5.1.2 The Lexical Grammar

A *lexical grammar* for ECMAScript is given in Section 7. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *InputElementDiv* or *InputElementRegExp*, that describe how sequences of Unicode characters are translated into a sequence of input elements.

…

5.1.5 Grammar Notation

…

The subscripted suffix "*opt*", which may appear after a terminal or nonterminal, indicates an *optional symbol*. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

VariableDeclaration **:**
Identifier Initializer$_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**
Identifier
Identifier Initializer

and that:

IterationStatement **:**
for (; *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

is a convenient abbreviation for:

IterationStatement **:**
for (; *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
for (; ; *Expression*$_{opt}$ **)** *Statement*
for (; *Expression* **;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **; ;** *Expression*$_{opt}$ **)** *Statement*
for (*Expression* **;** *Expression* **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
for (; ;) *Statement*
for (; ; *Expression* **)** *Statement*
for (; *Expression* **;)** *Statement*
for (; *Expression* **;** *Expression* **)** *Statement*
for (*Expression* **; ;)** *Statement*
for (*Expression* **; ;** *Expression* **)** *Statement*
for (*Expression* **;** *Expression* **;)** *Statement*
for (*Expression* **;** *Expression* **;** *Expression* **)** *Statement*

so the nonterminal *IterationStatement* actually has eight alternative right-hand sides.

If the phrase "[empty]" appears as the right-hand side of a production, it indicates that the production's right-hand side contains no terminals or nonterminals.

If the phrase "[lookahead ∉ *set*]" appears in the right-hand side of a production, it indicates that the production may not be used if the immediately following input terminal is a member of the given *set*. The *set* can be written as a list of terminals enclosed in curly braces. For convenience, the 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 definitions

DecimalDigit **:: one of**
0 1 2 3 4 5 6 7 8 9

DecimalDigits **::**
DecimalDigit
DecimalDigits DecimalDigit

the definition

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.

If the phrase "[no *LineTerminator* here]" appears in the right-hand side of a production of the syntactic grammar, it indicates that the production is *a restricted production*: it may not be used if a *LineTerminator* occurs in the input stream at the indicated position. For example, the production:

ReturnStatement **:**
return [no *LineTerminator* here] *Expression*$_{opt}$ **;**

indicates that the production may not be used if a *LineTerminator* occurs in the program between the **return** token and the *Expression*.

…

7 Lexical Conventions

The source text of an ECMAScript program is first converted into a sequence of input elements, which are either tokens, line terminators, comments, or white space. The source text is scanned from left to right, repeatedly taking the longest possible sequence of characters as the next input element.

There are two goal symbols for the lexical grammar. The *InputElementDiv* symbol is used in any syntactic grammar context where a division (**/**) or division-assignment (**/=**) operator is permitted. The *InputElementRegExp* symbol is used in any other syntactic grammar context.

Note that contexts exist in the syntactic grammar where both a division and a *RegularExpressionLiteral* are both permitted by the syntactic grammar; however, since the lexical grammar uses the *InputElementDiv* goal symbol in such cases, the opening slash is not recognized as starting a regular expression literal in such a context. As a workaround, one may enclose the regular expression literal in parentheses.

Syntax

InputElementDiv **::**
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp **::**
WhiteSpace
LineTerminator
Comment
Token
RegularExpressionLiteral

7.6 Punctuators

Syntax

Punctuator **:: one of**

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

DivPunctuator **:: one of**

/	/=

7.7.4 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.

Syntax

StringLiteral **::**
" *DoubleStringCharacters*$_{opt}$ "
' *SingleStringCharacters*$_{opt}$ '

DoubleStringCharacters **::**
DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::**
SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **::**
SourceCharacter **but not** *double-quote* **"** **or** *backslash* **** **or** *LineTerminator*
**** *EscapeSequence*

SingleStringCharacter **::**
SourceCharacter **but not** *single-quote* **'** **or** *backslash* **** **or** *LineTerminator*
**** *EscapeSequence*

EscapeSequence **::**
CharacterEscapeSequence
OctalEscapeSequence
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **::**
SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter **::** **one of**
' **"** **** **b** **f** **n** **r** **t** **v**

NonEscapeCharacter **::**
SourceCharacter **but not** *EscapeCharacter* **or** *LineTerminator*

EscapeCharacter **::**
SingleEscapeCharacter
OctalDigit
x
u

HexEscapeSequence **::**
x *HexDigit HexDigit*

OctalEscapeSequence **::**
OctalDigit [lookahead ∉ *OctalDigit*]
ZeroToThree OctalDigit [lookahead ∉ *OctalDigit*]
FourToSeven OctalDigit
ZeroToThree OctalDigit OctalDigit

ZeroToThree **:: one of**
0 **1** **2** **3**

FourToSeven **:: one of**
4 **5** **6** **7**

UnicodeEscapeSequence **::**
u *HexDigit HexDigit HexDigit HexDigit*

The definitions of the nonterminals *HexDigit* and *OctalDigit* are given in section 7.7.3. *SourceCharacter* is described in sections 2 and 6.

~~The above grammar contains an ambiguity where sequences like **\00** can be interpreted either as a one-digit *OctalEscapeSequence* followed by a *SourceCharacter* or as a two-digit *OctalEscapeSequence*. This ambiguity is resolved in favor of the longer *OctalEscapeSequence*. Specifically, we amend the grammar to state that a one-digit *OctalEscapeSequence* expansion applies only if the next character is not an *OctalDigit*. A two-digit *OctalEscapeSequence* expansion whose first digit is between **0** and **3**, inclusive, applies only if the next character is not an *OctalDigit*.~~

A string literal stands for a value of the String type. The string value (SV) of the literal is described in terms of character values (CV) contributed by the various parts of the string literal. As part of this process, some characters within the string literal are interpreted as having a mathematical value (MV), as described below or in section 7.7.3.

- The SV of *StringLiteral* **:: ""** is the empty character sequence.
- The SV of *StringLiteral* **:: ''** is the empty character sequence.
- The SV of *StringLiteral* **::** **"** *DoubleStringCharacters* **"** is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral* **::** **'** *SingleStringCharacters* **'** is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* **::** *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* **::** *DoubleStringCharacter DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter SingleStringCharacters* is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The CV of *DoubleStringCharacter* **::** *SourceCharacter* **but not** *double-quote* **"** **or** *backslash* **** **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *DoubleStringCharacter* **::** **** *EscapeSequence* is the CV of the *EscapeSequence*.

- The CV of *SingleStringCharacter* **::** *SourceCharacter* **but not** *single-quote* **'** **or** *backslash* **** **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *SingleStringCharacter* **::** **** *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *EscapeSequence* **::** *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* **::** *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *EscapeSequence* **::** *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence* **::** *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *CharacterEscapeSequence* **::** *SingleEscapeCharacter* is the Unicode character whose Unicode value is determined by the *SingleEscapeCharacter* according to the following table:

Escape Sequence	*Unicode Value*	*Name*	*Symbol*
\b	**\u0008**	backspace	<BS>
\t	**\u0009**	horizontal tab	<HT>
\n	**\u000A**	line feed (new line)	<LF>
\v	**\u000B**	vertical tab	<VT>
\f	**\u000C**	form feed	<FF>
\r	**\u000D**	carriage return	<CR>
\"	**\u0022**	double quote	**"**
\'	**\u0027**	single quote	**'**
****	**\u005C**	backslash	****

- The CV of *CharacterEscapeSequence* **::** *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* **::** *SourceCharacter* **but not** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* **::** **x** *HexDigit HexDigit* is the Unicode character whose code is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *OctalEscapeSequence* **::** *OctalDigit* is the Unicode character whose code is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** *ZeroToThree OctalDigit* is the Unicode character whose code is (8 times the MV of the *ZeroToThree*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** *FourToSeven OctalDigit* is the Unicode character whose code is (8 times the MV of the *FourToSeven*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** *ZeroToThree OctalDigit OctalDigit* is the Unicode character whose code is (64 (that is, 8^2) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree* **::** **0** is 0.
- The MV of *ZeroToThree* **::** **1** is 1.
- The MV of *ZeroToThree* **::** **2** is 2.
- The MV of *ZeroToThree* **::** **3** is 3.
- The MV of *FourToSeven* **::** **4** is 4.
- The MV of *FourToSeven* **::** **5** is 5.
- The MV of *FourToSeven* **::** **6** is 6.
- The MV of *FourToSeven* **::** **7** is 7.
- The CV of *UnicodeEscapeSequence* **::** **u** *HexDigit HexDigit HexDigit HexDigit* is the Unicode character whose code is (4096 (that is, 16^3) times the MV of the first *HexDigit*) plus (256 (that is, 16^2) times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

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**.

