0. INTRODUCTION

La especificación, de junio del 2017 es la última publicada a la fecha del comienzo de esto apuntes (mayo 2018). Los apuntes estaràn en una mezcla de inglés y español conforme voy acostumbrándome a escribir y pensar en inglés.

El estándar ECMA-262 define el estandar del lenguaje **ECMAScript 2017**.
Es la octava edición a cargo de ECMA. La 1º publicación fue en 1997. ECMAScript está basado en varias tecnologías, algunas muy conocidas como JavaScript (Netscape) y JScript (Microsoft).

El lenguaje fue inventado por **Brendan Eich** en Netscape y apareció por primera vez en el browser de la compañía "*Navigator 2.0*" y en todos los sucesivos además de estar incluido en todos los browsers de Microsoft a partir de "*Explorer 3.0*".

El desarrollo de la especificación del estandar para ECMAScript comenzó en **noviembre de 1996**. La **1º edición** fue publicada en **junio de 1997**. Ese estándar fue enviado como "ISO/IEC JTC 1" para la adopción bajo el fast-track procedure, y fue aprobado como estándar internacional en **abril de 1998** bajo el código ISO/IEC 16262. La **2º edición** fue aprobada en **junio de 1998** con cambios de naturaleza editorial.
La **3º edición** introdujo expresiones regulares, mejor manejo de cadenas, nuevas declaraciones de control, manejo de excepciones "*try/catch*", definiciones de errores más ajustadas, formateo de salidas numéricas, y cambios menores anticipádose al futuro crecimiento del lenguaje. Esta edición fue aprobada en **diciembre de 1999** y publicada bajo "*ISO/IEC 16262:2002*" en **junio de 2002**.

Luego de su publicación, se adoptó masivamente in conjunto con el consorcio de la "*World Wide Web*" donde fue adoptado como el lenguaje de programación soportado por todos los browsers. Aunque se hizo bastante trabajo para una **4º edición** esta no vio la luz. La **5º edición** fue enviada aprobada como estandar internacional bajo **ISO/IEC 16262:2011**. La **edición 5.1** fue aprobada en asamblea general de ECMA en **junio de 2011**.

El trabajo para la preparación de la **6º edición** comenzó en **2009** mientras la 5º edición era preparada para ser publicada. Sin embargo, este trabajo fue precedido por una significativa experimentación y esfuerzos de diseño enfocados al aumento del lenguaje que datan incluso de la fecha de publicación de la 3º edición. En cierto sentido, la 6º edición culmina el trabajo de 15 años. Los objetivos de esta edición incluyen proveer mejor soporte para grandes aplicaciones, creación de librerías, y el uso de este como el objeto de compilacion de otros lenguajes. Algunos de sus mayores mejoras incluyen *modulos, declaraciones de clases, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tails calls. La librería de *built-ins* de ECMAScript fue expandida para soportar abstracciones de datos adicionales como *maps, sets, and arrays of binary numeric values* asi como soporte adicional para caracteres Unicode. Estos built-ins se hicieron extensibles via *subclasses*. Esta edición provee la base de regulares incrementos del lenguaje y sus librerías. La **6º edición** (ES6) fue adoptada por la asamblea en **junio del 2015** (ES6 or ECMA2015).

ECMAScript 2016 fue la primera edición en ser lanzada bajo el nueva cadencia anual y desarrollo abierto "*Ecma TC'39*". Un documento fuente de texto plano fue elaborado desde ECMAScript 2015 que sirvió de base para el futuro desarrollo enteramente en GitHub. Durante el año de desarrollo del estándar, cientos de *issues* y *pull request* fueron archivados representando miles de correcciones de *bugs*, editoriales y otras mejoras. Adicionalmente, numerosas herramientas de software fueron desarrolladas para ayudar en este esfuerzo, incluyendo **Ecmarkup**, **Ecmarkdown**, y **Grammarkdown**. ES2016 también incluye soporte para un nuevo operador de exponenciación y añade un nuevo método a **Array.prototype** llamado '*includes*'.

Esta especificación introduce '*Async Functions, Shared Memory y Atomics*' junto a mejoras menores del lenguaje y librería, correción de *bugs* y actualizaciones editoriales. *Async funtions* mejoran la experiencia de programación asíncrona al proveer sintaxis para funciones *promise-returning*. *Shared Memory y Atomics* introducen un nuevo modelo de memoria que permite a programas *multiagente* comunicarse usando operaciones atómicas que aseguran una orden de ejecución bien definido incluso en CPUs paralelas. Esta especificación también incluye nuevos métodos estáticos en **Object:**

```
Object.values
Object.entries // and
Object.getOwnPropertyDescriptors
```

Decenas de personas representando a numerosas organizaciones han hecho significativas contribuciones dentro de Ecma TC39 para el desarrollo de esta edición y anteriores. Esta comunidad a revisado numerosos borradores, archivado miles de reportes de errores (*bugs*), ejecutado implementaciones experimentales, contribuido con suites de testeo y educado a la comunidad global de desarrolladores acerca de ECMAScript. Desafortunadamente, es imposible identificar y agradecer a cada persona y organización que ha contruibuido a este esfuerzo.

> **Allen Wirfs-Brock**, ECMA-262 6th Edition Project Editor
> **Brian Terlson**, ECMA-262 7th Edition Project Editor

1. Scope

Este estandar define el lenguaje de programación de próposito general **ECMAScript 2017**.

2. Conformance

- Una implementación conforme de ECMAScript debe ser provista y soportar todos los tipos, valores, objetos, propiedades, funciones y síntaxis y semántica de programación descritas en esta especificación.
- Una implementación conforme de ECMAScript debe interpretar una entrada de texto fuente en acuerdo con la última versión del estándar **Unicode** y el **ISO/IEC 10646**.
- Una implementación conforme de ECMAScript que provea una interfaz de programación y que soporte programas que necesiten adaptarse a las convenciones culturales y de lenguaje usadas por los diferentes idiomas y países del mundo debe implementar una interfaz definida en la más reciente edition de ECMA-402, que es compatible con los requerimientos de esta especificación.
- Una implementación conforme de ECMAScript puede proveer tipos, valores, objetos, propiedades, y funciones adicionales a aquellas descritas en esta especificación, y valores para esas propiedades y objetos adicionales a los descritos en esta especificación.
- Una implementación conforme de ECMAScript puede soportar sintaxis de programación y de expresiones regulares no descritas en esta especificación. En particular, una implementación conforme de ECMAScript puede soportar sintaxis de programación que hace uso de "**futuras palabras reservadas**" de aquellas listadas en la subcláusula 11.6.2.2 de esta misma especificación.
- Una implementación conforme de ECMAScript no debe implementar ninguna extensión de las listadas en la subcláusula 16.2 como "**Extensiones prohibidas**".

3. Normative References

Los siguientes documentos referenciados son indispensables para la aplicación de este documento. Para referencias fechadas, solo la referencia citada aplica. Para referencias no fechadas, la última edicion del citado documento -y todas sus correciones- es el que aplica.

- ISO/IEC 10646:2003: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005, Amendment 2:2006, Amendment 3:2008, and Amendment 4:2008, plus additional amendments and corrigenda, or successor
- ECMA-402, ECMAScript 2015 Internationalization API Specification.
- ECMA-404, The JSON Data Interchange Format.

