Ecma/TC39/2009/006

Standard ECMA-262
3rd Edition - December 1999

ECMA

Standardizing Information and Communication Systems

SES Language Specification – DRAFT as derived from the "Mountain View Draft" of ECMAScript 3.1

Mark S. Miller 1/19/09 3:53 PM
Deleted: ECMAScript 3.1

Mark S. Miller 1/19/09 3:52 PM
Deleted: -

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

19 January 2009

Standard ECMA-262

3rd Edition - December 1999

ECMA

Standardizing Information and Communication Systems

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

19 January 2009

Standard ECMA-262
3rd Edition - December 1999

ECMA
Standardizing Information and Communication Systems

SES Language Specification – DRAFT as derived from the “Mountain View Draft” of ECMAScript 3.1

Mark S. Miller 1/19/09 3:54 PM
Deleted: ECMAScript 3.1 Language Specification - DRAFT

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

19 January 2009

Standard ECMA-262
3rd Edition - December 1999

ECMA

Standardizing Information and Communication Systems

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

19 January 2009

Brief History

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

The development of this Standard started in November 1996. The first edition of this ECMA Standard was adopted by the ECMA General Assembly of June 1997.

That ECMA Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The ECMA General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.

The third edition of the Standard includes powerful regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and minor changes in anticipation of forthcoming internationalisation facilities and future language growth. The language documented by the third edition has come to be known as ECMAScript 3 or ES3.

The language documented by the edition 3.1 has come to be known as ECMAScript 3.1 or ES3.1.

Caja defines the elements of safe web mashups by choosing sanitized subsets of existing web standards; including html, css, the browser/dom API, and ECMAScript. It defines two subsets of ES3.1-strict: Cajita and Valija. Cajita is a true object-capability language, meant to be a robust platform for new code expressing capability-based security policies. Valija is similar to ES3.1-strict, with some compromises to facilitate safely emulating multiple virtual Valija environments within one Cajita environment, embedded in one ES3.1 environment. Objects from multiple Valija environments can interoperate with each other, and with Cajita objects and "tamed uncajoled" ES3.1 objects from their hosting environment. Cajita is a subset of Valija, with some compromises to enable it to translate well into the variants of ES3 implemented on current widely deployed browsers.

The present variant of the ES3.1 draft spec proposes a design for Secure ECMAScript, or SES, based on Cajita, without the compromises needed to accommodate pre-ES3.1 browsers. However, to ease incremental adoption, the design of SES is constrained to be easily and efficiently implementable by translation to ES3.1-strict.

Mark S. Miller 1/19/09 5:16 PM
Deleted: Work on the language is not complete. The technical committee is working on significant enhancements, including mechanisms for scripts to be created and used across the Internet, and tighter coordination with other standards bodies such as groups within the World Wide Web Consortium and the Wireless Application Protocol Forum.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

