokens. Symbols are written using a **bold sans-serif font**. A symbol is equal only to itself. Examples of symbols include **true**, **false**, **null**, **NaN**, and **identifier**. The symbol **true** is used to indicate a true statement, and **false** is used to indicate a false statement.

5.2.2 Numbers

Numbers written in this specification are to be understood to be exact mathematical real numbers, which include integers and rational numbers as subsets. Examples of numbers include -3, 0, 17, 10^{1000}, and π. Hexadecimal numbers are written by preceding them with "0x", so 4294967296, 0x100000000, and 2^{32} are all the same integer.

The usual mathematical operators +, –, * (multiplication), and / can be used on numbers and produce exact results. Numbers are never divided by zero in this specification.

Numbers can be compared using =, ≠, <, ≤, >, and ≥, and the result is either the symbol **true** or the symbol **false** as appropriate. Multiple relational operators can be cascaded, so $x < y < z$ is **true** only if both x is less than y and y is less than z.

Other numeric operations include:

x^y	x raised to the y^{th} power (used only when either $x \neq 0$ and y is an integer or x is any number and $y > 0$)
$\|x\|$	The absolute value of x, which is x if $x \geq 0$ and $-x$ otherwise.
$\lfloor x \rfloor$	The greatest integer less than or equal to x. $\lfloor 3.7 \rfloor$ is 3, $\lfloor -3.7 \rfloor$ is –4, and $\lfloor 5 \rfloor$ is 5.
$\lceil x \rceil$	The least integer greater than or equal to x. $\lceil 3.7 \rceil$ is 4, $\lceil -3.7 \rceil$ is –3, and $\lceil 5 \rceil$ is 5.

The set of all real numbers is denoted as **Real**, the set of all rational numbers is denoted as **Rational**, and the set of all integers is denoted as **Integer**.

5.2.3 Sets

A set is an unordered, possibly infinite collection of elements. Each element may occur at most once in a set. There must be a well-defined equivalence relation = on the elements of a set.

A set is denoted by enclosing a comma-separated list of values inside braces:

$\{element_1, element_2, \ldots, element_n\}$

For example, the set {3, 0, 10, 11, 12, 13, -5} contains seven integers. The empty set is written as {}.

A set can also be written using the set comprehension notation

$\{f(a) \mid predicate_1(a); \ldots ; predicate_n(a)\}$

which denotes the set of the results of evaluating expression f on all values a that simultaneously satisfy all *predicate* expressions. There can also be more than one free variable a. For example,

$\{x \mid x \in$ **Integer**$; x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}$

$\{x*10 + y \mid x \in \{1, 2, 4\}; y \in \{3, 5\}\} = \{13, 15, 23, 25, 43, 45\}$

Let A and B be sets and x be a value. The following notation is used on sets:

$\|A\|$	The number of elements in the set A (can only be used on finite sets)
min A	The value m that satisfies both $m \in A$ and for all elements $x \in A$, $x \geq m$ (can only be used on nonempty, finite sets)

max A	The value m that satisfies both $m \in A$ and for all elements $x \in A$, $x \leq m$ (can only be used on nonempty, finite sets)
$A \cap B$	The intersection of sets A and B (the set of all values that are present both in A and in B)
$A \cup B$	The union of sets A and B (the set of all values that are present in at least one of A or B)
$A - B$	The difference of sets A and B (the set of all values that are present in A but not B)
$x \in A$	**true** if x is an element of set A and **false** if not
$x, y \in A$	**true** if x and y are both elements of set A and **false** if not
$A = B$	**true** if sets A and B are equal and **false** otherwise; sets A and B are equal if every element of A is also in B and every element of B is also in A

5.2.4 Floating-Point Numbers

The set **Float64** denotes all representable double-precision floating-point IEEE 754 values, with all not-a-number values considered indistinguishable from each other. The set **Float64** is the union of the following sets:

Float64 = **NormalisedFloat64** ∪ **DenormalisedFloat64** ∪ {**+0.0**, **-0.0**, **+∞**, **-∞**, **NaN**}

There are 18428729675200069632 (that is, $2^{64}-2^{54}$) normalised values:

NormalisedFloat64 = $\{s*m*2^e \mid s \in \{-1, 1\};\ m, e \in$ **Integer**; $2^{52} \leq m < 2^{53}$; $-1074 \leq e \leq 971\}$

m is called the *significand*.

There are also 9007199254740990 (that is, $2^{53}-2$) denormalised non-zero values:

DenormalisedFloat64 = $\{s*m*2^{-1074} \mid s \in \{-1, 1\};\ m \in$ **Integer**; $0 < m < 2^{52}\}$

m is called the *significand*.

The remaining values are the symbols **+0.0** (positive zero), **-0.0** (negative zero), **+∞** (positive infinity), **-∞** (negative infinity), and **NaN** (not a number).

The function *realToFloat64* converts a real number x into the applicable element of **Float64** as follows:

realToFloat64(x)

1. Let S = **NormalisedFloat64** ∪ **DenormalisedFloat64** ∪ $\{0, 2^{1024}, -2^{1024}\}$.
2. Let a be the element of S closest to x (i.e. such that $|a-x|$ is as small as possible). If two elements of S are equally close, let a be the one with an even significand; for this purpose 0, 2^{1024}, and -2^{1024} are considered to have even significands.
3. If $a = 2^{1024}$, return **+∞**.
4. If $a = -2^{1024}$, return **-∞**.
5. If $a \neq 0$, return a.
6. If $x < 0$, return **-0.0**.
7. Return **+0.0**.

