

ECMAScript LANGUAGE SPECIFICATION

ECMA COMMITTEE #39

VERSION 0.17

APRIL 14, 1997

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

1 SCOPE
2 CONFORMANCE
3 NORMATIVE REFERENCES
4 OVERVIEW
4.1 WEB SCRIPTING
4.2 LANGUAGE OVERVIEW
4.2.1 <i>Objects</i>
4.3 DEFINITIONS
4.3.1 <i>Type</i>
4.3.2 <i>Primitive Value</i>
4.3.3 <i>Object</i>
4.3.4 <i>Constructor</i>
4.3.5 <i>Prototype</i>
4.3.6 <i>Native Object</i>
4.3.7 <i>Host Object</i>
4.3.8 <i>Undefined</i>
4.3.9 <i>Undefined Type</i>
4.3.10 <i>Null</i>
4.3.11 <i>Null Type</i>
4.3.12 <i>Boolean Value</i>
4.3.13 <i>Boolean Type</i>
4.3.14 <i>Boolean Object</i>
4.3.15 <i>String Value</i>
4.3.16 <i>String Type</i>
4.3.17 <i>String Object</i>
4.3.18 <i>Number Value</i>
4.3.19 <i>Number Type</i>
4.3.20 <i>Number Object</i>
4.3.21 <i>Infinity</i>
4.3.22 <i>NaN</i>
4.4 DEFINITIONS
5 NOTATIONAL CONVENTIONS
5.1 SYNTACTIC AND LEXICAL GRAMMARS
5.1.1 <i>Context-Free Grammars</i>
5.1.2 <i>The Lexical Grammar</i>
5.1.3 <i>The Numeric String Grammar</i>
5.1.4 <i>The Syntactic Grammar</i>
5.1.5 <i>Grammar Notation</i>
5.2 ALGORITHM CONVENTIONS
6 SOURCE TEXT
7 LEXICAL CONVENTIONS
7.1 WHITE SPACE
7.2 LINE TERMINATORS
7.3 COMMENTS
7.4 TOKENS
7.4.1 <i>Reserved Words</i>
7.4.2 <i>Keywords</i>
7.4.3 <i>Future Reserved Words</i>
7.5 IDENTIFIERS
7.6 PUNCTUATORS
7.7 LITERALS
7.7.1 <i>Null Literals</i>
7.7.2 <i>Boolean Literals</i>

7.7.3	<i>Numeric Literals</i>	
7.7.4	<i>String Literals</i>	
7.8	AUTOMATIC SEMICOLON INSERTION.....	
8	TYPES	
8.1	THE UNDEFINED TYPE.....	
8.2	THE NULL TYPE.....	
8.3	THE BOOLEAN TYPE.....	
8.4	THE STRING TYPE.....	
8.5	THE NUMBER TYPE.....	
8.6	THE OBJECT TYPE.....	
8.6.1	<i>Property Attributes</i>	
8.6.2	<i>Internal Properties and Methods</i>	
8.6.2.1	[[Get]](P).....	
8.6.2.2	[[Put]](P, V).....	
8.6.2.3	[[CanPut]](P).....	
8.6.2.4	[[HasProperty]](P).....	
8.7	THE REFERENCE TYPE.....	
8.7.1	<i>GetBase(V)</i>	
8.7.2	<i>GetPropertyName(V)</i>	
8.7.3	<i>GetValue(V)</i>	
8.7.4	<i>PutValue(V, W)</i>	
8.8	THE LIST TYPE.....	
8.9	THE COMPLETION TYPE.....	
9	TYPE CONVERSION	
9.1	TOPRIMITIVE.....	
9.2	TOBOOLEAN.....	
9.3	TONUMBER.....	
9.3.1	<i>ToNumber Applied to the String Type</i>	
9.4	TOINTEGER.....	
9.5	TOINT32: (SIGNED 32 BIT INTEGER).....	
9.6	TOUINT32: (UNSIGNED 32 BIT INTEGER).....	
9.7	TOUINT16: (UNSIGNED 16 BIT INTEGER).....	
9.8	TOSTRING.....	
9.8.1	<i>ToString Applied to the Number Type</i>	
9.9	TOOBJECT.....	
10	EXECUTION CONTEXTS	
10.1	DEFINITIONS.....	
10.1.1	<i>Function Objects</i>	
10.1.2	<i>Types of Executable Code</i>	
10.1.3	<i>Variable Instantiation</i>	
10.1.4	<i>Scope Chain and Identifier Resolution</i>	
10.1.5	<i>Global Object</i>	
10.1.6	<i>Activation Object</i>	
10.1.7	<i>This</i>	
10.1.8	<i>Arguments Object</i>	
10.2	ENTERING AN EXECUTION CONTEXT.....	
10.2.1	<i>Global Code</i>	
10.2.2	<i>EvalCode</i>	
10.2.3	<i>Function and Anonymous Code</i>	
10.2.4	<i>Host Code</i>	
11	EXPRESSIONS	
11.1	PRIMARY EXPRESSIONS.....	
11.1.1	<i>The this Keyword</i>	
11.1.2	<i>Identifier Reference</i>	
11.1.3	<i>Literal Reference</i>	
11.1.4	<i>The Grouping Operator</i>	

11.2 LEFT-HAND-SIDE EXPRESSIONS.....	
11.2.1 Property Accessors.....	
11.2.2 The new Operator.....	
11.2.3 Function Calls.....	
11.2.4 Argument Lists.....	
11.3 POSTFIX EXPRESSIONS.....	
11.3.1 Postfix Increment Operator.....	
11.3.2 Postfix Decrement Operator.....	
11.4 UNARY OPERATORS.....	
11.4.1 The delete Operator.....	
11.4.2 The void Operator.....	
11.4.3 The typeof Operator.....	
11.4.4 Prefix Increment Operator.....	
11.4.5 Prefix Decrement Operator.....	
11.4.6 Unary + Operator.....	
11.4.7 Unary - Operator.....	
11.4.8 The Bitwise NOT Operator (~).....	
11.4.9 Logical NOT Operator (!).....	
11.5 MULTIPLICATIVE OPERATORS.....	
11.5.1 Applying the * Operator.....	
11.5.2 Applying the / Operator.....	
11.5.3 Applying the % Operator.....	
11.6 ADDITIVE OPERATORS.....	
11.6.1 The Addition Operator (+).....	
11.6.2 The Subtraction Operator (-).....	
11.6.3 Applying the Additive Operators (+, -) to Numbers.....	
11.7 BITWISE SHIFT OPERATORS.....	
11.7.1 The Left Shift Operator (<<).....	
11.7.2 The Signed Right Shift Operator (>>).....	
11.7.3 The Unsigned Right Shift Operator (>>>).....	
11.8 RELATIONAL OPERATORS.....	
11.9 EQUALITY OPERATORS.....	
11.10 BINARY BITWISE OPERATORS.....	
11.11 BINARY LOGICAL OPERATORS.....	
11.12 CONDITIONAL OPERATOR (?):.....	
11.13 ASSIGNMENT OPERATORS.....	
11.13.1 Simple Assignment (=).....	
11.13.2 Compound Assignment (op=).....	
11.14 COMMA OPERATOR (,).....	
12 STATEMENTS.....	
12.1 VARIABLE STATEMENT.....	
12.2 EMPTY STATEMENT.....	
12.3 EXPRESSION STATEMENT.....	
12.4 THE if STATEMENT.....	
12.5 ITERATION STATEMENTS.....	
12.5.1 The while Statement.....	
12.5.2 The for Statement.....	
12.5.3 The for..in Statement.....	
12.6 THE continue STATEMENT.....	
12.7 THE break STATEMENT.....	
12.8 THE return STATEMENT.....	
12.9 THE with STATEMENT.....	
13 FUNCTION DEFINITION.....	
14 PROGRAM.....	
15 NATIVE ECMASCRIPT OBJECTS.....	
15.1 THE GLOBAL OBJECT.....	

15.1.1	<i>Value Properties of the Global Object</i>	
15.1.1.1	NaN.....	
15.1.1.2	Infinity.....	
15.1.2	<i>Function Properties of the Global Object</i>	
15.1.2.1	eval(x).....	
15.1.2.2	parseInt(string, radix).....	
15.1.2.3	parseFloat(string).....	
15.1.2.4	escape(string).....	
15.1.2.5	unescape(string).....	
15.1.2.6	isNaN(number).....	
15.1.2.7	isFinite(number).....	
15.2	OBJECT OBJECTS.....	
15.2.1	<i>The Object Function</i>	
15.2.1.1	Object(value).....	
15.2.1.2	Object().....	
15.2.2	<i>The Object Constructor</i>	
15.2.2.1	new Object(value).....	
15.2.2.2	new Object().....	
15.2.3	<i>Properties of the Object Constructor</i>	
15.2.3.1	Object.prototype.....	
15.2.4	<i>Properties of the Object Prototype Object</i>	
15.2.4.1	constructor.....	
15.2.4.2	toString().....	
15.2.4.3	valueOf().....	
15.2.5	<i>Properties of Object Instances</i>	
15.3	FUNCTION OBJECTS.....	
15.3.1	<i>The Function Function</i>	
15.3.1.1	Function(p1, p2, . . . , pn, body).....	
15.3.2	<i>The Function Constructor</i>	
15.3.2.1	Function(p1, p2, . . . , pn, body).....	
15.3.3	<i>Properties of the Function Constructor</i>	
15.3.3.1	Function.prototype.....	
15.3.3.2	Function.length.....	
15.3.4	<i>Properties of the Function Prototype Object</i>	
15.3.4.1	constructor.....	
15.3.4.2	toString().....	
15.3.4.3	valueOf().....	
15.3.5	<i>Properties of Function Instances</i>	
15.3.5.1	length.....	
15.3.5.2	prototype.....	
15.3.5.3	arguments.....	
15.4	ARRAY OBJECTS.....	
15.4.1	<i>The Array Constructor</i>	
15.4.1.1	new Array(item0, item1, . . .).....	
15.4.1.2	new Array(len).....	
15.4.1.3	new Array().....	
15.4.2	<i>Properties of the Array Constructor</i>	
15.4.2.1	Array.prototype.....	
15.4.2.2	Array.length.....	
15.4.3	<i>Properties of the Array Prototype Object</i>	
15.4.3.1	constructor.....	
15.4.3.2	toString().....	
15.4.3.3	valueOf().....	
15.4.3.4	join(separator).....	
15.4.3.5	reverse().....	
15.4.3.6	sort().....	
15.4.4	<i>Properties of Array Instances</i>	
15.4.4.1	[[Put]](P, V).....	
15.4.4.2	length.....	
15.5	STRING OBJECTS.....	
15.5.1	<i>The String Function</i>	
15.5.1.1	String(value).....	
15.5.1.2	String().....	
15.5.2	<i>The String Constructor</i>	
15.5.2.1	new String(value).....	

15.5.2.2	new String()
15.5.3	<i>Properties of the String Constructor</i>
15.5.3.1	String.prototype
15.5.3.2	String.fromCharCode(char0, char1, ...)
15.5.4	<i>Properties of the String Prototype Object</i>
15.5.4.1	constructor
15.5.4.2	toString()
15.5.4.3	valueOf()
15.5.4.4	charAt(pos)
15.5.4.5	charCodeAt(pos)
15.5.4.6	indexOf(searchString, position)
15.5.4.7	lastIndexOf(searchString, position)
15.5.4.8	split(separator)
15.5.4.9	substring(start)
15.5.4.10	substring(start, end)
15.5.4.11	toLowerCase
15.5.4.12	toUpperCase
15.5.5	<i>Properties of String Instances</i>
15.5.5.1	length
15.6	BOOLEAN OBJECTS
15.6.1	<i>The Boolean Function</i>
15.6.1.1	Boolean(value)
15.6.1.2	Boolean()
15.6.2	<i>The Boolean Constructor</i>
15.6.2.1	new Boolean(value)
15.6.2.2	new Boolean()
15.6.3	<i>Properties of the Boolean Constructor</i>
15.6.3.1	Boolean.prototype
15.6.4	<i>Properties of the Boolean Prototype Object</i>
15.6.4.1	constructor
15.6.4.2	toString()
15.6.4.3	valueOf()
15.6.5	<i>Properties of Boolean Instances</i>
15.7	NUMBER OBJECTS
15.7.1	<i>The Number Function</i>
15.7.1.1	Number(value)
15.7.1.2	Number()
15.7.2	<i>The Number Constructor</i>
15.7.2.1	new Number(value)
15.7.2.2	new Number()
15.7.3	<i>Properties of the Number Constructor</i>
15.7.3.1	Number.prototype
15.7.3.2	Number.MAX_VALUE
15.7.3.3	Number.MIN_VALUE
15.7.3.4	Number.NaN
15.7.3.5	Number.NEGATIVE_INFINITY
15.7.3.6	Number.POSITIVE_INFINITY
15.7.4	<i>Properties of the Number Prototype Object</i>
15.7.4.1	constructor
15.7.4.2	toString()
15.7.4.3	valueOf()
15.7.5	<i>Properties of Number Instances</i>
15.8	THE MATH OBJECT
15.8.1	<i>Value Properties of the Math Object</i>
15.8.1.1	E
15.8.1.2	LN10
15.8.1.3	LN2
15.8.1.4	LOG2E
15.8.1.5	LOG10E
15.8.1.6	PI
15.8.1.7	SQRT1_2
15.8.1.8	SQRT2
15.8.2	<i>Function Properties of the Math Object</i>
15.8.2.1	abs(x)
15.8.2.2	acos(x)
15.8.2.3	asin(x)

15.8.2.4	atan(x)
15.8.2.5	atan2(y, x)
15.8.2.6	ceil(x)
15.8.2.7	cos(x)
15.8.2.8	exp(x)
15.8.2.9	floor(x)
15.8.2.10	log(x)
15.8.2.11	max(x, y)
15.8.2.12	min(x, y)
15.8.2.13	pow(x, y)
15.8.2.14	random()
15.8.2.15	round(x)
15.8.2.16	sin(x)
15.8.2.17	sqrt(x)
15.8.2.18	tan(x)
15.9	DATE OBJECTS
15.9.1	<i>Overview of Date Objects</i>
15.9.1.1	Time Range
15.9.1.2	Day Number and Time within Day
15.9.1.3	Year Number
15.9.1.4	Month Number
15.9.1.5	Date Number
15.9.1.6	Week Day
15.9.1.7	Local Time Zone Adjustment
15.9.1.8	Daylight Saving Time Adjustment
15.9.1.9	Local Time
15.9.1.10	Hours, Minutes, Second, and Milliseconds
15.9.1.11	MakeTime(hour, min, sec, ms)
15.9.1.12	MakeDay(year, month, date)
15.9.1.13	MakeDate(day, time)
15.9.1.14	TimeClip(time)
15.9.2	<i>The Date Constructor</i>
15.9.2.1	new Date(year, month, date, hours, minutes, seconds)
15.9.2.2	new Date(year, month, date, hours, minutes)
15.9.2.3	new Date(year, month, date, hours)
15.9.2.4	new Date(year, month, day)
15.9.2.5	new Date(value)
15.9.2.6	new Date()
15.9.3	<i>Properties of the Date Constructor</i>
15.9.3.1	Date.prototype
15.9.3.2	Date.parse()
15.9.3.3	Date.UTC()
15.9.4	<i>Properties of the Date Prototype Object</i>
15.9.4.1	constructor
15.9.4.2	toString()
15.9.4.3	valueOf()
15.9.4.4	getTime()
15.9.4.5	getFullYear()
15.9.4.6	getFullYear()
15.9.4.7	getUTCFullYear()
15.9.4.8	getMonth()
15.9.4.9	getUTCMonth()
15.9.4.10	getDate()
15.9.4.11	getUTCDate()
15.9.4.12	getDay()
15.9.4.13	getUTCDay()
15.9.4.14	getHours()
15.9.4.15	getUTCHours()
15.9.4.16	getMinutes()
15.9.4.17	getUTCMinutes()
15.9.4.18	getSeconds()
15.9.4.19	getUTCSeconds()
15.9.4.20	getMilliseconds()
15.9.4.21	getUTCMilliseconds()
15.9.4.22	getTimezoneOffset()
15.9.4.23	setTime(time)
15.9.4.24	setMilliseconds(ms)

- 15.9.4.25 setUTCMilliseconds(ms).....
- 15.9.4.26 setSeconds(sec [, ms]).....
- 15.9.4.27 setUTCSeconds(sec [, ms]).....
- 15.9.4.28 setMinutes(min [, sec [, ms]]).....
- 15.9.4.29 setUTCMinutes(min [, sec [, ms]]).....
- 15.9.4.30 setHours(hour [, min [, sec [, ms]]]).....
- 15.9.4.31 setUTCHours(hour [, min [, sec [, ms]]]).....
- 15.9.4.32 setDate(date).....
- 15.9.4.33 setUTCDate(date).....
- 15.9.4.34 setMonth(mon [, date]).....
- 15.9.4.35 setUTCMonth(mon [, date]).....
- 15.9.4.36 setFullYear(year [, mon [, date]]).....
- 15.9.4.37 setUTCFullYear(year [, mon [, date]]).....
- 15.9.4.38 setYear(year).....
- 15.9.5 *Properties of Date Instances*.....

16 ERRORS.....

17 REFERENCES.....

APPENDIX A: OPEN ISSUES.....

- A.1 STRING NUMERIC LITERALS.....
- A.2 ARGUMENT.....
- A.3 VAR STATEMENTS AND EVAL.....
- A.4 USING ARRAY METHODS ON NON-ARRAYS.....
- A.5 REVERSE OF NONEXISTENT PROPERTIES.....

APPENDIX B: PROPOSED EXTENSIONS.....

- B.1 THE CLASS STATEMENT¹.....
- B.2 THE TRY AND THROW STATEMENTS¹.....
 - B.2.1 *The try Statement*¹.....
 - B.2.2 *The Throw Statment*¹.....
- B.3 THE DATE TYPE¹.....
 - B.3.1 *ToDate*¹.....
 - B.3.2 *ToDate Applied to the String Type*.....
- B.4 IMPLICIT THIS³.....
- B.5 THE switch STATEMENT^{1,3}.....
- B.6 CONVERSION FUNCTIONS.....
- B.7 ASSIGNMENT-ONLY OPERATOR (:=)¹.....
- B.8 SEALING OF AN OBJECT².....
- B.9 THE ARGUMENTS KEYWORD³.....
- B.10 PREPROCESSOR.....
- B.11 THE DO..WHILE STATEMENT.....
- B.12 BINARY OBJECT.....
- B.13 LABELS WITH BREAK AND CONTINUE.....

APPENDIX C: PEOPLE CONTACTS.....

APPENDIX D: RESOLUTION HISTORY.....

- D.1 JANUARY 15, 1997.....
 - D.1.1 *White Space*.....
 - D.1.2 *Keywords*.....
 - D.1.3 *Future Reserved Words*.....
 - D.1.4 *Octal And Hex Escape Sequence Issue*.....
 - D.1.5 *ToPrimitive*.....
 - D.1.6 *Hex in ToNumber*.....
 - D.1.7 *Attributes of Declared Functions and Built-in Objects*.....
 - D.1.8 *The Grouping Operator*.....
 - D.1.9 *Prefix Increment and Decrement Operators*.....
 - D.1.10 *Unary Plus*.....
 - D.1.11 *Multiplicative Operators*.....
 - D.1.12 *Additive Operators*.....

D.1.13	Left Shift Operator.....	
D.1.14	Binary Bitwise Operators.....	
D.1.15	Conditional Operator (? :).....	
D.1.16	Simple Assignment.....	
D.1.17	The for..in Statement.....	
D.1.18	The return Statement.....	
D.1.19	New Proposed Extensions.....	
D.2	JANUARY 24, 1997.....	
D.2.1	End Of Source.....	
D.2.2	Future Reserved Words.....	
D.2.3	White Space.....	
D.2.4	Comments.....	
D.2.5	Identifiers.....	
D.2.6	Numeric Literals.....	
D.2.7	String Literals.....	
D.2.8	Automatic Semicolon Insertion.....	
D.2.9	Property Attributes.....	
D.2.10	ToPrimitive.....	
D.2.11	ToNumber.....	
D.2.12	White Space.....	
D.2.13	ToNumber Applied to the String Type.....	
D.2.14	ToString.....	
D.2.15	Postfix Increment and Decrement Operators.....	
D.2.16	The typeof operator.....	
D.2.17	Prefix Increment and Decrement Operators.....	
D.2.18	Multiplicative Operators.....	
D.2.19	The Subtraction Operator.....	
D.2.20	The Subtraction Operator.....	
D.2.21	Applying the Additive Operators (+, -).....	
D.2.22	Equality Operators.....	
D.2.23	ToPrimitive Usage.....	
D.2.24	Binary Logical Operators.....	
D.3	JANUARY 31, 1997.....	
D.3.1	MultiLineComment.....	
D.3.2	String Literals.....	
D.3.3	Automatic Semicolon Insertion.....	
D.3.4	The Number Type.....	
D.3.5	Put with Explicit Access Mode.....	
D.3.6	Put with Implicit Access Mode.....	
D.3.7	The String type.....	
D.3.8	ToNumber.....	
D.3.9	ToNumber Applied to the String Type.....	
D.3.10	ToInt32.....	
D.3.11	ToUint32.....	
D.3.12	Execution Contexts (Variables).....	
D.3.13	Function Calls.....	
D.3.14	The typeof Operator.....	
D.3.15	Applying the % Operator.....	
D.3.16	The Addition Operator (+).....	
D.3.17	Relational Operators.....	
D.3.18	Conditional Operator (? :).....	
D.3.19	Compound Assignment (op=).....	
D.4	FEBRUARY 21, 1997.....	
D.4.1	Unicode Escape Sequences.....	
D.4.2	Future Reserved Words.....	
D.4.3	Automatic Semicolon Insertion.....	
D.4.4	The Number Type.....	
D.4.5	NotImplicit and NotExplicit Property Attributes Deleted.....	
D.4.6	ToInt32 and ToUint32.....	
D.4.7	Grouping Operator.....	

D.4.8	Shift Expressions.....	
D.4.9	Conversion Rules for Relational Operators.....	
D.4.10	&& and Semantics.....	
D.4.11	Conditional Operator.....	
D.4.12	Assignment Operators.....	
D.4.13	Syntax of Class Statement.....	
D.4.14	Syntax of Try Statement.....	
D.5	FEBRUARY 27, 1997.....	
D.5.1	Grammar Notation.....	
D.5.2	End of Medium Character Is No Longer WhiteSpace.....	
D.5.3	Meaning of Null Literal.....	
D.5.4	Meaning of Boolean Literals.....	
D.5.5	Meaning of Numeric Literals.....	
D.5.6	Automatic Semicolon Insertion.....	
D.5.7	The Number Type.....	
D.5.8	ToString on Numbers.....	
D.5.9	New Operator.....	
D.5.10	Delete Operator.....	
D.5.11	== Semantics.....	
D.5.12	&& and Semantics.....	
D.5.13	Separate Productions for Continue, Break, Return.....	
D.5.14	Dead Code Is Not Protected from Compile-Time Analysis.....	
D.6	MARCH 6, 1997.....	
D.6.1	Reformatted the Entire Document.....	
D.6.2	Designed a Section Outline for Chapter 11.....	
D.6.3	Defined Math Functions.....	
D.7	MARCH 10, 1997.....	
D.7.1	Added Definition of The Number Value for x^O	
D.7.2	atan and atan2 May Use Implementation-Dependent Values for π , etc.....	
D.7.3	Improved Discussion of Input Stream for Syntactic Grammar.....	
D.7.4	Improved Treatment of LineTerminator in Lexical Grammar.....	
D.7.5	Clarify Behavior of Unicode Escape Sequences.....	
D.7.6	Add Careful Description of the String Value of a String Literal.....	
D.7.7	Description of Identifiers Reworded.....	
D.7.8	Table of Punctuators Corrected.....	
D.7.9	Improved Descriptions of ToInt32 and ToUInt32.....	
D.7.10	Changes to ToString Applied to the Number Type.....	
D.7.11	Revised Syntax for NewExpression and MemberExpression.....	
D.7.12	Clarify Multiplicative and Additive Operators.....	
D.7.13	Addition Operator No Longer Gives Hint Number.....	
D.7.14	Correct Description of Relational Operators.....	
D.7.15	Assignment Operator LHS Must Be PostfixExpression.....	
D.7.16	Changes to For-in Loops.....	
D.7.17	Break and Continue Must Occur within While or For Loop.....	
D.8	MARCH 12, 1997.....	
D.8.1	Added Overview Chapter.....	
D.8.2	More Exposition about Internal Properties.....	
D.8.3	Date Object.....	
D.8.4	Array, String, Boolean, Number Objects.....	
D.8.5	Math Object.....	
D.9	MARCH 24, 1997.....	
D.9.1	Numeric Literals.....	
D.9.2	String Numeric Literals.....	
D.9.3	Prefix and Postfix Increment and Decrement Operators.....	
D.9.4	Left-Hand-Side Expressions.....	
D.9.5	Reference Type.....	
D.9.6	Infinities and Zeros.....	
D.9.7	Miscellaneous Small Corrections.....	
D.10	MARCH 27, 1997.....	
D.10.1	Corrections to [[CanPut]] and [[HasProperty]].....	