19 January 2009

Table of contents

Mark S. Miller 1/20/09 12:00 AM
Deleted: 1
Mark S. Miller 1/20/09 12:00 AM
Deleted: 1
Mark S. Miller 1/20/09 12:00 AM
Deleted: 1
Mark S. Miller 1/20/09 12:00 AM
Deleted: 1
Mark S. Miller 1/20/09 12:00 AM
Deleted: 2
Mark S. Miller 1/20/09 12:00 AM
Deleted: 2
Mark S. Miller 1/20/09 12:00 AM
Deleted: 2
Mark S. Miller 1/20/09 12:00 AM
Deleted: 3
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 4
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 5
Mark S. Miller 1/20/09 12:00 AM
Deleted: 6
Mark S. Miller 1/20/09 12:00 AM
Deleted: 6
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Deleted: 7
Mark S. Miller 1/20/09 12:00 AM
Deleted: 7
Mark S. Miller 1/20/09 12:00 AM
Deleted: 7
Mark S. Miller 1/20/09 12:00 AM
Deleted: 8
Mark S. Miller 1/20/09 12:00 AM
Deleted: 10
Mark S. Miller 1/20/09 12:00 AM
Deleted: **11**
Mark S. Miller 1/20/09 12:00 AM
Deleted: **12**
Mark S. Miller 1/20/09 12:00 AM
Deleted: 12
Mark S. Miller 1/20/09 12:00 AM
Deleted: 12
Mark S. Miller 1/20/09 12:00 AM
Deleted: 13
Mark S. Miller 1/20/09 12:00 AM
Deleted: 14
Mark S. Miller 1/20/09 12:00 AM
Deleted: 15
Mark S. Miller 1/20/09 12:00 AM
Deleted: 15
Mark S. Miller 1/20/09 12:00 AM
Deleted: 15
Mark S. Miller 1/20/09 12:00 AM
Deleted: 15
Mark S. Miller 1/20/09 12:00 AM
Deleted: 16
Mark S. Miller 1/20/09 12:00 AM
Deleted: 17
Mark S. Miller 1/20/09 12:00 AM
Deleted: 17
Mark S. Miller 1/20/09 12:00 AM
Deleted: 17
Mark S. Miller 1/20/09 12:00 AM
Deleted: 17
Mark S. Miller 1/20/09 12:00 AM
Deleted: 17
Mark S. Miller 1/20/09 12:00 AM
Deleted: 20
Mark S. Miller 1/20/09 12:00 AM
Deleted: 22
Mark S. Miller 1/20/09 12:00 AM
Deleted: 23
Mark S. Miller 1/20/09 12:00 AM
Deleted: 23
Mark S. Miller 1/20/09 12:00 AM
Deleted: 24
Mark S. Miller 1/20/09 12:00 AM
Deleted: **25**
Mark S. Miller 1/20/09 12:00 AM
Deleted: 26
Mark S. Miller 1/20/09 12:00 AM
Deleted: 26
Mark S. Miller 1/20/09 12:00 AM
Deleted: 26
Mark S. Miller 1/20/09 12:00 AM
Deleted: 26
Mark S. Miller 1/20/09 12:00 AM
Deleted: 26
Mark S. Miller 1/20/09 12:00 AM
Deleted: 27
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Deleted: 34
Mark S. Miller 1/20/09 12:00 AM
Deleted: 34
Mark S. Miller 1/20/09 12:00 AM
Deleted: 34
Mark S. Miller 1/20/09 12:00 AM
Deleted: 35
Mark S. Miller 1/20/09 12:00 AM
Deleted: 35
Mark S. Miller 1/20/09 12:00 AM
Deleted: 35
Mark S. Miller 1/20/09 12:00 AM
Deleted: 35
Mark S. Miller 1/20/09 12:00 AM
Deleted: 36
Mark S. Miller 1/20/09 12:00 AM
Deleted: 36
Mark S. Miller 1/20/09 12:00 AM
Deleted: 36
Mark S. Miller 1/20/09 12:00 AM
Deleted: 36
Mark S. Miller 1/20/09 12:00 AM
Deleted: 36
Mark S. Miller 1/20/09 12:00 AM
Deleted: 37
Mark S. Miller 1/20/09 12:00 AM
Deleted: 38
Mark S. Miller 1/20/09 12:00 AM
Deleted: 38
Mark S. Miller 1/20/09 12:00 AM
Deleted: 39
Mark S. Miller 1/20/09 12:00 AM
Deleted: 39
Mark S. Miller 1/20/09 12:00 AM
Deleted: 39
Mark S. Miller 1/20/09 12:00 AM
Deleted: 42
Mark S. Miller 1/20/09 12:00 AM
Deleted: 42
Mark S. Miller 1/20/09 12:00 AM
Deleted: 42
Mark S. Miller 1/20/09 12:00 AM
Deleted: 43
Mark S. Miller 1/20/09 12:00 AM
Deleted: 43
Mark S. Miller 1/20/09 12:00 AM
Deleted: 43
Mark S. Miller 1/20/09 12:00 AM
Deleted: 44
Mark S. Miller 1/20/09 12:00 AM
Deleted: 44
Mark S. Miller 1/20/09 12:00 AM
Deleted: 45
Mark S. Miller 1/20/09 12:00 AM
Deleted: 45
Mark S. Miller 1/20/09 12:00 AM
Deleted: 45
Mark S. Miller 1/20/09 12:00 AM
Deleted: 45
Mark S. Miller 1/20/09 12:00 AM
Deleted: 46
Mark S. Miller 1/20/09 12:00 AM
Deleted: 46
Mark S. Miller 1/20/09 12:00 AM
Deleted: 46
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/19/09 5:14 PM