NOTE This procedure corresponds exactly to the behaviour of the IEEE 754 "round to nearest" mode.

5.2.5 Other Notation

5.3 Algorithm Conventions

5.4 Grammars

The lexical and syntactic structure of ECMAScript programs is described in terms of *context-free grammars*. A context-free grammar consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet. A *grammar symbol* is either a terminal or a nonterminal.

Each grammar contains at least one distinguished nonterminal called the *goal symbol*. If there is more than one goal symbol, the grammar specifies which one is to be used. A *sentential form* is a possibly empty sequence of grammar symbols that satisfies the following recursive constraints:

- The sequence consisting of only the goal symbol is a sentential form.
- Given any sentential form α that contains a nonterminal *N*, one may replace an occurrence of *N* in α with the right-hand side of any production for which *N* is the left-hand side. The resulting sequence of grammar symbols is also a sentential form.

A *derivation* is a record, usually expressed as a tree, of which production was applied to expand each intermediate nonterminal to obtain a sentential form starting from the goal symbol. The grammars in this document are unambiguous, so each sentential form has exactly one derivation.

A *sentence* is a sentential form that contains only terminals. A *sentence prefix* is any prefix of a sentence, including the empty prefix consisting of no terminals and the complete prefix consisting of the entire sentence.

A *language* is the (perhaps infinite) set of a grammar's sentences.

5.4.1 Grammar Notation

Terminal symbols are either literal characters (section 5.1), sequences of literal characters (syntactic grammar only), or other terminals such as **Identifier** defined by the grammar. These other terminals are denoted in **bold**.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a ⇒ and one or more expansions of the nonterminal separated by vertical bars (|). The expansions are usually listed on separate lines but may be listed on the same line if they are short. An empty expansion is denoted as «empty».

To aid in reading the grammar, some rules contain informative cross-references to sections where nonterminals used in the rule are defined. These cross-references appear in parentheses in the right margin.

For example, the syntactic definition

SampleList ⇒
«empty»
| . . . *Identifier* (*Identifier*: **Error! Reference source not found.**)
| *SampleListPrefix*
| *SampleListPrefix* , . . . *Identifier*

states that the nonterminal *SampleList* can represent one of four kinds of sequences of input tokens:

- It can represent nothing (indicated by the «empty» alternative).
- It can represent the terminal . . . followed by any expansion of the nonterminal *Identifier*.
- It can represent any expansion of the nonterminal *SampleListPrefix*.

- It can represent any expansion of the nonterminal *SampleListPrefix* followed by the terminals `,` and `...` and any expansion of the nonterminal *Identifier*.

5.4.2 Lookahead Constraints

If the phrase "[lookahead ∉ *set*]" appears in the expansion of a nonterminal, it indicates that that expansion may not be used if the immediately following terminal is a member of the given *set*. That *set* can be written as a list of terminals enclosed in curly braces. For convenience, *set* can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand.

For example, given the rules

DecimalDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

DecimalDigits ⇒
    DecimalDigit
  | *DecimalDigits DecimalDigit*

the rule

LookaheadExample ⇒
    `n` [lookahead ∉ {`1`, `3`, `5`, `7`, `9`}] *DecimalDigits*
  | *DecimalDigit* [lookahead ∉ {*DecimalDigit*}]

matches either the letter `n` followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

5.4.3 Line Break Constraints

If the phrase "[no line break]" appears in the expansion of a production, it indicates that this production cannot be used if there is a line break in the input stream at the indicated position. Line break constraints are only present in the syntactic grammar. For example, the rule

ReturnStatement ⇒
    `return`
  | **`return`** [no line break] *ListExpression*^{allowIn}

indicates that the second production may not be used if a line break occurs in the program between the **`return`** token and the *ListExpression*^{allowIn}.

Unless the presence of a line break is forbidden by a constraint, any number of line breaks may occur between any two consecutive terminals in the input to the syntactic grammar without affecting the syntactic acceptability of the program.

5.4.4 Parameterised Rules

Many rules in the grammars occur in groups of analogous rules. Rather than list them individually, these groups have been summarised using the shorthand illustrated by the example below:

Metadefinitions such as

$\alpha \in$ {normal, initial}

$\beta \in$ {allowIn, noIn}

introduce grammar arguments α and β. If these arguments later parameterise the nonterminal on the left side of a rule, that rule is implicitly replicated into a set of rules in each of which a grammar argument is consistently substituted by one of its variants. For example, the sample rule

AssignmentExpression$^{\alpha,\beta}$ ⇒
    ConditionalExpression$^{\alpha,\beta}$
  | *LeftSideExpression*$^{\alpha}$ **`=`** *AssignmentExpression*$^{normal,\beta}$
  | *LeftSideExpression*$^{\alpha}$ *CompoundAssignment AssignmentExpression*$^{normal,\beta}$

expands into the following four rules:

AssignmentExpression^{normal,allowIn} ⇒
    ConditionalExpression^{normal,allowIn}
  | *LeftSideExpression*^{normal} **=** *AssignmentExpression*^{normal,allowIn}
  | *LeftSideExpression*^{normal} *CompoundAssignment AssignmentExpression*^{normal,allowIn}