D.10.2	Discussion of Number Type.....
D.10.3	Infinity and NaN.....
D.10.4	charCodeAt and String.fromCharCode.....
D.10.5	Last fraction digit from ToString applied to a number.....
D.10.6	Multi-line comment containing line terminator treated as line terminator.....
D.10.7	Automatic semicolon insertion at end of source.....
D.10.8	Added proposed extension for labelled break and continue.....
D.10.9	Lowercase <i>o</i> for scientific notation in ToString of a number.....
D.10.10	Evaluation of argument lists.....
D.10.11	For ToPrimitive of native objects, no hint is same as hint Number.....
D.10.12	Major overhaul of equality and relational operators.....
D.10.13	String type.....
D.11	APRIL 9, 1997.....
D.11.1	Added mathematical operators to notation section.....
D.11.2	Added overview text.....
D.11.3	Added Date stuff.....
D.11.4	Lots of work on native objects chapter.....
D.12	APRIL 14, 1997.....
D.12.1	Lots of corrections.....
D.12.2	Reworked Date type.....
APPENDIX E: LALR(1) SYNTACTIC GRAMMAR.....	

1SCOPE

2 CONFORMANCE

3 NORMATIVE REFERENCES

4 OVERVIEW

ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host* objects, whose description and behavior are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

A *scripting language* is a programming language that is used to manipulate, customize, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scripting language is a mechanism for exposing that functionality to program control. In this way, the existing system is said to provide a host environment of objects and facilities which completes the capabilities of the scripting language. A scripting language is intended for use by both professional and non-professional programmers, and therefore there may be a number of informalities and built into the language.

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

4.1 WEB SCRIPTING

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

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

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

4.2 LANGUAGE OVERVIEW

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

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. An ECMAScript *object* is an unordered collection of *properties* each with 0 or more *attributes* which determine how each property can be used. For example, when the *ReadOnly* attribute for a property is set to true, the property may be only read. Properties are containers that hold other objects, *primitive values*, or *methods*. A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, and **String**; an object is a member of the remaining built-in type **Object**; and a method is a function associated with an object via a property.

ECMAScript defines a collection of *native objects* which round out the definition of ECMAScript entities. These native objects are the **Global** object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, and the **Date** object.

ECMAScript also defines a set of built-in *operators* which may not be, strictly speaking, functions or methods. ECMAScript operators include various unary operations, multiplicative operators, additive operators, bitwise shift operators, relational operators, equality operators, binary bitwise operators, binary logical operators, assignment operators, and the comma operator.

ECMAScript syntax intentionally resembles Java syntax. ECMAScript syntax is relaxed to enable it to serve as an easy-to-use scripting language. For example, a variable is not required to have its type declared nor are types associated with properties, and if the host environment permits, functions are not required to be defined before they are used

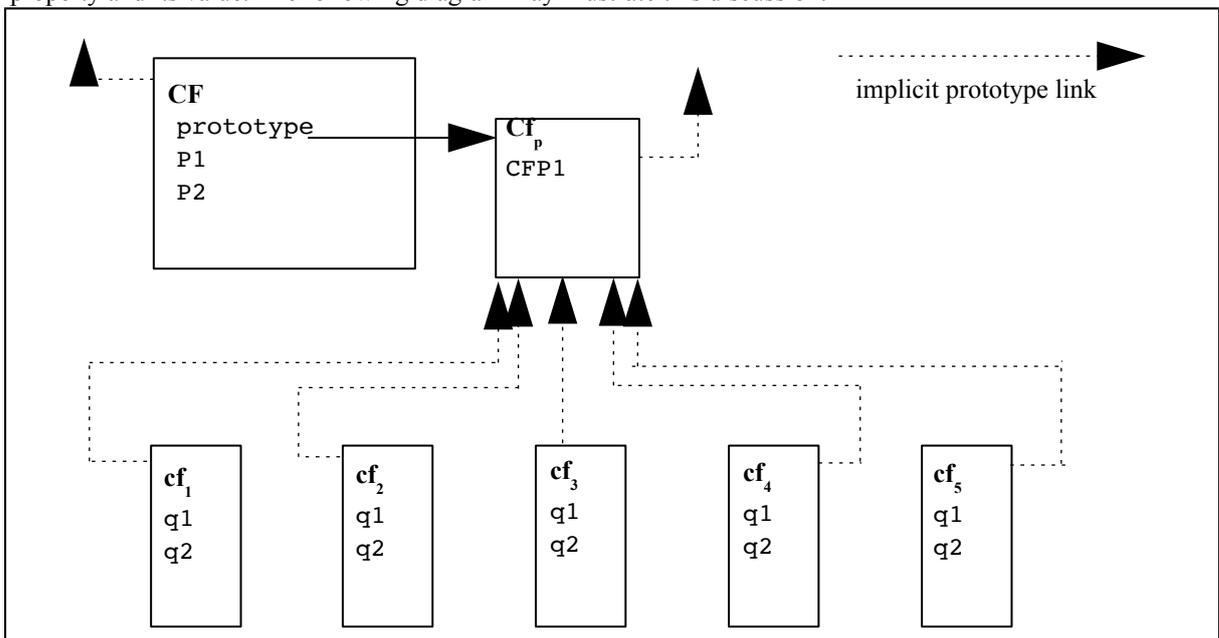
4.2.1 Objects

ECMAScript does not contain proper classes such as those in C++, Smalltalk, or Java, but rather, supports *constructors* which create objects by executing code that allocates storage for the objects and initializes all or part of them by assigning initial values to their properties. All functions including constructors are objects, but not all objects are constructors. Each constructor has a **Prototype** property which is used to implement *prototype-based inheritance* and *shared properties*. Objects are created by using constructors in **new** expressions, for example, `new String("A String")` creates a new string object. Invoking a constructor without using **new** has consequences that depend on the constructor. For example, `String("A String")` produces a primitive string, not an object.

ECMAScript supports *prototype-based inheritance*. Every constructor has an associated prototype, and every object created by that constructor has an implicit reference to the prototype (called the *object's prototype*) associated with its constructor. Furthermore, a prototype may have a non-null implicit reference to its prototype, and so on; this is called the *prototype chain*. When a reference is made to a property in an object, that reference is to the property of that name in the first object in the prototype chain that contains a property of that name. In other words, first the object mentioned directly is examined for such a property; if that object contains the named property, that is the property to which the reference refers; if that object does not contain the named property, the prototype for that object is examined next; and so on.

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

All objects that do not directly contain a particular property that their prototype contains share that property and its value. The following diagram may illustrate this discussion:



CF is a constructor (and also an object). Five objects have been created by using **new** expressions: **cf₁**, **cf₂**, **cf₃**, **cf₄**, and **cf₅**. Each of these objects contains properties named **q1** and **q2**. The dashed lines

represent the implicit prototype relationship; so, for example, cf_3 's prototype is CF_p . The constructor, CF , has two properties itself, named $P1$ and $P2$, which are not visible to CF_p , cf_1 , cf_2 , cf_3 , cf_4 , or cf_5 . The property named $CFP1$ in CF_p is shared by: cf_1 , cf_2 , cf_3 , cf_4 , and cf_5 , as are any properties found in CF_p 's implicit prototype chain which are not named $q1$, $q2$, or $CFP1$. Notice that there is no implicit prototype link between CF_p and CF .

Unlike class-based object languages, properties can be added to objects on the fly simply by assigning values to them. That is, constructors are not required to name or assign values to all or any of its properties. In the above diagram, one could add a new shared property for cf_1 , cf_2 , cf_3 , cf_4 , and cf_5 by assigning a new value to the property in CF_p .

4.3 DEFINITIONS

The following are informal definitions of key terms associated with ECMAScript.

4.3.1 Type

A *type* is a set of data values. In general, the correct functioning of a program is not affected if different data values of the same type are substituted for others.

4.3.2 Primitive Value

A *primitive value* is a member of one of the types **Undefined**, **Null**, **Boolean**, **Number**, or **String**. A primitive value is a data object which is represented directly at the lowest level of the language implementation.

4.3.3 Object

An *object* is a member of the type **Object**. It is an unordered collection of properties which contain primitive values, objects, or functions. A function stored in the property of an object is called a method.

4.3.4 Constructor

A *constructor* is a function object which creates and initializes objects. Each constructor has an associated prototype object which is used to implement inheritance and shared properties.

4.3.5 Prototype

A *prototype* is an object used to implement structure, state, and behavior inheritance in ECMAScript. When a constructor creates an object, that object implicitly references the constructor's associated prototype for the purpose of resolving property references. The constructor's associated prototype can be referenced by the program expression `constructor.prototype`, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype.

4.3.6 Native Object

A *native object* is any object supplied by an ECMAScript implementation independent of the host environment. Standard native objects are defined in this specification, and the ECMAScript implementation may specify and define others.

4.3.7 Host Object

A *host object* is any object supplied by the host environment to complete the execution environment of ECMAScript.

4.3.8 Undefined

Undefined is a primitive value used when a variable has not been assigned a value.

4.3.9 Undefined Type

The type **Undefined** has exactly one value, called **Undefined**.

4.3.10 Null

Null is a primitive value that represents the null, empty, or nonexistent reference.

4.3.11 Null Type

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

4.3.12 Boolean Value

A *boolean value* is a member of the type **Boolean** and is one of either two unique values, **true** and **false**.

4.3.13 Boolean Type

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

4.3.14 Boolean Object

A *Boolean object* is a member of the type **Object** and is an instance of the Boolean object which is a constructor. That is, a boolean object is created by using the Boolean constructor in a new expression, supplying a boolean as an argument. The resulting object has an implicit (unnamed) property which is the boolean. A boolean object can be coerced to a boolean value. A boolean object can be used anywhere a boolean value is expected.

This is an example of one of the conveniences built into ECMAScript. In this case it is to accommodate programmers of varying backgrounds. Those familiar with imperative or procedural programming languages may find number values more natural, while those familiar with object-oriented languages may find number objects more intuitive.

4.3.15 String Value

A *string value* is a member of the type **String** and is the set of all finite ordered sequences of zero or more unicode characters.

4.3.16 String Type

The type **String** is the set of all finite ordered sequences of zero or more unicode characters.

4.3.17 String Object

A *string object* is a member of the type **Object** and is an instance of the String object which is a constructor. That is, a string object is created by using the String constructor in a new expression, supplying a string as an argument. The resulting object has an implicit (unnamed) property which is the string. A string object can be coerced to a string value. A string object can be used anywhere a string value is expected.

4.3.18 Number Value

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

4.3.19 Number Type

The type **Number** is a set of values representing numbers. In ECMAScript the set of values represent the double-precision 64-bit format IEEE 754 value along with a special "Not-a-Number" (NaN) value, positive infinity, and negative infinity.

4.3.20 Number Object

A *number object* is a member of the type **Object** and is an instance of the Number object which is a constructor. That is, a number object is created by using the Number constructor in a new expression, supplying a number as an argument. The resulting object has an implicit (unnamed) property which is the number. A number object can be coerced to a number value. A number object can be used anywhere

a number value is expected. Note that a number object can have shared properties by adding them to the Number prototype.

4.3.21 Infinity

The primitive value `Infinity` represents the positive infinite number value.

4.3.22 NaN

The primitive value `NaN` represents the set of IEEE Standard "Not-a-Number" values.

4.4 DEFINITIONS

object

primitive value

number value

Number object (and *a* Number object versus *the* Number object)

Number type

string value

String object (and *a* String object versus *the* String object)

String type

boolean value

Boolean object (and *a* Boolean object versus *the* Boolean object)

Boolean type

null

Null type

type

undefined

undefined type

infinity

NaN

prototype

constructor

host object

native object

built-in object

5 NOTATIONAL CONVENTIONS

5.1 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.

5.1.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of 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 (perhaps infinite) set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

5.1.2 The Lexical 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.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion. Simple white space and single-line comments are simply discarded and do not appear in the stream of input elements for the syntactic grammar. A multi-line comment is likewise simply discarded if it contains no line terminator; but if a multi-line comment contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

Productions of the lexical grammar are distinguished by having two colons `::=` as separating punctuation.

5.1.3 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 and has as its terminal symbols the characters of the Unicode character set. This grammar appears in Chapter 5.

Productions of the numeric string grammar are distinguished by having three colons `::=::` as punctuation.

5.1.4 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 (see section 5.1.2). It defines a set of productions, starting from the goal symbol *Program*, that describe how sequences of tokens can form syntactically correct ECMAScript programs.

When a stream of Unicode characters is to be parsed as an ECMAScript program, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements

is then parsed by a single application of the syntax grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *program*, with no tokens left over.

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.

5.1.5 Grammar Notation

Terminal symbols of the lexical and string grammars, and some of the terminal symbols of the syntactic grammar, are shown in **fixed width** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a 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:

```
WithStatement :  
    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 Initializeropt
```

is a convenient abbreviation for:

```
VariableDeclaration :  
    Identifier  
    Identifier Initializer
```

and that:

```
IterationStatement :  
    for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
```

is a convenient abbreviation for:

IterationStatement :

```

for ( ; Expressionopt ; Expressionopt ) Statement
for ( Expression ; Expressionopt ; Expressionopt ) Statement

```

which in turn is an abbreviation for:

IterationStatement :

```

for ( ; ; Expressionopt ) Statement
for ( ; Expression ; Expressionopt ) Statement
for ( Expression ; ; Expressionopt ) Statement
for ( Expression ; Expression ; Expressionopt ) 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 *[no LineTerminator here]* appears in the right-hand side of a production of the syntactic grammar, it indicates that the production is a *restricted production*: it may not be used if a *LineTerminator* occurs in the input stream at the indicated position. For example, the production:

ReturnStatement :

```

return [no LineTerminator here] Expressionopt ;

```

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

Unless the presence of a *LineTerminator* is forbidden by a restricted production, any number of occurrences of *LineTerminator* may appear between any two consecutive tokens in the stream of input elements without affecting the syntactic acceptability of the program.

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

5.2 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 $\text{Result}(n)$ as shorthand for “the result of step n ”. We also use $\text{Type}(x)$ as shorthand for “the type of x ”.

Mathematical operations such as addition, subtraction, multiplication, and division should always be understood as computing exact mathematical results. Algorithms that model floating-point arithmetic include explicit steps to perform rounding, where necessary.

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

The mathematical function $\text{sign}(x)$ yields 1 if x is positive, 0 if x is zero, and -1 if x is negative.

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

The mathematical function $\text{floor}(x)$ yields the largest integer (closest to positive infinity) that is not larger than x . Note that $\text{floor}(x) = x - (x \text{ modulo } 1)$.

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.

6 SOURCE TEXT

ECMAScript source text is represented as a sequence of characters representable using the Unicode version 2.0 character encoding.

SourceCharacter ::
any Unicode character

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 contexts, any Unicode character may be expressed as a Unicode escape sequence consisting of six ASCII characters, namely `\u` plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment; within a string literal, the Unicode escape sequence contributes one character to the string value of the literal.

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

7 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.

7.1 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, and may occur within strings (where they are considered significant characters forming part of the literal string value), but cannot appear within any other kind of token.

The following characters are considered to be white space:

<i>Unicode Value</i>	<i>Name</i>	<i>Formal Name</i>
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>

Syntax

```
WhiteSpace ::  
  <TAB>  
  <VT>  
  <FF>  
  <SP>
```

7.2 LINE TERMINATORS

Line terminator characters, like whitespace characters, are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. Unlike whitespace characters, line terminators have some influence over the behavior of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token (not even a string. Line terminators also affect the process of automatic semicolon insertion (see section).

The following characters are considered to be line terminators:

<i>Unicode Value</i>	<i>Name</i>	<i>Formal Name</i>
\u000A	Line Feed	<LF>
\u000D	Carriage Return	<CR>

Syntax

```
LineTerminator ::  
  <LF>  
  <CR>
```

7.3 COMMENTS

Description

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

Because a single-line comment can contain any character except a *LineTerminator* character, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all characters from the *//* marker to the end of the line. However, the *LineTerminator* at the end of the line is not considered to be part of the single-line comment; it is recognized separately by the lexical grammar and becomes part of the stream of input elements for the syntactic grammar. This point is very important, because it implies that the presence or absence of single-line comments does not affect the process of automatic semicolon insertion (see section).

Syntax

Comment ::

MultiLineComment

SingleLineComment

MultiLineComment ::

/ MultiLineCommentChars_{opt} */*

MultiLineCommentChars ::

MultiLineNotAsteriskChar MultiLineCommentChars_{opt}

** PostAsteriskCommentChars_{opt}*

PostAsteriskCommentChars ::

MultiLineNotForwardSlashChar MultiLineCommentChars_{opt}

MultiLineNotAsteriskChar::

SourceCharacter **but not** *asterisk* *****

MultiLineNotForwardSlashChar::

SourceCharacter **but not** *forward-slash* */*

SingleLineComment ::

// SingleLineCommentChars_{opt}

SingleLineCommentChars ::

SingleLineCommentChar SingleLineCommentChars_{opt}

SingleLineCommentChar ::

SourceCharacter **but not** *LineTerminator*

7.4 TOKENS

Syntax

Token ::

ReservedWord

Identifier

Punctuator

Literal

7.4.1 Reserved Words

Description

Reserved words cannot be used as identifiers.

ReservedWord ::
 Keyword
 FutureReservedWord
 NullLiteral
 BooleanLiteral

7.4.2 Keywords

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

Syntax

Keyword: *one of*

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

7.4.3 Future Reserved Words

The following words are used as keywords in proposed extensions and are therefore reserved to allow for the possibility of future adoption of those extensions.

Syntax

FutureReservedWord : *one of*

case	do	method	try
catch	extends	super	
class	finally	switch	
default	import	throw	

7.5 IDENTIFIERS

Description

An identifier is a character sequence of unlimited length, where each character in the sequence must be a letter, a decimal digit, an underscore (`_`) character, or a dollar sign (`$`) character, and the first character may not be a decimal digit. ECMAScript identifiers are case sensitive: identifiers whose characters differ in any way, even if only in case, are considered to be distinct.

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
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

7.6 PUNCTUATORS

Syntax

Punctuator :: *one of*

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

7.7 LITERALS

Syntax

Literal ::
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral

7.7.1 Null Literals

Syntax

NullLiteral ::
null

Semantics

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

7.7.2 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**.

7.7.3 Numeric Literals

Syntax

NumericLiteral ::
DecimalLiteral
HexIntegerLiteral
OctalIntegerLiteral

DecimalLiteral ::
DecimalIntegerLiteral
DecimalIntegerLiteral . *DecimalDigits*_{opt} *ExponentPart*_{opt}
. *DecimalDigits* *ExponentPart*_{opt}
DecimalIntegerLiteral *ExponentPart*

DecimalIntegerLiteral ::
0
NonZeroDigit *DecimalDigits*_{opt}

DecimalDigits ::
DecimalDigit
DecimalDigits *DecimalDigit*

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

ExponentPart ::
ExponentIndicator *SignedInteger*

ExponentIndicator :: one of
e E

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

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

Semantics

A numeric literal stands for a value of the Number type. This value is determined in two steps: first, a mathematical value (MV) is derived from the literal; second, this mathematical value is rounded, ideally using IEEE 754 round-to-nearest mode, to a representable value of the number type.

- The MV of *NumericLiteral* :: *DecimalLiteral* is the MV of *DecimalLiteral*.
- The MV of *NumericLiteral* :: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *NumericLiteral* :: *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* . is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *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 *DecimalLiteral* :: *DecimalIntegerLiteral* . *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *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 *DecimalLiteral* :: . *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n} , where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* :: . *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^{-n} , where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalIntegerLiteral* :: **0** is 0.
- The MV of *DecimalIntegerLiteral* :: *NonZeroDigit* *DecimalDigits* is (the MV of *NonZeroDigit* times 10^n) plus the MV of *DecimalDigits*, where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalDigits* :: *DecimalDigit* is the MV of *DecimalDigit*.

- The MV of *DecimalDigits* :: *DecimalDigits* *DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- 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 MV of *DecimalDigits*.
- The MV of *SignedInteger* :: – *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* :: 0 or of *HexDigit* :: 0 or of *OctalDigit* :: 0 is 0.
- 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 is 8.
- The MV of *DecimalDigit* :: 9 or of *NonZeroDigit* :: 9 or of *HexDigit* :: 9 is 9.
- The MV of *HexDigit* :: a or of *HexDigit* :: A is 10.
- The MV of *HexDigit* :: b or of *HexDigit* :: B is 11.
- The MV of *HexDigit* :: c or of *HexDigit* :: C is 12.
- The MV of *HexDigit* :: d or of *HexDigit* :: D is 13.
- The MV of *HexDigit* :: e or of *HexDigit* :: E is 14.
- The MV of *HexDigit* :: f or of *HexDigit* :: F is 15.
- The MV of *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*.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be *the* number value for the MV (in the sense defined in section 8.4), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is *significant* if it is not part of an *ExponentPart* and (either it is not 0 or it is an *important zero* or there is no decimal point \dot{O} in the literal). A digit 0 is an *important zero* if there is at least one *important item* to its left and at least one *important item* to its right within the literal. Any digit that is not 0 and is not part of an *ExponentPart* is an *important item*; a decimal point \dot{O} is also an *important item*.

7.7.4 String Literals

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

Syntax

```
StringLiteral ::
    " DoubleStringCharactersopt "
    ' SingleStringCharactersopt '

DoubleStringCharacters ::
    DoubleStringCharacter DoubleStringCharactersopt

SingleStringCharacters ::
    SingleStringCharacter SingleStringCharactersopt
```

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

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

EscapeSequence ::
CharacterEscapeSequence
OctalEscapeSequence
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence ::
\ SingleEscapeCharacter
\ NonEscapeCharacter

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

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

EscapeCharacter ::
SingleEscapeCharacter
OctalDigit
x
u

HexEscapeSequence ::
\x HexDigit HexDigit

OctalEscapeSequence ::
\ OctalDigit
\ OctalDigit OctalDigit
\ ZeroToThree OctalDigit OctalDigit

ZeroToThree :: one of
0 1 2 3

UnicodeEscapeSequence ::
\u HexDigit HexDigit HexDigit

The definitions of the nonterminals *HexDigit* and *OctalDigit* are given in section 7.7.3.

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

- The SV of *StringLiteral* :: "" is the empty character sequence .
- The SV of *StringLiteral* :: ' ' is the empty character sequence.
- The SV of *StringLiteral* :: " *DoubleStringCharacters* " is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral* :: ' *SingleStringCharacters* ' is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* *DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.

- The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter SingleStringCharacters* is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The CV of *DoubleStringCharacter* :: *SourceCharacter* **but not double-quote " or backslash \ or LineTerminator** is the *SourceCharacter* character itself.
- The CV of *DoubleStringCharacter* :: *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *SingleStringCharacter* :: *SourceCharacter* **but not single-quote ' or backslash \ or LineTerminator** is the *SourceCharacter* character itself.
- The CV of *SingleStringCharacter* :: *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *EscapeSequence*:: *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence*:: *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *EscapeSequence*:: *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence*:: *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *CharacterEscapeSequence* :: *\ SingleEscapeCharacter* is the Unicode character whose Unicode value is determined by the *SingleEscapeCharacter* according to the following table:

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

- The CV of *CharacterEscapeSequence* :: *\ NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* :: *SourceCharacter* **but not EscapeCharacter or LineTerminator** is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* :: *\x HexDigit HexDigit* is the Unicode character whose code is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *OctalEscapeSequence* :: *\ OctalDigit* is the Unicode character whose code is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *\ OctalDigit OctalDigit* is the Unicode character whose code is (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *\ ZeroToThree OctalDigit OctalDigit* is the Unicode character whose code is (64 (that is, 8²) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree* :: **0** is 0.
- The MV of *ZeroToThree* :: **1** is 1.
- The MV of *ZeroToThree* :: **2** is 2.
- The MV of *ZeroToThree* :: **3** is 3.
- The CV of *UnicodeEscapeSequence* :: *\u HexDigit HexDigit HexDigit HexDigit* is the Unicode character whose code is (4096 (that is, 16³) times the MV of the first *HexDigit*) plus (256 (that is, 16²) times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

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

7.8 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 **}**.
- When, as the program is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript *Program*, then a semicolon is automatically inserted at the end of the input stream.

However, there is an additional overriding condition on the preceding rules: 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 is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation \hat{O} [no *LineTerminator* here] within the restricted production (and therefore such a token is called a restricted token), 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:

MemberExpression :

new *MemberExpression* [no *LineTerminator* here] *Arguments*

CallExpression :

MemberExpression [no *LineTerminator* here] *Arguments*

CallExpression [no *LineTerminator* here] *Arguments*

PostfixExpression :

LeftHandSideExpression [no *LineTerminator* here] **++**

LeftHandSideExpression [no *LineTerminator* here] **--**

ReturnStatement :

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

The practical effect of these restricted productions is as follows:

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

The resulting practical advice to ECMAScript programmers is:

1. The **(** that starts an argument list should be on the same line as the expression that indicates the function to be called.
2. A postfix **++** or **--** operator should appear on the same line as its operand.
3. An *Expression* in a **return** statement should start on the same line as the **return** token.

For example, the source

```
{ 1 2 } 3
```

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

```
{ 1
 2 } 3
```

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

```
{ 1
;2 ;} 3;
```

which is a valid ECMAScript sentence.

The source

```
for (a; b
)
```

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
```

is transformed by automatic semicolon insertion into the following:

```
return;
a + b;
```

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
```

is transformed by automatic semicolon insertion into the following:

```
a = b;
++c;
```

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
```

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.