Deleted: 56
Mark S. Miller 1/20/09 12:00 AM
Deleted: 56
Mark S. Miller 1/20/09 12:00 AM
Deleted: 56
Mark S. Miller 1/20/09 12:00 AM
Deleted: 56
Mark S. Miller 1/20/09 12:00 AM
Deleted: 57
Mark S. Miller 1/20/09 12:00 AM
Deleted: 59
Mark S. Miller 1/20/09 12:00 AM
Deleted: 60
Mark S. Miller 1/20/09 12:00 AM
Deleted: 60
Mark S. Miller 1/20/09 12:00 AM
Deleted: 61
Mark S. Miller 1/20/09 12:00 AM
Deleted: 61
Mark S. Miller 1/20/09 12:00 AM
Deleted: 61
Mark S. Miller 1/20/09 12:00 AM
Deleted: 62
Mark S. Miller 1/20/09 12:00 AM
Deleted: 62
Mark S. Miller 1/20/09 12:00 AM
Deleted: 62
Mark S. Miller 1/20/09 12:00 AM
Deleted: 62
Mark S. Miller 1/20/09 12:00 AM
Deleted: 62
Mark S. Miller 1/20/09 12:00 AM
Deleted: 63
Mark S. Miller 1/20/09 12:00 AM
Deleted: 63
Mark S. Miller 1/20/09 12:00 AM
Deleted: 63
Mark S. Miller 1/20/09 12:00 AM
Deleted: 64
Mark S. Miller 1/20/09 12:00 AM
Deleted: 64
Mark S. Miller 1/20/09 12:00 AM
Deleted: 64
Mark S. Miller 1/20/09 12:00 AM
Deleted: 64
Mark S. Miller 1/20/09 12:00 AM
Deleted: 65
Mark S. Miller 1/20/09 12:00 AM
Deleted: 65
Mark S. Miller 1/20/09 12:00 AM
Deleted: 65
Mark S. Miller 1/20/09 12:00 AM
Deleted: 65
Mark S. Miller 1/20/09 12:00 AM
Deleted: 66
Mark S. Miller 1/20/09 12:00 AM
Deleted: 66
Mark S. Miller 1/20/09 12:00 AM
Deleted: 67
Mark S. Miller 1/20/09 12:00 AM
Deleted: 67
Mark S. Miller 1/20/09 12:00 AM
Deleted: 67
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Deleted: 71
Mark S. Miller 1/20/09 12:00 AM
Deleted: 72
Mark S. Miller 1/20/09 12:00 AM
Deleted: 72
Mark S. Miller 1/20/09 12:00 AM
Deleted: 72
Mark S. Miller 1/20/09 12:00 AM
Deleted: 73
Mark S. Miller 1/20/09 12:00 AM
Deleted: 73
Mark S. Miller 1/20/09 12:00 AM
Deleted: 73
Mark S. Miller 1/20/09 12:00 AM
Deleted: 74
Mark S. Miller 1/20/09 12:00 AM
Deleted: 74
Mark S. Miller 1/20/09 12:00 AM
Deleted: 75
Mark S. Miller 1/20/09 12:00 AM
Deleted: 76
Mark S. Miller 1/20/09 12:00 AM
Deleted: 76
Mark S. Miller 1/20/09 12:00 AM
Deleted: 76
Mark S. Miller 1/20/09 12:00 AM
Deleted: 77
Mark S. Miller 1/20/09 12:00 AM
Deleted: 78
Mark S. Miller 1/20/09 12:00 AM
Deleted: 78
Mark S. Miller 1/20/09 12:00 AM
Deleted: 79
Mark S. Miller 1/20/09 12:00 AM
Deleted: 80
Mark S. Miller 1/20/09 12:00 AM
Deleted: 80
Mark S. Miller 1/20/09 12:00 AM
Deleted: 80
Mark S. Miller 1/20/09 12:00 AM
Deleted: 81
Mark S. Miller 1/20/09 12:00 AM
Deleted: 81
Mark S. Miller 1/20/09 12:00 AM
Deleted: 81
Mark S. Miller 1/20/09 12:00 AM
Deleted: 81
Mark S. Miller 1/20/09 12:00 AM
Deleted: 82
Mark S. Miller 1/20/09 12:00 AM
Deleted: 82
Mark S. Miller 1/20/09 12:00 AM
Deleted: 83
Mark S. Miller 1/20/09 12:00 AM
Deleted: 84
Mark S. Miller 1/20/09 12:00 AM
Deleted: 84
Mark S. Miller 1/20/09 12:00 AM
Deleted: 84
Mark S. Miller 1/20/09 12:00 AM
Deleted: 85
Mark S. Miller 1/20/09 12:00 AM
Deleted: 85
Mark S. Miller 1/20/09 12:00 AM
Deleted: 87
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Deleted: 94
Mark S. Miller 1/20/09 12:00 AM
Deleted: 95