4. Overview

Esta sección contiene un panorama no-normativo del lenguaje ECMAScript.
ECMAScript es un lenguaje de programación **orientado a objetos** para realizar computación y manipulación de objetos computacionales dentro de un entorno anfitrión. ECMAScript, tal como está definido aquí, no tiene la intención de ser computacionalmente autosuficiente. De hecho, no hay indicaciones en esta especificación para la entrada de datos externos, o la salida de resultados computados. En vez de eso, se espera que le entorno computacional de un programa ECMAScript provea no

solo los objetos y recursos descritos en esta especificación sino también ciertos objetos específicos del entorno, cuya descripción y comportamiento están más allá del alcance de esta especificación excepto para indicar que ellos podrían proveer ciertas propiedades que deberían ser accesibles, y ciertas funciones que pueden ser llamadas desde un programa ECMAScript.

ECMAScript fue originalmente diseñado para ser usado como un lenguaje de *scripting* pero ha llegado a ser un lenguaje de programación de propósito general. Un lenguaje de *scripting* es un lenguaje de programación usado para manipular, personalizar y automatizar los recursos de un sistema determinado. En aquellos sistemas, la funcionalidad básica ya está disponible a traves de una interfaz de usuario, y un lenguaje de *scripting* es un mecanismo para exponer esas funcionalidades a un programa de control. En este sentido, el sistema mencionado, se dice que provee el 'entorno anfitrión' de objetos y recursos, que completan las capacidades del lenguaje de *scripting*. Un lenguaje de este tipo está pensado para ser usado tanto por programadores profesionales, como por aquellos que no lo son.

ECMAScript fue originalmente diseñado para ser usado como un lenguaje de "***scripting Web***", proveyendo un mecanismo para animar páginas web en navegadores, y para ejecutar computacion de servidor, como parte de un modelo *Web-based client-server architecture*. (Modelo de arquitectura de cliente servidor basado en web). ECMAScript hoy es usado para proveer capacidades de '*core scripting*' para una variedad de ambientes anfitriones. Por lo mismo, el núcleo del lenguaje es especificado aquí fuera de cualqueir ambiente anfitrión particular.

El uso de ECMAScript se ha movido más allá del simple *scripting* y ahora es utilizado para un amplio espectro de tareas de programación en muchos ambientes y escala. Asi como el uso de ECMAScript se expande, también lo hacen las características y recursos que este entrega. ECMAScript es ahora un lenguaje de programación de propósito general pleno de caracteríticas y funcionalidades.

Algunos de los recursos de ECMAScript son similares a aquellos utilizados en otros lenguajes de programación, en particular ***C, Java, Self y Scheme*** como se describen en:

> - 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 Procededings*, pp. 227-241, Orlando, FL, October 1987.
> - *IEEE Standard for the Scheme Programming Language*. IEEE Std 1178-1990.

4.1 Web Scripting

Un **navegador web** (*web browser*) otorga un entorno anfitrión a ECMAScript par la computación de lado del cliente (*client-side*) incluyendo, por ejemplo, objetos que representan ventanas, menus, pop-ups, cajas de diálogos, áreas de texto, enlaces, *frames*, historial, *cookies*, y entrada/salida. Además, ese entorno anfitrión provee un manera de asociar código de *scripting* a eventos como el cambio de foco, la carga de páginas e imágenes, descargas, error y cancelación, selección, envío de formularios, y acciones del mouse. El código (*de scripting*) aparece dentro del HTML y la página visualizada es una combinació de elementos de la interfaz de usuario y texto e imágenes fijas y dinámicas (*computed*). El *scripting code* reacciona a la interacción del usuario y no hay necesidad de un programa principal.

Un servidor web provee entorno anfitrión distinto para la computación de elementos de lado del servidor (*server-side*) incluyendo objetos que representan requests, clientes y archivos; y mecanismoa para bloquear y compartir datos. Al utilizar programación de scripting *browser-side* y *server-side* juntas, es posible distribuir las necesidades de computación (*tareas, rutinas, cálculos, etc*) entre el cliente y el servidor a través de una aplicación *Web-based* con una interfaz de usuario personalizada para esa aplicación.

Cada servidor web que soporta ECMAScript provee su propio entorno anfitrión, completando el entorno de ejecución ECMAScript.

4.2 ECMAScript Overview

La siguiente es una vista general informal de ECMAScript -no todas las partes del lenguaje son descritas. Este panorama no es parte del estándar propiamente tal.

> ECMAScript es un lenguaje orientado a objetos: los elementos básicos del lenguaje y sus recursos son provistos por objetos, de tal manera que un programa ECMAScript es un *cluster* (aglutinación) de objetos comunicantes, y en ECMAScript, un *objeto* es una colección de cero o más ***propiedades*** cada una con ***atributos*** que determinan como aquella propiedad puede ser usada.

Por ejemplo, cuando el atributo `Writable` para una propiedad es *seteado* (configurado) a `false` cualquier intento de ejecutar código ECMAScript que asigne un valor diferente a esa propiedad va a fallar. Las *propiedades* son **CONTAINERS** que guardan: *otros objetos, valores primitivos o funciones*.

Un ***valor primitivo*** es un miembro de uno de los siguientes **tipos integrados**:

- Undefined
- Null
- Boolean
- Number
- String
- Symbol

Por su parte, cualquier objeto será miembro del **tipo integrado** "**Object**", mientras que una ***función*** es un "objeto *callable*" (llamable), y por último una función que es asociada con un objeto a través de una propiedad se llama "**method** (método):

- objeto -> `Object`
- objeto callable -> `function`
- function-propiedad de un objeto -> `method`

ECMAScript define una colección de objetos incorporados que completan la definición de las entidades de ECMAScript. Estos objetos incorporados incluyen el ***global object*** el "objeto global"; esto es, objetos que son fundamentales para el valor semático de algunos elementos del lenguaje incluyendo:

- `Object`,
- `Function`,
- `Boolean`,
- `Symbol`,
- varios objetos del tipo "`Error`",

Objetos que manipulan y representan valores numéricos, incluyendo:

- `Math`,
- `Number`,
- `Date`,

Objetos que procesan texto:

- `String`,
- `RegExp`,

Objetos que son colecciones indexadas de valores, incluyendo:

- `Array`,
- y nueve clases de "`Typed Arrays`" todos los cuales representan un tipo de dato numérico específico,

objetos que representan colecciones con "clave", (claves pareadas):

- `Map`,
- `Set`, objetos que soportan estructuras de datos incluyendo:
- `JSON`
- `ArrayBuffer`
- `SharedArrayBuffer`
- `DataView` y por último, objetos que soportan abstracciones de control como los generadores de funciones y el objeto `Promise` y *reflection objects* incluyendo `Proxy` y `Reflect`.

ECMAScript también define un conjunto de operadores incorporados, incluyendo de tipo.

- unario
- multiplicativos
- aditivos

- de cambio de bits (*bitwise shift*),
- relacionales
- de igualdad
- binarios de bit (*binary bitwise*)
- logicos binarios
- de asignación
- y el operador coma (,)

Grandes programas en ECMAScript son soportados a través de *modules*, módulos que permiten dividir un solo gran programa en múltiples secuencias de sentencias y declaraciones (*statements and declarations*). Cada módulo idenfica explícitamente que declaraciones que usa son provistas por otros módulos, y cuáles de sus declaraciones están disponibles para ser usadas por otros módulos.
(NOTA PERSONAL(N.P): `import` and `export`??)