8 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.

8.1 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.

8.2 THE NULL TYPE

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

8.3 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**.

8.4 THE STRING TYPE

The String type consists of the set of all finite ordered sequences of zero or more Unicode characters. Each character is regarded as occupying a position within the sequence. These positions are identified by nonnegative integers. The leftmost character (if any) is at position 0, the next character (if any) at position 1, and so on. The length of a string is the number of distinct positions within it. The empty string has length zero and therefore contains no characters.

8.5 THE NUMBER TYPE

The Number type has exactly $2^{64} - 2^{53} + 3$ (that is, 18437736874454810627) 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 "Not-a-Number" values of the IEEE Standard are represented in ECMAScript as a single special **NaN** value. (Note that the NaN value is produced by the program expression **NaN**, assuming that the globally defined variable **NaN** has not been altered by program execution.)

There are two other special values, called **positive Infinity** and **negative Infinity**. For brevity, these values are also referred to for expository purposes by the symbols $+\infty$ and $-\infty$, respectively. (Note that these two infinite number values are produced by the program expressions **+Infinity** (or simply **Infinity**) and **-Infinity**, assuming that the globally defined variable **Infinity** has not been altered by program execution.)

The other $2^{64} - 2^{53}$ (that is, 18437736874454810624) 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**. For brevity, these values are also referred to for expository purposes by the symbols **+0** and **-0**, respectively. (Note that these two zero number values are produced by the program expressions **+0** (or simply **0**) and **-0**.)

The $2^{64} - 2^{53} - 2$ (that is, 18428729675200069632) finite nonzero values are of two kinds:

$2^{64} - 2^{54}$ 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 ranging from -1074 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^{52} , and e is -1074 .

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

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

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

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 convert each such value to one of 2^{32} integer values. See the descriptions of the `ToInt32` and `ToUint32` operators in sections 9.5 and 9.6, respectively.

8.6 THE OBJECT TYPE

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

8.6.1 Property Attributes

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

Attribute	Description
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 enumerated by a for-in enumeration (section 12.5.3)
DontDelete	Attempts to delete the property will be ignored. See the description of the delete operator in section 11.4.1.
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.

8.6.2 Internal Properties and Methods

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

Property	Parameters	Description
<code>[[Prototype]]</code>	none	The prototype of this object.
<code>[[Class]]</code>	none	The kind of this object.
<code>[[Value]]</code>	none	Internal state information associated with this object.
<code>[[Get]]</code>	(PropertyName)	Returns the value of the property.
<code>[[Put]]</code>	(PropertyName, Value)	Sets the specified property to Value.
<code>[[CanPut]]</code>	(PropertyName, Value)	Returns a boolean value indicating whether a <code>[[Put]]</code> operation with the same arguments will succeed.
<code>[[HasProperty]]</code>	(PropertyName)	Returns a boolean value indicating whether the object already has a member with the given name.
<code>[[DefaultValue]]</code>	(Hint)	Returns the default value of the object, which should be a primitive value (not an object or reference).
<code>[[Construct]]</code>	optional user-provided parameters	(Constructor) Constructs an object. Invoked via the new operator.
<code>[[Call]]</code>	optional user-provided parameters	(Function) Executes code associated with the object. Invoked via a function call expression.

Every ECMAScript object must implement the `[[Class]]` property and the `[[Get]]`, `[[Put]]`, `[[HasProperty]]`, and `[[DefaultValue]]` methods, even host objects.

The value of the `[[Prototype]]` property must be either an object or **null**, and every `[[Prototype]]` chain must have finite length (that is, starting from any object, recursively accessing the `[[Prototype]]` property must eventually lead to a **null** value).

The value of the `[[Class]]` property is defined by this specification for every kind of built-in object. The value of the `[[Class]]` property of a host object may be any value, even a value used by a built-in object for its `[[Class]]` property. Note that this specification does not provide any means for a program to access the value of a `[[Class]]` property; it is used internally to distinguish different kinds of built-in objects.

Every built-in object implements the `[[Get]]`, `[[Put]]`, `[[CanPut]]`, and `[[HasProperty]]` methods in the manner described in sections 8.6.2.1, 8.6.2.2, and 8.6.2.3, respectively, except that Array objects have a slightly different implementation of the `[[Put]]` method (section). Host objects may implement these methods in any manner; for example, one possibility is that `[[Get]]` and `[[Put]]` for a particular host object indeed fetch and store property values but `[[HasProperty]]` always generates **false**.

In the following algorithm descriptions, assume *O* is a native ECMAScript object and *P* is a string.

8.6.2.1 `[[Get]](P)`

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).

8.6.2.2 `[[Put]](P, V)`

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

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

Note, however, that if *O* is an Array object, it has a more elaborate `[[Put]]` method (section 15.4.4.1).

8.6.2.3 `[[CanPut]](P)`

The `[[CanPut]]` method is used only by the `[[Put]]` method.

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

14. If *O* doesn't have a property with name *P*, go to step 4.
15. If the property has the `ErrorOnWrite` attribute, generate a runtime error.
16. If the property has the `ReadOnly` attribute, return **false**.
17. If the `[[Prototype]]` of *O* is not implemented or its value is not an object, return **true**.
18. Call the `[[CanPut]]` method of `[[Prototype]]` of *O* with property Name *P*.
19. Return `Result(5)`.

8.6.2.4 `[[HasProperty]](P)`

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

20. If *O* has a property with name *P*, return **true**.
21. If the `[[Prototype]]` of *O* is not implemented or its value is not an object, return **false**.
22. Call the `[[HasProperty]]` method of `[[Prototype]]` with property name *P*.
23. Return `Result(3)`.

8.7 THE REFERENCE TYPE

The internal Reference type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon references in the manner described here. However, a value of type **Reference** is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The Reference type is used to explain the behavior of assignment operators: the left-hand operand of an assignment is expected to produce a reference. This behavior could, instead, be explained entirely in terms of a case analysis on the syntactic form of the left-hand operand of an assignment operator, but for one difficulty: function calls are permitted to return references. This possibility is admitted purely for the sake of host objects. No built-in ECMAScript function defined by this specification returns a reference and there is no provision for a user-defined function to return a reference.

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

The following abstract operations are used in this specification to describe the behavior of references:

- `GetBase(V)`. Returns the base object component of the reference *V*.
- `GetPropertyname(V)`. Returns the property-name component of the reference *V*.
- `GetValue(V)`. Returns the value of the property indicated by the reference *V*.
- `PutValue(V, W)`. Changes the value of the property indicated by the reference *V* to be *W*.

8.7.1 `GetBase(V)`

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

8.7.2 `GetPropertyname(V)`

26. If `Type(V)` is `Reference`, return the property-name component of *V*.
27. Generate a runtime error.

8.7.3 GetValue(V)

28. If Type(V) is not Reference, return V.
29. Call GetBase(V).
30. If Result(2) is **null**, generate a runtime error.
31. Call the [[Get]] method of Result(2), passing GetPropertyName(V) for the property-name.
32. Return Result(4).

8.7.4 PutValue(V, W)

33. If Type(V) is not Reference, generate a runtime error.
34. Call GetBase(V).
35. If Result(2) is **null**, go to step 6.
36. Call the [[Put]] method of Result(2), passing GetPropertyName(V) for the property name and W for the value.
37. Return.
38. Call the [[Put]] method for the global object, passing GetPropertyName(V) for the property name and W for the value.
39. Return.

8.8 THE LIST TYPE

The internal List type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon List values in the manner described here. However, a value of the List type is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The List type is used to explain the evaluation of argument lists (section 11.2.4) in **new** expressions and in function calls. Values of the List type are simply ordered sequences of values. These sequences may be of any length.

8.9 THE COMPLETION TYPE

The internal Completion type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon Completion values in the manner described here. However, a value of the Completion type is used only as an intermediate result of statement evaluation and cannot be stored as the value of a variable or property.

The Completion type is used to explain the behavior of statements (**break**, **continue**, and **return**) that perform nonlocal transfers of control. Values of the Completion type have one of the following forms:

- \hat{O} normal completion \hat{O}
- \hat{O} normal completion after value V \hat{O}
- \hat{O} abrupt completion because of **break** \hat{O}
- \hat{O} abrupt completion after value V because of **break** \hat{O}
- \hat{O} abrupt completion because of **continue** \hat{O}
- \hat{O} abrupt completion after value V because of **continue** \hat{O}
- \hat{O} abrupt completion because of **return** V \hat{O} where V is a value

Any completion of one of the four forms that carries a value V is called a *value completion*. Any completion of one of the first two forms is called a *normal completion*; any other completion is called an *abrupt completion*. Any completion of a form that mentions **break** is called a **break completion**. Any completion of a form that mentions **continue** is called a **continue completion**. Any completion of a form that mentions **return** is called a **return completion**.

9 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.

9.1 TO PRIMITIVE

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	The result equals the input argument (no conversion).
Null	The result equals the input argument (no conversion).
Boolean	The result equals the input argument (no conversion).
Number	The result equals the input argument (no conversion).
String	The result equals the input argument (no conversion).
Object	Return the default value of the Object. The default value of an object is retrieved by calling the internal <code>[[DefaultValue]]</code> method of the object, passing the optional hint <i>PreferredType</i> . The behavior of the <code>[[DefaultValue]]</code> 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.

9.2 TO BOOLEAN

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	The result equals the input argument (no conversion).
Number	The result is false if the argument is +0 , -0 , or NaN ; otherwise the result is true .
String	The result is false if the argument is the empty string (its length is zero); otherwise the result is true .
Object	true

9.3 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	The result is 1 if the argument is true . The result is +0 if the argument is false .
Number	The result equals the input argument (no conversion).
String	See grammar and discussion below.
Object	Apply the following steps: 40. Call ToPrimitive(input argument, hint Number). 41. Call ToNumber(Result(1)). 42. Return Result(2).

9.3.1 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 as an expansion of *StringNumericLiteral*, 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 :::

StrDecimalLiteral

+ *StrDecimalLiteral*

- *StrDecimalLiteral*

HexIntegerLiteral

StrDecimalLiteral :::

Infinity

DecimalDigits_{opt}

DecimalDigits . *DecimalDigits_{opt} ExponentPart_{opt}*

. *DecimalDigits ExponentPart_{opt}*

DecimalDigits ExponentPart

DecimalDigits :::

DecimalDigit

DecimalDigits DecimalDigit

DecimalDigit ::: one of

0 1 2 3 4 5 6 7 8 9

ExponentPart :::

ExponentIndicator SignedInteger

ExponentIndicator ::: one of
e E

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

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

Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (section 7.7.3):

- A *StringNumericLiteral* may be preceded and/or followed by whitespace and/or line terminators.
- A *StringNumericLiteral* may not use octal notation.
- A *StringNumericLiteral* that is decimal may have any number of leading **0** digits.
- A *StringNumericLiteral* that is decimal may be preceded by **+** or **-** to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only whitespace and/or one occurrence of **+** is converted to **+0**.
- A *StringNumericLiteral* that contains only whitespace and/or one occurrence of **-** is converted to **-0**.

The conversion of a string to a number value is similar overall to the determination of the number value for a numeric literal (section 7.7.3), but some of the details are different, so the process for converting a string numeric literal to a value of *Number* type is given here in full. This value is determined in two steps: first, a mathematical value (MV) is derived from the string numeric literal; second, this mathematical value is rounded, ideally using IEEE 754 round-to-nearest mode, to a representable value of the number type.

- The MV of *StringNumericLiteral* ::: *StrWhiteSpace_{opt}* *StrNumericLiteral* *StrWhiteSpace_{opt}* is the MV of *StrNumericLiteral*, no matter whether whitespace is present or not.
- The MV of *StrNumericLiteral* ::: *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* ::: **+** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* ::: **-** *StrDecimalLiteral* is the negative of the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* ::: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* ::: **Infinity** is 10^{100} (a value so large that it will round to $+\infty$).
- The MV of *StrDecimalLiteral* ::: (an empty character sequence) is 0.
- The MV of *StrDecimalLiteral* ::: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* ::: *DecimalDigits* . is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n}), where n is the number of characters in the second *DecimalDigits*.
- The MV of *StrDecimalLiteral* ::: *DecimalDigits* . *ExponentPart* is the MV of *DecimalDigits* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* *ExponentPart* is (the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n})) times 10^e , where n is the number of characters in the second *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* ::: . *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n} , where n is the number of characters in *DecimalDigits*.
- The MV of *StrDecimalLiteral* ::: . *DecimalDigits* *ExponentPart* 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 *StrDecimalLiteral* ::: *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalDigits* ::: *DecimalDigit* is the MV of *DecimalDigit*.

- The MV of *DecimalDigits* ::: *DecimalDigits* *DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- 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 MV of *DecimalDigits*.
- The MV of *SignedInteger* ::: – *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* :: 0 or of *HexDigit* :: 0 is 0.
- The MV of *DecimalDigit* :: 1 or of *HexDigit* :: 1 is 1.
- The MV of *DecimalDigit* :: 2 or of *HexDigit* :: 2 is 2.
- The MV of *DecimalDigit* :: 3 or of *HexDigit* :: 3 is 3.
- The MV of *DecimalDigit* :: 4 or of *HexDigit* :: 4 is 4.
- The MV of *DecimalDigit* :: 5 or of *HexDigit* :: 5 is 5.
- The MV of *DecimalDigit* :: 6 or of *HexDigit* :: 6 is 6.
- The MV of *DecimalDigit* :: 7 or of *HexDigit* :: 7 is 7.
- The MV of *DecimalDigit* :: 8 or of *HexDigit* :: 8 is 8.
- The MV of *DecimalDigit* :: 9 or of *HexDigit* :: 9 is 9.
- The MV of *HexDigit* :: a or of *HexDigit* :: A is 10.
- The MV of *HexDigit* :: b or of *HexDigit* :: B is 11.
- The MV of *HexDigit* :: c or of *HexDigit* :: C is 12.
- The MV of *HexDigit* :: d or of *HexDigit* :: D is 13.
- The MV of *HexDigit* :: e or of *HexDigit* :: E is 14.
- The MV of *HexDigit* :: f or of *HexDigit* :: F is 15.
- The MV of *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*.

Once the exact MV for a string numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non-whitespace character in the string numeric literal is $\hat{O}-\tilde{O}$, in which case the rounded value is -0. Otherwise, the rounded value must be *the* number value for the MV (in the sense defined in section 8.4), unless the literal includes a *StrDecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is *significant* if it is not part of an *ExponentPart* and (either it is not 0 or it is an *important zero* or there is no decimal point $\hat{O}.\tilde{O}$ in the literal). A digit 0 is an *important zero* if there is at least one important item to its left and at least one *important item* to its right within the literal. Any digit that is not 0 and is not part of an *ExponentPart* is an important item; a decimal point $\hat{O}.\tilde{O}$ is also an important item.

9.4 ToInteger

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

43. Call ToNumber on the input argument.
44. If Result(1) is NaN, return +0.
45. If Result(1) is +∞ or -∞, return Result(1).
46. Compute sign(Result(1)) * floor(abs(Result(1))).
47. Return Result(4).

9.5 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:

48. Call ToNumber on the input argument.
49. If Result(1) is NaN, +∞, or -∞, return +0.
50. Compute sign(Result(1)) * floor(abs(Result(1))).
51. Compute Result(3) modulo 2^{32} ; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{32} .

52. If Result(4) is greater than or equal to 2^{31} , return Result(5)− 2^{32} ; otherwise return Result(5).

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 $+\infty$ and $-\infty$ are mapped to +0.)

Note that ToInt32 maps −0 to +0.

9.6ToUINT32: (UNSIGNED 32 BIT INTEGER)

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:

53. Call ToNumber on the input argument.

54. If Result(1) is NaN, $+\infty$, or $-\infty$, return +0.

55. Compute sign(Result(1)) * floor(abs(Result(1))).

56. Compute Result(3) modulo 2^{32} ; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{32} .

57. Return Result(4).

Discussion:

Note that 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 $+\infty$ and $-\infty$ are mapped to +0.)

Note that ToUint32 maps −0 to +0.

9.7ToUINT16: (UNSIGNED 16 BIT INTEGER)

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

58. Call ToNumber on the input argument.

59. If Result(1) is NaN, $+\infty$, or $-\infty$, return +0.

60. Compute sign(Result(1)) * floor(abs(Result(1))).

61. Compute Result(3) modulo 2^{16} ; that is, a finite integer value k of Number type with positive sign and less than 2^{16} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{16} .

62. Return Result(4).

Discussion:

Note that the substitution of 2^{16} for 2^{32} in step 4 is the only difference between ToUint32 and ToUnit16.

Note that ToUint16 maps −0 to +0.

9.8ToSTRING

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: 63. Call ToPrimitive(input argument, hint String). 64. Call ToString(Result(1)). 65. Return Result(2).

9.8.1 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".
- If the argument is +0 or -0, the result is "0".
- If the argument is +∞, the result is "Infinity".
- If the argument is -∞, the result is "-Infinity".
- Otherwise, the result is a string that represents the sign and finite nonzero magnitude (absolute value) of the argument. If the sign is negative, the first character of the result is $\hat{O}-\tilde{O}$; if the sign is positive, no sign character appears in the result. As for the magnitude m :
 - If m is an integer less than 10^{21} , 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^{-6} but less than 10^{21} , 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 $\hat{O}.\tilde{O}$, followed by one or more decimal digits (see below) representing the fractional part of m .
 - If m is less than 10^{-6} or not less than 10^{21} , 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 $\hat{O}.\tilde{O}$, followed by one or more decimal digits (see below) representing the fractional part of a , followed by the lowercase letter $\hat{O}e\tilde{O}$, followed by a representation of n as a decimal integer (first a minus sign $\hat{O}-\tilde{O}$ if n is negative or a plus sign $\hat{O}+\tilde{O}$ if 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 the Number 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.

There remains some choice as to the last digit generated for a fractional part. The following specification was considered but not adopted:

(This paragraph is not part of the ECMAScript specification.) The decimal string produced must be as close in its mathematical value to the mathematical value of the original number as any other decimal string with the same number of digits; and if two decimal strings of the same minimal length would be equally close in value to the original number, then the decimal string whose last digit is even should be chosen.

While such a strategy is recommended to implementors, the actual rule is somewhat more permissive:

- If x is any number value, then ToNumber(ToString(x)) must be exactly the same as x .

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].

9.9 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 new Boolean object whose default value is the value of the boolean. See section 15.6 for a description of Boolean objects.
Number	Create a new Number object whose default value is the value of the number. See section 15.7 for a description of Number objects.
String	Create a new String object whose default value is the value of the string. See section 15.5 for a description of String objects.
Object	The result is the input argument (no conversion).

10 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.

10.1 DEFINITIONS

10.1.1 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.

10.1.2 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*.

10.1.3 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.

10.1.4 Scope Chain and Identifier Resolution

Every execution context has associated with it 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:

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

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

10.1.5 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.

10.1.6 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.

10.1.7 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.

10.1.8 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.

3.1 Issue: Should the arguments object have a caller property?

10.2 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.

10.2.1 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.

10.2.2 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.

10.2.3 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.

10.2.4 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 run time 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 { DontDelete }.
- The **this** value is determined just as for function and anonymous code.

11 EXPRESSIONS

11.1 PRIMARY EXPRESSIONS

Syntax

PrimaryExpression :
 this
 Identifier
 Literal
 (*Expression*)

11.1.1 The `this` Keyword

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

11.1.2 Identifier Reference

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

11.1.3 Literal Reference

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

11.1.4 The Grouping Operator

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

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

11.2 LEFT-HAND-SIDE EXPRESSIONS

Syntax

MemberExpression :
 PrimaryExpression
 MemberExpression [*Expression*]
 MemberExpression . *Identifier*
 new *MemberExpression* [no *LineTerminator* here] *Arguments*

NewExpression :
 MemberExpression
 new *NewExpression*

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

Arguments :
()
(*ArgumentList*)

ArgumentList :
AssignmentExpression
ArgumentList , *AssignmentExpression*

LeftHandSideExpression :
NewExpression
CallExpression

11.2.1 Property Accessors

Properties are accessed by name, using either the dot notation:

MemberExpression . *Identifier*
CallExpression . *Identifier*

or the bracket notation:

MemberExpression [*Expression*]
CallExpression [*Expression*]

The dot notation is explained by the following syntactic conversion:

MemberExpression . *Identifier*

is identical in its behavior to

MemberExpression [<identifier-string>]

and similarly

CallExpression . *Identifier*

is identical in its behavior to

CallExpression [<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:

73. Evaluate *MemberExpression*.
74. Call GetValue(Result(1)).
75. Evaluate *Expression*.
76. Call GetValue(Result(3)).
77. Call ToObject(Result(2)).
78. Call ToString(Result(4)).
79. Return a value of type Reference whose base object is Result(5) and whose property name is Result(6).

The production *CallExpression* : *MemberExpression* [*Expression*] is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

11.2.2 The new Operator

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

80. Evaluate *NewExpression*.
81. Call GetValue(Result(1)).
82. If Type(Result(2)) is not Object, generate a runtime error.
83. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
84. Call the [[Construct]] method on Result(2), providing no arguments (that is, an empty list of arguments).
85. If Type(Result(5)) is not Object, generate a runtime error.
86. Return Result(5).

The production *NewCallExpression* : **new** *NewExpression* *Arguments* is evaluated as follows:

87. Evaluate *NewExpression*.
88. Call `GetValue(Result(1))`.
89. Evaluate *Arguments*, producing an internal list of argument values (section 11.2.4).
90. If `Type(Result(2))` is not `Object`, generate a runtime error.
91. If `Result(2)` does not implement the internal `[[Construct]]` method, generate a runtime error.
92. Call the `[[Construct]]` method on `Result(2)`, providing the list `Result(3)` as the argument values.
93. If `Type(Result(6))` is not `Object`, generate a runtime error.
94. Return `Result(6)`.

11.2.3 Function Calls

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

95. Evaluate *MemberExpression*.
96. Evaluate *Arguments*, producing an internal list of argument values (section 11.2.4).
97. Call `GetValue(Result(1))`.
98. If `Type(Result(3))` is not `Object`, generate a runtime error.
99. If `Result(3)` does not implement the internal `[[Call]]` method, generate a runtime error.
100. If `Type(Result(1))` is `Reference`, `Result(6)` is `GetBase(Result(1))`. Otherwise, `Result(6)` is **null**.
101. If `Result(6)` is an activation object, `Result(7)` is **null**. Otherwise, `Result(7)` is the same as `Result(6)`.
102. Call the `[[Call]]` method on `Result(3)`, providing `Result(7)` as the **this** value and providing the list `Result(2)` as the argument values.
103. Return `Result(8)`.

The production *CallExpression : CallExpression Arguments* is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

Note: `Result(8)` will never be of type `Reference` if `Result(3)` is a native ECMAScript object. Whether calling a host object can return a value of type `Reference` is implementation-dependent.

11.2.4 Argument Lists

The evaluation of an argument list produces an internal list of values (section 8.8).

The production *Arguments : ()* is evaluated as follows:

104. Return an empty internal list of values.

The production *Arguments : (ArgumentList)* is evaluated as follows:

105. Evaluate *ArgumentList*.
106. Return `Result(1)`.

The production *ArgumentList : AssignmentExpression* is evaluated as follows:

107. Evaluate *AssignmentExpression*.
108. Call `GetValue(Result(1))`.
109. Return an internal list whose sole item is `Result(2)`.

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

110. Evaluate *ArgumentList*.
111. Evaluate *AssignmentExpression*.
112. Call `GetValue(Result(2))`.
113. Return an internal list whose length is one greater than the length of `Result(1)` and whose items are the items of `Result(1)`, in order, followed at the end by `Result(3)`, which is the last item of the new list.

