ECMAScript Language Specification

ECMA Committee #39
Version 0.9

February 27, 1997

Feedback

Please send feedback regarding this document to Guy Steele (Guy.Steele@east.sun.com).

CHAPTER 1

NOTATIONAL CONVENTIONS

1.1 SYNTACTIC AND LEXICAL GRAMMARS

This section describes the context-free grammars used in this specification to define the lexical and syntactic structure of an ECMAScript program.

1.1.1 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 one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the (pehaps 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.

1.1.2 The Lexical Grammar

A *lexical grammar* for ECMAScript is given in Chapter 3. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input*, that describe how sequences of Unicode characters are translated into a sequence of input elements.
These input elements, with white space and comments discarded, 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.
Productions of the lexical grammar are distinguished by having two colons “**::**” as separating punctuation.

1.1.3 The Numeric String Grammar

A second grammar is used for translating strings into numeric values; this grammar is similar to the part of the lexical grammar having to do with numeric literals. This grammar appears in Chapter 5.
Productions of the numeric string grammar are distinguished by having three colons “**:::**” as punctuation.

1.1.4 The Syntactic Grammar

The *syntactic grammar* for ECMAScript is given in Chapters 7, 8, 9, and 10. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol *Program*, that describe how sequences of tokens can form syntactically correct ECMAScript programs.
Productions of the syntactic grammar are distinguished by having just one colon “**:**” as punctuation.
The syntactic grammar as presented in Chapters 7, 8, 9, and 10 is actually not a complete account of which token sequences are accepted as correct ECMAScript programs. 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 end-of-line characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if an end-of-line character appears in certain “awkward” places.
A LALR(1) version of the syntactic grammar is presented in Appendix E. This version provides an exact account of which token sequences are acceptable ECMAScript programs without needing special rules about automatically adding semicolons or forbidding end-of-line characters. However, it is much more complex than the grammar presented in Chapters 7, 8, 9, and 10.

1.1.5 Grammar Notation

Terminal symbols are shown in **`fixed width`** font in the productions of all the grammars, and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal 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:

> *WithnStatement* **:**
>
> > **`with (`** *Expression* **`)`** *Statement*

states that the nonterminal *WithStatement* represents the token **`with`**, 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 to say, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

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

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

is a convenient abbreviation for:

> *VariableDeclaration* **:**
>
> > *Identifier*
> > *Identifier Initializer*

and that:

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

is a convenient abbreviation for:

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

which in turn is an abbreviation for:

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

which in turn is an abbreviation for:

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

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

If the phrase “” appears in the right-hand side of a production, it indicates that the production is a *restricted production* it may not be used if a *LineTerminator* occurs in the input stream at the indicated position. For example, the production:

ReturnStatement **:**

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

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

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:

ZeroToThree **:: one of**

```
0          1          2          3
```

which is merely a convenient abbreviation for:

ZeroToThree **::**

```
0
1
2
3
```

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multicharacter token, it represents the sequence of characters that would make up such a token.

The right-hand side of a production may specify that certain expansions are not permitted by using the phrase “**but not**” and then indicating the expansions to be excluded. For example, the production:

Identifier **::**

IdentifierName **but not** *ReservedWord*

means that the nonterminal *Identifier* may be replaced by any sequence of characters that could replace *IdentifierName* provided that the same sequence of characters could not replace *ReservedWord*.

Finally, a few nonterminal symbols are described by a descriptive phrase in roman type in cases where it would be impractical to list all the alternatives:

SourceCharacter:

any Unicode character

1.2 ALGORITHM CONVENTIONS

We often use a numbered list to specify steps in an algorithm. When the algorithm is to produce a value as a result, we use the directive “return x” to indicate that the result of the algorithm is the value of x and that the algorithm should terminate. We use the notation Result(n) as short hand for “the result of step n”. We also use Type(x) as short hand for “the type of x”. If an algorithm is defined to “generate a runtime error”, execution of the algorithm (and any calling algorithms) is terminated and no result is returned.

These algorithms are used to clarify semantics. In practice, there may be more efficient algorithms available to implement a given feature.

CHAPTER 2

SOURCE TEXT

2.1 UNICODE

ECMAScript source text is represented as a sequence of characters representable using the Unicode version 2.0 character encoding. However, it is possible to represent every ECMAScript program using only ASCII characters (which are equivalent to the first 128 Unicode characters). Non-ASCII Unicode characters may appear only within comments and string literals; in both of those contents, any Unicode character may be expressed as a Unicode escape sequence consisting of six ASCII characters, namely **\u** plus four hexadecimal digits, and the effect is exactly the same as if the Unicode character itself had appeared in place of the escape sequence.

SourceCharacter **::**
any Unicode character

2.2 END OF SOURCE

For purposes of describing the grammar of ECMAScript, the source text is assumed to be terminated by a logical “end of source” character. We represent the end-of-source character by <EOS>.

EndOfSource **::**
<EOS>

CHAPTER 3

LEXICAL CONVENTIONS

The source text of a ECMAScript program is first converted into a sequence of tokens and white space. A token is a sequence of characters that comprise a lexical unit. The source text is scanned from left to right, repeatedly taking the longest possible sequence of characters as the next token.

3.1 WHITE SPACE

White space characters are used to improve source text readability and to separate tokens, indivisible lexical units, from each other but are otherwise insignificant. White space may occur between any two tokens, but not within a token. White space may also occur inside a string, where it is significant.

The following characters are considered white space:

Unicode Value	Name	Formal Name
\u0009	Tab	<TAB>
\u000A	Line Feed	<LF>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u000D	Carriage Return	<CR>
\u0020	Space	<SP>

Syntax

WhiteSpace **::**
  SimpleWhiteSpace WhiteSpace$_{opt}$
  LineTerminator WhiteSpace$_{opt}$
  Comment WhiteSpace$_{opt}$

SimpleWhiteSpace **::**
  <TAB>
  <SP>
  <FF>
  <VT>

LineTerminator **::**
  <CR>
  <LF>

LineEnd **::**
  LineTerminator
  <EOS>

3.2 COMMENTS

Description
Comments can be either single or multi-line. Multi-line comments cannot nest.

Syntax

Comment **::**
  MultiLineComment
  SingleLineComment

MultiLineComment **::**

`/*` *MultiLineCommentChars*$_{opt}$ `*/`

MultiLineCommentChars **::**
MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
`*` *PostAsteriskCommentChars*$_{opt}$

PostAsteriskCommentChars **::**
MultiLineNotFowardSlashChar
MultiLineCommentChars$_{opt}$

MultiLineNotAsteriskChar ***::***
SourceCharacter **but not** *asterisk* `*` **or** *<EOS>*

MultiLineNotFowardSlashChar ***::***
SourceCharacter **but not** *forward-slash* `/` **or** *<EOS>*

SingleLineComment **::**
`//` *SingleLineCommentChars*$_{opt}$ *LineTerminator*
`//` *SingleLineCommentChars*$_{opt}$ *EndOfSource*

SingleLineCommentChars **::**
SingleLineCommentChar SingleLineCommentChars$_{opt}$

SingleLineCommentChar **::**

SourceCharacter **but not** *LineEnd*

3.3 TOKENS

Syntax

Token **::**
ReservedWord
Identifier
Punctuator
Literal
EndOfSource

3.3.1 Reserved Words

Description

Reserved words cannot be used as identifiers.

ReservedWord **::**
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

3.3.1.1 Keywords

The following keywords are in use in either the the Borland ECMAScript implementation, the Netscape 1.1 ECMAScript implementation, the Microsoft JScript implementation or all three.

Syntax

Keyword: **one of**

```
break          continue       delete         else
for            function       if             in
new            return         this           typeof
var            void           while          with
```

3.3.1.2 Future Reserved Words

The following words are used as keywords in proposed extensions and are thus reserved to allow for the adoption for those extensions.

Syntax

FutureReservedWord: **one of**

```
arguments      case           catch          class
default        do             extends        finally
implicit       import         super          switch
throw          try
```

3.3.2 IDENTIFIERS

Description

An identifier is a sequence of letters, digits and special characters that must begin with either a letter, the underscore (**_**) character or the dollar sign (**$**) character. ECMAScript identifiers are case sensitive: identifiers whose characters differ only in case are considered unique.

Syntax

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

IdentifierName **::**
 IdentifierLetter
 IdentifierName IdentifierLetter
 IdentifierName DecimalDigit

IdentifierLetter **:: one of**

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

DecimalDigit **:: one of**

```
0 1 2 3 4 5 6 7 8 9
```

3.3.3 PUNCTUATORS

Syntax

Punctuator **:: one of**

```
=       >       <       ==      <=      >=
!=      ,       !       ~       ?       :
.       &&      ||      ++      --      _
```

`-`	**`*`**	**`/`**	**`&`**	**`|`**	**`^`**
`%`	**`<<`**	**`>>`**	**`>>>`**	**`+=`**	**`-=`**
`*=`	**`/=`**	**`&=`**	**`|=`**	**`^=`**	**`%=`**
`<<=`	**`>>=`**	**`>>>=`**	**`(`**	**`)`**	**`{`**
`}`	**`[`**	**`]`**	**`;`**		

3.3.4 LITERALS

Syntax

Literal **::**
- *NullLiteral*
- *BooleanLiteral*
- *NumericLiteral*
- *StringLiteral*

3.3.4.1 Null Literals

Syntax

NullLiteral **::**
- **`null`**

Semantics

The value of the null literal **`true`** is the sole value of the null type, namely **null**.

3.3.4.2 Boolean Literals

Syntax

BooleanLiteral **::**
- **`true`**
- **`false`**

Semantics

The value of the Boolean literal **`true`** is a value of the Boolean type, namely **true**.

The value of the Boolean literal **`false`** is a value of the Boolean type, namely **false**.3.3.4.3 Numeric Literals

Syntax

NumericLiteral **::**
- *IntegerLiteral*
- *FloatingPointLiteral*

IntegerLiteral **::**
- *DecimalIntegerLiteral*
- *HexIntegerLiteral*
- *OctalIntegerLiteral*

DecimalIntegerLiteral **::**
- **`0`**
- *NonZeroDigit DecimalDigits*$_{opt}$

DecimalDigits **::**
- *DecimalDigit*
- *DecimalDigits DecimalDigit*

NonZeroDigit **:: one of**

`1        2        3        4        5        6        7        8        9`

HexIntegerLiteral **::**
- **`0x`** *HexDigit*

`0X` *HexDigit*
HexIntegerLiteral HexDigit

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

OctalIntegerLiteral **::**
`0` *OctalDigit*
OctalLiteral OctalDigit

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

FloatingPointLiteral **::**
DecimalIntegerLiteral **`.`** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
`.` *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart

ExponentPart **::**
ExponentIndicator SignedInteger

ExponentIndicator **:: one of**
`e E`

SignedInteger **::**
DecimalDigits
`+` *DecimalDigits*
`-` *DecimalDigits*

Semantics
A numeric literal stands for a value of the number type. This value is determined in two steps: first, a mathematically accurate value is derived from the literal; second, this mathematical value (MV) is rounded, using IEEE 754 round-to-nearest mode , to a representable value of the number type.
For any production *A* :: *B* with a single nonterminal on its right-hand side, the MV of *A* is the MV of *B*.
The MV of *DecimalLiteral* **::** **`0`** is positive zero.
The MV of *DecimalLiteral* **::** *NonZeroDigit Digits* is (the MV of *NonZeroDigit* times 10^n) plus the MV of *Digits*, where *n* is the number of characters in *Digits*.
The MV of *DecimalDigits* **::** *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
The MV of *DecimalDigit* **::** **`0`** or of *HexDigit* **::** **`0`** or of *OctalDigit* **::** **`0`** is positive zero.
The MV of *DecimalDigit* **::** **`1`** or of *NonZeroDigit* **::** **`1`** or of *HexDigit* **::** **`1`** or of *OctalDigit* **::** **`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`** or of *OctalDigit* **::** **`8`** is 8.
The MV of *DecimalDigit* **::** **`9`** or of *NonZeroDigit* **::** **`9`** or of *HexDigit* **::** **`9`** or of *OctalDigit* **::** **`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 *HexIntegerLiteral* **::** `0x` *HexDigit* is the MV of *HexDigit*.
The MV of *HexIntegerLiteral* **::** `0X` *HexDigit* is the MV of *HexDigit*.
The MV of *HexIntegerLiteral* **::** *HexIntegerLiteral HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.
The MV of *OctalIntegerLiteral* **::** `0` *OctalDigit* is the MV of *OctalDigit*.
The MV of *OctalIntegerLiteral* **::** *OctalIntegerLiteral OctalDigit* is (the MV of *OctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.
The MV of *FloatingPointLiteral* **::** *DecimalIntegerLiteral* `.` is the MV of *DecimalIntegerLiteral*.
The MV of *FloatingPointLiteral* **::** *DecimalIntegerLiteral* `.` *DecimalDigits* is the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n}), where n is the number of characters in *DecimalDigits*.
The MV of *FloatingPointLiteral* **::** *DecimalIntegerLiteral* `.` *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
The MV of *FloatingPointLiteral* **::** *DecimalIntegerLiteral* `.` *DecimalDigits ExponentPart* is (the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n})) times 10^{e}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
The MV of *FloatingPointLiteral* **::** `.` *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where n is the number of characters in *DecimalDigits*.
The MV of *FloatingPointLiteral* **::** `.` *DecimalDigits ExponentPartDecimalDigits* is the MV of *DecimalDigits* times 10^{e-n}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
The MV of *FloatingPointLiteral* **::** *DecimalIntegerLiteral ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
The MV of *ExponentPart* **::** *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
The MV of *SignedInteger* **::** `+` *DecimalDigits* is the MV of *DecimalDigits*.
The MV of *SignedInteger* **::** `-` *DecimalDigits* is the negative of the MV of *DecimalDigits*.