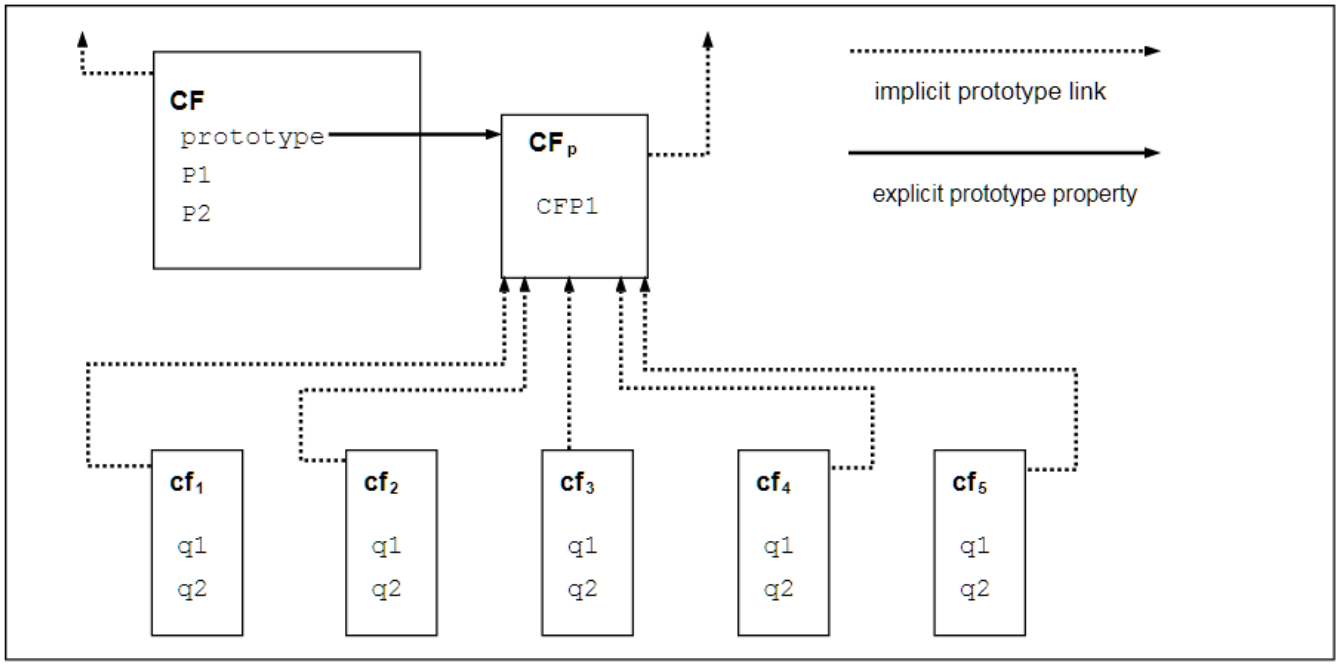
La sintaxis de ECMAScript se parece intencionalmente a la de Java, aunque esta se ha relajado para que sirva como lenguaje de scripting fácil de usar. Por ejemplo, una variable no requiere tener su tipo declarado, ni son tipos asociados con propiedades, y una función definida no requiere tener una declaración previa antes de poder llamarla (N.P: C++, Java)

4.2.1 Objects

Aun cuando ECMAScript incluye sintaxis para la definición de clases, los objetos ECMAScript no son fundamentalmente orientados a clases (*class-based*) como aquellos que podemos encontrar en C++, Smalltalk or Java. En vez de ello, los objetos pueden ser creados mediante varias maneras, incluyendo **notación literal** o a través de ***constructors*** que crean objetos y luego ejecutan código que los inicializa totalmente o en parte, asignando valores iniciales a sus propiedades. Cada constructor es una función que tiene una propiedad llamada "`prototype`" que es usada para implementar herencia basada en prototipado (*prototype-based inheritance*) y propiedades compartidas (*shared properties*). Objetos son creados usando constructores con la expresion que incluye la palabra `new`. Por ejemplo:
`new Date(2009,11)` crea un nuevo objeto del tipo "Date". Invocar un constructor sin usar la palabra `new`tiene consecuencias que dependen del constructor invocado. Por ejemplo, `Date()` produce una cadena que representa la fecha y hora actuales en vez de un objeto propialmente tal.

Cada objeto creado a través de un constructor tiene una referencia implícita (llamada el "prototipo del objeto") al valor de la propiedad `prototype`de su constructor. Además, un prototipo puede una referencia 'non-null' implicita a su prototipo, y así: esto es llamado: **cadena de prototipado** ('*prototype chain*'). Cuando es hecha una referencia a una propiedad en un objeto, esa referencia es a la propiedad de ese nombre en el primer objeto en la cadena de prototipado que contiene una propiedad con ese nombre. En otras palabras, primero el objeto mencionado directamente es examinado en busca de tal propiedad; si ese objeto contiene la propiedad nombrada, esa es la propiedad a la cual la referencia apunta, si el objeto no contiene la propiedad nombrada, se examina el prototipo de ese objeto, y así.

Figure 1: Object/Prototype Relationships



> En un lenguaje basado en clases, orientado a objetos, en general, el estado es portado por instancias, los métodos son portados por clases, y la herencia es solo de estructura y comportamiento. En ECMAScript, el estado y los métodos con portados por objetos, mientras la estructura, comportamiento, y estado, son todos heredados.

Obsevando la Figura anterior: **CF** es un constructor (y también es un objeto). Cinco objetos han sido creados usando expresiones con `new`: **cf1**, **cf2**, **cf3**, **cf4** y **cf5**. Cada uno de estos contiene propiedades llamadas **q1** y **q2**. Las líneas punteadas representan una relación de prototipado implícito, asi, por ejemplo, el prototipo de **cf3** es **CFp**. El constructor, **CF**, tiene dos propiedades él mismo, llamadas **P1** y **P2** las cuales no son visibles a **CFp**, **cf1**, **cf2**, **cf3**, **cf4** o **cf5**. La propiedad llamada **CFP1** en **CFp** es compartida por **cf1**, **cf2**, **cf3**, **cf4** y **cf5** (pero no por **CF**), así como lo es cualquier propiedad encontrada en la cadena de prototipado implícito de **CFp** que no sean **q1**, **q2**, o **CFP1**. Observe que no existe una relación de prototipado implícito entre **CF** y **CFp**.

A diferencia de la mayoría de los lenguajes basados en clases, las propiedades pueden ser agregadas a objetos dinámicamente asignándoles valores. Esto es, los constructores no requieren nombrar o asignar valores a todas o cualquiera de las propiedades del objeto construido. En el diagrama anterior, uno puede agregar una nueva propiedad compartida para **cf1**, **cf2**, **cf3**, **cf4** y **cf5** al asignarle un nuevo valor a esa propiedad en **CFp**.

