



INTERNATIONAL

ECMA-262, 10th edition, June 2019

ECMAScript® 2019

Language Specification

Contributing to this Specification

This specification is developed on GitHub with the help of the ECMAScript community. There are a number of ways to contribute to the development of this specification:

GitHub Repository: <https://github.com/tc39/ecma262>

Issues: [All Issues](#), [File a New Issue](#)

Pull Requests: [All Pull Requests](#), [Create a New Pull Request](#)

Test Suite: [Test262](#)

Editors:

- Brian Terlson ([@bterlson](#))
- Bradley Farias ([@bradleymeck](#))
- Jordan Harband ([@ljharb](#))

Community:

- Mailing list: [es-discuss](#)
- IRC: [#tc39](#) on freenode

Refer to the [colophon](#) for more information on how this document is created.

Introduction

This Ecma Standard defines the ECMAScript 2019 Language. It is the tenth edition of the ECMAScript Language Specification. Since publication of the first edition in 1997, ECMAScript has grown to be one of the world's most widely used general-purpose programming languages. It is best known as the language embedded in web browsers but has also been widely adopted for server and embedded applications.

ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.

providing syntax for promise-returning functions. Shared Memory and Atomics introduce a new [memory model](#) that allows multi-agent programs to communicate using atomic operations that ensure a well-defined execution order even on parallel CPUs. This specification also includes new static methods on Object: `Object.values`, `Object.entries`, and `Object.getOwnPropertyDescriptors`.

ECMAScript 2018 introduced support for asynchronous iteration via the AsyncIterator protocol and `async` generators. It also included four new regular expression features: the `dotAll` flag, named capture groups, Unicode property escapes, and look-behind assertions. Lastly it included rest parameter and spread operator support for object properties.

This specification, the 10th edition, introduces a few new built-in functions: `flat` and `flatMap` on `Array.prototype` for flattening arrays, `Object.fromEntries` for directly turning the return value of `Object.entries` into a new Object, and `trimStart` and `trimEnd` on `String.prototype` as better-named alternatives to the widely implemented but non-standard `String.prototype.trimLeft` and `trimRight` built-ins. In addition, this specification includes a few minor updates to syntax and semantics. Updated syntax includes optional catch binding parameters and allowing U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) in string literals to align with JSON. Other updates include requiring that `Array.prototype.sort` be a stable sort, requiring that `JSON.stringify` return well-formed UTF-8 regardless of input, and clarifying `Function.prototype.toString` by requiring that it either return the corresponding original source text or a standard placeholder.

Dozens of individuals representing many organizations have made very significant contributions within Ecma TC39 to the development of this edition and to the prior editions. In addition, a vibrant community has emerged supporting TC39's ECMAScript efforts. This community has reviewed numerous drafts, filed thousands of bug reports, performed implementation experiments, contributed test suites, and educated the world-wide developer community about ECMAScript. Unfortunately, it is impossible to identify and acknowledge every person and organization who has contributed to this effort.

Allen Wirfs-Brock
ECMA-262, Project Editor, 6th Edition

Brian Terlson
ECMA-262, Project Editor, 7th through 10th Editions

1 Scope

This Standard defines the ECMAScript 2019 general-purpose programming language.

2 Conformance

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax and semantics described in this specification.

A conforming implementation of ECMAScript must interpret source text input in conformance with the latest version of the Unicode Standard and ISO/IEC 10646.

A conforming implementation of ECMAScript that provides an application programming interface (API) that supports

scripting language. A scripting language is intended for use by both professional and non-professional programmers.

ECMAScript was originally designed to be a *Web scripting language*, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript is now used to provide core scripting capabilities for a variety of host environments. Therefore the core language is specified in this document apart from any particular host environment.

ECMAScript usage has moved beyond simple scripting and it is now used for the full spectrum of programming tasks in many different environments and scales. As the usage of ECMAScript has expanded, so has the features and facilities it provides. ECMAScript is now a fully featured general-purpose programming language.

Some of the facilities of ECMAScript are similar to those used in other programming languages; in particular C, Java™, Self, and Scheme as described in:

ISO/IEC 9899:1996, *Programming Languages – C*.

Gosling, James, Bill Joy and Guy Steele. *The Java™ Language Specification*. Addison Wesley Publishing Co., 1996.

Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October 1987.

IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

4.1 Web Scripting

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. Further, the host environment provides a means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. Scripting code appears within the HTML and the displayed page is a combination of user interface elements and fixed and computed text and images. The scripting code is reactive to user interaction, and there is no need for a main program.

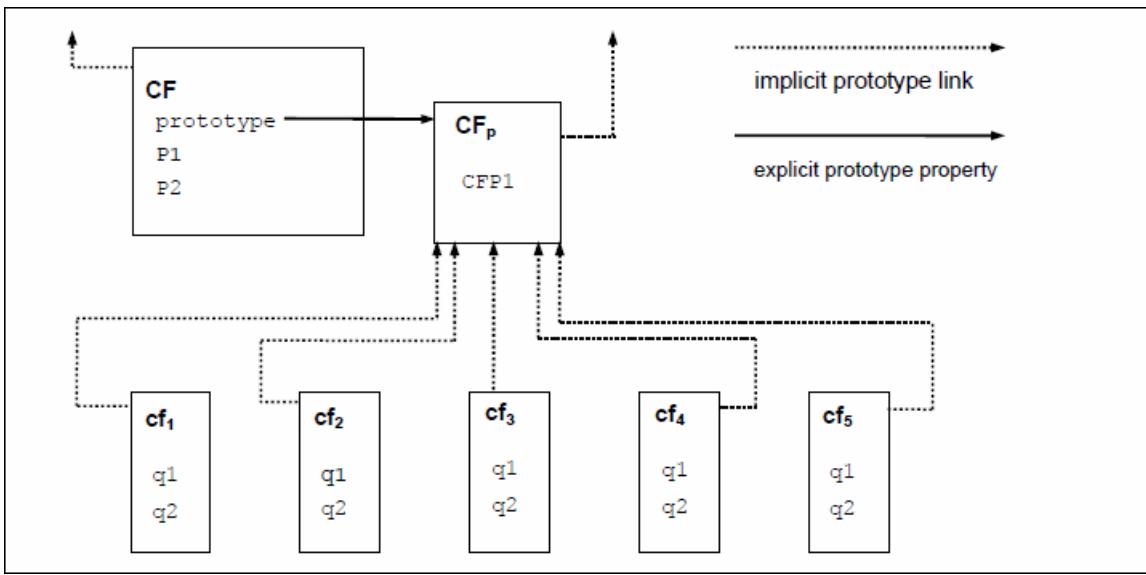
A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data. By using browser-side and server-side scripting together, it is possible to distribute computation between the client and server while providing a customized user interface for a Web-based application.

Each Web browser and server that supports ECMAScript supplies its own host environment, completing the ECMAScript execution environment.

4.2 ECMAScript Overview

The following is an informal overview of ECMAScript—not all parts of the language are described. This overview is not part of the standard proper.

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. In ECMAScript, an *object* is a collection of zero or more *properties* each with *attributes* that determine how each property can be used—for example, when the *Writable* attribute for a property is set to **false**, any attempt by executed ECMAScript code to assign a different value to the property fails. Properties are containers that hold other objects, *primitive values*, or *functions*. A primitive value is a member of one of the following



In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, while structure, behaviour, and state are all inherited.

All objects that do not directly contain a particular property that their prototype contains share that property and its value. Figure 1 illustrates this:

CF is a [constructor](#) (and also an object). Five objects have been created by using `new` expressions: **cf₁**, **cf₂**, **cf₃**, **cf₄**, and **cf₅**. Each of these objects contains properties named **q₁** and **q₂**. The dashed lines represent the implicit prototype relationship; so, for example, **cf₃**'s prototype is **CF_p**. The [constructor](#), **CF**, has two properties itself, named **P₁** and **P₂**, which are not visible to **CF_p**, **cf₁**, **cf₂**, **cf₃**, **cf₄**, or **cf₅**. The property named **CFP1** in **CF_p** is shared by **cf₁**, **cf₂**, **cf₃**, **cf₄**, and **cf₅** (but not by **CF**), as are any properties found in **CF_p**'s implicit prototype chain that are not named **q₁**, **q₂**, or **CFP1**. Notice that there is no implicit prototype link between **CF** and **CF_p**.

Unlike most class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for **cf₁**, **cf₂**, **cf₃**, **cf₄**, and **cf₅** by assigning a new value to the property in **CF_p**.

Although ECMAScript objects are not inherently class-based, it is often convenient to define class-like abstractions based upon a common pattern of [constructor](#) functions, prototype objects, and methods. The ECMAScript built-in objects themselves follow such a class-like pattern. Beginning with ECMAScript 2015, the ECMAScript language includes syntactic class definitions that permit programmers to concisely define objects that conform to the same class-like abstraction pattern used by the built-in objects.

4.2.2 The Strict Variant of ECMAScript

The ECMAScript Language recognizes the possibility that some users of the language may wish to restrict their usage of some features available in the language. They might do so in the interests of security, to avoid what they consider to be error-prone features, to get enhanced error checking, or for other reasons of their choosing. In support of this possibility, ECMAScript defines a strict variant of the language. The strict variant of the language excludes some specific syntactic and semantic features of the regular ECMAScript language and modifies the detailed semantics of some features. The strict variant also specifies additional error conditions that must be reported by throwing error exceptions in situations that are not specified as errors by the non-strict form of the language.

object that provides shared properties for other objects

NOTE

When a `constructor` creates an object, that object implicitly references the `constructor`'s `prototype` property for the purpose of resolving property references. The `constructor`'s `prototype` property can be referenced by the program expression `constructor.prototype`, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype. Alternatively, a new object may be created with an explicitly specified prototype by using the `Object.create` built-in function.

4.3.6 ordinary object

object that has the default behaviour for the essential internal methods that must be supported by all objects

4.3.7 exotic object

object that does not have the default behaviour for one or more of the essential internal methods

NOTE

Any object that is not an ordinary object is an `exotic object`.

4.3.8 standard object

object whose semantics are defined by this specification

4.3.9 built-in object

object specified and supplied by an ECMAScript implementation

NOTE

Standard built-in objects are defined in this specification. An ECMAScript implementation may specify and supply additional kinds of built-in objects. A *built-in constructor* is a built-in object that is also a `constructor`.

4.3.10 undefined value

primitive value used when a variable has not been assigned a value

4.3.11 Undefined type

type whose sole value is the `undefined` value

4.3.12 null value

primitive value that represents the intentional absence of any object value

4.3.20 Number value

primitive value corresponding to a double-precision 64-bit binary format IEEE 754-2008 value

NOTE

A Number value is a member of the Number type and is a direct representation of a number.

4.3.21 Number type

set of all possible Number values including the special “Not-a-Number” (NaN) value, positive infinity, and negative infinity

4.3.22 Number object

member of the Object type that is an instance of the standard built-in **Number** constructor

NOTE

A Number object is created by using the **Number** constructor in a **new** expression, supplying a number value as an argument. The resulting object has an internal slot whose value is the number value. A Number object can be coerced to a number value by calling the **Number** constructor as a function (20.1.1.1).

4.3.23 Infinity

number value that is the positive infinite number value

4.3.24 NaN

number value that is an IEEE 754-2008 “Not-a-Number” value

4.3.25 Symbol value

primitive value that represents a unique, non-String Object property key

4.3.26 Symbol type

set of all possible Symbol values

4.3.27 Symbol object

member of the Object type that is an instance of the standard built-in **Symbol** constructor

4.3.28 function

member of the Object type that may be invoked as a subroutine

NOTE

property of an object that is not an own property but is a property (either own or inherited) of the object's prototype

4.4 Organization of This Specification

The remainder of this specification is organized as follows:

Clause 5 defines the notational conventions used throughout the specification.

Clauses 6-9 define the execution environment within which ECMAScript programs operate.

Clauses 10-16 define the actual ECMAScript programming language including its syntactic encoding and the execution semantics of all language features.

Clauses 17-26 define the ECMAScript standard library. They include the definitions of all of the standard objects that are available for use by ECMAScript programs as they execute.

Clause 27 describes the memory consistency model of accesses on SharedArrayBuffer-backed memory and methods of the Atomics object.

5 Notational Conventions

5.1 Syntactic and Lexical Grammars

5.1.1 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 chain production is a production that has exactly one nonterminal symbol on its right-hand side along with zero or more terminal symbols.

Starting from a sentence consisting of a single distinguished nonterminal, called the goal symbol, a given context-free grammar specifies a *language*, namely, the (perhaps infinite) set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

5.1.2 The Lexical and RegExp Grammars

A *lexical grammar* for ECMAScript is given in clause 11. This grammar has as its terminal symbols Unicode code points that conform to the rules for *SourceCharacter* defined in 10.1. It defines a set of productions, starting from the *goal symbol* *InputElementDiv*, *InputElementTemplateTail*, or *InputElementRegExp*, or *InputElementRegExpOrTemplateTail*, that describe how sequences of such code points are translated into a sequence of input elements.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become

Parse Nodes are specification artefacts, and implementations are not required to use an analogous data structure.

Productions of the syntactic grammar are distinguished by having just one colon “:” as punctuation.

The syntactic grammar as presented in clauses 12, 13, 14 and 15 is not a complete account of which token sequences are accepted as a correct ECMAScript *Script* or *Module*. Certain additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if a line terminator character appears in certain “awkward” places.

In certain cases, in order to avoid ambiguities, the syntactic grammar uses generalized productions that permit token sequences that do not form a valid ECMAScript *Script* or *Module*. For example, this technique is used for object literals and object destructuring patterns. In such cases a more restrictive *supplemental grammar* is provided that further restricts the acceptable token sequences. Typically, an [early error](#) rule will then define an error condition if "*P*" is not covering an "*N*", where *P* is a Parse Node (an instance of the generalized production) and *N* is a nonterminal from the supplemental grammar. Here, the sequence of tokens originally matched by *P* is parsed again using *N* as the [goal symbol](#). (If *N* takes grammatical parameters, then they are set to the same values used when *P* was originally parsed.) An error occurs if the sequence of tokens cannot be parsed as a single instance of *N*, with no tokens left over. Subsequently, algorithms access the result of the parse using a phrase of the form "the *N* that is covered by *P*". This will always be a Parse Node (an instance of *N*, unique for a given *P*), since any parsing failure would have been detected by an [early error](#) rule.

5.1.5 Grammar Notation

Terminal symbols of the lexical, RegExp, and numeric string grammars are shown in **fixed width** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a script exactly as written. All terminal symbol code points specified in this way are to be understood as the appropriate Unicode code points from the Basic Latin range, as opposed to any similar-looking code points from other Unicode ranges.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal (also called a “production”) is introduced by the name of the nonterminal being defined followed by one or more colons. (The number of colons indicates to which grammar the production belongs.) One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

```
WhileStatement :  
  while ( Expression ) Statement
```

states that the nonterminal *WhileStatement* represents the token **while**, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*. The occurrences of *Expression* and *Statement* are themselves nonterminals. As another example, the syntactic definition:

```
ArgumentList :  
  AssignmentExpression  
  ArgumentList , AssignmentExpression
```

states that an *ArgumentList* may represent either a single *AssignmentExpression* or an *ArgumentList*, followed by a comma, followed by an *AssignmentExpression*. This definition of *ArgumentList* is recursive, that is, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments, separated by commas, where each argument expression is an *AssignmentExpression*. Such recursive definitions of nonterminals are common.

and that:

```
StatementList [Return, In] :  
  ReturnStatement  
  ExpressionStatement
```

is an abbreviation for:

```
StatementList :  
  ReturnStatement  
  ExpressionStatement
```

```
StatementList_Return :  
  ReturnStatement  
  ExpressionStatement
```

```
StatementList_In :  
  ReturnStatement  
  ExpressionStatement
```

```
StatementList_Return_In :  
  ReturnStatement  
  ExpressionStatement
```

Multiple parameters produce a combinatory number of productions, not all of which are necessarily referenced in a complete grammar.

References to nonterminals on the right-hand side of a production can also be parameterized. For example:

```
StatementList :  
  ReturnStatement  
  ExpressionStatement [+In]
```

is equivalent to saying:

```
StatementList :  
  ReturnStatement  
  ExpressionStatement_In
```

and:

```
StatementList :  
  ReturnStatement  
  ExpressionStatement [~In]
```

is equivalent to:

```
StatementList :  
  ReturnStatement  
  ExpressionStatement
```

A nonterminal reference may have both a parameter list and an “opt” suffix. For example:

StatementList_Return :

ExpressionStatement

When the words “**one of**” follow the colon(s) in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for ECMAScript contains the production:

```
NonZeroDigit :: one of
  1 2 3 4 5 6 7 8 9
```

which is merely a convenient abbreviation for:

```
NonZeroDigit ::
```

```
 1
  2
  3
  4
  5
  6
  7
  8
  9
```

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 \notin *set*]” appears in the right-hand side of a production, it indicates that the production may not be used if the immediately following input token sequence is a member of the given *set*. The *set* can be written as a comma separated list of one or two element terminal sequences enclosed in curly brackets. 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. If the *set* consists of a single terminal the phrase “[lookahead \neq *terminal*]” may be used.

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  $\notin$  { 1 , 3 , 5 , 7 , 9 }] DecimalDigits  
  DecimalDigit [lookahead  $\notin$  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.

Similarly, if the phrase “[lookahead \in *set*]” appears in the right-hand side of a production, it indicates that the production may only be used if the immediately following input token sequence is a member of the given *set*. If the *set* consists of a

- b. Substep.
 - i. Subsubstep.
 - 1. Subsubsubstep
 - a. Subsubsubsubstep
 - i. Subsubsubsubsubstep

A step or substep may be written as an “if” predicate that conditions its substeps. In this case, the substeps are only applied if the predicate is true. If a step or substep begins with the word “else”, it is a predicate that is the negation of the preceding “if” predicate step at the same level.

A step may specify the iterative application of its substeps.

A step that begins with “Assert:” asserts an invariant condition of its algorithm. Such assertions are used to make explicit algorithmic invariants that would otherwise be implicit. Such assertions add no additional semantic requirements and hence need not be checked by an implementation. They are used simply to clarify algorithms.

Algorithm steps may declare named aliases for any value using the form “Let *x* be *someValue*”. These aliases are reference-like in that both *x* and *someValue* refer to the same underlying data and modifications to either are visible to both. Algorithm steps that want to avoid this reference-like behaviour should explicitly make a copy of the right-hand side: “Let *x* be a copy of *someValue*” creates a shallow copy of *someValue*.

Once declared, an alias may be referenced in any subsequent steps and must not be referenced from steps prior to the alias's declaration. Aliases may be modified using the form “Set *x* to *someOtherValue*”.

5.2.1 Abstract Operations

In order to facilitate their use in multiple parts of this specification, some algorithms, called abstract operations, are named and written in parameterized functional form so that they may be referenced by name from within other algorithms. Abstract operations are typically referenced using a functional application style such as *OperationName(arg1, arg2)*. Some abstract operations are treated as polymorphically dispatched methods of class-like specification abstractions. Such method-like abstract operations are typically referenced using a method application style such as *someValue.OperationName(arg1, arg2)*.

5.2.2 Syntax-Directed Operations

A syntax-directed operation is a named operation whose definition consists of algorithms, each of which is associated with one or more productions from one of the ECMAScript grammars. A production that has multiple alternative definitions will typically have a distinct algorithm for each alternative. When an algorithm is associated with a grammar production, it may reference the terminal and nonterminal symbols of the production alternative as if they were parameters of the algorithm. When used in this manner, nonterminal symbols refer to the actual alternative definition that is matched when parsing the source text. The source text matched by a grammar production is the portion of the source text that starts at the beginning of the first terminal that participated in the match and ends at the end of the last terminal that participated in the match.

When an algorithm is associated with a production alternative, the alternative is typically shown without any “[]” grammar annotations. Such annotations should only affect the syntactic recognition of the alternative and have no effect on the associated semantics for the alternative.

Syntax-directed operations are invoked with a parse node and, optionally, other parameters by using the conventions on steps 1, 3, and 4 in the following algorithm:

Any reference to a **Completion Record** value that is in a context that does not explicitly require a complete **Completion Record** value is equivalent to an explicit reference to the **[[Value]]** field of the **Completion Record** value unless the **Completion Record** is an **abrupt completion**.

5.2.3.2 Throw an Exception

Algorithms steps that say to throw an exception, such as

1. Throw a **TypeError** exception.

mean the same things as:

1. Return **ThrowCompletion**(a newly created **TypeError** object).

5.2.3.3 ReturnIfAbrupt

Algorithms steps that say or are otherwise equivalent to:

1. **ReturnIfAbrupt**(*argument*).

mean the same thing as:

1. If *argument* is an **abrupt completion**, return *argument*.
2. Else if *argument* is a **Completion Record**, set *argument* to *argument*.**[[Value]]**.

Algorithms steps that say or are otherwise equivalent to:

1. **ReturnIfAbrupt**(**AbstractOperation**()).

mean the same thing as:

1. Let *hygienicTemp* be **AbstractOperation**().
2. If *hygienicTemp* is an **abrupt completion**, return *hygienicTemp*.
3. Else if *hygienicTemp* is a **Completion Record**, set *hygienicTemp* to *hygienicTemp*.**[[Value]]**.

Where *hygienicTemp* is ephemeral and visible only in the steps pertaining to **ReturnIfAbrupt**.

Algorithms steps that say or are otherwise equivalent to:

1. Let *result* be **AbstractOperation**(**ReturnIfAbrupt**(*argument*)).

mean the same thing as:

1. If *argument* is an **abrupt completion**, return *argument*.
2. If *argument* is a **Completion Record**, set *argument* to *argument*.**[[Value]]**.
3. Let *result* be **AbstractOperation**(*argument*).

5.2.3.4 ReturnIfAbrupt Shorthands

Invocations of **abstract operations** and syntax-directed operations that are prefixed by **?** indicate that **ReturnIfAbrupt** should be applied to the resulting **Completion Record**. For example, the step:

1. **?** **OperationName**().

A special kind of static semantic rule is an Early Error Rule. [Early error](#) rules define [early error](#) conditions (see clause 16) that are associated with specific grammar productions. Evaluation of most [early error](#) rules are not explicitly invoked within the algorithms of this specification. A conforming implementation must, prior to the first evaluation of a *Script* or *Module*, validate all of the [early error](#) rules of the productions used to parse that *Script* or *Module*. If any of the [early error](#) rules are violated the *Script* or *Module* is invalid and cannot be evaluated.

5.2.5 Mathematical Operations

Mathematical operations such as addition, subtraction, negation, multiplication, division, and the mathematical functions defined later in this clause should always be understood as computing exact mathematical results on mathematical real numbers, which unless otherwise noted do not include infinities and do not include a negative zero that is distinguished from positive zero. Algorithms in this standard that model floating-point arithmetic include explicit steps, where necessary, to handle infinities and signed zero and to perform rounding. If a mathematical operation or function is applied to a floating-point number, it should be understood as being applied to the exact mathematical value represented by that floating-point number; such a floating-point number must be finite, and if it is **+0** or **-0** then the corresponding mathematical value is simply 0.

The mathematical function $\text{abs}(x)$ produces the absolute value of x , which is $-x$ if x is negative (less than zero) and otherwise is x itself.

The mathematical function $\text{min}(x_1, x_2, \dots, x_N)$ produces the mathematically smallest of x_1 through x_N . The mathematical function $\text{max}(x_1, x_2, \dots, x_N)$ produces the mathematically largest of x_1 through x_N . The domain and range of these mathematical functions include $+\infty$ and $-\infty$.

The notation “ x modulo y ” (y must be finite and nonzero) computes a value k of the same sign as y (or zero) such that $\text{abs}(k) < \text{abs}(y)$ and $x - k = q \times y$ for some integer q .

The mathematical function $\text{floor}(x)$ produces the largest integer (closest to positive infinity) that is not larger than x .

NOTE

$\text{floor}(x) = x - (\text{x modulo } 1)$.

6 ECMAScript Data Types and Values

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further subclassified into ECMAScript language types and specification types.

Within this specification, the notation “Type(x)” is used as shorthand for “the type of x ” where “type” refers to the ECMAScript language and specification types defined in this clause. When the term “empty” is used as if it was naming a value, it is equivalent to saying “no value of any type”.

6.1 ECMAScript Language Types

An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Symbol, Number, and Object. An ECMAScript language value is a value that is characterized by an ECMAScript language type.

not contain any Unicode escape sequences.

In this specification, the phrase "the string-concatenation of *A*, *B*, ..." (where each argument is a String value, a code unit, or a sequence of code units) denotes the String value whose sequence of code units is the concatenation of the code units (in order) of each of the arguments (in order).

6.1.5 The Symbol Type

The Symbol type is the set of all non-String values that may be used as the key of an Object property (6.1.7).

Each possible Symbol value is unique and immutable.

Each Symbol value immutably holds an associated value called [[Description]] that is either **undefined** or a String value.

6.1.5.1 Well-Known Symbols

Well-known symbols are built-in Symbol values that are explicitly referenced by algorithms of this specification. They are typically used as the keys of properties whose values serve as extension points of a specification algorithm. Unless otherwise specified, well-known symbols values are shared by all realms (8.2).

Within this specification a well-known symbol is referred to by using a notation of the form @@name, where “name” is one of the values listed in Table 1.

Table 1: Well-known Symbols

Specification Name	[[Description]]	Value and Purpose
@@asyncIterator	" Symbol.asyncIterator "	A method that returns the default AsyncIterator for an object. Called by the semantics of the for-await-of statement.
@@hasInstance	" Symbol.hasInstance "	A method that determines if a constructor object recognizes an object as one of the constructor 's instances. Called by the semantics of the instanceof operator.
@@isConcatSpreadable	" Symbol.isConcatSpreadable "	A Boolean valued property that if true indicates that an object should be flattened to its array elements by Array.prototype.concat .
@@iterator	" Symbol.iterator "	A method that returns the default Iterator for an object. Called by the semantics of the for-of statement.
@@match	" Symbol.match "	A regular expression method that matches the regular expression against a string. Called by the String.prototype.match method.
@@replace	" Symbol.replace "	A regular expression method that replaces matched substrings of a string. Called by the String.prototype.replace method.

Note that there is both a **positive zero** and a **negative zero**. For brevity, these values are also referred to for expository purposes by the symbols **+0** and **-0**, respectively. (Note that these two different zero Number values are produced by the program expressions **+0** (or simply **0**) and **-0**.)

The 18437736874454810622 (that is, $2^{64} - 2^{53} - 2$) finite nonzero values are of two kinds:

18428729675200069632 (that is, $2^{64} - 2^{54}$) of them are normalized, having the form

$s \times m \times 2^e$

where s is +1 or -1, m is a positive integer less than 2^{53} but not less than 2^{52} , and e is an integer ranging from -1074 to 971, inclusive.

The remaining 9007199254740990 (that is, $2^{53} - 2$) values are denormalized, having the form

$s \times m \times 2^e$

where s is +1 or -1, m is a positive integer less than 2^{52} , and e is -1074.

Note that all the positive and negative integers whose magnitude is no greater than 2^{53} are representable in the Number type (indeed, the integer 0 has two representations, **+0** and **-0**).

A finite number has an *odd significand* if it is nonzero and the integer m used to express it (in one of the two forms shown above) is odd. Otherwise, it has an *even significand*.

In this specification, the phrase “the Number value for x ” where x represents an exact real mathematical quantity (which might even be an irrational number such as π) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with **-0** removed and with two additional values added to it that are not representable in the Number type, namely 2^{1024} (which is $+1 \times 2^{53} \times 2^{971}$) and -2^{1024} (which is $-1 \times 2^{53} \times 2^{971}$). Choose the member of this set that is closest in value to x . If two values of the set are equally close, then the one with an even significand is chosen; for this purpose, the two extra values 2^{1024} and -2^{1024} are considered to have even significands. Finally, if 2^{1024} was chosen, replace it with $+\infty$; if -2^{1024} was chosen, replace it with $-\infty$; if **+0** was chosen, replace it with **-0** if and only if x is less than zero; any other chosen value is used unchanged. The result is the Number value for x . (This procedure corresponds exactly to the behaviour of the IEEE 754-2008 “round to nearest, ties to even” mode.)

Some ECMAScript operators deal only with integers in specific ranges such as -2^{31} through $2^{31} - 1$, inclusive, or in the range 0 through $2^{16} - 1$, inclusive. These operators accept any value of the Number type but first convert each such value to an integer value in the expected range. See the descriptions of the numeric conversion operations in [7.1](#).

6.1.7 The Object Type

An Object is logically a collection of properties. Each property is either a data property, or an accessor property:

A data property associates a key value with an [ECMAScript language value](#) and a set of Boolean attributes.

An accessor property associates a key value with one or two accessor functions, and a set of Boolean attributes. The accessor functions are used to store or retrieve an [ECMAScript language value](#) that is associated with the property.

Properties are identified using key values. A property key value is either an ECMAScript String value or a Symbol value. All String and Symbol values, including the empty string, are valid as property keys. A property name is a property key

[[Set]]	Object Undefined	If the value is an Object it must be a function object . The function's [[Call]] internal method (Table 6) is called with an arguments list containing the assigned value as its sole argument each time a set access of the property is performed. The effect of a property's [[Set]] internal method may, but is not required to, have an effect on the value returned by subsequent calls to the property's [[Get]] internal method.
[[Enumerable]]	Boolean	If true , the property is to be enumerated by a for-in enumeration (see 13.7.5). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If false , attempts to delete the property, change the property to be a data property , or change its attributes will fail.

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in [Table 4](#) is used.

Table 4: Default Attribute Values

Attribute Name	Default Value
[[Value]]	undefined
[[Get]]	undefined
[[Set]]	undefined
[[Writable]]	false
[[Enumerable]]	false
[[Configurable]]	false

6.1.7.2 Object Internal Methods and Internal Slots

The actual semantics of objects, in ECMAScript, are specified via algorithms called *internal methods*. Each object in an ECMAScript engine is associated with a set of internal methods that defines its runtime behaviour. These internal methods are not part of the ECMAScript language. They are defined by this specification purely for expository purposes. However, each object within an implementation of ECMAScript must behave as specified by the internal methods associated with it. The exact manner in which this is accomplished is determined by the implementation.

Internal method names are polymorphic. This means that different object values may perform different algorithms when a common internal method name is invoked upon them. That actual object upon which an internal method is invoked is the “target” of the invocation. If, at runtime, the implementation of an algorithm attempts to use an internal method of an object that the object does not support, a **TypeError** exception is thrown.

Internal slots correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms. Internal slots are not object properties and they are not inherited. Depending upon the specific internal slot specification, such state may consist of values of any [ECMAScript language type](#) or of specific ECMAScript specification type values. Unless explicitly specified otherwise, internal slots are allocated as part of the process of

[[Get]]	(<i>propertyKey</i> , <i>Receiver</i>) → <i>any</i>	Return the value of the property whose key is <i>propertyKey</i> from this object. If any ECMAScript code must be executed to retrieve the property value, <i>Receiver</i> is used as the this value when evaluating the code.
[[Set]]	(<i>propertyKey</i> , <i>value</i> , <i>Receiver</i>) → Boolean	Set the value of the property whose key is <i>propertyKey</i> to <i>value</i> . If any ECMAScript code must be executed to set the property value, <i>Receiver</i> is used as the this value when evaluating the code. Returns true if the property value was set or false if it could not be set.
[[Delete]]	(<i>propertyKey</i>) → Boolean	Remove the own property whose key is <i>propertyKey</i> from this object. Return false if the property was not deleted and is still present. Return true if the property was deleted or is not present.
[[OwnPropertyKeys]]	() → <i>List</i> of <i>propertyKey</i>	Return a <i>List</i> whose elements are all of the own property keys for the object.

Table 6 summarizes additional essential internal methods that are supported by objects that may be called as functions. A function object is an object that supports the [[Call]] internal method. A constructor is an object that supports the [[Construct]] internal method. Every object that supports [[Construct]] must support [[Call]]; that is, every **constructor** must be a **function object**. Therefore, a **constructor** may also be referred to as a **constructor function** or **constructor function object**.

Table 6: Additional Essential Internal Methods of Function Objects

Internal Method	Signature	Description
[[Call]]	(<i>any</i> , a <i>List</i> of <i>any</i>) → <i>any</i>	Executes code associated with this object. Invoked via a function call expression. The arguments to the internal method are a this value and a list containing the arguments passed to the function by a call expression. Objects that implement this internal method are <i>callable</i> .
[[Construct]]	(a <i>List</i> of <i>any</i> , Object) → Object	Creates an object. Invoked via the new or super operators. The first argument to the internal method is a list containing the arguments of the operator. The second argument is the object to which the new operator was initially applied. Objects that implement this internal method are called <i>constructors</i> . A function object is not necessarily a constructor and such non- constructor function objects do not have a [[Construct]] internal method.

The semantics of the essential internal methods for ordinary objects and standard exotic objects are specified in clause 9. If any specified use of an internal method of an **exotic object** is not supported by an implementation, that usage must throw a **TypeError** exception when attempted.

6.1.7.3 Invariants of the Essential Internal Methods

The Internal Methods of Objects of an ECMAScript engine must conform to the list of invariants specified below. Ordinary ECMAScript Objects as well as all standard exotic objects in this specification maintain these invariants. ECMAScript Proxy objects maintain these invariants by means of runtime checks on the result of traps invoked on the [[ProxyHandler]] object.

If P 's attributes other than [[Writable]] may change over time or if the property might be deleted, then P 's [[Configurable]] attribute must be **true**.

If the [[Writable]] attribute may change from **false** to **true**, then the [[Configurable]] attribute must be **true**.

If the target is non-extensible and P is non-existent, then all future calls to [[GetOwnProperty]] (P) on the target must describe P as non-existent (i.e. [[GetOwnProperty]] (P) must return **undefined**).

NOTE 2

As a consequence of the third invariant, if a property is described as a **data property** and it may return different values over time, then either or both of the [[Writable]] and [[Configurable]] attributes must be **true** even if no mechanism to change the value is exposed via the other internal methods.

[[DefineOwnProperty]] ($P, Desc$)

The Type of the return value must be Boolean.

[[DefineOwnProperty]] must return **false** if P has previously been observed as a non-configurable own property of the target, unless either:

1. P is a writable **data property**. A non-configurable writable **data property** can be changed into a non-configurable non-writable **data property**.
2. All attributes of $Desc$ are the **SameValue** as P 's attributes.

[[DefineOwnProperty]] ($P, Desc$) must return **false** if target is non-extensible and P is a non-existent own property. That is, a non-extensible target object cannot be extended with new properties.

[[HasProperty]] (P)

The Type of the return value must be Boolean.

If P was previously observed as a non-configurable own **data** or **accessor property** of the target, [[HasProperty]] must return **true**.

[[Get]] ($P, Receiver$)

If P was previously observed as a non-configurable, non-writable own **data property** of the target with value V , then [[Get]] must return the **SameValue** as V .

If P was previously observed as a non-configurable own **accessor property** of the target whose [[Get]] attribute is **undefined**, the [[Get]] operation must return **undefined**.

[[Set]] ($P, V, Receiver$)

The Type of the return value must be Boolean.

If P was previously observed as a non-configurable, non-writable own **data property** of the target, then [[Set]] must return **false** unless V is the **SameValue** as P 's [[Value]] attribute.

If P was previously observed as a non-configurable own **accessor property** of the target whose [[Set]] attribute is **undefined**, the [[Set]] operation must return **false**.

[[Delete]] (P)

The Type of the return value must be Boolean.

If P was previously observed as a non-configurable own **data** or **accessor property** of the target, [[Delete]] must return **false**.

[[OwnPropertyKeys]] ()

The return value must be a **List**.

%ArrayProto_values%	Array.prototype.values	The initial value of the values data property of %ArrayPrototype% (22.1.3.32)
%AsyncFromSyncIteratorPrototype%		The prototype of async-from-sync iterator objects (25.1.4)
%AsyncFunction%		The constructor of async function objects (25.7.1)
%AsyncFunctionPrototype%		The initial value of the prototype data property of %AsyncFunction%
%AsyncGenerator%		The initial value of the prototype property of %AsyncGeneratorFunction%
%AsyncGeneratorFunction%		The constructor of async iterator objects (25.3.1)
%AsyncGeneratorPrototype%		The initial value of the prototype property of %AsyncGenerator%
%AsyncIteratorPrototype%		An object that all standard built-in async iterator objects indirectly inherit from
%Atomics%	Atomics	The Atomics object (24.4)
%Boolean%	Boolean	The Boolean constructor (19.3.1)
%BooleanPrototype%	Boolean.prototype	The initial value of the prototype data property of %Boolean% (19.3.3)
%DataView%	DataView	The DataView constructor (24.3.2)
%DataViewPrototype%	DataView.prototype	The initial value of the prototype data property of %DataView%
%Date%	Date	The Date constructor (20.3.2)
%DatePrototype%	Date.prototype	The initial value of the prototype data property of %Date%.
%decodeURI%	decodeURI	The decodeURI function (18.2.6.2)

%GeneratorPrototype%		The initial value of the prototype data property of %Generator%
%Int8Array%	Int8Array	The Int8Array constructor (22.2)
%Int8ArrayPrototype%	Int8Array.prototype	The initial value of the prototype data property of %Int8Array%
%Int16Array%	Int16Array	The Int16Array constructor (22.2)
%Int16ArrayPrototype%	Int16Array.prototype	The initial value of the prototype data property of %Int16Array%
%Int32Array%	Int32Array	The Int32Array constructor (22.2)
%Int32ArrayPrototype%	Int32Array.prototype	The initial value of the prototype data property of %Int32Array%
%isFinite%	isFinite	The isFinite function (18.2.2)
%isNaN%	isNaN	The isNaN function (18.2.3)
%IteratorPrototype%		An object that all standard built-in iterator objects indirectly inherit from
%JSON%	JSON	The JSON object (24.5)
%JSONParse%	JSON.parse	The initial value of the parse data property of %JSON%
%JSONStringify%	JSON.stringify	The initial value of the stringify data property of %JSON%
%Map%	Map	The Map constructor (23.1.1)
%MapIteratorPrototype%		The prototype of Map iterator objects (23.1.5)
%MapPrototype%	Map.prototype	The initial value of the prototype data property of %Map%
%Math%	Math	The Math object (20.2)
%Number%	Number	The Number constructor (20.1.1)

%ReferenceError%	ReferenceError	The ReferenceError constructor (19.5.5.3)
%ReferenceErrorPrototype%	ReferenceError.prototype	The initial value of the prototype data property of %ReferenceError%
%Reflect%	Reflect	The Reflect object (26.1)
%RegExp%	RegExp	The RegExp constructor (21.2.3)
%RegExpPrototype%	RegExp.prototype	The initial value of the prototype data property of %RegExp%
%Set%	Set	The Set constructor (23.2.1)
%SetIteratorPrototype%		The prototype of Set iterator objects (23.2.5)
%SetPrototype%	Set.prototype	The initial value of the prototype data property of %Set%
%SharedArrayBuffer%	SharedArrayBuffer	The SharedArrayBuffer constructor (24.2.2)
%SharedArrayBufferPrototype%	SharedArrayBuffer.prototype	The initial value of the prototype data property of %SharedArrayBuffer%
%String%	String	The String constructor (21.1.1)
%StringIteratorPrototype%		The prototype of String iterator objects (21.1.5)
%StringPrototype%	String.prototype	The initial value of the prototype data property of %String%
%Symbol%	Symbol	The Symbol constructor (19.4.1)
%SymbolPrototype%	Symbol.prototype	The initial value of the prototype data property of %Symbol% (19.4.3)
%SyntaxError%	SyntaxError	The SyntaxError constructor (19.5.5.4)
%SyntaxErrorPrototype%	SyntaxError.prototype	The initial value of the prototype data property of %SyntaxError%

%WeakMap%	WeakMap	The WeakMap constructor (23.3.1)
%WeakMapPrototype%	WeakMap.prototype	The initial value of the prototype data property of %WeakMap%
%WeakSet%	WeakSet	The WeakSet constructor (23.4.1)
%WeakSetPrototype%	WeakSet.prototype	The initial value of the prototype data property of %WeakSet%

6.2 ECMAScript Specification Types

A specification type corresponds to meta-values that are used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The specification types include [Reference](#), [List](#), [Completion](#), [Property Descriptor](#), [Lexical Environment](#), [Environment Record](#), and [Data Block](#). Specification type values are specification artefacts that do not necessarily correspond to any specific entity within an ECMAScript implementation. Specification type values may be used to describe intermediate results of ECMAScript expression evaluation but such values cannot be stored as properties of objects or values of ECMAScript language variables.