7.7.5 Regular Expression Literals

A regular expression literal is an input element that becomes converted an object of type **RegExp** when it is scanned. The object is created before the 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 **===** even if the two literals' contents are identical. A **RegExp** object may also be created at runtime by calling the **RegExp** constructor as a function.

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 comprised of the *RegularExpressionBody* and the *RegularExpressionFlags* 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 *RegularExpressionBody* and *RegularExpressionFlags* productions or the productions used by these productions.

Note that regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters `//` start a single-line comment.

Syntax

RegularExpressionLiteral **::**
/ RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody **::**
RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars **::**
[empty]
RegularExpressionChars RegularExpressionChar

RegularExpressionFirstChar **::**
NonTerminator **but not * or \ or /**
BackslashSequence

RegularExpressionChar **::**
NonTerminator **but not \ or /**
BackslashSequence

BackslashSequence **::**
**** *NonTerminator*

NonTerminator **::**
SourceCharacter **but not** *LineTerminator*

RegularExpressionFlags **::**
[empty]
RegularExpressionFlags IdentifierPart

Semantics

A regular expression literal stands for a value of the Object type. This value is determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpreted into two strings Pattern and Flags, respectively. Then the **`new RegExp`** constructor is called with two arguments Pattern and Flags and the result becomes the value of the *RegularExpressionLiteral*. If the call to **`new RegExp`** generates an error, an implementation may, at its discretion, either report the error immediately while scanning the program, or it may defer the error until the regular expression literal is evaluated in the course of program execution.

8.6.2 Internal Properties and Methods

Internal properties and methods are not exposed in the language. For the purposes of this document, their names are enclosed in double square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a runtime error is generated.

There are two types of access for exposed properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

Native ECMAScript objects have an internal property called [[Prototype]]. The value of this property is either **null** or an object and is used for implementing inheritance. Properties of the [[Prototype]] object are exposed as properties of the child object for the purposes of get access, but not for put access.

The following table summarises the internal properties used by this specification. The description indicates their behaviour for native ECMAScript objects. Host objects may implement these internal methods with any implementation-dependent behaviour, or it may be that a host object implements only some internal methods and not others.

Property	***Parameters***	***Description***
[[Prototype]]	none	The prototype of this object.
[[Class]]	none	A string value indicating the kind of this object.
[[Value]]	none	Internal state information associated with this object.
[[Get]]	(PropertyName)	Returns the value of the property.
[[Put]]	(PropertyName, Value)	Sets the specified property to Value.
[[CanPut]]	(PropertyName)	Returns a boolean value indicating whether a [[Put]] operation with the specified PropertyName will succeed.
[[HasProperty]]	(PropertyName)	Returns a boolean value indicating whether the object already has a member with the given name.
[[Delete]]	(PropertyName)	Removes the specified property from the object.
[[DefaultValue]]	(Hint)	Returns a default value for the object, which should be a primitive value (not an object or reference).
[[Construct]]	a list of argument values provided by the caller	Constructs an object. Invoked via the **new** operator. Objects that implement this internal method are called *constructors*.
[[Call]]	a list of argument values provided by the caller	Executes code associated with the object. Invoked via a function call expression. Objects that implement this internal method are called *functions*.
[[HasInstance]]	(Value)	Returns a boolean value indicating whether the Value delegates behaviour to this object. Of the native ECMAScript objects, only Function objects implement [[HasInstance]].
[[Closure]]	none	A scope chain that defines the environment in which a Function object is executed.
[[Match]]	(string, integer)	Tests for a regular expression match and returns a MatchResult value (see section 15.10.2.1).

Every object must implement the [[Class]] property and the [[Get]], [[Put]], [[HasProperty]], [[Delete]], and [[DefaultValue]] methods, even host objects. (Note, however, that the [[DefaultValue]] method may, for some objects, simply generate a runtime error.)

12.4 Expression Statement

Syntax

ExpressionStatement **:**
[lookahead ∉ {**{**}] *Expression* **;**

Note that an *ExpressionStatement* cannot start with an opening curly brace because that might make it ambiguous with a *Block*:

Semantics

The production *ExpressionStatement* **:** *Expression* **;** is evaluated as follows:

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Return (**normal**, Result(2), **empty**).

15.1.2.1 eval (x)

When the **eval** function is called with one argument *x*, the following steps are taken:

1. If *x* is not a string value, return *x*.
2. Parse *x* as an ECMAScript *Program*. If the parse fails, generate a runtime error.
3. Evaluate the program from step 2.
4. If the completion type of Result(3) is "normal" and its completion value is a value *V*, then return the value *V*.
5. If the completion type of Result(3) is "normal" and its completion value is **empty**, then return the value **undefined**.
6. The completion type of Result(3) must be "throw". Throw its completion value as an exception.

If value of the **eval** property is used in any way other than a direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **eval** property is assigned to, a runtime error may be generated.

15.1.3.8 RegExp (. . .)

See section 15.10.3.

15.5.4.10 String.prototype.split (separator [, limit])

Returns an Array object into which substrings of the result of converting this object to a string have been stored. The substrings are determined by searching from left to right for occurrences of the given separator; these occurrences are not part of any substring in the returned array, but serve to divide up the string value. The separator may be a string of any length or it may be an object of type **RegExp**.

The separator may be an empty string, an empty regular expression, or a regular expression which can match an empty string. Such a separator does not match the empty substring at the beginning or end of the input string, nor does it match the empty substring at the end of the previous separator match. (For example, if the separator is the empty string, the string is split up into individual characters; the length of the result array equals the length of the string, and each substring contains one character.) If the separator is a regular expression, only the first match at a given position of the this string is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, **"ab".split(/a*?/)** evaluates to the array **["a","b"]**, while **"ab".split(/a*/)** evaluates to the array **["","b"]**.)

If the **this** string is empty, the result depends on whether the separator can match the empty string. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty string.

If the separator is a regular expression that contains capturing parentheses, then each time the separator is matched the results (including any **undefined** results) of the capturing parentheses are spliced into the output array. (For example, **"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/)** evaluates to the array **["A", undefined, "B", "bold", "/", "B", "and", undefined, "CODE", "coded", "/", "CODE", ""]**.)

If the separator is not supplied, then the result array contains just one string, which is the **this** string.

If *limit* is supplied, then the output array is truncated so that it contains no more than *limit* elements.

When the **split** method is called, the following steps are taken:

1. Let *S* = ToString(**this**).
2. Let *A* be a new array created as if by the expression **new Array()**.
3. If *limit* is **undefined** or not supplied, let *lim* = 2^{32}-1; else let *lim* = ToUint32(*limit*).
4. Let *s* be the number of characters in *S*.
5. Let *p* = 0.
6. If *separator* is a RegExp object, let *R* = *separator*; otherwise let *R* = ToString(*separator*).
7. If *lim* = 0, return *A*.
8. If *separator* is **undefined** or not supplied, go to step 33.
9. If *s* = 0, go to step 31.
10. Let *q* = *p*.
11. If *q* = *s*, go to step 28.