A pesar de que los objetos en ECMAScript no son intrínsecamente basados en clases, a menudo es conveniente definir abstracciones similares a las clases, basadas en un patrón común de constructor de funciones, prototipos y métodos. Los mismos objetos integrados en ECMAScript siguen tal patron similar a clases. Comenzando con ECMAScript 2015, el lenguaje ECMAScript incluye definiciones de clase sintácticas que permiten a los programadores definir de manera concisa objetos que se ajustan al patrón de abstracción parecido a clases que usan los propios objetos integrados.

4.2.2 The Strict Variant of ECMAScript

El lenguaje ECMAScript reconoce la posibilidad de que algunos del usuario puedan desear restringir el uso de algunas características disponibles en el lenguaje. Ellos podrían hacerlo por intereses de seguridad, para evitar lo que consideran una característica que propende a errores, para obtener control de errores mejorado, o por otras razones de su elección. Para soportar esta posibilidad, ECMAScript define una variante estricta del lenguaje. La variante estricta del lenguaje excluye algunas características sintácticas y semánticas del lenguaje EMCAScript regular y modifica la semántica detallada de algunas características. La variante estricta también especifica condiciones de error adicionales que deben ser reportadas arrojando `error exception`s en situaciones que no están consideradas errores por la forma no estricta del lenguaje.

La variante estricta de ECMAScript es comúnmente referida como el `strict mode` del lenguaje. La selección de este modo y el uso de sus características sintácticas y semánticas, es realizado explícitamente al nivel de los elementos de código fuente individuales. Debido a que el modo estricto es elegido a nivel de la unidad de código fuente, el modo estricto solo impone restricciones que tienen efecto local dentro de aquél código fuente. El modo estricto no restringe o modifica ningún aspecto de la semántica de ECMAScript que deba funcionar de manera consistente a través de múltiples unidades de código fuente. Un programa completo ECMAScript puede estar compuesto de unidades de código fuente de ambos modos, estricto y no estricto. En estos casos, el modo estricto aplica solo cuando se está ejecutando código que es definido dentro de una unidad de código fuente en modo estricto.

Para ajustarse a esta especificación, una implementación ECMAScript debe implementar ambos, la variante completa sin restricciones y la variante estricta de ECMAScript según son definidas en esta especificación. Además, una implementación debe soportar la combinación de codigo estricto y no estricto en un solo programa.

4.3 Terms and Definitions

Para los propósitos de esta documentación, aplican las siguientes definiciones:

4.3.1 type

Conjunto de valores de datos, tal como están definidos en la cláusula 6 de esta especificación.

4.3.2 primitive value

miembro de una de los siguientes tipos: `Undefined, Null, Boolean, Number, Symbol o String` tal como están definidas en la cláusula 6.

> Nota: Un valor primitivo es un dato que se representa en en nivel más bajo de la implementación del lenguaje.

4.3.3 object

miembro del tipo `Object`

> Nota: un objeto es una colección de propiedades y debe tener **un solo** objeto `prototype`(N.P. "propiedad"). El prototipo puede ser un valor `null`.

4.3.4 constructor

objeto función que crea e inicializa objetos

> Nota: El valor de la propiedad `prototype` de un constructor es un objeto `prototype`que es usado para implementar herencia y propiedades compartidas.

4.3.5 prototype

objeto que provee propiedades compartidas para otros objetos.

> Nota: Cuando un constructor crea un objeto, ese objeto referencia implícitamente a la propiedad `prototype` de su constructor, con el objeto de resolver la referencia de propiedades. El propiedad del constructor `prototype`puede ser referenciada por la expresion del programa `constructor.prototype`, y las propiedades agregadas al prototipo de un objeto son compartidas, a través de herencia, por todos los objetos que comparten el prototipo. De manera alternativa, un nuevo objeto puede ser creado con un prototipo explícitamente especificado usando la la función integrada `Object.create`.

4.3.6 ordinary object

objeto que tiene el comportamiento por defecto para los metodos internos esenciales que deben ser soportados por todos los objetos.

4.3.7 exotic object

objeto que no tiene el comportamiento por defecto para uno o más de sus métodos internos esenciales.

> Nota: Todo objeto que no es un objeto ordinario es un objeto exótico.

4.3.8 standard object

objeto cuyas semánticas están definidas por esta especificación.

4.3.9 built-in object

objeto definido y provisto por la implementación ECMAScript.

> Nota: Los objetos integrados estándar son definidos en esta especificación. Una implementación ECMAScript puede especificar y entregar objetos integrados adicionales. Un *built-in constructor* es un objeto integrado que también es a la vez un constructor.

4.3.10 undefined value

valor primitivo utilizado cuando una variable no tiene un valor asignado.

4.3.11 Undefined type

tipo cuyo único valor puede ser `undefined`.

4.3.12 null value

valor primitivo que representa la ausencia intencional de cualquier valor.

4.3.13 Null type

tipo cuyo único valor puede ser Null.

4.3.14 Boolean value

miembro del tipo Boolean.

> Hay solo dos tipos valores booleanos, `true`y `false`.

4.3.15 Boolean type

tipo consistente de los valores `true` y `false`.

4.3.16 Boolean object

miembro del tipo `Object`que es una instancia del constructor estándar integrado `Boolean`

> Nota: un objeto booleano es creado usando el constructor `Boolean` con la expresión `new`, dándole un valor booleano como argumento. El objeto resultante tiene una ranura interna cuyo valor es un valor booleano. Un objeto booleano puede ser coaccionado a ser un valor booleano. (N.P. *the resulting object has an internal slot whose value...*)

4.3.17 String value

valor primitivo que es una secuencia finita ordenada de cero o más valores de 16 bits `unsigned integer`.

> Nota: un valor `String`es un miembro del tipo `String`. Cada valor `integer` en la secuencia usualmente representa una sola unidad (caracter) de 16 bits de tipo UTF-16. Sin embargo, ECMAScript no coloca ningún tipo de restricción o requerimientos a los valores excepto que ellos deben ser enteros-unsigned de 16 bits.

4.3.18 String type

conjunto de todo los valores posibles del valor `String`.

4.3.19 String object

miembro del tipo `Object` que es una instancia del constructor estándar integrado `String`.

> Nota: Un objeto `String` es creado usando el constructor `String`con la expresión `new`, dándole un valor `String`como argumento. El objeto resultante tiene un ranura interna cuyo valor es un valor `String`. un objeto `String` puede ser forzado a ser un valor `String` llamando al constructor `String` como una función. (Cláusula 21.1.1.1)

4.3.20 Number value

valor primitivo que corresponde a un valor binario de doble precision de 64 bits con el formato IEEE 754-2008.

> Nota: un valor `Number` es miembro del tipo `Number` que es representación direcrta de un número.

4.3.21 Number type

conjunto de todos los valores posibles del valor `Number`incluyendo: el valor especial "Not-a-number" (`NaN`), infinito positivo, e infinito negativo.

4.3.22 Number object

miembro del tipo `Object`que es una instancia del constructor estándar integrado `Number`

> Nota: Un objeto `Number`es crado usando el constructor `Number` con la expresión `new`dándole un valor numérico como argumento. El objeto resultante tiene una ranura interna cuyo valor es el valor número. Un objeto `Number`puede ser forzado a ser un valor `number`llamando el constructor `Number`como una función. (20.1.1.1)

4.3.23 Infinity

valor numérico que el valor numérico positivo.