11.3 POSTFIX EXPRESSIONS

Syntax

PostfixExpression :

LeftHandSideExpression

LeftHandSideExpression [no *LineTerminator* here] **++**

LeftHandSideExpression [no *LineTerminator* here] **--**

11.3.1 Postfix Increment Operator

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

114. Evaluate *MemberExpression*.

115. Call GetValue(Result(1)).

116. Call ToNumber(Result(2)).

117. Add the value 1 to Result(3), using the same rules as for the + operator (section 11.6.3).

118. Call PutValue(Result(1), Result(4)).

119. Return Result(3).

11.3.2 Postfix Decrement Operator

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

120. Evaluate *MemberExpression*.

121. Call GetValue(Result(1)).

122. Call ToNumber(Result(2)).

123. Subtract the value 1 from Result(3), using the same rules as for the - operator (section 11.6.3).

124. Call PutValue(Result(1), Result(4)).

125. Return Result(3).

11.4 UNARY OPERATORS

Syntax

UnaryExpression :

PostfixExpression

delete *UnaryExpression*

void *UnaryExpression*

typeof *UnaryExpression*

++ *UnaryExpression*

-- *UnaryExpression*

+ *UnaryExpression*

- *UnaryExpression*

~ *UnaryExpression*

! *UnaryExpression*

11.4.1 The delete Operator

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

126. Evaluate *UnaryExpression*.

127. Call GetBase(Result(1)).

128. Call GetPropertyName(Result(1)).

129. If Type(Result(2)) is not Object, return **true**.

130. If Result(2) does not implement the internal [[Delete]] method, go to step 8.

131. Call the [[Delete]] method on Result(2), providing Result(3) as the property name to delete.

132. Return Result(6).

133. Call the [[HasProperty]] method on Result(2)), providing Result(3) as the property name to check for.

134. If Result(8) is **true**, return **false**.

135. Return **true**.

11.4.2 The void Operator

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

136. Evaluate *UnaryExpression*.

137. Call GetValue(Result(1)).

138. Return undefined.

11.4.3 The type of Operator

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

139. Evaluate *UnaryExpression*.

140. If Type(Result(1)) is Reference and GetBase(Result(1)) is **null**, return "**undefined**".

141. Call GetValue(Result(1)).

142. 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 (host)	Implementation-dependent

11.4.4 Prefix Increment Operator

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

143. Evaluate *UnaryExpression*.

144. Call GetValue(Result(1)).

145. Call ToNumber(Result(2)).

146. Add the value **1** to Result(3), using the same rules as for the **+** operator (section 11.6.3).

147. Call PutValue(Result(1), Result(4)).

148. Return Result(4).

11.4.5 Prefix Decrement Operator

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

149. Evaluate *UnaryExpression*.

150. Call GetValue(Result(1)).

151. Call ToNumber(Result(2)).

152. Subtract the value **1** from Result(3), using the same rules as for the **-** operator (section 11.6.3).

153. Call PutValue(Result(1), Result(4)).

154. Return Result(4).

11.4.6 Unary + Operator

The unary **+** operator converts its operand to Number type.

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

155. Evaluate *UnaryExpression*.

156. Call GetValue(Result(1)).

157. Call ToNumber(Result(2)).

158. Return Result(3).

11.4.7 Unary - Operator

The unary **-** operator converts its operand to Number type and then negates it. Note that negating **+0** produces **-0**, and negating **-0** produces **+0**.

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

159. Evaluate *UnaryExpression*.

160. Call GetValue(Result(1)).

161. Call ToNumber(Result(2)).
162. If Result(3) is NaN, return NaN.
163. Negate Result(3); that is, compute a number with the same magnitude but opposite sign.
164. Return Result(5).

11.4.8 The Bitwise NOT Operator (~)

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

165. Evaluate *UnaryExpression*.
166. Call GetValue(Result(1)).
167. Call ToInt32(Result(2)).
168. Apply bitwise complement to Result(3). The result is a signed 32-bit integer.
169. Return Result(4).

11.4.9 Logical NOT Operator (!)

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

170. Evaluate *UnaryExpression*.
171. Call GetValue(Result(1)).
172. Call ToBoolean(Result(2)).
173. If Result(3) is true, return false.
174. Return true.

11.5 MULTIPLICATIVE OPERATORS

Syntax

MultiplicativeExpression :
 UnaryExpression
 MultiplicativeExpression * *UnaryExpression*
 MultiplicativeExpression / *UnaryExpression*
 MultiplicativeExpression % *UnaryExpression*

Semantics

The production *MultiplicativeExpression* : *MultiplicativeExpression* @ *UnaryExpression*, where @ stands for one of the operators in the above definitions, is evaluated as follows:

175. Evaluate *MultiplicativeExpression*.
176. Call GetValue(Result(1)).
177. Evaluate *UnaryExpression*.
178. Call GetValue(Result(3)).
179. Call ToNumber(Result(2)).
180. Call ToNumber(Result(4)).
181. Apply the specified operation (*, /, or %) to Result(5) and Result(6). See the discussions below (7.4.1, 7.4.2, 7.4.3).
182. Return Result(7).

11.5.1 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 an infinity results in an infinity. The sign is determined by the rule already stated above.

- 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.

11.5.2 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 zero results in an infinity. The sign is determined by the rule already stated above.
- 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.

11.5.3 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 the same as the dividend.
- 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 .

11.6 ADDITIVE OPERATORS

Syntax

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

11.6.1 The Addition Operator (+)

The addition operator either performs string concatenation or numeric addition.

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

183. Evaluate *AdditiveExpression*.
184. Call `GetValue(Result(1))`.
185. Evaluate *MultiplicativeExpression*.
186. Call `GetValue(Result(3))`.
187. Call `ToPrimitive(Result(2))`.
188. Call `ToPrimitive(Result(4))`.
189. If `Type(Result(5))` is `String` or `Type(Result(6))` is `String`, go to step 13. (Note that this step differs from step 3 in the algorithm for comparison for the relational operators in using *or* instead of *and*.)
190. Call `ToNumber(Result(5))`.
191. Call `ToNumber(Result(6))`.
192. Apply the addition operation to `Result(8)` and `Result(9)`. See the discussion below.
193. Return `Result(10)`.
194. Call `ToString(Result(5))`.
195. Call `ToString(Result(6))`.
196. Concatenate `Result(12)` followed by `Result(13)`.
197. Return `Result(14)`.

Note that no hint is provided in the calls to `ToPrimitive` in steps 5 and 6. All native ECMAScript objects handle the absence of a hint as if the hint `Number` were given, but host objects may handle the absence of a hint in some other manner.

11.6.2 The Subtraction Operator (-)

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

198. Evaluate *AdditiveExpression*.
199. Call `GetValue(Result(1))`.
200. Evaluate *MultiplicativeExpression*.
201. Call `GetValue(Result(3))`.
202. Call `ToNumber(Result(2))`.
203. Call `ToNumber(Result(4))`.
204. Apply the subtraction operation to `Result(5)` and `Result(6)`. See the discussion below (7.5.3).
205. Return `Result(7)`.

11.6.3 Applying the Additive Operators (+ , -) to Numbers

The `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The `-` operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 double-precision arithmetic:

- If either operand is `NaN`, the result is `NaN`.
- The sum of two infinities of opposite sign is `NaN`.
- The sum of two infinities of the same sign is the infinity of that sign.

- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two negative zeros is **-0**. The sum of two positive zeros, or of two zeros of opposite sign, is **+0**.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is **+0**.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 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)*.

11.7 BITWISE SHIFT OPERATORS

Syntax

```

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

```

Semantics

The result of evaluating *ShiftExpression* is always truncated to 32 bits. If the result of evaluating *ShiftExpression* produces a fractional component, the fractional component is discarded. The result of evaluating an *AdditiveExpression* that is the right-hand operand of a shift operator is always truncated to five bits.

11.7.1 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:

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

11.7.2 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:

215. Evaluate *ShiftExpression*.
216. Call GetValue(Result(1)).
217. Evaluate *AdditiveExpression*.
218. Call GetValue(Result(3)).
219. Call ToInt32(Result(2)).
220. Call ToInt32(Result(4)).
221. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.

222. 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.
223. Return Result(8).

11.7.3 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:

224. Evaluate *ShiftExpression*.
225. Call GetValue(Result(1)).
226. Evaluate *AdditiveExpression*.
227. Call GetValue(Result(3)).
228. Call ToUInt32(Result(2)).
229. Call ToInt32(Result(4)).
230. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
231. 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.
232. Return Result(8).

11.8 RELATIONAL OPERATORS

Syntax

RelationalExpression :

ShiftExpression

RelationalExpression < *ShiftExpression*

RelationalExpression > *ShiftExpression*

RelationalExpression <= *ShiftExpression*

RelationalExpression >= *ShiftExpression*

Semantics

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

233. Evaluate *RelationalExpression*.
234. Call GetValue(Result(1)).
235. Evaluate *ShiftExpression*.
236. Call GetValue(Result(3)).
237. Perform the comparison Result(2) < Result(4). (See below.)
238. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

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

239. Evaluate *RelationalExpression*.
240. Call GetValue(Result(1)).
241. Evaluate *ShiftExpression*.
242. Call GetValue(Result(3)).
243. Perform the comparison Result(4) < Result(2). (See below.)
244. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

The production *RelationalExpression*: *RelationalExpression* <= *ShiftExpression* is evaluated as follows:

245. Evaluate *RelationalExpression*.
246. Call GetValue(Result(1)).
247. Evaluate *ShiftExpression*.
248. Call GetValue(Result(3)).
249. Perform the comparison Result(4) < Result(2). (See below.)
250. If Result(5) is **true** or **undefined**, return **false**. Otherwise, return **true**.

The production *RelationalExpression*: *RelationalExpression* >= *ShiftExpression* is evaluated as follows:

251. Evaluate *RelationalExpression*.
252. Call GetValue(Result(1)).
253. Evaluate *ShiftExpression*.
254. Call GetValue(Result(3)).
255. Perform the comparison Result(2) < Result(4). (See below.)
256. If Result(5) is **true** or **undefined**, return **false**. Otherwise, return **true**.
257. The comparison $x < y$, where x and y are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). Such a comparison is performed as follows: Call ToPrimitive(x , hint Number).
258. Call ToPrimitive(y , hint Number).
259. If Type(Result(1)) is String and Type(Result(2)) is String, go to step 16. (Note that this step differs from step 7 in the algorithm for the addition operator **+** in using *and* instead of *or*.)
260. Call ToNumber(Result(1)).
261. Call ToNumber(Result(2)).
262. If Result(4) is **NaN**, return **undefined**.
263. If Result(5) is **NaN**, return **undefined**.
264. If Result(4) and Result(5) are the same number value, return **false**.
265. If Result(4) is **+0** and Result(5) is **-0**, return **false**.
266. If Result(4) is **-0** and Result(5) is **+0**, return **false**.
267. If Result(4) is **+∞**, return **false**.
268. If Result(4) is **-∞**, return **true**.
269. If Result(5) is **+∞**, return **true**.
270. If Result(5) is **-∞**, return **false**.
271. If the (finite, nonzero) mathematical value of Result(4) is less than the (finite, nonzero) mathematical value of Result(5), return **true**. Otherwise, return **false**.
272. If Result(2) is a prefix of Result (1), return **false**. (A string value p is a prefix of string value q if q can be the result of concatenating p and some other string r . Note that any string is a prefix of itself, because r may be the empty string.)
273. If Result(1) is a prefix of Result (2), return **true**.
274. Let k be the smallest nonnegative integer such that the character at position k within Result(1) is different from the character at position k within Result(2). (There must be such a k , for neither string is a prefix of the other.)
 Let m be the integer that is the Unicode encoding for the character at position k within Result(1).
 Let n be the integer that is the Unicode encoding for the character at position k within Result(2).
 If $m < n$, return **true**. Otherwise, return **false**.

11.9 EQUALITY OPERATORS

Syntax

EqualityExpression :

RelationalExpression

EqualityExpression **==** *RelationalExpression*

EqualityExpression **!=** *RelationalExpression*

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

275. Evaluate *EqualityExpression*.
276. Call GetValue(Result(1)).
277. Evaluate *RelationalExpression*.
278. Call GetValue(Result(3)).
279. Perform the comparison Result(4) == Result(2). (See below.)
280. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

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

281. Evaluate *EqualityExpression*.
282. Call GetValue(Result(1)).
283. Evaluate *RelationalExpression*.
284. Call GetValue(Result(3)).
285. Perform the comparison Result(4) == Result(2). (See below.)
286. If Result(5) is **true** or **undefined**, return **false**. Otherwise, return **true**.

The comparison $x == y$, where x and y are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is NaN). Such a comparison is performed as follows:

287. If Type(x) is different from Type(y), go to step 14.
288. If Type(x) is Undefined, return **true**.
289. If Type(x) is Null, return **true**.
290. If Type(x) is not Number, go to step 11.
291. If x is NaN, return **undefined**.
292. If y is NaN, return **undefined**.
293. If x is the same number value as y , return **true**.
294. If x is +0 and y is -0, return **true**.
295. If x is -0 and y is +0, return **true**.
296. Return **false**.
297. If Type(x) is String, then return **true** if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
298. If Type(x) is Boolean, return **true** if x and y are both **true** or both **false**. Otherwise, return **false**.
299. Return **true** if x and y refer to the same object. Otherwise, return **false**.
300. If x is **null** and y is **undefined**, return **true**.
301. If x is **undefined** and y is **null**, return **true**.
302. If Type(x) is Number and Type(y) is String, return the result of the comparison ToString(x) == y .
303. If Type(x) is String and Type(y) is Number, return the result of the comparison x == ToString(y).
304. Return **false**.

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 $B == C$, $\Rightarrow A == C$, assuming no side effects.

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

11.10 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:

305. Evaluate A.

306. Call GetValue(Result(1)).

307. Evaluate B.

308. Call GetValue(Result(3)).

309. Call ToInt32(Result(2)).

310. Call ToInt32(Result(4)).

311. Apply the bitwise operator @ to Result(5) and Result(6). The result is a signed 32 bit integer.

312. Return Result(7).

11.11 BINARY LOGICAL OPERATORS

Syntax

LogicalANDExpression :

BitwiseORExpression

LogicalANDExpression && *BitwiseORExpression*

LogicalORExpression :

LogicalANDExpression

LogicalORExpression || *LogicalANDExpression*

Semantics

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

313. Evaluate LogicalANDExpression.

314. Call GetValue(Result(1)).

315. Call ToBoolean(Result(2)).

316. If Result(3) is false, return Result(2).

317. Evaluate BitwiseORExpression.

318. Call GetValue((Result(5))).

319. Return Result(6).

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

320. Evaluate LogicalORExpression.

321. Call GetValue(Result(1)).

322. Call ToBoolean(Result(2)).

323. If Result(3) is true, return Result(2).

324. Evaluate LogicalANDExpression.

325. Call GetValue(Result(5)).

326. Return Result(6).

11.12 CONDITIONAL OPERATOR (? :)

Syntax

ConditionalExpression :

LogicalORExpression

LogicalORExpression ? *AssignmentExpression* : *AssignmentExpression*

Semantics

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

327. Evaluate LogicalORExpression.

328. Call GetValue(Result(1)).

329. Call ToBoolean(Result(2)).

- 330. If Result(3) is false, go to step 8.
- 331. Evaluate the first AssignmentExpression.
- 332. Call GetValue(Result(5)).
- 333. Return Result(6).
- 334. Evaluate the second AssignmentExpression.
- 335. Call GetValue(Result(8)).
- 336. Return Result(9).

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

11.13 ASSIGNMENT OPERATORS

Syntax

AssignmentExpression :
ConditionalExpression
LeftHandSideExpression *AssignmentOperator* *AssignmentExpression*

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

11.13.1 Simple Assignment (=)

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

- 337. Evaluate *LeftHandSideExpression*.
- 338. Evaluate AssignmentExpression.
- 339. Call GetValue(Result(2)).
- 340. Call PutValue(Result(1), Result(3)).
- 341. Return Result(3).

11.13.2 Compound Assignment (op=)

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

- 342. Evaluate *LeftHandSideExpression*.
- 343. Call GetValue(Result(1)).
- 344. Evaluate AssignmentExpression.
- 345. Call GetValue(Result(2)).
- 346. Apply operator @ to Result(3) and Result(4).
- 347. Call PutValue(Result(1), Result(5)).
- 348. Return Result(5).

11.14 COMMA OPERATOR (,)

Syntax

Expression :
AssignmentExpression
Expression , *AssignmentExpression*

Semantics

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

- 349. Evaluate Expression.
- 350. Call GetValue(Result(1)).
- 351. Evaluate AssignmentExpression.
- 352. Call GetValue(Result(3)).
- 353. Return Result(4).

12 STATEMENTS

Syntax

Statement :

- Block*
- VariableStatement*
- EmptyStatement*
- ExpressionStatement*
- IfStatement*
- IterationStatement*
- ContinueStatement*
- BreakStatement*
- ReturnStatement*
- WithStatement*

Block :

- { *StatementList*_{opt} }

StatementList :

- Statement*
- StatementList Statement*

Semantics

The production *Block* : { } is evaluated as follows:

354. Return $\hat{\text{normal completion}}$.

The production *Block* : { *StatementList* } is evaluated as follows:

355. Evaluate *StatementList*.

356. Return *Result*(1).

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

357. Evaluate *StatementList*.

358. If *Result*(1) is an abrupt completion, return *Result*(1).

359. Evaluate *Statement*.

360. If *Result*(3) is a value completion, return *Result*(3).

361. If *Result*(1) is not a value completion, return *Result*(3).

362. Let *V* be the value carried by *Result*(1).

363. If *Result*(3) is $\hat{\text{abrupt completion}}$ because of **break**,
return $\hat{\text{abrupt completion}}$ after value *V* because of **break**.

364. If *Result*(3) is $\hat{\text{abrupt completion}}$ because of **continue**,
return $\hat{\text{abrupt completion}}$ after value *V* because of **continue**.

365. Return $\hat{\text{normal completion}}$ after value *V*.

12.1 VARIABLE STATEMENT

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 grammar 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:

366. Evaluate *VariableDeclarationList*.

367. Return Ònormal completionÓ.

The production *VariableDeclarationList* : *VariableDeclaration* is evaluated as follows:

368. Evaluate *VariableDeclaration*.

The production *VariableDeclarationList* : *VariableDeclarationList* , *VariableDeclaration* is evaluated as follows:

369. Evaluate *VariableDeclarationList*.

370. Evaluate *VariableDeclaration*.

The production *VariableDeclaration* : *Identifier* is evaluated as follows:

371. Evaluate *Identifier*.

ISSUE: Does it really evaluate the identifier, or does it take no action?

The production *VariableDeclaration* : *Identifier* *Initializer* is evaluated as follows:

372. Evaluate *Identifier*.

373. Evaluate *Initializer*.

374. Call GetValue(Result(2)).

375. Call PutValue(Result(1), Result(3)).

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

376. Evaluate *AssignmentExpression*.

377. Return Result(1).

12.2 EMPTY STATEMENT

Syntax

EmptyStatement :
 ;

Semantics

The production *EmptyStatement* : ; is evaluated as follows:

378. Return Ònormal completionÓ.

12.3 EXPRESSION STATEMENT

Syntax

ExpressionStatement :
Expression ;

Semantics

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

379. Evaluate *Expression*.

380. Call GetValue(Result(1)).

381. Return \hat{O} normal completion after value $V\hat{O}$, where the value V is Result(2).

12.4 THE **if** STATEMENT

Syntax

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

Semantics

The production *IfStatement* : **if** (*Expression*) *Statement* **else** *Statement* is evaluated as follows:

382. Evaluate *Expression*.

383. Call GetValue(Result(1)).

384. Call ToBoolean(Result(2)).

385. If Result(3) is **false**, go to step 7.

386. Evaluate the first *Statement*.

387. Return Result(5).

388. Evaluate the second *Statement*.

389. Return Result(7).

The production *IfStatement* : **if** (*Expression*) *Statement* is evaluated as follows:

390. Evaluate *Expression*.

391. Call GetValue(Result(1)).

392. Call ToBoolean(Result(2)).

393. If Result(3) is **false**, return \hat{O} normal completion \hat{O} .

394. Evaluate *Statement*.

395. Return Result(5).

12.5 ITERATION STATEMENTS

Syntax

IterationStatement :
while (*Expression*) *Statement*
for (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (*LeftHandSideExpression* **in** *Expression*) *Statement*
for (**var** *Identifier* *Initializer*_{opt} **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 a **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”.

12.5.1 The `while` Statement

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

396. Let *C* be \hat{O} normal completion \hat{O} .
397. Evaluate *Expression*.
398. Call GetValue(Result(1)).
399. Call ToBoolean(Result(2)).
400. If Result(3) is **false**, go to step 12.
401. Evaluate *Statement*.
402. If Result(6) a value completion, change *C* to be \hat{O} normal completion after value *V* \hat{O} where *V* is the value carried by Result(6).
403. If Result(6) is a **break** completion, go to step 12.
404. If Result(6) is a **continue** completion, go to step 1.
405. If Result(6) is a **return** completion, return Result(6).
406. Go to step 2.
407. Return *C*.

12.5.2 The `for` Statement

The production *IterationStatement* : **for** (*Expression* ; *Expression* ; *Expression*) *Statement* is evaluated as follows:

408. If the first *Expression* is not present, go to step 4.
409. Evaluate the first *Expression*.
410. Call GetValue(Result(2)). (This value is not used.)
411. Let *C* be \hat{O} normal completion \hat{O} .
412. If the second *Expression* is not present, go to step 9.
413. Evaluate the second *Expression*.
414. Call GetValue(Result(5)).
415. Call ToBoolean(Result(6)).
416. If Result(7) is **false**, go to step 19.
417. Evaluate *Statement*.
418. If Result(10) a value completion, change *C* to be \hat{O} normal completion after value *V* \hat{O} where *V* is the value carried by Result(6).
419. If Result(10) is a **break** completion, go to step 19.
420. If Result(10) is a **continue** completion, go to step 5.
421. If Result(10) is a **return** completion, return Result(10).
422. If the third *Expression* is not present, go to step 5.
423. Evaluate the third *Expression*.
424. Call GetValue(Result(11)). (This value is not used.)
425. Go to step 5.
426. Return *C*.

ISSUE: Finish the necessary changes for the other three forms of iteration statement for value returns.

The production *IterationStatement* : **for** (**var** *VariableDeclarationList* ; *Expression* ; *Expression*) *Statement* is evaluated as follows:

427. Evaluate *VariableDeclarationList*.
428. If the second *Expression* is not present, go to step 7.
429. Evaluate the second *Expression*.
430. Call GetValue(Result(3)).
431. Call ToBoolean(Result(4)).
432. If Result(5) is **false**, go to step 15.
433. Evaluate *Statement*.
434. If Result(7) is \hat{O} abrupt completion because of **break** \hat{O} , go to step 15.
435. If Result(7) is \hat{O} abrupt completion because of **continue** \hat{O} , go to step 11.

- 436. If Result(7) is *Ó*abrupt completion because of **return** *VÓ*, return Result(7).
- 437. If the third *Expression* is not present, go to step 2.
- 438. Evaluate the third *Expression*.
- 439. Call GetValue(Result(11)). (This value is not used.)
- 440. Go to step 2.
- 441. Return *Ó*normal completion*Ó*.

12.5.3 The `for...in` Statement

The production *IterationStatement* : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement* is evaluated as follows:

- 442. Evaluate the *Expression*.
- 443. Call GetValue(Result(1)).
- 444. Call ToObject(Result(2)).
- 445. Get the name of the next property of Result(3) which doesn't have the DontEnum attribute. If there is no such property, go to step 12.
- 446. Evaluate the *LeftHandSideExpression* (it may be evaluated repeatedly).
- 447. Call PutValue(Result(5), Result(4)).
- 448. Evaluate *Statement*.
- 449. If Result(7) is *Ó*abrupt completion because of **break***Ó*, go to step 12.
- 450. If Result(7) is *Ó*abrupt completion because of **continue***Ó*, go to step 4.
- 451. If Result(7) is *Ó*abrupt completion because of **return** *VÓ*, return Result(7).
- 452. Go to step 4.
- 453. Return *Ó*normal completion*Ó*.

The production *IterationStatement* : **for** (**var** *Identifier* *Initializer*_{opt} **in** *Expression*) *Statement* is evaluated as follows:

- 454. If the *Initializer* is not present, go to step 6.
- 455. Evaluate the *Identifier*.
- 456. Evaluate the *Initializer*.
- 457. Call GetValue(Result(3)).
- 458. Call PutValue(Result(2), Result(4)).
- 459. Evaluate the *Expression*.
- 460. Call GetValue(Result(6)).
- 461. Call ToObject(Result(7)).
- 462. Get the name of the next property of Result(8) which doesn't have the DontEnum attribute. If there is no such property, go to step 17.
- 463. Evaluate the *Identifier* (yes, it may be evaluated repeatedly).
- 464. Call PutValue(Result(10), Result(9)).
- 465. Evaluate *Statement*.
- 466. If Result(12) is *Ó*abrupt completion because of **break***Ó*, go to step 17.
- 467. If Result(12) is *Ó*abrupt completion because of **continue***Ó*, go to step 9.
- 468. If Result(12) is *Ó*abrupt completion because of **return** *VÓ*, return Result(12).
- 469. Go to step 9.
- 470. Return *Ó*normal completion*Ó*.

The mechanics of enumerating the properties (step 4) 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: Need to talk about enumerating properties of the prototype, and so on, recursively. Are shadowed properties of the prototype(s) enumerated? (I hope not!)

12.6 THE `continue` STATEMENT

Syntax

ContinueStatement :
continue ;

An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a **continue** statement that is not within at least one **while** or **for** statement. The **continue** statement is evaluated as:

471. Return abrupt completion because of **continue**.

12.7 THE **break** STATEMENT

Syntax

BreakStatement :
break ;

An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a **break** statement that is not within at least one **while** or **for** statement. The **break** statement is evaluated as:

472. Return abrupt completion because of **break**.

12.8 THE **return** STATEMENT

Syntax

ReturnStatement :
return [no *LineTerminator* here] *Expression_{opt}* ;

An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a **return** statement that is not within 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*.