12. Call *SplitMatch*(*R*, *S*, *q*) and let *z* be its MatchResult result.
13. If *z* is **failure**, go to step 26.
14. *z* must be a State. Let *e* be *z*'s *endIndex* and let *cap* be *z*'s *captures* array.
15. If *e* = *p*, go to step 26.
16. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *q* (exclusive).
17. Call the [[Put]] method of *A* with arguments *A*`.length` and *T*.
18. If *A*`.length` = *lim*, return *A*.
19. Let *p* = *e*.
20. Let *i* = 0.
21. If *i* is equal to the number of elements in *cap*, go to step 10.
22. Let *i* = *i*+1.
23. Call the [[Put]] method of *A* with arguments *A*`.length` and *cap*[*i*].
24. If *A*`.length` = *lim*, return *A*.
25. Go to step 21.
26. Let *q* = *q*+1.
27. Go to step 11.
28. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through s (exclusive).
29. Call the [[Put]] method of *A* with arguments *A*`.length` and *T*.
30. Return *A*.
31. Call *SplitMatch*(*R*, *S*, 0) and let *z* be its MatchResult result.
32. If *z* is not **failure**, return *A*.
33. Call the [[Put]] method of *A* with arguments `"0"` and *S*.
34. Return *A*.

The internal helper function *SplitMatch* takes three parameters, a string *S*, an integer *q*, and a string or RegExp *R*, and performs the following in order to return a MatchResult (see section 15.10.2.1):

1. If *R* is a RegExp object, go to step 8.
2. *R* must be a string. Let *r* be the number of characters in *R*.
3. Let *s* be the number of characters in *S*.
4. If *q*+*r* > *s* then return the MatchResult **failure**.
5. If there exists an integer *i* between 0 (inclusive) and *r* (exclusive) such that the character at position *q*+*i* of *S* is different from the character at position *i* of *R*, then return **failure**.
6. Let *cap* be an empty array of captures (see section 15.10.2.1).
7. Return the State (*q*+*r*, *cap*). (see section 15.10.2.1)
8. Call the [[Match]] method of *R* giving it the arguments *S* and *q*, and return the MatchResult result.

NOTE The `split` function is intentionally generic; it does not require that its `this` value be a string object. Therefore, it can be transferred to other kinds of objects for use as a method.

Note that the `split` method ignores the value of *separator*.global for separators that are RegExp objects.

15.5.4.14 String.prototype.match(regexp)

If *regexp* is not an object of type `RegExp`, it is replaced with the result of the expression `new RegExp(regexp)`. Let *string* denote the result of converting the `this` value to a string.

If *regexp*.global is `false`, return the result obtained by invoking `RegExp.prototype.exec` (see section 15.10.6.2) on *regexp* with *string* as parameter.

Otherwise, set the *regexp*.`lastIndex` property to 0 and invoke `RegExp.prototype.exec` repeatedly until there is no match. If there is a match with an empty string (in other words, if the value of *regexp*.`lastIndex` is left unchanged) increment *regexp*.`lastIndex` by 1. The value returned is an array with the properties 0 through *n*-1 corresponding to the first element of the result of each matching invocation of `RegExp.prototype.exec`.

Note that the `match` function is intentionally generic; it does not require that its `this` value be a string object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.15 String.prototype.replace(regexp, replaceValue)

If *regexp* is not an object of type **RegExp,** it is replaced with the result of the expression **new RegExp(regexp)**. Let *string* denote the result of converting the **this** value to a string.

if *regexp*.global is **false**, then *String* is searched for the first occurrence of the regular expression pattern *regexp*. If *regexp*.global is **true**, then *String* is searched for all occurrences of the regular expression pattern *regexp*. The search is done in the same manner as in **String.prototype.match**, including the update of *regexp*.**lastIndex**.

Let *m* be the number of left capturing parentheses in *regexp* (*NCapturingParens* as specified in section 15.10.2.1).

If **replaceValue** is a function, then for each matched substring, call the function with the following *m* + 3 arguments. Argument 1 is the substring that matched. The next *m* arguments are all of the captures in the MatchResult (see section 15.10.2.1). Argument *m* + 2 is the offset within *string* where the match occurred, and argument *m* + 3 is *string*. The result is a string value derived from the original input by replacing each matched substring with the corresponding return value of the function call, converted to a string if need be.

Otherwise, let *newstring* denote the result of converting **replaceValue** to a string. The result is a string value derived from the original input string by replacing each matched substring with a string derived from *newstring* by replacing characters in *newstring* by replacement text as specified in the following table. These **$** replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, **"$1,$2".replace(/(\$(\d))/g, "$$1-$1$2")** returns **"$1-$11,$1-$22"**. A **$** in *newstring* that does not match any of the forms below is left as is.

Characters	*Replacement text*
$$	**$**
$&	The matched substring.
$'	The portion of *string* that precedes the matched substring.
$'	The portion of *string* that follows the matched substring.
$*n*	The *n*th capture, where *n* is a single digit 1-9. If *n* *m* and the *n*th capture is **undefined**, use the empty string instead. If *n*>*m*, the result is implementation-defined.
$*nn*	The *nn*th capture, where *nn* is a two-digit decimal number 01-99. If *nn* *m* and the *nn*th capture is **undefined**, use the empty string instead. If *nn*>*m*, the result is implementation-defined.

Note that the **replace** function is intentionally generic; it does not require that its **this** value be a string object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.16 String.prototype.search(regexp)

If *regexp* is not an object of type **RegExp,** it is replaced with result of the expression **new RegExp(regexp)**. Let *string* denote the result of converting the **this** value to a string.

String is searched from its beginning for an occurrence of the regular expression pattern *regexp*. The result is a number indicating the offset within the string where the pattern matched, or -1 if there was no match.

Note that this method ignores the **lastIndex** and **global** properties of regexp, and that the **lastIndex** property of *regexp* is left unchanged.

Note that the **search** function is intentionally generic; it does not require that its **this** value be a string object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.10 RegExp (Regular Expression) Objects

A RegExp object contains a regular expression and the associated flags. The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

15.10.1 Patterns

The **RegExp** constructor applies the following grammar to the input pattern string. An error occurs if the grammar cannot interpret the string as an expansion of *Pattern*.

Syntax

Pattern **::**
Disjunction

Disjunction **::**
Alternative
Alternative **|** *Disjunction*

Alternative **::**
[empty]
Alternative Term

Term **::**
Assertion
Atom
Atom Quantifier

Assertion **::**
^
$
\ b
\ B

Quantifier **::**
QuantifierPrefix
QuantifierPrefix **?**

QuantifierPrefix **::**
*****
+
?
{ *DecimalDigits* **}**
{ *DecimalDigits* **, }**
{ *DecimalDigits* **,** *DecimalDigits* **}**

Atom **::**
PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(? : *Disjunction* **)**
(? = *Disjunction* **)**
(? ! *Disjunction* **)**

PatternCharacter **::**
SourceCharacter **but not any of:**
^ $ \ . * + ? ( ) [ ] { } |

AtomEscape **::**
DecimalOrOctalEscape
CharacterEscape
CharacterClassEscape

CharacterEscape **::**
ControlEscape
c *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

ControlEscape **:: one of**
f n r t v

ControlLetter **:: one of**
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

IdentityEscape **::**
SourceCharacter **but not** *IdentifierPart*

DecimalOrOctalEscape **::**
DecimalDigit [lookahead ∉ *DecimalDigit*]
ZeroToThree OctalDigit [lookahead ∉ *OctalDigit*]
ZeroToThree EightOrNine
FourToNine DecimalDigit
ZeroToThree OctalDigit OctalDigit

EightOrNine **:: one of**
8 9

FourToNine **:: one of**
4 5 6 7 8 9

CharacterClassEscape **:: one of**
d D s S w W

CharacterClass **::**
[[lookahead ∉ {**^**}] *ClassRanges* **]**
[^ *ClassRanges* **]**

ClassRanges **::**
[empty]
NonemptyClassRanges

NonemptyClassRanges **::**
ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom **-** *ClassAtom ClassRanges*

NonemptyClassRangesNoDash **::**
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash **-** *ClassAtom ClassRanges*

ClassAtom **::**
-
ClassAtomNoDash

ClassAtomNoDash **::**
SourceCharacter **but not one of \ ] -**
**** *ClassEscape*

ClassEscape **::**
DecimalOrOctalEscape
b
CharacterEscape
CharacterClassEscape

15.10.2 Pattern Semantics

A regular expression pattern is converted into an internal function using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same.