Issue: this description, as it stands, does not take into account the resolution that only the first 19 significant digits or so need contribute to the calculated mathematical value. This still needs to be addressed. (It could be addressed in the grammar itself, but it would be too messy: a couple of hundred productions!)3.3.4.4 String Literals

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

Syntax

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

DoubleStringCharacter **::**
  SourceCharacter **but not** *double-quote* `"` **or** *backslash* `\` **or** *LineEnd*
  EscapeSequence

SingleStringCharacter **::**
  SourceCharacter **but not** *single-quote* `'` **or** *backslash* `\` **or** *LineEnd*
  EscapeSequence

EscapeSequence **::**
  CharacterEscapeSequence
  OctalEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence **::**
    `\` *SingleEscapeCharacter*
    `\` *NonEscapeCharacter*

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

NonEscapeCharacter **::**

SourceCharacter **but not** *SingleEscapeCharacter* **or** *OctalDigit* **or `x` or `u` or** *LineEnd*
HexEscapeSequence **::**
    `\x` *HexDigit HexDigit*

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

OctalEscapeSequence **::**
    `\` *OctalDigit*
    `\` *OctalDigit OctalDigit*
    `\` *ZeroToThree OctalDigit OctalDigit*

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

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

UnicodeEscapeSequence **::**
    `\u` *HexDigit HexDigit HexDigit HexDigit*

Issue: Give a complete account of the interpretation of escape sequences.

The following table describes the set of character escape characters:

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

3.4 AUTOMATIC SEMICOLON INSERTION

Description
Certain ECMAScript statements (empty statement, variable statement, expression statement, **`continue`** statement, **`break`** statement, and **`return`** statement) must each be terminated with a semicolon. Such a semicolon may always appear explicitly in the source text. For convenience, however, such semicolons may be omitted from the source text in certain situations. We describe such situations by saying that semicolons are automatically inserted into the source code token stream in those situations:

- When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar and the parser is not currently parsing the header of a `for` statement, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:

  1. The offending token is separated from the previous token by at least one *LineTerminator*.
  2. The offending token is *EndOfSource*.
  3. The offending token is `}`.

  However, there is an additional overriding condition: a semicolon is never inserted automatically if the semicolon would then be parsed as an empty statement.

- When, as the program is parsed from left to right, a token (called the *restricted token*) is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the restricted token is separated from the previous token by at least one LineTerminator, then there are two cases:

  1. If the parser is not currently parsing the header of a `for` statement, a semicolon is automatically inserted before the restricted token.

  2. If the parser is currently parsing the header of a `for` statement, it is a syntax error.

  These are all the restricted productions in the grammar:

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

  *MemberExpression***:**
      MemberExpression [no *LineTerminator* here] *IncrementOperator*

  *CallExpression***:**
      MemberExpression [no *LineTerminator* here] *Arguments*
      NewCallExpression [no *LineTerminator* here] *Arguments*
      CallExpression [no *LineTerminator* here] *Arguments*

  The practical effect of these restricted productions is as follows:

  1. When the token `return` is encountered and a *LineTerminator* is encountered before the next token is encountered, a semicolon is automatically inserted after the token `return`.

  2. When the token `++` or `--` 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.

  3. When the token `(` is encountered where the parser would treat it as the first token of a parenthesized *Arguments* list, and at least one *LineTerminator* occurred between the preceding token and the `(` token, then a semicolon is automatically inserted before the `(` token.

  The resulting practical advice to ECMAScript programmers is:

  1. An *Expression* in a `return` statement should start on the same line as the `return` token.

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

3. The **(** that starts an argument list should be on the same line as the expression that indicates the function to be called.

-
-

For example, the source

```
{ 1 2 } 3<EOS>
```

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

```
{ 1
2 } 3<EOS>
```

is also not a valid ECMAScript sentence, but is transformed by automatic semicolon insertion into the following:

```
{ 1
;2 ;} 3;<EOS>
```

which is a valid ECMAScript sentence.

The source

```
for (a; b
)<EOS>
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion because the place where a semicolon is needed is within the header of a **for** statement. Automatic semicolon insertion never occurs within the header of a **for** statement.

The source

```
return
a + b<EOS>
```

is transformed by automatic semicolon insertion into the following:

```
return;
a + b;<EOS>
```

Note that the expression **a + b** is not treated as a value to be returned by the **return** statement, because a *LineTerminator* separates it from the token **return**.
The source

```
a = b
++c<EOS>
```

is transformed by automatic semicolon insertion into the following:

```
a = b;
++c;<EOS>
```

Note that the token **`++`** is not treated as a postfix operator applying to the variable **`b`**, because a *LineTerminator* occurs between **`b`** and **`++`**.

The source