6.2.1 The List and Record Specification Types

The List type is used to explain the evaluation of argument lists (see 12.3.6) in **new** expressions, in function calls, and in other algorithms where a simple ordered list of values is needed. Values of the List type are simply ordered sequences of list elements containing the individual values. These sequences may be of any length. The elements of a list may be randomly accessed using 0-origin indices. For notational convenience an array-like syntax can be used to access List elements. For example, *arguments*[2] is shorthand for saying the 3rd element of the List *arguments*.

For notational convenience within this specification, a literal syntax can be used to express a new List value. For example, « 1, 2 » defines a List value that has two elements each of which is initialized to a specific value. A new empty List can be expressed as « ».

The Record type is used to describe data aggregations within the algorithms of this specification. A Record type value consists of one or more named fields. The value of each field is either an ECMAScript value or an abstract value represented by a name associated with the Record type. Field names are always enclosed in double brackets, for example [[Value]].

For notational convenience within this specification, an object literal-like syntax can be used to express a Record value. For example, { [[Field1]]: 42, [[Field2]]: **false**, [[Field3]]: **empty** } defines a Record value that has three fields, each of which is initialized to a specific value. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

In specification text and algorithms, dot notation may be used to refer to a specific field of a Record value. For example, if R is the record shown in the previous paragraph then R.[[Field2]] is shorthand for “the field of R named [[Field2]]”.

Schema for commonly used Record field combinations may be named, and that name may be used as a prefix to a literal Record value to identify the specific kind of aggregations that is being described. For example: **PropertyDescriptor** { [[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true** }.

6.2.3.1 Await

Algorithm steps that say

1. Let *completion* be `Await(value)`.

mean the same thing as:

1. Let *asyncContext* be the `running execution context`.
2. Let *promise* be `? PromiseResolve(%Promise%, « value »)`.
3. Let *stepsFulfilled* be the algorithm steps defined in [Await Fulfilled Functions](#).
4. Let *onFulfilled* be `CreateBuiltInFunction(stepsFulfilled, « [[AsyncContext]] »)`.
5. Set *onFulfilled*.`[[AsyncContext]]` to *asyncContext*.
6. Let *stepsRejected* be the algorithm steps defined in [Await Rejected Functions](#).
7. Let *onRejected* be `CreateBuiltInFunction(stepsRejected, « [[AsyncContext]] »)`.
8. Set *onRejected*.`[[AsyncContext]]` to *asyncContext*.
9. Perform `! PerformPromiseThen(promise, onFulfilled, onRejected)`.
10. Remove *asyncContext* from the `execution context stack` and restore the `execution context` that is at the top of the `execution context stack` as the `running execution context`.
11. Set the code evaluation state of *asyncContext* such that when evaluation is resumed with a `Completion completion`, the following steps of the algorithm that invoked `Await` will be performed, with *completion* available.
12. Return.
13. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of *asyncContext*.

where all variables in the above steps, with the exception of *completion*, are ephemeral and visible only in the steps pertaining to `Await`.

NOTE

`Await` can be combined with the `?` and `!` prefixes, so that for example

1. Let *result* be `? Await(value)`.

means the same thing as:

1. Let *result* be `Await(value)`.
2. `ReturnIfAbrupt(result)`.

6.2.3.1.1 Await Fulfilled Functions

An `Await` fulfilled function is an anonymous built-in function that is used as part of the `Await` specification device to deliver the promise fulfillment value to the caller as a normal completion. Each `Await` fulfilled function has an `[[AsyncContext]]` internal slot.

When an `Await` fulfilled function is called with argument *value*, the following steps are taken:

1. Let *F* be the `active function object`.
2. Let *asyncContext* be *F*.`[[AsyncContext]]`.
3. Let *prevContext* be the `running execution context`.
4. `Suspend prevContext`.
5. Push *asyncContext* onto the `execution context stack`; *asyncContext* is now the `running execution context`.

1. **Assert:** If *completionRecord*.[[Type]] is either `return` or `throw`, then *completionRecord*.[[Value]] is not empty.
2. If *completionRecord*.[[Value]] is not empty, return `Completion(completionRecord)`.
3. Return `Completion { [[Type]]: completionRecord.[[Type]], [[Value]]: value, [[Target]]: completionRecord, [[Target]] }`.

6.2.4 The Reference Specification Type

NOTE

The Reference type is used to explain the behaviour of such operators as `delete`, `typeof`, the assignment operators, the `super` keyword and other language features. For example, the left-hand operand of an assignment is expected to produce a reference.

A Reference is a resolved name or property binding. A Reference consists of three components, the base value component, the referenced name component, and the Boolean-valued strict reference flag. The base value component is either `undefined`, an Object, a Boolean, a String, a Symbol, a Number, or an `Environment Record`. A base value component of `undefined` indicates that the Reference could not be resolved to a binding. The referenced name component is a String or Symbol value.

A Super Reference is a Reference that is used to represent a name binding that was expressed using the `super` keyword. A `Super Reference` has an additional `thisValue` component, and its base value component will never be an `Environment Record`.

The following `abstract operations` are used in this specification to operate on references:

6.2.4.1 GetBase (*V*)

1. **Assert:** `Type(V)` is `Reference`.
2. Return the base value component of *V*.

6.2.4.2 GetReferencedName (*V*)

1. **Assert:** `Type(V)` is `Reference`.
2. Return the referenced name component of *V*.

6.2.4.3 IsStrictReference (*V*)

1. **Assert:** `Type(V)` is `Reference`.
2. Return the strict reference flag of *V*.

6.2.4.4 HasPrimitiveBase (*V*)

1. **Assert:** `Type(V)` is `Reference`.
2. If `Type(V)`'s base value component is Boolean, String, Symbol, or Number, return `true`; otherwise return `false`.

6.2.4.5 IsPropertyReference (*V*)

1. **Assert:** `Type(V)` is `Reference`.
2. If either the base value component of *V* is an Object or `HasPrimitiveBase(V)` is `true`, return `true`; otherwise return `false`.

NOTE

The object that may be created in step 6.a.ii is not accessible outside of the above algorithm and the ordinary object [[Set]] internal method. An implementation might choose to avoid the actual creation of that object.

6.2.4.10 GetThisValue (V)

1. **Assert:** IsPropertyReference(V) is **true**.
2. If IsSuperReference(V) is **true**, then
 - a. Return the value of the thisValue component of the reference V .
3. Return GetBase(V).

6.2.4.11 InitializeReferencedBinding (V, W)

1. ReturnIfAbrupt(V).
2. ReturnIfAbrupt(W).
3. **Assert:** Type(V) is Reference.
4. **Assert:** IsUnresolvableReference(V) is **false**.
5. Let $base$ be GetBase(V).
6. **Assert:** $base$ is an Environment Record.
7. Return $base$.InitializeBinding(GetReferencedName(V), W).

6.2.5 The Property Descriptor Specification Type

The Property Descriptor type is used to explain the manipulation and reification of Object property attributes. Values of the Property Descriptor type are Records. Each field's name is an attribute name and its value is a corresponding attribute value as specified in 6.1.7.1. In addition, any field may be present or absent. The schema name used within this specification to tag literal descriptions of Property Descriptor records is “PropertyDescriptor”.

Property Descriptor values may be further classified as data Property Descriptors and accessor Property Descriptors based upon the existence or use of certain fields. A data Property Descriptor is one that includes any fields named either [[Value]] or [[Writable]]. An accessor Property Descriptor is one that includes any fields named either [[Get]] or [[Set]]. Any Property Descriptor may have fields named [[Enumerable]] and [[Configurable]]. A Property Descriptor value may not be both a data Property Descriptor and an accessor Property Descriptor; however, it may be neither. A generic Property Descriptor is a Property Descriptor value that is neither a data Property Descriptor nor an accessor Property Descriptor. A fully populated Property Descriptor is one that is either an accessor Property Descriptor or a data Property Descriptor and that has all of the fields that correspond to the property attributes defined in either Table 2 or Table 3.

The following **abstract operations** are used in this specification to operate upon Property Descriptor values:

6.2.5.1 IsAccessorDescriptor ($Desc$)

When the abstract operation IsAccessorDescriptor is called with **Property Descriptor** $Desc$, the following steps are taken:

1. If $Desc$ is **undefined**, return **false**.
2. If both $Desc$.[[Get]] and $Desc$.[[Set]] are absent, return **false**.
3. Return **true**.

6.2.5.2 IsDataDescriptor ($Desc$)

- b. Set `desc`.`[[Configurable]]` to `configurable`.
- 7. Let `hasValue` be `? HasProperty(Obj, "value")`.
- 8. If `hasValue` is `true`, then
 - a. Let `value` be `? Get(Obj, "value")`.
 - b. Set `desc`.`[[Value]]` to `value`.
- 9. Let `hasWritable` be `? HasProperty(Obj, "writable")`.
- 10. If `hasWritable` is `true`, then
 - a. Let `writable` be `ToBoolean(? Get(Obj, "writable"))`.
 - b. Set `desc`.`[[Writable]]` to `writable`.
- 11. Let `hasGet` be `? HasProperty(Obj, "get")`.
- 12. If `hasGet` is `true`, then
 - a. Let `getter` be `? Get(Obj, "get")`.
 - b. If `IsCallable(getter)` is `false` and `getter` is not `undefined`, throw a `TypeError` exception.
 - c. Set `desc`.`[[Get]]` to `getter`.
- 13. Let `hasSet` be `? HasProperty(Obj, "set")`.
- 14. If `hasSet` is `true`, then
 - a. Let `setter` be `? Get(Obj, "set")`.
 - b. If `IsCallable(setter)` is `false` and `setter` is not `undefined`, throw a `TypeError` exception.
 - c. Set `desc`.`[[Set]]` to `setter`.
- 15. If `desc`.`[[Get]]` is present or `desc`.`[[Set]]` is present, then
 - a. If `desc`.`[[Value]]` is present or `desc`.`[[Writable]]` is present, throw a `TypeError` exception.
- 16. Return `desc`.

6.2.5.6 CompletePropertyDescriptor (`Desc`)

When the abstract operation `CompletePropertyDescriptor` is called with `Property Descriptor Desc`, the following steps are taken:

- 1. **Assert:** `Desc` is a `Property Descriptor`.
- 2. Let `like` be `Record { [[Value]]: undefined, [[Writable]]: false, [[Get]]: undefined, [[Set]]: undefined, [[Enumerable]]: false, [[Configurable]]: false }`.
- 3. If `IsGenericDescriptor(Desc)` is `true` or `IsDataDescriptor(Desc)` is `true`, then
 - a. If `Desc` does not have a `[[Value]]` field, set `Desc`.`[[Value]]` to `like`.`[[Value]]`.
 - b. If `Desc` does not have a `[[Writable]]` field, set `Desc`.`[[Writable]]` to `like`.`[[Writable]]`.
- 4. Else,
 - a. If `Desc` does not have a `[[Get]]` field, set `Desc`.`[[Get]]` to `like`.`[[Get]]`.
 - b. If `Desc` does not have a `[[Set]]` field, set `Desc`.`[[Set]]` to `like`.`[[Set]]`.
- 5. If `Desc` does not have an `[[Enumerable]]` field, set `Desc`.`[[Enumerable]]` to `like`.`[[Enumerable]]`.
- 6. If `Desc` does not have a `[[Configurable]]` field, set `Desc`.`[[Configurable]]` to `like`.`[[Configurable]]`.
- 7. Return `Desc`.

6.2.6 The Lexical Environment and Environment Record Specification Types

The `Lexical Environment` and `Environment Record` types are used to explain the behaviour of name resolution in nested functions and blocks. These types and the operations upon them are defined in 8.1.

6.2.7 Data Blocks

6.2.7.3 CopyDataBlockBytes (*toBlock*, *toIndex*, *fromBlock*, *fromIndex*, *count*)

When the abstract operation CopyDataBlockBytes is called, the following steps are taken:

1. **Assert:** *fromBlock* and *toBlock* are distinct Data Block or Shared Data Block values.
2. **Assert:** *fromIndex*, *toIndex*, and *count* are integer values ≥ 0 .
3. Let *fromSize* be the number of bytes in *fromBlock*.
4. **Assert:** *fromIndex* + *count* \leq *fromSize*.
5. Let *toSize* be the number of bytes in *toBlock*.
6. **Assert:** *toIndex* + *count* \leq *toSize*.
7. Repeat, while *count* > 0
 - a. If *fromBlock* is a Shared Data Block, then
 - i. Let *execution* be the [[CandidateExecution]] field of the surrounding agent's Agent Record.
 - ii. Let *eventList* be the [[EventList]] field of the element in *execution*.[[EventsRecords]] whose [[AgentSignifier]] is *AgentSignifier*().
 - iii. Let *bytes* be a List of length 1 that contains a nondeterministically chosen byte value.
 - iv. NOTE: In implementations, *bytes* is the result of a non-atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the memory model to describe observable behaviour of hardware with weak consistency.
 - v. Let *readEvent* be `ReadSharedMemory { [[Order]]: "Unordered", [[NoTear]]: true, [[Block]]: fromBlock, [[ByteIndex]]: fromIndex, [[ElementSize]]: 1 }`.
 - vi. Append *readEvent* to *eventList*.
 - vii. Append Chosen Value Record { [[Event]]: *readEvent*, [[ChosenValue]]: *bytes* } to *execution*.[[ChosenValues]].
 - viii. If *toBlock* is a Shared Data Block, then
 1. Append `WriteSharedMemory { [[Order]]: "Unordered", [[NoTear]]: true, [[Block]]: toBlock, [[ByteIndex]]: toIndex, [[ElementSize]]: 1, [[Payload]]: bytes }` to *eventList*.
 - ix. Else,
 1. Set *toBlock[toIndex]* to *bytes[0]*.
 - b. Else,
 - i. **Assert:** *toBlock* is not a Shared Data Block.
 - ii. Set *toBlock[toIndex]* to *fromBlock[fromIndex]*.
 - c. Increment *toIndex* and *fromIndex* each by 1.
 - d. Decrement *count* by 1.
8. Return `NormalCompletion(empty)`.

7 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification.

7.1 Type Conversion

The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; they can accept a value of any ECMAScript language type. But no other specification types are used with

The abstract operation ToBoolean converts *argument* to a value of type Boolean according to [Table 9](#):

Table 9: ToBoolean Conversions

Argument Type	Result
Undefined	Return false .
Null	Return false .
Boolean	Return <i>argument</i> .
Number	If <i>argument</i> is +0 , -0 , or NaN , return false ; otherwise return true .
String	If <i>argument</i> is the empty String (its length is zero), return false ; otherwise return true .
Symbol	Return true .
Object	Return true .

7.1.3 ToNumber (*argument*)

The abstract operation ToNumber converts *argument* to a value of type Number according to [Table 10](#):

Table 10: ToNumber Conversions

Argument Type	Result
Undefined	Return NaN .
Null	Return +0 .
Boolean	If <i>argument</i> is true , return 1. If <i>argument</i> is false , return +0 .
Number	Return <i>argument</i> (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a TypeError exception.
Object	Apply the following steps: <ol style="list-style-type: none"> Let <i>primValue</i> be ? ToPrimitive(<i>argument</i>, hint Number). Return ? ToNumber(<i>primValue</i>).

7.1.3.1 ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points ([6.1.4](#)). If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of **ToNumber** is **NaN**.

NOTE 1

The terminal symbols of this grammar are all composed of characters in the Unicode Basic Multilingual Plane (BMP). Therefore, the result of **ToNumber** will be **NaN** if the string contains any **leading surrogate** or **trailing surrogate** code

of Number type is given here. This value is determined in two steps: first, a mathematical value (MV) is derived from the String numeric literal; second, this mathematical value is rounded as described below. The MV on any grammar symbol, not provided below, is the MV for that symbol defined in 11.8.3.1.

The MV of `StringNumericLiteral` :::: [empty] is 0.

The MV of `StringNumericLiteral` :::: `StrWhiteSpace` is 0.

The MV of `StringNumericLiteral` :::: `StrWhiteSpace StrNumericLiteral StrWhiteSpace` is the MV of `StrNumericLiteral`, no matter whether white space is present or not.

The MV of `StrNumericLiteral` :::: `StrDecimalLiteral` is the MV of `StrDecimalLiteral`.

The MV of `StrNumericLiteral` :::: `BinaryIntegerLiteral` is the MV of `BinaryIntegerLiteral`.

The MV of `StrNumericLiteral` :::: `OctalIntegerLiteral` is the MV of `OctalIntegerLiteral`.

The MV of `StrNumericLiteral` :::: `HexIntegerLiteral` is the MV of `HexIntegerLiteral`.

The MV of `StrDecimalLiteral` :::: `StrUnsignedDecimalLiteral` is the MV of `StrUnsignedDecimalLiteral`.

The MV of `StrDecimalLiteral` :::: `+ StrUnsignedDecimalLiteral` is the MV of `StrUnsignedDecimalLiteral`.

The MV of `StrDecimalLiteral` :::: `- StrUnsignedDecimalLiteral` is the negative of the MV of `StrUnsignedDecimalLiteral`. (Note that if the MV of `StrUnsignedDecimalLiteral` is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this signless mathematical zero to a floating-point `+0` or `-0` as appropriate.)

The MV of `StrUnsignedDecimalLiteral` :::: `Infinity` is 10^{10000} (a value so large that it will round to $+\infty$).

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits .` is the MV of `DecimalDigits`.

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits . DecimalDigits` is the MV of the first `DecimalDigits` plus (the MV of the second `DecimalDigits` times 10^{-n}), where `n` is the number of code points in the second `DecimalDigits`.

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits . ExponentPart` is the MV of `DecimalDigits` times 10^e , where `e` is the MV of `ExponentPart`.

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits . DecimalDigits ExponentPart` is (the MV of the first `DecimalDigits` plus (the MV of the second `DecimalDigits` times 10^{-n})) times 10^e , where `n` is the number of code points in the second `DecimalDigits` and `e` is the MV of `ExponentPart`.

The MV of `StrUnsignedDecimalLiteral` :::: `. DecimalDigits` is the MV of `DecimalDigits` times 10^{-n} , where `n` is the number of code points in `DecimalDigits`.

The MV of `StrUnsignedDecimalLiteral` :::: `. DecimalDigits ExponentPart` is the MV of `DecimalDigits` times 10^{e-n} , where `n` is the number of code points in `DecimalDigits` and `e` is the MV of `ExponentPart`.

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits` is the MV of `DecimalDigits`.

The MV of `StrUnsignedDecimalLiteral` :::: `DecimalDigits ExponentPart` is the MV of `DecimalDigits` times 10^e , where `e` is the MV of `ExponentPart`.

Once the exact MV for a String numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is `+0` unless the first non white space code point in the String numeric literal is `"-"`, in which case the rounded value is `-0`. Otherwise, the rounded value must be the Number value for the MV (in the sense defined in 6.1.6), unless the literal includes a `StrUnsignedDecimalLiteral` and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is significant if it is not part of an `ExponentPart` and

it is not `0`; or

there is a nonzero digit to its left and there is a nonzero digit, not in the `ExponentPart`, to its right.

∞ are mapped to **+0**.)

ToUint32 maps **-0** to **+0**.

7.1.7ToInt16 (*argument*)

The abstract operation ToInt16 converts *argument* to one of 2^{16} integer values in the range -32768 through 32767, inclusive. This abstract operation functions as follows:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is **NaN**, **+0**, **-0**, $+\infty$, or $-\infty$, return **+0**.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is `floor(abs(number))`.
4. Let *int16bit* be *int* modulo 2^{16} .
5. If *int16bit* $\geq 2^{15}$, return *int16bit* - 2^{16} ; otherwise return *int16bit*.

7.1.8ToUint16 (*argument*)

The abstract operation ToUint16 converts *argument* to one of 2^{16} integer values in the range 0 through $2^{16} - 1$, inclusive. This abstract operation functions as follows:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is **NaN**, **+0**, **-0**, $+\infty$, or $-\infty$, return **+0**.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is `floor(abs(number))`.
4. Let *int16bit* be *int* modulo 2^{16} .
5. Return *int16bit*.

NOTE

Given the above definition of ToUint16:

The substitution of 2^{16} for 2^{32} in step 4 is the only difference between ToUint32 and ToUint16.

ToUint16 maps **-0** to **+0**.

7.1.9ToInt8 (*argument*)

The abstract operationToInt8 converts *argument* to one of 2^8 integer values in the range -128 through 127, inclusive. This abstract operation functions as follows:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is **NaN**, **+0**, **-0**, $+\infty$, or $-\infty$, return **+0**.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is `floor(abs(number))`.
4. Let *int8bit* be *int* modulo 2^8 .
5. If *int8bit* $\geq 2^7$, return *int8bit* - 2^8 ; otherwise return *int8bit*.

7.1.10ToUint8 (*argument*)

Symbol	Throw a TypeError exception.
Object	Apply the following steps:
	<ol style="list-style-type: none"> 1. Let <i>primValue</i> be ? ToPrimitive(<i>argument</i>, hint String). 2. Return ? ToString(<i>primValue</i>).

7.1.12.1 NumberToString (*m*)

The abstract operation NumberToString converts a Number *m* to String format as follows:

1. If *m* is **NaN**, return the String "**NaN**".
2. If *m* is **+0** or **-0**, return the String "0".
3. If *m* is less than zero, return the **string-concatenation** of "-" and ! **NumberToString**(-*m*).
4. If *m* is **+∞**, return the String "**Infinity**".
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the **string-concatenation** of:
 - the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes)
 - n - k* occurrences of the code unit 0x0030 (DIGIT ZERO)
7. If $0 < n \leq 21$, return the **string-concatenation** of:
 - the code units of the most significant *n* digits of the decimal representation of *s*
 - the code unit 0x002E (FULL STOP)
 - the code units of the remaining *k - n* digits of the decimal representation of *s*
8. If $-6 < n \leq 0$, return the **string-concatenation** of:
 - the code unit 0x0030 (DIGIT ZERO)
 - the code unit 0x002E (FULL STOP)
 - n* occurrences of the code unit 0x0030 (DIGIT ZERO)
 - the code units of the *k* digits of the decimal representation of *s*
9. Otherwise, if *k* = 1, return the **string-concatenation** of:
 - the code unit of the single digit of *s*
 - the code unit 0x0065 (LATIN SMALL LETTER E)
 - the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n - 1* is positive or negative
 - the code units of the decimal representation of the integer **abs**(*n* - 1) (with no leading zeroes)
10. Return the **string-concatenation** of:
 - the code units of the most significant digit of the decimal representation of *s*
 - the code unit 0x002E (FULL STOP)
 - the code units of the remaining *k - 1* digits of the decimal representation of *s*
 - the code unit 0x0065 (LATIN SMALL LETTER E)
 - the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n - 1* is positive or negative
 - the code units of the decimal representation of the integer **abs**(*n* - 1) (with no leading zeroes)

NOTE 1

The following observations may be useful as guidelines for implementations, but are not part of the normative

7.1.14 ToPropertyKey (*argument*)

The abstract operation ToPropertyKey converts *argument* to a value that can be used as a property key by performing the following steps:

1. Let *key* be ? ToPrimitive(*argument*, hint String).
2. If Type(*key*) is Symbol, then
 - a. Return *key*.
3. Return ! ToString(*key*).

7.1.15 ToLength (*argument*)

The abstract operation ToLength converts *argument* to an integer suitable for use as the length of an array-like object. It performs the following steps:

1. Let *len* be ? ToInteger(*argument*).
2. If *len* $\leq +0$, return **+0**.
3. Return min(*len*, $2^{53} - 1$).

7.1.16 CanonicalNumericIndexString (*argument*)

The abstract operation CanonicalNumericIndexString returns *argument* converted to a numeric value if it is a String representation of a Number that would be produced by **ToString**, or the string "**-0**". Otherwise, it returns **undefined**. This abstract operation functions as follows:

1. **Assert:** Type(*argument*) is String.
2. If *argument* is "**-0**", return **-0**.
3. Let *n* be ! ToNumber(*argument*).
4. If SameValue(! ToString(*n*), *argument*) is **false**, return **undefined**.
5. Return *n*.

A *canonical numeric string* is any String value for which the CanonicalNumericIndexString abstract operation does not return **undefined**.

7.1.17 ToIndex (*value*)

The abstract operation ToIndex returns *value* argument converted to a numeric value if it is a valid **integer index** value. This abstract operation functions as follows:

1. If *value* is **undefined**, then
 - a. Let *index* be 0.
2. Else,
 - a. Let *integerIndex* be ? ToInteger(*value*).
 - b. If *integerIndex* < 0 , throw a **RangeError** exception.
 - c. Let *index* be ! ToLength(*integerIndex*).
 - d. If SameValueZero(*integerIndex*, *index*) is **false**, throw a **RangeError** exception.
3. Return *index*.

2. If *argument* has a [[Construct]] internal method, return **true**.
3. Return **false**.

7.2.5 IsExtensible (*O*)

The abstract operation IsExtensible is used to determine whether additional properties can be added to the object that is *O*. A Boolean value is returned. This abstract operation performs the following steps:

1. **Assert:** *Type(O)* is Object.
2. Return ? *O*.[[IsExtensible]]().

7.2.6 IsInteger (*argument*)

The abstract operation IsInteger determines if *argument* is a finite integer numeric value.

1. If *Type(argument)* is not Number, return **false**.
2. If *argument* is NaN, +∞, or -∞, return **false**.
3. If *floor(abs(argument))* ≠ *abs(argument)*, return **false**.
4. Return **true**.

7.2.7 IsPropertyKey (*argument*)

The abstract operation IsPropertyKey determines if *argument*, which must be an ECMAScript language value, is a value that may be used as a property key.

1. If *Type(argument)* is String, return **true**.
2. If *Type(argument)* is Symbol, return **true**.
3. Return **false**.

7.2.8 IsRegExp (*argument*)

The abstract operation IsRegExp with argument *argument* performs the following steps:

1. If *Type(argument)* is not Object, return **false**.
2. Let *matcher* be ? *Get(argument, @@match)*.
3. If *matcher* is not **undefined**, return *ToBoolean(matcher)*.
4. If *argument* has a [[RegExpMatcher]] internal slot, return **true**.
5. Return **false**.

7.2.9 IsStringPrefix (*p, q*)

The abstract operation IsStringPrefix determines if String *p* is a prefix of String *q*.

1. **Assert:** *Type(p)* is String.
2. **Assert:** *Type(q)* is String.
3. If *q* can be the string-concatenation of *p* and some other String *r*, return **true**. Otherwise, return **false**.
4. NOTE: Any String is a prefix of itself, because *r* may be the empty String.

7. If `Type(x)` is Symbol, then
 - a. If `x` and `y` are both the same Symbol value, return `true`; otherwise, return `false`.
8. If `x` and `y` are the same Object value, return `true`. Otherwise, return `false`.

7.2.13 Abstract Relational Comparison

The comparison `x < y`, where `x` and `y` are values, produces `true`, `false`, or `undefined` (which indicates that at least one operand is `NaN`). In addition to `x` and `y` the algorithm takes a Boolean flag named `LeftFirst` as a parameter. The flag is used to control the order in which operations with potentially visible side-effects are performed upon `x` and `y`. It is necessary because ECMAScript specifies left to right evaluation of expressions. The default value of `LeftFirst` is `true` and indicates that the `x` parameter corresponds to an expression that occurs to the left of the `y` parameter's corresponding expression. If `LeftFirst` is `false`, the reverse is the case and operations must be performed upon `y` before `x`. Such a comparison is performed as follows:

1. If the `LeftFirst` flag is `true`, then
 - a. Let `px` be ? `ToPrimitive(x, hint Number)`.
 - b. Let `py` be ? `ToPrimitive(y, hint Number)`.
2. Else the order of evaluation needs to be reversed to preserve left to right evaluation,
 - a. Let `py` be ? `ToPrimitive(y, hint Number)`.
 - b. Let `px` be ? `ToPrimitive(x, hint Number)`.
3. If `Type(px)` is String and `Type(py)` is String, then
 - a. If `IsStringPrefix(py, px)` is `true`, return `false`.
 - b. If `IsStringPrefix(px, py)` is `true`, return `true`.
 - c. Let `k` be the smallest nonnegative integer such that the code unit at index `k` within `px` is different from the code unit at index `k` within `py`. (There must be such a `k`, for neither String is a prefix of the other.)
 - d. Let `m` be the integer that is the numeric value of the code unit at index `k` within `px`.
 - e. Let `n` be the integer that is the numeric value of the code unit at index `k` within `py`.
 - f. If `m < n`, return `true`. Otherwise, return `false`.
4. Else,
 - a. NOTE: Because `px` and `py` are primitive values evaluation order is not important.
 - b. Let `nx` be ? `ToNumber(px)`.
 - c. Let `ny` be ? `ToNumber(py)`.
 - d. If `nx` is `NaN`, return `undefined`.
 - e. If `ny` is `NaN`, return `undefined`.
 - f. If `nx` and `ny` are the same Number value, return `false`.
 - g. If `nx` is `+0` and `ny` is `-0`, return `false`.
 - h. If `nx` is `-0` and `ny` is `+0`, return `false`.
 - i. If `nx` is `+∞`, return `false`.
 - j. If `ny` is `+∞`, return `true`.
 - k. If `ny` is `-∞`, return `false`.
 - l. If `nx` is `-∞`, return `true`.
 - m. If the mathematical value of `nx` is less than the mathematical value of `ny`—note that these mathematical values are both finite and not both zero—return `true`. Otherwise, return `false`.

NOTE 1

Step 3 differs from step 7 in the algorithm for the addition operator + (12.8.3) by using the logical-and operation instead of the logical-or operation.

NOTE 2

The comparison of Strings uses a simple lexicographic ordering on sequences of code unit values. There is no attempt to

The abstract operation Get is used to retrieve the value of a specific property of an object. The operation is called with arguments O and P where O is the object and P is the property key. This abstract operation performs the following steps:

1. **Assert:** $\text{Type}(O)$ is Object.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. Return ? $O.[[Get]](P, O)$.

7.3.2 GetV (V, P)

The abstract operation GetV is used to retrieve the value of a specific property of an ECMAScript language value. If the value is not an object, the property lookup is performed using a wrapper object appropriate for the type of the value. The operation is called with arguments V and P where V is the value and P is the property key. This abstract operation performs the following steps:

1. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
2. Let O be ? $\text{ToObject}(V)$.
3. Return ? $O.[[Get]](P, V)$.

7.3.3 Set ($O, P, V, Throw$)

The abstract operation Set is used to set the value of a specific property of an object. The operation is called with arguments O , P , V , and $Throw$ where O is the object, P is the property key, V is the new value for the property and $Throw$ is a Boolean flag. This abstract operation performs the following steps:

1. **Assert:** $\text{Type}(O)$ is Object.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. **Assert:** $\text{Type}(Throw)$ is Boolean.
4. Let $success$ be ? $O.[[Set]](P, V, O)$.
5. If $success$ is **false** and $Throw$ is **true**, throw a **TypeError** exception.
6. Return $success$.

7.3.4 CreateDataProperty (O, P, V)

The abstract operation CreateDataProperty is used to create a new own property of an object. The operation is called with arguments O , P , and V where O is the object, P is the property key, and V is the value for the property. This abstract operation performs the following steps:

1. **Assert:** $\text{Type}(O)$ is Object.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. Let $newDesc$ be the PropertyDescriptor { [[Value]]: V , [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true** }.
4. Return ? $O.[[DefineOwnProperty]](P, newDesc)$.

NOTE

This abstract operation creates a property whose attributes are set to the same defaults used for properties created by the ECMAScript language assignment operator. Normally, the property will not already exist. If it does exist and is not configurable or if O is not extensible, [[DefineOwnProperty]] will return **false**.

7.3.8 DeletePropertyOrThrow (O , P)

The abstract operation `DeletePropertyOrThrow` is used to remove a specific own property of an object. It throws an exception if the property is not configurable. The operation is called with arguments O and P where O is the object and P is the property key. This abstract operation performs the following steps:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is true.
3. Let $success$ be $? O.[[Delete]](P)$.
4. If $success$ is false, throw a `TypeError` exception.
5. Return $success$.

7.3.9 GetMethod (V , P)

The abstract operation `GetMethod` is used to get the value of a specific property of an `ECMAScript language value` when the value of the property is expected to be a function. The operation is called with arguments V and P where V is the `ECMAScript language value`, P is the property key. This abstract operation performs the following steps:

1. **Assert:** `IsPropertyKey(P)` is true.
2. Let $func$ be $? GetV(V, P)$.
3. If $func$ is either `undefined` or `null`, return `undefined`.
4. If `IsCallable(func)` is false, throw a `TypeError` exception.
5. Return $func$.

7.3.10 HasProperty (O , P)

The abstract operation `HasProperty` is used to determine whether an object has a property with the specified property key. The property may be either an own or inherited. A Boolean value is returned. The operation is called with arguments O and P where O is the object and P is the property key. This abstract operation performs the following steps:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is true.
3. Return $? O.[[HasProperty]](P)$.

7.3.11 HasOwnProperty (O , P)

The abstract operation `HasOwnProperty` is used to determine whether an object has an own property with the specified property key. A Boolean value is returned. The operation is called with arguments O and P where O is the object and P is the property key. This abstract operation performs the following steps:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is true.
3. Let $desc$ be $? O.[[GetOwnProperty]](P)$.
4. If $desc$ is `undefined`, return false.
5. Return true.

7.3.12 Call (F , V [, $argumentsList$])

The abstract operation `Call` is used to call the `[[Call]]` internal method of a `function object`. The operation is called with

8. Return **true**.

7.3.15 TestIntegrityLevel (*O*, *level*)

The abstract operation `TestIntegrityLevel` is used to determine if the set of own properties of an object are fixed. This abstract operation performs the following steps:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `level` is either "sealed" or "frozen".
3. Let `status` be ? `IsExtensible(O)`.
4. If `status` is **true**, return **false**.
5. NOTE: If the object is extensible, none of its properties are examined.
6. Let `keys` be ? `O.[[OwnPropertyKeys]]()`.
7. For each element `k` of `keys`, do
 - a. Let `currentDesc` be ? `O.[[GetOwnProperty]](k)`.
 - b. If `currentDesc` is not **undefined**, then
 - i. If `currentDesc.[[Configurable]]` is **true**, return **false**.
 - ii. If `level` is "frozen" and `IsDataDescriptor(currentDesc)` is **true**, then
 1. If `currentDesc.[[Writable]]` is **true**, return **false**.
8. Return **true**.

7.3.16 CreateArrayFromList (*elements*)

The abstract operation `CreateArrayFromList` is used to create an `Array` object whose elements are provided by a `List`. This abstract operation performs the following steps:

1. **Assert:** `elements` is a `List` whose elements are all ECMAScript language values.
2. Let `array` be ! `ArrayCreate(0)`.
3. Let `n` be 0.
4. For each element `e` of `elements`, do
 - a. Let `status` be `CreateDataProperty(array, ! ToString(n), e)`.
 - b. **Assert:** `status` is **true**.
 - c. Increment `n` by 1.
5. Return `array`.

7.3.17 CreateListFromArrayLike (*obj* [, *elementTypes*])

The abstract operation `CreateListFromArrayLike` is used to create a `List` value whose elements are provided by the indexed properties of an array-like object, `obj`. The optional argument `elementTypes` is a `List` containing the names of ECMAScript Language Types that are allowed for element values of the `List` that is created. This abstract operation performs the following steps:

1. If `elementTypes` is not present, set `elementTypes` to « Undefined, Null, Boolean, String, Symbol, Number, Object ».
2. If `Type(obj)` is not Object, throw a `TypeError` exception.
3. Let `len` be ? `ToLength(? Get(obj, "length"))`.
4. Let `list` be a new empty `List`.
5. Let `index` be 0.
6. Repeat, while `index < len`
 - a. Let `indexName` be ! `ToString(index)`.

8. Throw a **TypeError** exception.

7.3.21 EnumerableOwnPropertyNames (*O, kind*)

When the abstract operation `EnumerableOwnPropertyNames` is called with Object *O* and String *kind* the following steps are taken:

1. **Assert:** `Type(O)` is Object.
2. Let *ownKeys* be ? *O*.[[OwnPropertyKeys]]().
3. Let *properties* be a new empty List.
4. For each element *key* of *ownKeys* in List order, do
 - a. If `Type(key)` is String, then
 - i. Let *desc* be ? *O*.[[GetProperty]](*key*).
 - ii. If *desc* is not **undefined** and *desc*.[[Enumerable]] is **true**, then
 1. If *kind* is "key", append *key* to *properties*.
 2. Else,
 - a. Let *value* be ? `Get(O, key)`.
 - b. If *kind* is "value", append *value* to *properties*.
 - c. Else,
 - i. **Assert:** *kind* is "key+value".
 - ii. Let *entry* be `CreateArrayFromList(« key, value »).`
 - iii. Append *entry* to *properties*.
 5. Order the elements of *properties* so they are in the same relative order as would be produced by the Iterator that would be returned if the `EnumerateObjectProperties` internal method were invoked with *O*.
 6. Return *properties*.

7.3.22 GetFunctionRealm (*obj*)

The abstract operation `GetFunctionRealm` with argument *obj* performs the following steps:

1. **Assert:** *obj* is a callable object.
2. If *obj* has a [[Realm]] internal slot, then
 - a. Return *obj*.[[Realm]].
3. If *obj* is a Bound Function exotic object, then
 - a. Let *target* be *obj*.[[BoundTargetFunction]].
 - b. Return ? `GetFunctionRealm(target)`.
4. If *obj* is a Proxy exotic object, then
 - a. If *obj*.[[ProxyHandler]] is **null**, throw a **TypeError** exception.
 - b. Let *proxyTarget* be *obj*.[[ProxyTarget]].
 - c. Return ? `GetFunctionRealm(proxyTarget)`.
5. Return the current Realm Record.

NOTE

Step 5 will only be reached if *obj* is a non-standard function exotic object that does not have a [[Realm]] internal slot.