Mark S. Miller 1/20/09 12:00 AM
Deleted: 95
Mark S. Miller 1/20/09 12:00 AM
Deleted: 97
Mark S. Miller 1/20/09 12:00 AM
Deleted: 101
Mark S. Miller 1/20/09 12:00 AM
Deleted: 102
Mark S. Miller 1/20/09 12:00 AM
Deleted: 102
Mark S. Miller 1/20/09 12:00 AM
Deleted: 102
Mark S. Miller 1/20/09 12:00 AM
Deleted: 102
Mark S. Miller 1/20/09 12:00 AM
Deleted: 102
Mark S. Miller 1/20/09 12:00 AM
Deleted: 105
Mark S. Miller 1/20/09 12:00 AM
Deleted: 106
Mark S. Miller 1/20/09 12:00 AM
Deleted: 106
Mark S. Miller 1/20/09 12:00 AM
Deleted: 106
Mark S. Miller 1/20/09 12:00 AM
Deleted: 106
Mark S. Miller 1/20/09 12:00 AM
Deleted: 107
Mark S. Miller 1/20/09 12:00 AM
Deleted: 108
Mark S. Miller 1/20/09 12:00 AM
Deleted: 109
Mark S. Miller 1/20/09 12:00 AM
Deleted: 110
Mark S. Miller 1/20/09 12:00 AM
Deleted: 110
Mark S. Miller 1/20/09 12:00 AM
Deleted: 110
Mark S. Miller 1/20/09 12:00 AM
Deleted: 111
Mark S. Miller 1/20/09 12:00 AM
Deleted: 111
Mark S. Miller 1/20/09 12:00 AM
Deleted: 126
Mark S. Miller 1/20/09 12:00 AM
Deleted: 127
Mark S. Miller 1/20/09 12:00 AM
Deleted: 127
Mark S. Miller 1/20/09 12:00 AM
Deleted: 127
Mark S. Miller 1/20/09 12:00 AM
Deleted: 127
Mark S. Miller 1/20/09 12:00 AM
Deleted: 128
Mark S. Miller 1/20/09 12:00 AM
Deleted: 136
Mark S. Miller 1/20/09 12:00 AM
Deleted: 136
Mark S. Miller 1/20/09 12:00 AM
Deleted: 136
Mark S. Miller 1/20/09 12:00 AM
Deleted: 136
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Deleted: 152
Mark S. Miller 1/20/09 12:00 AM
Deleted: 152
Mark S. Miller 1/20/09 12:00 AM
Deleted: 153
Mark S. Miller 1/20/09 12:00 AM
Deleted: 154
Mark S. Miller 1/20/09 12:00 AM
Deleted: 161
Mark S. Miller 1/20/09 12:00 AM
Deleted: 161
Mark S. Miller 1/20/09 12:00 AM
Deleted: 161
Mark S. Miller 1/20/09 12:00 AM
Deleted: 163
Mark S. Miller 1/20/09 12:00 AM
Deleted: 175
Mark S. Miller 1/20/09 12:00 AM
Deleted: 175
Mark S. Miller 1/20/09 12:00 AM
Deleted: 175
Mark S. Miller 1/20/09 12:00 AM
Deleted: 176
Mark S. Miller 1/20/09 12:00 AM
Deleted: 177
Mark S. Miller 1/20/09 12:00 AM
Deleted: 177
Mark S. Miller 1/20/09 12:00 AM
Deleted: 177
Mark S. Miller 1/20/09 12:00 AM
Deleted: 177
Mark S. Miller 1/20/09 12:00 AM
Deleted: 178
Mark S. Miller 1/20/09 12:00 AM
Deleted: 178
Mark S. Miller 1/20/09 12:00 AM
Deleted: 178
Mark S. Miller 1/20/09 12:00 AM
Deleted: 178
Mark S. Miller 1/20/09 12:00 AM
Deleted: 179
Mark S. Miller 1/20/09 12:00 AM
Deleted: 180
Mark S. Miller 1/20/09 12:00 AM
Deleted: 181
Mark S. Miller 1/20/09 12:00 AM
Deleted: 182
Mark S. Miller 1/20/09 12:00 AM
Deleted: 186
Mark S. Miller 1/20/09 12:00 AM
Deleted: 187
Mark S. Miller 1/20/09 12:00 AM
Deleted: 187
Mark S. Miller 1/20/09 12:00 AM
Deleted: 193
Mark S. Miller 1/20/09 12:00 AM
Deleted: 194
Mark S. Miller 1/20/09 12:00 AM
Deleted: 198
Mark S. Miller 1/20/09 12:00 AM
Deleted: 200
Mark S. Miller 1/20/09 12:00 AM
Deleted: 201
Mark S. Miller 1/20/09 12:00 AM
Deleted: 202
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM
Mark S. Miller 1/20/09 12:00 AM