```
if (a > b)
else c = d<EOS>
```

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.

CHAPTER 4

TYPES

A value is an entity that takes on one of seven types. There are six standard types and one internal type called **Reference**. Values of type **Reference** are only used as intermediate results of expression evaluation and cannot be stored to properties of objects.

4.1 THE UNDEFINED TYPE

The Undefined type has exactly one value, called **undefined**. Any variable that has not been assigned a value is of type undefined

4.2 THE NULL TYPE

The Null type has exactly one value, called **null**.

4.3 THE BOOLEAN TYPE

The Boolean type represents a logical entity and consists of exactly two unique values. One is called **true** and the other is called **false**.

4.4 THE NUMBER TYPE

The Number type has exactly 18437736874454810627 (that is, $2^{64}–2^{53}+3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the 9007199254740990 (that is, $2^{53}–2$) distinct **NaN** values of the IEEE Standard are represented in ECMAScript as single special **NaN** value.

There are two other special values, called **Positive Infinity** and **Negative Infinity**. The other 18437736874454810624 (that is, $2^{64}–2^{53}$) values are called the finite numbers. Half of these are positive numbers and half are negative numbers; for every finite positive number there is a corresponding negative number having the same magnitude.

Note that there is both a positive zero and a negative zero.

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

$$s \cdot m \cdot 2^e$$

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 between –1073 to 971, inclusive.

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

$$s \cdot m \cdot 2^e$$

where s is +1 or –1, m is a positive integer less than 2^{53}, 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).

Some ECMAScript operators deal only with integers in the range -2^{31} through $2^{31}–1$, inclusive, or in the range 0 through 2^{32}-1, inclusive. These operators accept any value of the Number type but first converts each such value to one of 2^{32} integer values. See the descriptions of the ToInt32 and ToUint32 operators in sections 5.5 and 5.6 ToUint32: (unsigned 32 bit integer) respectively.

.

4.5 THE OBJECT TYPE

An Object is an unordered collection of properties. Each property consists of a name, a value and a set of attributes.

4.5.1 Property Attributes

A property can have zero or more attributes from the following set:

Attribute	**Descption**
ReadOnly	The property is a read-only property. Attempts to write to the property will be ignored.
ErrorOnWrite	This attribute has precedence over the ReadOnly attribute. Attempts to write to the property will result in a runtime error and the property will not be changed.
DontEnum	The property is not included in the for-in enumeration. See the description of the for-in statement in section 8.5.3 The `for..in` Statement
DontDelete	Attempts to delete the property will be ignored. See the description of the `delete` operator in section 7.3.1 The `delete` Operator.
Internal	Internal properties have no name and are not directly accessible via the property accessor operators. How these properties are accessed is implementation specific. How and when some of these properties are used is specified by the language specification.

4.5.2 Property Access

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

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

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

The following table summarizes the internal properties related to property access:

Property	**Parameters**	**Description**
[[Get]]	(PropertyName)	Returns the value of the property.
[[Put]]	(PropertyName, Value)	Sets the property to value.
[[Prototype]]	None	Returns the parent object.
[[HasProperty]]	(PropertyName)	Returns a boolean value indicating whether the object already has a member with the given name.
[[Construct]]	Optional user provided parameters	(Constructor) Constructs an object. Invoked via the `new` operator.
[[Call]]	Optional user provided parameters	(Function) Executes the object..

Assume *O* is an ECMAScript object and *P* is a string.

4.5.2.1 HasProperty

When the [[HasProperty]] method of *O* is called with property name *P*, the following steps are taken:

1. If *O* has a property with name *P*, return **`true`**.
2. If the [[Prototype]] of *O* is **`null`**, return **`false`**.
3. Call the [[HasProperty]] method of [[Prototype]] with property name *P*.
4. Return Result(3).

4.5.2.2 Get

When the [[Get]] method of *O* is called with property name *P*, the following steps are taken:

1. If *O* doesn't have a property with name *P*, go to step 4.
2. Get the value of the property.
3. Return Result(2).
4. If the [[Prototype]] of *O* is **`null`**, return **`undefined`**
5. Call the [[Get]] method of [[Prototype]] with property name *P*.
6. Return Result(5).

4.5.2.3 Put

To aid in defining the [[Put]] method, the [[CanPut]] method is first defined. As [[CanPut]] method is only used here (by the [[Put] method with explicit access mode), it is not included in the table in 4.5.2.

When the [[CanPut]] method of *O* is called with property *P*, the following steps are taken:

1. If *O* doesn't have a property with name *P*, go to step 4.
2. If the property has the ErrorOnWrite attribute, generate a runtime error.
3. If the property has the ReadOnly attribute, return **`false`**.
4. If the [[Prototype]] of *O* is **`null`**, return **`true`**.
5. Call the [[CanPut]] method of [[Prototype]] of *O* with property Name *P*.
6. Return Result(5).

When the [[Put]] method of *O* is called with property *P* and value *V*, the following steps are taken:

1. Call the [[CanPut]] method of *O* with name *P*.
2. If Result(1) is **`false`**, return.
3. If *O* doesn't have a property with name *P*, go to step 6.
4. Set the value of the property to *V*.
5. Return.
6. Create a property with name *P*, set its value to *V* and give it empty attributes.
7. Return.

4.6 THE STRING TYPE

The String type consists of the set of all finite sequences of zero or more Unicode characters.
Note: The concatenation operator (**`+`**), relational operators (**`<`**, **`>`**, **`<=`**, **`>=`**) and equality operators (**`==`**, **`!=`**) apply to this type.

4.7 THE INTERNAL REFERENCE TYPE

The Internal Reference Type is not a language data type Is it only defined here for the purposes of aiding this specification.

A **Reference** is a reference to an object's property. A **Reference** consists of two parts, the *base object* and the *property name.*

In defining the semantics of ECMAScript, the following methods are defined for internal operations:

- GetBase(). Returns the base object component.

- GetPropertyName(). Returns the propertyName component.
- GetValue(). Returns the value of the indicated property.
- PutValue(). Sets the indicated property to the indicated value.

Values of type **Reference** are only used as intermediate results of expression evaluation and cannot be stored to properties of objects.

4.7.1 GetBase

1. If Type(V) is a Reference, return the base object component of *V*.
2. Generate a runtime error.

4.7.2 GetPropertyName

1. If Type(V) is a Reference, return the propertyName component of *V*.
2. Generate a runtime error.

4.7.3 GetValue

1. If Type(V) is not a Reference, return *V*.
2. Call GetBase(*V*).
3. If Result(2) is **null**, generate a runtime error.
4. Call the [[Get]] method of Result(2), passing GetPropertyName(*V*) for the property name.
5. Return Result(4).

4.7.4 PutValue

For values *V* and *W*, PutValue(*V*, *W)* performs:

1. If type (V) is not a Reference, generate a runtime error.
2. Call GetBase(V).
3. If Result(2) is **null**, go to step 6.
4. Call the [[Put]] method of Result(2), passing GetPropertyName(V) for the property name and W for the value.
5. Return.
6. Call the [[Put]] method for the *global object*, passing GetPropertyName(V) for the property name and W for the value.
7. Return.

CHAPTER 5

TYPE CONVERSION

The ECMAScript runtime system performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion operators. These operators are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion operators are polymorphic; that is, they can accept a value of any standard type, but not of type Reference.

5.1 TOPRIMITIVE

The operator ToPrimitive takes a Value argument and an optional PreferredType argument. The operator ToPrimitive attempts to convert its value argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favor that type. Conversion occurs according to the following table:

Input Type	**Result**
Undefined	Return the input argument (no conversion)
Null	Return the input argument (no conversion)
Boolean	Return the input argument (no conversion)
Number	Return the input argument (no conversion)
String	Return the input argument (no conversion)
Object	Return the default value of the Object. The default value of an object is retrieved by calling the interal [[DefaultValue]] method of the object passing an optional hint *preferredType*. The behavior of the [[DefaultValue]] method is defined by this specification for all native ECMAScript objects. If the return value is of type Object or Reference, a runtime error is generated.

5.2 TOBOOLEAN

The operator ToBoolean attempts to convert its argument to a value of type Boolean according to the following table:

Input Type	**Result**
Undefined	**`false`**
Null	**`false`**
Boolean	Return the input argument (no conversion)

Number	**`0`** → **`false`**
	`NaN` → **`false`**
	≠ **`0`** and ≠ **`NaN`** → **`true`**

String	**`= ""`** → **`false`** `(where "" denotes an empty string)` ≠ **`""`** → **`true`**
Object	**`true`**

5.3 TONUMBER

The operator ToNumber attempts to convert its argument to a value of type Number according to the following table:

Input Type	**Result**
Undefined	**`NaN`**

Null	`NaN`

Boolean	`true` → `1`
	`false` → `0`

Number	Return the input argument (no conversion)
String	See grammer and discussion below.
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint Number). 2. Call ToNumber(Result(1)). 3. Return Result(2).

5.3.1 ToNumber Applied to the String Type

ToNumber applied to strings applies the following grammar to the input string. If the grammar cannot interpret the string then the result of ToNumber is `NaN`.

StringNumericLiteral **:::**
  StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::**
  StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **:::**
  <TAB>
  <SP>
  <FF>
  <VT>
  <CR>
  <LF>

StrNumericLiteral **:::**
  StrIntegerLiteral
  StrFloatingPointLiteral

StrIntegerLiteral **:::**
  Sign$_{opt}$ *Digits*$_{opt}$
  HexIntegerLiteral

HexIntegerLiteral **:::**
  `0x` *HexDigit*
  `0X` *HexDigit*
  HexIntegerLiteral HexDigit

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

StrFloatingPointLiteral **:::**
  Sign$_{opt}$ *Digits* **`.`** *Digits*$_{opt}$ *ExponentPart*$_{opt}$
  Sign$_{opt}$ **`.`** *Digits ExponentPart*$_{opt}$
  Sign$_{opt}$ *Digits ExponentPart*

ExponentPart **:::**
  ExponentIndicator SignedInteger

ExponentIndicator **::: one of**
  `e E`

SignedInteger **:::**
Sign$_{opt}$ Digits

Sign **::: one of**
+ -

5.4 TOINTEGER

The operator ToInteger attempts to convert its argument to an integral numeric value. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **`NaN`**, return **`0`** (positive zero).
3. If Result(1) is **`±Infinity`**, return Result(1).
4. Compute sign(Result(1)) * floor(abs(Result(1))).
5. Return Result(4).

5.5 TOINT32: (SIGNED 32 BIT INTEGER)

The operator ToInt32 converts its argument to one of 2^{32} integer values in the range -2^{31} through 2^{31}-1, inclusive. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is NaN, Positive Infinity, or Negative Infinity, return **`0`** (positive zero).
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. If Result(3) is positive zero or negative zero, return **`0`** (positive zero).
5. Compute Result(3) modulo 232; that is, if Result(3) is negative, compute the value of the expression 2^{32} - ((1-Result(3)) % 2^{32}) - 1; otherwise compute Result(3) % 2^{32}.
6. If Result(5) is greater than or equal to 2^{31}, return Result(5)-2^{32}; otherwise return Result(4).

Discussion:
Note that the ToInt32 operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.

Note also that ToInt32(ToUint32(x)) is equal to ToInt32(x) for all values of x.
(It is to preserve this latter property that Positive Infinity and Negative Infinity are mapped to zero.)

5.6 TOUINT32: (UNSIGNED 32 BIT INTEGER)

1. The operator ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through 2^{32}-1, inclusive. This operator functions as follows:Call ToNumber on the input argument.
2. If Result(1) is NaN, Positive Infinity, or Negative Infinity, return **`0`** (positive zero).
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. If Result(3) is positive zero or negative zero, return **`0`** (positive zero).
5. Compute Result(3) modulo 2^{32}; that is, if Result(3) is negative, compute the value of the expression 2^{32} - ((1-Result(3)) % 2^{32}) - 1; otherwise compute Result(3) % 2^{32}.
6. Return Result(5).

Discussion:
Note: Step 6 is the only difference between ToUint32 and ToInt32.

Note that the ToUint32 operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.

Note also that ToUint32(ToInt32(x)) is equal to ToUint32(x) for all values of x.

(It is to preserve this latter property that Positive Infinity and Negative Infinity are mapped to zero.)

5.7 TOSTRING

The operator ToString attempts to convert its argument to a value of type String according to the following table:

Input Type	Result
Undefined	`"undefined"`
Null	`"null"`

Boolean	`true` → `"true"`
	`false` → `"false"`

Number	See discussion below.
String	Return the input argument (no conversion)
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint String). 2. Call ToString(Result(1)). 3. Return Result(2).

5.7.1 ToString Applied to the Number Type

The operator ToString converts a number to string format as follows:

- If the argument is NaN, the result is the string `"NaN"`.

- Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. If the sign is negative, the first character of the result is '`-`'; if the sign is positive, no sign character appears in the result. As for the magnitude *m*:

  - If *m* is infinity, it is represented by the characters '`Infinity`'; thus, positive infinity produces the result `"Infinity"` and negative infinity produces the result `"-Infinity"`.

  - If *m* is zero, it is represented by the character '`0`'; thus, negative zero produces the result `"-0"` and positive zero produces the result `"0"`.

  - If *m* is an integer less than 10^{16}, then it is represented as that integer value in decimal form with no leading zeroes and no decimal point.

  - If *m* is greater than or equal to 10^{-3} but less than 10^{16}, and is not an exact integer value, then it is represented as the integer part (floor) of *m*, in decimal form with no leading zeroes, followed by a decimal point '`.`', followed by one or more decimal digits (see below) representing the fractional part of *m*.

  - If *m* is less than 10^{-3} or not less than 10^{16}, then it is represented in so-called "computerized scientific notation." Let *n* be the unique integer such that $10^n \leq m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 \leq a < 10$. The magnitude is then represented as the integer part (floor) of *a*, as a single decimal digit, followed by a decimal point '`.`', followed by one or more decimal digits (see below) representing the fractional part of *a*, followed by the letter '`E`', followed by a representation of *n* as a decimal integer (first a minus sign '`-`' if *n* is negative or nothing of *n* is not negative, followed by the decimal representation of the magnitude of *n* with no leading zeros).

How many digits must be printed for the fractional part of *m* or *a*? There must be at least one digit; beyond that, there must be as many, but only as many, more digits as are needed to uniquely distinguish the argument value from all other representable numeric values. That is, suppose that *x* is the exact mathematical value represented by the decimal representation produced by this method for a finite nonzero argument ; then *d* must be the value of number type nearest to *x*; or if two values of thenumber type are equally close to *x*, then *d* must be one of them and the least significant bit of *d* must be 0. A consequence of this specification is that ToString never produces trailing zero digits for a fractional part.

Implementors of ECMAScript may find useful the paper and code written by David M. Gay for binary-to-decimal conversion of floating-point numbers [Gay 1990].

5.8 TOOBJECT

The operator ToObject attempts to convert its argument to a value of type Object according to the following table:

Input Type	**Result**
Undefined	generate a runtime error
Null	generate a runtime error
Boolean	Create a Boolean object whose default value is the value of the boolean. See the Native ECMAScript Objects section for a description of the Boolean object.