7.3.23 CopyDataProperties (*target, source, excludedItems*)

When the abstract operation `CopyDataProperties` is called with arguments *target*, *source*, and *excludedItems*, the

7.4.2 IteratorNext (*iteratorRecord* [, *value*])

The abstract operation IteratorNext with argument *iteratorRecord* and optional argument *value* performs the following steps:

1. If *value* is not present, then
 - a. Let *result* be ? Call(*iteratorRecord*.[[NextMethod]], *iteratorRecord*.[[Iterator]], « »).
2. Else,
 - a. Let *result* be ? Call(*iteratorRecord*.[[NextMethod]], *iteratorRecord*.[[Iterator]], « *value* »).
3. If Type(*result*) is not Object, throw a **TypeError** exception.
4. Return *result*.

7.4.3 IteratorComplete (*iterResult*)

The abstract operation IteratorComplete with argument *iterResult* performs the following steps:

1. **Assert:** Type(*iterResult*) is Object.
2. Return ToBoolean(? Get(*iterResult*, "done")).

7.4.4 IteratorValue (*iterResult*)

The abstract operation IteratorValue with argument *iterResult* performs the following steps:

1. **Assert:** Type(*iterResult*) is Object.
2. Return ? Get(*iterResult*, "value").

7.4.5 IteratorStep (*iteratorRecord*)

The abstract operation IteratorStep with argument *iteratorRecord* requests the next value from *iteratorRecord*.[[Iterator]] by calling *iteratorRecord*.[[NextMethod]] and returns either **false** indicating that the iterator has reached its end or the IteratorResult object if a next value is available. IteratorStep performs the following steps:

1. Let *result* be ? IteratorNext(*iteratorRecord*).
2. Let *done* be ? IteratorComplete(*result*).
3. If *done* is **true**, return **false**.
4. Return *result*.

7.4.6 IteratorClose (*iteratorRecord*, *completion*)

The abstract operation IteratorClose with arguments *iteratorRecord* and *completion* is used to notify an iterator that it should perform any actions it would normally perform when it has reached its completed state:

1. **Assert:** Type(*iteratorRecord*.[[Iterator]]) is Object.
2. **Assert:** *completion* is a Completion Record.
3. Let *iterator* be *iteratorRecord*.[[Iterator]].
4. Let *return* be ? GetMethod(*iterator*, "return").
5. If *return* is **undefined**, return Completion(*completion*).
6. Let *innerResult* be Call(*return*, *iterator*, « »).
7. If *completion*.[[Type]] is throw, return Completion(*completion*).
8. If *innerResult*.[[Type]] is throw, return Completion(*innerResult*).

The ListIterator **next** method is a standard built-in [function object](#) (clause 17) that performs the following steps:

1. Let O be the **this** value.
2. **Assert:** $\text{Type}(O)$ is Object.
3. **Assert:** O has an $[[\text{IteratedList}]]$ internal slot.
4. Let $list$ be $O.[[\text{IteratedList}]]$.
5. Let $index$ be $O.[[\text{ListIteratorNextIndex}]]$.
6. Let len be the number of elements of $list$.
7. If $index \geq len$, then
 - a. Return [CreateIterResultObject\(undefined, true\)](#).
8. Set $O.[[\text{ListIteratorNextIndex}]]$ to $index + 1$.
9. Return [CreateIterResultObject\(list\[index\], false\)](#).

8 Executable Code and Execution Contexts

8.1 Lexical Environments

A Lexical Environment is a specification type used to define the association of *Identifiers* to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an [Environment Record](#) and a possibly null reference to an *outer* Lexical Environment. Usually a Lexical Environment is associated with some specific syntactic structure of ECMAScript code such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch* clause of a *TryStatement* and a new Lexical Environment is created each time such code is evaluated.

An [Environment Record](#) records the identifier bindings that are created within the scope of its associated Lexical Environment. It is referred to as the Lexical Environment's [EnvironmentRecord](#).

The outer environment reference is used to model the logical nesting of Lexical Environment values. The outer reference of a (inner) Lexical Environment is a reference to the Lexical Environment that logically surrounds the inner Lexical Environment. An outer Lexical Environment may, of course, have its own outer Lexical Environment. A Lexical Environment may serve as the outer environment for multiple inner Lexical Environments. For example, if a *FunctionDeclaration* contains two nested *FunctionDeclarations* then the Lexical Environments of each of the nested functions will have as their outer Lexical Environment the Lexical Environment of the current evaluation of the surrounding function.

A global environment is a Lexical Environment which does not have an outer environment. The [global environment](#)'s outer environment reference is **null**. A [global environment](#)'s [EnvironmentRecord](#) may be prepopulated with identifier bindings and includes an associated [global object](#) whose properties provide some of the [global environment](#)'s identifier bindings. As ECMAScript code is executed, additional properties may be added to the [global object](#) and the initial properties may be modified.

A module environment is a Lexical Environment that contains the bindings for the top level declarations of a *Module*. It also contains the bindings that are explicitly imported by the *Module*. The outer environment of a [module environment](#) is a [global environment](#).

A function environment is a Lexical Environment that corresponds to the invocation of an ECMAScript [function object](#). A [function environment](#) may establish a new **this** binding. A [function environment](#) also captures the state necessary to support **super** method invocations.

DeleteBinding(<i>N</i>)	Delete a binding from an Environment Record . The String value <i>N</i> is the text of the bound name. If a binding for <i>N</i> exists, remove the binding and return true . If the binding exists but cannot be removed return false . If the binding does not exist return true .
HasThisBinding()	Determine if an Environment Record establishes a this binding. Return true if it does and false if it does not.
HasSuperBinding()	Determine if an Environment Record establishes a super method binding. Return true if it does and false if it does not.
WithBaseObject()	If this Environment Record is associated with a with statement, return the with object. Otherwise, return undefined .

8.1.1.1 Declarative Environment Records

Each declarative [Environment Record](#) is associated with an ECMAScript program scope containing variable, constant, let, class, module, import, and/or function declarations. A declarative [Environment Record](#) binds the set of identifiers defined by the declarations contained within its scope.

The behaviour of the concrete specification methods for declarative Environment Records is defined by the following algorithms.

8.1.1.1.1 HasBinding (*N*)

The concrete [Environment Record](#) method HasBinding for declarative Environment Records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let *envRec* be the declarative [Environment Record](#) for which the method was invoked.
2. If *envRec* has a binding for the name that is the value of *N*, return **true**.
3. Return **false**.

8.1.1.1.2 CreateMutableBinding (*N*, *D*)

The concrete [Environment Record](#) method CreateMutableBinding for declarative Environment Records creates a new mutable binding for the name *N* that is uninitialized. A binding must not already exist in this [Environment Record](#) for *N*. If Boolean argument *D* has the value **true** the new binding is marked as being subject to deletion.

1. Let *envRec* be the declarative [Environment Record](#) for which the method was invoked.
2. **Assert:** *envRec* does not already have a binding for *N*.
3. Create a mutable binding in *envRec* for *N* and record that it is uninitialized. If *D* is **true**, record that the newly created binding may be deleted by a subsequent DeleteBinding call.
4. Return **NormalCompletion(empty)**.

8.1.1.1.3 CreateImmutableBinding (*N*, *S*)

The concrete [Environment Record](#) method CreateImmutableBinding for declarative Environment Records creates a new immutable binding for the name *N* that is uninitialized. A binding must not already exist in this [Environment Record](#) for *N*. If the Boolean argument *S* has the value **true** the new binding is marked as a strict binding.

1. Let *envRec* be the declarative [Environment Record](#) for which the method was invoked.

1. Let `envRec` be the declarative **Environment Record** for which the method was invoked.
2. **Assert:** `envRec` has a binding for `N`.
3. If the binding for `N` in `envRec` is an uninitialized binding, throw a **ReferenceError** exception.
4. Return the value currently bound to `N` in `envRec`.

8.1.1.1.7 DeleteBinding (`N`)

The concrete **Environment Record** method `DeleteBinding` for declarative Environment Records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let `envRec` be the declarative **Environment Record** for which the method was invoked.
2. **Assert:** `envRec` has a binding for the name that is the value of `N`.
3. If the binding for `N` in `envRec` cannot be deleted, return **false**.
4. Remove the binding for `N` from `envRec`.
5. Return **true**.

8.1.1.1.8 HasThisBinding ()

Regular declarative Environment Records do not provide a **this** binding.

1. Return **false**.

8.1.1.1.9 HasSuperBinding ()

Regular declarative Environment Records do not provide a **super** binding.

1. Return **false**.

8.1.1.1.10 WithBaseObject ()

Declarative Environment Records always return **undefined** as their `WithBaseObject`.

1. Return **undefined**.

8.1.1.2 Object Environment Records

Each object **Environment Record** is associated with an object called its *binding object*. An object **Environment Record** binds the set of string identifier names that directly correspond to the property names of its binding object. Property keys that are not strings in the form of an *IdentifierName* are not included in the set of bound identifiers. Both own and inherited properties are included in the set regardless of the setting of their `[[Enumerable]]` attribute. Because properties can be dynamically added and deleted from objects, the set of identifiers bound by an object **Environment Record** may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a result of such a side-effect are considered to be a mutable binding even if the `Writable` attribute of the corresponding property has the value **false**. Immutable bindings do not exist for object Environment Records.

Object Environment Records created for `with` statements (13.11) can provide their binding object as an implicit **this** value for use in function calls. The capability is controlled by a `withEnvironment` Boolean value that is associated with each object **Environment Record**. By default, the value of `withEnvironment` is **false** for any object **Environment Record**.

The behaviour of the concrete specification methods for object Environment Records is defined by the following algorithms.

call to InitializeBinding for the same name. Hence, implementations do not need to explicitly track the initialization state of individual object Environment Record bindings.

8.1.1.2.5 SetMutableBinding (*N*, *V*, *S*)

The concrete Environment Record method SetMutableBinding for object Environment Records attempts to set the value of the Environment Record's associated binding object's property whose name is the value of the argument *N* to the value of argument *V*. A property named *N* normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument *S*.

1. Let *envRec* be the object Environment Record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Return ? Set(*bindings*, *N*, *V*, *S*).

8.1.1.2.6 GetBindingValue (*N*, *S*)

The concrete Environment Record method GetBindingValue for object Environment Records returns the value of its associated binding object's property whose name is the String value of the argument identifier *N*. The property should already exist but if it does not the result depends upon the value of the *S* argument:

1. Let *envRec* be the object Environment Record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Let *value* be ? HasProperty(*bindings*, *N*).
4. If *value* is **false**, then
 - a. If *S* is **false**, return the value **undefined**; otherwise throw a **ReferenceError** exception.
5. Return ? Get(*bindings*, *N*).

8.1.1.2.7 DeleteBinding (*N*)

The concrete Environment Record method DeleteBinding for object Environment Records can only delete bindings that correspond to properties of the environment object whose [[Configurable]] attribute have the value **true**.

1. Let *envRec* be the object Environment Record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Return ? *bindings*.[[Delete]](*N*).

8.1.1.2.8 HasThisBinding ()

Regular object Environment Records do not provide a **this** binding.

1. Return **false**.

8.1.1.2.9 HasSuperBinding ()

Regular object Environment Records do not provide a **super** binding.

1. Return **false**.

8.1.1.2.10 WithBaseObject ()

Object Environment Records return **undefined** as their WithBaseObject unless their *withEnvironment* flag is **true**.

The behaviour of the additional concrete specification methods for function Environment Records is defined by the following algorithms:

8.1.1.3.1 BindThisValue (*V*)

1. Let *envRec* be the **function Environment Record** for which the method was invoked.
2. **Assert:** *envRec*.[[ThisBindingStatus]] is not "**lexical**".
3. If *envRec*.[[ThisBindingStatus]] is "**initialized**", throw a **ReferenceError** exception.
4. Set *envRec*.[[ThisValue]] to *V*.
5. Set *envRec*.[[ThisBindingStatus]] to "**initialized**".
6. Return *V*.

8.1.1.3.2 HasThisBinding ()

1. Let *envRec* be the **function Environment Record** for which the method was invoked.
2. If *envRec*.[[ThisBindingStatus]] is "**lexical**", return **false**; otherwise, return **true**.

8.1.1.3.3 HasSuperBinding ()

1. Let *envRec* be the **function Environment Record** for which the method was invoked.
2. If *envRec*.[[ThisBindingStatus]] is "**lexical**", return **false**.
3. If *envRec*.[[HomeObject]] has the value **undefined**, return **false**; otherwise, return **true**.

8.1.1.3.4 GetThisBinding ()

1. Let *envRec* be the **function Environment Record** for which the method was invoked.
2. **Assert:** *envRec*.[[ThisBindingStatus]] is not "**lexical**".
3. If *envRec*.[[ThisBindingStatus]] is "**uninitialized**", throw a **ReferenceError** exception.
4. Return *envRec*.[[ThisValue]].

8.1.1.3.5 GetSuperBase ()

1. Let *envRec* be the **function Environment Record** for which the method was invoked.
2. Let *home* be *envRec*.[[HomeObject]].
3. If *home* has the value **undefined**, return **undefined**.
4. **Assert:** **Type**(*home*) is Object.
5. Return ? *home*.[[GetPrototypeOf]]().

8.1.1.4 Global Environment Records

A global **Environment Record** is used to represent the outer most scope that is shared by all of the ECMAScript *Script* elements that are processed in a common **realm**. A global **Environment Record** provides the bindings for built-in globals (clause 18), properties of the **global object**, and for all top-level declarations (13.2.8, 13.2.10) that occur within a *Script*.

A global **Environment Record** is logically a single record but it is specified as a composite encapsulating an object **Environment Record** and a declarative **Environment Record**. The object **Environment Record** has as its base object the **global object** of the associated **Realm Record**. This **global object** is the value returned by the global **Environment Record**'s **GetThisBinding** concrete method. The object **Environment Record** component of a global **Environment Record** contains the bindings for all built-in globals (clause 18) and all bindings introduced by a *FunctionDeclaration*, *GeneratorDeclaration*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, or *VariableStatement* contained in

CanDeclareGlobalFunction (N)	Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument <i>N</i> .
CreateGlobalVarBinding(N, D)	Used to create and initialize to undefined a global var binding in the [[ObjectRecord]] component of a global Environment Record . The binding will be a mutable binding. The corresponding global object property will have attribute values appropriate for a var . The String value <i>N</i> is the bound name. If <i>D</i> is true the binding may be deleted. Logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows var declarations to receive special treatment.
CreateGlobalFunctionBinding(N, V, D)	Create and initialize a global function binding in the [[ObjectRecord]] component of a global Environment Record . The binding will be a mutable binding. The corresponding global object property will have attribute values appropriate for a function . The String value <i>N</i> is the bound name. <i>V</i> is the initialization value. If the Boolean argument <i>D</i> is true the binding may be deleted. Logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows function declarations to receive special treatment.

The behaviour of the concrete specification methods for global Environment Records is defined by the following algorithms.

8.1.1.4.1 HasBinding (*N*)

The concrete **Environment Record** method HasBinding for global Environment Records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let *envRec* be the global **Environment Record** for which the method was invoked.
2. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
3. If *DclRec*.HasBinding(*N*) is **true**, return **true**.
4. Let *ObjRec* be *envRec*.[[ObjectRecord]].
5. Return ? *ObjRec*.HasBinding(*N*).

8.1.1.4.2 CreateMutableBinding (*N, D*)

The concrete **Environment Record** method CreateMutableBinding for global Environment Records creates a new mutable binding for the name *N* that is uninitialized. The binding is created in the associated DeclarativeRecord. A binding for *N* must not already exist in the DeclarativeRecord. If Boolean argument *D* has the value **true** the new binding is marked as being subject to deletion.

1. Let *envRec* be the global **Environment Record** for which the method was invoked.
2. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
3. If *DclRec*.HasBinding(*N*) is **true**, throw a **TypeError** exception.
4. Return *DclRec*.CreateMutableBinding(*N, D*).

8.1.1.4.3 CreateImmutableBinding (*N, S*)

The concrete **Environment Record** method CreateImmutableBinding for global Environment Records creates a new immutable binding for the name *N* that is uninitialized. A binding must not already exist in this **Environment Record** for

The concrete [Environment Record](#) method `DeleteBinding` for global Environment Records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let `envRec` be the global [Environment Record](#) for which the method was invoked.
2. Let `DclRec` be `envRec`.`[[DeclarativeRecord]]`.
3. If `DclRec`.`HasBinding(N)` is **true**, then
 - a. Return `DclRec`.`DeleteBinding(N)`.
4. Let `ObjRec` be `envRec`.`[[ObjectRecord]]`.
5. Let `globalObject` be the binding object for `ObjRec`.
6. Let `existingProp` be `? HasOwnProperty(globalObject, N)`.
7. If `existingProp` is **true**, then
 - a. Let `status` be `? ObjRec`.`DeleteBinding(N)`.
 - b. If `status` is **true**, then
 - i. Let `varNames` be `envRec`.`[[VarNames]]`.
 - ii. If `N` is an element of `varNames`, remove that element from the `varNames`.
 - c. Return `status`.
8. Return **true**.

8.1.1.4.8 `HasThisBinding()`

1. Return **true**.

8.1.1.4.9 `HasSuperBinding()`

1. Return **false**.

8.1.1.4.10 `WithBaseObject()`

Global Environment Records always return **undefined** as their `WithBaseObject`.

1. Return **undefined**.

8.1.1.4.11 `GetThisBinding()`

1. Let `envRec` be the global [Environment Record](#) for which the method was invoked.
2. Return `envRec`.`[[GlobalThisValue]]`.

8.1.1.4.12 `HasVarDeclaration(N)`

The concrete [Environment Record](#) method `HasVarDeclaration` for global Environment Records determines if the argument identifier has a binding in this record that was created using a `VariableStatement` or a `FunctionDeclaration`:

1. Let `envRec` be the global [Environment Record](#) for which the method was invoked.
2. Let `varDeclaredNames` be `envRec`.`[[VarNames]]`.
3. If `varDeclaredNames` contains `N`, return **true**.
4. Return **false**.

8.1.1.4.13 `HasLexicalDeclaration(N)`

The concrete [Environment Record](#) method `HasLexicalDeclaration` for global Environment Records determines if the argument identifier has a binding in this record that was created using a lexical declaration such as a `LexicalDeclaration`

true }, return **true**.

8. Return **false**.

8.1.1.4.17 CreateGlobalVarBinding (*N*, *D*)

The concrete [Environment Record](#) method CreateGlobalVarBinding for global Environment Records creates and initializes a mutable binding in the associated object [Environment Record](#) and records the bound name in the associated `[[VarNames]]` [List](#). If a binding already exists, it is reused and assumed to be initialized.

1. Let *envRec* be the global [Environment Record](#) for which the method was invoked.
2. Let *ObjRec* be *envRec*.`[[ObjectRecord]]`.
3. Let *globalObject* be the binding object for *ObjRec*.
4. Let *hasProperty* be `? HasOwnProperty`(*globalObject*, *N*).
5. Let *extensible* be `? IsExtensible`(*globalObject*).
6. If *hasProperty* is **false** and *extensible* is **true**, then
 - a. Perform `? ObjRec.CreateMutableBinding`(*N*, *D*).
 - b. Perform `? ObjRec.InitializeBinding`(*N*, **undefined**).
7. Let *varDeclaredNames* be *envRec*.`[[VarNames]]`.
8. If *varDeclaredNames* does not contain *N*, then
 - a. Append *N* to *varDeclaredNames*.
9. Return [NormalCompletion](#)(empty).

8.1.1.4.18 CreateGlobalFunctionBinding (*N*, *V*, *D*)

The concrete [Environment Record](#) method CreateGlobalFunctionBinding for global Environment Records creates and initializes a mutable binding in the associated object [Environment Record](#) and records the bound name in the associated `[[VarNames]]` [List](#). If a binding already exists, it is replaced.

1. Let *envRec* be the global [Environment Record](#) for which the method was invoked.
2. Let *ObjRec* be *envRec*.`[[ObjectRecord]]`.
3. Let *globalObject* be the binding object for *ObjRec*.
4. Let *existingProp* be `? globalObject.[[GetOwnProperty]]`(*N*).
5. If *existingProp* is **undefined** or *existingProp*.`[[Configurable]]` is **true**, then
 - a. Let *desc* be the [PropertyDescriptor](#) { `[[Value]]`: *V*, `[[Writable]]`: **true**, `[[Enumerable]]`: **true**, `[[Configurable]]`: *D* }.
6. Else,
 - a. Let *desc* be the [PropertyDescriptor](#) { `[[Value]]`: *V* }.
7. Perform `? DefinePropertyOrThrow`(*globalObject*, *N*, *desc*).
8. [Record](#) that the binding for *N* in *ObjRec* has been initialized.
9. Perform `? Set`(*globalObject*, *N*, *V*, **false**).
10. Let *varDeclaredNames* be *envRec*.`[[VarNames]]`.
11. If *varDeclaredNames* does not contain *N*, then
 - a. Append *N* to *varDeclaredNames*.
12. Return [NormalCompletion](#)(empty).

NOTE

Global function declarations are always represented as own properties of the [global object](#). If possible, an existing own property is reconfigured to have a standard set of attribute values. Steps 8-9 are equivalent to what calling the [InitializeBinding](#) concrete method would do and if *globalObject* is a Proxy will produce the same sequence of Proxy trap calls.

1. **Assert:** This method is never invoked. See 12.5.3.1.

NOTE

Module Environment Records are only used within strict code and an **early error** rule prevents the delete operator, in strict code, from being applied to a **Reference** that would resolve to a module **Environment Record** binding. See 12.5.3.1.

8.1.1.5.3 HasThisBinding ()

Module Environment Records provide a **this** binding.

1. Return **true**.

8.1.1.5.4 GetThisBinding ()

1. Return **undefined**.

8.1.1.5.5 CreateImportBinding (*N*, *M*, *N2*)

The concrete **Environment Record** method **CreateImportBinding** for module Environment Records creates a new initialized immutable indirect binding for the name *N*. A binding must not already exist in this **Environment Record** for *N*. *M* is a **Module Record**, and *N2* is the name of a binding that exists in *M*'s module **Environment Record**. Accesses to the value of the new binding will indirectly access the bound value of the target binding.

1. Let *envRec* be the module **Environment Record** for which the method was invoked.
2. **Assert:** *envRec* does not already have a binding for *N*.
3. **Assert:** *M* is a **Module Record**.
4. **Assert:** When *M*.[[Environment]] is instantiated it will have a direct binding for *N2*.
5. Create an immutable indirect binding in *envRec* for *N* that references *M* and *N2* as its target binding and record that the binding is initialized.
6. Return **NormalCompletion(empty)**.

8.1.2 Lexical Environment Operations

The following **abstract operations** are used in this specification to operate upon lexical environments:

8.1.2.1 GetIdentifierReference (*lex*, *name*, *strict*)

The abstract operation **GetIdentifierReference** is called with a **Lexical Environment** *lex*, a String *name*, and a Boolean flag *strict*. The value of *lex* may be **null**. When called, the following steps are performed:

1. If *lex* is the value **null**, then
 - a. Return a value of type **Reference** whose base value component is **undefined**, whose referenced name component is *name*, and whose strict reference flag is *strict*.
2. Let *envRec* be *lex*'s **EnvironmentRecord**.
3. Let *exists* be ? *envRec*.HasBinding(*name*).
4. If *exists* is **true**, then
 - a. Return a value of type **Reference** whose base value component is *envRec*, whose referenced name component is *name*, and whose strict reference flag is *strict*.
5. Else,

2. Let *objRec* be a new object **Environment Record** containing *G* as the binding object.
3. Let *dclRec* be a new declarative **Environment Record** containing no bindings.
4. Let *globalRec* be a new global **Environment Record**.
5. Set *globalRec*.[[ObjectRecord]] to *objRec*.
6. Set *globalRec*.[[GlobalThisValue]] to *thisValue*.
7. Set *globalRec*.[[DeclarativeRecord]] to *dclRec*.
8. Set *globalRec*.[[VarNames]] to a new empty **List**.
9. Set *env*'s **EnvironmentRecord** to *globalRec*.
10. Set the outer lexical environment reference of *env* to **null**.
11. Return *env*.

8.1.2.6 NewModuleEnvironment (*E*)

When the abstract operation **NewModuleEnvironment** is called with a **Lexical Environment** argument *E* the following steps are performed:

1. Let *env* be a new **Lexical Environment**.
2. Let *envRec* be a new module **Environment Record** containing no bindings.
3. Set *env*'s **EnvironmentRecord** to *envRec*.
4. Set the outer lexical environment reference of *env* to *E*.
5. Return *env*.

8.2 Realms

Before it is evaluated, all ECMAScript code must be associated with a realm. Conceptually, a **realm** consists of a set of intrinsic objects, an ECMAScript **global environment**, all of the ECMAScript code that is loaded within the scope of that **global environment**, and other associated state and resources.

A **realm** is represented in this specification as a **Realm Record** with the fields specified in [Table 20](#):

Table 20: Realm Record Fields

Field Name	Value	Meaning
[[Intrinsics]]	Record whose field names are intrinsic keys and whose values are objects	The intrinsic values used by code associated with this realm
[[GlobalObject]]	Object	The global object for this realm
[[GlobalEnv]]	Lexical Environment	The global environment for this realm

their properties must be ordered to avoid any dependencies upon objects that have not yet been created.

14. Return *intrinsics*.

8.2.3 SetRealmGlobalObject (*realmRec*, *globalObj*, *thisValue*)

The abstract operation SetRealmGlobalObject with arguments *realmRec*, *globalObj*, and *thisValue* performs the following steps:

1. If *globalObj* is **undefined**, then
 - a. Let *intrinsics* be *realmRec*.[[Intrinsics]].
 - b. Set *globalObj* to ObjectCreate(*intrinsics*.[[%ObjectPrototype%]]).
2. **Assert:** Type(*globalObj*) is Object.
3. If *thisValue* is **undefined**, set *thisValue* to *globalObj*.
4. Set *realmRec*.[[GlobalObject]] to *globalObj*.
5. Let *newGlobalEnv* be NewGlobalEnvironment(*globalObj*, *thisValue*).
6. Set *realmRec*.[[GlobalEnv]] to *newGlobalEnv*.
7. Return *realmRec*.

8.2.4 SetDefaultGlobalBindings (*realmRec*)

The abstract operation SetDefaultGlobalBindings with argument *realmRec* performs the following steps:

1. Let *global* be *realmRec*.[[GlobalObject]].
2. For each property of the Global Object specified in clause 18, do
 - a. Let *name* be the String value of the *property name*.
 - b. Let *desc* be the fully populated *data property* descriptor for the property containing the specified attributes for the property. For properties listed in 18.2, 18.3, or 18.4 the value of the [[Value]] attribute is the corresponding intrinsic object from *realmRec*.
 - c. Perform ? DefinePropertyOrThrow(*global*, *name*, *desc*).
3. Return *global*.

8.3 Execution Contexts

An execution context is a specification device that is used to track the runtime evaluation of code by an ECMAScript implementation. At any point in time, there is at most one execution context per *agent* that is actually executing code. This is known as the *agent's* running execution context. All references to the *running execution context* in this specification denote the *running execution context* of the *surrounding agent*.

The execution context stack is used to track execution contexts. The *running execution context* is always the top element of this stack. A new execution context is created whenever control is transferred from the executable code associated with the currently *running execution context* to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the *running execution context*.

An execution context contains whatever implementation specific state is necessary to track the execution progress of its associated code. Each execution context has at least the state components listed in Table 21.

Table 21: State Components for All Execution Contexts

Component	Purpose
-----------	---------

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation. It is impossible for ECMAScript code to directly access or observe an execution context.

8.3.1 GetActiveScriptOrModule ()

The GetActiveScriptOrModule abstract operation is used to determine the running script or module, based on the [running execution context](#). GetActiveScriptOrModule performs the following steps:

1. If the [execution context stack](#) is empty, return **null**.
2. Let *ec* be the topmost [execution context](#) on the [execution context stack](#) whose ScriptOrModule component is not **null**.
3. If no such [execution context](#) exists, return **null**. Otherwise, return *ec*'s ScriptOrModule component.

8.3.2 ResolveBinding (*name* [, *env*])

The ResolveBinding abstract operation is used to determine the binding of *name* passed as a String value. The optional argument *env* can be used to explicitly provide the [Lexical Environment](#) that is to be searched for the binding. During execution of ECMAScript code, ResolveBinding is performed using the following algorithm:

1. If *env* is not present or if *env* is **undefined**, then
 - a. Set *env* to the [running execution context](#)'s LexicalEnvironment.
2. **Assert:** *env* is a [Lexical Environment](#).
3. If the code matching the syntactic production that is being evaluated is contained in [strict mode code](#), let *strict* be **true**, else let *strict* be **false**.
4. Return ? [GetIdentifierReference](#)(*env*, *name*, *strict*).

NOTE

The result of ResolveBinding is always a [Reference](#) value with its referenced name component equal to the *name* argument.

8.3.3 GetThisEnvironment ()

The abstract operation GetThisEnvironment finds the [Environment Record](#) that currently supplies the binding of the keyword **this**. GetThisEnvironment performs the following steps:

1. Let *lex* be the [running execution context](#)'s LexicalEnvironment.
2. Repeat,
 - a. Let *envRec* be *lex*'s EnvironmentRecord.
 - b. Let *exists* be *envRec*.HasThisBinding().
 - c. If *exists* is **true**, return *envRec*.
 - d. Let *outer* be the value of *lex*'s outer environment reference.
 - e. **Assert:** *outer* is not **null**.
 - f. Set *lex* to *outer*.

NOTE

The loop in step 2 will always terminate because the list of environments always ends with the [global environment](#) which has a **this** binding.

[[ScriptOrModule]]	A Script Record or Module Record	The script or module for the initial execution context when this PendingJob is initiated.
[[HostDefined]]	Any, default value is undefined .	Field reserved for use by host environments that need to associate additional information with a pending Job.

A Job Queue is a FIFO queue of PendingJob records. Each Job Queue has a name and the full set of available Job Queues are defined by an ECMAScript implementation. Every ECMAScript implementation has at least the Job Queues defined in [Table 25](#).

Each [agent](#) has its own set of named Job Queues. All references to a named job queue in this specification denote the named job queue of the [surrounding agent](#).

Table 25: Required Job Queues

Name	Purpose
ScriptJobs	Jobs that validate and evaluate ECMAScript <i>Script</i> and <i>Module</i> source text. See clauses 10 and 15.
PromiseJobs	Jobs that are responses to the settlement of a Promise (see 25.6).

A request for the future execution of a Job is made by enqueueing, on a Job Queue, a PendingJob record that includes a Job abstract operation name and any necessary argument values. When there is no [running execution context](#) and the [execution context stack](#) is empty, the ECMAScript implementation removes the first PendingJob from a Job Queue and uses the information contained in it to create an [execution context](#) and starts execution of the associated Job abstract operation.

The PendingJob records from a single Job Queue are always initiated in FIFO order. This specification does not define the order in which multiple Job Queues are serviced. An ECMAScript implementation may interweave the FIFO evaluation of the PendingJob records of a Job Queue with the evaluation of the PendingJob records of one or more other Job Queues. An implementation must define what occurs when there are no [running execution context](#) and all Job Queues are empty.

NOTE

Typically an ECMAScript implementation will have its Job Queues pre-initialized with at least one PendingJob and one of those Jobs will be the first to be executed. An implementation might choose to free all resources and terminate if the current Job completes and all Job Queues are empty. Alternatively, it might choose to wait for a some implementation specific [agent](#) or mechanism to enqueue new PendingJob requests.

The following [abstract operations](#) are used to create and manage Jobs and Job Queues:

8.4.1 EnqueueJob (*queueName, job, arguments*)

The EnqueueJob abstract operation requires three arguments: *queueName*, *job*, and *arguments*. It performs the following steps:

1. [Assert](#): [Type\(*queueName*\)](#) is String and its value is the name of a Job Queue recognized by this implementation.
2. [Assert](#): *job* is the name of a Job.

- c. Let *nextQueue* be a non-empty Job Queue chosen in an implementation-defined manner. If all Job Queues are empty, the result is implementation-defined.
- d. Let *nextPending* be the PendingJob record at the front of *nextQueue*. Remove that record from *nextQueue*.
- e. Let *newContext* be a new *execution context*.
- f. Set *newContext*'s Function to **null**.
- g. Set *newContext*'s Realm to *nextPending*.[[Realm]].
- h. Set *newContext*'s ScriptOrModule to *nextPending*.[[ScriptOrModule]].
- i. Push *newContext* onto the *execution context stack*; *newContext* is now the *running execution context*.
- j. Perform any implementation or host environment defined job initialization using *nextPending*.
- k. Let *result* be the result of performing the abstract operation named by *nextPending*.[[Job]] using the elements of *nextPending*.[[Arguments]] as its arguments.
- l. If *result* is an *abrupt completion*, perform *HostReportErrors*(« *result*.[[Value]] »).

8.7 Agents

An agent comprises a set of ECMAScript execution contexts, an *execution context stack*, a *running execution context*, a set of named job queues, an Agent Record, and an executing thread. Except for the *executing thread*, the constituents of an *agent* belong exclusively to that *agent*.

An *agent*'s *executing thread* executes the jobs in the *agent*'s job queues on the *agent*'s execution contexts independently of other agents, except that an *executing thread* may be used as the *executing thread* by multiple agents, provided none of the agents sharing the thread have an *Agent Record* whose [[CanBlock]] property is **true**.

NOTE 1

Some web browsers share a single *executing thread* across multiple unrelated tabs of a browser window, for example.

While an *agent*'s *executing thread* executes the jobs in the *agent*'s job queues, the *agent* is the surrounding agent for the code in those jobs. The code uses the *surrounding agent* to access the specification level execution objects held within the *agent*: the *running execution context*, the *execution context stack*, the named job queues, and the *Agent Record*'s fields.

Table 26: Agent Record Fields

Field Name	Value	Meaning
[[LittleEndian]]	Boolean	The default value computed for the <i>isLittleEndian</i> parameter when it is needed by the algorithms <i>GetValueFromBuffer</i> and <i>SetValueInBuffer</i> . The choice is implementation-dependent and should be the alternative that is most efficient for the implementation. Once the value has been observed it cannot change.
[[CanBlock]]	Boolean	Determines whether the <i>agent</i> can block or not.
[[Signifier]]	Any globally- unique value	Uniquely identifies the <i>agent</i> within its <i>agent cluster</i> .
[[IsLockFree1]]	Boolean	true if atomic operations on one-byte values are lock-free, false otherwise.
[[IsLockFree2]]	Boolean	true if atomic operations on two-byte values are lock-free, false otherwise.

An agent cluster is a maximal set of agents that can communicate by operating on shared memory.

NOTE 1

Programs within different agents may share memory by unspecified means. At a minimum, the backing memory for SharedArrayBuffer objects can be shared among the agents in the cluster.

There may be agents that can communicate by message passing that cannot share memory; they are never in the same agent cluster.

Every [agent](#) belongs to exactly one agent cluster.

NOTE 2

The agents in a cluster need not all be alive at some particular point in time. If [agent A](#) creates another [agent B](#), after which [A](#) terminates and [B](#) creates [agent C](#), the three agents are in the same cluster if [A](#) could share some memory with [B](#) and [B](#) could share some memory with [C](#).

All agents within a cluster must have the same value for the `[[LittleEndian]]` property in their respective [Agent](#) Records.

NOTE 3

If different agents within an agent cluster have different values of `[[LittleEndian]]` it becomes hard to use shared memory for multi-byte data.

All agents within a cluster must have the same values for the `[[IsLockFree1]]` property in their respective [Agent](#) Records; similarly for the `[[IsLockFree2]]` property.

All agents within a cluster must have different values for the `[[Signifier]]` property in their respective [Agent](#) Records.

An embedding may deactivate (stop forward progress) or activate (resume forward progress) an [agent](#) without the [agent's](#) knowledge or cooperation. If the embedding does so, it must not leave some agents in the cluster active while other agents in the cluster are deactivated indefinitely.

NOTE 4

The purpose of the preceding restriction is to avoid a situation where an [agent](#) deadlocks or starves because another [agent](#) has been deactivated. For example, if an HTML shared worker that has a lifetime independent of documents in any windows were allowed to share memory with the dedicated worker of such an independent document, and the document and its dedicated worker were to be deactivated while the dedicated worker holds a lock (say, the document is pushed into its window's history), and the shared worker then tries to acquire the lock, then the shared worker will be blocked until the dedicated worker is activated again, if ever. Meanwhile other workers trying to access the shared worker from other windows will starve.

The implication of the restriction is that it will not be possible to share memory between agents that don't belong to the same suspend/wake collective within the embedding.

An embedding may terminate an [agent](#) without any of the [agent's](#) cluster's other agents' prior knowledge or cooperation. If an [agent](#) is terminated not by programmatic action of its own or of another [agent](#) in the cluster but by forces external to the cluster, then the embedding must choose one of two strategies: Either terminate all the agents in the cluster, or provide reliable APIs that allow the agents in the cluster to coordinate so that at least one remaining member of the cluster will be able to detect the termination, with the termination data containing enough information to identify the

properties of the child object) for the purposes of get access, but not for set access. Accessor properties are inherited for both get access and set access.

Every ordinary object has a Boolean-valued [[Extensible]] internal slot which is used to fulfill the extensibility-related internal method invariants specified in 6.1.7.3. Namely, once the value of an object's [[Extensible]] internal slot has been set to **false**, it is no longer possible to add properties to the object, to modify the value of the object's [[Prototype]] internal slot, or to subsequently change the value of [[Extensible]] to **true**.

In the following algorithm descriptions, assume O is an ordinary object, P is a property key value, V is any ECMAScript language value, and $Desc$ is a Property Descriptor record.

Each ordinary object internal method delegates to a similarly-named abstract operation. If such an abstract operation depends on another internal method, then the internal method is invoked on O rather than calling the similarly-named abstract operation directly. These semantics ensure that exotic objects have their overridden internal methods invoked when ordinary object internal methods are applied to them.

9.1.1 [[GetPrototypeOf]] ()

When the [[GetPrototypeOf]] internal method of O is called, the following steps are taken:

1. Return ! OrdinaryGetPrototypeOf(O).

9.1.1.1 OrdinaryGetPrototypeOf (O)

When the abstract operation OrdinaryGetPrototypeOf is called with Object O , the following steps are taken:

1. Return $O.[[Prototype]]$.

9.1.2 [[SetPrototypeOf]] (V)

When the [[SetPrototypeOf]] internal method of O is called with argument V , the following steps are taken:

1. Return ! OrdinarySetPrototypeOf(O , V).

9.1.2.1 OrdinarySetPrototypeOf (O , V)

When the abstract operation OrdinarySetPrototypeOf is called with Object O and value V , the following steps are taken:

1. **Assert:** Either Type(V) is Object or Type(V) is Null.
2. Let $extensible$ be $O.[[Extensible]]$.
3. Let $current$ be $O.[[Prototype]]$.
4. If SameValue(V , $current$) is **true**, return **true**.
5. If $extensible$ is **false**, return **false**.
6. Let p be V .
7. Let $done$ be **false**.
8. Repeat, while $done$ is **false**,
 - a. If p is **null**, set $done$ to **true**.
 - b. Else if SameValue(p , O) is **true**, return **false**.
 - c. Else,
 - i. If $p.[[GetPrototypeOf]]$ is not the ordinary object internal method defined in 9.1.1, set $done$ to **true**.
 - ii. Else, set p to $p.[[Prototype]]$.