15.10.2.1 Notation

The descriptions below use the following variables:

- *Input* is the string being matched by the regular expression pattern. The notation *input*[*n*] means the *n*th character of *input*, where *n* can range between 0 (inclusive) and *InputLength* (exclusive).
- *InputLength* is the number of characters in the *Input* string.
- *NCapturingParens* is the total number of left capturing parentheses (i.e. the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded) in the pattern. A left capturing parenthesis is any **(** pattern character that is matched by the **(** terminal of the *Atom* :: **(** *Disjunction* **)** production.
- *IgnoreCase* is the setting of the RegExp object's **`ignoreCase`** property.
- *Multiline* is the setting of the RegExp object's **`multiline`** property.

Furthermore, the descriptions below use the following internal data structures:

- A *CharSet* is a mathematical set of characters.
- A *State* is an ordered pair (*endIndex*, *captures*) where *endIndex* is an integer and *captures* is an internal array of *NCapturingParens* values. States are used to represent partial match states in the regular expression matching algorithms. The *endIndex* is one plus the index of the last input character matched so far by the pattern, while *captures* holds the results of capturing parentheses. The *n*th element of *captures* is either a string that represents the value obtained by the *n*th set of capturing parentheses or **undefined** if the *n*th set of capturing parentheses hasn't been reached yet. Due to backtracking, many states may be in use at any time during the matching process.
- A *MatchResult* is either a State or the special token **failure** that indicates that the match failed.
- A *Continuation* function is an internal closure (i.e. an internal function with some arguments already bound to values) that takes one State argument and returns a MatchResult result. If an internal closure references variables bound in the function that creates the closure, the closure uses the values that these variables had at the time the closure was created. The continuation attempts to match the remaining portion (specified by the closure's already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. If the match succeeds, the continuation returns the final State that it reached; if the match fails, the continuation returns **failure**.
- A *Matcher* function is an internal closure that takes two arguments -- a State and a Continuation -- and returns a MatchResult result. The matcher attempts to match a middle subpattern (specified by the closure's already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. The Continuation argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new State, the matcher then calls Continuation on that state to test if the rest of the pattern can match as well. If it can, the matcher returns the state returned by the continuation; if not, the matcher may try different choices at its choice points, repeatedly calling Continuation until it either succeeds or all possibilities have been exhausted.
- An *AssertionTester* function is an internal closure that takes a State argument and returns a boolean result. The assertion tester tests a specific condition (specified by the closure's already-bound arguments) against the current place in the input string and returns **true** if the condition matched or **false** if not.
- An *EscapeValue* is either a character or an integer. An EscapeValue is used to denote the interpretation of a *DecimalOrOctalEscape* escape sequence: a character *ch* means that the escape sequence is interpreted as the character *ch*, while an integer *n* means that the escape sequence is interpreted as a backreference to the *n*th set of capturing parentheses.

15.10.2.2 Pattern

The production *Pattern* **::** *Disjunction* evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal closure that takes two arguments, a string *str* and an integer *index*, and performs the following:
    1. Let *Input* be the given string *str*. This variable will be used throughout the functions in section 15.10.2.
    2. Let *InputLength* be the length of *Input*. This variable will be used throughout the functions in section 15.10.2.
    3. Let *c* be a Continuation that always returns its State argument as a successful MatchResult.
    4. Let *cap* be an internal array of *NCapturingParens* **undefined** values, indexed 1 through *NCapturingParens*.
    5. Let *x* be the State (*index*, *cap*).
    6. Call *m*(*x*, *c*) and return its result.

Informative comments: A *Pattern* evaluates ("compiles") to an internal function value. **`RegExp.prototype.exec`** can then apply this function to a string and an offset within the string to determine whether the pattern would match starting at exactly that offset within the string, and, if it does match, what the values of the capturing parentheses would be. The algorithms in section 15.10.2 are designed so that compiling a pattern may throw a RegExpError exception; on the other hand, once the pattern is successfully compiled, applying its result function to find a match in a string cannot throw an exception (except for exceptions that can occur anywhere such as out-of-memory).

15.10.2.3 Disjunction

The production *Disjunction* **::** *Alternative* evaluates by evaluating *Alternative* to obtain a Matcher and returning that Matcher.

The production *Disjunction* **::** *Alternative* **|** *Disjunction* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher *m1*.
2. Evaluate *Disjunction* to obtain a Matcher *m2*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Call *m1*(*x*, *c*) and let *r* be its result.
    2. If *r* isn't **failure**, return *r*.
    3. Call *m2*(*x*, *c*) and return its result.

Informative comments: The **|** regular expression operator separates two alternatives. The pattern first tries to match the left *Alternative* (followed by the sequel of the regular expression); if it fails, it tries to match the right *Disjunction* (followed by the sequel of the regular expression). If the left *Alternative*, the right *Disjunction*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*. Any capturing parentheses inside a portion of the pattern skipped by **|** produce **`undefined`** values instead of strings. Thus, for example,

```
/a|ab/.exec("abc")
```

returns the result **`"a"`** and not **`"ab"`**. Moreover,

```
/((a)|(ab))((c)|(bc))/.exec("abc")
```

returns the array

```
["abc", "a", "a", undefined, "bc", undefined, "bc"]
```

and not

```
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

15.10.2.4 Alternative

The production *Alternative* **::** [empty] evaluates by returning a Matcher that takes two arguments, a State *x* and a Continuation *c*, and returns the result of calling *c*(*x*).

The production *Alternative* **::** *Alternative Term* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher *m1*.
2. Evaluate *Term* to obtain a Matcher *m2*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Create a Continuation *d* that takes a State argument *y* and returns the result of calling *m2*(*y*, *c*).
    2. Call *m1*(*x*, *d*) and return its result.

Informative comments: Consecutive *Terms* try to simultaneously match consecutive portions of the input string. If the left *Alternative*, the right *Term*, and the sequel of the regular expression all have choice points, all choices in the sequel are tried before moving on to the next choice in the right *Term*, and all choices in the right *Term* are tried before moving on to the next choice in the left *Alternative*.

15.10.2.5 Term

The production *Term* **::** *Assertion* evaluates by returning an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

1. Evaluate *Assertion* to obtain an AssertionTester *t*.
2. Call *t*(*x*) and let *r* be the resulting boolean value.
3. If *r* is **false**, return **failure**.
4. Call *c*(*x*) and return its result.

The production *Term* **::** *Atom* evaluates by evaluating *Atom* to obtain a Matcher and returning that Matcher.

The production *Term* **::** *Atom Quantifier* evaluates as follows:

1. Evaluate *Atom* to obtain a Matcher *m*.
2. Evaluate *Quantifier* to obtain the three results: an integer *min*, an integer (or) *max*, and boolean *greedy*.
3. If *max* is finite and less than *min*, then throw a RegExpError exception.
4. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's *Term*. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this production's *Term* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *Term*.
5. Let *parenCount* be the number of left capturing parentheses in the expansion of this production's *Atom*. This is the total number of *Atom* :: **(** *Disjunction* **)** productions enclosed by this production's *Atom*.
6. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Call *RepeatMatcher*(*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*) and return its result.

The internal helper function *RepeatMatcher* takes eight parameters, a Matcher *m*, an integer *min*, an integer (or) *max*, a boolean *greedy*, a State *x*, a Continuation *c*, an integer *parenIndex*, and an integer *parenCount*, and performs the following:

1. If *max* is zero, then call *c*(*x*) and return its result.
2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
    1. If *min* is zero, and *max* is , and *y*'s *endIndex* is equal to *x*'s *endIndex*, then return **failure**.
    2. If *min* is zero, then let *min2* be zero; otherwise let *min2* be *min*–1.
    3. If *max* is , then let *max2* be ; otherwise let *max2* be *max*–1.
    4. Call *RepeatMatcher*(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*) and return its result.
3. Let *cap* be a fresh copy of *x*'s *captures* internal array.
4. For every integer *k* that satisfies *parenIndex* < *k* and *k* *parenIndex*+*parenCount*, set *cap*[*k*] to **undefined**.
5. Let *e* be *x*'s *endIndex*.
6. Let *xr* be the State (*e*, *cap*).
7. If *min* is not zero, then call *m*(*xr*, *d*) and return its result.
8. If *greedy* is **true**, then go to step 12.

9. Call *c*(*x*) and let *z* be its result.
10. If *z* is not **failure**, return *z*.
11. Call *m*(*xr*, *d*) and return its result.
12. Call *m*(*xr*, *d*) and let *z* be its result.
13. If *z* is not **failure**, return *z*.
14. Call *c*(*x*) and return its result.

Informative comments: An *Atom* followed by a *Quantifier* is repeated the number of times specified by the *Quantifier*. A quantifier can be non-greedy, in which case the *Atom* pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the *Atom* pattern is repeated as many times as possible while still matching the sequel. The *Atom* pattern is repeated rather than the input string which it matches, so different repetitions of the Atom can match different input substrings.

If the *Atom* and the sequel of the regular expression all have choice points, the *Atom* is first matched as many (or as few, if non-greedy) times as possible. All choices in the sequel are tried before moving on to the next choice in the last repetition of *Atom*. All choices in the last (*n*th) repetition of *Atom* are tried before moving on to the next choice in the next-to-last (*n*-1st) repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the *n*-1st repetition of *Atom* and so on.

Compare