The production *ReturnStatement* :: **return** [no *LineTerminator* here] *Expression_{opt}* ; is evaluated as:

473. If the *Expression* is not present, return abrupt completion because of **return undefined**.

474. Evaluate *Expression*.

475. Call GetValue(Result(2)).

476. Return abrupt completion because of **return V**, where the value *V* is Result(3).

12.9 THE **with** STATEMENT

Syntax

WithStatement :
with (*Expression*) *Statement*

Description

The **with** statement adds a computed object to the front of the scope chain of the current execution context, then executes a statement with this augmented scope chain, then restores the scope chain.

Semantics

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

477. Evaluate *Expression*.

478. Call GetValue(Result(1)).

479. Call ToObject(Result(2)).

480. Add Result(3) to the front of the scope chain.

481. Evaluate *Statement* using the augmented scope chain from step 4.

482. Remove Result(3) from the front of the scope chain.

483. Return Result(5).

Discussion

Note that no matter how control leaves the embedded *Statement*, whether normally or by some form of abrupt completion, the scope chain is always restored to its former state.

13FUNCTION 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 instead of to the global object..

14PROGRAM

Syntax

Program :
SourceElements

SourceElements :
SourceElement
SourceElements SourceElement

SourceElement :
Statement
FunctionDefinition

15 NATIVE ECMAScript OBJECTS

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

Many built-in objects are functions: they can be invoked with arguments. Some of them furthermore are constructors: they are functions intended for use with the **new** operator. A few of them are intended to be used both as ordinary functions and as constructors. For each built-in function, this specification describes the arguments required by that function and properties of the function object. For each built-in constructor, this specification furthermore describes properties of the prototype object of that constructor and properties of specific object instances returned by a **new** expression that invokes that constructor.

Every built-in function object described in this section—whether as a constructor, an ordinary function, or both—has a **length** property whose value is an integer. Unless otherwise specified, this value is equal to the number of named arguments shown in the section heading for the function description; for example, the function object that is the initial value of the **indexOf** property (section of the String prototype object is described under the section heading `indexOf(searchString, position)` which shows the two named arguments *searchString* and *position*; therefore the value of the **length** property of that function object is **2**. Sometimes the same function object is described under more than one heading to emphasize its different behaviors when given different numbers of actual arguments; in such a case, unless otherwise specified, the **length** value is the largest number of arguments shown in any applicable section heading. For example, the function object that is the initial value of the **Object** property of the global object is described under four separate headings: as a function of one argument (section 15.2.1.1), as a function of zero arguments (section 15.2.1.2), as a constructor of one argument (15.2.2.1), and as a constructor of zero arguments (15.2.2.2). The largest number of arguments described is 1, so that value of the **length** property of that function object is **1**.

In every case, a **length** property of a built-in function object described in this section has the attributes `ReadOnly`, `DontDelete`, and `DontEnum` (and no others). Every other property described in this section has the attribute `DontEnum` (and no others) unless otherwise specified.

15.1 THE GLOBAL OBJECT

The global object does not have a `[[Construct]]` property; it is not possible to use the global object as a constructor with the **new** operator.

15.1.1 Value Properties of the Global Object

15.1.1.1 NaN

The initial value of **NaN** is `NaN`.

15.1.1.2 Infinity

The initial value of **Infinity** is `+∞`.

15.1.2 Function Properties of the Global Object

15.1.2.1 eval(x)

484. If *x* is not a string value, return *x*.

485. Parse *x* as an ECMAScript *Program*. If the parse fails, generate a runtime error

486. Evaluate the program from step 2.

487.If Result(3) is Ònormal completion after value V Ó, return the value V .

488.Return **undefined**.

15.1.2.2parseInt(string, radix)

15.1.2.3parseFloat(string)

15.1.2.4escape(string)

15.1.2.5unescape(string)

15.1.2.6isNaN(number)

Applies ToNumber to its argument, then returns **true** if the result is **NaN**, and otherwise returns **false**.

15.1.2.7isFinite(number)

Applies ToNumber to its argument, then returns **false** if the result is **NaN**, $+\infty$, or $-\infty$, and otherwise returns **true**.

15.2OBJECT OBJECTS

15.2.1The Object Function

When **Object** is called as a function rather than as a constructor, it performs a type conversion.

15.2.1.1Object(value)

When the **Object** function is called with one argument *value*, the following steps are taken:

489.If the *value* is **null** or **undefined**, create and return a newobject with no properties (other than internal properties) exactly as for the expression **new Object()**.

490.Return ToObject(*value*).

15.2.1.2Object()

When the **Object** function is called with no arguments, the following step is taken:

491.Create and return a newobject with no properties (other than internal properties) exactly as for the expression **new Object()**.

15.2.2The Object Constructor

When **Object** is called as part of a **new** expression, it is a constructor that may create an object.

15.2.2.1new Object(value)

When the **Object** constructor is called with one argument *value*, the following steps are taken:

492.If the type of the *value* is not Object, go to step 6.

493.If the value is a native ECMAScript object, do not create a new object; simply return *value*.

494.If the value is a host object, then either step 4 or step 5 may be executed, the choice being made in an implementation-dependent manner that may depend on the host object.

495.Return *value*.

496.Create a new native ECMAScript object to serve as a ÒwrapperÓ for the host object.

The [[Prototype]] property of the newly constructed object is set to *value*.

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

The [[Value]] property of the newly constructed object is set to *value*.

Return the newly created native object.

497.If the type of the value is not String, go to step 8.

498.Create a new native ECMAScript object to serve as a ÒwrapperÓ for the string value, exactly as for the expression **new String(value)**.

The [[Prototype]] property of the newly constructed object is set to the String prototype object.

- The `[[Class]]` property of the newly constructed object is set to **"String"**.
 The `[[Value]]` property of the newly constructed object is set to *value*.
 Return the newly created native object.
- 499.If the type of the value is not Boolean, go to step 10.
- 500.Create a new native ECMAScript object to serve as a wrapper for the boolean value, exactly as for the expression **new Boolean(value)**.
 The `[[Prototype]]` property of the newly constructed object is set to the Boolean prototype object.
 The `[[Class]]` property of the newly constructed object is set to **"Boolean"**.
 The `[[Value]]` property of the newly constructed object is set to *value*.
 Return the newly created native object.
- 501.If the type of the value is not Number, go to step 12.
- 502.Create a new native ECMAScript object to serve as a wrapper for the number value, exactly as for the expression **new Number(value)**.
 The `[[Prototype]]` property of the newly constructed object is set to the Number prototype object.
 The `[[Class]]` property of the newly constructed object is set to **"Number"**.
 The `[[Value]]` property of the newly constructed object is set to *value*.
 Return the newly created native object.
- 503.Create a new native ECMAScript object.
 The `[[Prototype]]` property of the newly constructed object is set to the Object prototype object.
 The `[[Class]]` property of the newly constructed object is set to **"Object"**.
 The newly constructed object has no `[[Value]]` property.
 Return the newly created native object.

15.2.2.2new Object()

When the **Object** constructor is called with no argument, the following step is taken:

- 504.Create a new native ECMAScript object.
 The `[[Prototype]]` property of the newly constructed object is set to the Object prototype object.
 The `[[Class]]` property of the newly constructed object is set to **"Object"**.
 The newly constructed object has no `[[Value]]` property.
 Return the newly created native object.

15.2.3Properties of the Object Constructor

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

Besides the internal `[[Call]]` and `[[Construct]]` properties and the **length** property, the Object constructor has the following properties:

15.2.3.1Object.prototype

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

15.2.4Properties of the Object Prototype Object

15.2.4.1constructor

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

15.2.4.2toString()

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

- 505.Get the `[[Class]]` property of this object.
 506.Call `ToString(Result(1))`.
 507.Compute a string value by concatenating the three strings "**object**", `Result(2)`, and `"]`".
 508.Return `Result(3)`.

15.2.4.3valueOf()

As a rule, the `valueOf` method for an object simply returns the object; but if the object is a `Wrapper` for a host object, as may perhaps be created by the `Object` constructor (see section 15.2.2.1), then the contained host object should be returned.

When the `valueOf` method is called, the following steps are taken:

509. Get the `[[Class]]` property of this object.
510. If `Result(1)` is `"Object"`, return this object.
511. Get the `[[Value]]` property of this object.
512. If `Result(3)` is `undefined`, return this object.
513. Return `Result(3)`.

15.2.5Properties of Object Instances

Object instances have no special properties beyond those inherited from the `Object` prototype object.

15.3FUNCTION OBJECTS

15.3.1The Function Function

When `Function` is called as a function rather than as a constructor, it creates and initializes a new function object. Thus the function call `Function(...)` is equivalent to the object creation expression `new Function(...)` with the same arguments.

15.3.1.1Function(*p*₁, *p*₂, . . . , *p*_{*n*}, *body*)

When the `Function` function is called with some arguments *p*₁, *p*₂, . . . , *p*_{*n*}, *body* (where *n* might be 0, that is, there are no arguments), the following steps are taken:

514. Create and return a new `Function` object exactly as for the expression
`new Function(p1, p2, . . . , pn, body)`.

15.3.2The Function Constructor

When `Function` is called as part of a `new` expression, it is a constructor: it initializes the newly created object.

15.3.2.1Function(*p*₁, *p*₂, . . . , *p*_{*n*}, *body*)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

When the `Function` constructor is called with some arguments *p*₁, *p*₂, . . . , *p*_{*n*}, *body* (where *n* might be 0, that is, there are no arguments), the following steps are taken:

515. Let *P* be the empty string.
516. If no arguments were given, let *body* be the empty string and go to step 13.
517. If one argument was given, call it *body* and go to step 13.
518. Let *A* be the first argument.
519. Let *P* be `Tostring(Result(4))`.
520. Let *k* be 2.
521. If *k* equals the number of arguments, call the *k*th argument *body* and go to step 13.
522. Let *A* be the *k*th argument.
523. Call `Tostring(Result(8))`.
524. Let *P* be the result of concatenating the previous value of *P*, the string `,` (a comma), and `Result(9)`.
525. Increase *k* by 1.
526. Go to step 7.
527. Call `Tostring(body)`.
528. Let *F* be the newly constructed `Function` object.
529. The `[[Class]]` property of *F* is set to `"Function"`.
530. Create an executable function whose name is `"Anonymous"`, whose formal parameters are specified by *P* and whose body is specified by `Result(13)`. The string value *P* must be parsable

as a *FormalParameterList_{opt}*; the string value result(13) must be parsable as a *StatementList_{opt}*. (Note that both *P* and Result(13) may contain whitespace, line terminators, and comments.) However, if either *P* or Result(13) is syntactically incorrect, or otherwise cannot be interpreted as part of a correct ECMAScript function definition, then return an implementation-dependent value that is not a Function object.

531. The `[[Value]]` property of *F* is set to Result(16).

532. Compute, as an integer number value of positive sign, the number of formal parameters that resulted from the parse of *P* as a *FormalParameterList_{opt}*.