AssignmentExpression^{normal,noIn} ⇒
    ConditionalExpression^{normal,noIn}
  | *LeftSideExpression*^{normal} **=** *AssignmentExpression*^{normal,noIn}
  | *LeftSideExpression*^{normal} *CompoundAssignment AssignmentExpression*^{normal,noIn}

AssignmentExpression^{initial,allowIn} ⇒
    ConditionalExpression^{initial,allowIn}
  | *LeftSideExpression*^{initial} **=** *AssignmentExpression*^{normal,allowIn}
  | *LeftSideExpression*^{initial} *CompoundAssignment AssignmentExpression*^{normal,allowIn}

AssignmentExpression^{initial,noIn} ⇒
    ConditionalExpression^{initial,noIn}
  | *LeftSideExpression*^{initial} **=** *AssignmentExpression*^{normal,noIn}
  | *LeftSideExpression*^{initial} *CompoundAssignment AssignmentExpression*^{normal,noIn}

AssignmentExpression^{normal,allowIn} is now an unparametrised nonterminal and processed normally by the grammar.

Some of the expanded rules (such as the fourth one in the example above) may be unreachable from the grammar's starting nonterminal; these are ignored.

5.4.5 Special Lexical Rules

A few lexical rules have too many expansions to be practically listed. These are specified by descriptive text instead of a list of expansions after the ⇒.

Some lexical rules contain the metaword **except**. These rules match any expansion that is listed before the **except** but that does not match any expansion after the **except**; if multiple expansions are listed after the **except**, then they are separated by vertical bars (|). All of these rules ultimately expand into single characters. For example, the rule below matches any single *UnicodeCharacter* except the * and / characters:

NonAsteriskOrSlash ⇒ *UnicodeCharacter* **except** * | /

6 Source Text

ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 2.1 or later, using the UTF-16 transformation format. The text is expected to have been normalised to Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

ECMAScript source text can contain any of the Unicode characters. All Unicode white space characters are treated as white space, and all Unicode line/paragraph separators are treated as line separators. Non-Latin Unicode characters are allowed in identifiers, string literals, regular expression literals and comments.

In string literals, regular expression literals and identifiers, any character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely **\u** plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

NOTE Although this document sometimes refers to a "transformation" between a "character" within a "string" and the 16-bit unsigned integer that is the UTF-16 encoding of that character, there is actually no transformation because a "character" within a "string" is actually represented using that 16-bit unsigned value.

NOTE ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence **\u000A**, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character **000A** is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence **\u000A** occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write **\n** instead of **\u000A** to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

6.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category Cf in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a. Unicode escape sequence (see section **Error! Reference source not found.**) to include a Unicode format-control character inside a string or regular expression literal.

7 Lexical Grammar

This section defines ECMAScript's *lexical grammar*. This grammar translates the source text into a sequence of *input elements*, which are either tokens or the special markers **lineBreak** and **endOfInput**.

A *token* is one of the following:

- A **keyword** token, which is either:
  - One of the reserved words `abstract`, `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `enum`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `goto`, `if`, `implements`, `import`, `in`, `instanceof`, `interface`, `namespace`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `typeof`, `use`, `var`, `volatile`, `while`, `with`.
  - One of the non-reserved words `exclude`, `get`, `include`, `set`.
- A **punctuator** token, which is one of `!`, `!=`, `!==`, `#`, `%`, `%=`, `&`, `&&`, `&&=`, `&=`, `(`, `)`, `*`, `*=`, `+`, `++`, `+=`, `,`, `-`, `--`, `-=`, `->`, `.`, `..`, `...`, `/`, `/=`, `:`, `::`, `;`, `<`, `<<`, `<<=`, `<=`, `=`, `==`, `===`, `>`, `>=`, `>>`, `>>=`, `>>>`, `>>> =`, `?`, `@`, `[`, `]`, `^`, `^=`, `^^`, `^^=`, `{`, `|`, `|=`, `||`, `||=`, `}`, `~`.
- An **identifier** token, which carries a string that is the identifier's name.
- A **number** token, which carries a number that is the string's value.
- A **string** token, which carries a string that is the string's value.
- A **regularExpression** token, which carries two strings — the regular expression's body and its flags.

A **lineBreak**, although not considered to be a token, also becomes part of the stream of input elements and guides the process of automatic semicolon insertion (section **Error! Reference source not found.**). **endOfInput** signals the end of the source text.

NOTE The lexical grammar discards simple white space and single-line comments. They do not appear in the stream of input elements for the syntactic grammar. Comments spanning several lines become **lineBreak**s.

The lexical grammar has individual characters as its terminal symbols plus the special terminal **End**, which is appended after the last input character. The lexical grammar defines three goal symbols *NextInputElement*^{re}, *NextInputElement*^{div}, and *NextInputElement*^{unit}, a set of productions, and instructions for translating the source text into input elements. The choice of the goal symbol depends on the syntactic grammar, which means that lexical and syntactic analysis are interleaved.

NOTE The grammar uses *NextInputElement*^{unit} if the previous token was a number, *NextInputElement*^{re} if the previous token was not a number and a / should be interpreted as starting a regular expression, and *NextInputElement*^{div} if the previous token was not a number and a / should be interpreted as a division or division-assignment operator.