```
/a[a-z]{2,4}/.exec("abcdefghi")
```

which returns **"abcde"** with

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

which returns **"abc"**.

Consider also

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

which, by the choice point ordering above, returns the array

```
["aaba", "ba"]
```

and not any of:

```
["aabaac", "aabaac"]
```

```
["aabaac", "c"]
```

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/,"$1")
```

which returns the gcd in unary notation **"aaaaa"**.

Step 4 of the *RepeatMatcher* clears *Atom*'s captures each time *Atom* is repeated. We can see its behavior in the regular expression

```
/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

which returns the array

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost ***** clears all captured strings contained in the quantified *Atom*, which in this case includes capture strings numbered 2, 3, and 4.

Step 1 of the *RepeatMatcher*'s closure *d* states that, once the minimum number of repetitions has been satisfied, any more expansions of *Atom* that match the empty string are not considered for further repetitions if the maximum number of repetitions is infinity. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```
/(a*)*/.exec("b")
```

or the slightly more complicated:

```
/(a*)b\1+/.exec("baaaac")
```

which returns the array

```
["b", ""]
```

15.10.2.6 Assertion

The production *Assertion* **::** **^** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is zero, return **true**.
3. If *Multiline* is **false**, return **false**.
4. If the character *Input*[*e*–1] is one of the line terminator characters <LF>, <CR>, <LS>, or <PS>, return **true**.
5. Return **false**.

The production *Assertion* **::** **$** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is equal to *InputLength*, return **true**.
3. If *multiline* is **false**, return **false**.
4. If the character *Input*[*e*] is one of the line terminator characters <LF>, <CR>, <LS>, or <PS>, return **true**.
5. Return **false**.

The production *Assertion* **::** **\ b** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. Call *IsWordChar*(*e*–1) and let *a* be the boolean result.
3. Call *IsWordChar*(*e*) and let *b* be the boolean result.
4. If *a* is **true** and *b* is **false**, return **true**.
5. If *a* is **false** and *b* is **true**, return **true**.
6. Return **false**.

The production *Assertion* **::** **\ B** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. Call *IsWordChar*(*e*–1) and let *a* be the boolean result.
3. Call *IsWordChar*(*e*) and let b be the boolean result.
4. If a is **true** and b is **false**, return **false**.
5. If a is **false** and b is **true**, return **false**.
6. Return **true**.

The internal helper function *IsWordChar* takes an integer parameter *e* and performs the following:

1. If *e*=–1 or *e*=*InputLength*, return **false**.
2. Let *c* be the character *Input*[*e*].
3. If *c* is one of the sixty-three Unicode characters in the table below, return **true**.
4. Return **false**.

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9 _

15.10.2.7 Quantifier

The production *Quantifier* :: *QuantifierPrefix* evaluates as follows:

1. Evaluate *QuantifierPrefix* to obtain the two results: an integer *min* and an integer (or) *max*.
2. Return the three results *min* , *max*, and **true**.

The production *Quantifier* **::** *QuantifierPrefix* **?** evaluates as follows:

1. Evaluate *QuantifierPrefix* to obtain the two results: an integer *min* and an integer (or) *max*.
2. Return the three results *min* , *max*, and **false**.

The production *QuantifierPrefix* **:: *** evaluates by returning the two results 0 and .

The production *QuantifierPrefix* **:: +** evaluates by returning the two results 1 and .

The production *QuantifierPrefix* **:: ?** evaluates by returning the two results 0 and 1.

The production *QuantifierPrefix* **:: {** *DecimalDigits* **}** evaluates as follows:

1. Let *i* be the MV of *DecimalDigits* (see section 7.7.3).
2. Return the two results *i* and *i*.

The production *QuantifierPrefix* **:: {** *DecimalDigits* **, }** evaluates as follows:

1. Let *i* be the MV of *DecimalDigits*.
2. Return the two results *i* and .

The production *QuantifierPrefix* **:: {** *DecimalDigits* **,** *DecimalDigits* **}** evaluates as follows:

1. Let *i* be the MV of the first *DecimalDigits*.
2. Let *j* be the MV of the second *DecimalDigits*.
3. Return the two results *i* and *j*.

15.10.2.8 Atom

The production *Atom* **::** *PatternCharacter* evaluates as follows:

1. Let *ch* be the character represented by *PatternCharacter*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* **::** **.** evaluates as follows:

1. Let *A* be the set of all characters except the four line terminator characters <LF>, <CR>, <LS>, or <PS>.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* **::** **** *AtomEscape* evaluates by evaluating *AtomEscape* to obtain a Matcher and returning that Matcher.

The production *Atom* **::** *CharacterClass* evaluates as follows:

1. Evaluate *CharacterClass* to obtain a CharSet *A* and a boolean *invert*.
2. Call *CharacterSetMatcher*(*A*, *invert*) and return its Matcher result.

The production *Atom* **::** **(** *Disjunction* **)** evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this production's *Atom* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *Atom*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
        1. Let *cap* be a fresh copy of *y*'s *captures* internal array.
        2. Let *xe* be *x*'s *endIndex*.
        3. Let *ye* be *y*'s *endIndex*.
        4. Let *s* be a fresh string whose characters are the characters of *Input* at positions *xe* (inclusive) through *ye* (exclusive).
        5. Set *cap*[*parenIndex*+1] to *s*.
        6. Let *z* be the State (*ye*, *cap*).
        7. Call *c*(*z*) and return its result.
    2. Call *m*(*x*, *d*) and return its result.

The production *Atom* **::** **(** **?** **:** *Disjunction* **)** evaluates by evaluating *Disjunction* to obtain a Matcher and returning that Matcher.

The production *Atom* **::** **(** **?** **=** *Disjunction* **)** evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
        1. Let *cap* be *y*'s *captures* internal array.
        2. Let *xe* be *x*'s *endIndex*.
        3. Let *z* be the State (*xe*, *cap*).
        4. Call *c*(*z*) and return its result.
    2. Call *m*(*x*, *d*) and return its result.

The production *Atom* **::** **(** **?** **!** *Disjunction* **)** evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
    2. Call *m*(*x*, *d*) and let *r* be its result.
    3. If *r* isn't **failure**, return **failure**.
    4. Call *c*(*x*) and return its result.

The internal helper function *CharacterSetMatcher* takes two arguments, a CharSet *A* and a boolean flag *invert*, and performs the following:

1. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Let *e* be *x*'s *endIndex*.
    2. If *e*=*InputLength*, return **failure**.
    3. Let *c* be the character *Input*[*e*].
    4. Let *cc* be the result of *Canonicalize*(*c*).
    5. If *invert* is **true**, go to step 8.
    6. If there does not exist a member *a* of set *A* such that *Canonicalize*(*a*) == *cc*, then return **failure**.
    7. Go to step 9.
    8. If there exists a member *a* of set *A* such that *Canonicalize*(*a*) == *cc*, then return **failure**.
    9. Let *cap* be *x*'s *captures* internal array.