6. Else X is an accessor property,
 - a. Set $D.[[Get]]$ to the value of X 's $[[Get]]$ attribute.
 - b. Set $D.[[Set]]$ to the value of X 's $[[Set]]$ attribute.
7. Set $D.[[Enumerable]]$ to the value of X 's $[[Enumerable]]$ attribute.
8. Set $D.[[Configurable]]$ to the value of X 's $[[Configurable]]$ attribute.
9. Return D .

9.1.6 $[[DefineOwnProperty]](P, Desc)$

When the $[[DefineOwnProperty]]$ internal method of O is called with property key P and Property Descriptor $Desc$, the following steps are taken:

1. Return $\text{? OrdinaryDefineOwnProperty}(O, P, Desc)$.

9.1.6.1 OrdinaryDefineOwnProperty ($O, P, Desc$)

When the abstract operation OrdinaryDefineOwnProperty is called with Object O , property key P , and Property Descriptor $Desc$, the following steps are taken:

1. Let $current$ be $\text{? } O.[[GetOwnProperty]](P)$.
2. Let $extensible$ be $\text{? IsExtensible}(O)$.
3. Return $\text{ValidateAndApplyPropertyDescriptor}(O, P, extensible, Desc, current)$.

9.1.6.2 IsCompatiblePropertyDescriptor ($Extensible, Desc, Current$)

When the abstract operation IsCompatiblePropertyDescriptor is called with Boolean value $Extensible$, and Property Descriptors $Desc$, and $Current$, the following steps are taken:

1. Return $\text{ValidateAndApplyPropertyDescriptor}(\text{undefined}, \text{undefined}, Extensible, Desc, Current)$.

9.1.6.3 ValidateAndApplyPropertyDescriptor ($O, P, extensible, Desc, current$)

When the abstract operation ValidateAndApplyPropertyDescriptor is called with Object O , property key P , Boolean value $extensible$, and Property Descriptors $Desc$, and $current$, the following steps are taken:

NOTE

If undefined is passed as O , only validation is performed and no object updates are performed.

1. **Assert:** If O is not undefined , then $\text{IsPropertyKey}(P)$ is **true**.
2. If $current$ is undefined , then
 - a. If $extensible$ is **false**, return **false**.
 - b. **Assert:** $extensible$ is **true**.
 - c. If $\text{IsGenericDescriptor}(Desc)$ is **true** or $\text{IsDataDescriptor}(Desc)$ is **true**, then
 - i. If O is not undefined , create an own **data property** named P of object O whose $[[Value]], [[Writable]], [[Enumerable]]$ and $[[Configurable]]$ attribute values are described by $Desc$. If the value of an attribute field of $Desc$ is absent, the attribute of the newly created property is set to its default value.
 - d. Else $Desc$ must be an accessor **Property Descriptor**,
 - i. If O is not undefined , create an own **accessor property** named P of object O whose $[[Get]], [[Set]], [[Enumerable]]$ and $[[Configurable]]$ attribute values are described by $Desc$. If the value of an attribute

- a. Return ? `parent`.`[[HasProperty]](P)`.
6. Return **false**.

9.1.8 [[Get]] (*P, Receiver*)

When the [[Get]] internal method of *O* is called with property key *P* and ECMAScript language value *Receiver*, the following steps are taken:

1. Return ? `OrdinaryGet(O, P, Receiver)`.

9.1.8.1 OrdinaryGet (*O, P, Receiver*)

When the abstract operation OrdinaryGet is called with Object *O*, property key *P*, and ECMAScript language value *Receiver*, the following steps are taken:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *desc* be ? *O*.`[[GetOwnProperty]](P)`.
3. If *desc* is **undefined**, then
 - a. Let *parent* be ? *O*.`[[GetPrototypeOf]]()`.
 - b. If *parent* is **null**, return **undefined**.
 - c. Return ? `parent`.`[[Get]](P, Receiver)`.
4. If `IsDataDescriptor(desc)` is **true**, return *desc*.`[[Value]]`.
5. **Assert:** `IsAccessorDescriptor(desc)` is **true**.
6. Let *getter* be *desc*.`[[Get]]`.
7. If *getter* is **undefined**, return **undefined**.
8. Return ? `Call(getter, Receiver)`.

9.1.9 [[Set]] (*P, V, Receiver*)

When the [[Set]] internal method of *O* is called with property key *P*, value *V*, and ECMAScript language value *Receiver*, the following steps are taken:

1. Return ? `OrdinarySet(O, P, V, Receiver)`.

9.1.9.1 OrdinarySet (*O, P, V, Receiver*)

When the abstract operation OrdinarySet is called with Object *O*, property key *P*, value *V*, and ECMAScript language value *Receiver*, the following steps are taken:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *ownDesc* be ? *O*.`[[GetOwnProperty]](P)`.
3. Return `OrdinarySetWithOwnDescriptor(O, P, V, Receiver, ownDesc)`.

9.1.9.2 OrdinarySetWithOwnDescriptor (*O, P, V, Receiver, ownDesc*)

When the abstract operation OrdinarySetWithOwnDescriptor is called with Object *O*, property key *P*, value *V*, ECMAScript language value *Receiver*, and **Property Descriptor** (or **undefined**) *ownDesc*, the following steps are taken:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. If *ownDesc* is **undefined**, then

When the abstract operation OrdinaryOwnPropertyKeys is called with Object O , the following steps are taken:

1. Let $keys$ be a new empty List.
2. For each own property key P of O that is an array index, in ascending numeric index order, do
 - a. Add P as the last element of $keys$.
3. For each own property key P of O that is a String but is not an array index, in ascending chronological order of property creation, do
 - a. Add P as the last element of $keys$.
4. For each own property key P of O that is a Symbol, in ascending chronological order of property creation, do
 - a. Add P as the last element of $keys$.
5. Return $keys$.

9.1.12 ObjectCreate ($proto$ [, $internalSlotsList$])

The abstract operation ObjectCreate with argument $proto$ (an object or null) is used to specify the runtime creation of new ordinary objects. The optional argument $internalSlotsList$ is a List of the names of additional internal slots that must be defined as part of the object. If the list is not provided, a new empty List is used. This abstract operation performs the following steps:

1. If $internalSlotsList$ is not present, set $internalSlotsList$ to a new empty List.
2. Let obj be a newly created object with an internal slot for each name in $internalSlotsList$.
3. Set obj 's essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set $obj.[[Prototype]]$ to $proto$.
5. Set $obj.[[Extensible]]$ to true.
6. Return obj .

9.1.13 OrdinaryCreateFromConstructor ($constructor$, $intrinsicDefaultProto$ [, $internalSlotsList$])

The abstract operation OrdinaryCreateFromConstructor creates an ordinary object whose [[Prototype]] value is retrieved from a $constructor$'s **prototype** property, if it exists. Otherwise the intrinsic named by $intrinsicDefaultProto$ is used for [[Prototype]]. The optional $internalSlotsList$ is a List of the names of additional internal slots that must be defined as part of the object. If the list is not provided, a new empty List is used. This abstract operation performs the following steps:

1. **Assert:** $intrinsicDefaultProto$ is a String value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the [[Prototype]] value of an object.
2. Let $proto$ be ? GetPrototypeOfConstructor($constructor$, $intrinsicDefaultProto$).
3. Return ObjectCreate($proto$, $internalSlotsList$).

9.1.14 GetPrototypeOfConstructor ($constructor$, $intrinsicDefaultProto$)

The abstract operation GetPrototypeOfConstructor determines the [[Prototype]] value that should be used to create an object corresponding to a specific $constructor$. The value is retrieved from the $constructor$'s **prototype** property, if it exists. Otherwise the intrinsic named by $intrinsicDefaultProto$ is used for [[Prototype]]. This abstract operation performs the following steps:

1. **Assert:** $intrinsicDefaultProto$ is a String value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the [[Prototype]] value of an object.

[[Strict]]	Boolean	true if this is a strict function , false if this is a non-strict function .
[[HomeObject]]	Object	If the function uses super , this is the object whose [[GetPrototypeOf]] provides the object where super property lookups begin.
[[SourceText]]	String	The source text that defines the function.

All ECMAScript function objects have the [[Call]] internal method defined here. ECMAScript functions that are also constructors in addition have the [[Construct]] internal method.

9.2.1 [[Call]] (*thisArgument*, *argumentsList*)

The [[Call]] internal method for an ECMAScript function object *F* is called with parameters *thisArgument* and *argumentsList*, a List of ECMAScript language values. The following steps are taken:

1. **Assert:** *F* is an ECMAScript function object.
2. If *F*.[[FunctionKind]] is "classConstructor", throw a **TypeError** exception.
3. Let *callerContext* be the running execution context.
4. Let *calleeContext* be [PrepareForOrdinaryCall](#)(*F*, **undefined**).
5. **Assert:** *calleeContext* is now the running execution context.
6. Perform [OrdinaryCallBindThis](#)(*F*, *calleeContext*, *thisArgument*).
7. Let *result* be [OrdinaryCallEvaluateBody](#)(*F*, *argumentsList*).
8. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
9. If *result*.[[Type]] is **return**, return [NormalCompletion](#)(*result*.[[Value]]).
10. [ReturnIfAbrupt](#)(*result*).
11. Return [NormalCompletion](#)(**undefined**).

NOTE

When *calleeContext* is removed from the execution context stack in step 8 it must not be destroyed if it is suspended and retained for later resumption by an accessible generator object.

9.2.1.1 [PrepareForOrdinaryCall](#) (*F*, *newTarget*)

When the abstract operation [PrepareForOrdinaryCall](#) is called with function object *F* and ECMAScript language value *newTarget*, the following steps are taken:

1. **Assert:** [Type](#)(*newTarget*) is Undefined or Object.
2. Let *callerContext* be the running execution context.
3. Let *calleeContext* be a new ECMAScript code execution context.
4. Set the Function of *calleeContext* to *F*.
5. Let *calleeRealm* be *F*.[[Realm]].
6. Set the Realm of *calleeContext* to *calleeRealm*.
7. Set the ScriptOrModule of *calleeContext* to *F*.[[ScriptOrModule]].
8. Let *localEnv* be [NewFunctionEnvironment](#)(*F*, *newTarget*).
9. Set the LexicalEnvironment of *calleeContext* to *localEnv*.
10. Set the VariableEnvironment of *calleeContext* to *localEnv*.
11. If *callerContext* is not already suspended, suspend *callerContext*.

7. **Assert:** *calleeContext* is now the running execution context.
8. If *kind* is "base", perform *OrdinaryCallBindThis*(*F*, *calleeContext*, *thisArgument*).
9. Let *constructorEnv* be the LexicalEnvironment of *calleeContext*.
10. Let *envRec* be *constructorEnv*'s EnvironmentRecord.
11. Let *result* be *OrdinaryCallEvaluateBody*(*F*, *argumentsList*).
12. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
13. If *result*.[[Type]] is return, then
 - a. If *Type(result.[[Value]])* is Object, return *NormalCompletion(result.[[Value]])*.
 - b. If *kind* is "base", return *NormalCompletion(thisArgument)*.
 - c. If *result.[[Value]]* is not undefined, throw a **TypeError** exception.
14. Else, *ReturnIfAbrupt(result)*.
15. Return ? *envRec*.GetThisBinding().

9.2.3 FunctionAllocate (*functionPrototype*, *strict*, *functionKind*)

The abstract operation FunctionAllocate requires the three arguments *functionPrototype*, *strict* and *functionKind*. FunctionAllocate performs the following steps:

1. **Assert:** *Type(functionPrototype)* is Object.
2. **Assert:** *functionKind* is either "normal", "non-constructor", "generator", "async", or "async generator".
3. If *functionKind* is "normal", let *needsConstruct* be true.
4. Else, let *needsConstruct* be false.
5. If *functionKind* is "non-constructor", set *functionKind* to "normal".
6. Let *F* be a newly created ECMAScript function object with the internal slots listed in [Table 27](#). All of those internal slots are initialized to undefined.
7. Set *F*'s essential internal methods to the default ordinary object definitions specified in [9.1](#).
8. Set *F*.[[Call]] to the definition specified in [9.2.1](#).
9. If *needsConstruct* is true, then
 - a. Set *F*.[[Construct]] to the definition specified in [9.2.2](#).
 - b. Set *F*.[[ConstructorKind]] to "base".
10. Set *F*.[[Strict]] to *strict*.
11. Set *F*.[[FunctionKind]] to *functionKind*.
12. Set *F*.[[Prototype]] to *functionPrototype*.
13. Set *F*.[[Extensible]] to true.
14. Set *F*.[[Realm]] to the current Realm Record.
15. Return *F*.

9.2.4 FunctionInitialize (*F*, *kind*, *ParameterList*, *Body*, *Scope*)

The abstract operation FunctionInitialize requires the arguments: a function object *F*, *kind* which is one of (Normal, Method, Arrow), a parameter list Parse Node specified by *ParameterList*, a body Parse Node specified by *Body*, a Lexical Environment specified by *Scope*. FunctionInitialize performs the following steps:

1. Let *len* be the ExpectedArgumentCount of *ParameterList*.
2. Perform ! *SetFunctionLength*(*F*, *len*).
3. Let *Strict* be *F*.[[Strict]].
4. Set *F*.[[Environment]] to *Scope*.

2. Let F be ! FunctionAllocate(*functionPrototype*, *Strict*, "async").
3. Return ! FunctionInitialize(F , *kind*, *parameters*, *body*, *Scope*).

9.2.9 AddRestrictedFunctionProperties (F , *realm*)

The abstract operation AddRestrictedFunctionProperties is called with a function object F and Realm Record *realm* as its argument. It performs the following steps:

1. **Assert:** *realm*.[[Intrinsics]].[[%ThrowTypeError%]] exists and has been initialized.
2. Let *thrower* be *realm*.[[Intrinsics]].[[%ThrowTypeError%]].
3. Perform ! DefinePropertyOrThrow(F , "caller", PropertyDescriptor { [[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: false, [[Configurable]]: true }).
4. Return ! DefinePropertyOrThrow(F , "arguments", PropertyDescriptor { [[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: false, [[Configurable]]: true }).

9.2.9.1 %ThrowTypeError% ()

The %ThrowTypeError% intrinsic is an anonymous built-in function object that is defined once for each *realm*. When %ThrowTypeError% is called it performs the following steps:

1. Throw a **TypeError** exception.

The value of the [[Extensible]] internal slot of a %ThrowTypeError% function is **false**.

The "length" property of a %ThrowTypeError% function has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

9.2.10 MakeConstructor (F [, *writablePrototype* [, *prototype*]])

The abstract operation MakeConstructor requires a Function argument F and optionally, a Boolean *writablePrototype* and an object *prototype*. If *prototype* is provided it is assumed to already contain, if needed, a "**constructor**" property whose value is F . This operation converts F into a **constructor** by performing the following steps:

1. **Assert:** F is an ECMAScript function object.
2. **Assert:** IsConstructor(F) is **true**.
3. **Assert:** F is an extensible object that does not have a **prototype** own property.
4. If *writablePrototype* is not present, set *writablePrototype* to **true**.
5. If *prototype* is not present, then
 - a. Set *prototype* to ObjectCreate(%ObjectPrototype%).
 - b. Perform ! DefinePropertyOrThrow(*prototype*, "constructor", PropertyDescriptor { [[Value]]: F , [[Writable]]: *writablePrototype*, [[Enumerable]]: false, [[Configurable]]: true }).
6. Perform ! DefinePropertyOrThrow(F , "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: *writablePrototype*, [[Enumerable]]: false, [[Configurable]]: false }).
7. Return NormalCompletion(**undefined**).

9.2.11 MakeClassConstructor (F)

The abstract operation MakeClassConstructor with argument F performs the following steps:

1. **Assert:** F is an ECMAScript function object.

initializers exist, a second **Environment Record** is created for the body declarations. Formal parameters and functions are initialized as part of **FunctionDeclarationInstantiation**. All other bindings are initialized during evaluation of the function body.

FunctionDeclarationInstantiation is performed as follows using arguments *func* and *argumentsList*. *func* is the **function object** for which the **execution context** is being established.

1. Let *calleeContext* be the **running execution context**.
2. Let *env* be the **LexicalEnvironment** of *calleeContext*.
3. Let *envRec* be *env*'s **EnvironmentRecord**.
4. Let *code* be *func*.**[[ECMAScriptCode]]**.
5. Let *strict* be *func*.**[[Strict]]**.
6. Let *formals* be *func*.**[[FormalParameters]]**.
7. Let *parameterNames* be the **BoundNames** of *formals*.
8. If *parameterNames* has any duplicate entries, let *hasDuplicates* be **true**. Otherwise, let *hasDuplicates* be **false**.
9. Let *simpleParameterList* be **IsSimpleParameterList** of *formals*.
10. Let *hasParameterExpressions* be **ContainsExpression** of *formals*.
11. Let *varNames* be the **VarDeclaredNames** of *code*.
12. Let *varDeclarations* be the **VarScopedDeclarations** of *code*.
13. Let *lexicalNames* be the **LexicallyDeclaredNames** of *code*.
14. Let *functionNames* be a new empty **List**.
15. Let *functionsToInitialize* be a new empty **List**.
16. For each *d* in *varDeclarations*, in reverse list order, do
 - a. If *d* is neither a **VariableDeclaration** nor a **ForBinding** nor a **BindingIdentifier**, then
 - i. **Assert:** *d* is either a **FunctionDeclaration**, a **GeneratorDeclaration**, an **AsyncFunctionDeclaration**, or an **AsyncGeneratorDeclaration**.
 - ii. Let *fn* be the sole element of the **BoundNames** of *d*.
 - iii. If *fn* is not an element of *functionNames*, then
 1. Insert *fn* as the first element of *functionNames*.
 2. NOTE: If there are multiple function declarations for the same name, the last declaration is used.
 3. Insert *d* as the first element of *functionsToInitialize*.
17. Let *argumentsObjectNeeded* be **true**.
18. If *func*.**[[ThisMode]]** is **lexical**, then
 - a. NOTE: Arrow functions never have an **arguments** objects.
 - b. Set *argumentsObjectNeeded* to **false**.
19. Else if "**arguments**" is an element of *parameterNames*, then
 - a. Set *argumentsObjectNeeded* to **false**.
20. Else if *hasParameterExpressions* is **false**, then
 - a. If "**arguments**" is an element of *functionNames* or if "**arguments**" is an element of *lexicalNames*, then
 - i. Set *argumentsObjectNeeded* to **false**.
21. For each String *paramName* in *parameterNames*, do
 - a. Let *alreadyDeclared* be *envRec*.**HasBinding**(*paramName*).
 - b. NOTE: Early errors ensure that duplicate parameter names can only occur in non-strict functions that do not have parameter default values or rest parameters.
 - c. If *alreadyDeclared* is **false**, then
 - i. Perform ! *envRec*.**CreateMutableBinding**(*paramName*, **false**).
 - ii. If *hasDuplicates* is **true**, then
 1. Perform ! *envRec*.**InitializeBinding**(*paramName*, **undefined**).

29. NOTE: Annex [B.3.3.1](#) adds additional steps at this point.
30. If `strict` is **false**, then
 - a. Let `lexEnv` be `NewDeclarativeEnvironment(varEnv)`.
 - b. NOTE: Non-strict functions use a separate lexical `Environment Record` for top-level lexical declarations so that a `direct eval` can determine whether any var scoped declarations introduced by the eval code conflict with pre-existing top-level lexically scoped declarations. This is not needed for strict functions because a strict `direct eval` always places all declarations into a new `Environment Record`.
31. Else, let `lexEnv` be `varEnv`.
32. Let `lexEnvRec` be `lexEnv`'s `EnvironmentRecord`.
33. Set the `LexicalEnvironment` of `calleeContext` to `lexEnv`.
34. Let `lexDeclarations` be the `LexicallyScopedDeclarations` of `code`.
35. For each element `d` in `lexDeclarations`, do
 - a. NOTE: A lexically declared name cannot be the same as a function/generator declaration, formal parameter, or a var name. Lexically declared names are only instantiated here but not initialized.
 - b. For each element `dn` of the `BoundNames` of `d`, do
 - i. If `IsConstantDeclaration` of `d` is **true**, then
 1. Perform ! `lexEnvRec.CreateImmutableBinding(dn, true)`.
 - ii. Else,
 1. Perform ! `lexEnvRec.CreateMutableBinding(dn, false)`.
36. For each `Parse Node f` in `functionsToInitialize`, do
 - a. Let `fn` be the sole element of the `BoundNames` of `f`.
 - b. Let `fo` be the result of performing `InstantiateFunctionObject` for `f` with argument `lexEnv`.
 - c. Perform ! `varEnvRec.SetMutableBinding(fn, fo, false)`.
37. Return `NormalCompletion(empty)`.

NOTE 2

[B.3.3](#) provides an extension to the above algorithm that is necessary for backwards compatibility with web browser implementations of ECMAScript that predate ECMAScript 2015.

NOTE 3

Parameter `Initializers` may contain `direct eval` expressions. Any top level declarations of such evals are only visible to the eval code ([10.2](#)). The creation of the environment for such declarations is described in [14.1.19](#).

9.3 Built-in Function Objects

The built-in function objects defined in this specification may be implemented as either ECMAScript function objects ([9.2](#)) whose behaviour is provided using ECMAScript code or as implementation provided function exotic objects whose behaviour is provided in some other manner. In either case, the effect of calling such functions must conform to their specifications. An implementation may also provide additional built-in function objects that are not defined in this specification.

If a built-in function object is implemented as an exotic object it must have the ordinary object behaviour specified in [9.1](#). All such function exotic objects also have `[[Prototype]]`, `[[Extensible]]`, `[[Realm]]`, and `[[ScriptOrModule]]` internal slots.

Unless otherwise specified every built-in function object has the `%FunctionPrototype%` object as the initial value of its `[[Prototype]]` internal slot.

The behaviour specified for each built-in function via algorithm steps or other means is the specification of the function

The [[Construct]] internal method for built-in **function object** F is called with parameters $argumentsList$ and $newTarget$. The steps performed are the same as [[Call]] (see 9.3.1) except that step 10 is replaced by:

10. Let $result$ be the **Completion Record** that is the result of evaluating F in an implementation-defined manner that conforms to the specification of F . The **this** value is uninitialized, $argumentsList$ provides the named parameters, and $newTarget$ provides the NewTarget value.

9.3.3 CreateBuiltinFunction ($steps$, $internalSlotsList$ [, $realm$ [, $prototype$]])

The abstract operation CreateBuiltinFunction takes arguments $steps$, $internalSlotsList$, $realm$, and $prototype$. The argument $internalSlotsList$ is a **List** of the names of additional internal slots that must be defined as part of the object. CreateBuiltinFunction returns a built-in **function object** created by the following steps:

1. **Assert:** $steps$ is either a set of algorithm steps or other definition of a function's behaviour provided in this specification.
2. If $realm$ is not present, set $realm$ to the current Realm Record.
3. **Assert:** $realm$ is a **Realm Record**.
4. If $prototype$ is not present, set $prototype$ to $realm.[[Intrinsics]].[[\%FunctionPrototype%]]$.
5. Let $func$ be a new built-in **function object** that when called performs the action described by $steps$. The new **function object** has internal slots whose names are the elements of $internalSlotsList$. The initial value of each of those internal slots is **undefined**.
6. Set $func.[[Realm]]$ to $realm$.
7. Set $func.[[Prototype]]$ to $prototype$.
8. Set $func.[[Extensible]]$ to **true**.
9. Set $func.[[ScriptOrModule]]$ to **null**.
10. Return $func$.

Each built-in function defined in this specification is created by calling the CreateBuiltinFunction abstract operation.

9.4 Built-in Exotic Object Internal Methods and Slots

This specification defines several kinds of built-in exotic objects. These objects generally behave similar to ordinary objects except for a few specific situations. The following exotic objects use the ordinary object internal methods except where it is explicitly specified otherwise below:

9.4.1 Bound Function Exotic Objects

A bound function is an **exotic object** that wraps another **function object**. A bound function is callable (it has a [[Call]] internal method and may have a [[Construct]] internal method). Calling a bound function generally results in a call of its wrapped function.

Bound function objects do not have the internal slots of ECMAScript function objects defined in Table 27. Instead they have the internal slots defined in Table 28.

Table 28: Internal Slots of Bound Function Exotic Objects

Internal Slot	Type	Description
---------------	------	-------------

- a. Set $\text{obj}[[\text{Construct}]]$ as described in 9.4.1.2.
7. Set $\text{obj}[[\text{Prototype}]]$ to proto .
8. Set $\text{obj}[[\text{Extensible}]]$ to **true**.
9. Set $\text{obj}[[\text{BoundTargetFunction}]]$ to targetFunction .
10. Set $\text{obj}[[\text{BoundThis}]]$ to boundThis .
11. Set $\text{obj}[[\text{BoundArguments}]]$ to boundArgs .
12. Return obj .

9.4.2 Array Exotic Objects

An *Array object* is an *exotic object* that gives special treatment to *array index* property keys (see 6.1.7). A property whose *property name* is an *array index* is also called an *element*. Every *Array object* has a non-configurable "**length**" property whose value is always a nonnegative integer less than 2^{32} . The value of the "**length**" property is numerically greater than the name of every own property whose name is an *array index*; whenever an own property of an *Array object* is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever an own property is added whose name is an *array index*, the value of the "**length**" property is changed, if necessary, to be one more than the numeric value of that *array index*; and whenever the value of the "**length**" property is changed, every own property whose name is an *array index* whose value is not smaller than the new length is deleted. This constraint applies only to own properties of an *Array object* and is unaffected by "**length**" or *array index* properties that may be inherited from its prototypes.

NOTE

A String *property name* P is an *array index* if and only if $\text{ToString}(\text{ToUint32}(P))$ is equal to P and $\text{ToUint32}(P)$ is not equal to $2^{32} - 1$.

Array exotic objects provide an alternative definition for the $[[\text{DefineOwnProperty}]]$ internal method. Except for that internal method, *Array exotic objects* provide all of the other essential internal methods as specified in 9.1.

9.4.2.1 $[[\text{DefineOwnProperty}]](P, \text{Desc})$

When the $[[\text{DefineOwnProperty}]]$ internal method of an *Array exotic object* A is called with property key P , and *Property Descriptor* Desc , the following steps are taken:

1. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
2. If P is "**length**", then
 - a. Return $? \text{ArraySetLength}(A, \text{Desc})$.
3. Else if P is an *array index*, then
 - a. Let oldLenDesc be $\text{OrdinaryGetOwnProperty}(A, "length")$.
 - b. **Assert:** oldLenDesc will never be **undefined** or an accessor descriptor because *Array objects* are created with a length *data property* that cannot be deleted or reconfigured.
 - c. Let oldLen be $\text{oldLenDesc}[[\text{Value}]]$.
 - d. Let index be $! \text{ToUint32}(P)$.
 - e. If $\text{index} \geq \text{oldLen}$ and $\text{oldLenDesc}[[\text{Writable}]]$ is **false**, return **false**.
 - f. Let succeeded be $! \text{OrdinaryDefineOwnProperty}(A, P, \text{Desc})$.
 - g. If succeeded is **false**, return **false**.
 - h. If $\text{index} \geq \text{oldLen}$, then
 - i. Set $\text{oldLenDesc}[[\text{Value}]]$ to $\text{index} + 1$.
 - ii. Let succeeded be $\text{OrdinaryDefineOwnProperty}(A, "length", \text{oldLenDesc})$.

defined using `ArraySpeciesCreate`.

9.4.2.4 `ArraySetLength (A, Desc)`

When the abstract operation `ArraySetLength` is called with an `Array` exotic object `A`, and `Property Descriptor Desc`, the following steps are taken:

1. If `Desc.[[Value]]` is absent, then
 - a. Return `OrdinaryDefineOwnProperty(A, "length", Desc)`.
2. Let `newLenDesc` be a copy of `Desc`.
3. Let `newLen` be ? `ToUInt32(Desc.[[Value]])`.
4. Let `numberLen` be ? `ToNumber(Desc.[[Value]])`.
5. If `newLen ≠ numberLen`, throw a **RangeError** exception.
6. Set `newLenDesc.[[Value]]` to `newLen`.
7. Let `oldLenDesc` be `OrdinaryGetOwnProperty(A, "length")`.
8. **Assert:** `oldLenDesc` will never be **undefined** or an accessor descriptor because `Array` objects are created with a `length` data property that cannot be deleted or reconfigured.
9. Let `oldLen` be `oldLenDesc.[[Value]]`.
10. If `newLen ≥ oldLen`, then
 - a. Return `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
11. If `oldLenDesc.[[Writable]]` is **false**, return **false**.
12. If `newLenDesc.[[Writable]]` is absent or has the value **true**, let `newWritable` be **true**.
13. Else,
 - a. Need to defer setting the `[[Writable]]` attribute to **false** in case any elements cannot be deleted.
 - b. Let `newWritable` be **false**.
 - c. Set `newLenDesc.[[Writable]]` to **true**.
14. Let `succeeded` be ! `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
15. If `succeeded` is **false**, return **false**.
16. Repeat, while `newLen < oldLen`,
 - a. Decrease `oldLen` by 1.
 - b. Let `deleteSucceeded` be ! `A.[[Delete]](! ToString(oldLen))`.
 - c. If `deleteSucceeded` is **false**, then
 - i. Set `newLenDesc.[[Value]]` to `oldLen + 1`.
 - ii. If `newWritable` is **false**, set `newLenDesc.[[Writable]]` to **false**.
 - iii. Perform ! `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
 - iv. Return **false**.
17. If `newWritable` is **false**, then
 - a. Return `OrdinaryDefineOwnProperty(A, "length", PropertyDescriptor { [[Writable]]: false })`. This call will always return **true**.
18. Return **true**.

NOTE

In steps 3 and 4, if `Desc.[[Value]]` is an object then its `valueOf` method is called twice. This is legacy behaviour that was specified with this effect starting with the 2nd Edition of this specification.

9.4.3 String Exotic Objects

- a. Add P as the last element of $keys$.
9. Return $keys$.

9.4.3.4 StringCreate ($value$, $prototype$)

The abstract operation StringCreate with arguments $value$ and $prototype$ is used to specify the creation of new String exotic objects. It performs the following steps:

1. **Assert:** $\text{Type}(value)$ is String.
2. Let S be a newly created String *exotic object*.
3. Set $S.[[StringData]]$ to $value$.
4. Set S 's essential internal methods to the default ordinary object definitions specified in 9.1.
5. Set $S.[[GetOwnProperty]]$ as specified in 9.4.3.1.
6. Set $S.[[DefineOwnProperty]]$ as specified in 9.4.3.2.
7. Set $S.[[OwnPropertyKeys]]$ as specified in 9.4.3.3.
8. Set $S.[[Prototype]]$ to $prototype$.
9. Set $S.[[Extensible]]$ to **true**.
10. Let $length$ be the number of code unit elements in $value$.
11. Perform ! $\text{DefinePropertyOrThrow}(S, "length", \text{PropertyDescriptor} \{ [[Value]]: length, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \})$.
12. Return S .

9.4.3.5 StringGetOwnProperty (S, P)

The abstract operation StringGetOwnProperty called with arguments S and P performs the following steps:

1. **Assert:** S is an Object that has a $[[[StringData]]$ internal slot.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. If $\text{Type}(P)$ is not String, return **undefined**.
4. Let $index$ be ! $\text{CanonicalNumericIndexString}(P)$.
5. If $index$ is **undefined**, return **undefined**.
6. If $\text{IsInteger}(index)$ is **false**, return **undefined**.
7. If $index = -0$, return **undefined**.
8. Let str be $S.[[StringData]]$.
9. **Assert:** $\text{Type}(str)$ is String.
10. Let len be the length of str .
11. If $index < 0$ or $len \leq index$, return **undefined**.
12. Let $resultStr$ be the String value of length 1, containing one code unit from str , specifically the code unit at index $index$.
13. Return a $\text{PropertyDescriptor} \{ [[Value]]: resultStr, [[Writable]]: false, [[Enumerable]]: true, [[Configurable]]: false \}$.

9.4.4 Arguments Exotic Objects

Most ECMAScript functions make an arguments object available to their code. Depending upon the characteristics of the function definition, its arguments object is either an ordinary object or an *arguments exotic object*. An arguments *exotic object* is an *exotic object* whose *array index* properties map to the formal parameters bindings of an invocation of its associated ECMAScript function.

The [[DefineOwnProperty]] internal method of an arguments *exotic object* when called with a property key *P* and *Property Descriptor Desc* performs the following steps:

1. Let *args* be the arguments object.
2. Let *map* be *args*.[[ParameterMap]].
3. Let *isMapped* be HasOwnProperty(*map*, *P*).
4. Let *newArgDesc* be *Desc*.
5. If *isMapped* is **true** and IsDataDescriptor(*Desc*) is **true**, then
 - a. If *Desc*.[[Value]] is not present and *Desc*.[[Writable]] is present and its value is **false**, then
 - i. Set *newArgDesc* to a copy of *Desc*.
 - ii. Set *newArgDesc*.[[Value]] to Get(*map*, *P*).
6. Let *allowed* be ? OrdinaryDefineOwnProperty(*args*, *P*, *newArgDesc*).
7. If *allowed* is **false**, return **false**.
8. If *isMapped* is **true**, then
 - a. If IsAccessorDescriptor(*Desc*) is **true**, then
 - i. Call *map*.[[Delete]](*P*).
 - b. Else,
 - i. If *Desc*.[[Value]] is present, then
 1. Let *setStatus* be Set(*map*, *P*, *Desc*.[[Value]], **false**).
 2. **Assert:** *setStatus* is **true** because formal parameters mapped by argument objects are always writable.
 - ii. If *Desc*.[[Writable]] is present and its value is **false**, then
 1. Call *map*.[[Delete]](*P*).
9. Return **true**.

9.4.4.3 [[Get]] (*P*, *Receiver*)

The [[Get]] internal method of an arguments *exotic object* when called with a property key *P* and ECMAScript language value *Receiver* performs the following steps:

1. Let *args* be the arguments object.
2. Let *map* be *args*.[[ParameterMap]].
3. Let *isMapped* be ! HasOwnProperty(*map*, *P*).
4. If *isMapped* is **false**, then
 - a. Return ? OrdinaryGet(*args*, *P*, *Receiver*).
5. Else *map* contains a formal parameter mapping for *P*,
 - a. Return Get(*map*, *P*).

9.4.4.4 [[Set]] (*P*, *V*, *Receiver*)

The [[Set]] internal method of an arguments *exotic object* when called with property key *P*, value *V*, and ECMAScript language value *Receiver* performs the following steps:

1. Let *args* be the arguments object.
2. If SameValue(*args*, *Receiver*) is **false**, then
 - a. Let *isMapped* be **false**.
3. Else,
 - a. Let *map* be *args*.[[ParameterMap]].
 - b. Let *isMapped* be ! HasOwnProperty(*map*, *P*).

6. Set *obj*.[[Get]] as specified in 9.4.4.3.
7. Set *obj*.[[Set]] as specified in 9.4.4.4.
8. Set *obj*.[[Delete]] as specified in 9.4.4.5.
9. Set the remainder of *obj*'s essential internal methods to the default ordinary object definitions specified in 9.1.
10. Set *obj*.[[Prototype]] to %ObjectPrototype%.
11. Set *obj*.[[Extensible]] to **true**.
12. Let *map* be ObjectCreate(null).
13. Set *obj*.[[ParameterMap]] to *map*.
14. Let *parameterNames* be the BoundNames of *formals*.
15. Let *numberOfParameters* be the number of elements in *parameterNames*.
16. Let *index* be 0.
17. Repeat, while *index* < *len*,
 - a. Let *val* be *argumentsList*[*index*].
 - b. Perform CreateDataProperty(*obj*, ! ToString(*index*), *val*).
 - c. Increase *index* by 1.
18. Perform DefinePropertyOrThrow(*obj*, "length", PropertyDescriptor { [[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
19. Let *mappedNames* be a new empty List.
20. Let *index* be *numberOfParameters* - 1.
21. Repeat, while *index* ≥ 0,
 - a. Let *name* be *parameterNames*[*index*].
 - b. If *name* is not an element of *mappedNames*, then
 - i. Add *name* as an element of the list *mappedNames*.
 - ii. If *index* < *len*, then
 1. Let *g* be MakeArgGetter(*name*, *env*).
 2. Let *p* be MakeArgSetter(*name*, *env*).
 3. Perform *map*.[[DefineOwnProperty]](! ToString(*index*), PropertyDescriptor { [[Set]]: *p*, [[Get]]: *g*, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
 - c. Decrease *index* by 1.
22. Perform ! DefinePropertyOrThrow(*obj*, @@iterator, PropertyDescriptor { [[Value]]: %ArrayProto_values%, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
23. Perform ! DefinePropertyOrThrow(*obj*, "callee", PropertyDescriptor { [[Value]]: *func*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
24. Return *obj*.

9.4.4.7.1 MakeArgGetter (*name*, *env*)

The abstract operation MakeArgGetter called with String *name* and Environment Record *env* creates a built-in function object that when executed returns the value bound for *name* in *env*. It performs the following steps:

1. Let *steps* be the steps of an ArgGetter function as specified below.
2. Let *getter* be CreateBuiltinFunction(*steps*, « [[Name]], [[Env]] »).
3. Set *getter*.[[Name]] to *name*.
4. Set *getter*.[[Env]] to *env*.
5. Return *getter*.

An ArgGetter function is an anonymous built-in function with [[Name]] and [[Env]] internal slots. When an ArgGetter function that expects no arguments is called it performs the following steps:

- b. If *numericIndex* is not **undefined**, then
 - i. Let *value* be ? *IntegerIndexedElementGet*(*O*, *numericIndex*).
 - ii. If *value* is **undefined**, return **undefined**.
 - iii. Return a *PropertyDescriptor* { [[Value]]: *value*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **false** }.
- 4. Return *OrdinaryGetOwnProperty*(*O*, *P*).

9.4.5.2 [[HasProperty]] (*P*)

When the [[HasProperty]] internal method of an *Integer-Indexed exotic object O* is called with property key *P*, the following steps are taken:

- 1. **Assert:** *IsPropertyKey*(*P*) is **true**.
- 2. **Assert:** *O* is an Object that has a [[ViewedArrayBuffer]] internal slot.
- 3. If *Type*(*P*) is String, then
 - a. Let *numericIndex* be ! *CanonicalNumericIndexString*(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. Let *buffer* be *O*.[[ViewedArrayBuffer]].
 - ii. If *IsDetachedBuffer*(*buffer*) is **true**, throw a **TypeError** exception.
 - iii. If *IsInteger*(*numericIndex*) is **false**, return **false**.
 - iv. If *numericIndex* = -0, return **false**.
 - v. If *numericIndex* < 0, return **false**.
 - vi. If *numericIndex* ≥ *O*.[[ArrayLength]], return **false**.
 - vii. Return **true**.
- 4. Return ? *OrdinaryHasProperty*(*O*, *P*).