Mark S. Miller 1/20/09 12:00 AM
Deleted: 207

Mark S. Miller 1/20/09 12:00 AM
Deleted: 208

Mark S. Miller 1/20/09 12:00 AM
Deleted: 208

Mark S. Miller 1/20/09 12:00 AM
Deleted: 209

Mark S. Miller 1/20/09 12:00 AM
Deleted: 209

Mark S. Miller 1/20/09 12:00 AM
Deleted: 209

Mark S. Miller 1/20/09 12:00 AM
Deleted: 210

Mark S. Miller 1/20/09 12:00 AM
Deleted: 210

Mark S. Miller 1/20/09 12:00 AM
Deleted: 210

Mark S. Miller 1/20/09 12:00 AM
Deleted: 210

Mark S. Miller 1/20/09 12:00 AM
Deleted: 210

Mark S. Miller 1/20/09 12:00 AM
Deleted: 211

Mark S. Miller 1/20/09 12:00 AM
Deleted: 211

Mark S. Miller 1/20/09 12:00 AM
Deleted: 212

Mark S. Miller 1/20/09 12:00 AM
Deleted: 212

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1 Scope

This Standard defines the SES language.

Mark S. Miller 1/19/09 4:07 PM
Deleted: ECMAScript scripting

2 Conformance

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

Mark S. Miller 1/19/09 4:12 PM
Deleted: ECMAScript

A conforming implementation of this International standard shall interpret characters in conformance with the Unicode Standard, Version 3.0 or later and ISO/IEC 10646-1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300. If the adopted encoding form is not otherwise specified, it presumed to be the UTF-16 encoding form.

Mark S. Miller 1/19/09 4:13 PM
Deleted: A conforming implementation of ECMAScript is permitted to provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification.
A conforming implementation of ECMAScript is permitted to support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the "future reserved words" listed in 7.5.3 of this specification.

3 References

ISO/IEC 9899:1996 Programming Languages – C, including amendment 1 and technical corrigenda 1 and 2.

ISO/IEC 10646-1:1993 Information Technology -- Universal Multiple-Octet Coded Character Set (UCS) plus its amendments and corrigenda.

The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: The Unicode Standard, Version 3.0 (Boston, MA, Addison-Wesley, 2000. ISBN 0-201-61635-5).

Unicode Inc. (1998), Unicode Technical Report #15: Unicode Normalization Forms.

ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronic Engineers, New York (1985).

The "Mountain View Draft" of ECMAScript 3.1 15jan2009

4 Overview

This section contains a non-normative overview of the SES language.

Mark S. Miller 1/19/09 4:14 PM
Deleted: ECMAScript

SES is an object-capability programming language for performing computations and manipulating computational objects within a host environment. SES 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 SES program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host* objects, whose description and behaviour 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 SES program.

Mark S. Miller 1/19/09 4:14 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:14 PM
Deleted: oriented

Mark S. Miller 1/19/09 4:15 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:15 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:15 PM
Deleted: ECMAScript

A ***scripting language*** is a programming language that is used to manipulate, customise, 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.

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.

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

- Gosling, James, Bill Joy and Guy Steele. The Java™ Language Specification. Addison Wesley Publishing Co., 1996.
- Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227–241, Orlando, FL, October 1987.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

4.1 Web Scripting

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

A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data. By using browser-side and server-side scripting together, it is possible to distribute computation between the client and server while providing a customised 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 SES —not all parts of the language are described. This overview is not part of the standard proper.