10. Let *y* be the State (*e*+1, *cap*).
11. Call *c*(*y*) and return its result.

The internal helper function *Canonicalize* takes a character parameter *ch* and performs the following:

1. If *IgnoreCase* is **false**, return *ch*.
2. Let *u* be *ch* converted to upper case as if by calling **`String.prototype.toUpperCase`** on the one-character string *ch*.
3. If *u* does not consist of a single character, return *ch*.
4. Let *cu* be *u*'s character.
5. If *ch*'s Unicode value is greater than or equal to decimal 128 and *cu*'s Unicode value is less than decimal 128, then return *ch*.
6. Return *cu*.

Informative comments: Parentheses of the form **(** *Disjunction* **)** serve both to group the components of the *Disjunction* pattern together and to save the result of the match. The result can be used either in a backreference (\ followed by a one- or two-digit decimal number), referenced in a replace string, or returned as part of an array from the regular expression matching function. To inhibit the capturing behavior of parentheses, use the form **(?:** *Disjunction* **)** instead.

The form **(?=** *Disjunction* **)** specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside *Disjunction* must match at the current position, but the current position is not advanced before matching the sequel. If *Disjunction* can match at the current position in several ways, backtracking can cause all of these ways to be tried. This only matters when the *Disjunction* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures; in other cases backtracking into a zero-width positive lookahead *Disjunction* can be turned off without affecting the semantics of the regular expression.

For example,

```
/(?=(a+))/.exec("baaabac")
```

matches the empty string immediately after the first **b** and therefore returns the array:

```
["", "aaa"]
```

To illustrate backtracking into the lookahead, consider:

```
/(?=(a+))a*b\1/.exec("baaabac")
```

This expression returns

```
["aaaba", "a"]
```

and not:

```
["aba", "a"]
```

The form **(?!** *Disjunction* **)** specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside *Disjunction* must fail to match at the current position. The current position is not advanced before matching the sequel. *Disjunction* can contain capturing parentheses, but backreferences to them only make sense from within *Disjunction* inself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **undefined** because the negative lookahead must fail for the pattern to succeed. For example,

```
/(.*?)a(?!(a+)b\2c)\2(.*)/.exec("baaabaac")
```

looks for an **a** not immediately followed by some positive number *n* of **a**'s, a **b**, another *n* **a**'s (specified by the first **\2**) and a **c**. The second **\2** is outside the negative lookahead, so it matches against **undefined** and therefore always succeeds. The whole expression returns the array:

```
["baaabaac", "ba", undefined, "abaac"]
```

In case-insignificant matches all characters are implicitly converted to upper case immediately before they are compared. However, if converting a character to upper case would expand that character into more than one character (such as converting **"ß"** (\u00DF) into **"SS"**), then the character is left as-is instead. The character is also left as-is if it is not an ASCII character but converting it to upper case would make it into an ASCII character. This prevents Unicode characters such as \u0131 and \u017F from matching regular expressions such as **/[a-z]/i**, which are only intended to match ASCII letters. Furthermore, if these conversions were allowed, then **/[^\W]/i** would match each of **a**, **b**, …, **h**, but not **i** or **s**.

15.10.2.9 AtomEscape

The production *AtomEscape* **::** *DecimalOrOctalEscape* evaluates as follows:

1. Evaluate *DecimalOrOctalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then go to step 6.
3. Let *ch* be *E*'s character.
4. Let *A* be a one-element CharSet containing the character *ch*.
5. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.
6. *E* must be an integer. Let *n* be that integer.
7. If *n*=0 or *n*>*NCapturingParens* then throw a RegExpError exception.
8. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Let *cap* be *x*'s *captures* internal array.
    2. Let *s* be *cap*[*n*].
    3. If *s* is **undefined**, then call *c*(*x*) and return its result.
    4. Let *e* be *x*'s *endIndex*.
    5. Let *len* be *s*'s length.
    6. Let *f* be *e*+*len*.
    7. If *f*>*InputLength*, return **failure**.
    8. If there exists an integer *i* between 0 (inclusive) and *len* (exclusive) such that *Canonicalize*(*s*[*i*]) is not the same character as *Canonicalize*(*Input* [*e*+*i*]), then return **failure**.
    9. Let *y* be the State (*f*, *cap*).
    10. Call *c*(*y*) and return its result.

The production *AtomEscape* **::** *CharacterEscape* evaluates as follows:

1. Evaluate *CharacterEscape* to obtain a character *ch*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *AtomEscape* **::** *CharacterClassEscape* evaluates as follows:

1. Evaluate *CharacterClassEscape* to obtain a CharSet *A*.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

Informative comments: An escape sequence of the form \ followed by a one- or two-digit decimal number *n* matches either a character *ch* or the result of the *n*th set of capturing parentheses (see section 15.10.2.11). If the escape sequence is a backreference to the *n*th set of capturing parentheses, then it is an error if the regular expression has fewer than *n* capturing parentheses. If the regular expression has *n* or more capturing parentheses but the *n*th one is **undefined** because it hasn't captured anything, then the backreference always succeeds.

15.10.2.10 CharacterEscape

The production *CharacterEscape* **::** *ControlEscape* evaluates by returning the character according to the table below:

ControlEscape	***Unicode Value***	***Name***	***Symbol***
t	**\u0009**	horizontal tab	<HT>
n	**\u000A**	line feed (new line)	<LF>
v	**\u000B**	vertical tab	<VT>
f	**\u000C**	form feed	<FF>
r	**\u000D**	carriage return	<CR>

The production *CharacterEscape* **::** **c** *ControlLetter* evaluates as follows:

1. Let *ch* be the character represented by *ControlLetter*.
2. Let *i* be *ch*'s Unicode number.
3. Let *j* be the remainder of dividing *i* by 32.
4. Return the Unicode character numbered *j*.

The production *CharacterEscape* **::** *HexEscapeSequence* evaluates by evaluating the CV of the *HexEscapeSequence* (see section 7.7.4) and returning its character result.

The production *CharacterEscape* **::** *UnicodeEscapeSequence* evaluates by evaluating the CV of the *UnicodeEscapeSequence* (see section 7.7.4) and returning its character result.

The production *CharacterEscape* **::** *IdentityEscape* evaluates by returning the character represented by *IdentityEscape*.

15.10.2.11 DecimalOrOctalEscape

The production *DecimalOrOctalEscape* **::** *DecimalDigit* [lookahead ∉ *DecimalDigit*] evaluates as follows:

1. Let *i* be the MV of *DecimalDigit* (see section 7.7.3).
2. If *i* is zero, return the EscapeValue consisting of a <NUL> character (Unicode value 0000).
3. Return the EscapeValue consisting of the integer *i*.

The production *DecimalOrOctalEscape* **::** *ZeroToThree OctalDigit* [lookahead ∉ *OctalDigit*] evaluates as follows:

1. Let *i* be the MV of *ZeroToThree* (see section 7.7.4).
2. Let *j* be the MV of *OctalDigit* (see section 7.7.3).
3. Call *TwoDigitEscape*(*i*, *j*) and return its EscapeValue result.

The production *DecimalOrOctalEscape* **::** *ZeroToThree EightOrNine* evaluates as follows:

1. Let *i* be the MV of *ZeroToThree* (see section 7.7.4).
2. Let *j* be the MV of *EightOrNine*.
3. Call *TwoDigitEscape*(*i*, *j*) and return its EscapeValue result.

The production *DecimalOrOctalEscape* **::** *FourToNine DecimalDigit* evaluates as follows:

1. Let *i* be the MV of *FourToNine*.
2. Let *j* be the MV of *DecimalDigit* (see section 7.7.3).
3. Call *TwoDigitEscape*(*i*, *j*) and return its EscapeValue result.

The production *DecimalOrOctalEscape* **::** *ZeroToThree OctalDigit OctalDigit* evaluates as follows:

1. Let *i* be the MV of *ZeroToThree* (see section 7.7.4).
2. Let *j* be the MV of the first *OctalDigit* (see section 7.7.3).
3. Let *k* be the MV of the second *OctalDigit* (see section 7.7.3).
4. Let *ch* be the Unicode character numbered 64**i* + 8**j* + *k*.
5. Return the EscapeValue consisting of the character *ch*.