Number	Create a Number object whose default value is the value of the number. See the Native ECMAScript Objects section for a description of the Number object.
String	Create a String object whose default value is the value of the string. See the Native ECMAScript Objects section for a description of the String object.
Object	Return the input argument (no conversion)

CHAPTER 6

EXECUTION CONTEXTS

When control is transferred to ECMAScript executable code, we say that control is entering an *execution context*. Active execution contexts logically form a stack. The top execution context on this logical stack is the running execution context.

6.1 DEFINITIONS

6.1.1 Function Objects

There are four types of function objects:

- *Declared functions* are defined in source text by a *FunctionDeclaration*.
- *Anonymous functions* are created dynamically by using the built-in **`Function`** Object as a constructor which we refer to as instantiating **`Function`**.
- *Host functions* are created at the request of the host with source text supplied by the host. The mechanism for their creation is implementation dependent. Host functions may have any subset of the following attributes { ImplicitThis, ImplicitParents }. These attributes are described below.
- *Internal functions* are built-in objects of the language, such as **`parseInt`** and **`Math.exp`**. These functions do not contain executable code defined by the ECMAScript grammar, so are excluded from this discussion of execution contexts.

6.1.2 Types of Executable Code

There are five types of executable ECMAScript source text:

- *Global code* is source text that is outside all function declarations. More precisely, the global code of a particular ECMAScript *Program* consists of all *SourceElements* in the *Program* production which come from the *Statement* definition.
- *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 *Program*. The eval code for a particular invocation of **`eval`** is the global code portion of the string parameter.
- *Function code* is source text that is inside a function declaration. More precisely, the function code of a particular ECMAScript *FunctionDeclaration* consists of the *Block* in the definition of *FunctionDeclaration*.
- *Anonymous code* is the source text supplied when instantiating **`Function`**. More precisely, the last parameter provided in an instantiation of **`Function`** is converted to a string and treated as the *StatementList* of the *Block* of a *FunctionDeclaration*. If more than one parameter is provided in an instantiation of **`Function`**, all parameters except the last one are converted to strings and concatenated together, separated by commas. The resulting string is interpreted as the *FormalParameterList* of a *FunctionDeclaration* for the *StatementList* defined by the last parameter.
- *Host code* is the source text supplied by the host when creating a host function. The source text is treated as the *StatementList* of the *Block* of a *FunctionDeclaration*. Depending on the implementation, the host may also supply a *FormalParameterList*.

6.1.3 Variable Instantiation

Every execution context has associated with it a *variable object*. Variables declared in the source text are
added as properties of the variable object. For global and eval code, functions defined in the source text are added as properties of the variable object. Function declarations in other types of code are not

allowed by the grammar. For function, anonymous and host code, parameters are added as properties of the variable object.

Which object is used as the variable object and what attributes are used for the properties depends on the
type of code, but the remainder of the behavior is generic:

- For each *FunctionDeclaration* in the code, in source text order, instantiate a declared function from the *FunctionDeclaration* and create a property of the variable object whose name is the Identifier in the *FunctionDeclaration*, whose value is the declared function and whose attributes are determined by the type of code. If the variable object already has a property with this name, replace its value and attributes.
- For each formal parameter, as defined in the *FormalParameterList*, create a property of the variable object whose name is the *Identifier* and whose attributes are determined by the type of code. The values of the parameters are supplied by the caller. If the caller supplies fewer parameter values than there are formal parameters, the extra formal parameters have value **`undefined`**. If two or more formal parameters share the same name, hence the same property, the corresponding property is given the value that was supplied for the last parameter with this name. if the value of this last parameter was not supplied by the caller, the value of the corresponding property is **`undefined`**.
- For each *VariableDeclaration* in the code, create a property of the variable object whose name is the *Identifier* in *VariableDeclaration*, whose value is **`undefined`** and whose attributes are determined by the type of code. If there is already a property of the variable object with the name of a declared - variable, the value of the property and its attributes are not changed. Semantically, this step must follow the creation of the *FunctionDeclaration* and *FormalParameterlist* properties. In particular, if a declared variable has the same name as a declared function or formal parameter, the variable declaration does not disturb the existing property.

6.1.4 Scope Chain and Identifier Resolution

Every execution context has associated with it its own a *scope chain*. This is logically a list of objects that are searched when *binding* an *Identifier.* When control enters an execution context, the scope chain is created and is populated with an initial set of objects, depending on the type of code. When control leaves the execution context, the scope chain is destroyed.

During execution, the scope chain of the execution context is affected only by *WithStatement*. When execution enters a **`with`** block, the object specified in the **`with`** statement is added to the front of the scope chain. When execution leaves a with block, whether normally or via a **`break`** or **`continue`** statement, the object is removed from the scope chain. The object being removed will always be the first object in the scope chain.

During execution, the syntactic production *PrimaryExpression*: *Identifier* is evaluated using the following algorithm:

1. Get the next object in the scope chain. If there isn't one, go to step 5.
2. Call the [[HasProperty]] method of Result(l), passing the *Identifier* as the property.
3. If Result(2) is **`true`**, return a value of type Reference whose base object is Result(l), property name is the identifier.
4. Go to step 1.
5. Return a value of type Reference whose base object is **`null`** and whose property name is Identifier.

The result of binding an identifier is always a value of type Reference with its member name component equal to the identifier string.

6.1.5 Global Object

There is a unique *global object* which is created before control enters any execution context. Initially the global object has the following properties:

- Built-in objects such as Math, String, Date, parseInt, etc. These have attributes { DontEnum }.
- Additional host defined properties. This may include a property whose value is the global object itself, for example window in HTML.

As control enters execution contexts, and as ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be changed.

6.1.6 Activation Object

When control enters an execution context for function code, anonymous code or host code, an object called the activation object is created and associated with the execution context. The activation object is initialized with a single property with name **`arguments`** and property attributes { DontDelete }. The initial value of this property is the arguments object described below. The activation object is then used as the variable object for the purposes of variable instantiation.

The activation object is purely a specification mechanism. It is impossible for an ECMAScript program to access the activation object. It can access members of the activation object, but not the activation object itself. When the call operation is applied to a Reference value whose base object is an activation object, **`null`** is used as the **`this`** value of the call.

6.1.7 LabelStacks

The definitions of the control flow statements use two 1ogical stacks, the *break label stack* and the *continue label slack*. These are to facilitate the semantic definition of these statements and are not intended to imply a particular implementation. Each execution context has its own label stacks, which are created and initialized to empty when control enters the execution context When control leaves the execution context, the label stacks are destroyed.

6.1.8 This

There is a **`this`** value associated with every active execution context. The **`this`** value depends on the caller and the type of code being executed and is determined when control enters the execution context. The **`this`** value associated with an execution context is immutable.

6.1.9 Arguments Object

When control enters an execution context for function, anonymous or host code, an arguments object is created and initialized as follows:

〈 A property is created with name **`callee`** and property attributes { DontEnum }. The initial value of this property is the function object being executed. This allows anonymous functions to be recursive.

〈 A property is created with name **`length`** and property attributes { DontEnum }. The initial value of this property is the number of actual parameter values supplied by the caller.

〈 For each non-negative integer, iarg, less than the value of the **`length`** property, a property is created with name ToString(iarg) and property attributes { DontEnum }. The initial value of this property is the value of the corresponding actual parameter supplied by the caller. The first actual parameter value corresponds to iarg = 0, the second to iarg = 1 and so on. In the case when iarg is less than the number of formal parameters for the function object, this property shares its value with the corresponding property of the activation object. This means that changing this property changes the corresponding property of the activation object and vice versa. The value sharing mechanism depends on the implementation.

Issue: Should the arguments object have a caller property?

6.2 Entering An Execution Context

When control enters an execution context, the scope chain is created and initialized, variable instantiation is performed, the break label and continue label stacks are created and initialized to empty, and the **`this`** value is determined

The initialization of the scope chain, variable instantiation, and the determination of the **`this`** value depend on the type of code being entered.

6.2.1 Global Code

- The scope chain is created and initialized to contain the global object and no others.
- Variable instantiation is performed using the global object as the variable object and using empty property attributes.
- The **`this`** value is the global object.

6.2.2 EvalCode

When control enters an execution context for eval code, the previous active execution context, referred to as the *calling context*, is used to determine the scope chain, the variable object, and the **`this`** value. If there is no calling context, then initializing the scope chain, variable instantiation, and determination of the **`this`** value are performed just as for global code.

- The scope chain is initialized to contain the same objects, in the same order, as the calling context's scope chain. This includes objects added to the calling context's scope chain by *WithStatement.*
- Variable instantiation is performed using the calling context's variable object and using empty property attributes.
- The **`this`** value is the same as the **`this`** value of the calling context.

6.2.3 Function and Anonymous Code

- The scope chain is initialized to contain the activation object followed by the global object.
- Variable instantiation is performed using the activation object as the variable object and using property attributes {, DontDelete }.
- The caller provides the **`this`** value. If the **`this`** value provided by the caller is not an object (including the case where it is **`null`**), then the **`this`** value is the global object.

6.2.4 Host Code

- The scope chain is initialized to contain the activation object as its first element.
- If the host function has the ImplicitThis attribute, the this value is placed in the scope chain after the activation object.
- If the host function has the ImplicitParents attribute, a list of objects determined solely by the **`this`** value, is inserted in the scope chain after the activation object and **`this`** object. Note that this list is determined at runtime by the **`this`** value. It is not determined by any form of lexical scoping.
- The global object is placed in the scope chain after all other objects.
- Variable instantiation is performed using the activation object as the variable object and using attributes { DontEnum, DontDelete}
- The **`this`** value is determined just as for function and anonymous code.

CHAPTER 7

EXPRESSIONS

7.1 PRIMARY EXPRESSIONS

Syntax

PrimaryExpression:
: **`this`**
: *Identifier*
: *Literal*
: **`(`** *Expression* **`)`**

7.1.1 The `this` Keyword

The **`this`** keyword evaluates to the **`this`** value of the execution context.

7.1.2 Identifier Reference

An *Identifier* is evaluated using the scoping rules stated in section 6.1.4 Scope Chain and Identifier Resolution. The result of an *Identifier* is always a value of type Reference.

7.1.3 Literal Reference

A *Literal* is evaluated as described in section 3.3.4 Literals.

7.1.4 The Grouping Operator

The production *PrimaryExpression*: **`(`** *Expression* **`)`** is evaluated as follows:

1. Evaluate *Expression*. This may be of type Reference.
2. Return Result(1).

7.2 POSTFIX EXPRESSIONS

Syntax

MemberExpression:
: *PrimaryExpression*
: *MemberExpression* **`[`** *Expression* **`]`**
: *MemberExpression* **`.`** *Identifier*
: *MemberExpression* [no *LineTerminator* here] *IncrementOperator*

IncrementOperator:
: **`++`**
: **`--`**

NewExpression:
: **`new`** *MemberExpression*

NewCallExpression:
: **`new`** *MemberExpression Arguments*

CallExpression:
: *MemberExpression* [no *LineTerminator* here] *Arguments*
: *NewCallExpression* [no *LineTerminator* here] *Arguments*
: *CallExpression* [no *LineTerminator* here] *Arguments*
: *CallExpression* **`[`** *Expression* **`]`**
: *CallExpression* **`.`** *Identifier*

CallExpression IncrementOperator

Arguments:
: `( )`
: `(` *ArgumentList* `)`

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

PostfixExpression:
: *MemberExpression*
: *CallExpression*
: *NewExpression*

The postfix increment operators and property accessor operators `[ ]` and `.` appear in both the *MemberExpression* and *CallExpression* productions. Generally we will refer to the productions involving *MemberExpression* with the understanding that the same remarks apply to *CallExpression*. Similarly, the *CallExpression* production includes three definitions involving the *Arguments* non-terminal. We will refer to the definition involving *CallExpression*.

7.2.1 Property Accessors

Properties are accessed by name, using either the dot notation *MemberExpression* `.` *Identifier* or the bracket notation *MemberExpression* `[` *Expression* `]`.

The dot notation is transformed using the following syntactic conversion:

MemberExpression `.` *Identifier*

is exactly equivalent to:

MemberExpression `[` <identifier-string> `]`

where <identifier-string> is a string literal containing the same sequence of characters as the identifier.

The production *MemberExpression*: *MemberExpression* `[` *Expression* `]` is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *Expression*.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(2)).
6. Call ToString(Result(4)).
7. Return a value of type Reference whose base object is Result(5), member name is Result(6) and access mode is explicit.

7.2.2 Postfix Increment and Decrement Operators

The production *MemberExpression*: *MemberExpression IncrementOperator* is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. For `++`, Result(4) is Result(3) increased by one. For `--`, Result(4) is Result(3) decreased by one. In either case, if Result(3) is **`NaN`** or **`±Infinity`**, Result(4) is the same as Result(3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(32).

7.2.3 The `new` Operator

The production *NewExpression* **:** `new` *MemberExpression* is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. If Type(Result(2)) is not Object, generate a runtime error.
4. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