9.4.5.3 [[DefineOwnProperty]] (*P*, *Desc*)

When the [[DefineOwnProperty]] internal method of an *Integer-Indexed exotic object O* is called with property key *P*, and *Property Descriptor Desc*, the following steps are taken:

- 1. **Assert:** *IsPropertyKey*(*P*) is **true**.
- 2. **Assert:** *O* is an Object that has a [[ViewedArrayBuffer]] internal slot.
- 3. If *Type*(*P*) is String, then
 - a. Let *numericIndex* be ! *CanonicalNumericIndexString*(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. If *IsInteger*(*numericIndex*) is **false**, return **false**.
 - ii. If *numericIndex* = -0, return **false**.
 - iii. If *numericIndex* < 0, return **false**.
 - iv. Let *length* be *O*.[[ArrayLength]].
 - v. If *numericIndex* ≥ *length*, return **false**.
 - vi. If *IsAccessorDescriptor*(*Desc*) is **true**, return **false**.
 - vii. If *Desc* has a [[Configurable]] field and if *Desc*.[[Configurable]] is **true**, return **false**.
 - viii. If *Desc* has an [[Enumerable]] field and if *Desc*.[[Enumerable]] is **false**, return **false**.
 - ix. If *Desc* has a [[Writable]] field and if *Desc*.[[Writable]] is **false**, return **false**.
 - x. If *Desc* has a [[Value]] field, then
 - 1. Let *value* be *Desc*.[[Value]].
 - 2. Return ? *IntegerIndexedElementSet*(*O*, *numericIndex*, *value*).
 - xi. Return **true**.

1. **Assert:** *internalSlotsList* contains the names [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]].
2. Let *A* be a newly created object with an internal slot for each name in *internalSlotsList*.
3. Set *A*'s essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set *A*.[[GetOwnProperty]] as specified in 9.4.5.1.
5. Set *A*.[[HasProperty]] as specified in 9.4.5.2.
6. Set *A*.[[DefineOwnProperty]] as specified in 9.4.5.3.
7. Set *A*.[[Get]] as specified in 9.4.5.4.
8. Set *A*.[[Set]] as specified in 9.4.5.5.
9. Set *A*.[[OwnPropertyKeys]] as specified in 9.4.5.6.
10. Set *A*.[[Prototype]] to *prototype*.
11. Set *A*.[[Extensible]] to **true**.
12. Return *A*.

9.4.5.8 IntegerIndexedElementGet (*O*, *index*)

The abstract operation IntegerIndexedElementGet with arguments *O* and *index* performs the following steps:

1. **Assert:** *Type(index)* is Number.
2. **Assert:** *O* is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let *buffer* be *O*.[[ViewedArrayBuffer]].
4. If *IsDetachedBuffer(buffer)* is **true**, throw a **TypeError** exception.
5. If *IsInteger(index)* is **false**, return **undefined**.
6. If *index* = -0, return **undefined**.
7. Let *length* be *O*.[[ArrayLength]].
8. If *index* < 0 or *index* ≥ *length*, return **undefined**.
9. Let *offset* be *O*.[[ByteOffset]].
10. Let *arrayTypeName* be the String value of *O*.[[TypedArrayName]].
11. Let *elementSize* be the Number value of the Element Size value specified in Table 59 for *arrayTypeName*.
12. Let *indexedPosition* be (*index* × *elementSize*) + *offset*.
13. Let *elementType* be the String value of the Element Type value in Table 59 for *arrayTypeName*.
14. Return *GetValueFromBuffer(buffer, indexedPosition, elementType, true, "Unordered")*.

9.4.5.9 IntegerIndexedElementSet (*O*, *index*, *value*)

The abstract operation IntegerIndexedElementSet with arguments *O*, *index*, and *value* performs the following steps:

1. **Assert:** *Type(index)* is Number.
2. **Assert:** *O* is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let *numValue* be ? *ToNumber(value)*.
4. Let *buffer* be *O*.[[ViewedArrayBuffer]].
5. If *IsDetachedBuffer(buffer)* is **true**, throw a **TypeError** exception.
6. If *IsInteger(index)* is **false**, return **false**.
7. If *index* = -0, return **false**.
8. Let *length* be *O*.[[ArrayLength]].
9. If *index* < 0 or *index* ≥ *length*, return **false**.
10. Let *offset* be *O*.[[ByteOffset]].

9.4.6.3 [[PreventExtensions]] ()

When the [[PreventExtensions]] internal method of a module namespace *exotic object O* is called, the following steps are taken:

1. Return **true**.

9.4.6.4 [[GetOwnProperty]] (*P*)

When the [[GetOwnProperty]] internal method of a module namespace *exotic object O* is called with property key *P*, the following steps are taken:

1. If *Type(P)* is Symbol, return *OrdinaryGetOwnProperty(O, P)*.
2. Let *exports* be *O.[[Exports]]*.
3. If *P* is not an element of *exports*, return **undefined**.
4. Let *value* be ? *O.[[Get]](P, O)*.
5. Return *PropertyDescriptor* { [[Value]]: *value*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **false** }.

9.4.6.5 [[DefineOwnProperty]] (*P, Desc*)

When the [[DefineOwnProperty]] internal method of a module namespace *exotic object O* is called with property key *P* and *Property Descriptor Desc*, the following steps are taken:

1. If *Type(P)* is Symbol, return *OrdinaryDefineOwnProperty(O, P, Desc)*.
2. Let *current* be ? *O.[[GetOwnProperty]](P)*.
3. If *current* is **undefined**, return **false**.
4. If *IsAccessorDescriptor(Desc)* is **true**, return **false**.
5. If *Desc.[[Writable]]* is present and has value **false**, return **false**.
6. If *Desc.[[Enumerable]]* is present and has value **false**, return **false**.
7. If *Desc.[[Configurable]]* is present and has value **true**, return **false**.
8. If *Desc.[[Value]]* is present, return *SameValue(Desc.[[Value]], current.[[Value]])*.
9. Return **true**.

9.4.6.6 [[HasProperty]] (*P*)

When the [[HasProperty]] internal method of a module namespace *exotic object O* is called with property key *P*, the following steps are taken:

1. If *Type(P)* is Symbol, return *OrdinaryHasProperty(O, P)*.
2. Let *exports* be *O.[[Exports]]*.
3. If *P* is an element of *exports*, return **true**.
4. Return **false**.

9.4.6.7 [[Get]] (*P, Receiver*)

When the [[Get]] internal method of a module namespace *exotic object O* is called with property key *P* and ECMAScript language value *Receiver*, the following steps are taken:

1. **Assert:** *IsPropertyKey(P)* is **true**.
2. If *Type(P)* is Symbol, then
 - a. Return ? *OrdinaryGet(O, P, Receiver)*.

1. **Assert:** *module* is a **Module Record**.
2. **Assert:** *module*.[[Namespace]] is **undefined**.
3. **Assert:** *exports* is a **List** of String values.
4. Let *M* be a newly created object.
5. Set *M*'s essential internal methods to the definitions specified in 9.4.6.
6. Set *M*.[[Module]] to *module*.
7. Let *sortedExports* be a new **List** containing the same values as the list *exports* where the values are ordered as if an Array of the same values had been sorted using **Array.prototype.sort** using **undefined** as *comparefn*.
8. Set *M*.[[Exports]] to *sortedExports*.
9. Create own properties of *M* corresponding to the definitions in 26.3.
10. Set *module*.[[Namespace]] to *M*.
11. Return *M*.

9.4.7 Immutable Prototype Exotic Objects

An immutable prototype exotic object is an **exotic object** that has a [[Prototype]] internal slot that will not change once it is initialized.

Immutable prototype exotic objects have the same internal slots as ordinary objects. They are exotic only in the following internal methods. All other internal methods of immutable prototype exotic objects that are not explicitly defined below are instead defined as in **ordinary objects**.

9.4.7.1 [[SetPrototypeOf]] (*V*)

When the [[SetPrototypeOf]] internal method of an **immutable prototype exotic object** *O* is called with argument *V*, the following steps are taken:

1. Return ? **SetImmutablePrototype**(*O*, *V*).

9.4.7.2 SetImmutablePrototype (*O*, *V*)

When the **SetImmutablePrototype** abstract operation is called with arguments *O* and *V*, the following steps are taken:

1. **Assert:** Either **Type**(*V*) is Object or **Type**(*V*) is Null.
2. Let *current* be ? *O*.[[GetPrototypeOf]]().
3. If **SameValue**(*V*, *current*) is **true**, return **true**.
4. Return **false**.

9.5 Proxy Object Internal Methods and Internal Slots

A proxy object is an **exotic object** whose essential internal methods are partially implemented using ECMAScript code. Every proxy object has an internal slot called [[ProxyHandler]]. The value of [[ProxyHandler]] is an object, called the proxy's *handler object*, or **null**. Methods (see Table 30) of a handler object may be used to augment the implementation for one or more of the proxy object's internal methods. Every proxy object also has an internal slot called [[ProxyTarget]] whose value is either an object or the **null** value. This object is called the proxy's *target object*.

Table 30: Proxy Handler Methods

Internal Method	Handler Method
-----------------	----------------

4. Let `target` be $O.[[ProxyTarget]]$.
5. Let `trap` be ? `GetMethod(handler, "getPrototypeOf")`.
6. If `trap` is **undefined**, then
 - a. Return ? `target.[[GetPrototypeOf]]()`.
7. Let `handlerProto` be ? `Call(trap, handler, « target »)`.
8. If `Type(handlerProto)` is neither Object nor Null, throw a **TypeError** exception.
9. Let `extensibleTarget` be ? `IsExtensible(target)`.
10. If `extensibleTarget` is **true**, return `handlerProto`.
11. Let `targetProto` be ? `target.[[GetPrototypeOf]]()`.
12. If `SameValue(handlerProto, targetProto)` is **false**, throw a **TypeError** exception.
13. Return `handlerProto`.

NOTE

`[[GetPrototypeOf]]` for proxy objects enforces the following invariants:

The result of `[[GetPrototypeOf]]` must be either an Object or **null**.

If the target object is not extensible, `[[GetPrototypeOf]]` applied to the proxy object must return the same value as `[[GetPrototypeOf]]` applied to the proxy object's target object.

9.5.2 `[[SetPrototypeOf]]` (V)

When the `[[SetPrototypeOf]]` internal method of a Proxy exotic object O is called with argument V , the following steps are taken:

1. **Assert:** Either `Type(V)` is Object or `Type(V)` is Null.
2. Let `handler` be $O.[[ProxyHandler]]$.
3. If `handler` is **null**, throw a **TypeError** exception.
4. **Assert:** `Type(handler)` is Object.
5. Let `target` be $O.[[ProxyTarget]]$.
6. Let `trap` be ? `GetMethod(handler, "setPrototypeOf")`.
7. If `trap` is **undefined**, then
 - a. Return ? `target.[[SetPrototypeOf]](V)`.
8. Let `booleanTrapResult` be `ToBoolean(? Call(trap, handler, « target, V »))`.
9. If `booleanTrapResult` is **false**, return **false**.
10. Let `extensibleTarget` be ? `IsExtensible(target)`.
11. If `extensibleTarget` is **true**, return **true**.
12. Let `targetProto` be ? `target.[[GetPrototypeOf]]()`.
13. If `SameValue(V, targetProto)` is **false**, throw a **TypeError** exception.
14. Return **true**.

NOTE

`[[SetPrototypeOf]]` for proxy objects enforces the following invariants:

The result of `[[SetPrototypeOf]]` is a Boolean value.

If the target object is not extensible, the argument value must be the same as the result of `[[GetPrototypeOf]]` applied to target object.

9.5.5 [[GetOwnProperty]] (*P*)

When the [[GetOwnProperty]] internal method of a Proxy *exotic object O* is called with property key *P*, the following steps are taken:

1. **Assert:** IsPropertyKey(*P*) is **true**.
2. Let *handler* be *O*.[[ProxyHandler]].
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert:** Type(*handler*) is Object.
5. Let *target* be *O*.[[ProxyTarget]].
6. Let *trap* be ? GetMethod(*handler*, "getOwnPropertyDescriptor").
7. If *trap* is **undefined**, then
 - a. Return ? *target*.[[GetOwnProperty]](*P*).
8. Let *trapResultObj* be ? Call(*trap*, *handler*, « *target*, *P* »).
9. If Type(*trapResultObj*) is neither Object nor Undefined, throw a **TypeError** exception.
10. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
11. If *trapResultObj* is **undefined**, then
 - a. If *targetDesc* is **undefined**, return **undefined**.
 - b. If *targetDesc*.[[Configurable]] is **false**, throw a **TypeError** exception.
 - c. Let *extensibleTarget* be ? IsExtensible(*target*).
 - d. If *extensibleTarget* is **false**, throw a **TypeError** exception.
 - e. Return **undefined**.
12. Let *extensibleTarget* be ? IsExtensible(*target*).
13. Let *resultDesc* be ? ToPropertyDescriptor(*trapResultObj*).
14. Call CompletePropertyDescriptor(*resultDesc*).
15. Let *valid* be IsCompatiblePropertyDescriptor(*extensibleTarget*, *resultDesc*, *targetDesc*).
16. If *valid* is **false**, throw a **TypeError** exception.
17. If *resultDesc*.[[Configurable]] is **false**, then
 - a. If *targetDesc* is **undefined** or *targetDesc*.[[Configurable]] is **true**, then
 - i. Throw a **TypeError** exception.
18. Return *resultDesc*.

NOTE

[[GetOwnProperty]] for proxy objects enforces the following invariants:

The result of [[GetOwnProperty]] must be either an Object or **undefined**.

A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.

A property cannot be reported as non-existent, if it exists as an own property of the target object and the target object is not extensible.

A property cannot be reported as existent, if it does not exist as an own property of the target object and the target object is not extensible.

A property cannot be reported as non-configurable, if it does not exist as an own property of the target object or if it exists as a configurable own property of the target object.

9.5.6 [[DefineOwnProperty]] (*P*, *Desc*)

When the [[DefineOwnProperty]] internal method of a Proxy *exotic object O* is called with property key *P* and *Property Descriptor Desc*, the following steps are taken:

- a. Return ? *target*.[[HasProperty]](*P*).
8. Let *booleanTrapResult* be ToBoolean(? Call(*trap*, *handler*, « *target*, *P* »)).
9. If *booleanTrapResult* is **false**, then
 - a. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
 - b. If *targetDesc* is not **undefined**, then
 - i. If *targetDesc*.[[Configurable]] is **false**, throw a **TypeError** exception.
 - ii. Let *extensibleTarget* be ? IsExtensible(*target*).
 - iii. If *extensibleTarget* is **false**, throw a **TypeError** exception.
10. Return *booleanTrapResult*.

NOTE

[[HasProperty]] for proxy objects enforces the following invariants:

The result of [[HasProperty]] is a Boolean value.

A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
 A property cannot be reported as non-existent, if it exists as an own property of the target object and the target object is not extensible.

9.5.8 [[Get]] (*P*, *Receiver*)

When the [[Get]] internal method of a Proxy exotic object *O* is called with property key *P* and ECMAScript language value *Receiver*, the following steps are taken:

1. **Assert:** IsPropertyKey(*P*) is **true**.
2. Let *handler* be *O*.[[ProxyHandler]].
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert:** Type(*handler*) is Object.
5. Let *target* be *O*.[[ProxyTarget]].
6. Let *trap* be ? GetMethod(*handler*, "get").
7. If *trap* is **undefined**, then
 - a. Return ? *target*.[[Get]](*P*, *Receiver*).
8. Let *trapResult* be ? Call(*trap*, *handler*, « *target*, *P*, *Receiver* »).
9. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
10. If *targetDesc* is not **undefined** and *targetDesc*.[[Configurable]] is **false**, then
 - a. If IsDataDescriptor(*targetDesc*) is **true** and *targetDesc*.[[Writable]] is **false**, then
 - i. If SameValue(*trapResult*, *targetDesc*.[[Value]]) is **false**, throw a **TypeError** exception.
 - b. If IsAccessorDescriptor(*targetDesc*) is **true** and *targetDesc*.[[Get]] is **undefined**, then
 - i. If *trapResult* is not **undefined**, throw a **TypeError** exception.
11. Return *trapResult*.

NOTE

[[Get]] for proxy objects enforces the following invariants:

The value reported for a property must be the same as the value of the corresponding target object property if the target object property is a non-writable, non-configurable own **data property**.

The value reported for a property must be **undefined** if the corresponding target object property is a non-configurable own **accessor property** that has **undefined** as its [[Get]] attribute.

8. Let *booleanTrapResult* be `ToBoolean(? Call(trap, handler, « target, P »)).`
9. If *booleanTrapResult* is **false**, return **false**.
10. Let *targetDesc* be `? target.[[GetOwnProperty]](P)`.
11. If *targetDesc* is **undefined**, return **true**.
12. If *targetDesc*.[[Configurable]] is **false**, throw a **TypeError** exception.
13. Return **true**.

NOTE

`[[Delete]]` for proxy objects enforces the following invariants:

The result of `[[Delete]]` is a Boolean value.

A property cannot be reported as deleted, if it exists as a non-configurable own property of the target object.

9.5.11 `[[OwnPropertyKeys]]()`

When the `[[OwnPropertyKeys]]` internal method of a Proxy exotic object *O* is called, the following steps are taken:

1. Let *handler* be *O*.[[ProxyHandler]].
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** `Type(handler)` is Object.
4. Let *target* be *O*.[[ProxyTarget]].
5. Let *trap* be `? GetMethod(handler, "ownKeys")`.
6. If *trap* is **undefined**, then
 - a. Return `? target.[[OwnPropertyKeys]]()`.
7. Let *trapResultArray* be `? Call(trap, handler, « target »)`.
8. Let *trapResult* be `? CreateListFromArrayLike(trapResultArray, « String, Symbol »)`.
9. If *trapResult* contains any duplicate entries, throw a **TypeError** exception.
10. Let *extensibleTarget* be `? IsExtensible(target)`.
11. Let *targetKeys* be `? target.[[OwnPropertyKeys]]()`.
12. **Assert:** *targetKeys* is a List containing only String and Symbol values.
13. **Assert:** *targetKeys* contains no duplicate entries.
14. Let *targetConfigurableKeys* be a new empty List.
15. Let *targetNonconfigurableKeys* be a new empty List.
16. For each element *key* of *targetKeys*, do
 - a. Let *desc* be `? target.[[GetOwnProperty]](key)`.
 - b. If *desc* is not **undefined** and *desc*.[[Configurable]] is **false**, then
 - i. Append *key* as an element of *targetNonconfigurableKeys*.
 - c. Else,
 - i. Append *key* as an element of *targetConfigurableKeys*.
17. If *extensibleTarget* is **true** and *targetNonconfigurableKeys* is empty, then
 - a. Return *trapResult*.
18. Let *uncheckedResultKeys* be a new List which is a copy of *trapResult*.
19. For each *key* that is an element of *targetNonconfigurableKeys*, do
 - a. If *key* is not an element of *uncheckedResultKeys*, throw a **TypeError** exception.
 - b. Remove *key* from *uncheckedResultKeys*.
20. If *extensibleTarget* is **true**, return *trapResult*.
21. For each *key* that is an element of *targetConfigurableKeys*, do
 - a. If *key* is not an element of *uncheckedResultKeys*, throw a **TypeError** exception.
 - b. Remove *key* from *uncheckedResultKeys*.

8. Let `argArray` be `CreateArrayFromList(argumentsList)`.
9. Let `newObj` be ? `Call(trap, handler, « target, argArray, newTarget »)`.
10. If `Type(newObj)` is not Object, throw a **TypeError** exception.
11. Return `newObj`.

NOTE 1

A Proxy **exotic object** only has a `[[Construct]]` internal method if the initial value of its `[[ProxyTarget]]` internal slot is an object that has a `[[Construct]]` internal method.

NOTE 2

`[[Construct]]` for proxy objects enforces the following invariants:

The result of `[[Construct]]` must be an Object.

9.5.14 ProxyCreate (`target`, `handler`)

The abstract operation `ProxyCreate` with arguments `target` and `handler` is used to specify the creation of new Proxy exotic objects. It performs the following steps:

1. If `Type(target)` is not Object, throw a **TypeError** exception.
2. If `target` is a Proxy **exotic object** and `target.[[ProxyHandler]]` is `null`, throw a **TypeError** exception.
3. If `Type(handler)` is not Object, throw a **TypeError** exception.
4. If `handler` is a Proxy **exotic object** and `handler.[[ProxyHandler]]` is `null`, throw a **TypeError** exception.
5. Let `P` be a newly created object.
6. Set `P`'s essential internal methods (except for `[[Call]]` and `[[Construct]]`) to the definitions specified in 9.5.
7. If `IsCallable(target)` is `true`, then
 - a. Set `P.[[Call]]` as specified in 9.5.12.
 - b. If `IsConstructor(target)` is `true`, then
 - i. Set `P.[[Construct]]` as specified in 9.5.13.
8. Set `P.[[ProxyTarget]]` to `target`.
9. Set `P.[[ProxyHandler]]` to `handler`.
10. Return `P`.

10 ECMAScript Language: Source Code

10.1 Source Text

Syntax

SourceCharacter ::

any Unicode code point

ECMAScript code is expressed using Unicode. ECMAScript source text is a sequence of code points. All Unicode code point values from U+0000 to U+10FFFF, including surrogate code points, may occur in source text where permitted by the ECMAScript grammars. The actual encodings used to store and interchange ECMAScript source text is not relevant to this specification. Regardless of the external source text encoding, a conforming ECMAScript implementation

GeneratorExpression, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

Eval code is the source text supplied to the built-in **eval** function. More precisely, if the parameter to the built-in **eval** function is a String, it is treated as an ECMAScript *Script*. The eval code for a particular invocation of **eval** is the global code portion of that *Script*.

Function code is source text that is parsed to supply the value of the **[[ECMAScriptCode]]** and **[[FormalParameters]]** internal slots (see 9.2) of an ECMAScript **function object**. The function code of a particular ECMAScript function does not include any source text that is parsed as the function code of a nested *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

Module code is source text that is code that is provided as a *ModuleBody*. It is the code that is directly evaluated when a module is initialized. The module code of a particular module does not include any source text that is parsed as part of a nested *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

NOTE

Function code is generally provided as the bodies of Function Definitions (14.1), Arrow Function Definitions (14.2), Method Definitions (14.3), Generator Function Definitions (14.4), Async Function Definitions (14.7), Async Generator Function Definitions (14.5), and Async Arrow Functions (14.8). Function code is also derived from the arguments to the **Function constructor** (19.2.1.1), the **GeneratorFunction constructor** (25.2.1.1), and the **AsyncFunction constructor** (25.7.1.1).

10.2.1 Strict Mode Code

An ECMAScript *Script* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. Code is interpreted as strict mode code in the following situations:

Global code is strict mode code if it begins with a **Directive Prologue** that contains a **Use Strict Directive**.

Module code is always strict mode code.

All parts of a *ClassDeclaration* or a *ClassExpression* are strict mode code.

Eval code is strict mode code if it begins with a **Directive Prologue** that contains a **Use Strict Directive** or if the call to **eval** is a **direct eval** that is contained in strict mode code.

Function code is strict mode code if the associated *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, or *AsyncArrowFunction* is contained in strict mode code or if the code that produces the value of the function's **[[ECMAScriptCode]]** internal slot begins with a **Directive Prologue** that contains a **Use Strict Directive**.

Function code that is supplied as the arguments to the built-in **Function**, **Generator**, **AsyncFunction**, and **AsyncGenerator** constructors is strict mode code if the last argument is a String that when processed is a *FunctionBody* that begins with a **Directive Prologue** that contains a **Use Strict Directive**.

ECMAScript code that is not strict mode code is called non-strict code.

10.2.2 Non-ECMAScript Functions


```

WhiteSpace
LineTerminator
Comment
CommonToken
RightBracePunctuator
RegularExpressionLiteral

```

InputElementRegExpOrTemplateTail ::

```

WhiteSpace
LineTerminator
Comment
CommonToken
RegularExpressionLiteral
TemplateSubstitutionTail

```

InputElementTemplateTail ::

```

WhiteSpace
LineTerminator
Comment
CommonToken
DivPunctuator
TemplateSubstitutionTail

```

11.1 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 format-control characters in source text to facilitate editing and display. All format control characters may be used within comments, and within string literals, template literals, and regular expression literals.

U+200C (ZERO WIDTH NON-JOINER) and U+200D (ZERO WIDTH JOINER) are format-control characters that are used to make necessary distinctions when forming words or phrases in certain languages. In ECMAScript source text these code points may also be used in an *IdentifierName* after the first character.

U+FEFF (ZERO WIDTH NO-BREAK SPACE) is a format-control character used primarily at the start of a text to mark it as Unicode and to allow detection of the text's encoding and byte order. <ZWNBSP> characters intended for this purpose can sometimes also appear after the start of a text, for example as a result of concatenating files. In ECMAScript source text <ZWNBSP> code points are treated as white space characters (see [11.2](#)).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarized in [Table 31](#).

Table 31: Format-Control Code Point Usage

Code Point	Name	Abbreviation	Usage
U+200C	ZERO WIDTH NON-JOINER	<ZWNJ>	<i>IdentifierPart</i>
U+200D	ZERO WIDTH JOINER	<ZWJ>	<i>IdentifierPart</i>

11.3 Line Terminators

Like white space code points, line terminator code points are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space code points, 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. Line terminators also affect the process of automatic semicolon insertion (11.9). A line terminator cannot occur within any token except a *StringLiteral*, *Template*, or *TemplateSubstitutionTail*. <LF> and <CR> line terminators cannot occur within a *StringLiteral* token except as part of a *LineContinuation*.

A line terminator can occur within a *MultiLineComment* but cannot occur within a *SingleLineComment*.

Line terminators are included in the set of white space code points that are matched by the \s class in regular expressions.

The ECMAScript line terminator code points are listed in [Table 33](#).

Table 33: Line Terminator Code Points

Code Point	Unicode Name	Abbreviation
U+000A	LINE FEED (LF)	<LF>
U+000D	CARRIAGE RETURN (CR)	<CR>
U+2028	LINE SEPARATOR	<LS>
U+2029	PARAGRAPH SEPARATOR	<PS>

Only the Unicode code points in [Table 33](#) are treated as line terminators. Other new line or line breaking Unicode code points are not treated as line terminators but are treated as white space if they meet the requirements listed in [Table 32](#). The sequence <CR><LF> is commonly used as a line terminator. It should be considered a single *SourceCharacter* for the purpose of reporting line numbers.

Syntax

LineTerminator ::

<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence ::

<LF>
<CR>[lookahead ≠ <LF>]
<LS>
<PS>
<CR><LF>

11.4 Comments

Punctuator
NumericLiteral
StringLiteral
Template

NOTE

The *DivPunctuator*, *RegularExpressionLiteral*, *RightBracePunctuator*, and *TemplateSubstitutionTail* productions derive additional tokens that are not included in the *CommonToken* production.

11.6 Names and Keywords

IdentifierName and *ReservedWord* are tokens that are interpreted according to the Default Identifier Syntax given in Unicode Standard Annex #31, Identifier and Pattern Syntax, with some small modifications. *ReservedWord* is an enumerated subset of *IdentifierName*. The syntactic grammar defines *Identifier* as an *IdentifierName* that is not a *ReservedWord*. The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The Unicode code points in the specified categories in the latest version of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations. ECMAScript implementations may recognize identifier code points defined in later editions of the Unicode Standard.

NOTE 1

This standard specifies specific code point additions: U+0024 (DOLLAR SIGN) and U+005F (LOW LINE) are permitted anywhere in an *IdentifierName*, and the code points U+200C (ZERO WIDTH NON-JOINER) and U+200D (ZERO WIDTH JOINER) are permitted anywhere after the first code point of an *IdentifierName*.

Unicode escape sequences are permitted in an *IdentifierName*, where they contribute a single Unicode code point to the *IdentifierName*. The code point is expressed by the *CodePoint* of the *UnicodeEscapeSequence* (see 11.8.4). The \ preceding the *UnicodeEscapeSequence* and the u and { } code units, if they appear, do not contribute code points to the *IdentifierName*. A *UnicodeEscapeSequence* cannot be used to put a code point into an *IdentifierName* that would otherwise be illegal. In other words, if a \ *UnicodeEscapeSequence* sequence were replaced by the *SourceCharacter* it contributes, the result must still be a valid *IdentifierName* that has the exact same sequence of *SourceCharacter* elements as the original *IdentifierName*. All interpretations of *IdentifierName* within this specification are based upon their actual code points regardless of whether or not an escape sequence was used to contribute any particular code point.

Two *IdentifierNames* that are canonically equivalent according to the Unicode standard are *not* equal unless, after replacement of each *UnicodeEscapeSequence*, they are represented by the exact same sequence of code points.

Syntax

IdentifierName ::
 IdentifierStart
 IdentifierName IdentifierPart

IdentifierStart ::
 UnicodeIDStart
 \$
 —
 \ *UnicodeEscapeSequence*

Syntax

```
ReservedWord ::  
    Keyword  
    FutureReservedWord  
    NullLiteral  
    BooleanLiteral
```

NOTE

The *ReservedWord* definitions are specified as literal sequences of specific *SourceCharacter* elements. A code point in a *ReservedWord* cannot be expressed by a \ *UnicodeEscapeSequence*.

11.6.2.1 Keywords

The following tokens are ECMAScript keywords and may not be used as *Identifiers* in ECMAScript programs.

Syntax

```
Keyword :: one of  
    await break case catch class const continue debugger default delete do  
    else export extends finally for function if import in instanceof new  
    return super switch this throw try typeof var void while with yield
```

NOTE

In some contexts **yield** and **await** are given the semantics of an *Identifier*. See 12.1.1. In **strict mode code**, **let** and **static** are treated as reserved words through static semantic restrictions (see 12.1.1, 13.3.1.1, 13.7.5.1, and 14.6.1) rather than the lexical grammar.

11.6.2.2 Future Reserved Words

The following tokens are reserved for use as keywords in future language extensions.

Syntax

```
FutureReservedWord ::  
    enum
```

NOTE

Use of the following tokens within **strict mode code** is also reserved. That usage is restricted using static semantic restrictions (see 12.1.1) rather than the lexical grammar:

```
implements package protected  
interface private public
```

11.7 Punctuators

DecimalDigits ::
 DecimalDigit
 DecimalDigits DecimalDigit

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

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

ExponentPart ::
 ExponentIndicator SignedInteger

ExponentIndicator :: **one of**
 e E

SignedInteger ::
 DecimalDigits
 + DecimalDigits
 - DecimalDigits

BinaryIntegerLiteral ::
 0b BinaryDigits
 0B BinaryDigits

BinaryDigits ::
 BinaryDigit
 BinaryDigits BinaryDigit

BinaryDigit :: **one of**
 0 1

OctalIntegerLiteral ::
 0o OctalDigits
 0O OctalDigits

OctalDigits ::
 OctalDigit
 OctalDigits OctalDigit

OctalDigit :: **one of**
 0 1 2 3 4 5 6 7

HexIntegerLiteral ::
 0x HexDigits
 0X HexDigits

HexDigits ::
 HexDigit
 HexDigits HexDigit

HexDigit :: **one of**

The MV of `DecimalDigit :: 1` or of `NonZeroDigit :: 1` or of `HexDigit :: 1` or of `OctalDigit :: 1` or of `BinaryDigit :: 1` is 1.

The MV of `DecimalDigit :: 2` or of `NonZeroDigit :: 2` or of `HexDigit :: 2` or of `OctalDigit :: 2` is 2.

The MV of `DecimalDigit :: 3` or of `NonZeroDigit :: 3` or of `HexDigit :: 3` or of `OctalDigit :: 3` is 3.

The MV of `DecimalDigit :: 4` or of `NonZeroDigit :: 4` or of `HexDigit :: 4` or of `OctalDigit :: 4` is 4.

The MV of `DecimalDigit :: 5` or of `NonZeroDigit :: 5` or of `HexDigit :: 5` or of `OctalDigit :: 5` is 5.

The MV of `DecimalDigit :: 6` or of `NonZeroDigit :: 6` or of `HexDigit :: 6` or of `OctalDigit :: 6` is 6.

The MV of `DecimalDigit :: 7` or of `NonZeroDigit :: 7` or of `HexDigit :: 7` or of `OctalDigit :: 7` is 7.

The MV of `DecimalDigit :: 8` or of `NonZeroDigit :: 8` or of `HexDigit :: 8` is 8.

The MV of `DecimalDigit :: 9` or of `NonZeroDigit :: 9` or of `HexDigit :: 9` is 9.

The MV of `HexDigit :: a` or of `HexDigit :: A` is 10.

The MV of `HexDigit :: b` or of `HexDigit :: B` is 11.

The MV of `HexDigit :: c` or of `HexDigit :: C` is 12.

The MV of `HexDigit :: d` or of `HexDigit :: D` is 13.

The MV of `HexDigit :: e` or of `HexDigit :: E` is 14.

The MV of `HexDigit :: f` or of `HexDigit :: F` is 15.

The MV of `BinaryIntegerLiteral :: 0b BinaryDigits` is the MV of `BinaryDigits`.

The MV of `BinaryIntegerLiteral :: 0B BinaryDigits` is the MV of `BinaryDigits`.

The MV of `BinaryDigits :: BinaryDigit` is the MV of `BinaryDigit`.

The MV of `BinaryDigits :: BinaryDigits BinaryDigit` is (the MV of `BinaryDigits` × 2) plus the MV of `BinaryDigit`.

The MV of `OctalIntegerLiteral :: 0o OctalDigits` is the MV of `OctalDigits`.

The MV of `OctalIntegerLiteral :: 0O OctalDigits` is the MV of `OctalDigits`.

The MV of `OctalDigits :: OctalDigit` is the MV of `OctalDigit`.

The MV of `OctalDigits :: OctalDigits OctalDigit` is (the MV of `OctalDigits` × 8) plus the MV of `OctalDigit`.

The MV of `HexIntegerLiteral :: 0x HexDigits` is the MV of `HexDigits`.

The MV of `HexIntegerLiteral :: 0X HexDigits` is the MV of `HexDigits`.

The MV of `HexDigits :: HexDigit` is the MV of `HexDigit`.

The MV of `HexDigits :: HexDigits HexDigit` is (the MV of `HexDigits` × 16) plus the MV of `HexDigit`.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is `+0`; otherwise, the rounded value must be the Number value for the MV (as specified in 6.1.6), unless the literal is a `DecimalLiteral` and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a `0` digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a `0` digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an `ExponentPart` and

it is not `0`; or

there is a nonzero digit to its left and there is a nonzero digit, not in the `ExponentPart`, to its right.

11.8.4 String Literals

NOTE 1

A string literal is zero or more Unicode code points enclosed in single or double quotes. Unicode code points may also be represented by an escape sequence. All code points may appear literally in a string literal except for the closing quote code points, U+005C (REVERSE SOLIDUS), U+000D (CARRIAGE RETURN), and U+000A (LINE FEED). Any code points may appear in the form of an escape sequence. String literals evaluate to ECMAScript String values. When

SourceCharacter but not one of *EscapeCharacter* or *LineTerminator*

EscapeCharacter ::

SingleEscapeCharacter

DecimalDigit

x

u

HexEscapeSequence ::

x *HexDigit* *HexDigit*

UnicodeEscapeSequence ::

u *Hex4Digits*

u{ *CodePoint* **}**

Hex4Digits ::

HexDigit *HexDigit* *HexDigit* *HexDigit*

The definition of the nonterminal *HexDigit* is given in 11.8.3. *SourceCharacter* is defined in 10.1.

NOTE 2

<LF> and <CR> cannot appear in a string literal, except as part of a *LineContinuation* to produce the empty code points sequence. The proper way to include either in the String value of a string literal is to use an escape sequence such as \n or \u000A.

11.8.4.1 Static Semantics: StringValue

StringLiteral ::

 " *DoubleStringCharacters*_{opt} "

 ' *SingleStringCharacters*_{opt} '

1. Return the String value whose code units are the SV of this *StringLiteral*.

11.8.4.2 Static Semantics: SV

A string literal stands for a value of the String type. The String value (SV) of the literal is described in terms of code unit values contributed by the various parts of the string literal. As part of this process, some Unicode code points within the string literal are interpreted as having a mathematical value (MV), as described below or in 11.8.3.

The SV of *StringLiteral* :: " " is the empty code unit sequence.

The SV of *StringLiteral* :: ' ' is the empty code unit 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 up to two code units that is the SV of *DoubleStringCharacter*.

The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* *DoubleStringCharacters* is a sequence of up to two code units that is the SV of *DoubleStringCharacter* followed by the code units of the SV of *DoubleStringCharacters* in order.

The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of up to two code units that is the SV

The SV of `Hex4Digits :: HexDigit HexDigit HexDigit HexDigit` is the code unit whose value is (0x1000 times the MV of the first `HexDigit`) plus (0x100 times the MV of the second `HexDigit`) plus (0x10 times the MV of the third `HexDigit`) plus the MV of the fourth `HexDigit`.

The SV of `UnicodeEscapeSequence :: u{ CodePoint }` is the [UTF16Encoding](#) of the MV of `CodePoint`.

11.8.5 Regular Expression Literals

NOTE 1

A regular expression literal is an input element that is converted to a `RegExp` object (see [21.2](#)) each time the literal is evaluated. 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` or calling the `RegExp constructor` as a function (see [21.2.3](#)).

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 source text comprising the `RegularExpressionBody` and the `RegularExpressionFlags` are subsequently parsed again using the more stringent ECMAScript Regular Expression grammar ([21.2.1](#)).

An implementation may extend the ECMAScript Regular Expression grammar defined in [21.2.1](#), but it must not extend the `RegularExpressionBody` and `RegularExpressionFlags` productions defined below or the productions used by these productions.

Syntax

`RegularExpressionLiteral ::`
 `/ RegularExpressionBody / RegularExpressionFlags`

`RegularExpressionBody ::`
 `RegularExpressionFirstChar RegularExpressionChars`

`RegularExpressionChars ::`
 `[empty]`
 `RegularExpressionChars RegularExpressionChar`

`RegularExpressionFirstChar ::`
 `RegularExpressionNonTerminator` but not one of `*` or `\` or `/` or `[`
 `RegularExpressionBackslashSequence`
 `RegularExpressionClass`

`RegularExpressionChar ::`
 `RegularExpressionNonTerminator` but not one of `\` or `/` or `[`
 `RegularExpressionBackslashSequence`
 `RegularExpressionClass`

`RegularExpressionBackslashSequence ::`
 `\ RegularExpressionNonTerminator`

`RegularExpressionNonTerminator ::`
 `SourceCharacter` but not `LineTerminator`

`RegularExpressionClass ::`

TemplateMiddle ::
 } *TemplateCharacters*_{opt} \${

TemplateTail ::
 } *TemplateCharacters*_{opt} `

TemplateCharacters ::
 TemplateCharacter *TemplateCharacters*_{opt}

TemplateCharacter ::
 \$ [lookahead ≠ {}]
 \ *EscapeSequence*
 \ *NotEscapeSequence*
 LineContinuation
 LineTerminatorSequence
 SourceCharacter but not one of ` or \ or \$ or *LineTerminator*

NotEscapeSequence ::
 0 *DecimalDigit*
 DecimalDigit but not 0
 x [lookahead ≠ *HexDigit*]
 x *HexDigit* [lookahead ≠ *HexDigit*]
 u [lookahead ≠ *HexDigit*] [lookahead ≠ {}]
 u *HexDigit* [lookahead ≠ *HexDigit*]
 u *HexDigit* *HexDigit* [lookahead ≠ *HexDigit*]
 u *HexDigit* *HexDigit* *HexDigit* [lookahead ≠ *HexDigit*]
 u { [lookahead ≠ *HexDigit*]
 u { *NotCodePoint* [lookahead ≠ *HexDigit*]
 u { *CodePoint* [lookahead ≠ *HexDigit*] [lookahead ≠ {}]

NotCodePoint ::
 HexDigits but only if MV of *HexDigits* > 0x10FFFF

CodePoint ::
 HexDigits but only if MV of *HexDigits* ≤ 0x10FFFF

A conforming implementation must not use the extended definition of *EscapeSequence* described in [B.1.2](#) when parsing a *TemplateCharacter*.