4.3.24 NaN

valor numérico que es un valor IEEE 754-2008 "Not-a-Number"

4.3.25 Symbol value

valor primitivo que representa una única clave de propiedad de un Objeto, que no es una cadena (`String`)

4.3.26 Symbol type

conjunto de todos los posibles valores para Symbol

4.3.27 Symbol object

miembro del tipo `Object`que es una instancia del constructor estándar integrado `Symbol`

4.3.28 function

miembro del tipo `Object`que puede ser invocado como una subrutina.

> Nota: Además de sus propiedades, una función contiene un código ejecutable y estado que determina como se comporta cuando es invocada. El código de una función puede o no estar escrito en ECMAScript.

4.3.29 built-in function

objeto integrado que es una función.

> Nota: Ejemplos de funciones integradas incluyen `parseInt` y `Math.exp`. Una implementación cualquiera puede proveer funciones integradas dependientes de la implementación que no estén descritas en esta especificación.

4.3.30 property

parte de un objeto que asocia una clave (valor que puede ser del tipo `String` o `Symbol`) con un valor.

> Nota: Dependiendo de la forma de la propiedad el valor puede ser representado directamente como un valor de datos (un valor primitivo, un objeto o un una función-objeto) o indirectamente a traves de un par de funciones de acceso.

4.3.31 method

función que es el valor de una propiedad.

> Nota: cuando una función es llamada como el método de un objeto, el objeto es pasado a la función como su valor `this`.

4.3.32 built-in method

método que es una función integrada.

> Nota: Los métodos estándar integrados son definido en esta especificación, y una implementación ECMAScript puede especificar y proveer otros métodos adicionales integrados.

4.3.33 attribute

valor interno que define alguna característica de una propiedad.

4.3.34 own property

propiedad que está directamente contenida por su objeto.

4.3.35 inherited property

propiedad de un objeto que no es una propiedad "propia" (*own property*), pero es una propiedad (tanto propia como heredada) del objeto prototype.

4.4 Organization of this Specification

El resto de esta especificación está organizado de la siguiente manera:

1. La cláusula **5** define las convenciones de notación usadas en esta especificación.
2. Las cláusulas **6** a **9** definen el entorno de ejecución dentro del cual operan los programas ECMAScript.
3. Las cláusulas **10 a 16** son las que definen el lenguaje de programación ECMAScript incluyendo su forma sintáctica y la ejecución semántica de todas las caractericas del lenguaje.
4. Las cláusulas **17 a 26** definen la bibioteca estándar ECMAScript. Esta incluye la definicion de todos los objetos estándar que están disponibles para su uso cuando se ejecuta un programa ECMAScript.

5. Notational Conventions

5.1 Syntactic and Lexical Grammars (Gramática sintáctica y léxica)

5.1.1 Context-free Grammars

Una *gramática libre-de-contexto* consiste en un número de *producciones*. Cada *producción* tiene un símbolo abstracto llamado *nonterminal* como su *mano-izquierdo* y una secuencia de cero o más símbolos *nonterminal* y *terminal* como su *mano-derecha*. Para cada gramática, los símbolos *terminal* son extraídos de un alfabeto especificado.

Una *cadena-de-producción* es una producción que tiene exactamente **un solo** símbolo *nonterminal* en su *mano-derecha* junto a cero o más símbolos *terminal*.

Comenzando de una sentencia consistente en un solo *nonterminal* distinguido, llamado ***goal symbol***, una gramática 'libre de contexto' dada, especifica un ***lenguaje***, esto es, el (quizà infinito) conjunto de posibles secuencias de símbolos *terminal* resultantes de reemplazar repetidamente cualquier *nonterminal* en la secuencia con un (UUUF heeelp please!) "*right-hand side of a production for which the nonterminal is the left-hand side*.

- (N.P) Context-free grammars = production(s) (set of +zero)
- Each production: nonterminal-symbol [left-hand side] => nonterminal(n1, n2...ny) + terminal(n1...nx) [right-hand side]

> **ESTO ES IMPORTANTE, REVISAR Y ENTENDER, ABAJO UN ARTICULO DE WIKIPEDIA BASTANTE ILUSTRATIVO PARA COMENZAR. COMENTAR DETALLADAMENTE ESTE ELEMENTO DE LA ESPECIFICACIÓN PARA QUE NO SEA NECESARIO BUSCAR MÁS INFORMACIÓN PARA OBTENER UNA NOCIÓN COMPLETA => ESTO ES TEORÍA DE LENGUAJES. PEDAZO DE TEMA CABEZÓN ACA SUGERENCIA: ESTO PUEDE SER ESTUDIADO DETENIDAMENTE, MIENTRAS SE AVANZA SIN CONTRATIEMPOS HASTA LA CLÁUSULAS 10 Y 11 QUE ENTREGAN DETALLES ESPECÍFICOS DEL LENGUAJE ECMASCRIPT QUE PUEDEN AYUDAR MUCHO A ACLARAR ESTOS PARRAFOS**

COMENZAR A BUSCAR ACA

5.1.2 The Lexical and RegExp Grammars

> (Gramática léxica y de expresiones regulares)

Una gramática léxica parae ECMAScript es entregada en la cláusula 11. Esta gramática tiene como sus simbolos terminales (*terminal symbol*) elementos Unicode que se ajustan a las reglas para el `SourceCharacter` (conjunto fuente) definido en 10.1. Este define un set de producciones, comenzando con el símbolo objetivo (`goal symbol`) `InputElementDiv, InputElementTemplateTail, o InputElementRegExp, o InputElementRegExpOrTemplateTail` que describe cómo secuencias de aquellos elemetos (Unicode) son interpretados en secuencias de elementos de entrada.

> (N.P todos los elementos en codigo tienen referencias cruzadas en la especificación)

Los elementos de entrada distintos a espacios en blanco y comentarios, forman los símbolos-terminales (*terminal symbols*) para la gramática sintáctica de ECMAScript y son llamados `tokens` en ECMAScript. Estos `tokens`de ECMAScript son:

- las palabras reservadas
- identificadores
- literales
- signos de puntuación del lenguaje.

Además, los finalizadoes de línea, aunque no son considerados `tokens` también forman parte del flujo de elementos de entrada y guían el proceso de inserción automática de punto-y-comas (😉 (Clausula 11.9). Un espacio en blanco (N.P 'espacios en blanco'?) y comentarios de una sola línea son descartados y no aparecen en el flujo de elementos de entrada para la gramática sintáctica. Un `comentario multi-línea` (esto es, del tipo /.../ independiente de si ocupa efectivamente más de una línea) es descartado de inmediato si no contiene un finalizador de línea; por el contrario, si un `comentario multi-línea` contiene uno o más finalizadores de línea, entonces es reemplazado por un (sic) '*single line terminator*', que pasa a ser parte del flujo de elementos de entrada para la gramática sintáctica.

Una gramática para `RegExp`s en ECMAScript es detallada en la cláusula 21.2.1. Esta gramática también tiene como símbolos-terminales elementos como aquellos definidos en `SourceCharacter` (cláusula 10). Esta gramática define un set de producciones, comenzando por el `patrón`-símbolo-objetivo (sic *goal symbol Pattern*), que describe cómo secuencias de estos elementos (como los definidos en `SourceCharacter`, Unicode, c.10) son interpretadas en patrones de expresiones regulares.