533. The **length** property of *F* is set to Result(18).

534. Create a new object as if by the expression **new Object()**.

535. The **prototype** property of *F* is set to Result(20).

536. The **arguments** property of *F* is set to **null**.

Note that it is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
new Function( a ,  b ,  c ,  return a+b+c )  
new Function( a, b, c ,  return a+b+c )  
new Function( a,b ,  c ,  return a+b+c )
```

A **prototype** property is automatically created for every function, against the possibility that the function will be used as a constructor. However, note that this prototype object is not automatically given a **constructor** property. It is conventional for a prototype object to have a **constructor** property whose value is the Function object whose **prototype** property has that prototype as its value, but the creation of this property, if that is desired, is the responsibility of the programmer.

The **length** property that is created in step 19 does *not* have the `ReadOnly`, `DontDelete`, or `DontEnum` attribute. The **prototype** property that is created in step 21 does *not* have the `DontEnum` attribute.

15.3.3 Properties of the Function Constructor

15.3.3.1 Function.prototype

The value of **Function.prototype** is the built-in Function prototype object.

15.3.3.2 Function.length

The **length** property is **1**. (Of course, the Function constructor accepts more than one argument, because it accepts a variable number of arguments.)

15.3.4 Properties of the Function Prototype Object

Note that the Function prototype object is itself a Function object.

WHAT HAPPENS WHEN YOU INVOKE IT???

15.3.4.1 constructor

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

15.3.4.2 toString()

An implementation-dependent representation of the function is returned. This representation has the syntax of a *FunctionDeclaration*. Note in particular that the use and placement of whitespace, line terminators, and semicolons within the representation string is implementation-dependent.

15.3.4.3 valueOf()

For Function objects, **valueOf** does the same thing as **toString**.

15.3.5 Properties of Function Instances

15.3.5.1 length

The value of the **length** property is usually an integer that indicates the  typical  number of arguments expected by the function. However, the language permits the function to be invoked with

some other number of arguments. The behavior of a function when invoked on a number of arguments other than the number specified by its **length** property depends on the function.

15.3.5.2 prototype

The value of the **prototype** property may be used to initialize the internal `[[Prototype]]` property of a newly created object when the Function object is invoked as a constructor for that newly created object.

15.3.5.3 arguments

The value of the **arguments** property is normally **null** if there is no outstanding invocation of the function in progress (that is, the function has been called but has not yet returned). When a Function object is invoked, its **arguments** property is `[[Dynamically bound]]` to a newly created Array object that contains the arguments on which it was invoked; that is, the old value of the **arguments** property is saved and the new Array object becomes its new value; then, when that function invocation has completed execution, the old value of the **arguments** property is restored as the function returns to its caller.

15.4 ARRAY OBJECTS

Array objects give special treatment to a certain class of property names. A property name P (in the form of a string value) is an *array index* if and only if `ToString(ToUint32(P))` is equal to P and `ToUint32(P)` is not equal to $2^{32}-1$. Every Array object has a **length** property whose value is always an integer with positive sign and less than 2^{32} . It is always the case that the **length** property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the **length** property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the **length** property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to properties of the Array object itself and is unaffected by **length** or array index properties that may be inherited from its prototype.

15.4.1 The Array Constructor

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

15.4.1.1 `new Array(item0, item1, . . .)`

This description applies if and only if the Array constructor is given two or more arguments.

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

The **length** property of the newly constructed object is set to the number of arguments.

The **0** property of the newly constructed object is set to *item0*; the **1** property of the newly constructed object is set to *item1*; and, in general, for as many arguments as there are, the k property of the newly constructed object is set to argument k , where the first argument is considered to be argument number **0**.

15.4.1.2 `new Array(len)`

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

If the argument *len* is a number, then the **length** property of the newly constructed object is set to *len*.

If the argument *len* is not a number, then the **length** property of the newly constructed object is set to **1** and the **0** property of the newly constructed object is set to *len*.

15.4.1.3 `new Array()`

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

The **length** property of the newly constructed object is set to **+0**.

15.4.2 Properties of the Array Constructor

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

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

15.4.2.1 Array.prototype

The value of `Array.prototype` is the built-in Array prototype object.

15.4.2.2 Array.length

The `length` property is `1`. (Of course, the Array constructor accepts more than one argument, because it accepts a variable number of arguments.)

15.4.3 Properties of the Array Prototype Object

Note that the Array prototype object is itself an array; it has a `length` property (whose initial value is `+0`) and the special internal `[[Put]]` method described in section 15.4.4.1. In following descriptions of functions that are properties of the Array prototype object, the phrase “this object” refers to the object that is the `this` value for the invocation of the function. It is permitted for `this` to refer to an object for which the value of the internal `[[Class]]` property is not `"Array"`.

15.4.3.1 constructor

The initial value of `Array.prototype.constructor` is the built-in `Array` constructor.

15.4.3.2 toString()

The elements of this object are converted to strings, and these strings are then concatenated, separated by comma characters. The result is the same as if the built-in `join` method were invoked for this object with no argument.

15.4.3.3 valueOf()

The elements of this object are converted to strings, and these strings are then concatenated, separated by comma characters. The result is the same as if the built-in `join` method were invoked for this object with no argument.

In other words, for Array objects, `valueOf` does the same thing as `toString`.

15.4.3.4 join(separator)

The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the *separator*. If no separator is provided, a single comma is used as the separator.

When the `join` method is called with one argument *separator*, the following steps are taken:

537. Call the `[[Get]]` method of this object with argument `"length"`.

538. Call `ToUint32(Result(1))`.

539. If *separator* is undefined or not supplied, let *separator* be the single-character string `" , "`.

540. Call `ToString(separator)`.

541. If `Result(2)` is `0`, return the empty string.

542. Call the `[[Get]]` method of this object with argument `0`.

543. If `Result(6)` is `undefined` or `null`, use the empty string; otherwise, call `ToString(Result(6))`.

544. Let *R* be `Result(7)`.

545. Let *k* be `1`.

546. If *k* equals `Result(2)`, return *R*.

547. Let *S* be a string value produced by concatenating *R* and `Result(4)`.

548. Call the `[[Get]]` method of this object with argument *k*.

549. If `Result(12)` is `undefined` or `null`, use the empty string; otherwise, call `ToString(Result(12))`.

550. Let *R* be a string value produced by concatenating *S* and `Result(13)`.

551. Increase *k* by 1.

552. Go to step 10.

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

15.4.3.5 reverse()

The elements of the array are rearranged so as to reverse their order. This object is returned as the result of the call.

553. Call the `[[Get]]` method of this object with argument **"length"**.

554. Call `ToUint32(Result(1))`.

555. Compute `floor(Result(2)/2)`.

556. Let k be **0**.

557. If k equals `Result(3)`, return this object.

558. Compute `Result(2) - k - 1`.

559. Call the `[[Get]]` method of this object with argument k .

560. Call the `[[Get]]` method of this object with argument `Result(6)`.

561. Call the `[[Put]]` method of this object with arguments k and `Result(8)`.

562. Call the `[[Put]]` method of this object with arguments `Result(6)` and `Result(7)`.

563. Increase k by 1.

564. Go to step 5.

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

15.4.3.6 sort()

565. Call the `[[Get]]` method of this object with argument **"length"**.

566. Call `ToUint32(Result(1))`.

567. Perform an implementation-dependent sequence of calls to the `[[Get]]` and `[[Put]]` methods of this object where the first argument for each call is a nonnegative integer less than `Result(2)`. After this sequence is complete, this object must have the following two properties. First, there must be some mathematical permutation π of the nonnegative integers less than `Result(2)`, such that for every nonnegative integer j less than `Result(2)`, **new** $[\pi(j)]$ is exactly the same value as **old** $[j]$.

Second, for all nonnegative integers j and k , each less than `Result(2)`, if either **old** $[j] < \text{old}[k]$, or both $j < k$ and **old** $[j] \leq \text{old}[k]$, then $\pi(j) < \pi(k)$. Here we use the notation **old** $[j]$ to refer to the hypothetical result of calling the `[[Get]]` method of this object with argument j before this step is executed, and the notation **new** $[j]$ to refer to the hypothetical result of calling the `[[Get]]` method of this object with argument j after this step has been completely executed.

568. Return this object.

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

15.4.4 Properties of Array Instances

Array instances inherit properties from the Array prototype object and also have the following properties.

15.4.4.1 `[[Put]](P, V)`

Array objects use a variation of the `[[Put]]` method used for other native ECMAScript objects (section 8.6.2.2).

Assume A is an Array object and P is a string.

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

569. Call the `[[CanPut]]` method of A with name P .

570. If `Result(1)` is false, return.

571. If A doesn't have a property with name P , go to step 7.

572. If P is **"length"**, go to step 12.

- 573. Set the value of property *P* of *A* to *V*.
- 574. Go to step 8.
- 575. Create a property with name *P*, set its value to *V* and give it empty attributes.
- 576. If *P* is not an array index, return.
- 577. If *A* itself has a property (not an inherited property) named "**length**", and `ToUint32(P)` is less than the value of the **length** property of *A*, then return.
- 578. Change (or set) the value of the **length** property of *A* to `ToUint32(P)+1`.
- 579. Return.
- 580. Compute `ToUint32(V)`.
- 581. For every integer *k* that is less than the value of the **length** property of *A* but not less than `Result(12)`, if *A* itself has a property (not an inherited property) named `ToString(k)`, then delete that property.
- 582. Set the value of property *P* of *A* to `Result(12)`.
- 583. Return.

15.4.4.2length

The **length** property of this Array object is always numerically greater than the name of every property whose name is an array index.

The **length** property has the `DontDelete` attribute.

15.5STRING OBJECTS

15.5.1The String Function

When **string** is called as a function rather than as a constructor, it performs a type conversion.

15.5.1.1String(value)

Returns a string value (not a String object) computed by `ToString(value)`.

15.5.1.2String()

Returns the empty string "".

15.5.2The String Constructor

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

15.5.2.1new String(value)

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

The `[[Value]]` property of the newly constructed object is set to `ToString(value)`.

15.5.2.2new String()

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

The `[[Value]]` property of the newly constructed object is set to the empty string.

15.5.3Properties of the String Constructor

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

Besides the internal `[[Call]]` and `[[Construct]]` properties and the **length** property, the String constructor has the following properties:

15.5.3.1String.prototype

The value of **String.prototype** is the built-in String prototype object.

15.5.3.2 String.fromCharCode(char0, char1, . . .)

Returns a string value containing as many characters as the number of arguments. Each argument specifies one character of the resulting string, with the first argument specifying the first character, and so on, from left to right. An argument is converted to a character by applying the operation `ToUint16` (section 9.7) and regarding the resulting 16-bit integer as the Unicode encoding of a character. If no arguments are supplied, the result is the empty string.

15.5.4 Properties of the String Prototype Object

In following descriptions of functions that are properties of the String prototype object, the phrase "this String object" refers to the object that is the **this** value for the invocation of the function; it is an error if **this** does not refer to an object for which the value of the internal `[[Class]]` property is not "**String**". Also, the phrase "this string value" refers to the string value represented by this String object, that is, the value of the internal `[[Value]]` property of this String object.

15.5.4.1 constructor

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

15.5.4.2 toString()

Returns this string value. (Note that, for a String object, the `toString` method happens to return the same thing as the `valueOf` method.)

15.5.4.3 valueOf()

Returns this string value.

15.5.4.4 charAt(pos)

Returns a string containing the character at position *pos* in this string. If there is no character at that position, the result is the empty string. The result is a string value, not a String object.

If *pos* is a value of Number type, then the result of `x.charAt(pos)` is equal to the result of `x.substring(pos, pos+1)` except in the strange case where *pos* is greater than `-1` but less than `0`.

When the `charAt` method is called with one argument *pos*, the following steps are taken:

584. Call `ToString`, giving it the **this** value as its argument.

585. Call `ToInteger(pos)`.

586. Compute the number of characters in `Result(1)`.

587. If `Result(2)` is less than `0` or is not less than `Result(3)`, return the empty string.

588. Return a string of length `1`, containing one character from `Result(1)`, namely the character at position `Result(2)`, where the first (leftmost) character in `Result(1)` is considered to be at position `0`, the next one at position `1`, and so on.

Note that the `charAt` function is intentionally generic; it does not require that its **this** value be an array. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.5 charCodeAt(pos)

Returns a number (a nonnegative integer less than 2^{16}) representing the Unicode encoding of the character at position *pos* in this string. If there is no character at that position, the result is **NaN**.

When the `charCodeAt` method is called with one argument *pos*, the following steps are taken:

589. Call `ToString`, giving it the **this** value as its argument.

590. Call `ToInteger(pos)`.

591. Compute the number of characters in `Result(1)`.

592. If `Result(2)` is less than `0` or is not less than `Result(3)`, return **NaN**.

593. Return a value of Number type, of positive sign, whose magnitude is the Unicode encoding of one character from `Result(1)`, namely the character at position `Result(2)`, where the first (leftmost) character in `Result(1)` is considered to be at position `0`, the next one at position `1`, and so on.

Note that the `charCodeAt` function is intentionally generic; it does not require that its **this** value be an array. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.6indexOf(searchString, position)

If the given `searchString` appears as a substring of the result of converting this object to a string, at one or more positions that are at or to the right of the specified position, then the index of the leftmost such position is returned; otherwise **-1** is returned. If position is undefined or not supplied, 0 is assumed, so as to search all of the string.

When the **indexOf** method is called with two arguments *searchString* and *position*, the following steps are taken:

594. Call `ToString`, giving it the **this** value as its argument.

595. Call `ToString(searchString)`.

596. Call `ToInteger(position)`. (If *position* is **undefined** or not supplied, this step produces the value **0**).

597. Compute the number of characters in `Result(1)`.

598. Compute `min(max(Result(3), 0), Result(4))`.

599. Compute the number of characters in the string that is `Result(2)`.

600. Compute the smallest possible integer *k* not smaller than `Result(5)` such that *k*+`Result(6)` is not greater than `Result(4)`, and for all nonnegative integers *j* less than `Result(6)`, the character at position *k*+*j* of `Result(1)` is the same as the character at position *j* of `Result(2)`; but if there is no such integer *k*, then compute the value **-1**.

601. Return `Result(7)`.

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

15.5.4.7lastIndexOf(searchString, position)

If the given `searchString` appears as a substring of the result of converting this object to a string, at one or more positions that are at or to the left of the specified position, then the index of the rightmost such position is returned; otherwise **-1** is returned. If position is undefined or not supplied, the length of this string value is assumed, so as to search all of the string.

When the **lastIndexOf** method is called with two arguments *searchString* and *position*, the following steps are taken:

602. Call `ToString`, giving it the **this** value as its argument.

603. Call `ToString(searchString)`.

604. Call `ToNumber(position)`.

605. If `Result(3)` is **NaN**, use **+∞**; otherwise, call `ToInteger(Result(3))`.

606. Compute the number of characters in `Result(1)`.

607. Compute `min(max(Result(4), 0), Result(5))`.

608. Compute the number of characters in the string that is `Result(2)`.

609. Compute the largest possible integer *k* not larger than `Result(6)` such that *k*+`Result(7)` is not greater than `Result(5)`, and for all nonnegative integers *j* less than `Result(7)`, the character at position *k*+*j* of `Result(1)` is the same as the character at position *j* of `Result(2)`; but if there is no such integer *k*, then compute the value **-1**.

610. Return `Result(8)`.

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

15.5.4.8split(separator)

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

As a special case, if the separator is the empty string, the string is split up into individual characters; the length of the result array equals the length of the string, and each substring contains one character.

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

When the **split** method is called with one argument *separator*, the following steps are taken:

611. Call `ToString`, giving it the **this** value as its argument.

612. Create a new Array object of length **0** and call it *A*.

613. If *separator* is undefined or not supplied, call the `[[Put]]` method of *A* with `0` and `Result(1)` as arguments, and then return *A*.
614. Call `ToString(separator)`.
615. Compute the number of characters in `Result(1)`.
616. Compute the number of characters in the string that is `Result(4)`.
617. Let *p* be `0`.
618. If `Result(6)` is zero (the separator string is empty), go to step 17.
619. Compute the smallest possible integer *k* not smaller than *p* such that *k*+`Result(6)` is not greater than `Result(5)`, and for all nonnegative integers *j* less than `Result(6)`, the character at position *k*+*j* of `Result(1)` is the same as the character at position *j* of `Result(2)`; but if there is no such integer *k*, then go to step 14.
620. Compute a string value equal to the substring of `Result(1)`, consisting of the characters at positions *p* through *k*-1, inclusive.
621. Call the `[[Put]]` method of *A* with *A*.`length` and `Result(10)` as arguments.
622. Let *p* be *k*+`Result(6)`.
623. Go to step 9.
624. Compute a string value equal to the substring of `Result(1)`, consisting of the characters from position *p* to the end of `Result(1)`.
625. Call the `[[Put]]` method of *A* with *A*.`length` and `Result(13)` as arguments.
626. Return *A*.
627. If *p* equals `Result(5)`, return *A*.
628. Compute a string value equal to the substring of `Result(1)`, consisting of the single character at position *p*.
629. Call the `[[Put]]` method of *A* with *A*.`length` and `Result(17)` as arguments.
630. Increase *p* by 1.
631. Go to step 17.

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

15.5.4.9 substring(start)

Returns a substring of the result of converting this object to a string, starting from character position *start* and running to the end of the string. The result is a string value, not a String object.

If the argument is NaN or negative, it is replaced with zero; if the argument is larger than the length of the string, it is replaced with the length of the string.

When the `substring` method is called with one argument *start*, the following steps are taken:

632. Call `ToString`, giving it the `this` value as its argument.
633. Call `ToInteger(start)`.
634. Compute the number of characters in the string that is the value of the `[[Value]]` property of `this`.
635. Compute `min(max(Result(2), 0), Result(3))`.
636. Return a string whose length is the difference between `Result(3)` and `Result(4)`, containing characters from the string that is the value of the `[[Value]]` property of `this`, namely the characters with indices `Result(4)` through `Result(3)-1`, in ascending order.

15.5.4.10 substring(start, end)

Returns a substring of the result of converting this object to a string, starting from character position *start* and running to character position *end* of the string. The result is a string value, not a String object.

If either argument is NaN or negative, it is replaced with zero; if either argument is larger than the length of the string, it is replaced with the length of the string.

If *start* is larger than *end*, they are swapped.

When the `substring` method is called with two arguments *start* and *end*, the following steps are taken:

637. Call `ToString`, giving it the `this` value as its argument.
638. Call `ToInteger(start)`.
639. Call `ToInteger(end)`.
640. Compute the number of characters in `Result(1)`.

- 641. Compute `min(max(Result(2), 0), Result(4))`.
- 642. Compute `min(max(Result(3), 0), Result(4))`.
- 643. Compute `min(Result(5), Result(6))`.
- 644. Compute `max(Result(5), Result(6))`.
- 645. Return a string whose length is the difference between `Result(8)` and `Result(7)`, containing characters from `Result(1)`, namely the characters with indices `Result(7)` through `Result(8)-1`, in ascending order.

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

15.5.4.11toLowerCase

Returns a string equal in length to the length of the result of converting this object to a string. The result is a string value, not a String object.

Every character of the result is equal to the corresponding character of the string, unless that character has a Unicode 2.0 lowercase equivalent, in which case the lowercase equivalent is used instead.

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

15.5.4.12toUpperCase

Returns a string equal in length to the length of the result of converting this object to a string. The result is a string value, not a String object.

Every character of the result is equal to the corresponding character of the string, unless that character has a Unicode 2.0 uppercase equivalent, in which case the uppercase equivalent is used instead.

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

15.5.5 Properties of String Instances

String instances inherit properties from the String prototype object and also have a `[[Value]]` property and a **length** property.

The `[[Value]]` property is the string value represented by this String object.

15.5.5.1length

The number of characters in the String value represented by this String object.

Once a String object is created, this property is unchanging.

15.6 BOOLEAN OBJECTS

15.6.1 The Boolean Function

When **Boolean** is called as a function rather than as a constructor, it performs a type conversion.

15.6.1.1 Boolean(value)

Returns a boolean value (not a Boolean object) computed by `ToBoolean(value)`.

15.6.1.2 Boolean()

Returns **false**.

15.6.2 The Boolean Constructor

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

15.6.2.1 new Boolean(value)

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

The `[[Value]]` property of the newly constructed Boolean object is set to `ToBoolean(value)`.

15.6.2.2 `new Boolean()`

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

The `[[Value]]` property of the newly constructed Boolean object is set to `false`.

15.6.3 Properties of the Boolean Constructor

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

Besides the internal `[[Call]]` and `[[Construct]]` properties and the `length` property, the Boolean constructor has the following property:

15.6.3.1 `Boolean.prototype`

The value of `Boolean.prototype` is the built-in Boolean prototype object.

15.6.4 Properties of the Boolean Prototype Object

In following descriptions of functions that are properties of the Boolean prototype object, the phrase "this Boolean object" refers to the object that is the `this` value for the invocation of the function; it is an error if `this` does not refer to an object for which the value of the internal `[[Class]]` property is not `"Boolean"`. Also, the phrase "this boolean value" refers to the boolean value represented by this Boolean object, that is, the value of the internal `[[Value]]` property of this Boolean object.

15.6.4.1 `constructor`

The initial value of `Boolean.prototype.constructor` is the built-in `Boolean` constructor.

15.6.4.2 `toString()`

If this boolean value is `true`, then the string `"true"` is returned. Otherwise, this boolean value must be `false`, and therefore the string `"false"` is returned.

15.6.4.3 `valueOf()`

Returns this boolean value.

15.6.5 Properties of Boolean Instances

Boolean instances have no special properties beyond those inherited from the Boolean prototype object.

15.7 NUMBER OBJECTS

15.7.1 The Number Function

When `Number` is called as a function rather than as a constructor, it performs a type conversion.

15.7.1.1 `Number(value)`

Returns a number value (not a Number object) computed by `ToNumber(value)`.

15.7.1.2 `Number()`

Returns `+0`.

15.7.2 The Number Constructor

When `Number` is called as part of a `new` expression, it is a constructor: it initializes the newly created object.

15.7.2.1 new Number(value)

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

The `[[Value]]` property of the newly constructed object is set to `ToNumber(value)`.

15.7.2.2 new Number()

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

The `[[Value]]` property of the newly constructed object is set to **+0**.

15.7.3 Properties of the Number Constructor

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

Besides the internal `[[Call]]` and `[[Construct]]` properties and the **length** property, the Number constructor has the following property:

15.7.3.1 Number.prototype

The value of **Number.prototype** is the built-in Number prototype object.

15.7.3.2 Number.MAX_VALUE

The value of **Number.MIN_VALUE** is the largest positive finite value of the number type, which is approximately **1.7976931348623157e308**.

15.7.3.3 Number.MIN_VALUE

The value of **Number.MIN_VALUE** is the smallest positive nonzero value of the number type, which is approximately **5e-324**.

15.7.3.4 Number.NaN

The value of **Number.NaN** is NaN.

15.7.3.5 Number.NEGATIVE_INFINITY

The value of **Number.NEGATIVE_INFINITY** is $-\infty$.

15.7.3.6 Number.POSITIVE_INFINITY

The value of **Number.POSITIVE_INFINITY** is $+\infty$.

15.7.4 Properties of the Number Prototype Object

In following descriptions of functions that are properties of the Number prototype object, the phrase "this Number object" refers to the object that is the **this** value for the invocation of the function; it is an error if **this** does not refer to an object for which the value of the internal `[[Class]]` property is not **"Number"**. Also, the phrase "this number value" refers to the number value represented by this Number object, that is, the value of the internal `[[Value]]` property of this Number object.

15.7.4.1 constructor

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

15.7.4.2 toString()

This number value is given as an argument to the `ToString` operator ; the resulting string value is returned.

15.7.4.3 valueOf()

Returns this number value.

15.7.5 Properties of Number Instances

Number instances have no special properties beyond those inherited from the Number prototype object.

15.8 THE MATH OBJECT

The Math object is merely a single object that has some named properties, some of which are functions. The Math object does not have a `[[Construct]]` property; it is not possible to use the Math object as a constructor with the `new` operator.

Recall that, in this specification, the phrase “the number value for x ” means “the value of number type, not NaN but possibly infinite, that is closer than any other value of number type to the mathematical value x , but if x lies exactly halfway between two such values then the number value whose least significant bit is 0 is chosen”.

15.8.1 Value Properties of the Math Object

15.8.1.1 E

The number value for e , the base of the natural logarithms, which is approximately **2.7182818284590452354**.

15.8.1.2 LN10

The number value for the natural logarithm of 10, which is approximately **2.302585092994046**.

15.8.1.3 LN2

The number value for the natural logarithm of 2, which is approximately **0.6931471805599453**.

15.8.1.4 LOG2E

The number value for the base-2 logarithm of e , the base of the natural logarithms; this value is approximately **1.4426950408889634**. (Note that the value of `Math.LOG2E` is approximately the reciprocal of the value of `Math.LN2`.)

15.8.1.5 LOG10E

The number value for the base-10 logarithm of e , the base of the natural logarithms; this value is approximately **0.4342944819032518**. (Note that the value of `Math.LOG2E` is approximately the reciprocal of the value of `Math.LN2`.)

15.8.1.6 PI

The number value for π , the ratio of the circumference of a circle to its diameter, which is approximately **3.14159265358979323846**.

15.8.1.7 SQRT1_2

The number value for the square root of 1/2, which is approximately **0.7071067811865476**. (Note that the value of `Math.SQRT1_2` is approximately the reciprocal of the value of `Math.SQRT2`.)

15.8.1.8 SQRT2

The number value for the square root of 2, which is approximately **1.4142135623730951**.

15.8.2 Function Properties of the Math Object

Every function listed in this section applies the `ToNumber` operator to each of its arguments (in left-to-right order if there is more than one) and then performs a computation on the resulting number value(s).

The behavior of the functions `acos`, `asin`, `atan`, `atan2`, `cos`, `exp`, `log`, `pow`, `sin`, and `sqrt` is not precisely specified here. They are intended to compute approximations to the results of familiar mathematical functions, but some latitude is allowed in the choice of approximation algorithms. The general intent is that an implementor should be able to use the same mathematical library for

ECMAScript on a given hardware platform that is available to C programmers on that platform. Nevertheless, this specification recommends (though it does not require) the approximation algorithms for IEEE 754 arithmetic contained in **fdlibm**, the freely distributable mathematical library [XXXREF]. This specification also requires specific results for certain argument values that represent boundary cases of interest.

15.8.2.1abs(x)

Returns the absolute value of its argument; in general, the result has the same magnitude as the argument but has positive sign.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **-0**, the result is **+0**.
- If the argument is **-∞**, the result is **+∞**.

15.8.2.2acos(x)

Returns an implementation-dependent approximation to the arc cosine of the argument. The result is expressed in radians and ranges from **+0** to $+\pi$.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is greater than **1**, the result is **NaN**.
- If the argument is less than **-1**, the result is **NaN**.
- If the argument is exactly **1**, the result is **+0**.

15.8.2.3asin(x)

Returns an implementation-dependent approximation to the arc sine of the argument. The result is expressed in radians and ranges from $-\pi/2$ to $+\pi/2$.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is greater than **1**, the result is **NaN**.
- If the argument is less than **-1**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **-0**, the result is **-0**.

15.8.2.4atan(x)

Returns an implementation-dependent approximation to the arc tangent of the argument. The result is expressed in radians and ranges from $-\pi/2$ to $+\pi/2$.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **-0**, the result is **-0**.
- If the argument is **+∞**, the result is an implementation-dependent approximation to $+\pi/2$.
- If the argument is **-∞**, the result is an implementation-dependent approximation to $-\pi/2$.

15.8.2.5atan2(y, x)

Returns an implementation-dependent approximation to the arc tangent of the quotient **y/x** of the arguments **y** and **x**, where the signs of the arguments are used to determine the quadrant of the result. Note that it is intentional and traditional for the two-argument arc tangent function that the argument named **y** be first and the argument named **x** be second. The result is expressed in radians and ranges from $-\pi$ to $+\pi$.

- If either argument is **NaN**, the result is **NaN**.
- If **y>0** and **x** is **+0**, the result is an implementation-dependent approximation to $+\pi/2$.
- If **y>0** and **x** is **-0**, the result is an implementation-dependent approximation to $+\pi/2$.
- If **y** is **+0** and **x>0**, the result is **+0**.
- If **y** is **+0** and **x** is **+0**, the result is **+0**.
- If **y** is **+0** and **x** is **-0**, the result is an implementation-dependent approximation to $+\pi$.
- If **y** is **+0** and **x<0**, the result is an implementation-dependent approximation to $+\pi$.
- If **y** is **-0** and **x>0**, the result is **-0**.
- If **y** is **-0** and **x** is **+0**, the result is **-0**.

- If y is -0 and x is -0 , the result is an implementation-dependent approximation to $-\pi$.
- If y is -0 and $x < 0$, the result is an implementation-dependent approximation to $-\pi$.
- If $y < 0$ and x is $+0$, the result is an implementation-dependent approximation to $-\pi/2$.
- If $y < 0$ and x is -0 , the result is an implementation-dependent approximation to $-\pi/2$.
- If $y > 0$ and y is finite and x is $+\infty$, the result is $+0$.
- If $y > 0$ and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $+\pi$.
- If $y < 0$ and y is finite and x is $+\infty$, the result is -0 .
- If $y < 0$ and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $-\pi$.
- If y is $+\infty$ and x is finite, the result is an implementation-dependent approximation to $+\pi/2$.
- If y is $-\infty$ and x is finite, the result is an implementation-dependent approximation to $-\pi/2$.
- If y is $+\infty$ and x is $+\infty$, the result is an implementation-dependent approximation to $+\pi/4$.
- If y is $+\infty$ and x is $-\infty$, the result is an implementation-dependent approximation to $+3\pi/4$.