5. Call the [[Construct]] method on Result(2), providing no arguments (that is, an empty list of arguments).
6. If Type(Result(5)) is not Object, generate a runtime error.
7. Return Result(5).

The production *NewCallExpression* **:** `new` *MemberExpression Arguments* is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. For each *AssignmentExpression* in *ArgumentList*, in left to right order, evaluate *AssignmentExpression* and call GetValue on the result. Keep all of these values in an internal list.
4. If Type(Result(2)) is not Object, generate a runtime error.
5. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
6. Call the [[Construct]] method on Result(2), providing the list generated in step 3 as the parameters.
7. If Type(Result(6)) is not Object, generate a runtime error.
8. Return Result(6).

7.2.4 Function Calls

The production *CallExpression* **:** *CallExpression Arguments* is evaluated as follows:

1. Evaluate *CallExpression*.
2. For each *AssignmentExpression* in *ArgumentList*, in left to right order, evaluate *AssignmentExpression* and call GetValue on the result. Keep all of these values in an internal list.
3. Call GetValue(Result(1)).
4. If Type(Result(3)) is not Object, generate a runtime error.
5. If Result(3) does not implement the internal [[Call]] method, generate a runtime error.
6. If Type(Result(1)) is Reference, Result(6) is GetBase(Result(1)). Otherwise, Result(6) is `null`.
7. If Result(6) is an activation object, Result(7) is `null`. Otherwise, Result(7) is the same as Result(6).
8. Call the [[Call]] method on Result(3), providing Result(7) as the `this` value and providing the list generated in step 2 as the parameters.
9. Return Result(8).

Note: Result(8) will never be of type Reference for native ECMAScript objects. Whether an external object can return a value of type Reference is implementation dependent.

7.3 UNARY OPERATORS

Syntax

UnaryExpression **:**
- *PostfixExpression*
- `delete` *UnaryExpression*
- `void` *UnaryExpression*
- `typeof` *UnaryExpression*
- *IncrementOperator UnaryExpression*
- `+` *UnaryExpression*
- `-` *UnaryExpression*
- `~` *UnaryExpression*
- `!` *UnaryExpression*

7.3.1 The `delete` Operator

The production *UnaryExpression*: **`delete`** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. Call GetBase(Result(1)).
3. Call GetPropertyName(Result(1)).
4. If Type(Result(2)) is not Object, return **`true`**.
5. If Result(2) does not implement the internal [[Delete]] method, go to step 8.
6. Call the [[Delete]] method on Result(2), providing Result(3) as the property name to delete.
7. Return Result(7).
8. Call the [[HasProperty]] method on Result(2), providing Result(3) as the property name to check for.
9. If Result(8) is **`true`**, return **`false`**.
10. Return **`true`**.

7.3.2 The `void` Operator

The production *UnaryExpression*: **`void`** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. Call GetValue(Result(1)).
3. Return **`undefined`**.

7.3.3 The `typeof` Operator

The production *UnaryExpression*: **`typeof`** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. If Type(Result(1)) is Reference and GetBase(Result(1)) is **`null`**, return **`"undefined"`**.
3. Call GetValue(Result(1)).
4. Return a string determined by Type(Result(3)) according to the following table:

Type	**Result**
Undefined	**`"undefined"`**
Null	**`"object"`**
Boolean	**`"boolean"`**
Number	**`"number"`**
String	**`"string"`**
Object (native and doesn't implement [[Call]])	**`"object"`**
Object (native and implements [[Call]])	**`"function"`**
Object (external)	unspecified

Issue: What does typeof return for external objects?

7.3.4 Prefix Increment and Decrement Operators

The production *UnaryExpression*: *IncrementOperator UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. For **`++`**, Result(4) is Result(3) increased by one. For **`--`**, Result(4) is Result(3) decreased by one. In either case, if Result(3) is **`NaN`** or **`±Infinity`**, Result(4) is the same as Result(3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(4).

7.3.5 Unary `+` and `-` Operators

The production *UnaryExpression*: **`+`** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.

2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Return Result(3).

The production *UnaryExpression*: `-` *UnaryExpression* is evaluated as follows:
1. Evaluate *UnaryExpression*
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. If Result(3) is **`NaN`**, return **`NaN`**.
5. Negate Result(3).
6. Return Result(5).

7.3.6 The Bitwise NOT Operator (`~`)

The production *UnaryExpression*: `~` *UnaryExpression* is evaluated as follows:
1. Evaluate *UnaryExpression*
2. Call GetValue(Result(1)).
3. Call ToInt32(Result(2)).
4. Apply bitwise complement to Result(3).
5. Return Result(4).

7.3.7 Logical NOT Operator (`!`)

The production *UnaryExpression*: `~` *UnaryExpression* is evaluated as follows:
1. Evaluate *UnaryExpression*
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **`true`**, return **`false`**.
5. Return **`true`**.

7.4 MULTIPLICATIVE OPERATORS

Syntax

MultiplicativeExpression:
- *UnaryExpression*
- *MultiplicativeExpression* `*` *UnaryExpression*
- *MultiplicativeExpression* `/` *UnaryExpression*
- *MultiplicativeExpression* `%` *UnaryExpression*

Semanitcs

The production *MultiplicativeExpression*: *MultiplicativeExpression @ UnaryExpression*, where @ stands for one of the operators in the above definitions, is evaluated as follows:
1. Evaluate *MultiplicativeExpression*
2. Call GetValue(Result(1)).
3. Evaluate *UnaryExpression*
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the specified operation (`*`, `/`, or `%`) to Result(5) and Result(6). See the discussions below (7.4.1, 7.4.2, 7.4.3).
8. Return Result(7).

7.4.1 Applying the `*` Operator

The `*` operator performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in ECMAScript, because of finite precision. The result of a floating-point multiplication is governed by the rules of IEEE 754 double-precision 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.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite non-zero value results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest 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.

7.4 2 Applying the `/` Operator

The `/` operator 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 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 non-zero 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.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in zero.
- Division of a non-zero 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 round-to-nearest mode. If the magnitude is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, we say the operation underflows and the result is zero. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

7.4 3 Applying the `%` Operator

The binary `%` operator is said to yield the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor. In C and C++, the remainder operator accepts only integral operands, but 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. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a truncating division, and so its behavior 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 a 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.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result is zero.

- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical relation r = n - (d * q) where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d.

7.5 ADDITIVE OPERATORS

Syntax

AdditiveExpression **:**
MultiplicativeExpression
AdditiveExpression **+** *MultiplicativeExpression*
AdditiveExpression **-** *MultiplicativeExpression*

7.5.1 The Addition Operator (+)

The addition operator either performs string concatenation or numeric addition.

The production *AdditiveExpression* **:** *AdditiveExpression* **+** *MultiplicativeExpression* is evaluated as follows:

1. Evaluate *AdditiveExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *MultiplicativeExpression*.
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2), hint Number).
6. Call ToPrimitive(Result(4), hint Number).
7. If Type(Result(5)) is String or Type(Result(6)) is String, go to step 13.
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. If Result(8) or Result(9) is **NaN**, return **NaN**.
11. Apply the addition operation to Result(8) and Result(9). See the discussion below.
12. Return Result(11).
13. Call ToString(Result(5)).
14. Call ToString(Result(6)).
15. Concatenate Result(13) followed by Result(14).
16. Return Result(15).

7.5.2 The Subtraction Operator (-)

The production *AdditiveExpression* **:** *AdditiveExpression* **-** *MultiplicativeExpression* is evaluated as follows:

1. Evaluate *AdditiveExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *MultiplicativeExpression*.
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the subtraction operation to Result(5) and Result(6). See the discussion below (7.5.3).
8. Return Result(7).

7.5.3 Applying the Additive Operators (**+**, **-**)

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 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 zeros is zero.
- 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 zero.
- 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 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is zero. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.
- The **-** operator performs subtraction when applied to two operands of numeric type producing the difference of its operands; the left operand is the minuend and the right operand is the subtrahend. Given numeric operands a and b, it is always the case that a**-** b produces the same result as a **+** (**-**b).

7.6 BITWISE SHIFT OPERATORS

Syntax

ShiftExpression **:**
  AdditiveExpression
  ShiftExpression **<<** *AdditiveExpression*
  ShiftExpression **>>** *AdditiveExpression*
  ShiftExpression **>>>** *AdditiveExpression*

Discussion
The result of evaluating ShiftExpression is always truncated to 32 bits. If the result of evaluating ShiftExpression produces a fractional component, the factional component is discarded. The result of evaluating AdditiveExpresion is always truncated to five bits.

7.6.1 The Left Shift Operator (<<)

Performs a bitwise left shift operation on the left argument by the amount specified by the right argument.

The production *ShiftExpression* **:** *ShiftExpression* **<<** *AdditiveExpression* is evaluated as follows:

1. Evaluate *ShiftExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *AdditiveExpression*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Left shift Result(5) by Result(7) bits. The result is a signed 32 bit integer.
9. Return Result(8).

7.6.2 The Signed Right Shift Operator (>>)

Performs a sign-filling bitwise right shift operation on the left argument by the amount specified by the right argument.

The production *ShiftExpression* **:** *ShiftExpression* **>>** *AdditiveExpression* is evaluated as follows:

1. Evaluate *ShiftExpression*.

2. Call GetValue(Result(1)).
3. Evaluate *AdditiveExpression*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform sign-extending right shift of Result(5) by Result(7) bits. The most significant bit is propagated. The result is a signed 32 bit integer.
9. Return Result(8).

7.6.3 The Unsigned Right Shift Operator (**>>>**)

Performs a zero-filling bitwise right shift operation on the left argument by the amount specified by the right argument.

The production *ShiftExpression* **:** *ShiftExpression* **>>>** *AdditiveExpression* is evaluated as follows:
1. Evaluate *ShiftExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *AdditiveExpression*.
4. Call GetValue(Result(3)).
5. Call ToUint32(Result(2)).
6. Call ToInt32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform zero-filling right shift of Result(5) by Result(7) bits. Vacated bits are filled with zero. The result is an unsigned 32 bit integer.
9. Return Result(8).

7.7 RELATIONAL OPERATORS

Syntax

RelationalExpression **:**
  ShiftExpression
  RelationalExpression **<** *ShiftExpression*
  RelationalExpression **>** *ShiftExpression*
  RelationalExpression **<=** *ShiftExpression*
  RelationalExpression **>=** *ShiftExpression*

Semantics

In the discussion below, the following special operators will be used:

Operator	**Meaning**
Numeric@	Where @ represents one of the relational operators. The operands are of type Number. This is the standard IEEE operator with the provision that if either operand is **`NaN`**, the result is **`false`**.
Character@	Where @ represents one of the relational operators. The operands are of type String. The operands are compared character by character lexicographically in the unicode character set. If the operands are of different length and all corresponding characters up to the length of the shorter operand are the same, the longer string is considered to be greater.

The production *RelationalExpression* **:** *RelationalExpression @ ShiftExpression*, where @ represents one of the relational operators, is evaluated as follows:
1. Evaluate *RelationalExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *ShiftExpression*.
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2), hint Number).
6. Call ToPrimitive(Result(4), hint Number).
7. If Type(Result(5)) is Stringand Type(Result(6)) is String, go to step 13.

8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply Numeric@ to Result(8) and Result(9).
11. Return Result(10).
12. Call ToString(Result(5)).
13. Call ToString(Result(6)).
14. Apply Character@ to Result(12) and Result(13).
15. Return Result(14).

7.8 EQUALITY OPERATORS

Syntax

EqualityExpression **:**
- *RelationalExpression*
- *EqualityExpression* `==` *RelationalExpression*
- *EqualityExpression* `!=` *RelationalExpression*

The production *EqualityExpression* **:** *EqualityExpression* `==` *RelationalExpression* is evaluated as follows:

1. Evaluate *EqualityExpression*
2. Call GetValue(Result(1)).
3. Evaluate *RelationalExpression*
4. Call GetValue(Result(3)).
5. If Type(Result(2)) is different from Type(Result(4)) go to step 12
6. If Type(Result(2)) is Undefined, return `true`.
7. If Type(Result(2)) is Null, return `true`.
8. If Type(Result(2)) is Number, apply Numeric== to Result(2) and Result(4) and return the result.
9. If Type(Result(2)) is String, apply Character== to Result(2) and Result(4) and return the result.
10. If Type(Result(2)) is Boolean, return `true` when Result(2) and Result(4) are both `true` or both `false`. Otherwise, return `false`.
11. Return `true` if Result(2) and Result(4) refer to the same object. Otherwise, return `false`.
12. If Result(2) is `null` and Result(4) is `undefined`, return `true`.
13. If Result(2) is `undefined` and Result(4) is `null`, return `true`.
14. If Type(Result(2)) is Number and Type(Result(4)) is String, return the result of the comparison ToString(Result(2)) `==` Result(4).
15. If Type(Result(2)) is String and Type(Result(4)) is Number, return the result of the comparison Result(2) `==` ToString(Result(4)).
16. Return `false`.

The production *EqualityExpression* **:** *EqualityExpression* `!=` *RelationalExpression* is evaluated as follows:

1. Evaluate the production *EqualityExpression* `==` *RelationalExpression*
2. If Result(1) is `true`, return `false`.
3. Return `true`.

Discussion

String comparison can be forced by: `"" + a == "" + b`
Numeric comparison can be forced by: `a - 0 == b - 0`
Boolean comparison can be forced by: `!a == !b`.

The equality operators maintain the following invariants:

1. `A != B` is equivalent to `!(A == B)`.
2. `A == B` is equivalent to `B == A`, except in the order of evaluation of A and B.
3. `if A == B` and `B == C` , => `A == C`, assuming no side effects.

As no conversions are applied to the operands, equality is always transitive.

7.9 BINARY BITWISE OPERATORS

Syntax

BitwiseANDExpression:
  EqualityExpression
  BitwiseANDExpression **&** *EqualityExpression*

BitwiseXORExpression:
  BitwiseANDExpression
  BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseORExpression:
  BitwiseXORExpression
  BitwiseORExpression **|** *BitwiseXORExpression*

Semantics

The production *A* **:** *A @ B*, where @ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Evaluate *A*.
2. Call GetValue(Result(1)).
3. Evaluate *B*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Apply the bitwise operator @ to Result(5) and Result(6). The result is a signed 32 bit integer.
8. Return Result(7).

7.10 BINARY LOGICAL OPERATORS

Syntax

LogicalANDExpression:
  BitwiseORExpression
  LogicalANDExpression **&&** *BitwiseORExpression*

LogicalORExpression:
  LogicalANDExpression
  LogicalORExpression **||** *LogicalANDExpression*

Semantics

The production *LogicalANDExpression*: *LogicalANDExpression* **&&** *BitwiseORExpression* is evaluated as follows:

1. Evaluate *LogicalANDExpression*
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **`false`**, return Result(2).
5. Evaluate *BitwiseORExpression*
6. Call GetValue((Result(5)).
7. Return Result(6).