Producciones de las gramáticas léxicas y de expresiones regulares (*RegExp*) son distinguibles fácilmente por tener dos (2) dos-puntos-> "::" como signo de puntuación. Las gramáticas léxicas y de expresiones regulares comparten algunas producciones.

5.1.3 The Numeric String Grammar

Otra gramática es utilizada para interpretar `Strings`en valores numéricos. Esta gramática es similar a la parte de la gramática léxica que tiene relación con los literales numéricos y tiene como símbolos-terminales elementos `SourceCharacter`. Esta gramática aparece en c.7.1.3.1.

Las producciones de la gramática de cadenas numéricas (sic *numeric string*) son distinguibles por tener tres (3) dos-puntos-> ":::" como signo de puntuación.

5.1.4 The Syntactic Grammar

La gramática sintáctica para ECMAScript es entregada en las cláusulas 11, 12, 13, 14, y 15. Esta gramática tiene `tokens` ECMAScript definidos por la gramática léxica como sus símbolos-terminales (c. 5.1.2). Esta define un conjunto de producciones, partiendo de dos alternativas de símbolo-objetivo: `Script` y `Module`que describen como las secuencias de `tokens` forman componentes independientes sintácticamente correctos en programas ECMAScript.

Cuando un flujo de unidades de còdigo (sic *code points*) es interpretado como un `Script` o `Module` de ECMAScript, primero es convertido a un flujo de elementos de entrada por la aplicación repetida de la gramática léxica; este flujo de elementos de entrada luego es interpretado (frecuentemente tb.: '*parseado*') por una sola aplicación de la gramática sintáctica. El flujo de entrada es sintácticamente incorrecto (sic *in error*) si los `tokens` en el flujo de elementos de entrada no puede ser interpretado como una sola instancia el símbolo no-terminal objetivo (`Script` o `Module`), sin `tokens` a su izquierda.

Cuando una interpretación es exitosa, esta construye un '*parse tree*', una estructura en forma de árbol en la cual nada nodo es un '*Parse Node*. Cada *Parse Node* es una `instance` de un símbolo en la gramática, representa un trozo del código fuente que puede ser derivado en ese símbolo. El nodo raíz del árbol parseado, representativo de todo el código fuente [N.P. de un archivo? de una sentencia o 'cadena de produccion'? ambiguo...], es una instancia de la interpretación del sìmbolo objetivo (*goal symbol*). Cuando un *Parse Node* es una instancia de un símbolo no-terminal, es también la instancia de alguna producción que tiene ese símbolo no-terminal como su lado-izquierdo (sic *left-hand side*). Además, tiene cero o más hijos (*children*), uno por cada símbolo en el lado derecho de la producción: cada 'hijo' es un *Parse Node* que es una instancia de su correspondiente símbolo.

Las producciones de la gramática sintáctica se distinguen por tener solo un (1) dos puntos-> ":" como puntuador.

La gramática sintáctica como se presenta en las cláusulas 12, 13, 14, y 15, no es una revisión completa de cuáles secuencias de `token`s son aceptables como correctos `Script` o `Module` de ECMAScript. Ciertas secuencias de `token`s adicionales también son aceptadas, es decir, aquellas que serían descritas por la gramática si solo se añadieran punto-y-coma (😉 en ciertos puntos de la secuencia (como antes de los caracteres finalizadores de línea). Además, ciertas secuencias de `token`s descritas por la gramática no se considerarán aceptables si aparece un finalizador de línea en lugares 'raros'. (sic: '*awkward*').

En ciertos casos, para evitar ambiguedades, la gramática sintáctica usa producciones comunes o 'estandar' (sic: *generalized*) que permiten ciertas secuencias de `token`s que no forman un `Script` o `Module` valido de ECMAScript. por ejemplo esta técnica es usada para `objects literals` y `object destructuring patterns`

5.1.5 Grammar Notation

> N.P.1 - ACLARATORIA: EN ESTA SUBSECCIÓN, TODO ELEMENTO EN ITÁLICA (CURSIVA) CORRESPONDE A UNA REFERENCIA CRUZADA EN LA REFERENCIA, QUE DEBE SER IMPLEMENTADA EN EL CUERPO DEL HTML. (EN ALGÚN MOMENTO)

> N.P.2. - PARA EFECTOS PRACTICOS Y MANTENER UN POCO EL FORMATO DE LA REFERENCIA, LAS "PRODUCCIONES" Y DEFINICIONES SON PRESENTADAS COMO CITAS CON INDENTADO.

> N.P.3 - LOS ELEMENTOS REPRESENTADOS MEDIANTE ENCERRADOS EN GUIONES BAJOS ESCAPADOS: _elemento_ INDICAN QUE EN LA REFERENCIA, 'elemento' APARECE COMO UN SUBÍNDICE, Y DEBE SER IMPLEMENTADO EN EL CUERPO DEL HTML.

> N.P.4 - LOS ELEMENTOS REPRESENTADOS DE LA SIGUIENTE MANERA: [_elemento_] INDICAN QUE ESE ELEMENTO, UNCLUIDOS LOS CORCHETES APARECE EN LA REFERENCIA CON FORMATO SUBÍNDICE. PARA SER IMPLEMENTADO.

Los símbolos terminales en las gramáticas léxicas, de expresiones regulares (RegExp) y de cadenas numéricas, son mostradas con fuente **negrita_minúscula**, *(original: "are shown in fixed width font") tanto las producciones de las gramáticas, como a través de toda la especificación, en cualquier texto que se refiera directamente a ese símbolo terminal. Estos aparecenán en un `script` exactamente como deben escribirse. Todos los elementos representando símbolos terminales que aparezcan así deben ser entendidos como elementos apropiados del rango 'Basic Latin'(sic) de Unicode, al contrario de cualquier otros similar de otros rangos Unicode.

Los símbolos no-terminales son mostrados en *itálica*. La definición de un no-terminal (también llamada "producción") es introducida por el nombre del elemento no-terminal que se está definiendo seguido de uno o mas dos-puntos ":" La cantidad de dos-puntos indica a cuál gramática pertence esa producción.
Recuerde:

1. gramática sintáctica =uno=> **:**
2. gramáticas léxica y de expresiones regulares =dos=> **::**
3. gramática de cadenas numéricas =tres=> **:::**

Luego de ello, una o más alternativas de "mano-derecha" para el no-terminal se muestran en líneas sucesivas. Por ejemplo, la definición sintáctica:

> *WhileStatement* **:**
>
> > **while (** *Expression* **)** *Statement*

declara que el no-terminal *WhileStatement* representa al `token` **while**, seguido por el `token` parentesis izquierdo, seguido por un elemento *Expression*, seguido por un `token` paréntesis derecho, seguido de un *Statement*. Las apariciones de *Statement* y *Expression* son no-terminales.
En otro ejemplo, la definición sintáctica:

> *ArgumentList* **:**
>
> > *AssignementExpresion*
> > *ArgumentList* **,** *AssignementExpression*

declara que un *ArgumentList* puede representar tanto una sola *AssignmentExpression* o un *ArgumentList*, seguido de una coma, seguido de un *AssignmentExpression*. Esta definición de *ArgumentList* es recursiva, o sea, se define a través de si misma. El resultado es que un *ArgumentList* puede contener cualquier cantidad positiva de argumentosm separados por comas, donde cada argumento es un *AssignmentExpression*. Tales definiciones recursivas de no-terminales son comunes.