The sequence of input elements *inputElements* is obtained as follows:

1. Let *inputElements* be an empty sequence of input elements.
2. Let *input* be the input sequence of characters. Append a special placeholder **End** to the end of *input*.
3. Let *state* be a variable that holds one of the constants **re**, **div**, or **unit**. Initialise it to **re**.
4. Repeat the following steps until exited:
5. Find the longest possible prefix *P* of *input* that is a member of the lexical grammar's language (see section 5.4). Use the start symbol *NextInputElement*^{re}, *NextInputElement*^{div}, or *NextInputElement*^{unit} depending on whether *state* is **re**, **div**, or **unit**, respectively. If the parse failed, signal a syntax error.
6. Compute the action *Lex* on the derivation of *P* to obtain an input element *e*.
7. If *e* is **endOfInput**, then exit the repeat loop and go to step **Error! Bookmark not defined.**~~14~~.
8. Remove the prefix *P* from *input*, leaving only the yet-unprocessed suffix of *input*.
9. Append *e* to the end of the *inputElements* sequence.
10. If the *inputElements* sequence does not form a valid sentence prefix of the language defined by the syntactic grammar, then:
11. If *e* is not **lineBreak**, but the next-to-last element of *inputElements* is **lineBreak**, then insert a **VirtualSemicolon** terminal between the next-to-last element and *e* in *inputElements*.
12. If *inputElements* still does not form a valid sentence prefix of the language defined by the syntactic grammar, signal a syntax error.

    End if
13. If *e* is a **Number** token, then set *state* to **unit**. Otherwise, if the *inputElements* sequence followed by the terminal / forms a valid sentence prefix of the language defined by the syntactic grammar, then set *state* to **div**; otherwise, set *state* to **re**.

    End repeat
14. If the *inputElements* sequence does not form a valid sentence of the context-free language defined by the syntactic grammar, signal a syntax error and stop.
15. Return *inputElements*.

7.1 Input Elements

Syntax

NextInputElement^{re} ⇒ *WhiteSpace InputElement*^{re} (*WhiteSpace*: 7.2)

NextInputElement^{div} ⇒ *WhiteSpace InputElement*^{div}

NextInputElement^{unit} ⇒
  [lookahead∉ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace InputElement*^{div}
| [lookahead∉ {_}] *IdentifierName* (*IdentifierName*: 7.5)
| _ *IdentifierName*

InputElement^{re} ⇒
    LineBreaks (*LineBreaks*: 7.3)
  | *IdentifierOrKeyword* (*IdentifierOrKeyword*: 7.5)
  | *Punctuator* (*Punctuator*: 7.6)
  | *NumericLiteral* (*NumericLiteral*: 7.7)
  | *StringLiteral* (*StringLiteral*: 7.8)
  | *RegExpLiteral* (*RegExpLiteral*: 7.9)
  | *EndOfInput*

InputElement^{div} ⇒
    LineBreaks
  | *IdentifierOrKeyword*
  | *Punctuator*
  | *DivisionPunctuator* (*DivisionPunctuator*: 7.6)
  | *NumericLiteral*
  | *StringLiteral*
  | *EndOfInput*

EndOfInput ⇒
    End
  | *LineComment* **End** (*LineComment*: 7.4)

Semantics

The grammar parameter *v* can be either re or div.

Lex[*NextInputElement*^{re} ⇒ *WhiteSpace InputElement*^{re}] = *Lex*[*InputElement*^{re}]

Lex[*NextInputElement*^{div} ⇒ *WhiteSpace InputElement*^{div}] = *Lex*[*InputElement*^{div}]

Lex[*NextInputElement*^{unit} ⇒ [lookahead∉ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace* *InputElement*^{div}] = *Lex*[*InputElement*^{div}]

Lex[*NextInputElement*^{unit} ⇒ [lookahead∉ {_}] *IdentifierName*]
Lex[*NextInputElement*^{unit} ⇒ _ *IdentifierName*]
  Return a **string** token with string contents *LexString*[*IdentifierName*].

Lex[*InputElement*^{v} ⇒ *LineBreaks*] = **lineBreak**

Lex[*InputElement*^{v} ⇒ *IdentifierOrKeyword*] = *Lex*[*IdentifierOrKeyword*]

Lex[*InputElement*^{v} ⇒ *Punctuator*] = *Lex*[*Punctuator*]

Lex[*InputElement*^{div} ⇒ *DivisionPunctuator*] = *Lex*[*DivisionPunctuator*]

Lex[*InputElement*^{v} ⇒ *NumericLiteral*] = *Lex*[*NumericLiteral*]

Lex[*InputElement*^{v} ⇒ *StringLiteral*] = *Lex*[*StringLiteral*]

Lex[*InputElement*^{re} ⇒ *RegExpLiteral*] = *Lex*[*RegExpLiteral*]

Lex[*InputElement*^{v} ⇒ *EndOfInput*] = **endOfInput**

7.2 White space

Syntax

WhiteSpace ⇒
«empty»
| *WhiteSpace WhiteSpaceCharacter*
| *WhiteSpace SingleLineBlockComment* (*SingleLineBlockComment*: 7.4)

WhiteSpaceCharacter ⇒
«TAB» | «VT» | «FF» | «SP» | «u00A0»
| Any other character in category Zs in the Unicode Character Database

NOTE White space characters are used to improve source text readability and to separate tokens from each other, but are otherwise insignificant. White space may occur between any two tokens except between a number and an unquoted unit.