NOTE

TemplateSubstitutionTail is used by the *InputElementTemplateTail* alternative lexical goal.

11.8.6.1 Static Semantics: TV and TRV

A template literal component is interpreted as a sequence of Unicode code points. The Template Value (TV) of a literal component is described in terms of code unit values (SV, [11.8.4](#)) contributed by the various parts of the template literal component. As part of this process, some Unicode code points within the template component are interpreted as having a mathematical value (MV, [11.8.3](#)). In determining a TV, escape sequences are replaced by the UTF-16 code unit(s) of the Unicode code point represented by the escape sequence. The Template Raw Value (TRV) is similar to a Template Value

The TRV of `NotEscapeSequence :: x HexDigit [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0078 (LATIN SMALL LETTER X) followed by the code units of the TRV of `HexDigit`.

The TRV of `NotEscapeSequence :: u [lookahead ≠ HexDigit] [lookahead ≠ {}]` is the code unit 0x0075 (LATIN SMALL LETTER U).

The TRV of `NotEscapeSequence :: u HexDigit [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code units of the TRV of `HexDigit`.

The TRV of `NotEscapeSequence :: u HexDigit HexDigit [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code units of the TRV of the first `HexDigit` followed by the code units of the TRV of the second `HexDigit`.

The TRV of `NotEscapeSequence :: u HexDigit HexDigit HexDigit [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code units of the TRV of the first `HexDigit` followed by the code units of the TRV of the second `HexDigit` followed by the code units of the TRV of the third `HexDigit`.

The TRV of `NotEscapeSequence :: u { [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code unit 0x007B (LEFT CURLY BRACKET).

The TRV of `NotEscapeSequence :: u { NotCodePoint [lookahead ≠ HexDigit]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code unit 0x007B (LEFT CURLY BRACKET) followed by the code units of the TRV of `NotCodePoint`.

The TRV of `NotEscapeSequence :: u { CodePoint [lookahead ≠ HexDigit] [lookahead ≠ {}]` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code unit 0x007B (LEFT CURLY BRACKET) followed by the code units of the TRV of `CodePoint`.

The TRV of `DecimalDigit :: one of 0 1 2 3 4 5 6 7 8 9` is the SV of the `SourceCharacter` that is that single code point.

The TRV of `CharacterEscapeSequence :: SingleEscapeCharacter` is the TRV of the `SingleEscapeCharacter`.

The TRV of `CharacterEscapeSequence :: NonEscapeCharacter` is the SV of the `NonEscapeCharacter`.

The TRV of `SingleEscapeCharacter :: one of ' " \ b f n r t v` is the SV of the `SourceCharacter` that is that single code point.

The TRV of `HexEscapeSequence :: x HexDigit HexDigit` is the sequence consisting of the code unit 0x0078 (LATIN SMALL LETTER X) followed by TRV of the first `HexDigit` followed by the TRV of the second `HexDigit`.

The TRV of `UnicodeEscapeSequence :: u Hex4Digits` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by TRV of `Hex4Digits`.

The TRV of `UnicodeEscapeSequence :: u{ CodePoint }` is the sequence consisting of the code unit 0x0075 (LATIN SMALL LETTER U) followed by the code unit 0x007B (LEFT CURLY BRACKET) followed by TRV of `CodePoint` followed by the code unit 0x007D (RIGHT CURLY BRACKET).

The TRV of `Hex4Digits :: HexDigit HexDigit HexDigit HexDigit` is the sequence consisting of the TRV of the first `HexDigit` followed by the TRV of the second `HexDigit` followed by the TRV of the third `HexDigit` followed by the TRV of the fourth `HexDigit`.

The TRV of `HexDigits :: HexDigit` is the TRV of `HexDigit`.

The TRV of `HexDigits :: HexDigits HexDigit` is the sequence consisting of TRV of `HexDigits` followed by TRV of `HexDigit`.

The TRV of a `HexDigit` is the SV of the `SourceCharacter` that is that `HexDigit`.

The TRV of `LineContinuation :: \ LineTerminatorSequence` is the sequence consisting of the code unit 0x005C (REVERSE SOLIDUS) followed by the code units of TRV of `LineTerminatorSequence`.

The TRV of `LineTerminatorSequence :: <LF>` is the code unit 0x000A (LINE FEED).

The TRV of `LineTerminatorSequence :: <CR>` is the code unit 0x000A (LINE FEED).

The TRV of `LineTerminatorSequence :: <LS>` is the code unit 0x2028 (LINE SEPARATOR).

The TRV of `LineTerminatorSequence :: <PS>` is the code unit 0x2029 (PARAGRAPH SEPARATOR).


```

LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] --
ContinueStatement[Yield, Await] :
  continue ;
  continue [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;

BreakStatement[Yield, Await] :
  break ;
  break [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;

ReturnStatement[Yield, Await] :
  return ;
  return [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;

ThrowStatement[Yield, Await] :
  throw [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;

ArrowFunction[In, Yield, Await] :
  ArrowParameters[?Yield, ?Await] [no LineTerminator here] => ConciseBody[?In]

YieldExpression[In, Await] :
  yield [no LineTerminator here] AssignmentExpression[?In, +Yield, ?Await]
  yield [no LineTerminator here] * AssignmentExpression[?In, +Yield, ?Await]

```

The practical effect of these restricted productions is as follows:

When a `++` or `--` token is encountered where the parser would treat it as a postfix operator, and at least one *LineTerminator* occurred between the preceding token and the `++` or `--` token, then a semicolon is automatically inserted before the `++` or `--` token.

When a `continue`, `break`, `return`, `throw`, or `yield` token is encountered and a *LineTerminator* is encountered before the next token, a semicolon is automatically inserted after the `continue`, `break`, `return`, `throw`, or `yield` token.

The resulting practical advice to ECMAScript programmers is:

A postfix `++` or `--` operator should appear on the same line as its operand.

An *Expression* in a `return` or `throw` statement or an *AssignmentExpression* in a `yield` expression should start on the same line as the `return`, `throw`, or `yield` token.

A *LabelIdentifier* in a `break` or `continue` statement should be on the same line as the `break` or `continue` token.

11.9.2 Examples of Automatic Semicolon Insertion

The source

```
{ 1 2 } 3
```

is not a valid sentence in the ECMAScript grammar, even with the automatic semicolon insertion rules. In contrast, the


```
else c = d
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion before the **else** token, even though no production of the grammar applies at that point, because an automatically inserted semicolon would then be parsed as an empty statement.

The source

```
a = b + c  
(d + e).print()
```

is *not* transformed by automatic semicolon insertion, because the parenthesized expression that begins the second line can be interpreted as an argument list for a function call:

```
a = b + c(d + e).print()
```

In the circumstance that an assignment statement must begin with a left parenthesis, it is a good idea for the programmer to provide an explicit semicolon at the end of the preceding statement rather than to rely on automatic semicolon insertion.

12 ECMAScript Language: Expressions

12.1 Identifiers

Syntax

IdentifierReference [Yield, Await] :

Identifier
[~Yield] **yield**
[~Await] **await**

BindingIdentifier [Yield, Await] :

Identifier
yield
await

LabelIdentifier [Yield, Await] :

Identifier
[~Yield] **yield**
[~Await] **await**

Identifier :
IdentifierName but not *ReservedWord*

NOTE

yield and **await** are permitted as *BindingIdentifier* in the grammar, and prohibited with *static semantics* below, to prohibit automatic semicolon insertion in cases such as

```
let
```


12.1.2 Static Semantics: BoundNames

BindingIdentifier : *Identifier*

1. Return a new `List` containing the `StringValue` of *Identifier*.

BindingIdentifier : **yield**

1. Return a new `List` containing "**yield**".

BindingIdentifier : **await**

1. Return a new `List` containing "**await**".

12.1.3 Static Semantics: AssignmentTargetType

IdentifierReference : *Identifier*

1. If this *IdentifierReference* is contained in `strict mode code` and `StringValue` of *Identifier* is "**eval**" or "**arguments**", return `strict`.
2. Return `simple`.

IdentifierReference : **yield**

1. Return `simple`.

IdentifierReference : **await**

1. Return `simple`.

12.1.4 Static Semantics: StringValue

IdentifierReference : **yield**

BindingIdentifier : **yield**

LabelIdentifier : **yield**

1. Return "**yield**".

IdentifierReference : **await**

BindingIdentifier : **await**

LabelIdentifier : **await**

1. Return "**await**".

Identifier : *IdentifierName* but not *ReservedWord*

1. Return the `StringValue` of *IdentifierName*.

12.1.5 Runtime Semantics: BindingInitialization

With parameters *value* and *environment*.

NOTE

undefined is passed for *environment* to indicate that a `PutValue` operation should be used to assign the initialization value. This is the case for `var` statements and formal parameter lists of some non-strict functions (See 9.2.15). In those


```

PrimaryExpression[Yield, Await] :
  this
  IdentifierReference[?Yield, ?Await]
  Literal
  ArrayLiteral[?Yield, ?Await]
  ObjectLiteral[?Yield, ?Await]
  FunctionExpression
  ClassExpression[?Yield, ?Await]
  GeneratorExpression
  AsyncFunctionExpression
  AsyncGeneratorExpression
  RegularExpressionLiteral
  TemplateLiteral[?Yield, ?Await, ~Tagged]
  CoverParenthesizedExpressionAndArrowParameterList[?Yield, ?Await]

```

```

CoverParenthesizedExpressionAndArrowParameterList[Yield, Await] :
  ( Expression[+In, ?Yield, ?Await] )
  ( Expression[+In, ?Yield, ?Await] , )
  ( )
  ( ... BindingIdentifier[?Yield, ?Await] )
  ( ... BindingPattern[?Yield, ?Await] )
  ( Expression[+In, ?Yield, ?Await] , ... BindingIdentifier[?Yield, ?Await] )
  ( Expression[+In, ?Yield, ?Await] , ... BindingPattern[?Yield, ?Await] )

```

Supplemental Syntax

When processing an instance of the production

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

the interpretation of *CoverParenthesizedExpressionAndArrowParameterList* is refined using the following grammar:

```

ParenthesizedExpression[Yield, Await] :
  ( Expression[+In, ?Yield, ?Await] )

```

12.2.1 Semantics

12.2.1.1 Static Semantics: CoveredParenthesizedExpression

CoverParenthesizedExpressionAndArrowParameterList : (*Expression*)

1. Return the *ParenthesizedExpression* that is **covered** by *CoverParenthesizedExpressionAndArrowParameterList*.

12.2.1.2 Static Semantics: HasName

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoveredParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. If *IsFunctionDefinition* of *expr* is **false**, return **false**.
3. Return *HasName* of *expr*.

AsyncGeneratorExpression

RegularExpressionLiteral

TemplateLiteral

1. Return invalid.

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoverParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return *AssignmentTargetType* of *expr*.

12.2.2 The **this** Keyword

12.2.2.1 Runtime Semantics: Evaluation

PrimaryExpression : **this**

1. Return `?ResolveThisBinding()`.

12.2.3 Identifier Reference

See [12.1](#) for *IdentifierReference*.

12.2.4 Literals

Syntax

Literal :

NullLiteral

BooleanLiteral

NumericLiteral

StringLiteral

12.2.4.1 Runtime Semantics: Evaluation

Literal : *NullLiteral*

1. Return **null**.

Literal : *BooleanLiteral*

1. If *BooleanLiteral* is the token **false**, return **false**.
2. If *BooleanLiteral* is the token **true**, return **true**.

Literal : *NumericLiteral*

1. Return the number whose value is MV of *NumericLiteral* as defined in [11.8.3](#).

Literal : *StringLiteral*

1. Return the *StringValue* of *StringLiteral* as defined in [11.8.4.1](#).

3. Let *initValue* be ? GetValue(*initResult*).
4. Let *created* be CreateDataProperty(*array*, ToString(ToUint32(*nextIndex* + *padding*)), *initValue*).
5. **Assert:** *created* is true.
6. Return *nextIndex* + *padding* + 1.

ElementList : Elision SpreadElement

1. Let *padding* be the ElisionWidth of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing ArrayAccumulation for *SpreadElement* with arguments *array* and *nextIndex* + *padding*.

ElementList : ElementList , Elision AssignmentExpression

1. Let *postIndex* be the result of performing ArrayAccumulation for *ElementList* with arguments *array* and *nextIndex*.
2. **ReturnIfAbrupt**(*postIndex*).
3. Let *padding* be the ElisionWidth of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Let *initResult* be the result of evaluating *AssignmentExpression*.
5. Let *initValue* be ? GetValue(*initResult*).
6. Let *created* be CreateDataProperty(*array*, ToString(ToUint32(*postIndex* + *padding*)), *initValue*).
7. **Assert:** *created* is true.
8. Return *postIndex* + *padding* + 1.

ElementList : ElementList , Elision SpreadElement

1. Let *postIndex* be the result of performing ArrayAccumulation for *ElementList* with arguments *array* and *nextIndex*.
2. **ReturnIfAbrupt**(*postIndex*).
3. Let *padding* be the ElisionWidth of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Return the result of performing ArrayAccumulation for *SpreadElement* with arguments *array* and *postIndex* + *padding*.

SpreadElement : . . . AssignmentExpression

1. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
2. Let *spreadObj* be ? GetValue(*spreadRef*).
3. Let *iteratorRecord* be ? GetIterator(*spreadObj*).
4. Repeat,
 - a. Let *next* be ? IteratorStep(*iteratorRecord*).
 - b. If *next* is false, return *nextIndex*.
 - c. Let *nextValue* be ? IteratorValue(*next*).
 - d. Let *status* be CreateDataProperty(*array*, ToString(ToUint32(*nextIndex*)), *nextValue*).
 - e. **Assert:** *status* is true.
 - f. Increase *nextIndex* by 1.

NOTE

CreateDataProperty is used to ensure that own properties are defined for the array even if the standard built-in Array prototype object has been modified in a manner that would preclude the creation of new own properties using [[Set]].

12.2.5.3 Runtime Semantics: Evaluation

ArrayLiteral : [Elision]


```

PropertyName [Yield, Await] :
  LiteralPropertyName
  ComputedPropertyName [?Yield, ?Await]

LiteralPropertyName :
  IdentifierName
  StringLiteral
  NumericLiteral

ComputedPropertyName [Yield, Await] :
  [ AssignmentExpression [+In, ?Yield, ?Await] ]

CoverInitializedName [Yield, Await] :
  IdentifierReference [?Yield, ?Await] Initializer [+In, ?Yield, ?Await]

Initializer [In, Yield, Await] :
  = AssignmentExpression [?In, ?Yield, ?Await]

```

NOTE 2

MethodDefinition is defined in 14.3.

NOTE 3

In certain contexts, *ObjectLiteral* is used as a cover grammar for a more restricted secondary grammar. The *CoverInitializedName* production is necessary to fully cover these secondary grammars. However, use of this production results in an early Syntax Error in normal contexts where an actual *ObjectLiteral* is expected.

12.2.6.1 Static Semantics: Early Errors

PropertyDefinition : *MethodDefinition*

It is a Syntax Error if HasDirectSuper of *MethodDefinition* is **true**.

In addition to describing an actual object initializer the *ObjectLiteral* productions are also used as a cover grammar for *ObjectAssignmentPattern* and may be recognized as part of a *CoverParenthesizedExpressionAndArrowParameterList*. When *ObjectLiteral* appears in a context where *ObjectAssignmentPattern* is required the following Early Error rules are **not applied**. In addition, they are not applied when initially parsing a *CoverParenthesizedExpressionAndArrowParameterList* or *CoverCallExpressionAndAsyncArrowHead*.

PropertyDefinition : *CoverInitializedName*

Always throw a Syntax Error if code matches this production.

NOTE

This production exists so that *ObjectLiteral* can serve as a cover grammar for *ObjectAssignmentPattern*. It cannot occur in an actual object initializer.

12.2.6.2 Static Semantics: ComputedPropertyContains

With parameter *symbol*.

PropertyName : *LiteralPropertyName*

LiteralPropertyName : *NumericLiteral*

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.
2. Return ! `ToString(nbr)`.

ComputedPropertyName : [*AssignmentExpression*]

1. Return empty.

12.2.6.6 Static Semantics: **PropertyNameList**

PropertyDefinitionList : *PropertyDefinition*

1. If *PropName* of *PropertyDefinition* is `empty`, return a new empty *List*.
2. Return a new *List* containing *PropName* of *PropertyDefinition*.

PropertyDefinitionList : *PropertyDefinitionList* , *PropertyDefinition*

1. Let *list* be *PropertyNameList* of *PropertyDefinitionList*.
2. If *PropName* of *PropertyDefinition* is `empty`, return *list*.
3. Append *PropName* of *PropertyDefinition* to the end of *list*.
4. Return *list*.

12.2.6.7 Runtime Semantics: **Evaluation**

ObjectLiteral : { }

1. Return `ObjectCreate(%ObjectPrototype%)`.

ObjectLiteral :

{ *PropertyDefinitionList* }
{ *PropertyDefinitionList* , }

1. Let *obj* be `ObjectCreate(%ObjectPrototype%)`.
2. Perform ? *PropertyDefinitionEvaluation* of *PropertyDefinitionList* with arguments *obj* and `true`.
3. Return *obj*.

LiteralPropertyName : *IdentifierName*

1. Return *StringValue* of *IdentifierName*.

LiteralPropertyName : *StringLiteral*

1. Return the String value whose code units are the *SV* of the *StringLiteral*.

LiteralPropertyName : *NumericLiteral*

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.
2. Return ! `ToString(nbr)`.

ComputedPropertyName : [*AssignmentExpression*]

1. Let *exprValue* be the result of evaluating *AssignmentExpression*.
2. Let *propName* be ? `GetValue(exprValue)`.
3. Return ? `ToPropertyKey(propName)`.

See 14.7 for *PrimaryExpression* : *AsyncFunctionExpression* .

See 14.5 for *PrimaryExpression* : *AsyncGeneratorExpression* .

12.2.8 Regular Expression Literals

Syntax

See 11.8.5.

12.2.8.1 Static Semantics: Early Errors

PrimaryExpression : *RegularExpressionLiteral*

It is a Syntax Error if BodyText of *RegularExpressionLiteral* cannot be recognized using the goal symbol *Pattern* of the ECMAScript RegExp grammar specified in 21.2.1.

It is a Syntax Error if FlagText of *RegularExpressionLiteral* contains any code points other than "g", "i", "m", "s", "u", or "y", or if it contains the same code point more than once.

12.2.8.2 Runtime Semantics: Evaluation

PrimaryExpression : *RegularExpressionLiteral*

1. Let *pattern* be the String value consisting of the *UTF16Encoding* of each code point of BodyText of *RegularExpressionLiteral*.
2. Let *flags* be the String value consisting of the *UTF16Encoding* of each code point of FlagText of *RegularExpressionLiteral*.
3. Return *RegExpCreate*(*pattern*, *flags*).

12.2.9 Template Literals

Syntax

TemplateLiteral [Yield, Await, Tagged] :

NoSubstitutionTemplate

SubstitutionTemplate [?Yield, ?Await, ?Tagged]

SubstitutionTemplate [Yield, Await, Tagged] :

TemplateHead *Expression* [+In, ?Yield, ?Await] *TemplateSpans* [?Yield, ?Await, ?Tagged]

TemplateSpans [Yield, Await, Tagged] :

TemplateTail

TemplateMiddleList [?Yield, ?Await, ?Tagged] *TemplateTail*

TemplateMiddleList [Yield, Await, Tagged] :

TemplateMiddle *Expression* [+In, ?Yield, ?Await]

TemplateMiddleList [?Yield, ?Await, ?Tagged] *TemplateMiddle* *Expression* [+In, ?Yield, ?Await]

12.2.9.1 Static Semantics: Early Errors

TemplateLiteral : *NoSubstitutionTemplate*

3. Else,
 - a. Let *tail* be the TRV of *TemplateTail*.
4. Return a *List* containing the elements, in order, of *middle* followed by *tail*.

TemplateMiddleList : *TemplateMiddle Expression*

1. If *raw* is **false**, then
 - a. Let *string* be the TV of *TemplateMiddle*.
2. Else,
 - a. Let *string* be the TRV of *TemplateMiddle*.
3. Return a *List* containing the single element, *string*.

TemplateMiddleList : *TemplateMiddleList TemplateMiddle Expression*

1. Let *front* be *TemplateStrings* of *TemplateMiddleList* with argument *raw*.
2. If *raw* is **false**, then
 - a. Let *last* be the TV of *TemplateMiddle*.
3. Else,
 - a. Let *last* be the TRV of *TemplateMiddle*.
4. Append *last* as the last element of the *List* *front*.
5. Return *front*.

12.2.9.3 Runtime Semantics: ArgumentListEvaluation

TemplateLiteral : *NoSubstitutionTemplate*

1. Let *templateLiteral* be this *TemplateLiteral*.
2. Let *siteObj* be *GetTemplateObject*(*templateLiteral*).
3. Return a *List* containing the one element which is *siteObj*.

SubstitutionTemplate : *TemplateHead Expression TemplateSpans*

1. Let *templateLiteral* be this *TemplateLiteral*.
2. Let *siteObj* be *GetTemplateObject*(*templateLiteral*).
3. Let *firstSubRef* be the result of evaluating *Expression*.
4. Let *firstSub* be ? *GetValue*(*firstSubRef*).
5. Let *restSub* be *SubstitutionEvaluation* of *TemplateSpans*.
6. *ReturnIfAbrupt*(*restSub*).
7. *Assert*: *restSub* is a *List*.
8. Return a *List* whose first element is *siteObj*, whose second elements is *firstSub*, and whose subsequent elements are the elements of *restSub*, in order. *restSub* may contain no elements.

12.2.9.4 Runtime Semantics: GetTemplateObject (*templateLiteral*)

The abstract operation *GetTemplateObject* is called with a *Parse Node*, *templateLiteral*, as an argument. It performs the following steps:

1. Let *rawStrings* be *TemplateStrings* of *templateLiteral* with argument **true**.
2. Let *realm* be the current Realm Record.
3. Let *templateRegistry* be *realm*.[[*TemplateMap*]].
4. For each element *e* of *templateRegistry*, do
 - a. If *e*.[[Site]] is the same *Parse Node* as *templateLiteral*, then

TemplateMiddleList : *TemplateMiddleList* *TemplateMiddle Expression*

1. Let *preceding* be the result of SubstitutionEvaluation of *TemplateMiddleList*.
2. *ReturnIfAbrupt*(*preceding*).
3. Let *nextRef* be the result of evaluating *Expression*.
4. Let *next* be ? *GetValue*(*nextRef*).
5. Append *next* as the last element of the List *preceding*.
6. Return *preceding*.

12.2.9.6 Runtime Semantics: Evaluation

TemplateLiteral : *NoSubstitutionTemplate*

1. Return the String value whose code units are the elements of the TV of *NoSubstitutionTemplate* as defined in [11.8.6](#).

SubstitutionTemplate : *TemplateHead* *Expression* *TemplateSpans*

1. Let *head* be the TV of *TemplateHead* as defined in [11.8.6](#).
2. Let *subRef* be the result of evaluating *Expression*.
3. Let *sub* be ? *GetValue*(*subRef*).
4. Let *middle* be ? *ToString*(*sub*).
5. Let *tail* be the result of evaluating *TemplateSpans*.
6. *ReturnIfAbrupt*(*tail*).
7. Return the string-concatenation of *head*, *middle*, and *tail*.

NOTE 1

The string conversion semantics applied to the *Expression* value are like `String.prototype.concat` rather than the `+` operator.

TemplateSpans : *TemplateTail*

1. Let *tail* be the TV of *TemplateTail* as defined in [11.8.6](#).
2. Return the String value consisting of the code units of *tail*.

TemplateSpans : *TemplateMiddleList* *TemplateTail*

1. Let *head* be the result of evaluating *TemplateMiddleList*.
2. *ReturnIfAbrupt*(*head*).
3. Let *tail* be the TV of *TemplateTail* as defined in [11.8.6](#).
4. Return the string-concatenation of *head* and *tail*.

TemplateMiddleList : *TemplateMiddle Expression*

1. Let *head* be the TV of *TemplateMiddle* as defined in [11.8.6](#).
2. Let *subRef* be the result of evaluating *Expression*.
3. Let *sub* be ? *GetValue*(*subRef*).
4. Let *middle* be ? *ToString*(*sub*).
5. Return the sequence of code units consisting of the code units of *head* followed by the elements of *middle*.

NOTE 2

The string conversion semantics applied to the *Expression* value are like `String.prototype.concat` rather than

2. Return the result of performing NamedEvaluation for *Expression* with argument *name*.

12.2.10.5 Runtime Semantics: Evaluation

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoveredParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the result of evaluating *expr*.

ParenthesizedExpression : (*Expression*)

1. Return the result of evaluating *Expression*. This may be of type **Reference**.

NOTE

This algorithm does not apply **GetValue** to the result of evaluating *Expression*. The principal motivation for this is so that operators such as **delete** and **typeof** may be applied to parenthesized expressions.

12.3 Left-Hand-Side Expressions

Syntax

```
MemberExpression [Yield, Await] :
  PrimaryExpression [?Yield, ?Await]
  MemberExpression [?Yield, ?Await] [ Expression [+In, ?Yield, ?Await] ]
  MemberExpression [?Yield, ?Await] . IdentifierName
  MemberExpression [?Yield, ?Await] TemplateLiteral [?Yield, ?Await, +Tagged]
  SuperProperty [?Yield, ?Await]
  MetaProperty
  new MemberExpression [?Yield, ?Await] Arguments [?Yield, ?Await]
```

```
SuperProperty [Yield, Await] :
  super [ Expression [+In, ?Yield, ?Await] ]
  super . IdentifierName
```

```
MetaProperty :
  NewTarget
```

```
NewTarget :
  new . target
```

```
NewExpression [Yield, Await] :
  MemberExpression [?Yield, ?Await]
  new NewExpression [?Yield, ?Await]
```

```
CallExpression [Yield, Await] :
  CoverCallExpressionAndAsyncArrowHead [?Yield, ?Await]
  SuperCall [?Yield, ?Await]
  CallExpression [?Yield, ?Await] Arguments [?Yield, ?Await]
```


SuperProperty : **super** . *IdentifierName*

1. If *symbol* is the *ReservedWord* **super**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and *StringValue* of *symbol* is the same value as the *StringValue* of *IdentifierName*, return **true**.
4. Return **false**.

CallExpression : *CallExpression* . *IdentifierName*

1. If *CallExpression* Contains *symbol* is **true**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and *StringValue* of *symbol* is the same value as the *StringValue* of *IdentifierName*, return **true**.
4. Return **false**.

12.3.1.3 Static Semantics: IsFunctionDefinition

MemberExpression :

MemberExpression [*Expression*]
MemberExpression . *IdentifierName*
MemberExpression *TemplateLiteral*
SuperProperty
MetaProperty
new *MemberExpression Arguments*

NewExpression :

new *NewExpression*

LeftHandSideExpression :

CallExpression

1. Return **false**.

12.3.1.4 Static Semantics: IsDestructuring

MemberExpression : *PrimaryExpression*

1. If *PrimaryExpression* is either an *ObjectLiteral* or an *ArrayLiteral*, return **true**.
2. Return **false**.

MemberExpression :

MemberExpression [*Expression*]
MemberExpression . *IdentifierName*
MemberExpression *TemplateLiteral*
SuperProperty
MetaProperty
new *MemberExpression Arguments*

NewExpression :

new *NewExpression*

1. Return invalid.

12.3.2 Property Accessors

NOTE

Properties are accessed by name, using either the dot notation:

MemberExpression . IdentifierName

CallExpression . IdentifierName

or the bracket notation:

MemberExpression [Expression]

CallExpression [Expression]

The dot notation is explained by the following syntactic conversion:

MemberExpression . IdentifierName

is identical in its behaviour to

MemberExpression [<identifier-name-string>]

and similarly

CallExpression . IdentifierName

is identical in its behaviour to

CallExpression [<identifier-name-string>]

where *<identifier-name-string>* is the result of evaluating *StringValue* of *IdentifierName*.

12.3.2.1 Runtime Semantics: Evaluation

MemberExpression : MemberExpression [Expression]

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be ? *GetValue(baseReference)*.
3. Let *propertyNameReference* be the result of evaluating *Expression*.
4. Let *propertyNameValue* be ? *GetValue(propertyNameReference)*.
5. Let *bv* be ? *RequireObjectCoercible(baseValue)*.
6. Let *propertyKey* be ? *ToPropertyKey(propertyNameValue)*.
7. If the code matched by this *MemberExpression* is *strict mode code*, let *strict* be **true**, else let *strict* be **false**.
8. Return a value of type *Reference* whose base value component is *bv*, whose referenced name component is *propertyKey*, and whose strict reference flag is *strict*.

MemberExpression : MemberExpression . IdentifierName

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be ? *GetValue(baseReference)*.
3. Let *bv* be ? *RequireObjectCoercible(baseValue)*.
4. Let *propertyNameString* be *StringValue* of *IdentifierName*.

6. If `Type(ref)` is `Reference` and `IsPropertyReference(ref)` is `false` and `GetReferencedName(ref)` is "`eval`", then
 - a. If `SameValue(func, %eval%)` is `true`, then
 - i. Let `argList` be `? ArgumentListEvaluation` of `arguments`.
 - ii. If `argList` has no elements, return `undefined`.
 - iii. Let `evalText` be the first element of `argList`.
 - iv. If the source code matching this `CallExpression` is `strict mode code`, let `strictCaller` be `true`. Otherwise let `strictCaller` be `false`.
 - v. Let `evalRealm` be the current Realm Record.
 - vi. Perform `? HostEnsureCanCompileStrings(evalRealm, evalRealm)`.
 - vii. Return `? PerformEval(evalText, evalRealm, strictCaller, true)`.
7. Let `thisCall` be this `CallExpression`.
8. Let `tailCall` be `IsInTailPosition(thisCall)`.
9. Return `? EvaluateCall(func, ref, arguments, tailCall)`.

A `CallExpression` evaluation that executes step 6.a.vii is a direct eval.

CallExpression : CallExpression Arguments

1. Let `ref` be the result of evaluating `CallExpression`.
2. Let `func` be `? GetValue(ref)`.
3. Let `thisCall` be this `CallExpression`.
4. Let `tailCall` be `IsInTailPosition(thisCall)`.
5. Return `? EvaluateCall(func, ref, Arguments, tailCall)`.

12.3.4.2 Runtime Semantics: EvaluateCall (`func, ref, arguments, tailPosition`)

The abstract operation `EvaluateCall` takes as arguments a value `func`, a value `ref`, a Parse Node `arguments`, and a Boolean argument `tailPosition`. It performs the following steps:

1. If `Type(ref)` is `Reference`, then
 - a. If `IsPropertyReference(ref)` is `true`, then
 - i. Let `thisValue` be `GetThisValue(ref)`.
 - b. Else the base of `ref` is an `Environment Record`,
 - i. Let `refEnv` be `GetBase(ref)`.
 - ii. Let `thisValue` be `refEnv.WithBaseObject()`.
2. Else `Type(ref)` is not `Reference`,
 - a. Let `thisValue` be `undefined`.
3. Let `argList` be `ArgumentListEvaluation` of `arguments`.
4. `ReturnIfAbrupt(argList)`.
5. If `Type(func)` is not `Object`, throw a `TypeError` exception.
6. If `IsCallable(func)` is `false`, throw a `TypeError` exception.
7. If `tailPosition` is `true`, perform `PrepareForTailCall()`.
8. Let `result` be `Call(func, thisValue, argList)`.
9. **Assert:** If `tailPosition` is `true`, the above call will not return here, but instead evaluation will continue as if the following return has already occurred.
10. **Assert:** If `result` is not an abrupt completion, then `Type(result)` is an ECMA Script language type.
11. Return `result`.

12.3.5 The `super` Keyword

5. Return a value of type **Reference** that is a **Super Reference** whose base value component is *bv*, whose referenced name component is *propertyKey*, whose thisValue component is *actualThis*, and whose strict reference flag is *strict*.

12.3.6 Argument Lists

NOTE

The evaluation of an argument list produces a **List** of values.

12.3.6.1 Runtime Semantics: ArgumentListEvaluation

Arguments : ()

1. Return a new empty **List**.

ArgumentList : *AssignmentExpression*

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be ? **GetValue**(*ref*).
3. Return a **List** whose sole item is *arg*.

ArgumentList : . . . *AssignmentExpression*

1. Let *list* be a new empty **List**.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadObj* be ? **GetValue**(*spreadRef*).
4. Let *iteratorRecord* be ? **GetIterator**(*spreadObj*).
5. Repeat,
 - a. Let *next* be ? **IteratorStep**(*iteratorRecord*).
 - b. If *next* is **false**, return *list*.
 - c. Let *nextArg* be ? **IteratorValue**(*next*).
 - d. Append *nextArg* as the last element of *list*.

ArgumentList : *ArgumentList* , *AssignmentExpression*

1. Let *precedingArgs* be **ArgumentListEvaluation** of *ArgumentList*.
2. **ReturnIfAbrupt**(*precedingArgs*).
3. Let *ref* be the result of evaluating *AssignmentExpression*.
4. Let *arg* be ? **GetValue**(*ref*).
5. Append *arg* to the end of *precedingArgs*.
6. Return *precedingArgs*.

ArgumentList : *ArgumentList* , . . . *AssignmentExpression*

1. Let *precedingArgs* be **ArgumentListEvaluation** of *ArgumentList*.
2. **ReturnIfAbrupt**(*precedingArgs*).
3. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
4. Let *iteratorRecord* be ? **GetIterator**(? **GetValue**(*spreadRef*)).
5. Repeat,
 - a. Let *next* be ? **IteratorStep**(*iteratorRecord*).
 - b. If *next* is **false**, return *precedingArgs*.
 - c. Let *nextArg* be ? **IteratorValue**(*next*).

UpdateExpression :

LeftHandSideExpression **++**
LeftHandSideExpression **--**

It is an early [Reference](#) Error if *AssignmentTargetType* of *LeftHandSideExpression* is invalid.

It is an early Syntax Error if *AssignmentTargetType* of *LeftHandSideExpression* is strict.

UpdateExpression :

++ *UnaryExpression*
-- *UnaryExpression*

It is an early [Reference](#) Error if *AssignmentTargetType* of *UnaryExpression* is invalid.

It is an early Syntax Error if *AssignmentTargetType* of *UnaryExpression* is strict.

12.4.2 Static Semantics: IsFunctionDefinition

UpdateExpression :

LeftHandSideExpression **++**
LeftHandSideExpression **--**
++ *UnaryExpression*
-- *UnaryExpression*

1. Return **false**.

12.4.3 Static Semantics: AssignmentTargetType

UpdateExpression :

LeftHandSideExpression **++**
LeftHandSideExpression **--**
++ *UnaryExpression*
-- *UnaryExpression*

1. Return **invalid**.

12.4.4 Postfix Increment Operator

12.4.4.1 Runtime Semantics: Evaluation

UpdateExpression : *LeftHandSideExpression* **++**

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be `? ToNumber(? GetValue(lhs))`.
3. Let *newValue* be the result of adding the value 1 to *oldValue*, using the same rules as for the **+** operator (see [12.8.5](#)).
4. Perform `? PutValue(lhs, newValue)`.
5. Return *oldValue*.

12.4.5 Postfix Decrement Operator

! UnaryExpression [?Yield, ?Await]
[+Await] **AwaitExpression** [?Yield]

12.5.1 Static Semantics: IsFunctionDefinition

UnaryExpression :
delete *UnaryExpression*
void *UnaryExpression*
typeof *UnaryExpression*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*
AwaitExpression

1. Return **false**.

12.5.2 Static Semantics: AssignmentTargetType

UnaryExpression :
delete *UnaryExpression*
void *UnaryExpression*
typeof *UnaryExpression*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*
AwaitExpression

1. Return **invalid**.

12.5.3 The **delete** Operator

12.5.3.1 Static Semantics: Early Errors

UnaryExpression : **delete** *UnaryExpression*

It is a Syntax Error if the *UnaryExpression* is contained in **strict mode code** and the derived *UnaryExpression* is
PrimaryExpression : *IdentifierReference* .

It is a Syntax Error if the derived *UnaryExpression* is

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

and *CoverParenthesizedExpressionAndArrowParameterList* ultimately derives a phrase that, if used in place of
UnaryExpression, would produce a Syntax Error according to these rules. This rule is recursively applied.

NOTE

The last rule means that expressions such as **delete** (((**foo**))) produce early errors because of recursive application of the first rule.

4. Return a String according to [Table 35](#).

Table 35: `typeof` Operator Results

Type of <code>val</code>	Result
Undefined	" <code>undefined</code> "
Null	" <code>object</code> "
Boolean	" <code>boolean</code> "
Number	" <code>number</code> "
String	" <code>string</code> "
Symbol	" <code>symbol</code> "
Object (ordinary and does not implement <code>[[Call]]</code>)	" <code>object</code> "
Object (standard exotic and does not implement <code>[[Call]]</code>)	" <code>object</code> "
Object (implements <code>[[Call]]</code>)	" <code>function</code> "
Object (non-standard exotic and does not implement <code>[[Call]]</code>)	Implementation-defined. Must not be " <code>undefined</code> ", " <code>boolean</code> ", " <code>function</code> ", " <code>number</code> ", " <code>symbol</code> ", or " <code>string</code> ".

NOTE

Implementations are discouraged from defining new `typeof` result values for non-standard exotic objects. If possible "`object`" should be used for such objects.

12.5.6 Unary + Operator

NOTE

The unary + operator converts its operand to Number type.

12.5.6.1 Runtime Semantics: Evaluation

UnaryExpression : + *UnaryExpression*

1. Let `expr` be the result of evaluating *UnaryExpression*.
2. Return ? `ToNumber(? GetValue(expr))`.

12.5.7 Unary - Operator

NOTE

The unary - operator converts its operand to Number type and then negates it. Negating +0 produces -0, and negating -0 produces +0.

1. Return invalid.

12.6.3 Runtime Semantics: Evaluation

ExponentiationExpression : *UpdateExpression* ****** *ExponentiationExpression*

1. Let *left* be the result of evaluating *UpdateExpression*.
2. Let *leftValue* be ? *GetValue(left)*.
3. Let *right* be the result of evaluating *ExponentiationExpression*.
4. Let *rightValue* be ? *GetValue(right)*.
5. Let *base* be ? *ToNumber(leftValue)*.
6. Let *exponent* be ? *ToNumber(rightValue)*.
7. Return the result of Applying the ****** operator with *base* and *exponent* as specified in 12.6.4.

12.6.4 Applying the ****** Operator

Returns an implementation-dependent approximation of the result of raising *base* to the power *exponent*.

If *exponent* is **NaN**, the result is **NaN**.

If *exponent* is **+0**, the result is 1, even if *base* is **NaN**.

If *exponent* is **-0**, the result is 1, even if *base* is **NaN**.

If *base* is **NaN** and *exponent* is nonzero, the result is **NaN**.

If *abs(base)* > 1 and *exponent* is **+∞**, the result is **+∞**.

If *abs(base)* > 1 and *exponent* is **-∞**, the result is **+0**.

If *abs(base)* is 1 and *exponent* is **+∞**, the result is **NaN**.

If *abs(base)* is 1 and *exponent* is **-∞**, the result is **NaN**.

If *abs(base)* < 1 and *exponent* is **+∞**, the result is **+0**.

If *abs(base)* < 1 and *exponent* is **-∞**, the result is **+∞**.

If *base* is **+∞** and *exponent* > 0, the result is **+∞**.

If *base* is **+∞** and *exponent* < 0, the result is **+0**.

If *base* is **-∞** and *exponent* > 0 and *exponent* is an odd integer, the result is **-∞**.

If *base* is **-∞** and *exponent* > 0 and *exponent* is not an odd integer, the result is **+∞**.

If *base* is **-∞** and *exponent* < 0 and *exponent* is an odd integer, the result is **-0**.

If *base* is **-∞** and *exponent* < 0 and *exponent* is not an odd integer, the result is **+0**.

If *base* is **+0** and *exponent* > 0, the result is **+0**.

If *base* is **+0** and *exponent* < 0, the result is **+∞**.

If *base* is **-0** and *exponent* > 0 and *exponent* is an odd integer, the result is **-0**.

If *base* is **-0** and *exponent* > 0 and *exponent* is not an odd integer, the result is **+0**.

If *base* is **-0** and *exponent* < 0 and *exponent* is an odd integer, the result is **-∞**.

If *base* is **-0** and *exponent* < 0 and *exponent* is not an odd integer, the result is **+∞**.

If *base* < 0 and *base* is finite and *exponent* is finite and *exponent* is not an integer, the result is **NaN**.

NOTE

The result of *base* ****** *exponent* when *base* is **1** or **-1** and *exponent* is **+Infinity** or **-Infinity** differs from IEEE 754-2008.

The first edition of ECMAScript specified a result of **NaN** for this operation, whereas later versions of IEEE 754-2008 specified **1**. The historical ECMAScript behaviour is preserved for compatibility reasons.

In the remaining cases, where neither an infinity nor **NaN** is involved, the product is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the result is then a zero of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754-2008.

12.7.3.2 Applying the / Operator

The */ MultiplicativeOperator* performs division, producing the quotient of its operands. The left operand is the dividend and the right operand is the divisor. ECMAScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification of IEEE 754-2008 arithmetic:

If either operand is **NaN**, the result is **NaN**.

The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
Division of an infinity by an infinity results in **NaN**.

Division of an infinity by a zero results in an infinity. The sign is determined by the rule already stated above.

Division of an infinity by a nonzero finite value results in a signed infinity. The sign is determined by the rule already stated above.

Division of a finite value by an infinity results in zero. The sign is determined by the rule already stated above.

Division of a zero by a zero results in **NaN**; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.

Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.

In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode. If the magnitude is too large to represent, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754-2008.

12.7.3.3 Applying the % Operator

The *% MultiplicativeOperator* yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

NOTE

In C and C++, the remainder operator accepts only integral operands; in ECMAScript, it also accepts floating-point operands.

The result of a floating-point remainder operation as computed by the **%** operator is not the same as the “remainder” operation defined by IEEE 754-2008. The IEEE 754-2008 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behaviour is not analogous to that of the usual integer remainder operator. Instead the ECMAScript language defines **%** on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function `fmod`.

The result of an ECMAScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

If either operand is **NaN**, the result is **NaN**.

The sign of the result equals the sign of the dividend.

6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
 - a. Let *lstr* be ? ToString(*lprim*).
 - b. Let *rstr* be ? ToString(*rprim*).
 - c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

NOTE 1

No hint is provided in the calls to `ToPrimitive` in steps 5 and 6. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2

Step 7 differs from step 3 of the `Abstract Relational Comparison` algorithm, by using the logical-or operation instead of the logical-and operation.

12.8.4 The Subtraction Operator (-)

12.8.4.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* - *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lnum* be ? ToNumber(*lval*).
6. Let *rnum* be ? ToNumber(*rval*).
7. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 12.8.5.

12.8.5 Applying the Additive Operators to Numbers

The `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The `-` operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754-2008 binary double-precision arithmetic:

If either operand is **NaN**, the result is **NaN**.

The sum of two infinities of opposite sign is **NaN**.

The sum of two infinities of the same sign is the infinity of that sign.

The sum of an infinity and a finite value is equal to the infinite operand.

The sum of two negative zeroes is **-0**. The sum of two positive zeroes, or of two zeroes of opposite sign, is **+0**.

The sum of a zero and a nonzero finite value is equal to the nonzero operand.

The sum of two nonzero finite values of the same magnitude and opposite sign is **+0**.

In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE

2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *lnum* be ? *ToInt32(lval)*.
6. Let *rnum* be ? *ToUint32(rval)*.
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
8. Return the result of left shifting *lnum* by *shiftCount* bits. The result is a signed 32-bit integer.

12.9.4 The Signed Right Shift Operator ($>>$)

NOTE

Performs a sign-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

12.9.4.1 Runtime Semantics: Evaluation

ShiftExpression : *ShiftExpression* $>>$ *AdditiveExpression*

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *lnum* be ? *ToInt32(lval)*.
6. Let *rnum* be ? *ToUint32(rval)*.
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
8. Return the result of performing a sign-extending right shift of *lnum* by *shiftCount* bits. The most significant bit is propagated. The result is a signed 32-bit integer.

12.9.5 The Unsigned Right Shift Operator ($>>>$)

NOTE

Performs a zero-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

12.9.5.1 Runtime Semantics: Evaluation

ShiftExpression : *ShiftExpression* $>>>$ *AdditiveExpression*

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *lnum* be ? *ToUint32(lval)*.
6. Let *rnum* be ? *ToUint32(rval)*.
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
8. Return the result of performing a zero-filling right shift of *lnum* by *shiftCount* bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *r* be the result of performing *Abstract Relational Comparison lval < rval*.
6. *ReturnIfAbrupt(r)*.
7. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression : *RelationalExpression* > *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *r* be the result of performing *Abstract Relational Comparison rval < lval* with *LeftFirst* equal to **false**.
6. *ReturnIfAbrupt(r)*.
7. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression : *RelationalExpression* <= *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *r* be the result of performing *Abstract Relational Comparison rval < lval* with *LeftFirst* equal to **false**.
6. *ReturnIfAbrupt(r)*.
7. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

RelationalExpression : *RelationalExpression* >= *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Let *r* be the result of performing *Abstract Relational Comparison lval < rval*.
6. *ReturnIfAbrupt(r)*.
7. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

RelationalExpression : *RelationalExpression* **instanceof** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? *GetValue(rref)*.
5. Return ? *InstanceofOperator(lval, rval)*.

RelationalExpression : *RelationalExpression* **in** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? *GetValue(lref)*.

1. Return **false**.

12.11.2 Static Semantics: AssignmentTargetType

EqualityExpression :

- EqualityExpression* **==** *RelationalExpression*
- EqualityExpression* **!=** *RelationalExpression*
- EqualityExpression* **==** **==** *RelationalExpression*
- EqualityExpression* **!=** **==** *RelationalExpression*

1. Return **invalid**.

12.11.3 Runtime Semantics: Evaluation

EqualityExpression : *EqualityExpression* **==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Return the result of performing **Abstract Equality Comparison** *rval* **==** *lval*.

EqualityExpression : *EqualityExpression* **!=** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Let *r* be the result of performing **Abstract Equality Comparison** *rval* **==** *lval*.
6. If *r* is **true**, return **false**. Otherwise, return **true**.

EqualityExpression : *EqualityExpression* **==** **==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Return the result of performing **Strict Equality Comparison** *rval* **==** *lval*.

EqualityExpression : *EqualityExpression* **!=** **==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Let *r* be the result of performing **Strict Equality Comparison** *rval* **==** *lval*.
6. If *r* is **true**, return **false**. Otherwise, return **true**.

NOTE 1

Given the above definition of equality:

12.12.2 Static Semantics: AssignmentTargetType

BitwiseANDExpression : *BitwiseANDExpression* & *EqualityExpression*

BitwiseXORExpression : *BitwiseXORExpression* ^ *BitwiseANDExpression*

BitwiseORExpression : *BitwiseORExpression* | *BitwiseXORExpression*

1. Return invalid.

12.12.3 Runtime Semantics: Evaluation

The production *A* : *A* @ *B*, where @ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Let *lref* be the result of evaluating *A*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *B*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lnum* be ?ToInt32(*lval*).
6. Let *rnum* be ?ToInt32(*rval*).
7. Return the result of applying the bitwise operator @ to *lnum* and *rnum*. The result is a signed 32-bit integer.

12.13 Binary Logical Operators

Syntax

LogicalANDExpression [In, Yield, Await] :

BitwiseORExpression [?In, ?Yield, ?Await]

LogicalANDExpression [?In, ?Yield, ?Await] && *BitwiseORExpression* [?In, ?Yield, ?Await]

LogicalORExpression [In, Yield, Await] :

LogicalANDExpression [?In, ?Yield, ?Await]

LogicalORExpression [?In, ?Yield, ?Await] || *LogicalANDExpression* [?In, ?Yield, ?Await]

NOTE

The value produced by a && or || operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.

12.13.1 Static Semantics: IsFunctionDefinition

LogicalANDExpression : *LogicalANDExpression* && *BitwiseORExpression*

LogicalORExpression : *LogicalORExpression* || *LogicalANDExpression*

1. Return **false**.

12.13.2 Static Semantics: AssignmentTargetType

LogicalANDExpression : *LogicalANDExpression* && *BitwiseORExpression*

LogicalORExpression : *LogicalORExpression* || *LogicalANDExpression*

1. Return invalid.

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be *ToBoolean*(? *GetValue(lref)*).
3. If *lval* is **true**, then
 - a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
 - b. Return ? *GetValue(trueRef)*.
4. Else,
 - a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
 - b. Return ? *GetValue(falseRef)*.

12.15 Assignment Operators

Syntax