El sufijo subíndice "$_{opt}$", que puede aparecer tras un terminal o no-terminal, indica un símbolo opcional. La elemento conteniendo el sufijo subíndice, implica que en realidad hay dos maneras de construir la mano-derecha, una que omite el elemento opcional y otra que lo incluye.
Por ejemplo:

> *VariableDeclaration* **:**
>
> > *BindingIdentifier Initializer*$_{opt}$

es una abreviación para:

> *VariableDeclaration* **:**
>
> > *BindingIdentifier*
> > *BindingIdentifier Initializer*

y por ejemplo esto:

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

es una abreviación conveniente para:

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

que a su vez es una abreviación de esto:

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

asi, en este ejemplo, el no-terminal *IterationStatement* en realidad tiene cuatro alternativas de mano-derecha.

Una producción puede ser parametrizada por un el sufijo subíndice "$_{[parameters]}$", el cual puede aparecer tras el símbolo no-terminal definido por la producción. "$_{[parameters]}$" puede ser un sol nombre (palabra), o una lista de nombres separados por comas. Una producción parametrizada es un "atajo" (sic: shorthand) para un conjunto de producciones que definen todas las combinaciones de nombres de parámetros precedidos por un guion bajo (_), adjunto al símbolo no-terminal parametrizado.
O sea:

> *StatementList*$_{[Return]}$ **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*

es una abreviación para:

> *StatementList* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*
>
> ---
>
> *StatementList_Return* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*

o por ejemplo, esto:

> *StatementList*$_{[Return, In]}$ **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*

es una abreviación para:

> *StatementList* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*
>
> ---
>
> *StatementList_Return* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*
>
> ---
>
> *StatementList_In* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*
>
> ---
>
> *StatementList_Return_In* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*

Múltiples parámetros producen una número combinado de producciones, no todas las cuales van a ser referenciadas necesariamente en la gramática.

Referencias a no-terminales en el lado-derecho de una producción también pueden ser parametrizados.
Por ejemplo:

> *StatementList* **:**

> > *ReturnStatement*
> > *ExpressionStatement*$_{[+In]}$

es equivalente a:

> *StatementList* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement_In*

y esto:

> *StatementList* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*$_{[\sim In]}$

es equivalente a:

> *StatementList* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement* //(N.P **¿¿HAY EN ESTOS DOS ULTIMOS EJEMPLOS UN ERROR EDITORIAL??**)

Una referencia a un no-terminal puede tener ambas, una lista de de parámetros y un sufijo "$_{opt}$".
Por ejemplo:

> *VariableDeclaration* **:**
>
> > *BindingIdentifier Initializer*$_{[+In]\ opt}$

es una abreviación para:

> *VariableDeclaration* **:**
>
> > *BindingIdentifier*
> > *BindingIdentifier Initializer_In*

Prefijando el nombre de un parámetro con el caracter "?" en el no-terminal del lado-derecho de la producción, hace que el valor de ese parámetro dependiente de la aparición del parámetro del mismo nombre en el símbolo de lado-izquierdo de la producción mencionada.
Por ejemplo:

> *VariableDeclaration*$_{[In]}$ **:**
>
> > *BindingIdentifier Initializer*$_{[?In]}$

es una abreviación para:

> *VariableDeclaration* **:**
>
> > *BindingIdentifier Initializer*
>
> ---
>
> *VariableDeclaration_In* **:**
>
> > *BindingIdentifier Initializer_In*

Si una alternativa para lado-derecho es prefijada con "[+parameter]" esa alternativa solo está disponible si el parámetro prefijado fue usado para referenciar el símbolo no-terminal de la producción. Por el contrario, si una alternativa de lado-derecho es prefijada con "[~parameter]" esa alternativa solo estará disponible si el parámetro no es usado para referenciar el símbolo no-terminal de la producción.
Asi tenemos que:

StatementList$_{[Return]}$ **:**

> [+Return] *ReturnStatement*
> *ExpressionStatement*

---

es una abreviación para:

> *StatementList* **:**
>
> > *ExpressionStatement*
>
> ---
>
> *StatementList_Return* **:**
>
> > *ReturnStatement*
> > *ExpressionStatement*

y por ejemplo:

> *StatementList*$_{[Return]}$ **:**
>
> > [~Return] *ReturnStatement*
> > *ExpressionStatement*

---

es una abreviación para:

> *StatementList* **:**
>
> > *ReturnStatement ExpressionStatement*
>
> ---
>
> *StatementList_Return* **:**
>
> > *ExpressionStatement*

Cuando las palabras "**one of** siguen a el(los) dos-puntos en una definición gramática, ellos significan que cada uno de los símbolos terminales en la siguiente o siguientes líneas es una definición alternativa.
Por ejemplo, la gramática léxica de ECMAScript contiene la siguiente producción:

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

---

lo cual es una abreviación bastante conveniente para:

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

Si la frase "[empty]" aparece como el lado-derecho de una producción, eso indica que el lado-derecho de la producción no contiene ni terminales ni no-terminales.

Si la frase "[lookahead ∉ *set*]" aparece en el lado derecho de la producción, eso indica que la producción no puede ser usada si el `token` de entrada inmediatamente siguiente en la secuencia es un miembro del *set* dado. El *set* puede ser escrito como una lista separada por comas de una o dos secuencias de elementos terminales encerrados entre corchetes "{}". Por conveniencia, el *set* puede también ser escrito como un no-terminal, en cuyo caso este representa el conjunto de todos los terminales a los que aquel no-terminal puede expandirse. Si el *set* consiste en un solo terminal la frase "[lookahead ≠ *terminal*]" puede ser usada.

Por ejemplo, dadas las siguientes definiciones:

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

> *DecimalDigits* **: :**
>
> > *DecimalDigit*
> > *DecimalDigits DecimalDigit*

la siguiente definición:

> *LookaheadExample* **: :**
>
> > **n** [lookahead ∉ {**1 , 3 , 5 , 7 , 9**}] *DecimalDigits*
> > *DecimalDigit* [lookahead ∉ *DecimalDigit*]

hace match con cualquiera de las posibilidades, o la letra n seguida de uno o más decimales en los que el primero es par, o un dígito decimal no seguido de ningún digito decimal.

Si la frase "[no *LinerTerminator* here]" aparece en el lado-derecho de una producción de gramática sintáctica, esta indica que la producción es una 'producción restringida' (sic: *restricted production*): no puede ser usada si un *Line Terminator* ocurre en el flujo de entrada en la posición indicada.
Por ejemplo, la producción:

> *ThrowStatement* **:**
>
> > **throw** [no *LineTerminator* here] *Expression* **;**

indica que la producción no puede ser usada si un *LineTerminator* ocurre en el código entre el `token`**throw** y *Expression*.

A no ser que la presencia de un *LineTerminator* esté prohibida por una producción restringida, cualquier número de apariciones de *LineTerminator* puede ocurrir entre dos `tokens`consecutivos cualquiera dentro del flujo de elementos de entrada sin afectar la validez sintáctica del código (sic: script).

Cuando una alternativa en una producción de gramática léxica o de cadenas numéricas (*numeric string*) parece ser un `token` de 'puntos de código múltiples' (sic: 'multi-code point'), este representa la secuencia de 'puntos de código' que conformarían tal `token`.