- If y is $-\infty$ and x is $+\infty$, the result is an implementation-dependent approximation to $-\pi/4$.
- If y is $-\infty$ and x is $-\infty$, the result is an implementation-dependent approximation to $-3\pi/4$.

15.8.2.6 `ceil(x)`

Returns the smallest (closest to $-\infty$) number value that is not less than the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$, the result is $+\infty$.
- If the argument is $-\infty$, the result is $-\infty$.
- If the argument is less than 0 but greater than -1 , the result is -0 .

The value of **Math.ceil(x)** is the same as the value of **-Math.floor(-x)**.

15.8.2.7 `cos(x)`

Returns an implementation-dependent approximation to the cosine of the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is 1 .
- If the argument is -0 , the result is 1 .
- If the argument is $+\infty$, the result is **NaN**.
- If the argument is $-\infty$, the result is **NaN**.

15.8.2.8 `exp(x)`

Returns an implementation-dependent approximation to the exponential function of the argument (e raised to the power of the argument, where e is the base of the natural logarithms).

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is 1 .
- If the argument is -0 , the result is 1 .
- If the argument is $+\infty$, the result is $+\infty$.
- If the argument is $-\infty$, the result is $+0$.

15.8.2.9 `floor(x)`

Returns the greatest (closest to $+\infty$) number value that is not greater than the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$, the result is $+\infty$.
- If the argument is $-\infty$, the result is $-\infty$.
- If the argument is greater than 0 but less than 1 , the result is $+0$.

The value of **Math.floor(x)** is the same as the value of **-Math.ceil(-x)**.

15.8.2.10log(x)

Returns an implementation-dependent approximation to natural logarithm of the argument.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is less than **0**, the result is **NaN**.
- If the argument is **+0** or **-0**, the result is **-∞**.
- If the argument is **1**, the result is **+0**.
- If the argument is **+∞**, the result is **+∞**.

15.8.2.11max(x, y)

Returns the larger of the two arguments.

- If either argument is **NaN**, the result is **NaN**.
- If **x>y**, the result is **x**.
- If **y>x**, the result is **y**.
- If **x** is **+0** and **y** is **+0**, the result is **+0**.
- If **x** is **+0** and **y** is **-0**, the result is **+0**.
- If **x** is **-0** and **y** is **+0**, the result is **+0**.
- If **x** is **-0** and **y** is **-0**, the result is **-0**.

15.8.2.12min(x, y)

Returns the smaller of the two arguments.

- If either argument is **NaN**, the result is **NaN**.
- If **x<y**, the result is **x**.
- If **y<x**, the result is **y**.
- If **x** is **+0** and **y** is **+0**, the result is **+0**.
- If **x** is **+0** and **y** is **-0**, the result is **-0**.
- If **x** is **-0** and **y** is **+0**, the result is **-0**.
- If **x** is **-0** and **y** is **-0**, the result is **-0**.

15.8.2.13pow(x, y)

Returns an implementation-dependent approximation to the result of raising **x** to the power **y**.

- If **y** is **NaN**, the result is **NaN**.
- If **y** is **+0**, the result is **1**, even if **x** is **NaN**.
- If **y** is **-0**, the result is **1**, even if **x** is **NaN**.
- If **x** is **NaN** and **y** is nonzero, the result is **NaN**.
- If **abs(x)>1** and **y** is **+∞**, the result is **+∞**.
- If **abs(x)>1** and **y** is **-∞**, the result is **+0**.
- If **abs(x)==1** and **y** is **+∞**, the result is **NaN**.
- If **abs(x)==1** and **y** is **-∞**, the result is **NaN**.
- If **abs(x)<1** and **y** is **+∞**, the result is **+0**.
- If **abs(x)<1** and **y** is **-∞**, the result is **+∞**.
- If **x** is **+∞** and **y>0**, the result is **+∞**.
- If **x** is **+∞** and **y<0**, the result is **+0**.
- If **x** is **-∞** and **y>0** and **y** is an odd integer, the result is **-∞**.
- If **x** is **-∞** and **y>0** and **y** is not an odd integer, the result is **+∞**.
- If **x** is **-∞** and **y<0** and **y** is an odd integer, the result is **-0**.
- If **x** is **-∞** and **y<0** and **y** is not an odd integer, the result is **+0**.
- If **x** is **+0** and **y>0**, the result is **+0**.
- If **x** is **+0** and **y<0**, the result is **+∞**.
- If **x** is **-0** and **y>0** and **y** is an odd integer, the result is **-0**.
- If **x** is **-0** and **y>0** and **y** is not an odd integer, the result is **+0**.
- If **x** is **-0** and **y<0** and **y** is an odd integer, the result is **-∞**.

- If x is -0 and $y < 0$ and y is not an odd integer, the result is $+\infty$.
- If $x < 0$ and x is finite and y is finite and y is not an integer, the result is **NaN**.

15.8.2.14 random()

Returns a number value with positive sign, greater than or equal to 0 but less than 1, chosen randomly or pseudorandomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

15.8.2.15 round(x)

Returns the number value that is closest to the argument and is equal to a mathematical integer. If two integer number values are equally close to the argument, then the result is the number value that is closer to $+\infty$. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$, the result is $+\infty$.
- If the argument is $-\infty$, the result is $-\infty$.
- If the argument is greater than 0 but less than 0.5, the result is $+0$.
- If the argument is less than 0 but greater than or equal to -0.5 , the result is -0 .

Note that **Math.round(3.5)** returns 4, but **Math.round(-3.5)** returns -3.

The value of **Math.round(x)** is the same as the value of **Math.floor(x+0.5)**, except when x is -0 ; for this case **Math.round(x)** returns -0 , but **Math.floor(x+0.5)** returns $+0$.

15.8.2.16 sin(x)

Returns an implementation-dependent approximation to the sine of the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$ or $-\infty$, the result is **NaN**.

15.8.2.17 sqrt(x)

Returns an implementation-dependent approximation to the square root of the argument.

- If the argument is **NaN**, the result is **NaN**.
- If the argument less than 0, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$, the result is $+\infty$.

15.8.2.18 tan(x)

Returns an implementation-dependent approximation to the tangent of the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is $+0$, the result is $+0$.
- If the argument is -0 , the result is -0 .
- If the argument is $+\infty$ or $-\infty$, the result is **NaN**.

15.9 DATE OBJECTS

15.9.1 Overview of Date Objects

A Date object contains a number indicating a particular instant in time to within a millisecond. The number may also be **NaN**, indicating that the Date object does not represent a specific instant of time.

The following sections define a number of functions for operating on time values. Note that, in every case, if any argument to such a function is **NaN**, the result will be **NaN**.

15.9.1.1 Time Range

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. Leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript number values can represent all integers from $i_{\text{Min}} = \text{D}9,007,199,254,740,991$ to $i_{\text{Max}} = 9,007,199,254,740,991$; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly $\text{D}100,000,000$ days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000 milliseconds to either side of 01 January, 1970 UTC. This span easily covers all of recorded human history and a fair amount of unrecorded human history.

15.9.1.2 Day Number and Time within Day

A given time value t belongs to day number

$$\text{Day}(t) = \text{floor}(t / \text{msPerDay})$$

where the number of milliseconds per day is

$$\text{msPerDay} = 86400000$$

The remainder is called the time within the day:

$$\text{TimeWithinDay}(t) = t \text{ modulo } \text{msPerDay}$$

15.9.1.3 Year Number

ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number y is therefore defined by

$$\begin{aligned} \text{DaysInYear}(y) &= 365 && \text{if } (y \text{ modulo } 4) \neq 0 \\ &= 366 && \text{if } (y \text{ modulo } 4) = 0 \text{ and } (y \text{ modulo } 100) \neq 0 \\ &= 365 && \text{if } (y \text{ modulo } 100) = 0 \text{ and } (y \text{ modulo } 400) \neq 0 \\ &= 366 && \text{if } (y \text{ modulo } 400) = 0 \end{aligned}$$

Of course all non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year y is given by:

$$\text{DayFromYear}(y) = 365 \cdot (y - 1970) + \text{floor}((y - 1969) / 4) - \text{floor}((y - 1901) / 100) + \text{floor}((y - 1601) / 400)$$

The time value of the start of a year is:

$$\text{TimeFromYear}(y) = \text{msPerDay} \cdot \text{DayFromYear}(y)$$

A time value determines a year by:

$$\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that } \text{TimeFromYear}(y) \leq t$$

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned} \text{InLeapYear}(t) &= 0 && \text{if } \text{DaysInYear}(\text{YearFromTime}(y)) = 365 \\ &= 1 && \text{if } \text{DaysInYear}(\text{YearFromTime}(y)) = 366 \end{aligned}$$

15.9.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping $\text{MonthFromTime}(t)$ from a time value t to a month number is defined by:

$$\begin{aligned} \text{MonthFromTime}(t) &= 0 && \text{if } 0 \leq \text{DayWithinYear}(t) < 31 \\ &= 1 && \text{if } 31 \leq \text{DayWithinYear}(t) < \end{aligned}$$

$59+\text{InLeapYear}(t)$	= 2	if $59+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$90+\text{InLeapYear}(t)$	= 3	if $90+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$120+\text{InLeapYear}(t)$	= 4	if $120+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$151+\text{InLeapYear}(t)$	= 5	if $151+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$181+\text{InLeapYear}(t)$	= 6	if $181+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$212+\text{InLeapYear}(t)$	= 7	if $212+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$243+\text{InLeapYear}(t)$	= 8	if $243+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$273+\text{InLeapYear}(t)$	= 9	if $273+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$304+\text{InLeapYear}(t)$	= 10	if $304+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$334+\text{InLeapYear}(t)$	= 11	if $334+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) <$
$365+\text{InLeapYear}(t)$		

where

$$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(t)$$

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that $\text{MonthFromTime}(0) = 0$, corresponding to Thursday, 01 January, 1970.

15.9.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping $\text{DateFromTime}(t)$ from a time value t to a month number is defined by:

$\text{DateFromTime}(t) = \text{DayWithinYear}(t) + 1$	if $\text{MonthFromTime}(t) = 0$
$= \text{DayWithinYear}(t) - 30$	if $\text{MonthFromTime}(t) = 1$
$= \text{DayWithinYear}(t) - 58 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 2$
$= \text{DayWithinYear}(t) - 89 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 3$
$= \text{DayWithinYear}(t) - 119 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 4$
$= \text{DayWithinYear}(t) - 150 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 5$
$= \text{DayWithinYear}(t) - 180 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 6$
$= \text{DayWithinYear}(t) - 211 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 7$
$= \text{DayWithinYear}(t) - 242 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 8$
$= \text{DayWithinYear}(t) - 272 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 9$
$= \text{DayWithinYear}(t) - 303 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 10$
$= \text{DayWithinYear}(t) - 333 - \text{InLeapYear}(t)$	if $\text{MonthFromTime}(t) = 11$

15.9.1.6 Week Day

The week day for a particular time value t is defined as

$$\text{WeekDay}(t) = (\text{day}(t) + 4) \text{ modulo } 7$$

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that $\text{WeekDay}(0) = 4$, corresponding to Thursday, 01 January, 1970.

15.9.1.7 Local Time Zone Adjustment

An implementation of ECMAScript is expected to determine the local time zone adjustment by whatever means are available. The local time zone adjustment is a value `LocalTZA` measured in milliseconds which when added to UTC represents the local *standard* time. Daylight saving time is *not* reflected by `LocalTZA`. The value `LocalTZA` does not vary with time but depends only on the geographic location.

15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to determine the daylight saving time algorithm by whatever means are available. The algorithm to determine the daylight saving time adjustment `DaylightSavingTA(t)`, measured in milliseconds, must depend only on four things:

(1) the time since the beginning of the year

$$t \ominus \text{TimeFromYear}(\text{YearFromTime}(t))$$

(2) whether it is a leap year

$$\text{InLeapYear}(t)$$

(3) the week day of the beginning of the year

$$\text{WeekDay}(\text{TimeFromYear}(\text{YearFromTime}(t)))$$

and (4) the geographic location.

The implementation of ECMAScript should not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the USA observed daylight saving time year round.

If the underlying operating system provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the operating system provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

$$\text{LocalTime}(t) = t + \text{LocalTZA} + \text{DaylightSavingTA}(t)$$

Conversion from local time to UTC is defined by

$$\text{UTC}(t) = t \ominus \text{LocalTZA} \ominus \text{DaylightSavingTA}(t \ominus \text{LocalTZA})$$

Note that $\text{UTC}(\text{LocalTime}(t))$ is not necessarily always equal to t .

15.9.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

$$\text{HourFromTime}(t) = \text{floor}(t / \text{msPerHour}) \text{ modulo } \text{HoursPerDay}$$

$$\text{MinFromTime}(t) = \text{floor}(t / \text{msPerMinute}) \text{ modulo } \text{MinutesPerHour}$$

$$\text{SecFromTime}(t) = \text{floor}(t / \text{msPerSecond}) \text{ modulo } \text{SecondsPerMinute}$$

$$\text{msFromTime}(t) = t \text{ modulo } \text{msPerSecond}$$

where

$$\text{HoursPerDay} = 24$$

$$\text{MinutesPerHour} = 60$$

$$\text{SecondsPerMinute} = 60$$

$$\text{msPerSecond} = 1000$$

$\text{msPerMinute} = \text{msPerSecond} \cdot \text{SecondsPerMinute} = 60000$

$\text{msPerHour} = \text{msPerMinute} \cdot \text{MinutesPerHour} = 3600000$

15.9.1.11 MakeTime(hour, min, sec, ms)

The operator MakeTime calculates a number of milliseconds from its four arguments, which must be ECMAScript number values. This operator functions as follows:

646. If *hour* is not finite or *min* is not finite or *sec* is not finite or *ms* is not finite, return **NaN**.

647. Call ToInteger(*hour*).

648. Call ToInteger(*min*).

649. Call ToInteger(*sec*).

650. Call ToInteger(*ms*).

651. Compute $\text{Result}(2) \cdot \text{msPerHour} + \text{Result}(3) \cdot \text{msPerMinute} + \text{Result}(4) \cdot \text{msPerSecond} + \text{Result}(5)$, performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators \cdot and $+$).

652. Return Result(6).

15.9.1.12 MakeDay(year, month, date)

The operator MakeDay calculates a number of days from its three arguments, which must be ECMAScript number values. This operator functions as follows:

653. If *year* is not finite or *month* is not finite or *date* is not finite, return **NaN**.

654. Call ToInteger(*year*).

655. Call ToInteger(*month*).

656. Call ToInteger(*date*).

657. Compute $\text{Result}(2) + \text{floor}(\text{Result}(3)/12)$.

658. Compute Result(3) modulo 12.

659. Find a value *t* such that $\text{YearFromTime}(t) == \text{Result}(5)$ and $\text{MonthFromTime}(t) == \text{Result}(6)$ and $\text{DateFromTime}(t) == 1$; but if this is not possible (because some argument is out of range or fractional), return **NaN**.

660. Compute $\text{Day}(\text{Result}(7)) + \text{Result}(4) - 1$.

661. Return Result(8).

15.9.1.13 MakeDate(day, time)

The operator MakeDate calculates a number of milliseconds from its two arguments, which must be ECMAScript number values. This operator functions as follows:

662. If *day* is not finite or *time* is not finite, return **NaN**.

663. Compute $\text{day} \cdot \text{msPerDay} + \text{time}$.

664. Return TimeClip(Result(2)).

15.9.1.14 TimeClip(time)

The operator TimeClip calculates a number of milliseconds from its argument, which must be an ECMAScript number value. This operator functions as follows:

665. If *time* is not finite, return **NaN**.

666. If $\text{abs}(\text{Result}(2)) > 8.64 \cdot 10^{15}$ (that is, $8.64 \cdot 10^{15}$), return **NaN**.

667. Return ToInteger(Result(2)).

15.9.2 The Date Constructor

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

15.9.2.1 new Date(year, month, date, hours, minutes, seconds)

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

The [[Value]] property of the newly constructed object is set as follows:

668. Call ToNumber(*year*).
669. Call ToNumber(*month*).
670. Call ToNumber(*date*).
671. Call ToNumber(*hours*).
672. Call ToNumber(*minutes*).
673. Call ToNumber(*seconds*).
674. Compute MakeDay(Result(1), Result(2), Result(3)).
675. Compute MakeTime(Result(4), Result(5), Result(6), 0).
676. Compute MakeDate(Result(7), Result(8)).
677. Set the [[Value]] property of the newly constructed object to Result(9).

15.9.2.2new Date(year, month, date, hours, minutes)

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

The [[Value]] property of the newly constructed object is set as follows:

678. Call ToNumber(*year*).
679. Call ToNumber(*month*).
680. Call ToNumber(*date*).
681. Call ToNumber(*hours*).
682. Call ToNumber(*minutes*).
683. Compute MakeDay(Result(1), Result(2), Result(3)).
684. Compute MakeTime(Result(4), Result(5), 0, 0).
685. Compute MakeDate(Result(6), Result(7)).
686. Set the [[Value]] property of the newly constructed object to Result(8).

15.9.2.3new Date(year, month, date, hours)

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

The [[Value]] property of the newly constructed object is set as follows:

687. Call ToNumber(*year*).
688. Call ToNumber(*month*).
689. Call ToNumber(*date*).
690. Call ToNumber(*hours*).
691. Compute MakeDay(Result(1), Result(2), Result(3)).
692. Compute MakeTime(Result(4), 0, 0, 0).
693. Compute MakeDate(Result(5), Result(6)).
694. Set the [[Value]] property of the newly constructed object to Result(7).

15.9.2.4new Date(year, month, day)

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

The [[Value]] property of the newly constructed object is set as follows:

695. Call ToNumber(*year*).
696. Call ToNumber(*month*).
697. Call ToNumber(*date*).
698. Compute MakeDay(Result(1), Result(2), Result(3)).
699. Compute MakeDate(Result(4), 0).
700. Set the [[Value]] property of the newly constructed object to Result(5).

15.9.2.5new Date(value)

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

The [[Value]] property of the newly constructed object is set as follows:

701. Call ToPrimitive(*value*).
702. If Type(Result(1)) is String, then go to step 8.
703. Let *V* be ToNumber(Result(1)).
704. If *V* is NaN, go to step 7.

705. If $\text{abs}(\text{Result}(2)) > 8.64 \times 10^{15}$ (that is, $8.64 \cdot 10^{15}$), go to step 7.
706. Set the `[[Value]]` property of the newly constructed object to V and return.
707. Set the `[[Value]]` property of the newly constructed object to `NaN` and return.
708. Call the `[[Get]]` method of the global object, passing `"Date"` as the argument.
709. Call `ToObject(GetValue(Result(8)))`.
710. Call the `[[Get]]` method of `Result(9)`, passing `"Parse"` as the argument.
711. If `Result(10)` is not an object, generate a runtime error.
712. Construct a singleton internal list containing `Result(1)`.
713. Call the `[[Call]]` method of `Result(10)`, passing `Result(12)` as the argument list.
714. Let V be `ToNumber(Result(13))`.
715. Go to step 4.

15.9.2.6 `new Date()`

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

The `[[Value]]` property of the newly constructed object is set to `NaN`.

15.9.3 Properties of the Date Constructor

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

Besides the internal `[[Call]]` and `[[Construct]]` properties and the `length` property (whose value is `6`), the `Date` constructor has the following properties:

15.9.3.1 `Date.prototype`

The value of `Date.prototype` is the built-in `Date` prototype object.

15.9.3.2 `Date.parse()`

The.

15.9.3.3 `Date.UTC()`

The.

15.9.4 Properties of the Date Prototype Object

In following descriptions of functions that are properties of the `Date` prototype object, the phrase "this Date object" refers to the object that is the `this` value for the invocation of the function; it is an error if `this` does not refer to an object for which the value of the internal `[[Class]]` property is not `"Date"`. Also, the phrase "this time value" refers to the number value for the time represented by this `Date` object, that is, the value of the internal `[[Value]]` property of this `Date` object.

15.9.4.1 constructor

The initial value of `Date.prototype.constructor` is the built-in `Date` constructor.

15.9.4.2 `toString()`

15.9.4.3 `valueOf()`

15.9.4.4 `getTime()`

716. If the `this` value is not an object whose `[[Class]]` property is `"Date"`, generate a runtime error.

717. Return this time value.

15.9.4.5 `getFullYear()`

This function is specified here for backwards compatibility only. The function `getFullYear` is much to be preferred for nearly all purposes, because it avoids the "year 2000 problem".

718. Let t be this time value.

719. If t is **NaN**, return **NaN**.
720. Return `YearFromTime(LocalTime(t)) - 1900`.

15.9.4.6 `getFullYear()`

721. Let t be this time value.
722. If t is **NaN**, return **NaN**.
723. Return `YearFromTime(LocalTime(t))`.

15.9.4.7 `getUTCFullYear()`

724. Let t be this time value.
725. If t is **NaN**, return **NaN**.
726. Return `YearFromTime(t)`.

15.9.4.8 `getMonth()`

727. Let t be this time value.
728. If t is **NaN**, return **NaN**.
729. Return `MonthFromTime(LocalTime(t))`.

15.9.4.9 `getUTCMonth()`

730. Let t be this time value.
731. If t is **NaN**, return **NaN**.
732. Return `MonthFromTime(t)`.

15.9.4.10 `getDate()`

733. Let t be this time value.
734. If t is **NaN**, return **NaN**.
735. Return `DateFromTime(LocalTime(t))`.

15.9.4.11 `getUTCDate()`

736. Let t be this time value.
737. If t is **NaN**, return **NaN**.
738. Return `DateFromTime(t)`.

15.9.4.12 `getDay()`

739. Let t be this time value.
740. If t is **NaN**, return **NaN**.
741. Return `WeekDay(LocalTime(t))`.

15.9.4.13 `getUTCDay()`

742. Let t be this time value.
743. If t is **NaN**, return **NaN**.
744. Return `WeekDay(t)`.

15.9.4.14 `getHours()`

745. Let t be this time value.
746. If t is **NaN**, return **NaN**.
747. Return `HourFromTime(LocalTime(t))`.

15.9.4.15 `getUTCHours()`

748. Let t be this time value.
749. If t is **NaN**, return **NaN**.
750. Return `HourFromTime(t)`.

15.9.4.16getMinutes()

751. Let t be this time value.

752. If t is **NaN**, return **NaN**.

753. Return `MinFromTime(LocalTime(t))`.

15.9.4.17getUTCMinutes()

754. Let t be this time value.

755. If t is **NaN**, return **NaN**.

756. Return `MinFromTime(t)`.

15.9.4.18getSeconds()

757. Let t be this time value.

758. If t is **NaN**, return **NaN**.

759. Return `SecFromTime(LocalTime(t))`.

15.9.4.19getUTCSeconds()

760. Let t be this time value.

761. If t is **NaN**, return **NaN**.

762. Return `SecFromTime(t)`.

15.9.4.20getMilliseconds()

763. Let t be this time value.

764. If t is **NaN**, return **NaN**.

765. Return `msFromTime(LocalTime(t))`.

15.9.4.21getUTCMilliseconds()

766. Let t be this time value.

767. If t is **NaN**, return **NaN**.

768. Return `msFromTime(t)`.

15.9.4.22getTimezoneOffset()

Returns the difference between local time and UTC time in minutes.

769. Let t be this time value.

770. If t is **NaN**, return **NaN**.

771. Return $(t - \text{LocalTime}(t)) / \text{msPerMinute}$.

15.9.4.23setTime(time)

772. Call `ToNumber($time$)`.

773. Call `TimeClip(Result(1))`.

774. Set the `[[Value]]` property of the **this** value to `Result(2)`.

775. Return the value of the `[[Value]]` property of the **this** value.

776.

15.9.4.24setMilliseconds(ms)

777. Let t be the result of `LocalTime(this time value)`.

778. Call `ToNumber(ms)`.

779. If t is **NaN** or `Result(2)` is **NaN**, go to step 7.

780. Compute `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), Result(2))`.

781. Compute `UTC(MakeDate(Day(t), Result(4)))`.

782. Set the `[[Value]]` property of the **this** value to `Result(5)`.

783. Return the value of the `[[Value]]` property of the **this** value.

15.9.4.25setUTCMilliseconds(ms)

784. Let *t* be this time value.

785. Call `ToNumber(ms)`.

786. If *t* is **NaN** or `Result(2)` is **NaN**, go to step 7.

787. Compute `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), Result(2))`.

788. Compute `MakeDate(Day(t), Result(4))`.

789. Set the `[[Value]]` property of the **this** value to `Result(5)`.

790. Return the value of the `[[Value]]` property of the **this** value.

15.9.4.26setSeconds(sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value `getMilliseconds()`.

791. Let *t* be the result of `LocalTime(this time value)`.

792. Call `ToNumber(sec)`.

793. If *ms* is **undefined** or not specified, compute `msFromTime(t)`; otherwise, call `ToNumber(ms)`.

794. If *t* is **NaN** or `Result(2)` is **NaN** or `Result(3)` is **NaN**, go to step 8.

795. Compute `MakeTime(HourFromTime(t), MinFromTime(t), Result(2), Result(3))`.

796. Compute `UTC(MakeDate(Day(t), Result(5)))`.

797. Set the `[[Value]]` property of the **this** value to `Result(6)`.

798. Return the value of the `[[Value]]` property of the **this** value.

15.9.4.27setUTCSeconds(sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value `getUTCMilliseconds()`.

799. Let *t* be this time value.

800. Call `ToNumber(sec)`.

801. If *ms* is **undefined** or not specified, compute `msFromTime(t)`; otherwise, call `ToNumber(ms)`.

802. If *t* is **NaN** or `Result(2)` is **NaN** or `Result(3)` is **NaN**, go to step 8.

803. Compute `MakeTime(HourFromTime(t), MinFromTime(t), Result(2), Result(3))`.

804. Compute `MakeDate(Day(t), Result(5))`.

805. Set the `[[Value]]` property of the **this** value to `Result(6)`.

806. Return the value of the `[[Value]]` property of the **this** value.

15.9.4.28setMinutes(min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value `getSeconds()`.

If *ms* is not specified, this behaves as if *ms* were specified with the value `getMilliseconds()`.

807. Let *t* be the result of `LocalTime(this time value)`.

808. Call `ToNumber(min)`.

809. If *sec* is **undefined** or not specified, compute `SecFromTime(t)`; otherwise, call `ToNumber(sec)`.

810. If *ms* is **undefined** or not specified, compute `msFromTime(t)`; otherwise, call `ToNumber(ms)`.

811. If *t* is **NaN** or `Result(2)` is **NaN** or `Result(3)` is **NaN** or `Result(4)` is **NaN**, go to step 9.

812. Compute `MakeTime(HourFromTime(t), Result(2), Result(3), Result(4))`.

813. Compute `UTC(MakeDate(Day(t), Result(6)))`.

814. Set the `[[Value]]` property of the **this** value to `Result(7)`.

815. Return the value of the `[[Value]]` property of the **this** value.

15.9.4.29setUTCMinutes(min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value `getUTCSeconds()`.

If *ms* is not specified, this behaves as if *ms* were specified with the value `getUTCMilliseconds()`.

816. Let *t* be this time value.

817. Call `ToNumber(min)`.

818. If *sec* is **undefined** or not specified, compute `SecFromTime(t)`; otherwise, call `ToNumber(sec)`.

819. If *ms* is **undefined** or not specified, compute `msFromTime(t)`; otherwise, call `ToNumber(ms)`.

820. If *t* is **NaN** or `Result(2)` is **NaN** or `Result(3)` is **NaN** or `Result(4)` is **NaN**, go to step 9.

821. Compute `MakeTime(HourFromTime(t), Result(2), Result(3), Result(4))`.

822. Compute `MakeDate(Day(t), Result(6))`.

823. Set the [[Value]] property of the **this** value to Result(7).
824. Return the value of the [[Value]] property of the **this** value.

15.9.4.30 setHours(hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value getMinutes().
If *sec* is not specified, this behaves as if *sec* were specified with the value getSeconds().
If *ms* is not specified, this behaves as if *ms* were specified with the value getMilliseconds().
825. Let *t* be the result of LocalTime(this time value).

826. Call ToNumber(*hour*).
827. If *min* is **undefined** or not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
828. If *sec* is **undefined** or not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
829. If *ms* is **undefined** or not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
830. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN** or Result(4) is **NaN** or Result(5) is **NaN**, go to step 10.
831. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
832. Compute UTC(MakeDate(Day(*t*), Result(7))).
833. Set the [[Value]] property of the **this** value to Result(8).
834. Return the value of the [[Value]] property of the **this** value.

15.9.4.31 setUTCHours(hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value getUTCMinutes().
If *sec* is not specified, this behaves as if *sec* were specified with the value getUTCSeconds().
If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds().
835. Let *t* be this time value.
836. Call ToNumber(*hour*).
837. If *min* is **undefined** or not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
838. If *sec* is **undefined** or not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
839. If *ms* is **undefined** or not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
840. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN** or Result(4) is **NaN** or Result(5) is **NaN**, go to step 10.
841. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
842. Compute MakeDate(Day(*t*), Result(7)).
843. Set the [[Value]] property of the **this** value to Result(8).
844. Return the value of the [[Value]] property of the **this** value.

15.9.4.32 setDate(date)

845. Let *t* be the result of LocalTime(this time value).
846. Call ToNumber(*date*).
847. If *t* is **NaN** or Result(2) is **NaN**, go to step 7.
848. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
849. Compute UTC(MakeDate(Result(4), TimeWithinDay(*t*))).
850. Set the [[Value]] property of the **this** value to Result(5).
851. Return the value of the [[Value]] property of the **this** value.

15.9.4.33 setUTCDate(date)

852. Let *t* be this time value.
853. Call ToNumber(*date*).
854. If *t* is **NaN** or Result(2) is **NaN**, go to step 7.
855. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
856. Compute MakeDate(Result(4), TimeWithinDay(*t*)).
857. Set the [[Value]] property of the **this** value to Result(5).
858. Return the value of the [[Value]] property of the **this** value.

15.9.4.34setMonth(mon [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getDate().

859. Let *t* be the result of LocalTime(this time value).

860. Call ToNumber(*date*).

861. If *date* is **undefined** or not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).

862. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN**, go to step 8.

863. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).

864. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).

865. Set the [[Value]] property of the **this** value to Result(6).

866. Return the value of the [[Value]] property of the **this** value.

15.9.4.35setUTCMonth(mon [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate().

867. Let *t* be this time value.

868. Call ToNumber(*date*).

869. If *date* is **undefined** or not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).

870. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN**, go to step 8.

871. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).

872. Compute MakeDate(Result(5), TimeWithinDay(*t*)).

873. Set the [[Value]] property of the **this** value to Result(6).

874. Return the value of the [[Value]] property of the **this** value.

15.9.4.36setFullYear(year [, mon [, date]])

If *mon* is not specified, this behaves as if *mon* were specified with the value getMonth().

If *date* is not specified, this behaves as if *date* were specified with the value getDate().

875. Let *t* be the result of LocalTime(this time value).

876. Call ToNumber(*year*).

877. If *mon* is **undefined** or not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*mon*).

878. If *date* is **undefined** or not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).

879. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN** or Result(4) is **NaN**, go to step 8.

880. Compute MakeDay(Result(2), Result(3), Result(4)).

881. Compute UTC(MakeDate(Result(6), TimeWithinDay(*t*))).

882. Set the [[Value]] property of the **this** value to Result(7).

883. Return the value of the [[Value]] property of the **this** value.

15.9.4.37setUTCFullYear(year [, mon [, date]])

If *mon* is not specified, this behaves as if *mon* were specified with the value getUTCMonth().

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate().

884. Let *t* be this time value.

885. Call ToNumber(*year*).

886. If *mon* is **undefined** or not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*mon*).

887. If *date* is **undefined** or not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).

888. If *t* is **NaN** or Result(2) is **NaN** or Result(3) is **NaN** or Result(4) is **NaN**, go to step 8.

889. Compute MakeDay(Result(2), Result(3), Result(4)).

890. Compute MakeDate(Result(6), TimeWithinDay(*t*)).

891. Set the [[Value]] property of the **this** value to Result(7).

892. Return the value of the [[Value]] property of the **this** value.

15.9.4.38setYear(year)

This function is specified here for backwards compatibility only. The function **setFullYear** is much to be preferred for nearly all purposes, because it avoids the Year 2000 problem.

893. Let *t* be the result of LocalTime(this time value).

894. Call `ToNumber(year)`.
895. If t is `NaN` or `Result(2)` is `NaN`, go to step 8.
896. Call `ToInteger(Result(1))`.
897. If $0 \leq \text{Result}(3) \leq 99$, then `Result(4)` is `Result(3) + 1900`. Otherwise, `Result(4)` is `Result(3)`.
898. Compute `MakeDay(Result(5), MonthFromTime(t), DateFromTime(t))`.
899. Compute `UTC(MakeDate(Result(6), TimeWithinDay(t)))`.
900. Set the `[[Value]]` property of the **this** value to `Result(7)`.
901. Return the value of the `[[Value]]` property of the **this** value.

15.9.5 Properties of Date Instances

Date instances have no special properties beyond those inherited from the Date prototype object.

16 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?

17 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 Manuscript 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.

0APPENDIX A: OPEN ISSUES

0A.1 STRING NUMERIC LITERALS

Is it really true that an empty string, or a string containing only whitespace and/or a sign, is a valid string literal? Is the result a zero (negative zero if minus sign present)? Or is it necessary that at least one digit be present (or **Infinity**)?

1A.2 ARGUMENT

The arguments property of a function object is dynamically bound every time that function is called and restored as the call returns. (Agreed to on March 14; still needs to be done.)

Also need to write up a step-by-step, algorithmic account of function calls.

2A.3 VAR STATEMENTS AND EVAL

It was agreed that only the values of expression statements contribute to the result of eval. It follows that a var statement never contributes to the result of eval, or can its initializers contribute?

3A.4 USING ARRAY METHODS ON NON-ARRAYS

It was agreed that **join**, **reverse**, and **sort** could be used on non-arrays. All that is needed is a length. What is the behavior if the **length** property does not exist, or if its value is not of the Number type, or is not a nonnegative integer?

For now, I have assumed that ToUint32 is applied to the result of getting the **length** property.

4A.5 REVERSE OF NONEXISTENT PROPERTIES

Is **reverse** intended to preserve the status of nonexistent properties? For example, if an array object has properties **0**, **1**, and **3**, and its **length** is **4**, then will applying reverse cause it to have properties **0**, **2**, and **3** but no property **1**?

For now, I assume not; rather, **reverse** simply does gets and puts and therefore will cause all properties to exist, if only with **undefined** values.

1 APPENDIX B: PROPOSED EXTENSIONS

5B.1 THE CLASS STATEMENT¹

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.

6B.2 THE TRY AND THROW STATEMENTS¹

0B.2.1 The try Statement¹

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 transferred 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