7.3 Line Breaks

Syntax

LineBreak ⇒
LineTerminator
| *LineComment LineTerminator* (*LineComment*: 7.4)
| *MultiLineBlockComment* (*MultiLineBlockComment*: 7.4)

LineBreaks ⇒
LineBreak
| *LineBreaks WhiteSpace LineBreak* (*WhiteSpace*: 7.2)

LineTerminator ⇒ «LF» | «CR» | «u2028» | «u2029»

NOTE Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section **Error! Reference source not found.**).

7.4 Comments

Syntax

LineComment ⇒ / / *LineCommentCharacters*

LineCommentCharacters ⇒
«empty»
| *LineCommentCharacters NonTerminator*

SingleLineBlockComment ⇒ / * *BlockCommentCharacters* * /

BlockCommentCharacters ⇒
«empty»
| *BlockCommentCharacters NonTerminatorOrSlash*
| *PreSlashCharacters* /

PreSlashCharacters ⇒
«empty»
| *BlockCommentCharacters NonTerminatorOrAsteriskOrSlash*
| *PreSlashCharacters* /

MultiLineBlockComment ⇒ / * *MultiLineBlockCommentCharacters BlockCommentCharacters* * /

MultiLineBlockCommentCharacters ⇒
    BlockCommentCharacters LineTerminator (*LineTerminator*: 7.3)
  | *MultiLineBlockCommentCharacters BlockCommentCharacters LineTerminator*

UnicodeCharacter ⇒ Any character

NonTerminator ⇒ *UnicodeCharacter* **except** *LineTerminator*

NonTerminatorOrSlash ⇒ *NonTerminator* **except** `/`

NonTerminatorOrAsteriskOrSlash ⇒ *NonTerminator* **except** `*` | `/`

NOTE Comments can be either line comments or block comments. Line comments start with a `//` and continue to the end of the line. Block comments start with `/*` and end with `*/`. Block comments can span multiple lines but cannot nest.

Except when it is on the last line of input, a line comment is always followed by a *LineTerminator*. That *LineTerminator* is not considered to be part of that line comment; it is recognised separately and becomes a **lineBreak**. A block comment that actually spans more than one line is also considered to be a **lineBreak**.

7.5 Keywords and Identifiers

Syntax

IdentifierOrKeyword ⇒ *IdentifierName*

IdentifierName ⇒
    InitialIdentifierCharacterOrEscape
  | *NullEscapes InitialIdentifierCharacterOrEscape*
  | *IdentifierName ContinuingIdentifierCharacterOrEscape*
  | *IdentifierName NullEscape*

Semantics

Lex[*IdentifierOrKeyword* ⇒ *IdentifierName*]

1 Let *id* be the string *LexString*[*IdentifierName*].

2 If *IdentifierName* contains no escape sequences (i.e. expansions of the *NullEscape* or *HexEscape* nonterminals) and exactly matches one of the keywords `abstract`, `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `enum`, `exclude`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `get`, `goto`, `if`, `implements`, `import`, `in`, `include`, `instanceof`, `interface`, `namespace`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `set`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `typeof`, `use`, `var`, `volatile`, `while`, `with`, then return a **keyword** token with string contents *id*.

3 Return an **identifier** token with string contents *id*.

NOTE Even though the lexical grammar treats `exclude`, `get`, `include`, and `set` as keywords, the syntactic grammar contains productions that permit them to be used as identifier names. The other keywords are reserved and may not be used as identifier names. However, an *IdentifierName* can never be a keyword if it contains any escape characters, so, for example, one can use `new` as the name of an identifier by including an escape sequence in it; `\_new` is one possibility, and `n\x65w` is another.

LexString[*IdentifierName* ⇒ *InitialIdentifierCharacterOrEscape*]

LexString[*IdentifierName* ⇒ *NullEscapes InitialIdentifierCharacterOrEscape*]

Return a one-character string with the character *LexChar*[*InitialIdentifierCharacterOrEscape*].

LexString[*IdentifierName* ⇒ *IdentifierName*$_1$ *ContinuingIdentifierCharacterOrEscape*]

Return a string consisting of the string *LexString*[*IdentifierName*$_1$] concatenated with the character *LexChar*[*ContinuingIdentifierCharacterOrEscape*].

LexString[*IdentifierName* ⇒ *IdentifierName*$_1$ *NullEscape*]
Return the string *LexString*[*IdentifierName*$_1$].

Syntax

NullEscapes ⇒
NullEscape
| *NullEscapes NullEscape*

NullEscape ⇒ \ _

InitialIdentifierCharacterOrEscape ⇒
InitialIdentifierCharacter
| \ *HexEscape* (*HexEscape*: 7.8)

InitialIdentifierCharacter ⇒ *UnicodeInitialAlphabetic* | $ | _

UnicodeInitialAlphabetic ⇒ Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), or Nl (letter number) in the Unicode Character Database

ContinuingIdentifierCharacterOrEscape ⇒
ContinuingIdentifierCharacter
| \ *HexEscape*

ContinuingIdentifierCharacter ⇒ *UnicodeAlphanumeric* | $ | _

UnicodeAlphanumeric ⇒ Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), Nd (decimal number), Nl (letter number), Mn (non-spacing mark), Mc (combining spacing mark), or Pc (connector punctuation) in the Unicode Character Database

Semantics

LexChar[*InitialIdentifierCharacterOrEscape* ⇒ *InitialIdentifierCharacter*]
Return the character *InitialIdentifierCharacter*.