```

AssignmentExpression [In, Yield, Await] :
  ConditionalExpression [?In, ?Yield, ?Await]
  [+Yield] YieldExpression [?In, ?Await]
  ArrowFunction [?In, ?Yield, ?Await]
  AsyncArrowFunction [?In, ?Yield, ?Await]
  LeftHandSideExpression [?Yield, ?Await] = AssignmentExpression [?In, ?Yield, ?Await]
  LeftHandSideExpression [?Yield, ?Await] AssignmentOperator
    AssignmentExpression [?In, ?Yield, ?Await]

AssignmentOperator : one of
  *= /= %= += -= <<= >>= >>>= &= ^= |= **=

```

12.15.1 Static Semantics: Early Errors

AssignmentExpression : *LeftHandSideExpression* = *AssignmentExpression*

It is a Syntax Error if *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and *LeftHandSideExpression* is not *covering* an *AssignmentPattern*.

It is an early *Reference* Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and *AssignmentTargetType* of *LeftHandSideExpression* is invalid.

It is an early Syntax Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and *AssignmentTargetType* of *LeftHandSideExpression* is strict.

AssignmentExpression : *LeftHandSideExpression* *AssignmentOperator* *AssignmentExpression*

It is an early *Reference* Error if *AssignmentTargetType* of *LeftHandSideExpression* is invalid.

It is an early Syntax Error if *AssignmentTargetType* of *LeftHandSideExpression* is strict.

12.15.2 Static Semantics: IsFunctionDefinition

```

AssignmentExpression :
  ArrowFunction
  AsyncArrowFunction

```

1. Return **true**.

NOTE

When an assignment occurs within **strict mode code**, it is a runtime error if *lref* in step 1.f of the first algorithm or step 7 of the second algorithm is an unresolvable reference. If it is, a **ReferenceError** exception is thrown. The *LeftHandSideExpression* also may not be a reference to a **data property** with the attribute value `{ [[Writable]]: false }`, to an **accessor property** with the attribute value `{ [[Set]]: undefined }`, nor to a non-existent property of an object for which the **IsExtensible** predicate returns the value **false**. In these cases a **TypeError** exception is thrown.

12.15.5 Destructuring Assignment

Supplemental Syntax

In certain circumstances when processing an instance of the production *AssignmentExpression* :

LeftHandSideExpression = *AssignmentExpression* the following grammar is used to refine the interpretation of *LeftHandSideExpression*.

```
AssignmentPattern[Yield, Await] ::  
  ObjectAssignmentPattern[?Yield, ?Await]  
  ArrayAssignmentPattern[?Yield, ?Await]  
  
ObjectAssignmentPattern[Yield, Await] ::  
  {}  
  { AssignmentRestProperty[?Yield, ?Await] }  
  { AssignmentPropertyList[?Yield, ?Await] }  
  { AssignmentPropertyList[?Yield, ?Await] , AssignmentRestProperty[?Yield, ?Await] opt }  
  
ArrayAssignmentPattern[Yield, Await] ::  
  [ Elisionopt AssignmentRestElement[?Yield, ?Await] opt ]  
  [ AssignmentElementList[?Yield, ?Await] ]  
  [ AssignmentElementList[?Yield, ?Await] , Elisionopt  
    AssignmentRestElement[?Yield, ?Await] opt ]  
  
AssignmentRestProperty[Yield, Await] ::  
  ... DestructuringAssignmentTarget[?Yield, ?Await]  
  
AssignmentPropertyList[Yield, Await] ::  
  AssignmentProperty[?Yield, ?Await]  
  AssignmentPropertyList[?Yield, ?Await] , AssignmentProperty[?Yield, ?Await]  
  
AssignmentElementList[Yield, Await] ::  
  AssignmentElisionElement[?Yield, ?Await]  
  AssignmentElementList[?Yield, ?Await] , AssignmentElisionElement[?Yield, ?Await]  
  
AssignmentElisionElement[Yield, Await] ::  
  Elisionopt AssignmentElement[?Yield, ?Await]  
  
AssignmentProperty[Yield, Await] ::  
  IdentifierReference[?Yield, ?Await] Initializer[+In, ?Yield, ?Await] opt
```


2. Let *result* be the result of performing IteratorDestructuringAssignmentEvaluation of *Elision* with *iteratorRecord* as the argument.
3. If *iteratorRecord*.[[Done]] is **false**, return ? IteratorClose(*iteratorRecord*, *result*).
4. Return *result*.

ArrayAssignmentPattern : [*Elision AssignmentRestElement*]

1. Let *iteratorRecord* be ? GetIterator(*value*).
2. If *Elision* is present, then
 - a. Let *status* be the result of performing IteratorDestructuringAssignmentEvaluation of *Elision* with *iteratorRecord* as the argument.
 - b. If *status* is an abrupt completion, then
 - i. **Assert:** *iteratorRecord*.[[Done]] is **true**.
 - ii. Return Completion(*status*).
3. Let *result* be the result of performing IteratorDestructuringAssignmentEvaluation of *AssignmentRestElement* with *iteratorRecord* as the argument.
4. If *iteratorRecord*.[[Done]] is **false**, return ? IteratorClose(*iteratorRecord*, *result*).
5. Return *result*.

ArrayAssignmentPattern : [*AssignmentElementList*]

1. Let *iteratorRecord* be ? GetIterator(*value*).
2. Let *result* be the result of performing IteratorDestructuringAssignmentEvaluation of *AssignmentElementList* using *iteratorRecord* as the argument.
3. If *iteratorRecord*.[[Done]] is **false**, return ? IteratorClose(*iteratorRecord*, *result*).
4. Return *result*.

ArrayAssignmentPattern : [*AssignmentElementList* , *Elision AssignmentRestElement*]

1. Let *iteratorRecord* be ? GetIterator(*value*).
2. Let *status* be the result of performing IteratorDestructuringAssignmentEvaluation of *AssignmentElementList* using *iteratorRecord* as the argument.
3. If *status* is an abrupt completion, then
 - a. If *iteratorRecord*.[[Done]] is **false**, return ? IteratorClose(*iteratorRecord*, *status*).
 - b. Return Completion(*status*).
4. If *Elision* is present, then
 - a. Set *status* to the result of performing IteratorDestructuringAssignmentEvaluation of *Elision* with *iteratorRecord* as the argument.
 - b. If *status* is an abrupt completion, then
 - i. **Assert:** *iteratorRecord*.[[Done]] is **true**.
 - ii. Return Completion(*status*).
5. If *AssignmentRestElement* is present, then
 - a. Set *status* to the result of performing IteratorDestructuringAssignmentEvaluation of *AssignmentRestElement* with *iteratorRecord* as the argument.
6. If *iteratorRecord*.[[Done]] is **false**, return ? IteratorClose(*iteratorRecord*, *status*).
7. Return Completion(*status*).

ObjectAssignmentPattern : { *AssignmentRestProperty* }

1. Perform ? RequireObjectCoercible(*value*).
2. Let *excludedNames* be a new empty List.

AssignmentRestProperty : . . . *DestructuringAssignmentTarget*

1. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
2. *ReturnIfAbrupt(lref)*.
3. Let *restObj* be *ObjectCreate(%ObjectPrototype%)*.
4. Perform ? *CopyDataProperties(restObj, value, excludedNames)*.
5. Return *PutValue(lref, restObj)*.

12.15.5.5 Runtime Semantics: IteratorDestructuringAssignmentEvaluation

With parameter *iteratorRecord*.

AssignmentElementList : *AssignmentElisionElement*

1. Return the result of performing *IteratorDestructuringAssignmentEvaluation* of *AssignmentElisionElement* using *iteratorRecord* as the argument.

AssignmentElementList : *AssignmentElementList* , *AssignmentElisionElement*

1. Perform ? *IteratorDestructuringAssignmentEvaluation* of *AssignmentElementList* using *iteratorRecord* as the argument.
2. Return the result of performing *IteratorDestructuringAssignmentEvaluation* of *AssignmentElisionElement* using *iteratorRecord* as the argument.

AssignmentElisionElement : *AssignmentElement*

1. Return the result of performing *IteratorDestructuringAssignmentEvaluation* of *AssignmentElement* with *iteratorRecord* as the argument.

AssignmentElisionElement : *Elision AssignmentElement*

1. Perform ? *IteratorDestructuringAssignmentEvaluation* of *Elision* with *iteratorRecord* as the argument.
2. Return the result of performing *IteratorDestructuringAssignmentEvaluation* of *AssignmentElement* with *iteratorRecord* as the argument.

Elision : ,

1. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be *IteratorStep(iteratorRecord)*.
 - b. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - c. *ReturnIfAbrupt(next)*.
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
2. Return *NormalCompletion(empty)*.

Elision : *Elision* ,

1. Perform ? *IteratorDestructuringAssignmentEvaluation* of *Elision* with *iteratorRecord* as the argument.
2. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be *IteratorStep(iteratorRecord)*.
 - b. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - c. *ReturnIfAbrupt(next)*.
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.

- i. Let `nextValue` be `IteratorValue(next)`.
 - ii. If `nextValue` is an abrupt completion, set `iteratorRecord`.`[[Done]]` to `true`.
 - iii. `ReturnIfAbrupt(nextValue)`.
 - iv. Let `status` be `CreateDataProperty(A, ! ToString(n), nextValue)`.
 - v. `Assert: status` is `true`.
 - vi. Increment `n` by 1.
5. If `DestructuringAssignmentTarget` is neither an `ObjectLiteral` nor an `ArrayLiteral`, then
- a. Return `? PutValue(lref, A)`.
6. Let `nestedAssignmentPattern` be the `AssignmentPattern` that is `covered` by `DestructuringAssignmentTarget`.
7. Return the result of performing `DestructuringAssignmentEvaluation` of `nestedAssignmentPattern` with `A` as the argument.

12.15.5.6 Runtime Semantics: KeyedDestructuringAssignmentEvaluation

With parameters `value` and `propertyName`.

AssignmentElement : *DestructuringAssignmentTarget Initializer*

- 1. If `DestructuringAssignmentTarget` is neither an `ObjectLiteral` nor an `ArrayLiteral`, then
 - a. Let `lref` be the result of evaluating `DestructuringAssignmentTarget`.
 - b. `ReturnIfAbrupt(lref)`.
- 2. Let `v` be `? GetV(value, propertyName)`.
- 3. If `Initializer` is present and `v` is `undefined`, then
 - a. If `IsAnonymousFunctionDefinition(Initializer)` and `IsIdentifierRef` of `DestructuringAssignmentTarget` are both `true`, then
 - i. Let `rhsValue` be the result of performing `NamedEvaluation` for `Initializer` with argument `GetReferencedName(lref)`.
 - b. Else,
 - i. Let `defaultValue` be the result of evaluating `Initializer`.
 - ii. Let `rhsValue` be `? GetValue(defaultValue)`.
- 4. Else, let `rhsValue` be `v`.
- 5. If `DestructuringAssignmentTarget` is an `ObjectLiteral` or an `ArrayLiteral`, then
 - a. Let `assignmentPattern` be the `AssignmentPattern` that is `covered` by `DestructuringAssignmentTarget`.
 - b. Return the result of performing `DestructuringAssignmentEvaluation` of `assignmentPattern` with `rhsValue` as the argument.
- 6. Return `? PutValue(lref, rhsValue)`.

12.16 Comma Operator (,)

Syntax

```
Expression[In, Yield, Await] :
  AssignmentExpression[?In, ?Yield, ?Await]
  Expression[?In, ?Yield, ?Await] , AssignmentExpression[?In, ?Yield, ?Await]
```

12.16.1 Static Semantics: IsFunctionDefinition

Expression : *Expression , AssignmentExpression*


```

HoistableDeclaration[Yield, Await, Default] :
  FunctionDeclaration[?Yield, ?Await, ?Default]
  GeneratorDeclaration[?Yield, ?Await, ?Default]
  AsyncFunctionDeclaration[?Yield, ?Await, ?Default]
  AsyncGeneratorDeclaration[?Yield, ?Await, ?Default]

BreakableStatement[Yield, Await, Return] :
  IterationStatement[?Yield, ?Await, ?Return]
  SwitchStatement[?Yield, ?Await, ?Return]

```

13.1 Statement Semantics

13.1.1 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

```

Statement :
  VariableStatement
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement

```

1. Return **false**.

13.1.2 Static Semantics: ContainsUndefinedBreakTarget

With parameter *labelSet*.

```

Statement :
  VariableStatement
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement

```

1. Return **false**.

13.1.3 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

```

Statement :

```


13.1.6 Static Semantics: VarScopedDeclarations

Statement :

EmptyStatement
ExpressionStatement
ContinueStatement
BreakStatement
ReturnStatement
ThrowStatement
DebuggerStatement

1. Return a new empty [List](#).

13.1.7 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

BreakableStatement : *IterationStatement*

1. Let *stmtResult* be the result of performing LabelledEvaluation of *IterationStatement* with argument *labelSet*.
2. If *stmtResult*.[[Type]] is **break**, then
 - a. If *stmtResult*.[[Target]] is empty, then
 - i. If *stmtResult*.[[Value]] is empty, set *stmtResult* to **NormalCompletion(undefined)**.
 - ii. Else, set *stmtResult* to **NormalCompletion(stmtResult.[[Value]])**.
3. Return [Completion\(stmtResult\)](#).

BreakableStatement : *SwitchStatement*

1. Let *stmtResult* be the result of evaluating *SwitchStatement*.
2. If *stmtResult*.[[Type]] is **break**, then
 - a. If *stmtResult*.[[Target]] is empty, then
 - i. If *stmtResult*.[[Value]] is empty, set *stmtResult* to **NormalCompletion(undefined)**.
 - ii. Else, set *stmtResult* to **NormalCompletion(stmtResult.[[Value]])**.
3. Return [Completion\(stmtResult\)](#).

NOTE

A *BreakableStatement* is one that can be exited via an unlabelled *BreakStatement*.

13.1.8 Runtime Semantics: Evaluation

HoistableDeclaration :

GeneratorDeclaration
AsyncFunctionDeclaration
AsyncGeneratorDeclaration

1. Return **NormalCompletion(empty)**.

HoistableDeclaration : *FunctionDeclaration*

1. Return the result of evaluating *FunctionDeclaration*.

13.2.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter *labelSet*.

Block : { }

1. Return **false**.

StatementList : *StatementList StatementListItem*

1. Let *hasUndefinedLabels* be ContainsUndefinedBreakTarget of *StatementList* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return ContainsUndefinedBreakTarget of *StatementListItem* with argument *labelSet*.

StatementListItem : *Declaration*

1. Return **false**.

13.2.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

Block : { }

1. Return **false**.

StatementList : *StatementList StatementListItem*

1. Let *hasUndefinedLabels* be ContainsUndefinedContinueTarget of *StatementList* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return ContainsUndefinedContinueTarget of *StatementListItem* with arguments *iterationSet* and « ».

StatementListItem : *Declaration*

1. Return **false**.

13.2.5 Static Semantics: LexicallyDeclaredNames

Block : { }

1. Return a new empty *List*.

StatementList : *StatementList StatementListItem*

1. Let *names* be LexicallyDeclaredNames of *StatementList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return LexicallyDeclaredNames of *LabelledStatement*.
2. Return a new empty *List*.

StatementListItem : *Declaration*

StatementListItem : *Statement*

1. Return a new empty [List](#).

StatementListItem : *Declaration*

1. If *Declaration* is *Declaration* : *HoistableDeclaration* , then
 - a. Return « ».
2. Return a new [List](#) containing *Declaration*.

13.2.9 Static Semantics: TopLevelVarDeclaredNames

Block : { }

1. Return a new empty [List](#).

StatementList : *StatementList StatementListItem*

1. Let *names* be TopLevelVarDeclaredNames of *StatementList*.
2. Append to *names* the elements of the TopLevelVarDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem : *Declaration*

1. If *Declaration* is *Declaration* : *HoistableDeclaration* , then
 - a. Return the BoundNames of *HoistableDeclaration*.
2. Return a new empty [List](#).

StatementListItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return TopLevelVarDeclaredNames of *Statement*.
2. Return VarDeclaredNames of *Statement*.

NOTE

At the top level of a function or script, inner function declarations are treated like var declarations.

13.2.10 Static Semantics: TopLevelVarScopedDeclarations

Block : { }

1. Return a new empty [List](#).

StatementList : *StatementList StatementListItem*

1. Let *declarations* be TopLevelVarScopedDeclarations of *StatementList*.
2. Append to *declarations* the elements of the TopLevelVarScopedDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return TopLevelVarScopedDeclarations of *Statement*.
2. Return VarScopedDeclarations of *Statement*.

StatementListItem : *Declaration*

No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

StatementList : *StatementList StatementListItem*

1. Let *sl* be the result of evaluating *StatementList*.
2. [ReturnIfAbrupt\(*sl*\)](#).
3. Let *s* be the result of evaluating *StatementListItem*.
4. Return [Completion\(UpdateEmpty\(*s*, *sl*\)\)](#).

NOTE 2

The value of a *StatementList* is the value of the last value-producing item in the *StatementList*. For example, the following calls to the **eval** function all return the value 1:

```
eval("1;;;;;")  
eval("1;{ }")  
eval("1;var a;")
```

13.2.14 Runtime Semantics: BlockDeclarationInstantiation (*code*, *env*)

NOTE

When a *Block* or *CaseBlock* is evaluated a new declarative [Environment Record](#) is created and bindings for each block scoped variable, constant, function, or class declared in the block are instantiated in the [Environment Record](#).

BlockDeclarationInstantiation is performed as follows using arguments *code* and *env*. *code* is the [Parse Node](#) corresponding to the body of the block. *env* is the [Lexical Environment](#) in which bindings are to be created.

1. Let *envRec* be *env*'s [EnvironmentRecord](#).
2. [Assert: *envRec* is a declarative Environment Record](#).
3. Let *declarations* be the LexicallyScopedDeclarations of *code*.
4. For each element *d* in *declarations*, do
 - a. For each element *dn* of the BoundNames of *d*, do
 - i. If IsConstantDeclaration of *d* is **true**, then
 1. Perform ! *envRec*.CreateImmutableBinding(*dn*, **true**).
 - ii. Else,
 1. Perform ! *envRec*.CreateMutableBinding(*dn*, **false**).
 - b. If *d* is a *FunctionDeclaration*, a *GeneratorDeclaration*, an *AsyncFunctionDeclaration*, or an *AsyncGeneratorDeclaration*, then
 - i. Let *fn* be the sole element of the BoundNames of *d*.
 - ii. Let *fo* be the result of performing [InstantiateFunctionObject](#) for *d* with argument *env*.
 - iii. Perform *envRec*.InitializeBinding(*fn*, *fo*).

13.3 Declarations and the Variable Statement

13.3.1 Let and Const Declarations

NOTE

let and **const** declarations define variables that are scoped to the [running execution context](#)'s LexicalEnvironment.

1. Return the BoundNames of *BindingPattern*.

13.3.1.3 Static Semantics: IsConstantDeclaration

LexicalDeclaration : *LetOrConst BindingList ;*

1. Return IsConstantDeclaration of *LetOrConst*.

LetOrConst : **let**

1. Return **false**.

LetOrConst : **const**

1. Return **true**.

13.3.1.4 Runtime Semantics: Evaluation

LexicalDeclaration : *LetOrConst BindingList ;*

1. Let *next* be the result of evaluating *BindingList*.
2. [ReturnIfAbrupt\(*next*\)](#).
3. Return [NormalCompletion\(empty\)](#).

BindingList : *BindingList , LexicalBinding*

1. Let *next* be the result of evaluating *BindingList*.
2. [ReturnIfAbrupt\(*next*\)](#).
3. Return the result of evaluating *LexicalBinding*.

LexicalBinding : *BindingIdentifier*

1. Let *lhs* be [ResolveBinding\(StringValue of *BindingIdentifier*\)](#).
2. Return [InitializeReferencedBinding\(*lhs*, **undefined**\)](#).

NOTE

A **static semantics** rule ensures that this form of *LexicalBinding* never occurs in a **const** declaration.

LexicalBinding : *BindingIdentifier Initializer*

1. Let *bindingId* be StringValue of *BindingIdentifier*.
2. Let *lhs* be [ResolveBinding\(*bindingId*\)](#).
3. If [IsAnonymousFunctionDefinition\(*Initializer*\)](#) is **true**, then
 - a. Let *value* be the result of performing [NamedEvaluation](#) for *Initializer* with argument *bindingId*.
4. Else,
 - a. Let *rhs* be the result of evaluating *Initializer*.
 - b. Let *value* be [? GetValue\(*rhs*\)](#).
5. Return [InitializeReferencedBinding\(*lhs*, *value*\)](#).

LexicalBinding : *BindingPattern Initializer*

1. Let *rhs* be the result of evaluating *Initializer*.
2. Let *value* be [? GetValue\(*rhs*\)](#).
3. Let *env* be the [running execution context](#)'s LexicalEnvironment.

VariableDeclarationList : *VariableDeclarationList* , *VariableDeclaration*

1. Let *declarations* be VarScopedDeclarations of *VariableDeclarationList*.
2. Append *VariableDeclaration* to *declarations*.
3. Return *declarations*.

13.3.2.4 Runtime Semantics: Evaluation

VariableStatement : **var** *VariableDeclarationList* ;

1. Let *next* be the result of evaluating *VariableDeclarationList*.
2. **ReturnIfAbrupt**(*next*).
3. Return **NormalCompletion**(empty).

VariableDeclarationList : *VariableDeclarationList* , *VariableDeclaration*

1. Let *next* be the result of evaluating *VariableDeclarationList*.
2. **ReturnIfAbrupt**(*next*).
3. Return the result of evaluating *VariableDeclaration*.

VariableDeclaration : *BindingIdentifier*

1. Return **NormalCompletion**(empty).

VariableDeclaration : *BindingIdentifier Initializer*

1. Let *bindingId* be StringValue of *BindingIdentifier*.
2. Let *lhs* be ? **ResolveBinding**(*bindingId*).
3. If **IsAnonymousFunctionDefinition**(*Initializer*) is **true**, then
 - a. Let *value* be the result of performing **NamedEvaluation** for *Initializer* with argument *bindingId*.
4. Else,
 - a. Let *rhs* be the result of evaluating *Initializer*.
 - b. Let *value* be ? **GetValue**(*rhs*).
5. Return ? **PutValue**(*lhs*, *value*).

NOTE

If a *VariableDeclaration* is nested within a *with* statement and the *BindingIdentifier* in the *VariableDeclaration* is the same as a **property name** of the binding object of the *with* statement's object **Environment Record**, then step 6 will assign *value* to the property instead of assigning to the *VariableEnvironment* binding of the *Identifier*.

VariableDeclaration : *BindingPattern Initializer*

1. Let *rhs* be the result of evaluating *Initializer*.
2. Let *rval* be ? **GetValue**(*rhs*).
3. Return the result of performing **BindingInitialization** for *BindingPattern* passing *rval* and **undefined** as arguments.

13.3.3 Destructuring Binding Patterns

Syntax

BindingPattern [**Yield**, **Await**] :
ObjectBindingPattern [**?Yield**, **?Await**]

1. Return a new empty *List*.

ArrayBindingPattern : [*Elision* *BindingRestElement*]

1. Return the *BoundNames* of *BindingRestElement*.

ArrayBindingPattern : [*BindingElementList* , *Elision*]

1. Return the *BoundNames* of *BindingElementList*.

ArrayBindingPattern : [*BindingElementList* , *Elision* *BindingRestElement*]

1. Let *names* be *BoundNames* of *BindingElementList*.
2. Append to *names* the elements of *BoundNames* of *BindingRestElement*.
3. Return *names*.

BindingPropertyList : *BindingPropertyList* , *BindingProperty*

1. Let *names* be *BoundNames* of *BindingPropertyList*.
2. Append to *names* the elements of *BoundNames* of *BindingProperty*.
3. Return *names*.

BindingElementList : *BindingElementList* , *BindingElisionElement*

1. Let *names* be *BoundNames* of *BindingElementList*.
2. Append to *names* the elements of *BoundNames* of *BindingElisionElement*.
3. Return *names*.

BindingElisionElement : *Elision* *BindingElement*

1. Return *BoundNames* of *BindingElement*.

BindingProperty : *PropertyName* : *BindingElement*

1. Return the *BoundNames* of *BindingElement*.

SingleNameBinding : *BindingIdentifier* *Initializer*

1. Return the *BoundNames* of *BindingIdentifier*.

BindingElement : *BindingPattern* *Initializer*

1. Return the *BoundNames* of *BindingPattern*.

13.3.3.2 Static Semantics: ContainsExpression

ObjectBindingPattern : { }

1. Return **false**.

ArrayBindingPattern : [*Elision*]

1. Return **false**.

ArrayBindingPattern : [*Elision* *BindingRestElement*]

BindingElement : *BindingPattern*

1. Return **false**.

BindingElement : *BindingPattern Initializer*

1. Return **true**.

SingleNameBinding : *BindingIdentifier*

1. Return **false**.

SingleNameBinding : *BindingIdentifier Initializer*

1. Return **true**.

13.3.3.4 Static Semantics: IsSimpleParameterList

BindingElement : *BindingPattern*

1. Return **false**.

BindingElement : *BindingPattern Initializer*

1. Return **false**.

SingleNameBinding : *BindingIdentifier*

1. Return **true**.

SingleNameBinding : *BindingIdentifier Initializer*

1. Return **false**.

13.3.3.5 Runtime Semantics: BindingInitialization

With parameters *value* and *environment*.

NOTE

When **undefined** is passed for *environment* it indicates that a **PutValue** operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

BindingPattern : *ObjectBindingPattern*

1. Perform ? **RequireObjectCoercible**(*value*).
2. Return the result of performing **BindingInitialization** for *ObjectBindingPattern* using *value* and *environment* as arguments.

BindingPattern : *ArrayBindingPattern*

1. Let *iteratorRecord* be ? **GetIterator**(*value*).
2. Let *result* be **IteratorBindingInitialization** for *ArrayBindingPattern* using *iteratorRecord* and *environment* as arguments.
3. If *iteratorRecord*.[[Done]] is **false**, return ? **IteratorClose**(*iteratorRecord*, *result*).

3. Perform ? KeyedBindingInitialization of *BindingElement* with *value*, *environment*, and *P* as the arguments.
4. Return a new *List* containing *P*.

13.3.3.7 Runtime Semantics: RestBindingInitialization

With parameters *value*, *environment*, and *excludedNames*.

BindingRestProperty : . . . *BindingIdentifier*

1. Let *lhs* be ? ResolveBinding(StringValue of *BindingIdentifier*, *environment*).
2. Let *restObj* be ObjectCreate(%ObjectPrototype%).
3. Perform ? CopyDataProperties(*restObj*, *value*, *excludedNames*).
4. If *environment* is **undefined**, return PutValue(*lhs*, *restObj*).
5. Return InitializeReferencedBinding(*lhs*, *restObj*).

13.3.3.8 Runtime Semantics: IteratorBindingInitialization

With parameters *iteratorRecord* and *environment*.

NOTE

When **undefined** is passed for *environment* it indicates that a *PutValue* operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

ArrayBindingPattern : []

1. Return NormalCompletion(empty).

ArrayBindingPattern : [*Elision*]

1. Return the result of performing IteratorDestructuringAssignmentEvaluation of *Elision* with *iteratorRecord* as the argument.

ArrayBindingPattern : [*Elision* *BindingRestElement*]

1. If *Elision* is present, then
 - a. Perform ? IteratorDestructuringAssignmentEvaluation of *Elision* with *iteratorRecord* as the argument.
2. Return the result of performing IteratorBindingInitialization for *BindingRestElement* with *iteratorRecord* and *environment* as arguments.

ArrayBindingPattern : [*BindingElementList*]

1. Return the result of performing IteratorBindingInitialization for *BindingElementList* with *iteratorRecord* and *environment* as arguments.

ArrayBindingPattern : [*BindingElementList* ,]

1. Return the result of performing IteratorBindingInitialization for *BindingElementList* with *iteratorRecord* and *environment* as arguments.

ArrayBindingPattern : [*BindingElementList* , *Elision*]

1. Perform ? IteratorBindingInitialization for *BindingElementList* with *iteratorRecord* and *environment* as arguments.

- a. If `IsAnonymousFunctionDefinition`(*Initializer*) is **true**, then
 - i. Set *v* to the result of performing `NamedEvaluation` for *Initializer* with argument *bindingId*.
 - b. Else,
 - i. Let *defaultValue* be the result of evaluating *Initializer*.
 - ii. Set *v* to ? `GetValue`(*defaultValue*).
6. If *environment* is **undefined**, return ? `PutValue`(*lhs*, *v*).
7. Return `InitializeReferencedBinding`(*lhs*, *v*).

BindingElement : *BindingPattern* *Initializer*

- 1. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be `IteratorStep`(*iteratorRecord*).
 - b. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - c. `ReturnIfAbrupt`(*next*).
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
 - e. Else,
 - i. Let *v* be `IteratorValue`(*next*).
 - ii. If *v* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - iii. `ReturnIfAbrupt`(*v*).
- 2. If *iteratorRecord*.[[Done]] is **true**, let *v* be **undefined**.
- 3. If *Initializer* is present and *v* is **undefined**, then
 - a. Let *defaultValue* be the result of evaluating *Initializer*.
 - b. Set *v* to ? `GetValue`(*defaultValue*).
- 4. Return the result of performing `BindingInitialization` of *BindingPattern* with *v* and *environment* as the arguments.

BindingRestElement : . . . *BindingIdentifier*

- 1. Let *lhs* be ? `ResolveBinding`(*StringValue* of *BindingIdentifier*, *environment*).
- 2. Let *A* be ! `ArrayCreate`(0).
- 3. Let *n* be 0.
- 4. Repeat,
 - a. If *iteratorRecord*.[[Done]] is **false**, then
 - i. Let *next* be `IteratorStep`(*iteratorRecord*).
 - ii. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - iii. `ReturnIfAbrupt`(*next*).
 - iv. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
 - b. If *iteratorRecord*.[[Done]] is **true**, then
 - i. If *environment* is **undefined**, return ? `PutValue`(*lhs*, *A*).
 - ii. Return `InitializeReferencedBinding`(*lhs*, *A*).
 - c. Let *nextValue* be `IteratorValue`(*next*).
 - d. If *nextValue* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - e. `ReturnIfAbrupt`(*nextValue*).
 - f. Let *status* be `CreateDataProperty`(*A*, ! `ToString`(*n*), *nextValue*).
 - g. **Assert:** *status* is **true**.
 - h. Increment *n* by 1.

BindingRestElement : . . . *BindingPattern*

- 1. Let *A* be ! `ArrayCreate`(0).
- 2. Let *n* be 0.

Syntax

EmptyStatement :

;

13.4.1 Runtime Semantics: Evaluation

EmptyStatement :

1. Return `NormalCompletion(empty)`.

13.5 Expression Statement

Syntax

ExpressionStatement [*Yield*, *Await*] :

[lookahead $\notin \{ \{, \text{function}, \text{async}$ [no LineTerminator here] $\text{function}, \text{class}, \text{let} \} \}]$

Expression [*+In*, *?Yield*, *?Await*] ;

NOTE

An *ExpressionStatement* cannot start with a U+007B (LEFT CURLY BRACKET) because that might make it ambiguous with a *Block*. An *ExpressionStatement* cannot start with the **function** or **class** keywords because that would make it ambiguous with a *FunctionDeclaration*, a *GeneratorDeclaration*, or a *ClassDeclaration*. An *ExpressionStatement* cannot start with **async function** because that would make it ambiguous with an *AsyncFunctionDeclaration* or a *AsyncGeneratorDeclaration*. An *ExpressionStatement* cannot start with the two token sequence **let** [because that would make it ambiguous with a **let** *LexicalDeclaration* whose first *LexicalBinding* was an *ArrayBindingPattern*.

13.5.1 Runtime Semantics: Evaluation

ExpressionStatement : *Expression* ;

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return `?GetValue(exprRef)`.

13.6 The **if** Statement

Syntax

IfStatement [*Yield*, *Await*, *Return*] :

if (*Expression* [*+In*, *?Yield*, *?Await*]) *Statement* [*?Yield*, *?Await*, *?Return*] **else**
Statement [*?Yield*, *?Await*, *?Return*]
if (*Expression* [*+In*, *?Yield*, *?Await*]) *Statement* [*?Yield*, *?Await*, *?Return*]

Each **else** for which the choice of associated **if** is ambiguous shall be associated with the nearest possible **if** that would otherwise have no corresponding **else**.

1. Return ContainsUndefinedContinueTarget of *Statement* with arguments *iterationSet* and « ».

13.6.5 Static Semantics: VarDeclaredNames

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *names* be VarDeclaredNames of the first *Statement*.
2. Append to *names* the elements of the VarDeclaredNames of the second *Statement*.
3. Return *names*.

IfStatement : **if** (*Expression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

13.6.6 Static Semantics: VarScopedDeclarations

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *declarations* be VarScopedDeclarations of the first *Statement*.
2. Append to *declarations* the elements of the VarScopedDeclarations of the second *Statement*.
3. Return *declarations*.

IfStatement : **if** (*Expression*) *Statement*

1. Return the VarScopedDeclarations of *Statement*.

13.6.7 Runtime Semantics: Evaluation

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(? GetValue(*exprRef*)).
3. If *exprValue* is **true**, then
 - a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
4. Else,
 - a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
5. Return Completion(UpdateEmpty(*stmtCompletion*, **undefined**)).

IfStatement : **if** (*Expression*) *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(? GetValue(*exprRef*)).
3. If *exprValue* is **false**, then
 - a. Return NormalCompletion(**undefined**).
4. Else,
 - a. Let *stmtCompletion* be the result of evaluating *Statement*.
 - b. Return Completion(UpdateEmpty(*stmtCompletion*, **undefined**)).

13.7 Iteration Statements

Syntax

IterationStatement :