The internal helper function *TwoDigitEscape* takes two integer parameters *i* and *j* and performs the following:

1. Let *n* = 10**i* + *j*.
2. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this *DecimalOrOctalEscape*. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this *DecimalOrOctalEscape* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *DecimalOrOctalEscape*.
3. If *n* 10 and *n* *parenIndex* then return the EscapeValue consisting of the integer *n*.
4. If *i* > 7 or *j* > 7 then throw a RegExpError exception.
5. Let *ch* be the Unicode character numbered 8**i* + *j*.
6. Return the EscapeValue consisting of the character *ch*.

- The MV of *EightOrNine* **:: 8** is 8.
- The MV of *EightOrNine* **:: 9** is 9.
- The MV of *FourToNine* **:: 4** is 4.
- The MV of *FourToNine* **:: 5** is 5.
- The MV of *FourToNine* **:: 6** is 6.
- The MV of *FourToNine* **:: 7** is 7.
- The MV of *FourToNine* **:: 8** is 8.
- The MV of *FourToNine* **:: 9** is 9.

Informative comments: Escapes of the form \ followed by a two- or three-digit octal number are used to denote either backreferences or ASCII characters by ASCII code. This ambiguity is resolved as follows: If \ is followed by at least a two-digit decimal number *n* whose first digit is not **0** and there are at least *n* left capturing parentheses in the regular expression prior to the escape sequence, then the escape sequence is considered to be a backreference. Also, if either of the first two digits after the \ is not an octal digit, then the sequence is considered to be a backreference. Otherwise it is either a two- or three-digit octal number that denotes a literal ASCII character.

If \ is followed by a one-digit decimal number *n*, then the escape sequence is considered to be a backreference unless *n* is zero. **\0** represents the NUL character. It is an error if *n* is greater than the total number of left capturing parentheses in the entire regular expression.

15.10.2.12 CharacterClassEscape

The production *CharacterClassEscape* **:: d** evaluates by returning the ten-element set of characters containing the characters **0** through **9** inclusive.

The production *CharacterClassEscape* **:: D** evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **:: d**.

The production *CharacterClassEscape* **:: s** evaluates by returning the set of characters containing the characters that are on the right-hand side of the *WhiteSpace* (section 7.1) or *LineTerminator* (section 7.2) productions.

The production *CharacterClassEscape* **:: S** evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **:: s**.

The production *CharacterClassEscape* **:: w** evaluates by returning the set of characters containing the sixty-three characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 _

The production *CharacterClassEscape* **:: W** evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **:: w**.

15.10.2.13 CharacterClass

The production *CharacterClass* **:: [** [lookahead ∉ {**^**}] *ClassRanges* **]** evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the boolean **false**.

The production *CharacterClass* **:: [^** *ClassRanges* **]** evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the boolean **true**.

15.10.2.14 ClassRanges

The production *ClassRanges* **::** [empty] evaluates by returning the empty CharSet.

The production *ClassRanges* **::** *NonemptyClassRanges* evaluates by evaluating *NonemptyClassRanges* to obtain a CharSet and returning that CharSet.

15.10.2.15 NonemptyClassRanges

The production *NonemptyClassRanges* **::** *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

The production *NonemptyClassRanges* **::** *ClassAtom NonemptyClassRangesNoDash* evaluates as follows:

1. Evaluate *ClassAtom* to obtain a CharSet *A*.
2. Evaluate *NonemptyClassRangesNoDash* to obtain a CharSet *B*.
3. Return the union of CharSets *A* and *B*.

The production *NonemptyClassRanges* **::** *ClassAtom* **-** *ClassAtom ClassRanges* evaluates as follows:

1. Evaluate the first *ClassAtom* to obtain a CharSet *A*.
2. Evaluate the second *ClassAtom* to obtain a CharSet *B*.
3. Evaluate *ClassRanges* to obtain a CharSet *C*.
4. Call *CharacterRange*(*A*, *B*) and let *D* be the resulting CharSet.
5. Return the union of CharSets *D* and *C*.

The internal helper function *CharacterRange* takes two CharSet parameters *A* and *B* and performs the following:

1. If *A* does not contain exactly one character or *B* does not contain exactly one character then throw a RegExpError exception.
2. Let *a* be the one character in CharSet *A*.
3. Let *b* be the one character in CharSet *B*.
4. Let *i* be the Unicode value of character *a*.
5. Let *j* be the Unicode value of character *b*.
6. If *i*>*j* then throw a RegExpError exception.
7. Return the set containing all Unicode characters numbered *i* through *j*, inclusive.

15.10.2.16 NonemptyClassRangesNoDash

The production *NonemptyClassRangesNoDash* **::** *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

The production *NonemptyClassRangesNoDash* **::** *ClassAtomNoDash NonemptyClassRangesNoDash* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *NonemptyClassRangesNoDash* to obtain a CharSet *B*.
3. Return the union of CharSets *A* and *B*.

The production *NonemptyClassRangesNoDash* **::** *ClassAtomNoDash* **-** *ClassAtom ClassRanges* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *ClassAtom* to obtain a CharSet *B*.
3. Evaluate *ClassRanges* to obtain a CharSet *C*.
4. Call *CharacterRange*(*A*, *B*) and let *D* be the resulting CharSet.
5. Return the union of CharSets *D* and *C*.

Informative comments: *ClassRanges* can expand into single *ClassAtoms* and/or ranges of two *ClassAtoms* separated by dashes. In the latter case the *ClassRanges* includes all Unicode character between the first *ClassAtom* and the second *ClassAtom*, inclusive; an error occurs if either *ClassAtom* does not represent a single character (for example, if one is **\w**) or if the first *ClassAtom's* Unicode number is greater than the second *ClassAtom's* Unicode number.

Even if the pattern ignores case, the case of the two ends of a range is significant in determining which characters belong to the range. Thus, for example, the pattern **/[E-F]/i** matches only the letters **E**, **F**, **e**, and **f**, while the pattern **/[E-f]/i** matches all upper and lower-case ASCII letters as well as the symbols **[**, ****, **]**, **^**, _, and **`**.

A **-** character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of *ClassRanges*, the beginning or end limit of a range specification, or immediately follows a range specification.

15.10.2.17 ClassAtom

The production *ClassAtom* **::** **-** evaluates by returning the CharSet containing the one character **-**.

The production *ClassAtom* **::** *ClassAtomNoDash* evaluates by evaluating *ClassAtomNoDash* to obtain a CharSet and returning that CharSet.

15.10.2.18 ClassAtomNoDash

The production *ClassAtomNoDash* **::** *SourceCharacter* **but not one of \ ] -** evaluates by returning a one-element CharSet containing the character represented by *SourceCharacter*.

The production *ClassAtomNoDash* **::** \ *ClassEscape* evaluates by evaluating *ClassEscape* to obtain a CharSet and returning that CharSet.

15.10.2.19 ClassEscape

The production *ClassEscape* **::** *DecimalOrOctalEscape* evaluates as follows:

1. Evaluate *DecimalOrOctalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then throw a RegExpError exception.
3. Let *ch* be *E*'s character.
4. Return the one-element CharSet containing the character *ch*.

The production *ClassEscape* **::** **b** evaluates by returning the CharSet containing the one character <BS> (Unicode value 0008).

The production *ClassEscape* **::** *CharacterEscape* evaluates by evaluating *CharacterEscape* to obtain a character and returning a one-element CharSet containing that character.

The production *ClassEscape* **::** *CharacterClassEscape* evaluates by evaluating *CharacterClassEscape* to obtain a CharSet and returning that CharSet.

Informative comments: A *ClassAtom* can use any of the escape sequences that are allowed in the rest of the regular expression except for **\b**, **\B**, and backreferences. Inside a *CharacterClass*, **\b** means the backspace character, while **\B** and backreferences raise errors. Octal escapes may be used inside a *CharacterClass* as long as they don't look like backreferences. Using a backreference inside a ClassAtom causes an error.

15.10.3 The RegExp Constructor Called as a Function

When **RegExp** is called as a function rather than as a constructor, it creates and initialises a new regular expression object.