The production *LogicalORExpression*: *LogicalORExpression* **||** *LogicalANDExpression* is evaluated as follows:

1. Evaluate *LogicalORExpression*
2. Call GetValue(Result(1)).

3. Call ToBoolean(Result(2)).
4. If Result(3) is `true`, return Result(2).
5. Evaluate *LogicalANDExpression*.
6. Call GetValue(Result(5)).
7. Return Result(6).

7.11 CONDITIONAL OPERATOR (`?:`)

Syntax

ConditionalExpression **:**
  LogicalORExpression
  LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

Semantics

The production *ConditionalExpression* **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression* is evaluated as follows:

1. Evaluate *LogicalORExpression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is `false`, go to step 8.
5. Evaluate the first *AssignmentExpression*.
6. Call GetValue(Result(5)).
7. Return Result(6).
8. Evaluate the second *AssignmentExpression*.
9. Call GetValue(Result(8)).
10. Return Result(9).

Issue: Add an explanation of how the grammar differs slightly from that of C and Java here.

7.12 ASSIGNMENT OPERATORS

Syntax

AssignmentExpression **:**
  ConditionalExpression
  PostfixExpression AssignmentOperator AssignmentExpression

AssignmentOperator **:: one of**
  `= *= /= %= += -= <<= >>= >>>= &= ^= |=`

7.12.1 Simple Assignment (`=`)

The production *AssignmentExpression* **:** *UnaryExpression* **=** *AssignmentExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. Evaluate *AssignmentExpression*.
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return Result(3).

7.12.2 Compound Assignment (`op=`)

The production *AssignmentExpression* **:** *UnaryExpression @***=** *AssignmentExpression*, where @ represents one of operators indicated above, is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *AssignmentExpression*.

4. Call GetValue(Result(2)).
5. Apply operator @ to Result(3) and Result(4).
6. Call PutValue(Result(1), Result(5)).
7. Return Result(5).

7.13 COMMA OPERATOR (,)

Syntax

Expression **:**
  AssignmentExpression
  Expression **,** *AssignmentExpression*

Semantics

The production *Expression* **:** *Expression* **,** *AssignmentExpression* is evaluated as follows:

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Evaluate *AssignmentExpression*
4. Call GetValue(Result(3)).
5. Return Result(4).

CHAPTER 8

STATEMENTS

Syntax

Statement **:**
 Block
 VariableStatement
 EmptyStatement
 ExpressionStatement
 IfStatement
 IterationStatement
 ContinueStatement
 BreakStatement
 ReturnStatement
 WithStatement

Block **:**
 { *StatementList*$_{opt}$ **}**

StatementList **:**
 Statement
 StatementList Statement

Semantics

The production *StatementList* **:** *StatementList Statement* is evaluated as follows:

1. Evaluate *StatementList*.
2. Evaluate *Statement*.

8.1 VARIABLESTATEMENT

Syntax

VariableStatement **:**
 var *VariableDeclarationList* **;**

VariableDeclarationList **:**
 VariableDeclaration
 VariableDeclarationList **,** *VariableDeclaration*

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

Initializer **:**
 = *AssignmentExpression*

Description

If the variable statement occurs inside a *FunctionDeclaration*, the variables are defined with function-local scope in that function. Otherwise, they are defined with global scope, that is, they are created as members of the global object as described in section **Error! Reference source not found.** Variables are created when the execution scope is entered. A *Block* does not define a new execution scope. Only *Program* and *FunctionDeclaration* produce a new scope. Eval code and anonymous code also define a new execution scope, but these are not an explicit part of the grammer of ECMAScript. Variables are initialized to the **`undefined`** value when created. A variable with an *Initializer* is assigned the value of its *AssignmentExpression* when the *VariableStatement* is executed.

Semantics
The production *VariableStatement* **:** **`var`** *VariableDeclarationList* **`;`** is evaluated as follows:
1. Evaluate *VariableDeclarationList*
2. Return.

The production *VariableDeclaractionList*: *VariableDeclarationList* **`,`** *VariableDeclaration* is evaluated as follows:
1. Evaluate *VariableDeclarationList*
2. Evaluate *VariableDeclaration*
3. Return.

The production *VariableDeclaration*: *Identifier* **`=`** *AssignmentExpression* is evaluated as follows:
1. Evaluate *Identifier.*
2. Evaluate *AssignmentExpression*
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return.

8.2 EMPTY STATEMENT

Syntax

EmptyStatement **:**
    `;`

Semantics
The production EmptyStatement **:** **`;`** is evaluated by taking no action.

8.3 EXPRESSION STATEMENT

Syntax

ExpressionStatement:
    Expression **`;`**

Semantics
The production *ExpressionStatement*: *Expression* **`;`** is evaluated as follows:
1. Evaluate *Expression*.
2. Call GetValue(Result(1)).

8.4 THE `if` STATEMENT

Syntax

IfStatement **:**
    `if (` *Expression* **`)`** *Statement* **`else`** *Statement*
    `if (` *Expression* **`)`** *Statement*

Semantics
The production *IfStatement* **:** **`if (`** *Expression* **`)`** *Statement*$_1$ **`else`** *Statement*$_2$ is evaluated as follows:
1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **`false`**, go to step 7.
5. Evaluate *Statement*$_1$.
6. Return.
7. Evaluate *Statement*$_2$.

8. Return.

The production *IfStatement* **: `if` `(`** *Expression* **`)`** *Statement* is evaluated as follows:
1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **`false`**, return.
5. Evaluate *Statement*.
6. Return.

8.5 ITERATION STATEMENTS

Syntax

IterationStatement **:**
- **`while (`** *Expression* **`)`** *Statement*
- **`for (`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*
- **`for ( var`** *VariableDeclarationList* **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*
- **`for (`** *Expression* **`in`** *Expression* **`)`** *Statement*
- **`for (`** $\texttt{var}_{opt}$ *Identifier* **`in`** *Expression* **`)`** *Statement*

Description

These statements all define a “continue label” and a “break label” for use by an enclosed **`continue`** or **`break`** statement. For the purposes of this specification, a label is a step number in an algorithm. Continue labels are held in a *continue label stack* and break labels are held in a *break label stack*. These stacks are local to the current execution scope. To execute **`continue`** or **`break`** statement, execution control is transferred to the label specified by the top value of the corresponding label stack. If an implementation of ECMAScript has distinct compile and execute phases, the label stacks need only be maintained during compilation as the label that a **`continue`** or **`break`** statement jumps to is not dependent on any runtime state.

The *WithStatement* affects both stacks for the purposes of clean up: to remove its object from the scope chain.

In algorithms, we use “PushBreak(n)” as short hand for “Push Step(n) on the break label stack”. Similarly we use “PushContinue(n)”, “PopBreak(n)” and “PopContinue(n)” as short hand for the obvious phrases. We use “JumpBreak” as short hand for “Transfer execution control to the position indicated by the top label of the break label stack” and similarly for “JumpContinue”.

8.5.1 The **`while`** Statement

The production *IterationStatement* **: `while (`** *Expression* **`)`** *Statement* is evaluated as follows:
1. PushContinue(3).
2. PushBreak(9).
3. Evaluate *Expression*.
4. Call GetValue(Result(3)).
5. Call ToBoolean(Result(4)).
6. If Result(5) is **`false`**, go to 9.
7. Evaluate *Statement*.
8. Go to step 3.
9. PopBreak(9).
10. PopContinue(3).
11. Return.

8.5.2 The **`for`** Statement

The production *IterationStatement* **: `for (`** $Expression_1$ **`;`** $Expression_2$ **`;`** $Expression_3$ **`)`** *Statement* is evaluated as follows:
1. PushContinue(10).

2. PushBreak(13).
3. Evaluate *Expression*$_1$.
4. Call GetValue(Result(3)).
5. Evaluate *Expression*$_2$.
6. Call GetValue(Result(5)).
7. Call ToBoolean(Result(6)).
8. If Result(7) is **`false`**, go to step 13.
9. Evaluate *Statement*.
10. Evaluate *Expression*$_3$.
11. Call GetValue(Result(10)).
12. Go to step 5.
13. PopBreak(13).
14. PopContinue(10).
15. Return.

If *Expression*$_1$ is omitted from the source text, steps 3 and 4 are omitted from execution. If *Expression*$_2$ is omitted from the source text, step 5 is omitted from execution and the result of step 5 is **`true`**. If *Expression*$_3$ is omitted from the source text, steps 10 and 11 are omitted from execution.

Issue: define the var version.

8.5.3 The **`for..in`** Statement

The production *IterationStatement* **:** **`for (`***Expression*$_1$ **`in`** *Expression*$_2$ **`)`** *Statement* is evaluated as follows:

1. PushContinue(6).
2. PushBreak(11).
3. Evaluate *Expression*$_2$.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(4)).
6. Get the name of the next property of Result(5) which doesn’t have the DontEnum attribute. If there is no such property, go to step 11.
7. Evaluate *Expression*$_1$.
8. Call PutValue(Result(7), Result(6)).
9. Evaluate *Statement*.
10. Go to step 6.
11. PopBreak(11).
12. PopContinue(6).
13. Return.

The mechanics of enumerating the properties (step 6) is implementation dependent. The order of enumeration is defined by the object.   Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration.

Issue: define the var version.

8.6 THE **`continue`** STATEMENT

Syntax

ContinueStatement **:**
  `continue ;`

The **`continue`** statement can only be used when the continue label stack contains at least one label. This is only the case inside a **while**, **for**, or **for..in** loop. The **`continue`** statement is evaluated as:

1. JumpContinue.

See section 8.5 Iteration Statements for a description of the continue label stack and the JumpContinue directive.

8.7 THE `break` STATEMENT

Syntax

BreakStatement **:**
    `break ;`

The **`break`** statement can only be used when the break label stack contains at least one label. This is only the case inside a **while**, **for** or **for..in** loop. The **`break`** statement is evaluated as:

1. JumpBreak

See section 8.5 Iteration Statements for a description of the break label stack and the JumpBreak directive.

8.8 THE `return` STATEMENT

Syntax

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

The **`return`** statement can only be used inside the *Block* of a *FunctionDeclaration*. It causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is the **`undefined`** value. Otherwise, the return value is the value of *Expression*.

8.9 THE `with` STATEMENT

Syntax

WithStatement **:**
    `with (` *Expression* **`)`** *Statement*

Description

The *WithStatement* affects the break label stack and continue label stack for clean up purposes only.

Semantics

The production *WithStatement* **:** **`with (`** *Expression* **`)`** *Statement* is evaluated as follows:

1. If the continue label stack is not empty, PushContinue(12).
2. If the break label stack is not empty, PushBreak(16).
3. Evaluate *Expression*.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(4)).
6. Add Result(5) to the front of the scope chain.
7. Evaluate *Statement*.
8. Remove Result(5) from the front of the scope chain.
9. If the break label stack is not empty, PopBreak(16).
10. If the continue label stack is not empty, PopContinue(12).
11. Return.
12. Remove Result(5) from the front of the scope chain.
13. If the break label stack is not empty, PopBreak(16).
14. PopContinue(12).
15. JumpContinue.
16. Remove Result(5) from the front of the scope chain.
17. PopBreak(16).
18. If the continue label stack is not empty, PopContinue(12).
19. JumpBreak.

Discussion

Most of the complexity of this algorithm is to handle jumps out of the *WithStatement*. Any jumps out of the *WithStatement* must be trapped to remove the object from the scope chain.

CHAPTER 9

FUNCTION DEFINITION

Syntax

FunctionDeclaration:
- **`function`** *Identifier* **`(`** *FormalParameterList*$_{opt}$ **`)`** *Block*

FormalParameterList:
- *Identifier*
- *FormalParameterList* **`,`** *Identifier*

Semantics

Defines a property of the global object whose name is the *Identifier* and whose value is a function object with the given parameter list and statements. If the function definition is supplied text to the **`eval`** function and the calling context has an activation object then the declared function is added to the activation object.

CHAPTER 10

PROGRAM

Syntax

Program **:**
  SourceElements EndOfSource

SourceElements **:**
  SourceElement
  SourceElements SourceElement

SourceElement **:**
  Statement
  FunctionDefinition

CHAPTER 11

NATIVE ECMASCRIPT OBJECTS

There are certain built-in objects available whenever an ECMAScript program begin execution. One, the global object, is in the scope chain of the executing program. Others are accessible as permanent properties of the global object.

Issue: What is a class? What can be used as the operand of the `new` operator?
Theory 1: A class is an object with a [[Construct]] method and a prototype property?
Theory 2: A class is an object with a [[Construct]] method, and the [[Construct]] method creates a prototype property if necessary?
Theory 3: Every function object created by the user automatically has a [[Construct]] method, but other kinds of objects may also have [[Construct]] methods?

For now, I assume that a class is an object that can be given to the `new` operator. I also assume that each built-in class, such as String, has a prototype property (ErrorOnWrite?) that becomes the [[Prototype]] property of every constructed instance of the class. Then, for each class, we have to describe properties of the class, properties of the prototytpe, and properties of each created instance.

How is the [[Prototype]] property of a user-defined class established?

Issue: may also be implementation-dependent objects lying around?

11.1 THE GLOBAL OBJECT

The global object does not have a [[Construct]] property; it is not possible to make instances of the global object using the `new` operator.

11.1 THE OBJECT PROTOTYPE OBJECT

Constructor
[[Get]]
[[Put]]
[[CanPut]]
[[Prototype]]
[[HasProperty]]
[[Construct]]

11.2 THE STRING CLASS

11.2.1 The String Constructor

11.2.2 Properties of the String Class

11.2.3 Properties of the String Prototype Object

11.2.4 Properties of String Instances

11.3 THE NUMBER CLASS

11.3.1 The Number Constructor

11.3.2 Properties of the Number Class

11.3.3 Properties of the Number Prototype Object

11.3.4 Properties of Number Instances

11.4 THE BOOLEAN CLASS

11.5 THE FUNCTION CLASS

11.6 The Array Class

11.7 THE DATE CLASS

11.8 THE MATH OBJECT

The Math object is not a class. It is merely a single object that has some named properties, some of which are functions.

11.8.1 Value Properties of the Math Object

E

LN10

LN2

LOG2E

LOG10E

PI

SQRT1_2

SQRT2

11.8.2 Function Properties of the Math Object

abs(x)

acos(x)

asin(x)

atan(x)

atan2(y, x)

ceil(x)

cos(x)

exp(x)

floor(x)

log(x)

max(x, y)

min(x, y)

pow(base, exponent)

random()

round(x)

sin(x)

sqrt(x)

TAN(X)

CHAPTER 12

ERRORS

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

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

Issue: If a compiler can prove that a construct cannot execute without error under any circumstances, then it may issue a compile-time error even though the construct might not be executed at all?

REFERENCES

ANSI X3.159-1989: *American National Standard for Information Systems - Programming Language - C*, American National Standards Institute (1989).

Gay, David M. Correctly Rounded Binary-Decimal and Decimal -Binary Conversions. Numerical Analysis Manucript 90-10. AT&T Bell Laboratories (Murray Hill, New Jersey). November 30, 1990. Available as **`http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz`**. Associated code available as **`http://cm.bell-labs.com/netlib/fp/dtoa.c.gz`** and as **`http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz`** and may also be found at the various **`netlib`** mirror sites.

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

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

APPENDIX A

OPEN ISSUES

A.1 Break and continue label stacks

The break and continue label stacks and their associated machinery complicate the description of control flow in ECMAScript. Moreover, the current description does not give a clear account of how JumpContinue discards the implicit control stacks that support the execution of the pseudocode procedures in this document.
I would like to propose the rewriting of the behavior of statements into the style used in the Java Language Specification, wherein one speaks of a statement as completing "normally" or "abruptly (for a reason)". The advantage of this descriptive strategy is that then there are no nonlocal transfers within the pseudocode and all descriptions of control flow behavior are local.
As examples, here are accounts of the **`break`**, **`continue`**, **`if`**, and **`while`** statements in this style, which should illustrate all the relevant concepts:

The production *BreakStatement* **`:`** **`break ;`** is evaluated as follows:

```
1.  Return "abrupt completion because of break".
```

The production *ContinueStatement* **`:`** **`continue ;`** is evaluated as follows:

```
1.  Return "abrupt completion because of continue".
```

The production *IfStatement* **`:`** **`if (`** *Expression* **`)`** *Statement*$_1$ **`else`** *Statement*$_2$ is evaluated as follows:

```
1.  Evaluate Expression.
2.  Call GetValue(Result(1)).
3.  Call ToBoolean(Result(2)).
4.  If Result(3) is false, go to step 7.
5.  Evaluate Statement_1.
6.  Return Result(5).
7.  Evaluate Statement_2.
8.  Return Result(7).
```

The production *IterationStatement* **`:`** **`while (`** *Expression* **`)`** *Statement* is evaluated as follows:

```
1.  Evaluate Expression.
2.  Call GetValue(Result(1)).
3.  Call ToBoolean(Result(2)).
4.  If Result(3) is false, go to step 10.
5.  Evaluate Statement.
6.  If Result(5) is "abrupt completion because of break", go to step 10.
7.  If Result(5) is "abrupt completion because of continue", go to step 1.
8.  If Result(5) is "abrupt completion because of return of value V", return Result(5).
9.  Go to step 3.
10. Return "normal completion".
```

Note that the only change to the description of **`if`** is to return the results of substatement evaluation. On the other hand, the description of **`while`** has to take the various kinds of abrupt completion into account. A break causes the **`while`** statement to complete normally; a continue is treated as if the substatement had completed normally; and a return causes the **`while`** statement to terminate immediately and to propagate the return action.

A.2 Eval function

Define object scoping within Eval block.

A.3 Host Supplied members of scope chains vs. Implict **`this`**.

A.4 Escape Sequences in String Literals

It was agreed at a previous meeting that any character could be preceded by a backslash in a string literal. Question: was it intended to allow <CR> or <LF> in a string literal if preceded by abackslash? I assumed not and wrote the grammar accordingly, but would like to have this point discussed.

A.5 Break, Continue, Return in Wrong Place

What is the behavior of an ECMAScript program if it executes a break or continue not textually contained within a loop, or a return not textually within a function body? Are such errors guaranteed to be caught at compile time, or may they be detected at run time? (JavaScript document says it mustbe a compile-time error, Jscript document is less clear.)

A.6 Math Functions

Are the math functions intended to be completely, guaranteed portable, or are they intended to be "whatever the host machine C library provides"? Should the boundary cases (infinites, zero, NaNs) be tied down in the manner now customary for IEEE arithmetic (I believe Java and C9X agree on these boundary cases)?

APPENDIX B

PROPOSED EXTENSIONS

B.1 THE CLASS STATEMENT[1]

Class Definition

Syntax

ClassDeclaration **:**
  `class` *Identifier FormalParameters*$_{opt}$ *ExtendsClause*$_{opt}$ **{** *ClassBody* **}**

FormalParameters **:**
  (*FormalParameterList*$_{opt}$ **)**
*FormalParameterList***:**
  Identifier
  *FormalParameterList***,** *Identifier*

ExtendsClause :
  `extends` *Identifier ActualArguments*$_{opt}$

*ActualArguments***:**
  (*ExpressionList*$_{opt}$ **)**
ClassBody **:**
  Constructor$_{opt}$ *Methods*$_{opt}$

Constructor :
  StatementList

Methods :
  FunctionDefinition
  Methods FunctionDefinition

Semantics

Similar to a function except:

- The class name space is global but distinct from the global function name space.
- The functions (methods) defined within a class definition are in a name space private to the class.
- The inclusion of methods automatically creates one property in the constructed object for each method defined.
- Classes may not be called directly but rather can only be used via the **`new`** operator.

B.2 THE TRY AND THROW STATEMENTS[1]

B.2.1 THE TRY STATEMENT[1]

A **`try`** statement executes a block. If a value is thrown and the **`try`** statement has one or more **`catch`** clauses that can catch it, then control will be transfered to the first such **`catch`** clause. If the **`try`** statement has a **`finally`** clause, then the **`finally`** block of code is executed no matter whether the **`try`** block completes normally or abruptly and regardless of whether a **`catch`** clause is first given control.

TryStatement :
  `try` *Block Catches*

`try` *Block Catchesopt FinallyClause*

Catches:
CatchClause
Catches CatchClause

CatchClause:
`catch` *(FormalParameter)Block*

FinallyClause:
`finally` *Block*

B.2.2 THE THROW STATMENT[1]

A throw statement causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and method invocations until a try statement is found that catches the thrown value. If no such try statement is found, then a runtime error is generated.

ThrowStatement:
`throw` *Expression*

B.3 THE DATE TYPE[1]

The Date Type is used to represent date and time. It is a Julian value on which certain operations such as date arithmetic are defined.Arithmetic operators, relational operators and equality operators apply to this type[1]
Note 1: Of the three current ECMAScript implementations, only the Borland implementation currently supports date operators. This feature is really just a convenience that can be implemented with Date Object methods. However, the same argument can be made for the String type.
Note 2: Of the three current ECMAScript implementations, only the Borland implementation currently implements dates as Julian dates and thus dates before (January 1970). Without this representation, dates are very limited in their usage (i.e. you cannot otherwise, represent arbitrary dates, for example from existing databases)

B.3.1 TODATE[1]

The operator ToDate attempts to convert its argument to a value of subtype Date Object according to the following table:

Input Type	**Result**
Undefined	Blank date value.
Null	Blank date value.
Boolean	Blank date value.
Number	Blank date value.
String	See discussion below.
Date	Return the input argument (no conversion)
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint Date). 2. Call ToDate(Result(1)). Return Result(2).

B.3.1.1 ToDate Applied to the String Type

Issue: define this.

B.4 IMPLICIT THIS[3]

In function code where the function definition specifies the **`implicit`** keyword, the **`this`** object is placed in the scope chain immediately before the global object.

B.5 THE `switch` STATEMENT[1, 3]

Syntax

SwitchStatement **:**
  `switch (` *Expression* **`)`** *CaseBlock*

CaseBlock **:**
  `{` *CaseClauses*$_{opt}$ **`}`**
  `{` *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **`}`**

CaseClauses **:**
  CaseClause
  CaseClauses CaseClause

CaseClause **:**
  `case` *Expression* **:** *StatementList*$_{opt}$

DefaultClause **:**
  `default` **:** *StatementList*$_{opt}$

Semantics

The *SwitchStatement* adds a label to the break label stack, which is described in section 8.5 Iteration Statements. It also adds a label to the continue label stack for clean up purposes only.

The production *SwitchStatement* **:** **`switch (`** *Expression* **`)`** *CaseBlock* is evaluated as follows:

1. If the continue label stack is not empty, PushContinue(9).
2. PushBreak(6).
3. Evaluate *Expression*.
4. Call GetValue(Result(3)).
5. Evaluate *CaseBlock*, passing it Result(4) as a parameter.
6. PopBreak(6).
7. If the continue label stack is not empty, PopContinue(9).
8. Return.
9. PopBreak(6).
10. PopContinue(9).
11. JumpContinue.

The production *CaseBlock* **:** **`{`** *CaseClauses*$_1$ *DefaultClause CaseClauses*$_2$ **`}`** is given an input parameter, *input*, and is evaluated as follows:

1. For the next *CaseClause* in *CaseClauses*$_1$, in source text order, evaluate *CaseClause*. If there is no such *CaseClause*, go to step 6.
2. If *input* is not equal to Result(1) (as defined by the **`!=`** operator), go to step 1.
3. Execute the *StatementList* of this *CaseClause*.
4. Execute the *StatementList* of each subsequent *CaseClause* in *CaseClauses*$_1$.
5. Go to step 11.
6. For the next *CaseClause* in *CaseClauses*$_2$, in source text order, evaluate *CaseClause*. If there is no such *CaseClause*, go to step 11.
7. If *input* is not equal to Result(6) (as defined by the **`!=`** operator), go to step 6.
8. Execute the *StatementList* of this *CaseClause*.
9. Execute the *StatementList* of each subsequent *CaseClause* in *CaseClauses*$_2$.
10. Return.
11. Execute the *StatementList* of *DefaultClause*.
12. Execute the *StatementList* of each *CaseClause* in *CaseClauses*$_2$.

13. Return.

If *CaseClauses*$_1$ is omitted, steps 1 through 5 are omitted from execution. If *DefaultClause* is omitted (in which case *CaseClauses*$_2$ is also omitted), steps 11 and 12 are omitted from execution. If *CaseClauses*$_2$ is omitted, steps 6 through 10 and 12 are omitted from execution.

Typically there will be a **`break`** statement in one or more *StatementList*, which will transfer execution back to the break label for the *SwitchStatement*.

The production *CaseClause* **:** **`case`** *Expression* **:** *StatementList*$_{opt}$ is evaluated as follows:
1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Return Result(2).

Note that evaluating *CaseClause* does not execute the associated *StatementList*. It simply evaluates the *Expression* and returns the value, which the *CaseBlock* algorithm uses to determine which *StatementList* to start executing.

B.6 Conversion Functions

The conversion functions, ToBoolean, ToNumber, ToInteger, ToInt32, ToUint32, ToString and ToObject are global functions that operate as described in this document.

B.7 Assignment-Only Operator (:=)[1]

The assignment-only operator operates identically to the assignment operator (=) except that if the given lvalue doesn't already exist, prior to the statements execution, a runtime error is generated.

B.8 Sealing of an Object[2]

A facility to prevent an object from being further expanded may be invoked at any time after an object has been constructed. This is semantically the dynamic equivalent to the static Java final class modifier. This facility may be implemented as a method of the object, a global function, or, if the **`class`** statement is adopted, as a class modifier to **`class`**. Once an object has been sealed or finalized, any attempt to add a new property to the object results in a runtime error.

B.9 The `arguments` Keyword[3]

The **`arguments`** keyword refers to the arguments object. Within global code, **`arguments`** returns **`null`**. Within eval code, **`arguments`** returns the same value as in the calling context.

Discussion:
This interpretation of the "arguments" within a function body differs from existing practice but has two important advantages over the current mechanism:
1. It can be much more efficiently implemented, especially in the case of recursive functions.
2. It eliminates some complex and confusing semantic issues that arise as a result of the arguments to an activation frame being accessible from a function object.

It solves scope resolution issues related to using arguments within a with block on an object that has an arguments member, such as Math.

B.10 PREPROCESSOR

B.11 THE DO..WHILE STATEMENT

B.12 BINARYOBJECT

APPENDIX C

PEOPLE CONTACTS

Brendan Eich (brendan@netscape.com)
C. Rand McKinney (rand@netscape.com)
Donna Converse (converse@netscape.com)
Clayton Lewis (clayton@netscape.com)
Randy T. Solton (rsolton@wpo.borland.com)
Mike Gardner (mgardner@wpo.borland.com)
Shon Katzenberger (shonk@microsoft.com)
Robert Welland (robwell@microsoft.com)
Guy Steele (guy.steele@east.sun.com)

APPENDIX D

RESOLUTION HISTORY

D.1 JANUARY 15, 1997

D.1.1 White Space

Updated the 3.1 White Space section to include form feed and vertical tab as white space.

D.1.2 Keywords

Updated the 3.3.1.1 Keywords section to exclude those keywords related to proposed extensions. Also updated this section to include the **`delete`** keyword which was missing.

D.1.3 Future Reserved Words

Update the 3.3.1.2 Future Reserved Words to only include keywords related to proposed extensions. We decided to remove words that had been only included as future reserved for Java compatibility purposes.

D.1.4 Octal And Hex Escape Sequence Issue

Decided to support octal and hex notation. Since only two hex digits are used with hex notation, many unicode characters cannot be represented this way. Furthermore, we were not sure if the high 128 characters match up with unicode. (Removed open issue at bottom of section 3.3.4.4 String Literals)

The argument against was that these notations are redundant since any character can be represented using the unicode escape sequence. The arguments for were that hex and octal notation are convenient and simple and also that there is a language tradition to be upheld.

D.1.5 ToPrimitive

Removed the erroneous note stating that errors are never generated as a result of calling ToPrimitive in the 5.1 ToPrimitive section.

D.1.6 Hex in ToNumber

We decided to allow hex in ToNumber but not octal. Looking at it from the user input source point of view, we decided that it was reasonable to use hex but not octal since it might be common to include leading zeros in a user input field. Furthermore we did not believe that the ability to use octal in data entry was desirable. (Removed open issue at the bottom of 5.3.1 ToNumber Applied to the String Type)

D.1.7 Attributes of Declared Functions and Built-in Objets

We decided that built-in objects will have attributes { DontEnum } and that variables declared in global code will have empty attributes. (Updated the 6.1.1 Global Object section)

D.1.8 The Grouping Operator

We decided that the grouping operator would return the result of GetValue() so that the result is never of type reference. (Updated the 7.1.4 The Grouping Operator and removed the open issue at the bottom of this section)

D.1.9 Prefix Increment and Decrement Operators

We decided to not to perform GetValue to the return value and thus leave the algorithm as is. (removed the open issue at the bottom of the 7.3.4 Prefix Increment and Decrement Operators)

D.1.10 Unary Plus

We decided to leave the algorithm for unary plus alone and continue to call GetValue() and ToNumber() after evaluating the unary expression which guarantees a numeric result as opposed to only evaluating the unary expression which would not guarantee a numeric result. (Updated the 7.3.5 Unary **+** and **-** Operators section)

D.1.11 Multiplicative Operators

Updated step nine in the 7.4 Multiplicative Operators section to refer to three new sections 7.41, 7.42 and 7.43 which define the behavior of *****, **/** and **%**.

D.1.12 Additive Operators

Updated step 11 in 7.5.1 and step 10 in 7.5.2 to refer to a new section 7.5.3 which define the behavior of **+** and **-**.

D.1.13 Left Shift Operator

We decided to leave the algorithm for left shift as is, which converts the left operand using ToInt32 rather than ToUint32. Although an unsigned conversion might be arguably preferred, we decided to continue to convert to signed, as we can always add a new operator (<<<) to accomplish an unsigned shift. (Removed the open issue at the bottom of 7.6.1 The Left Shift Operator (**<<**))

D.1.14 Binary Bitwise Operators

We decided to leave the algorithm for the binary bitwise operators as is, which uses signed conversion on the GetValue of its operands. (Removed the open issue at the bottom of 7.9 Binary Bitwise Operators)