```
do Statement while ( Expression ) ;
while ( Expression ) Statement
for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement
for ( LeftHandSideExpression in Expression ) Statement
for ( var ForBinding in Expression ) Statement
for ( ForDeclaration in Expression ) Statement
for ( LeftHandSideExpression of AssignmentExpression ) Statement
for ( var ForBinding of AssignmentExpression ) Statement
for ( ForDeclaration of AssignmentExpression ) Statement
for await ( LeftHandSideExpression of AssignmentExpression ) Statement
for await ( var ForBinding of AssignmentExpression ) Statement
for await ( ForDeclaration of AssignmentExpression ) Statement
```

It is a Syntax Error if `IsLabelledFunction(Statement)` is **true**.

NOTE

It is only necessary to apply this rule if the extension specified in [B.3.2](#) is implemented.

13.7.1.2 Runtime Semantics: LoopContinues (*completion*, *labelSet*)

The abstract operation `LoopContinues` with arguments *completion* and *labelSet* is defined by the following steps:

1. If *completion*.[[Type]] is **normal**, return **true**.
2. If *completion*.[[Type]] is not **continue**, return **false**.
3. If *completion*.[[Target]] is empty, return **true**.
4. If *completion*.[[Target]] is an element of *labelSet*, return **true**.
5. Return **false**.

NOTE

Within the *Statement* part of an *IterationStatement* a *ContinueStatement* may be used to begin a new iteration.

13.7.2 The do-while Statement

13.7.2.1 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

IterationStatement : `do Statement while (Expression) ;`

1. Return `ContainsDuplicateLabels` of *Statement* with argument *labelSet*.

13.7.2.2 Static Semantics: ContainsUndefinedBreakTarget

With parameter *labelSet*.

IterationStatement : **while** (*Expression*) *Statement*

1. Return ContainsUndefinedBreakTarget of *Statement* with argument *labelSet*.

13.7.3.3 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

IterationStatement : **while** (*Expression*) *Statement*

1. Return ContainsUndefinedContinueTarget of *Statement* with arguments *iterationSet* and « ».

13.7.3.4 Static Semantics: VarDeclaredNames

IterationStatement : **while** (*Expression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

13.7.3.5 Static Semantics: VarScopedDeclarations

IterationStatement : **while** (*Expression*) *Statement*

1. Return the VarScopedDeclarations of *Statement*.

13.7.3.6 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

IterationStatement : **while** (*Expression*) *Statement*

1. Let *V* be **undefined**.
2. Repeat,
 - a. Let *exprRef* be the result of evaluating *Expression*.
 - b. Let *exprValue* be ? GetValue(*exprRef*).
 - c. If ToBoolean(*exprValue*) is **false**, return NormalCompletion(*V*).
 - d. Let *stmtResult* be the result of evaluating *Statement*.
 - e. If LoopContinues(*stmtResult*, *labelSet*) is **false**, return Completion(UpdateEmpty(*stmtResult*, *V*)).
 - f. If *stmtResult*.[[Value]] is not empty, set *V* to *stmtResult*.[[Value]].

13.7.4 The **for** Statement

13.7.4.1 Static Semantics: Early Errors

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

It is a Syntax Error if any element of the BoundNames of *LexicalDeclaration* also occurs in the VarDeclaredNames of *Statement*.

13.7.4.2 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

IterationStatement :

2. Append to *declarations* the elements of the VarScopedDeclarations of *Statement*.
3. Return *declarations*.

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

1. Return the VarScopedDeclarations of *Statement*.

13.7.4.7 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

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

1. If the first *Expression* is present, then
 - a. Let *exprRef* be the result of evaluating the first *Expression*.
 - b. Perform ? GetValue(*exprRef*).
2. Return ? ForBodyEvaluation(the second *Expression*, the third *Expression*, *Statement*, « », *labelSet*).

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

1. Let *varDcl* be the result of evaluating *VariableDeclarationList*.
2. **ReturnIfAbrupt**(*varDcl*).
3. Return ? ForBodyEvaluation(the first *Expression*, the second *Expression*, *Statement*, « », *labelSet*).

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

1. Let *oldEnv* be the *running execution context*'s LexicalEnvironment.
2. Let *loopEnv* be NewDeclarativeEnvironment(*oldEnv*).
3. Let *loopEnvRec* be *loopEnv*'s EnvironmentRecord.
4. Let *isConst* be the result of performing IsConstantDeclaration of *LexicalDeclaration*.
5. Let *boundNames* be the BoundNames of *LexicalDeclaration*.
6. For each element *dn* of *boundNames*, do
 - a. If *isConst* is **true**, then
 - i. Perform ! *loopEnvRec*.CreateImmutableBinding(*dn*, **true**).
 - b. Else,
 - i. Perform ! *loopEnvRec*.CreateMutableBinding(*dn*, **false**).
7. Set the *running execution context*'s LexicalEnvironment to *loopEnv*.
8. Let *forDcl* be the result of evaluating *LexicalDeclaration*.
9. If *forDcl* is an abrupt completion, then
 - a. Set the *running execution context*'s LexicalEnvironment to *oldEnv*.
 - b. Return Completion(*forDcl*).
10. If *isConst* is **false**, let *perIterationLets* be *boundNames*; otherwise let *perIterationLets* be « ».
11. Let *bodyResult* be ForBodyEvaluation(the first *Expression*, the second *Expression*, *Statement*, *perIterationLets*, *labelSet*).
12. Set the *running execution context*'s LexicalEnvironment to *oldEnv*.
13. Return Completion(*bodyResult*).

13.7.4.8 Runtime Semantics: ForBodyEvaluation (*test*, *increment*, *stmt*, *perIterationBindings*, *labelSet*)

The abstract operation ForBodyEvaluation with arguments *test*, *increment*, *stmt*, *perIterationBindings*, and *labelSet* is performed as follows:

It is a Syntax Error if *AssignmentTargetType* of *LeftHandSideExpression* is not **simple**.

It is a Syntax Error if the *LeftHandSideExpression* is *CoverParenthesizedExpressionAndArrowParameterList* : (*Expression*) and *Expression* derives a phrase that would produce a Syntax Error according to these rules if that phrase were substituted for *LeftHandSideExpression*. This rule is recursively applied.

NOTE

The last rule means that the other rules are applied even if parentheses surround *Expression*.

IterationStatement :

```
for ( ForDeclaration in Expression ) Statement
for ( ForDeclaration of AssignmentExpression ) Statement
for await ( ForDeclaration of AssignmentExpression ) Statement
```

It is a Syntax Error if the *BoundNames* of *ForDeclaration* contains "**let**".

It is a Syntax Error if any element of the *BoundNames* of *ForDeclaration* also occurs in the *VarDeclaredNames* of *Statement*.

It is a Syntax Error if the *BoundNames* of *ForDeclaration* contains any duplicate entries.

13.7.5.2 Static Semantics: **BoundNames**

ForDeclaration : *LetOrConst ForBinding*

1. Return the *BoundNames* of *ForBinding*.

13.7.5.3 Static Semantics: **ContainsDuplicateLabels**

With parameter *labelSet*.

IterationStatement :

```
for ( LeftHandSideExpression in Expression ) Statement
for ( var ForBinding in Expression ) Statement
for ( ForDeclaration in Expression ) Statement
for ( LeftHandSideExpression of AssignmentExpression ) Statement
for ( var ForBinding of AssignmentExpression ) Statement
for ( ForDeclaration of AssignmentExpression ) Statement
for await ( LeftHandSideExpression of AssignmentExpression ) Statement
for await ( var ForBinding of AssignmentExpression ) Statement
for await ( ForDeclaration of AssignmentExpression ) Statement
```

1. Return *ContainsDuplicateLabels* of *Statement* with argument *labelSet*.

NOTE

This section is extended by Annex [B.3.6](#).

13.7.5.4 Static Semantics: **ContainsUndefinedBreakTarget**

With parameter *labelSet*.

IterationStatement :

This section is extended by Annex [B.3.6](#).

13.7.5.7 Static Semantics: VarDeclaredNames

IterationStatement : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement : **for** (**var** *ForBinding* **in** *Expression*) *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*.

IterationStatement : **for** (*ForDeclaration* **in** *Expression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement :

for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement :

for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for await (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*.

IterationStatement :

for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*
for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return the VarDeclaredNames of *Statement*.

NOTE

This section is extended by Annex [B.3.6](#).

13.7.5.8 Static Semantics: VarScopedDeclarations

IterationStatement : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement*

1. Return the VarScopedDeclarations of *Statement*.

IterationStatement : **for** (**var** *ForBinding* **in** *Expression*) *Statement*

1. Let *declarations* be a List containing *ForBinding*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *Statement*.
3. Return *declarations*.

3. For each element *name* of the BoundNames of *ForBinding*, do
 - a. If IsConstantDeclaration of *LetOrConst* is **true**, then
 - i. Perform ! *envRec*.CreateImmutableBinding(*name*, **true**).
 - b. Else,
 - i. Perform ! *envRec*.CreateMutableBinding(*name*, **false**).

13.7.5.11 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

IterationStatement : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *Expression*, enumerate).
2. Return ? *ForIn/OfBodyEvaluation*(*LeftHandSideExpression*, *Statement*, *keyResult*, enumerate, assignment, *labelSet*).

IterationStatement : **for** (**var** *ForBinding* **in** *Expression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *Expression*, enumerate).
2. Return ? *ForIn/OfBodyEvaluation*(*ForBinding*, *Statement*, *keyResult*, enumerate, varBinding, *labelSet*).

IterationStatement : **for** (*ForDeclaration* **in** *Expression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(BoundNames of *ForDeclaration*, *Expression*, enumerate).
2. Return ? *ForIn/OfBodyEvaluation*(*ForDeclaration*, *Statement*, *keyResult*, enumerate, lexicalBinding, *labelSet*).

IterationStatement : **for** (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *AssignmentExpression*, iterate).
2. Return ? *ForIn/OfBodyEvaluation*(*LeftHandSideExpression*, *Statement*, *keyResult*, iterate, assignment, *labelSet*).

IterationStatement : **for** (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *AssignmentExpression*, iterate).
2. Return ? *ForIn/OfBodyEvaluation*(*ForBinding*, *Statement*, *keyResult*, iterate, varBinding, *labelSet*).

IterationStatement : **for** (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(BoundNames of *ForDeclaration*, *AssignmentExpression*, iterate).
2. Return ? *ForIn/OfBodyEvaluation*(*ForDeclaration*, *Statement*, *keyResult*, iterate, lexicalBinding, *labelSet*).

IterationStatement : **for await** (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *AssignmentExpression*, async-iterate).
2. Return ? *ForIn/OfBodyEvaluation*(*LeftHandSideExpression*, *Statement*, *keyResult*, iterate, assignment, *labelSet*, *async*).

IterationStatement : **for await** (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

1. Let *keyResult* be ? *ForIn/OfHeadEvaluation*(« », *AssignmentExpression*, async-iterate).
2. Return ? *ForIn/OfBodyEvaluation*(*ForBinding*, *Statement*, *keyResult*, iterate, varBinding, *labelSet*, *async*).

IterationStatement : **for await** (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

- a. Let *nextResult* be ? Call(*iteratorRecord*.[[NextMethod]], *iteratorRecord*.[[Iterator]], « »).
- b. If *iteratorKind* is **async**, then set *nextResult* to ? Await(*nextResult*).
- c. If Type(*nextResult*) is not Object, throw a **TypeError** exception.
- d. Let *done* be ? IteratorComplete(*nextResult*).
- e. If *done* is **true**, return NormalCompletion(*V*).
- f. Let *nextValue* be ? IteratorValue(*nextResult*).
- g. If *lhsKind* is either **assignment** or **varBinding**, then
 - i. If *destructuring* is **false**, then
 1. Let *lhsRef* be the result of evaluating *lhs*. (It may be evaluated repeatedly.)
- h. Else,
 - i. **Assert:** *lhsKind* is **lexicalBinding**.
 - ii. **Assert:** *lhs* is a *ForDeclaration*.
 - iii. Let *iterationEnv* be NewDeclarativeEnvironment(*oldEnv*).
 - iv. Perform BindingInstantiation for *lhs* passing *iterationEnv* as the argument.
 - v. Set the **running execution context's** LexicalEnvironment to *iterationEnv*.
 - vi. If *destructuring* is **false**, then
 1. **Assert:** *lhs* binds a single name.
 2. Let *lhsName* be the sole element of BoundNames of *lhs*.
 3. Let *lhsRef* be ! ResolveBinding(*lhsName*).
- i. If *destructuring* is **false**, then
 - i. If *lhsRef* is an abrupt completion, then
 1. Let *status* be *lhsRef*.
 - ii. Else if *lhsKind* is **lexicalBinding**, then
 1. Let *status* be InitializeReferencedBinding(*lhsRef*, *nextValue*).
 - iii. Else,
 1. Let *status* be PutValue(*lhsRef*, *nextValue*).
- j. Else,
 - i. If *lhsKind* is **assignment**, then
 1. Let *status* be the result of performing DestructuringAssignmentEvaluation of *assignmentPattern* using *nextValue* as the argument.
 - ii. Else if *lhsKind* is **varBinding**, then
 1. **Assert:** *lhs* is a *ForBinding*.
 2. Let *status* be the result of performing BindingInitialization for *lhs* passing *nextValue* and **undefined** as the arguments.
 - iii. Else,
 1. **Assert:** *lhsKind* is **lexicalBinding**.
 2. **Assert:** *lhs* is a *ForDeclaration*.
 3. Let *status* be the result of performing BindingInitialization for *lhs* passing *nextValue* and *iterationEnv* as arguments.
- k. If *status* is an abrupt completion, then
 - i. Set the **running execution context's** LexicalEnvironment to *oldEnv*.
 - ii. If *iteratorKind* is **async**, return ? AsyncIteratorClose(*iteratorRecord*, *status*).
 - iii. If *iterationKind* is **enumerate**, then
 1. Return *status*.
 - iv. Else,
 1. **Assert:** *iterationKind* is **iterate**.
 2. Return ? IteratorClose(*iteratorRecord*, *status*).
- l. Let *result* be the result of evaluating *stmt*.


```

if (desc) {
    visited.add(key);
    if (desc.enumerable) yield key;
}
}

const proto = Reflect.getPrototypeOf(obj);
if (proto === null) return;
for (const protoKey of EnumerateObjectProperties(proto)) {
    if (!visited.has(protoKey)) yield protoKey;
}
}

```

13.8 The `continue` Statement

Syntax

ContinueStatement [Yield, Await] :
continue ;
continue [no LineTerminator here] *LabelIdentifier* [?Yield, ?Await] ;

13.8.1 Static Semantics: Early Errors

ContinueStatement : **continue** ;
ContinueStatement : **continue** *LabelIdentifier* ;

It is a Syntax Error if this *ContinueStatement* is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.

13.8.2 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

ContinueStatement : **continue** ;

1. Return **false**.

ContinueStatement : **continue** *LabelIdentifier* ;

1. If the StringValue of *LabelIdentifier* is not an element of *iterationSet*, return **true**.
2. Return **false**.

13.8.3 Runtime Semantics: Evaluation

ContinueStatement : **continue** ;

1. Return **Completion** { [[Type]]: `continue`, [[Value]]: empty, [[Target]]: empty }.

ContinueStatement : **continue** *LabelIdentifier* ;

NOTE

A **return** statement causes a function to cease execution and, in most cases, returns a value to the caller. If *Expression* is omitted, the return value is **undefined**. Otherwise, the return value is the value of *Expression*. A **return** statement may not actually return a value to the caller depending on surrounding context. For example, in a **try** block, a **return** statement's completion record may be replaced with another completion record during evaluation of the **finally** block.

13.10.1 Runtime Semantics: Evaluation

ReturnStatement : **return** ;

1. Return **Completion** { [[Type]]: **return**, [[Value]]: **undefined**, [[Target]]: empty }.

ReturnStatement : **return** *Expression* ;

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ? **GetValue**(*exprRef*).
3. If ! **GetGeneratorKind**() is **async**, set *exprValue* to ? **Await**(*exprValue*).
4. Return **Completion** { [[Type]]: **return**, [[Value]]: *exprValue*, [[Target]]: empty }.

13.11 The **with** Statement

Syntax

```
WithStatement[Yield, Await, Return] :  
  with ( Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
```

NOTE

The **with** statement adds an object **Environment Record** for a computed object to the lexical environment of the **running execution context**. It then executes a statement using this augmented lexical environment. Finally, it restores the original lexical environment.

13.11.1 Static Semantics: Early Errors

WithStatement : **with** (*Expression*) *Statement*

It is a Syntax Error if the code that matches this production is contained in **strict mode code**.

It is a Syntax Error if **IsLabelledFunction**(*Statement*) is **true**.

NOTE

It is only necessary to apply the second rule if the extension specified in **B.3.2** is implemented.

13.11.2 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

WithStatement : **with** (*Expression*) *Statement*

Syntax

```
SwitchStatement[Yield, Await, Return] :
  switch ( Expression[+In, ?Yield, ?Await] ) CaseBlock[?Yield, ?Await, ?Return]

CaseBlock[Yield, Await, Return] :
  { CaseClauses[?Yield, ?Await, ?Return] opt }
  { CaseClauses[?Yield, ?Await, ?Return] opt DefaultClause[?Yield, ?Await, ?Return]
    CaseClauses[?Yield, ?Await, ?Return] opt }

CaseClauses[Yield, Await, Return] :
  CaseClause[?Yield, ?Await, ?Return]
  CaseClauses[?Yield, ?Await, ?Return] CaseClause[?Yield, ?Await, ?Return]

CaseClause[Yield, Await, Return] :
  case Expression[+In, ?Yield, ?Await] : StatementList[?Yield, ?Await, ?Return] opt

DefaultClause[Yield, Await, Return] :
  default : StatementList[?Yield, ?Await, ?Return] opt
```

13.12.1 Static Semantics: Early Errors

SwitchStatement : **switch** (Expression) CaseBlock

It is a Syntax Error if the LexicallyDeclaredNames of CaseBlock contains any duplicate entries.
It is a Syntax Error if any element of the LexicallyDeclaredNames of CaseBlock also occurs in the VarDeclaredNames of CaseBlock.

13.12.2 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

SwitchStatement : **switch** (Expression) CaseBlock

1. Return ContainsDuplicateLabels of CaseBlock with argument *labelSet*.

CaseBlock : { }

1. Return **false**.

CaseBlock : { CaseClauses DefaultClause CaseClauses }

1. If the first CaseClauses is present, then
 - a. Let *hasDuplicates* be ContainsDuplicateLabels of the first CaseClauses with argument *labelSet*.
 - b. If *hasDuplicates* is **true**, return **true**.
2. Let *hasDuplicates* be ContainsDuplicateLabels of DefaultClause with argument *labelSet*.
3. If *hasDuplicates* is **true**, return **true**.
4. If the second CaseClauses is not present, return **false**.
5. Return ContainsDuplicateLabels of the second CaseClauses with argument *labelSet*.

CaseClauses : CaseClauses CaseClause

13.12.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return ContainsUndefinedContinueTarget of *CaseBlock* with arguments *iterationSet* and « ».

CaseBlock : { }

1. Return **false**.

CaseBlock : { *CaseClauses* *DefaultClause* *CaseClauses* }

1. If the first *CaseClauses* is present, then
 - a. Let *hasUndefinedLabels* be ContainsUndefinedContinueTarget of the first *CaseClauses* with arguments *iterationSet* and « ».
 - b. If *hasUndefinedLabels* is **true**, return **true**.
2. Let *hasUndefinedLabels* be ContainsUndefinedContinueTarget of *DefaultClause* with arguments *iterationSet* and « ».
3. If *hasUndefinedLabels* is **true**, return **true**.
4. If the second *CaseClauses* is not present, return **false**.
5. Return ContainsUndefinedContinueTarget of the second *CaseClauses* with arguments *iterationSet* and « ».

CaseClauses : *CaseClauses* *CaseClause*

1. Let *hasUndefinedLabels* be ContainsUndefinedContinueTarget of *CaseClauses* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return ContainsUndefinedContinueTarget of *CaseClause* with arguments *iterationSet* and « ».

CaseClause : **case** *Expression* : *StatementList*

1. If the *StatementList* is present, return ContainsUndefinedContinueTarget of *StatementList* with arguments *iterationSet* and « ».
2. Return **false**.

DefaultClause : **default** : *StatementList*

1. If the *StatementList* is present, return ContainsUndefinedContinueTarget of *StatementList* with arguments *iterationSet* and « ».
2. Return **false**.

13.12.5 Static Semantics: LexicallyDeclaredNames

CaseBlock : { }

1. Return a new empty *List*.

CaseBlock : { *CaseClauses* *DefaultClause* *CaseClauses* }

1. If the first *CaseClauses* is present, let *names* be the LexicallyDeclaredNames of the first *CaseClauses*.
2. Else, let *names* be a new empty *List*.
3. Append to *names* the elements of the LexicallyDeclaredNames of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.

CaseBlock : { }

1. Return a new empty List.

CaseBlock : { *CaseClauses DefaultClause CaseClauses* }

1. If the first *CaseClauses* is present, let *names* be the VarDeclaredNames of the first *CaseClauses*.
2. Else, let *names* be a new empty List.
3. Append to *names* the elements of the VarDeclaredNames of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Return the result of appending to *names* the elements of the VarDeclaredNames of the second *CaseClauses*.

CaseClauses : *CaseClauses CaseClause*

1. Let *names* be VarDeclaredNames of *CaseClauses*.
2. Append to *names* the elements of the VarDeclaredNames of *CaseClause*.
3. Return *names*.

CaseClause : **case** *Expression* : *StatementList*

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Return a new empty List.

DefaultClause : **default** : *StatementList*

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Return a new empty List.

13.12.8 Static Semantics: VarScopedDeclarations

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return the VarScopedDeclarations of *CaseBlock*.

CaseBlock : { }

1. Return a new empty List.

CaseBlock : { *CaseClauses DefaultClause CaseClauses* }

1. If the first *CaseClauses* is present, let *declarations* be the VarScopedDeclarations of the first *CaseClauses*.
2. Else, let *declarations* be a new empty List.
3. Append to *declarations* the elements of the VarScopedDeclarations of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *declarations*.
5. Return the result of appending to *declarations* the elements of the VarScopedDeclarations of the second *CaseClauses*.

CaseClauses : *CaseClauses CaseClause*

1. Let *declarations* be VarScopedDeclarations of *CaseClauses*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *CaseClause*.
3. Return *declarations*.

CaseClause : **case** *Expression* : *StatementList*

9. If *found* is **false**, then
 - a. For each *CaseClause* *C* in *B*, do
 - i. If *foundInB* is **false**, then
 1. Set *foundInB* to ? *CaseClauseIsSelected*(*C*, *input*).
 - ii. If *foundInB* is **true**, then
 1. Let *R* be the result of evaluating *CaseClause* *C*.
 2. If *R*.[[Value]] is not empty, set *V* to *R*.[[Value]].
 3. If *R* is an abrupt completion, return *Completion*(*UpdateEmpty*(*R*, *V*)).
10. If *foundInB* is **true**, return *NormalCompletion*(*V*).
11. Let *R* be the result of evaluating *DefaultClause*.
12. If *R*.[[Value]] is not empty, set *V* to *R*.[[Value]].
13. If *R* is an abrupt completion, return *Completion*(*UpdateEmpty*(*R*, *V*)).
14. For each *CaseClause* *C* in *B* (NOTE: this is another complete iteration of the second *CaseClauses*), do
 - a. Let *R* be the result of evaluating *CaseClause* *C*.
 - b. If *R*.[[Value]] is not empty, set *V* to *R*.[[Value]].
 - c. If *R* is an abrupt completion, return *Completion*(*UpdateEmpty*(*R*, *V*)).
15. Return *NormalCompletion*(*V*).

13.12.10 Runtime Semantics: *CaseClauseIsSelected* (*C*, *input*)

The abstract operation *CaseClauseIsSelected*, given *CaseClause* *C* and value *input*, determines whether *C* matches *input*.

1. **Assert:** *C* is an instance of the production *CaseClause* : **case** *Expression* : *StatementList* .
2. Let *exprRef* be the result of evaluating the *Expression* of *C*.
3. Let *clauseSelector* be ? *GetValue*(*exprRef*).
4. Return the result of performing **Strict Equality Comparison** *input* === *clauseSelector*.

NOTE

This operation does not execute *C*'s *StatementList* (if any). The *CaseBlock* algorithm uses its return value to determine which *StatementList* to start executing.

13.12.11 Runtime Semantics: Evaluation

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *switchValue* be ? *GetValue*(*exprRef*).
3. Let *oldEnv* be the **running execution context**'s LexicalEnvironment.
4. Let *blockEnv* be *NewDeclarativeEnvironment*(*oldEnv*).
5. Perform *BlockDeclarationInstantiation*(*CaseBlock*, *blockEnv*).
6. Set the **running execution context**'s LexicalEnvironment to *blockEnv*.
7. Let *R* be the result of performing *CaseBlockEvaluation* of *CaseBlock* with argument *switchValue*.
8. Set the **running execution context**'s LexicalEnvironment to *oldEnv*.
9. Return *R*.

NOTE

No matter how control leaves the *SwitchStatement* the LexicalEnvironment is always restored to its former state.

CaseClause : **case** *Expression* :

4. Return ContainsDuplicateLabels of *LabelledItem* with argument *newLabelSet*.

LabelledItem : *FunctionDeclaration*

1. Return **false**.

13.13.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter *labelSet*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the StringValue of *LabelIdentifier*.
2. Let *newLabelSet* be a copy of *labelSet* with *label* appended.
3. Return ContainsUndefinedBreakTarget of *LabelledItem* with argument *newLabelSet*.

LabelledItem : *FunctionDeclaration*

1. Return **false**.

13.13.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters *iterationSet* and *labelSet*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the StringValue of *LabelIdentifier*.
2. Let *newLabelSet* be a copy of *labelSet* with *label* appended.
3. Return ContainsUndefinedContinueTarget of *LabelledItem* with arguments *iterationSet* and *newLabelSet*.

LabelledItem : *FunctionDeclaration*

1. Return **false**.

13.13.5 Static Semantics: IsLabelledFunction (*stmt*)

The abstract operation IsLabelledFunction with argument *stmt* performs the following steps:

1. If *stmt* is not a *LabelledStatement*, return **false**.
2. Let *item* be the *LabelledItem* of *stmt*.
3. If *item* is *LabelledItem* : *FunctionDeclaration* , return **true**.
4. Let *subStmt* be the *Statement* of *item*.
5. Return IsLabelledFunction(*subStmt*).

13.13.6 Static Semantics: LexicallyDeclaredNames

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the LexicallyDeclaredNames of *LabelledItem*.

LabelledItem : *Statement*

1. Return a new empty *List*.

LabelledItem : *FunctionDeclaration*

1. Return a new *List* containing *FunctionDeclaration*.

13.13.12 Static Semantics: VarDeclaredNames

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the VarDeclaredNames of *LabelledItem*.

LabelledItem : *FunctionDeclaration*

1. Return a new empty *List*.

13.13.13 Static Semantics: VarScopedDeclarations

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the VarScopedDeclarations of *LabelledItem*.

LabelledItem : *FunctionDeclaration*

1. Return a new empty *List*.

13.13.14 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the StringValue of *LabelIdentifier*.
2. Append *label* as an element of *labelSet*.
3. Let *stmtResult* be LabelledEvaluation of *LabelledItem* with argument *labelSet*.
4. If *stmtResult*.[[Type]] is *break* and SameValue(*stmtResult*.[[Target]], *label*) is **true**, then
 - a. Set *stmtResult* to NormalCompletion(*stmtResult*.[[Value]]).
5. Return Completion(*stmtResult*).

LabelledItem : *Statement*

1. If *Statement* is either a *LabelledStatement* or a *BreakableStatement*, then
 - a. Return LabelledEvaluation of *Statement* with argument *labelSet*.
2. Else,
 - a. Return the result of evaluating *Statement*.

LabelledItem : *FunctionDeclaration*

1. Return the result of evaluating *FunctionDeclaration*.

13.13.15 Runtime Semantics: Evaluation

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *newLabelSet* be a new empty *List*.
2. Return LabelledEvaluation of this *LabelledStatement* with argument *newLabelSet*.

It is a Syntax Error if any element of the BoundNames of *CatchParameter* also occurs in the VarDeclaredNames of *Block*.

NOTE

An alternative [static semantics](#) for this production is given in [B.3.5](#).

13.15.2 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

TryStatement : **try** *Block* *Catch*

1. Let *hasDuplicates* be ContainsDuplicateLabels of *Block* with argument *labelSet*.
2. If *hasDuplicates* is **true**, return **true**.
3. Return ContainsDuplicateLabels of *Catch* with argument *labelSet*.

TryStatement : **try** *Block* *Finally*

1. Let *hasDuplicates* be ContainsDuplicateLabels of *Block* with argument *labelSet*.
2. If *hasDuplicates* is **true**, return **true**.
3. Return ContainsDuplicateLabels of *Finally* with argument *labelSet*.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *hasDuplicates* be ContainsDuplicateLabels of *Block* with argument *labelSet*.
2. If *hasDuplicates* is **true**, return **true**.
3. Let *hasDuplicates* be ContainsDuplicateLabels of *Catch* with argument *labelSet*.
4. If *hasDuplicates* is **true**, return **true**.
5. Return ContainsDuplicateLabels of *Finally* with argument *labelSet*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return ContainsDuplicateLabels of *Block* with argument *labelSet*.

13.15.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter *labelSet*.

TryStatement : **try** *Block* *Catch*

1. Let *hasUndefinedLabels* be ContainsUndefinedBreakTarget of *Block* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return ContainsUndefinedBreakTarget of *Catch* with argument *labelSet*.

TryStatement : **try** *Block* *Finally*

1. Let *hasUndefinedLabels* be ContainsUndefinedBreakTarget of *Block* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return ContainsUndefinedBreakTarget of *Finally* with argument *labelSet*.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Catch*.
3. Append to *names* the elements of the VarDeclaredNames of *Finally*.
4. Return *names*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return the VarDeclaredNames of *Block*.

13.15.6 Static Semantics: VarScopedDeclarations

TryStatement : **try** *Block* *Catch*

1. Let *declarations* be VarScopedDeclarations of *Block*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *Catch*.
3. Return *declarations*.

TryStatement : **try** *Block* *Finally*

1. Let *declarations* be VarScopedDeclarations of *Block*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *Finally*.
3. Return *declarations*.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *declarations* be VarScopedDeclarations of *Block*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *Catch*.
3. Append to *declarations* the elements of the VarScopedDeclarations of *Finally*.
4. Return *declarations*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return the VarScopedDeclarations of *Block*.

13.15.7 Runtime Semantics: CatchClauseEvaluation

With parameter *thrownValue*.

Catch : **catch** (*CatchParameter*) *Block*

1. Let *oldEnv* be the *running execution context*'s LexicalEnvironment.
2. Let *catchEnv* be *NewDeclarativeEnvironment*(*oldEnv*).
3. Let *catchEnvRec* be *catchEnv*'s EnvironmentRecord.
4. For each element *argName* of the BoundNames of *CatchParameter*, do
 - a. Perform ! *catchEnvRec*.CreateMutableBinding(*argName*, *false*).
5. Set the *running execution context*'s LexicalEnvironment to *catchEnv*.
6. Let *status* be the result of performing BindingInitialization for *CatchParameter* passing *thrownValue* and *catchEnv* as arguments.
7. If *status* is an abrupt completion, then
 - a. Set the *running execution context*'s LexicalEnvironment to *oldEnv*.
 - b. Return *Completion*(*status*).
8. Let *B* be the result of evaluating *Block*.

1. If an implementation-defined debugging facility is available and enabled, then
 - a. Perform an implementation-defined debugging action.
 - b. Let *result* be an implementation-defined **Completion** value.
2. Else,
 - a. Let *result* be **NormalCompletion**(empty).
3. Return *result*.

14 ECMAScript Language: Functions and Classes

NOTE

Various ECMAScript language elements cause the creation of ECMAScript function objects (9.2). Evaluation of such functions starts with the execution of their [[Call]] internal method (9.2.1).

14.1 Function Definitions

Syntax

```

FunctionDeclaration [Yield, Await, Default] :
  function BindingIdentifier [?Yield, ?Await] ( FormalParameters [~Yield, ~Await] ) {
    FunctionBody [~Yield, ~Await]
  }
  [+Default] function ( FormalParameters [~Yield, ~Await] ) { FunctionBody [~Yield, ~Await] }

FunctionExpression :
  function BindingIdentifier [~Yield, ~Await] opt ( FormalParameters [~Yield, ~Await] ) {
    FunctionBody [~Yield, ~Await]
  }

UniqueFormalParameters [Yield, Await] :
  FormalParameters [?Yield, ?Await]

FormalParameters [Yield, Await] :
  [empty]
  FunctionRestParameter [?Yield, ?Await]
  FormalParameterList [?Yield, ?Await]
  FormalParameterList [?Yield, ?Await] ,
  FormalParameterList [?Yield, ?Await] , FunctionRestParameter [?Yield, ?Await]

FormalParameterList [Yield, Await] :
  FormalParameter [?Yield, ?Await]
  FormalParameterList [?Yield, ?Await] , FormalParameter [?Yield, ?Await]

FunctionRestParameter [Yield, Await] :
  BindingRestElement [?Yield, ?Await]

FormalParameter [Yield, Await] :
  BindingElement [?Yield, ?Await]

```


UniqueFormalParameters : *FormalParameters*

It is a Syntax Error if *BoundNames* of *FormalParameters* contains any duplicate elements.

FormalParameters : *FormalParameterList*

It is a Syntax Error if *IsSimpleParameterList* of *FormalParameterList* is **false** and *BoundNames* of *FormalParameterList* contains any duplicate elements.

NOTE 2

Multiple occurrences of the same *BindingIdentifier* in a *FormalParameterList* is only allowed for functions which have simple parameter lists and which are not defined in [strict mode code](#).

FunctionBody : *FunctionStatementList*

It is a Syntax Error if the *LexicallyDeclaredNames* of *FunctionStatementList* contains any duplicate entries.

It is a Syntax Error if any element of the *LexicallyDeclaredNames* of *FunctionStatementList* also occurs in the *VarDeclaredNames* of *FunctionStatementList*.

It is a Syntax Error if *ContainsDuplicateLabels* of *FunctionStatementList* with argument « » is **true**.

It is a Syntax Error if *ContainsUndefinedBreakTarget* of *FunctionStatementList* with argument « » is **true**.

It is a Syntax Error if *ContainsUndefinedContinueTarget* of *FunctionStatementList* with arguments « » and « » is **true**.

14.1.3 Static Semantics: BoundNames

FunctionDeclaration : **function** *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return the *BoundNames* of *BindingIdentifier*.

FunctionDeclaration : **function** (*FormalParameters*) { *FunctionBody* }

1. Return « ***default*** ».

NOTE

« ***default*** » is used within this specification as a synthetic name for hoistable anonymous functions that are defined using export declarations.

FormalParameters : [empty]

1. Return a new empty [List](#).

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. Let *names* be *BoundNames* of *FormalParameterList*.
2. Append to *names* the *BoundNames* of *FunctionRestParameter*.
3. Return *names*.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. Let *names* be *BoundNames* of *FormalParameterList*.
2. Append to *names* the *BoundNames* of *FormalParameter*.
3. Return *names*.