SES is object-capability-based: basic language and host facilities are provided by objects, and an SES program is a cluster of communicating objects. A SES ***object*** is a collection of ***properties*** each with zero or more ***attributes*** that determine how each property can be used—for example, when the Writable attribute for a property is set to **false**, any attempt by executed SES code to change the value of the property fails. 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.

SES defines a collection of ***built-in objects*** that round out the definition of SES entities. These built-in objects include the **Object** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, the **Date** object, the **RegExp** object, the **JSON** object, and the Error objects **Error, EvalError**, **RangeError, ReferenceError, SyntaxError, TypeError** and **URIError**.

SES also defines a set of built-in ***operators***. SES 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.

SES syntax is a subset of the ES3.1-strict syntax, which intentionally resembles Java syntax. SES syntax is relaxed compared to Java, 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 defined functions are not required to have their declarations appear textually before calls to them.

4.2.1 Objects

SES does not contain classes such as those in C++, Smalltalk, or Java, but rather, supports ***object literals*** and ***constructor functions*** which create objects by executing code that allocates storage for the objects and initialises all or part of them by assigning initial values to their properties. For example, if **Point** is a constructor function, then **Point(3, 5)** might create and return a new point object.

SES supports a restricted form of ES3.1 *prototype-based inheritance* which we call *record inheritance*. Every object but one has an implicit reference (called the *object's parent*) to the object it inherits from. Furthermore, a parent object may have a non-null implicit reference to its parent, and so on; this is called the *inheritance 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 inheritance 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 parent of that object is examined next; and so on.

Mark S. Miller 1/19/09 4:16 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:16 PM
Deleted: ECMAScript …ECMAScript n…ECMAScript …ECMAScript ... [1]

Mark S. Miller 1/19/09 4:18 PM
Deleted: ECMAScript …ECMAScript …**Global** object, the …**Function** object, the ... [2]

Mark S. Miller 1/19/09 4:19 PM
Deleted: ECMAScript …ECMAScript ... [3]

Mark S. Miller 1/19/09 4:19 PM
Deleted: ECMAScript …ECMAScript ... [4]

Mark S. Miller 1/19/09 4:21 PM
Deleted: ECMAScript …All constructors are objects, but not all objects are constructors. Each constructor has a property named "`prototype`" that 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. ... [5]

pratapL 1/19/09 8:57 PM
Comment: From AWB: Need to make typography consistent.

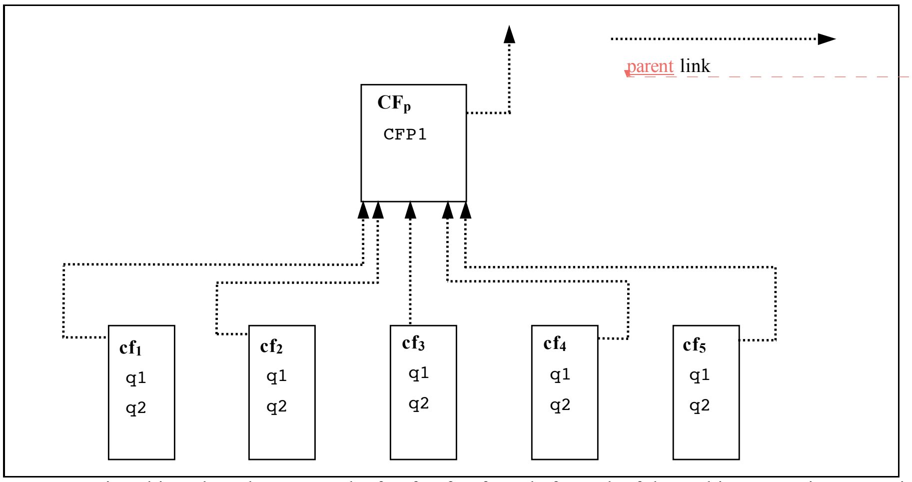
Mark S. Miller 1/19/09 4:23 PM
Deleted: `String("A String")` produces a primitive string, not an object

Mark S. Miller 1/19/09 4:25 PM
Deleted: ECMAScript …created by a constructor *prototype*…value of its constructor's "`prototype`" property…prototype …prototype…*prototype* …prototype prototype for ... [6]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

All objects that do not directly contain a particular property that their parent contains share that property and its value. The following diagram illustrates this:



Five objects have been created: cf_1, cf_2, cf_3, cf_4, and cf_5. Each of these objects contains properties named q1 and q2. The dashed lines represent the parent relationship; so, for example, cf_3's parent is CF_p. 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 CFp's implicit inheritance chain that are not named q1, q2, or CFP1.

Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. In the above diagram, one could add a new shared property for $\mathbf{cf_1}$, $\mathbf{cf_2}$, $\mathbf{cf_3}$, $\mathbf{cf_4}$, and $\mathbf{cf_5}$ by assigning a new value to the property in $\mathbf{CF_p}$.

4.3 Definitions

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

4.3.1 Type

A ***type*** is a set of data values as defined in section 8 of this specification.

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 datum that 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 a collection of properties.

4.3.4 N/A

4.3.5 N/A

4.3.6 Native Object

A ***native object*** is any object supplied by an SES implementation independent of the host environment. Standard native objects are defined in this specification. Some native objects are built-in; others may be constructed during the course of execution of an SES program.

4.3.7 Built-in Object

A ***built-in object*** is any object supplied by an SES implementation, independent of the host environment, which is present at the start of the execution of an SES program. Standard built-in objects are defined in this specification. Every built-in object is a native object. A ***built-in constructor*** is a built-in object that

Mark S. Miller 1/19/09 4:29 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:30 PM
Deleted: prototype

Mark S. Miller 1/19/09 4:30 PM
Deleted: implicit prototype

Mark S. Miller 1/19/09 4:35 PM
Deleted: CF is a constructor (and also an object).

Mark S. Miller 1/19/09 4:35 PM
Deleted: by using **new** expressions

Mark S. Miller 1/19/09 4:35 PM
Deleted: implicit prototype

Mark S. Miller 1/19/09 4:36 PM
Deleted: prototype

Mark S. Miller 1/19/09 4:36 PM
Deleted: 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.

Mark S. Miller 1/19/09 4:36 PM
Deleted: (but not by CF)

Mark S. Miller 1/19/09 4:36 PM
Deleted: prototype

Mark S. Miller 1/19/09 4:37 PM
Deleted: Notice that there is no implicit prototype link between CF and CF_p.

Mark S. Miller 1/19/09 4:37 PM
Deleted: That is, constructors are not required to name or assign values to all or any of the constructed object's properties.

Mark S. Miller 1/19/09 4:39 PM
Deleted: **4.2.2 The Strict Variant of ECMAScript**
The ECMAScript Language recognizes the possibility that some users of the language may wish to restrict their usage of some features availab ... [7]

Mark S. Miller 1/19/09 5:19 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:41 PM
Deleted: **Constructor**

Mark S. Miller 1/19/09 4:41 PM
Deleted: ... [8]

Mark S. Miller 1/19/09 4:41 PM
Deleted: **Prototype**

Mark S. Miller 1/19/09 4:41 PM
Deleted: A ***prototype*** is an object used to ... [9]

Mark S. Miller 1/19/09 4:42 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:42 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:43 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:43 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:44 PM
Deleted: , and an ECMAScript implement ... [10]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

is also a constructor. Objects directly constructed by built-in constructors are ***tamed native objects***. Native objects other than built-ins and tamed natives are ***cajoled objects***.

4.3.8 Tamed Host Object

A ***tamed host object*** is any object supplied by the host environment to complete the execution environment of SES. Any object that is not native is a host object.

4.3.9 Undefined Value

The ***undefined value*** is a primitive value used when a variable has not been assigned a value.

4.3.10 Undefined Type

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

4.3.11 Null Value

The ***null value*** is a primitive value that represents the null, empty, or non-existent reference.

4.3.12 Null Type

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

4.3.13 Boolean Value

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

4.3.14 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**.

4.3.15 N/A

4.3.16 String Value

A ***string value*** is a primitive value that is a member of the type **String** and is a finite ordered sequence of zero or more 16-bit unsigned integer values.

NOTE

Although each value usually represents a single 16-bit unit of UTF-16 text, the language does not place any restrictions or requirements on the values except that they be 16-bit unsigned integers.

4.3.17 String Type

The type **String** is the set of all string values.

4.3.18 N/A

4.3.19 Number Value

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

4.3.20 Number Type

The type **Number** is a set of primitive values representing numbers. In SES, the set of values represents the double-precision 64-bit format IEEE 754 values including the special “Not-a-Number” (NaN) values, positive infinity, and negative infinity.