D.1.15 Conditional Operator (**? :**)

We decided to leave the algorithm for the conditional operator as is, which performs a GetValue on the result before returning. Current implementations do not do this. (Removed the open issue at the bottom of
7.11 Conditional Operator (`?:`))

D.1.16 Simple Assignment

We decided to leave the algorithm for simple assignment as is. (Removed the open issue at the bottom of 7.12.1 Simple Assignment (**=**))

D.1.17 The **for..in** Statement

We decided to impose no restrictions on Expression1. (Removed the first open issue at the bottom of 8.5.3 The **for..in** Statement)

D.1.18 The **return** Statement

We decided to not generate an error if one return statement in a function returns a value and another return in the same function does not return a value. (Removed the first open issue at the bottom of the 8.8 The **return** Statement The second issue at the bottom of this section has been moved to 3.4 Automatic Semicolon Insertion)

D.1.19 New Proposed Extensions

Sections B.10 Preprocessor, B.11 The do..while Statement and B.12 Binary Object were added.

D.2 JANUARY 24, 1997

D.2.1 END OF SOURCE

Updated SourceCharacter ::
any Unicode *character*
2.2 End Of Source section to describe the end of source token as logical rather than physical \u0000 since strings may contain embedded \u0000 characters.

D.2.2 FUTURE RESERVED WORDS

Updated 3.3.1.2 Future Reserved Words section to include the word **do** and removed the footnotes indicating the origin of the proposed keywords.

D.2.3 WHITE SPACE

Updated 3.1 White Space section. Updated the lexical production for SimpleWhiteSpace to include <VT> and <FF> (already mentioned in the white table above).

D.2.4 COMMENTS

Added new issue to 3.2 regarding nested comments.

D.2.5 IDENTIFIERS

Updated section 3.3.2 to correctly state what is an allowable first character in an identifier.

D.2.6 NUMERIC LITERALS

Updated section 3.3.4.3 Numeric Literals to disallow leading zeros in floating point literals.

D.2.7 STRING LITERALS

Updated the table describing the set of character escape characters in section 3.3.4.4 String Literals, to include a new column indicating the unicode value. Also added a new issue to the end of this section.

D.2.8 AUTOMATIC SEMICOLON INSERTION

Added two new issues to the end of 3.4 Automatic Semicolon Insertion

D.2.9 PROPERTY ATTRIBUTES

Renamed *Permanent* to *DontDelete* in the property attributes table in the 4.5.1 Property Attributes section.

D.2.10 TOPRIMITIVE

Reworded section 5.1 ToPrimitive to better describe the optional hint *PreferredType*.

D.2.11 TONUMBER

Updated section 5.3 ToNumber. Added Hint Number in call to ToPrimitive. Also added new issue to the end of this section.

D.2.12 WHITE SPACE

Updated section 5.3.1 ToNumber Applied to the String Type Updated the lexical production for SimpleWhiteSpace to include <VT> and <FF>.

D.2.13 TONUMBER APPLIED TO THE STRING TYPE

Updated section 5.3.1, ToNumber Applied to the String Type. Reworked lexical productions to be similar to those used in section, 3.3.4.3 Numeric Literals. The difference between string numeric literals and numeric literals is that string numeric literals do not allow octal notation and do allow leading zeros.

D.2.14 ToString

Updated section 5.7 ToString. Added Hint String in call to ToPrimitive.

D.2.15 Postfix Increment and Decrement Operators

Updated section 7.2.2 Postfix Increment and Decrement Operators. Updated the algorithm to return Result(3) (the result of converting ToNumber), rather than (Result(2).

D.2.16 The typeof operator

Added a new issue at the end of section 7.3.3 The **`typeof`** Operator.

D.2.17 Prefix Increment and Decrement Operators

Removed extraneous calls to ToPrimitive from the algorithm in section 7.3.4 Prefix Increment and Decrement Operators.

D.2.18 Multiplicative Operators

Remove step 7 in the algorithm in section 7.4 (either operand NaN) and added a new rule to 7.4.1 and 7.4.2 to reiterate what was in the old step.

D.2.19 The Subtraction Operator

Removed extraneous calls to ToPrimitive from the algorithm in section 7.5.2.

D.2.20 The Subtraction Operator

Remove the old step 9 in the algorithm in section 7.5.2 (either operand NaN) and added a new rule to section 7.5.3 to reiterate what was in the old step.

D.2.21 Applying the Additive Operators (+, -)

Update the last rule in section 7.5.3 to clearly state that operands mentioned in the final sentence must be numeric.

D.2.22 Equality Operators

Moved the Semantic discussion at the beginning of 7.8 to the discussion section at the end of 7.8

D.2.23 ToPrimitive Usage

Added issue at the end of sections 7.5.1 and 7,7.

D.2.34 Binary Logical Operators

Added issue at the end of 7.10.

D.3 January 31, 1997

D.3.1 MultiLine Comment

Updated the lexical production *MultiLineComment* in section
LineEnd ***::***
   LineTerminator
   <EOS>
3.2 Comments, to allow empty multi-line comments. Also removed the issue at the end of this section regarding nested mutli-line comments. The *MultiLineComment* production continues to disallow multi-line comments.

D.3.2 STRING LITERALS

Removed open issue at the end of section 3.3.4.4 String Literals which stated that the maximum string constant supported must be at least 32000 characters long.

D.3.3 AUTOMATIC SEMICOLON INSERTION

Updated section 3.4 Automatic Semicolon Insertion, to include rules governing parsing the `for` statement and dealing with postfix `++` and postfix `--` tokens.

D.3.4 THE NUMBER TYPE

Updated the description in section 4.4 The Number Type.

D.3.5 PUT WITH EXPLICIT ACCESS MODE

Update section 4.5.2.3, Put with Explicit Access Mode to include looking in the prototype object for access violations.

D.3.6 PUT WITH IMPLICIT ACCESS MODE

Update section 4.5.2.4, Put with Implicit Access Mode to include looking in the prototype object for access violations.

D.3.7 THE STRING TYPE

Updated the description in section 4.6, The String Type.

D.3.8 TONUMBER

Updated section 5.3, ToNumber to return a `NaN` for an input type of `Null`.

D.3.9 TONUMBER APPLIED TO THE STRING TYPE

Updated the lexical production for SimpleWhiteSpace in section 5.3.1 to include <CR> and <LF>. Also updated the lexical productions StrFloatingPointLiteral and StrIntegerLiteral to allow signs.

D.3.10 TOINT32

Updated description in section 5.5, ToInt32: (signed 32 bit integer) to tentatively use Guy's Conversion modulo 2^32 algorithm.

D.3.11 TOUINT32

Updated description in section 5.6 ToUint32: (unsigned 32 bit integer) to tentatively use Guy's Conversion modulo 2^32 algorithm.

D.3.12 EXECUTION CONTEXTS (VARIABLES)

Section 6 (Variables) replaced by new section (Execution Contexts).

D.3.13 FUNCTION CALLS

Swapped steps 2 and 3 in section 7.2.4, Function Calls.

D.3.14 THE TYPEOF OPERATOR

Updated the table in section 7.3.3 The `typeof` Operator to specify the result when the input type is an external object. Removed related open issue at the end of this section.

D.3.15 APPLYING THE % OPERATOR

Removed step 7 in the algorithm in section 7.4.(either operand NaN) and added a new rule to 7.4.3 to reiterate what was in the old step.

D.3.16 THE ADDITION OPERATOR (+)

Added the hint Number in the calls to ToPrimitive in section 7.5.1, The Addition Operator (+). Removed related open issue at the end of this section.

D.3.17 RELATIONAL OPERATORS

Added the hint Number in the calls to ToPrimitive in section 7.7, Relational Operators. Removed related open issue at the end of this section.

D.3.18 CONDITIONAL OPERATOR (?:)

Updated the syntactic production, ConditionalExpression, in section 7.11 Conditional Operator (?:)

D.3.19 COMPOUND ASSIGNMENT (OP=)

Swapped steps 2 and 3 in section 7.12.2, Compound Assignment (**op=**)

D.4 FEBRUARY 21, 1997

D.4.1 UNICODE ESCAPE SEQUENCES

Rewrote section 2.1 Unicode to reflect the restriction that non-ASCII Unicode characters may appear only within comments and string literals. Moved the description of Unicode escape sequences to 3.3.4.4 String Literals.

D.4.2 FUTURE RESERVED WORDS

Added **`import`** and **`super`** to table in 3.3.1.2 Future Reserved Words

D.4.3 AUTOMATIC SEMICOLON INSERTION

Rewrote the rules for semicolon insertion in section 3.4 Automatic Semicolon Insertion to incorporate the rule that a semicolon is not inserted if it would be treated as an empty statement. Also, broke out the empty statement as a separate kind of statement for expository purposes in section 8.2 Empty Statement.

D.4.4 THE NUMBER TYPE

Corrected formatting of formulae in section 4.4 The Number Type

D.4.5 NOTIMPLICIT AND NOTEXPLICIT PROPERTY ATTRIBUTES DELETED

The NotImplicit and NotExplicit property attributes were deleted from the table in section 4.5.1 Property Attributes. Many changes throughout the rest of chapter 4 to reflect this deletion. Also, the [[TestPutExplicit]] helper method was renamed [[CanPut]].

D.4.6 TOINT32 AND TOUINT32

Corrected formatting of formulae in section 5.5 ToInt32: (signed 32 bit integer) and section 5.6 ToUint32: (unsigned 32 bit integer). Also, change the discarding of the fractional part to truncate toward zero rather than using a simple floor operation.

Correct an error in the descriptions by adding a new step 4 to each one, which makes sure that if the input is negative zero, the output is positive zero.

D.4.7 GROUPING OPERATOR

Delete step 2 from section 7.1.4 The Grouping Operator. Parentheses no longer force dereferencing.

D.4.8 SHIFT EXPRESSIONS

Correct the grammar for *ShiftExpression* by adding *AdditiveExpression* as an alternative in section 7.6 Bitwise Shift Operators

D.4.9 CONVERSION RULES FOR RELATIONAL OPERATORS

Updated description in section 7.7 Relational Operators so that lexicographic string ordering is used only if both operands become strings when converted to primitive type; if one is a string and one is a number, then numeric ordering is used. Thus relational operators differ from the `+` operator, which, if one operand is a string and one is a number, performs string concatenation rather than addition.

D.4.10 && AND || SEMANTICS

Updated description in section 7.10 Binary Logical Operators so that **`&&`** and **`||`** have PERL-like semantics; that is, the result of **`1||2`** is **`1`**, not true, and the result of **`0||"Hello"`** is **`"Hello"`**.

D.4.11 CONDITIONAL OPERATOR

Updated section 7.11 Conditional Operator (`?:`) to reflect the change that the second and third subexpressions should each be *AssignmentExpression*

D.4.12 ASSIGNMENT OPERATORS

Updated section 7.12 Assignment Operators to reflect the change that the left-hand side of an assignment should be a *PostfixExpression*. Also change two occurrences in subsections of SetVal to PutValue.

D.4.13 SYNTAX OF CLASS STATEMENT

Updated section B.1 The Class Statement1 to allow the parentheses in a class declaration to be optional.

D.4.14 SYNTAX OF TRY STATEMENT

Updated section B.2.1 The `try` Statement1 to require the body of a **`catch`** or **`finally`** clause to be a *Block*.

D.5 FEBRUARY 27, 1997

D.5.1 GRAMMAR NOTATION

Big rewrite of section 1.1 Syntactic and Lexical Grammars to make the description of grammar notation more detailed and rigorous. Is this okay? (Much of the text was borrowed, in form at least, from the Java Language Specification.) The notation is still a bit inconsistent throughout the document (example: "except" versus "but not"), and should be made consistent within itself and with section 1.1 Syntactic and Lexical Grammars.
Also decided to call out the grammar in Chapter 5 as a separate grammar and use triple colons on its productions.
Restructured some of the grammar in Chapter 3 to make it a bit more readable. Is this okay?

D.5.2 END OF MEDIUM CHARACTER IS NO LONGER WHITE SPACE

Deleted character \u0019 (End of Medium) from the table in section 3.1 White Space, and deleted <EOM> as an alternative for SimpleWhiteSpace in that same section. Also deleted <EOM> as an alternative for StrWhiteSpaceChar in section 5.3.1 ToNumber Applied to the String Type These changes reflect the decision that neither \u0019 (End of Medium, mistakenly also referred to in previous drafts of this document as ^Z) nor \u001A (Substitute, which really is ^Z) shall be considered whitespace in an ECMAScript program. It is expected that host environments will filter any ^Z character that might occur at the end of the host environment's representation of an ECMAScript program.

D.5.3 MEANING OF NULL LITERAL

Added to section 3.3.4.1 Null Literals a discussion of the meaning of a null literal.

D.5.4 MEANING OF BOOLEAN LITERALS

Added to section Semantics
The value of the null literal **`true`** is the sole value of the null type, namely **null.**
3.3.4.2 Boolean Literals a discussion of the meaning of a boolean literal.

D.5.5 MEANING OF NUMERIC LITERALS

Added to section 3.3.4.3 Numeric Literals a discussion of the meaning of a numeric literal. It does not yet address the restriction to 19 significant digits. Is this the style of description we want?

D.5.6 AUTOMATIC SEMICOLON INSERTION

Updated description of automatic semicolon insertion in section 3.4 Automatic Semicolon Insertion. Systematically replaced the word “injected” with “inserted”. Invented a new theory of “restricted productions” to explain in a general way why the parser inserts semicolons in places where there would otherwise be a valid parse without a semicolon. Added more examples and advice. Also modified productions in sections 7.2 Postfix Expressions and 8.8 The RETURN `Statement` to indicate the restrictions explicitly.

D.5.7 THE NUMBER TYPE

Updated section 4.4 The Number Type to provide explanations of those large numbers as sums and differences of powers of two.

D.5.8 TOSTRING ON NUMBERS

Updated section 5.7.1 ToString Applied to the Number Type have a draft specification of how this conversion ought to be done. This needs to be reviewed. This version requires that, when the number has a nonzero fractional part, the output must be correctly rounded and produce no more digits than necessary for the fractional part. Added a bibliographic reference to the paper and code of David M. Gay on this subject.

D.5.9 NEW OPERATOR

Updated description in section 7.2.3 The **`new`** Operator to describe the case where no argument list is provided. This needs to be reviewed.

D.5.10 DELETE OPERATOR

Updated description in section 7.3.1 The **`delete`** Operator to reflect decision that this operator shall return a boolean value; the value **`true`** indicates that, after the operation, the object is guaranteed not to have the specified property.

D.5.11 == SEMANTICS

Updated section 7.8 Equality Operators so that (a) **`null`** and **`undefined`** are considered equal, and (b) when a number meets a string, the number is converted to a string and then string equality is used.

D.5.12 && AND || SEMANTICS

Updated description in section 7.10 Binary Logical Operators to delete step 7 for each operator (the result of this step was no longer used).

D.5.13 SEPARATE PRODUCTIONS FOR CONTINUE, BREAK, RETURN

To make certain kinds of cross-reference in the document simpler, I broke out the continue, break, and return statements into separate grammatical productions, eliminating the production for

ControlFlowStatement (which was something of a misnomer anyway, and other statements also result in (structured) control flow.

D.5.14 DEAD CODE IS NOT PROTECTED FROM COMPILE-TIME ANALYSIS

Added text to chapter 12 (Errors).

APPENDIX E

LALR(1) SYNTACTIC GRAMMAR

Issue: To be supplied?

INDEX