LexChar[*InitialIdentifierCharacterOrEscape* ⇒ \ *HexEscape*]
1 Let *ch* be the character *LexChar*[*HexEscape*].
2 If *ch* is in the set of characters accepted by the nonterminal *InitialIdentifierCharacter*, then return *ch*.
3 Signal a syntax error.

LexChar[*ContinuingIdentifierCharacterOrEscape* ⇒ *ContinuingIdentifierCharacter*]
Return the character *ContinuingIdentifierCharacter*.

LexChar[*ContinuingIdentifierCharacterOrEscape* ⇒ \ *HexEscape*]
1 Let *ch* be the character *LexChar*[*HexEscape*].
2 If *ch* is in the set of characters accepted by the nonterminal *ContinuingIdentifierCharacter*, then return *ch*.
3 Signal a syntax error.

The characters in the specified categories in version 2.1 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations; however, conforming ECMAScript implementations may allow additional legal identifier characters based on the category assignment from later versions of Unicode.

NOTE Identifiers are interpreted according to the grammar given in Section 5.16 of version 3.0 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode standard. This standard specifies one departure from the grammar given

in the Unicode standard: $ and _ are permitted anywhere in an identifier. $ is intended for use only in mechanically generated code.

Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier. An escape sequence cannot be used to put a character into an identifier that would otherwise be illegal in that position of the identifier.

Two identifiers that are canonically equivalent according to the Unicode standard are *not* equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

7.6 Punctuators

Syntax

Punctuator ⇒

!	! =	! = =	#	%	% =	&
& &	& & =	& =	(	)	*	* =
+	+ +	+ =	,	-	- -	- =
- >	.	. .	. . .	:	: :	;
<	< <	< < =	< =	=	= =	= = =
>	> =	> >	> > =	> > >	> > > =	?
@	[	]	^	^ =	^ ^	^ ^ =
{	|	| =	| |	| | =	}	~

DivisionPunctuator ⇒
/ [lookahead∉ {/, *}]
| / =

Semantics

Lex[*Punctuator*]
Return a **punctuator** token with string contents *Punctuator*.

Lex[*DivisionPunctuator*]
Return a **punctuator** token with string contents *DivisionPunctuator*.

7.7 Numeric literals

Syntax

NumericLiteral ⇒
DecimalLiteral
| *HexIntegerLiteral* [lookahead∉ {*HexDigit*}]

DecimalLiteral ⇒
Mantissa
| *Mantissa LetterE SignedInteger*

LetterE ⇒ E | e

Mantissa ⇒
DecimalIntegerLiteral
| *DecimalIntegerLiteral* .
| *DecimalIntegerLiteral* . *DecimalDigits*
| . *Fraction*

DecimalIntegerLiteral ⇒
    `0`
  | *NonZeroDecimalDigits*

NonZeroDecimalDigits ⇒
    NonZeroDigit
  | *NonZeroDecimalDigits ASCIIDigit*

SignedInteger ⇒
    DecimalDigits
  | `+` *DecimalDigits*
  | `-` *DecimalDigits*

DecimalDigits ⇒
    ASCIIDigit
  | *DecimalDigits ASCIIDigit*

HexIntegerLiteral ⇒
    `0` *LetterX HexDigit*
  | *HexIntegerLiteral HexDigit*

LetterX ⇒ `X` | `x`

ASCIIDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

NonZeroDigit ⇒ `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

HexDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

Semantics

Lex[*NumericLiteral* ⇒ *DecimalLiteral*]

Return a **number** token with numeric contents *LexNumber*[*DecimalLiteral*].

~~1. Let x = *LexNumber*[*DecimalLiteral*].~~
~~2. If $x \neq 0$ then:~~
~~3. Let a be the integer such that $10^{19} \leq x*10^{a} < 10^{20}$.~~
~~4. Let $m = x*10^{a}$.~~
~~5. At the implementation's discretion either (i) let $n = m$, (ii) let $n = \lfloor m \rfloor$, or (iii) let $n = \lceil m \rceil$. Implementations are encouraged to make choice (i).~~
~~6. Set $x = n/10^{a}$.~~
~~End if~~
~~7. Return a **number** token with numeric contents *realToFloat64*(x).~~

~~**NOTE** The choice in step 5 above allows an implementation to ignore the values of digits past the 20th significant digit of the mantissa. If a mantissa has more than 20 significant digits, an implementation may either (i) use the exact value of the mantissa, (ii) round the mantissa down to the nearest 20-significant-digit mantissa, or (iii) round the mantissa up to the nearest 20-significant-digit mantissa.~~

Lex[*NumericLiteral* ⇒ *HexIntegerLiteral* [lookahead∉ {*HexDigit*}]]

Return a **number** token with numeric contents ~~*realToFloat64*(~~*LexNumber*[*HexIntegerLiteral*]~~)~~.

NOTE Note that all digits of hexadecimal literals are significant.