El lado-derecho de una producción puede especificar que ciertas expresiones no son permitidas, al utilizar la frase "**but not** y luego indicar las expansiones a ser excluidas. Por ejemplo, la producción:

> *Identifier* **: :**
>
> > *IdentifierName* but not *ReservedWord*

quiere decir que el no-terminal *Identifier* puede ser reemplazado por cualquier secuencia de 'puntos de codigo' que puedan reemplazar a *IdentifierName* pero la misma secuencia no puede reemplazar a *ReservedWord*.

Finalmente, unos pocos símbolos no-terminales son descritos por una frase descriptiva en 'sans-serif' (sic) en casos donde sería poco práctico listar todas las alternativas:

> *SourceCharacter* **: :**
>
> > any Unicode code point

(N.P. esa última línea se subentiende en sans-serif que es el uso en el estándar).

5.2 Algorithm Conventions

La especificación frecuentemente usa listas numeradas para especificar pasos en un algoritmo. Estos algoritmos son usados para precisar de manera puntual los requetimientos semáticos de una construcción en lenguaje ECMAScript. Los algoritmos no están pensados para usarlos necesariamente como una técnica de implementación específica. En la práctica, puede haber algoritmos más eficientes para implementar una característica dada.

Los algoritmos pueden ser explícitamente parametrizados, en cuyo caso, los nombres y uso de los parámetros, deben ser provistos como parte de la definición del algoritmo. Para poder facilitar su uso en múltiples partes de esta especificación, algunos algoritmos, llamados *operaciones abstractas*(sic: abstract operations), son nombradas y escritas en forma funcional parametrizada, para que ellas puedan ser referenciadas por su nombre en desde dentro de otros algoritmos. Las operaciones abstractas son típicamente referenciadas usando un estilo de aplicación funcional, tal como operationName(*arg1, arg2*). Algunas operaciones abstractas son tratadas como métodos despachados polimórficamente, en base a la especificación de abstracciones orientada a clases. Tales operaciones abstractas-similares-a-métodos (sic: such method-like abstract operations) son típicamente referenciadas usando un estilo de aplicación de métodos tal como *someValue*.operationName(*arg1, arg2*).

Las llamadas a las operaciones abstractas devuelven (sic: return) ***Completion Records***. {LinkReq} Las operaciones abstractas referenciadas usando los estilos de aplicación funcional o de aplicación de métodos que están prefijados con un **?** indican que **ReturnIfAbrupt** {LinkReq} debe ser aplicado al resultado de **Completion Record** {LinkReq}. Por ejemplo, `? operationName()`es equivalente a `ReturnIfAbrupt(operationName())`. De manera similar, `? someValue.operationName()` es equivalente a `ReturnIfAbrupt(someValue.operationName())`.

> (NP. {linkreq} o {Lrq} indica que la expresión que antecede a este es una referencia cruzada faltante)

El prefijo **!** es usado para indicar que una operación abstracta nunca deberá devolver un **abrupt completion** {Lrq} y que el valor resultante del **Completion Record** {Lrq} deberá ser usado en lugar del valor retornado por la operación.
Por ejemplo: `Let val be ! operationName()` es equivalente al siguiente algoritmo:

1. Let **val** be operationName().
2. **Assert** {Lrq}: val is never an **abrupt completion**.
3. if **val** is a **Completion Record**, let **val** be **val**.[[Value]].

Los algoritmos pueden ser asociados con producciones de una o más gramáticas de ECMAScript. Una producción que tiene múltiples definiciones alternativas, normalmente tendrá un algoritmo para cada alternativa. Cuando un algoritmo es asociado con una producción gramática, este puede referenciar los símbolos terminales y no-terminales de esa definición alternativa como si ellos fueran parámetros del algoritmo. Cuando son usados de esta manera, los símbolos no-terminales hacen referencia a la alternativa que ajusta al texto fuente parseado.

Cuando un algoritmo es asociado con una producción alternativa, esta alternativa es típicamente mostrada sin ninguna comentario gramático "[]". Aquellos comentarios deberían afectar solo el reconocimiento sintáctico de la alternativa y no deberian tener efecto en la semántica asociada de la alternativa.

A menos que se especifique explícitamente de otra manera, toda cadena de producciones (sic: chain productions) tienen una definición implícita para cada algoritmo que pudiera ser aplicado al no-terminal de la producción de lado-izquierdo. Esta definición implícita simplemente reaplica el mismo algoritmo con los mismos parámetros, si los hubiera, al no-terminal del lado-derecho de la cadena de producción, y luego devuelve el resultado.
Por ejemplo, existe esta presunta producción:

> *Block* **:**
>
> > { *StatementList* }

pero no hay ningún algoritmo "Evaluation" que esa explicitamente específico de esta producción. Entonces, si en algún algoritmo hay una declaración con esta forma: "Devuelve el resultado de evaluar *Block*" resulta implícito que existe un algoritmo "Evaluacion" con la forma:

> **Runtime Semantics: Evaluation**
> *Block* **: {** *StatementList* **}**
>
> > 1. Return the result of evaluating *StatementList*.

Para mayor claridad, los pasos de un algoritmo pueden ser subdivididos en sub-pasos secuenciales. Estos sub-pasos, están indentados y pueden a su vez subdividirse en otros sub-pasos indentados. Un número consecutivo al inicio de cada paso, es la convención usada para identificar un paso, con el primer nivel de sub-pasos identificado con letras minúsculas y el tercer nivel, si fuera necesario, con número romanos en minúscula. Si fuera necesario agregar más niveles de subpasos, estos se deben etiquetar repitiendo la formula, donde, por ejemplo, el cuarto nivel tendría números para identificar cada sub-paso y asi.
Por ejemplo:

1. Top-level step
    1. substep
    2. substep
        - subsubstep
            - subsubstep....

N.P debido a las limitaciones de Markdown, el ejemplo puede no representar bien el patrón de repetición indicado, pero lo importante es conocer su orden: [numero-letraMinuscula-numeroRomanoMinuscula] eso, en orden regularmente. No sea wn po!

UN paso o subpaso puede estar escrito como una sentencia "if" que condiciona el sub-paso. Si fuera el caso, el sub-paso solo aplica si el predicado evaluado es verdadero. Si un paso o sub-paso comienza con la palabra "else", es un predicado que niega el "if" precedente en el mismo nivel.

Un paso puede indicar la aplicación iterativa de sus sub-pasos.

Un paso que comienza con "*Assert:*" afirma (NP assert = afirmar) una condición invariable de su algoritmo. Tales afirmaciones (sic: assertions) son usadas para hacer explícito un elemnto invariable del algoritmo que de otra manera sería implícito. Estas afirmaciones no implican ningún requerimiento semántico adicional, y por lo tanto no necesitan ser verificados por una implementación. Son usados simplemente para aclarar algoritmos.

5.3 Static Semantic Rules

6. ECMAScript Data Types and Values

7. Abstract Operations

8. Executable Code and Execution Contexts

9. Ordinary and Exotic Objects Behaviours

10. ECMAScript Language: Source Code

11. ECMAScript Language: Lexical Grammar

12. ECMAScript Language: Expressions

13. ECMAScript Language: Statements and Declarations

14. ECMAScript Language: Functions and Classes

15. ECMAScript Language: Scripts and Modules