```

1B.2.2 The Throw Statement¹

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

```

7 B.3 THE DATE TYPE¹

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 type¹

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)

2B.3.1 ToDate¹

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).

3B.3.2 ToDate Applied to the String Type

Issue: define this.

8B.4 IMPLICIT THIS³

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.

9 B.5 THE **switch** STATEMENT^{1,3}

Syntax

```
SwitchStatement :  
    switch ( Expression ) CaseBlock  
  
CaseBlock :  
    { CaseClausesopt }  
    { CaseClausesopt DefaultClause CaseClausesopt }  
  
CaseClauses :  
    CaseClause  
    CaseClauses CaseClause  
  
CaseClause :  
    case Expression : StatementListopt  
  
DefaultClause :  
    default : StatementListopt
```

Semantics

The *SwitchStatement* adds a label to the break label stack, which is described in section 12.5. 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:

902. If the continue label stack is not empty, PushContinue(9).
903. PushBreak(6).
904. Evaluate Expression.
905. Call GetValue(Result(3)).
906. Evaluate CaseBlock, passing it Result(4) as a parameter.
907. PopBreak(6).
908. If the continue label stack is not empty, PopContinue(9).
909. Return.
910. PopBreak(6).
911. PopContinue(9).
912. JumpContinue.

The production *CaseBlock* : { *CaseClauses*₁ *DefaultClause* *CaseClauses*₂ } is given an input parameter, *input*, and is evaluated as follows:

913. For the next CaseClause in CaseClauses₁, in source text order, evaluate CaseClause. If there is no such CaseClause, go to step 6.
914. If input is not equal to Result(1) (as defined by the != operator), go to step 1.
915. Execute the StatementList of this CaseClause.
916. Execute the StatementList of each subsequent CaseClause in CaseClauses₁.
917. Go to step 11.
918. For the next CaseClause in CaseClauses₂, in source text order, evaluate CaseClause. If there is no such CaseClause, go to step 11.
919. If input is not equal to Result(6) (as defined by the != operator), go to step 6.
920. Execute the StatementList of this CaseClause.
921. Execute the StatementList of each subsequent CaseClause in CaseClauses₂.
922. Return.
923. Execute the StatementList of DefaultClause.
924. Execute the StatementList of each CaseClause in CaseClauses₂.

925. Return.

If *CaseClauses₁* is omitted, steps 1 through 5 are omitted from execution. If *DefaultClause* is omitted (in which case *CaseClauses₂* is also omitted), steps 11 and 12 are omitted from execution. If *CaseClauses₂* 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:

926. Evaluate *Expression*.

927. Call GetValue(Result(1)).

928. 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.

10 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.

11 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.

12 B.8 SEALING OF AN OBJECT2

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.

13 B.9 THE ARGUMENTS KEYWORD³

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.

14 B.10 PREPROCESSOR

15 B.11 THE DO..WHILE STATEMENT

16B.12 BINARY OBJECT

17B.13 LABELS WITH BREAK AND CONTINUE

As in Java, allow statements to be labeled with an identifier followed by a colon. Allow a label to appear in a **break** or **continue** statement. The label referred to by a **break** or **continue** statement must be an iteration statement that contains the **break** or **continue** statement in question.

The use of labels makes code more readable and more robust. In addition, it makes possible certain transfers of control that otherwise could not be easily expressed at all.

2APPENDIX 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)

3 APPENDIX D: RESOLUTION HISTORY

18 D.1 JANUARY 15, 1997

4D.1.1 White Space

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

5D.1.2 Keywords

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

6D.1.3 Future Reserved Words

Update the 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.

7D.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 Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.4), unless the literal is a DecimalLiteral and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is significant if it is not part of an ExponentPart and (either it is not 0 or it is an important zero or there is no decimal point `Ô.Ô` in the literal). A digit 0 is an important zero if there is at least one important item to its left and at least one important item to its right within the literal. Any digit that is not 0 and is not part of an ExponentPart is an important item; a decimal point `Ô.Ô` is also an important item.)

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.

8D.1.5 ToPrimitive

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

9D.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)

10D.1.7 Attributes of Declared Functions and Built-in Objects

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)

11D.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 The Grouping Operator and removed the open issue at the bottom of this section)

12D.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 Prefix Increment Operator)

13D.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 Prefix Decrement Operator section)

14D.1.11 Multiplicative Operators

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

15D.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 -.

16D.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 The Left Shift Operator (<<))

17D.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 Binary Bitwise Operators)

18D.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 Conditional Operator (?:))

19D.1.16 Simple Assignment

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

20D.1.17 The for...in Statement

We decided to impose no restrictions on Expression1. (Removed the first open issue at the bottom of ISSUE: Finish the necessary changes for the other three forms of iteration statement for value returns.)

21D.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 The return Statement The second issue at the bottom of this section has been moved to)

22D.1.19 New Proposed Extensions

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

19D.2 JANUARY 24, 1997

23D.2.1 End Of Source

Updated Error: Reference source not found section to describe the end of source token as logical rather than physical `\u0000` since strings may contain embedded `\u0000` characters.

24D.2.2 Future Reserved Words

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

25D.2.3 White Space

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

26D.2.4 Comments

Added new issue to 3.2 regarding nested comments.

27D.2.5 Identifiers

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

28D.2.6 Numeric Literals

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

29D.2.7 String Literals

Updated the table describing the set of character escape characters in section Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the `Number` type. If the MV is 0, then the rounded value is `+0`; otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.4), unless the literal is a `DecimalLiteral` and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is significant if it is not part of an `ExponentPart` and (either it is not 0 or it is an important zero or there is no decimal point `Ô.Õ` in the literal). A digit 0 is an important zero if there is at least one important item to its left and at least one important item to its right within the literal. Any digit that is not 0 and is not part of an `ExponentPart` is an important item; a decimal point `Ô.Õ` is also an important item., to include a new column indicating the unicode value. Also added a new issue to the end of this section.

30D.2.8 Automatic Semicolon Insertion

Added two new issues to the end of .

31D.2.9 Property Attributes

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

32D.2.10 ToPrimitive

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

33D.2.11 ToNumber

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

34D.2.12 White Space

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

35D.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, . The difference between string numeric literals and numeric literals is that string numeric literals do not allow octal notation and do allow leading zeros.

36D.2.14 ToString

Updated section Note that ToUint32 maps -0 to +0.. Added Hint String in call to ToPrimitive.

37D.2.15 Postfix Increment and Decrement Operators

Updated section Error: Reference source not found. Updated the algorithm to return Result(3) (the result of converting ToNumber), rather than (Result(2)).

38D.2.16 The typeof operator

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

39D.2.17 Prefix Increment and Decrement Operators

Removed extraneous calls to ToPrimitive from the algorithm in section Prefix Increment Operator.

40D.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.

41D.2.19 The Subtraction Operator

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

42D.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.

43D.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.

44D.2.22 Equality Operators

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

45D.2.23 ToPrimitive Usage

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

46D.2.24 Binary Logical Operators

Added issue at the end of 7.10.

20D.3 JANUARY 31, 1997

47D.3.1 MultiLineComment

Updated the lexical production *MultiLineComment* in section Comments, to allow empty multi-line comments. Also removed the issue at the end of this section regarding nested multi-line comments. The *MultiLineComment* production continues to disallow multi-line comments.

48D.3.2 String Literals

Removed open issue at the end of section Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.4), unless the literal is a DecimalLiteral and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is significant if it is not part of an ExponentPart and (either it is not 0 or it is an important zero or there is no decimal point $\dot{O}.\dot{O}$ in the literal). A digit 0 is an important zero if there is at least one important item to its left and at least one important item to its right within the literal. Any digit that is not 0 and is not part of an ExponentPart is an important item; a decimal point $\dot{O}.\dot{O}$ is also an important item. which stated that the maximum string constant supported must be at least 32000 characters long.

49D.3.3 Automatic Semicolon Insertion

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

50D.3.4 The Number Type

Updated the description in section The String Type.

51D.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.

52D.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.

53D.3.7 The String type

Updated the description in section 4.6, The String Type.

54D.3.8 ToNumber

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

55D.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.

56D.3.10 ToInt32

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

57D.3.11 ToUint32

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

58D.3.12 Execution Contexts (Variables)

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

59D.3.13 Function Calls

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

60D.3.14 The typeof Operator

Updated the table in section 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.

61D.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.

62D.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.

63D.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.

64D.3.18 Conditional Operator (? :)

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

65D.3.19 Compound Assignment (op=)

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

21D.4 FEBRUARY 21, 1997

66D.4.1 Unicode Escape Sequences

Rewrote section Error: Reference source not found 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 Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.4), unless the literal is a DecimalLiteral and the literal has more than 20 significant digits, in which case the number value may be any implementation-dependent approximation to the MV. A digit is significant if it is not part of an ExponentPart and (either it is not 0 or it is an important zero or there is no decimal point $\hat{O}.\hat{O}$ in the literal). A digit 0 is an important zero if there is at least one important item to its left and at least one important item to its right within the literal. Any digit that is not 0 and is not part of an ExponentPart is an important item; a decimal point $\hat{O}.\hat{O}$ is also an important item..

67D.4.2 Future Reserved Words

Added **import** and **super** to table in Future Reserved Words.

68D.4.3 Automatic Semicolon Insertion

Rewrote the rules for semicolon insertion in section 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 The production Initializer : = AssignmentExpression is evaluated as follows:.

69D.4.4 The Number Type

Corrected formatting of formulae in section The String Type.

70D.4.5 NotImplicit and NotExplicit Property Attributes Deleted

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

71D.4.6.ToInt32 and ToUInt32

Corrected formatting of formulae in section.ToInt32: (signed 32 bit integer) and section 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.

72D.4.7 Grouping Operator

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

73D.4.8 Shift Expressions

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

74D.4.9 Conversion Rules for Relational Operators

Updated description in section 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.

75D.4.10 && and || Semantics

Updated description in section 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 || 0Hello0` is `0Hello0`.

76D.4.11 Conditional Operator

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

77D.4.12 Assignment Operators

Updated section 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.

78D.4.13 Syntax of Class Statement

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

79D.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*.

22D.5 FEBRUARY 27, 1997

80D.5.1 Grammar Notation

Big rewrite of section 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: `ÓexceptÓ` versus `Óbut notÓ`), and should be made consistent within itself and with section 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?

81D.5.2 End of Medium Character Is No Longer WhiteSpace

Deleted character `\u0019` (End of Medium) from the table in section White Space, and deleted `<EOM>` as an alternative for `SimpleWhiteSpace` in that same section. Also deleted `<EOM>` as an alternative for `StrWhiteSpaceChar` in section 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.

82D.5.3 Meaning of Null Literal

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

83D.5.4 Meaning of Boolean Literals

Added to section Semantics a discussion of the meaning of a boolean literal.

84D.5.5 Meaning of Numeric Literals

Added to section 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?

85D.5.6 Automatic Semicolon Insertion

Updated description of automatic semicolon insertion in section . Systematically replaced the word `ÓinjectedÓ` with `ÓinsertedÓ`. Invented a new theory of `Órestricted productionsÓ` 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 Left-Hand-Side Expressions and The return Statement to indicate the restrictions explicitly.

86D.5.7 The Number Type

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

87D.5.8 ToString on Numbers

Updated section 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.

88D.5.9 New Operator

Updated description in section The production `CallExpression : MemberExpression [Expression]` is evaluated in exactly the same manner, except that the contained `CallExpression` is evaluated in step 1. to describe the case where no argument list is provided. This needs to be reviewed.

89D.5.10 Delete Operator

Updated description in section 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.

90D.5.11 == Semantics

Updated section If Result(2) is a prefix of Result (1), return false. (A string value p is a prefix of string value q if q can be the result of concatenating p and some other string r. Note that any string is a prefix of itself, because r may be the empty string.) 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.

91D.5.12 && and || Semantics

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

92D.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).

93D.5.14 Dead Code Is Not Protected from Compile-Time Analysis

Added text to chapter 12 (Errors).

23D.6 MARCH 6, 1997

94D.6.1 Reformatted the Entire Document

In order to make future revisions easier and to take better advantage of the desktop-publishing capabilities of Word, the entire document was reformatted using some newly defined Word styles. Heading numbering was turned on to facilitate automatic numbering of headings in the main text (sections of the appendices are still numbered manually, using new styles Appendix Heading 1, Appendix Heading 2, and Appendix Heading 3). A new style Algorithm is used for algorithmic steps; in some cases, the last step should be styled with AlgorithmLast to provide extra vertical space after the last step.

Added a style called MathSpecialCase (generates bullet lists for now).

The title page now uses styles Title and Subtitle, which were modified to use appropriate fonts and paragraph spacing.

Extraneous tab characters and multiple spaces were deleted from all headings.

The paragraph spacing of Normal, the various headings, Algorithm, AlgorithmLast, SyntaxRule, and SyntaxDefinition were adjusted so that the correct vertical space is inserted automatically. All blank paragraphs in the document were deleted.

The index and all index entries were deleted. Sorry, but they were somehow interfering with other formatting, and the index entries were terribly incomplete anyway. If we have time to do a good index, entries can be added semi-systematically.

The document was divided into three of what Word calls "sections" so that the pages of the Table of Contents could be numbered with the customary roman numerals, with the main text starting on page 1.

All the revisions listed in this item were accepted and the change bars reset before the following items were entered, so that all the changes of this item would not clutter the manuscript.

95D.6.2 Designed a Section Outline for Chapter 11

Filled in nearly all necessary section headings for Chapter 1 for describing Object, Function, Array, String, Boolean, Number, and Math and all their properties and methods. Added a fair amount of boilerplate text.

96D.6.3 Defined Math Functions

Added complete definitions for all properties in the Math object, following the example of C9X for the treatment of IEEE 754 special cases.

24D.7 MARCH 10, 1997

97D.7.1 Added Definition of The Number Value for xO

In section 8.4, the phrase "the number value for xO" is now defined. It encapsulates the entire IEEE 754 process for converting any nonzero mathematical value to a representable value by using round-to-nearest mode. This phrase is of great use in Chapter 15 and elsewhere.

Also corrected two typos in this section: -1073 replaced by -1074 , and 2^{53} replaced by 2^{52} .

98D.7.2 atan and atan2 May Use Implementation-Dependent Values for π , etc.

It was decided at the phone meeting that when `Math.atan`, for example, is supposed to return $\pi/2$, it need not return exactly one-half the initial value of `Math.pi`, but may produce an approximation. The motivation is to allow implementors the use of whatever C math library is present on the hardware platform at hand, whether or not it conforms to the high quality standards of, for example, the C9X proposal.

99D.7.3 Improved Discussion of Input Stream for Syntactic Grammar

Text added to section 5.1 to better explain the handling of whitespace, comments, and line terminators, and the fact that line terminators become part of the input stream for the syntactic grammar. Also corrected a typo in section 5.1.5 where the phrase "[no *LineTerminator* here]" had been inadvertently omitted.

100D.7.4 Improved Treatment of LineTerminator in Lexical Grammar

Eliminated the mythical `<EOS>` character. As a result, *LineEnd* is not needed either. The trick is not to include *LineEnd* (or *LineTerminator*) as part of the grammar of a single-line comment. This works out better, because a single-line comment still runs to the end of the line (as dictated by the longest-token-possible rule), but it doesn't swallow the *LineTerminator*, so it doesn't affect automatic semicolon insertion. (That the previous production did swallow the *LineTerminator* was thus a bug.)

The section on whitespace has been divided into two sections, one on *WhiteSpace* (formerly called *SimpleWhiteSpace*) and one on Line Terminators.

THIS CHANGE REQUIRES REVIEW.

101D.7.5 Clarify Behavior of Unicode Escape Sequences

In Chapter 6, clarify that a Unicode escape sequence such as `\u000D` does not produce a carriage return that could end a single-line comment, for example.

102D.7.6 Add Careful Description of the String Value of a String Literal

In imitation of the text already present describing the value of a numeric literal, text was added to section 7.7.4 to describe carefully the exact sequence of characters represented by a string literal. In the process, missing productions for *DoubleStringCharacters* and *SingleStringCharacters* were added, and the redundant definitions of *HexDigit* and *OctalDigit* were removed. Also dealt with an open issue by emphasizing that a *LineTerminator* may not appear within a string literal.

103D.7.7 Description of Identifiers Reworded

Improvements to the wording in section 7.5. Also repaired a typo (capital **I** replaced by lowercase **i**).

104D.7.8 Table of Punctuators Corrected

Underscore replaced by + operator in table in section 7.6.

105D.7.9 Improved Descriptions of `ToInt32` and `ToUint32`

Step 5 of the algorithms in sections 9.5 and 9.6 have been clarified to use a mathematical description rather than fragments of code.

106D.7.10 Changes to ToString Applied to the Number Type

See section 9.8.1. Negative zero now produces "0", not "-0".. Integers less than 10^{20} shall print without decimal points. Values less than 1 but not less than 10^{-6} will not require scientific notation.

107D.7.11 Revised Syntax for NewExpression and MemberExpression

Made the changes to section 11.2 as suggested by Shon, eliminating *NewCallExpression* and providing a pleasing symmetry in which the number of **new** operators can exceed or fall short of the number of argument lists.

108D.7.12 Clarify Multiplicative and Additive Operators

In section 11.5.1, describe the multiplication of infinity by infinity.

In section 11.5.2, describe the division of infinity by zero.

In section 11.5.3, better describe the remainder of a zero by a finite number.

In section 11.6, better describe the sum of two zeros and the sum of finite numbers of same magnitude and opposite sign.

109D.7.13 Addition Operator No Longer Gives Hint Number

When the addition operator **+** calls *ToPrimitive*, it no longer gives hint Number. Note that all built-in objects respond to *ToPrimitive* without a hint as if hint Number were given, so this change affects only external objects.

110D.7.14 Correct Description of Relational Operators

Miscellaneous small corrections.

111D.7.15 Assignment Operator LHS Must Be PostfixExpression

Change four occurrences of *UnaryExpression* to *PostfixExpression* in section 11.13.

112D.7.16 Changes to For-in Loops

Without **var**, the expression before **in** must be a *PostfixExpression* (as for an assignment),

With **var**, an optional *Initializer* is permitted after the *Identifier*.

A For-In loop enumerates not only properties of the given object itself, but also properties of its prototype, and so on, recursively.

ISSUE: Are shadowed properties of the prototype enumerated?

113D.7.17 Break and Continue Must Occur within While or For Loop

Added text to sections 12.6 and 12.7 to require **break** and **continue** to appear within loop statements.

25D.8 MARCH 12, 1997

114D.8.1 Added Overview Chapter

Added a chapter at the beginning as a placeholder for introductory exposition.

115D.8.2 More Exposition about Internal Properties

Renamed section 8.6.2 from *Property Access* to the more general *Internal Properties and Methods*.

Added properties *[[Class]]*, *[[Value]]*, *[[CanPut]]*, and *[[DefaultValue]]* to the table so as to complete the list.

Added some discussion of these internal properties.

116D.8.3 Date Object

Added the Date object to chapter 15 and method descriptions, etc.

117D.8.4 Array, String, Boolean, Number Objects

Tons of work in chapter 15 to add method descriptions, etc.

118D.8.5 Math Object

Corrections to atan2 and floor.

26D.9 MARCH 24, 1997

119D.9.1 Numeric Literals

Revamped the grammar for numeric literals to simplify it (per Shon's suggestion) and added prose indicating that the number value for a decimal literal need only be an implementation-dependent approximation if there are more than 20 significant digits. In the process, the words "floating-point" disappear from the grammar; floating-point literals are merely one kind of decimal literal.

120D.9.2 String Numeric Literals

Revamped the grammar for string numeric literals and added prose indicating that the number value for a decimal literal need only be an implementation-dependent approximation if there are more than 20 significant digits. In the process, the words "floating-point" disappear from the grammar; floating-point literals are merely one kind of decimal literal.

Also added the text describing how to calculate a mathematical value for a string numeric literal. The details are a bit different from those for ordinary numeric literals.

27D.9.3 Prefix and Postfix Increment and Decrement Operators

Revise the grammar (for expository reasons), restructure the sections, and revise the algorithms for the `++` and `--` operators to be more precise.

28D.9.4 Left-Hand-Side Expressions

Revise the grammar for *PostfixExpression* so that uses of the postfix `++` and `--` operators cannot occur to the left of a `.` or `=`, for example. Now only a *LeftHandSideExpression* may occur on the left-hand side of an assignment or `in` keyword. Also updated the list of restricted productions accordingly in the description of automatic semicolon insertion (section 7.8).

29D.9.5 Reference Type

Revise description of the internal reference type (section 8.7).

30D.9.6 Infinities and Zeros

Decided to use the forms **NaN**, **positive zero**, **negative zero**, **positive infinity**, and **negative infinity** (Times bold, no caps except for **NaN**) consistently throughout the document to refer to those quantities. (Overridden by D.10.2.)

31D.9.7 Miscellaneous Small Corrections

Among the small corrections is the deletion of step 10, which was redundant, in the algorithm for the addition operator (section 11.6.1).

32D.10 MARCH 27, 1997

121D.10.1 Corrections to `[[CanPut]]` and `[[HasProperty]]`

Allow for the possibility that the `[[Prototype]]` is not implemented or has an undefined or primitive value.

122D.10.2 Discussion of Number Type

Explicitly introduce $+\infty$, $-\infty$, $+0$, and -0 as symbols used for expository purposes in this specification, and briefly point out the program expressions **NaN**, **+Infinity**, **-Infinity**, **+0**, and **-0**.

Then override D.9.6 to use the symbols in preference to **positive zero**, **negative zero**, **positive infinity**, and **negative infinity** in most places, as they have turned out to be visually clumsy..

123D.10.3 Infinity and NaN

Add properties **NaN** (initial value is NaN) and **Infinity** (initial value is) to the global object.

Specify that *Sign_{opt}* **Infinity** be recognized when `ToNumber` is applied to a string.

While I am at it, clarify the process of converting a mathematical value (MV) to a rounded value of Number type, both for numeric literals and for string numeric literals.

124D.10.4 `charCodeAt` and `String.fromCharCode`

Add **charCodeAt** method for String objects and **String.fromCharCode** function to the String object.. In support of the description of **String.fromCharCode**, add the `ToUint16` abstract operator.

125D.10.5 Last fraction digit from `ToString` applied to a number

Added discussion of the rule that if x is a number, `ToNumber(ToString(x))` must be the same as x .

126D.10.6 Multi-line comment containing line terminator treated as line terminator

In section 5.1.2, added text to say that a multi-line comment is simply discarded if it contains no line terminator; but if a multi-line comment contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

127D.10.7 Automatic semicolon insertion at end of source

Reworked the description of automatic semicolon insertion yet again. Handle end of course as a special case in section 7.8; as a consequence, it is not necessary to specially append a line terminator to the input stream in section 5.1.4.

128D.10.8 Added proposed extension for labelled break and continue

New section B.13 proposes the use of labels as in Java to allow transfer to other than the innermost containing loop.

129D.10.9 Lowercase `òeó` for scientific notation in `ToString` of a number

A lowercase `òeó` shall be used, not uppercase `ÒEÓ` (section 9.8.1).

130D.10.10 Evaluation of argument lists

Added algorithmic explanation of the evaluation of argument lists (section 11.2.4). To this end, invented yet another fictitious expository data type, `List` (section 8.8).

131D.10.11 For `ToPrimitive` of native objects, no hint is same as hint Number

Added a helpful note to section 11.6.1.

132D.10.12 Major overhaul of equality and relational operators

Revised descriptions to make them more precise, especially about **NaN**.

133D.10.13 String type

Moved the section on the String type and augmented its description (section 8.4).

33D.11 APRIL 9, 1997

134D.11.1 Added mathematical operators to notation section

Added definitions of sign, abs, floor, modulo to section 5.2.

135D.11.2 Added overview text

Added overview text from Richard Gabriel. Renumbered sections so that Scope, Conformance, and Normative References come first (ECMA format).

136D.11.3 Added Date stuff

Added material provided by Shon to section 15.9 and did a lot of reformatting.

137D.11.4 Lots of work on native objects chapter

In particular, corrected lots of typos per Shon and added material about Object and Function.

34D.12 APRIL 14, 1997

138D.12.1 Lots of corrections

Finished all my outstanding corrections to the native objects chapter.

(Still to come: work on the functions and program chapters and on function invocation, including dynamic binding of the **arguments** property.)

139D.12.2 Reworked Date type

Provided a more mathematical treatment, relying less on invoking other methods.

4APPENDIX E: LALR(1) SYNTACTIC GRAMMAR

Issue: To be supplied?