LexNumber[*DecimalLiteral* ⇒ *Mantissa*] = *LexNumber*[*Mantissa*]

LexNumber[*DecimalLiteral* ⇒ *Mantissa LetterE SignedInteger*]

1 Let *e* = *LexNumber*[*SignedInteger*].
2 Return *LexNumber*[*Mantissa*]*10^{e}.

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral*] = *LexNumber*[*DecimalIntegerLiteral*]

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral* `.`] = *LexNumber*[*DecimalIntegerLiteral*]

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral* `.` *Fraction*]
Return *LexNumber*[*DecimalIntegerLiteral*] + *LexNumber*[*Fraction*].

LexNumber[*Mantissa* ⇒ `.` *Fraction*] = *LexNumber*[*Fraction*]

LexNumber[*DecimalIntegerLiteral* ⇒ `0`] = 0

LexNumber[*DecimalIntegerLiteral* ⇒ *NonZeroDecimalDigits*] = *LexNumber*[*NonZeroDecimalDigits*]

LexNumber[*NonZeroDecimalDigits* ⇒ *NonZeroDigit*] = *LexNumber*[*NonZeroDigit*]

LexNumber[*NonZeroDecimalDigits* ⇒ *NonZeroDecimalDigits*$_1$ *ASCIIDigit*]
= 10**LexNumber*[*NonZeroDecimalDigits*$_1$] + *LexNumber*[*ASCIIDigit*]

LexNumber[*Fraction* ⇒ *DecimalDigits*]

1 Let *n* be the number of characters in *DecimalDigits*.
2 Return *LexNumber*[*DecimalDigits*]/10^{n}.

LexNumber[*SignedInteger* ⇒ *DecimalDigits*] = *LexNumber*[*DecimalDigits*]

LexNumber[*SignedInteger* ⇒ `+` *DecimalDigits*] = *LexNumber*[*DecimalDigits*]

LexNumber[*SignedInteger* ⇒ `-` *DecimalDigits*] = –*LexNumber*[*DecimalDigits*]

LexNumber[*DecimalDigits* ⇒ *ASCIIDigit*] = *LexNumber*[*ASCIIDigit*]

LexNumber[*DecimalDigits* ⇒ *DecimalDigits*$_1$ *ASCIIDigit*]
= 10**LexNumber*[*DecimalDigits*$_1$] + *LexNumber*[*ASCIIDigit*]

LexNumber[*HexIntegerLiteral* ⇒ `0` *LetterX HexDigit*] = *LexNumber*[*HexDigit*]

LexNumber[*HexIntegerLiteral* ⇒ *HexIntegerLiteral*$_1$ *HexDigit*]
= 16**LexNumber*[*HexIntegerLiteral*$_1$] + *LexNumber*[*HexDigit*]

LexNumber[*ASCIIDigit*]
Return *ASCIIDigit*'s decimal value (a number between 0 and 9).

LexNumber[*NonZeroDigit*]
Return *NonZeroDigit*'s decimal value (a number between 1 and 9).

LexNumber[*HexDigit*]
Return *HexDigit*'s value (a number between 0 and 15). The letters `A`, `B`, `C`, `D`, `E`, and `F`, in either upper or lower case, have values 10, 11, 12, 13, 14, and 15, respectively.

7.8 String literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence starting with a backslash.

Syntax

The grammar parameter θ can be either single or double.

StringLiteral ⇒
  ' *StringChars*^{single} '
| " *StringChars*^{double} "

StringChars$^{\theta}$ ⇒
  «empty»
| *StringChars*$^{\theta}$ *StringChar*$^{\theta}$
| *StringChars*$^{\theta}$ *NullEscape* (*NullEscape*: 7.5)

StringChar$^{\theta}$ ⇒
  LiteralStringChar$^{\theta}$
| \ *StringEscape*

LiteralStringChar^{single} ⇒ *NonTerminator* **except** ' | \ (*NonTerminator*: 7.4)

LiteralStringChar^{double} ⇒ *NonTerminator* **except** " | \

StringEscape ⇒
  ControlEscape
| *ZeroEscape*
| *HexEscape*
| *IdentityEscape*

IdentityEscape ⇒ *NonTerminator* **except** _ | *UnicodeAlphanumeric* (*UnicodeAlphanumeric*: 7.5)

ControlEscape ⇒ b | f | n | r | t | v

ZeroEscape ⇒ 0 [lookahead∉ {*ASCIIDigit*}] (*ASCIIDigit*: 7.7)

HexEscape ⇒
  x *HexDigit HexDigit* (*HexDigit*: 7.7)
| u *HexDigit HexDigit HexDigit HexDigit*

Semantics

Lex[*StringLiteral* ⇒ ' *StringChars*^{single} ']
  Return a **string** token with string contents *LexString*[*StringChars*^{single}].

Lex[*StringLiteral* ⇒ " *StringChars*^{double} "]
  Return a **string** token with string contents *LexString*[*StringChars*^{double}].

LexString[*StringChars*$^{\theta}$ ⇒ «empty»] = ""

LexString[*StringChars*$^{\theta}$ ⇒ *StringChars*$^{\theta}_{1}$ *StringChar*$^{\theta}$]
  Return a string consisting of the string *LexString*[*StringChars*$^{\theta}_{1}$] concatenated with the character *LexChar*[*StringChar*$^{\theta}$].