15.10.3.1 RegExp(pattern, flags)

A regular expression object is created and returned as if by the expression **new RegExp (***pattern, flags***)** .

15.10.3.2 RegExp(pattern)

A regular expression object is created and returned as if by the expression **new RegExp (***pattern***)** .

15.10.3.3 RegExp()

A regular expression object is created and returned as if by the expression **new RegExp ( )** .

15.10.4 The RegExp Constructor

When **RegExp** is called as part of a **new** expression, it is a constructor: it initialises the newly created object.

15.10.4.1 new RegExp(pattern, flags)

Let *P* be the empty string if *pattern* is not provided and ToString(*pattern*) otherwise. Similarly, let *F* be the empty string if *flags* is not provided and ToString(*flags*) otherwise.

The **`global`** property of the newly constructed object is set to a Boolean value that is **true** if *F* contains the character **`'g'`** and **false** otherwise.

The **`ignoreCase`** property of the newly constructed object is set to a Boolean value that is **true** if *F* contains the character **`'i'`** and **false** otherwise.

The **`multiline`** property of the newly constructed object is set to a Boolean value that is **true** if *F* contains the character **`'m'`** and **false** otherwise.

If *F* contains any character other than **`'g'`**, **`'i'`**, or **`'m'`**, or if it contains the same one more than once, then throw a RegExpError exception.

If *P*'s characters do not have the form *Pattern,* then throw a RegExpError exception. Otherwise let the newly constructed object have a [[Match]] property obtained by evaluating ("compiling") *Pattern*. Note that evaluating *Pattern* may throw a RegExpError exception. (Note: if *pattern* is a *StringLiteral*, the usual escape sequence substitutions are performed before the string is processed by **`RegExp`**. If *pattern* must contain an escape sequence to be recognised by **`RegExp`**, the "****" character must be escaped within the *StringLiteral* to prevent its being removed when the contents of the *StringLiteral* are formed.)

The **`source`** property of the newly constructed object is set to an implementation-defined string value in the form of a *Pattern* based on *P*.

The **`lastIndex`** property of the newly constructed object is set to **0**.

The [[Prototype]] property of the newly constructed object is set to the original RegExp prototype object, the one that is the initial value of **`RegExp.prototype`**.

The [[Class]] property of the newly constructed object is set to **`"RegExp"`**.

15.10.4.2 new RegExp(pattern)

This is the same as calling **new RegExp(pattern, "")**.

15.10.4.3 new RegExp()

This is the same as calling **new RegExp("", "")**.

15.10.5 Properties of the RegExp Constructor

The value of the internal [[Prototype]] property of the RegExp constructor is the Function prototype object.

Besides the internal [[Call]] and [[Construct]] properties, the RegExp constructor has the following properties:

15.10.5.1 RegExp.prototype

The initial value of **`RegExp.prototype`** is the built-in RegExp prototype object.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.10.5.2 RegExp.length

The **`length`** property is **2**.

15.10.6 Properties of the RegExp Prototype Object

The value of the internal [[Prototype]] property of the RegExp prototype object is the Object prototype.

The Function prototype object does not have a **valueOf** property of its own; however, it inherits the **valueOf** property from the Object prototype object.

In the following descriptions of functions that are properties of the RegExp prototype object, the phrase "this RegExp object" refers to the object that is the **this** value for the invocation of the function; a **TypeError** is thrown when any of the functions below is called with **this** not referring to an object for which the value of the internal [[Class]] property is **"RegExp"**.

15.10.6.1 RegExp.prototype.constructor

The initial value of **RegExp.prototype.constructor** is the built-in **RegExp** constructor.

15.10.6.2 RegExp.prototype.exec(string)

Performs a regular expression match of *string* against the regular expression and returns an Array object containing the results of the match, or **null** if the string did not match

The string ToString(*string*) is searched for an occurrence of the regular expression pattern as follows:

1. Let *S* be the value of ToString(*string*).
2. Let *length* be the length of *S*.
3. Let *lastIndex* be the value of the **lastIndex** property.
4. Let *i* be the value of ToInteger(*lastIndex*).
5. If the **global** property is **false**, let *i* = 0.
6. If *i*<0 or *i*>*length* then set **lastIndex** to 0 and return **null**.
7. Call [[Match]], giving it the arguments *S* and *i*. If [[Match]] returned **failure**, go to step 8; otherwise let *r* be its State result and go to step 10.
8. Let *i* = *i*+1.
9. Go to step 6.
10. Let *e* be *r*'s *endIndex* value.
11. If the **global** property is **true**, set **lastIndex** to *e*.
12. Let *n* be the length of *r*'s *captures* array. (This is the same value as section 15.10.2.1's *NCapturingParens*.)
13. Return a new array with the following properties:

- The **index** property is set to the position of the matched substring within the complete string *S*.
- The **input** property is set to *S*.
- The **length** property is set to *n* + 1.
- The **0** property is set to the matched substring (i.e. the portion of *S* between offset *i* inclusive and offset *e* exclusive).
- For each integer *i* such that *i*>0 and *i* *n*, set the property named ToString(*i*) to the *i*th element of *r*'s *captures* array.

15.10.6.3 RegExp.prototype.test(string)

Equivalent to the expression **RegExp.prototype.exec(string) != null.**

15.10.6.4 RegExp.prototype.toString()

Returns a string value formed by concatenating the strings **"/"**, the value of the **source** property, and **"/"**; plus **"g"** if the **global** property is **true**, **"i"** if the **ignoreCase** property is **true**, and **"m"** if the **multiline** property is **true.**

15.10.7 Properties of RegExp Instances

RegExp instances inherit properties from their [[Prototype]] object as specified above and also have the following properties.

15.10.7.1 source

The value of the **source** property is string in the form of a *RegularExpressionLiteral* representing the current regular expression. This property shall have the attributes { DontDelete, ReadOnly }.

15.10.7.2 global

The value of the **`global`** property is a Boolean value indicating whether the flags contained the character **`'g'`**. This property shall have the attributes { DontDelete, ReadOnly }.

15.10.7.3 ignoreCase

The value of the **`ignoreCase`** property is a Boolean value indicating whether the flags contained the character **`'i'`**. This property shall have the attributes { DontDelete, ReadOnly }.

15.10.7.4 multiline

The value of the **`multiline`** property is a Boolean value indicating whether the flags contained the character **`'m'`**. This property shall have the attributes { DontDelete, ReadOnly }.

15.10.7.5 lastIndex

The value of the **`lastIndex`** property is an integer which specifies the string position at which to start the next match. This property shall have the attributes { DontDelete }.

16 Errors

An implementation should report runtime errors at the time the relevant language construct is evaluated. An implementation may report syntax errors in the program at the time the program is read in, or it may, at its option, defer reporting syntax errors until the relevant statement is reached. An implementation may report syntax errors in **`eval`** code at the time **`eval`** is called, or it may, at its option, defer reporting syntax errors until the relevant statement is reached.

An implementation may treat any instance of the following kinds of runtime errors as a syntax error and therefore report it early:

- Improper uses of **`return`**, **`break`**, and **`continue`**.
- Using the **`eval`** property other than via a direct call.
- Errors in regular expression literals.
- Attempts to call PutValue on a value that is not a reference (for example, executing the assignment statement 3=4).

An implementation shall not report other kinds of runtime errors early even if the compiler can prove that a construct cannot execute without error under any circumstances. An implementation may issue an early warning in such a case, but it should not report the error until the relevant construct is actually executed.

~~This specification specifies the last possible moment an error occurs. A given implementation may generate errors sooner (e.g., at compile-time). Doing so may cause differences in behaviour among implementations. Notably, if runtime errors become catchable in future versions, a given error would not be catchable if an implementation generates the error at compile-time rather than runtime.~~

~~An ECMAScript compiler should detect errors at compile time in all code presented to it, even code that detailed analysis might prove to be "dead" (never executed). A programmer should not rely on the trick of placing code within an **`if (false)`** statement, for example, to try to suppress compile-time error detection.~~

~~In general, if a compiler can prove that a construct cannot execute without error under any circumstances, then it may issue a compile-time error even though the construct might never be executed at all.~~