LexString[*StringChars*$^{\theta}$ ⇒ *StringChars*$^{\theta}_{1}$ *NullEscape*] = *LexString*[*StringChars*$^{\theta}_{1}$]

LexChar[*StringChar*$^{\theta}$ ⇒ *LiteralStringChar*$^{\theta}$]
  Return the character *LiteralStringChar*$^{\theta}$.

LexChar[*StringChar*$^{\theta}$ ⇒ \ *StringEscape*] = *LexChar*[*StringEscape*]

LexChar[*StringEscape* ⇒ *ControlEscape*] = *LexChar*[*ControlEscape*]

LexChar[*StringEscape* ⇒ *ZeroEscape*] = *LexChar*[*ZeroEscape*]

LexChar[*StringEscape* ⇒ *HexEscape*] = *LexChar*[*HexEscape*]

LexChar[*StringEscape* ⇒ *IdentityEscape*]
Return the character *IdentityEscape*.

NOTE A backslash followed by a non-alphanumeric character *c* other than _ or a line break represents character *c*.

LexChar[*ControlEscape* ⇒ b] = '«BS»'

LexChar[*ControlEscape* ⇒ f] = '«FF»'

LexChar[*ControlEscape* ⇒ n] = '«LF»'

LexChar[*ControlEscape* ⇒ r] = '«CR»'

LexChar[*ControlEscape* ⇒ t] = '«TAB»'

LexChar[*ControlEscape* ⇒ v] = '«VT»'

LexChar[*ZeroEscape* ⇒ 0 [lookahead∉ {*ASCIIDigit*}]] = '«NUL»'

LexChar[*HexEscape* ⇒ x $HexDigit_1$ $HexDigit_2$]
1 Let *n* = 16**LexNumber*[$HexDigit_1$] + *LexNumber*[$HexDigit_2$].
2 Return the character with code point value *n*.

LexChar[*HexEscape* ⇒ u $HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$]
1 Let *n* = 4096**LexNumber*[$HexDigit_1$] + 256**LexNumber*[$HexDigit_2$] + 16**LexNumber*[$HexDigit_3$] + *LexNumber*[$HexDigit_4$].
2 Return the character with code point value *n*.

NOTE A *LineTerminator* character cannot appear in a string literal, even if preceded by a backslash \. The correct way to cause a line terminator character to be part of the string value of a string literal is to use an escape sequence such as \n or \u000A.

7.9 Regular expression literals

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegExpBody* and the *RegExpFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it should not extend the *RegExpBody* and *RegExpFlags* productions or the productions used by these productions.

Syntax

RegExpLiteral ⇒ *RegExpBody RegExpFlags*

RegExpFlags ⇒
«empty» (*ContinuingIdentifierCharacterOrEscape*: 7.5)
| *RegExpFlags ContinuingIdentifierCharacterOrEscape*
| *RegExpFlags NullEscape* (*NullEscape*: 7.5)

RegExpBody ⇒ / [lookahead∉ {*}] *RegExpChars* /

RegExpChars ⇒
RegExpChar
| *RegExpChars RegExpChar*

RegExpChar ⇒
OrdinaryRegExpChar
| \ *NonTerminator* (*NonTerminator*: 7.4)

OrdinaryRegExpChar ⇒ *NonTerminator* **except** \ | /

Semantics

Lex[*RegExpLiteral* ⇒ *RegExpBody RegExpFlags*]

Return a **regularExpression** token with the body string *LexString*[*RegExpBody*] and flags string *LexString*[*RegExpFlags*].

LexString[*RegExpFlags* ⇒ «empty»] = ""

LexString[*RegExpFlags* ⇒ *RegExpFlags*$_1$ *ContinuingIdentifierCharacterOrEscape*]

Return a string consisting of the string *LexString*[*RegExpFlags*$_1$] concatenated with the character *LexChar*[*ContinuingIdentifierCharacterOrEscape*].

LexString[*RegExpFlags* ⇒ *RegExpFlags*$_1$ *NullEscape*] = *LexString*[*RegExpFlags*$_1$]

LexString[*RegExpBody* ⇒ / [lookahead∉ {*}] *RegExpChars* /] = *LexString*[*RegExpChars*]

LexString[*RegExpChars* ⇒ *RegExpChar*] = *LexString*[*RegExpChar*]

LexString[*RegExpChars* ⇒ *RegExpChars*$_1$ *RegExpChar*]

Return a string consisting of the string *LexString*[*RegExpChars*$_1$] concatenated with the string *LexString*[*RegExpChar*].

LexString[*RegExpChar* ⇒ *OrdinaryRegExpChar*]

Return a string consisting of the single character *OrdinaryRegExpChar*.

LexString[*RegExpChar* ⇒ \ *NonTerminator*]

Return a string consisting of the two characters '\' and *NonTerminator*.

NOTE A regular expression literal is an input element that is converted to a RegExp object (section **Error! Reference source not found.**) when it is scanned. The object is created before evaluation of the containing program or function begins. Evaluation of the literal produces a reference to that object; it does not create a new object. Two regular expression literals in a program evaluate to regular expression objects that never compare as **===** to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by **new RegExp** (section **Error! Reference source not found.**) or calling the **RegExp** constructor as a function (section **Error! Reference source not found.**).

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters // start a single-line comment. To specify an empty regular expression, use /(?:)/.

23.2 Internationalization

23.3 Units