4.3.21 N/A

4.3.22 Infinity

The primitive value **Infinity** represents the positive infinite number value. This value is a member of the Number type.

4.3.23 NaN

The primitive value **NaN** represents the set of IEEE Standard “Not-a-Number” values. This value is a member of the Number type.

Mark S. Miller 1/19/09 4:45 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:49 PM
Deleted: Boolean Object

Mark S. Miller 1/19/09 4:49 PM
Deleted: A ***Boolean object*** is a member of the type **Object** and is an instance of the built-in Boolean object. 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 that is the boolean. A Boolean object can be coerced to a boolean value.

Mark S. Miller 1/19/09 4:50 PM
Deleted: String Object

Mark S. Miller 1/19/09 4:50 PM
Deleted: A ***String object*** is a member of the type **Object** and is an instance of the built-in String object. 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 that is the string. A String object can be coerced to a string value by calling the String constructor as a function (15.5.1).

Mark S. Miller 1/19/09 5:19 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:50 PM
Deleted: Number Object

Mark S. Miller 1/19/09 4:50 PM
Deleted: A ***Number object*** is a member of the type **Object** and is an instance of the built-in Number object. 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 that is the number. A Number object can be coerced to a number value by calling the Number constructor as a function (15.7.1).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

4.3.24 Function

A ***function*** is a member of the type **Object** that may be invoked as a subroutine. In addition to its named properties, a function contains executable code and state that determine how it behaves when invoked. A function's code may or may not be written in SES.

Mark S. Miller 1/19/09 4:50 PM
Deleted: ECMAScript

4.3.25 Built-in Function

A built-in function is a function that is a built-in object of the language, such as **parseInt** and **Math.exp**.

Mark S. Miller 1/19/09 4:51 PM
Deleted: An implementation may also provide implementation-dependent built-in functions that are not described in this specification.

4.3.26 Property

A ***property*** is an association between a name and a value. Depending upon the form of the property the value may be represented either directly as a data value (a primitive value, an object, or a function) or indirectly by a pair of accessor functions.

4.3.27 Method

A ***method*** is a function that is the value of a property.

4.3.28 Attribute

An ***attribute*** is an internal value that defines some characteristic of a property.

4.3.29 Own Property

An own property of an object is a property that is directly present on that object.

4.3.30 Inherited Property

An ***inherited property*** is a property of an object that is not one of its own properties but is a property (either own or inherited) of the object's parent.

Mark S. Miller 1/19/09 4:52 PM
Deleted: prototype

4.3.31 Built-in Method

A ***built-in method*** is any method that is a built-in function. Standard built-in methods are defined in this specification. A built-in method is a Built-in function.

Mark S. Miller 1/19/09 4:53 PM
Deleted: , and an ECMAScript implementation may specify and define others

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 SES program.

Mark S. Miller 1/19/09 5:19 PM
Deleted: ECMAScript

5.1.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

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

5.1.2 The Lexical and RegExp Grammars

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

Mark S. Miller 1/19/09 4:54 PM
Deleted: ECMAScript

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for SES and are called SES *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the SES 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 (7.9). Simple white space and single-line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that is, a comment of the form "/*…*/" regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a *MultiLineComment* 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.

Mark S. Miller 1/19/09 4:54 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:54 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:54 PM
Deleted: ECMAScript

A *RegExp grammar* for SES is given in 15.10. This grammar also has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Pattern*, that describe how sequences of Unicode characters are translated into regular expression patterns.

Mark S. Miller 1/19/09 4:54 PM
Deleted: ECMAScript

Productions of the lexical and RegExp grammars are distinguished by having two colons "::" as separating punctuation. The lexical and RegExp grammars share some productions.

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 9.3.1.

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

5.1.4 The Syntactic Grammar

The *syntactic grammar* for SES is given in clauses 11, 12, 13 and 14. This grammar has SES tokens defined by the lexical grammar as its terminal symbols (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 SES programs.

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

When a stream of Unicode characters is to be parsed as an SES 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.

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

The syntactic grammar as presented in sections 11, 12, 13 and 14 is actually not a complete account of which token sequences are accepted as correct SES programs. Certain token sequences that are described by the grammar are not considered acceptable if a terminator character appears in certain "awkward" places.

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:26 PM
Deleted: additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain

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. All nonterminal characters specified in this way are to be understood as the appropriate Unicode character from the ASCII range, as opposed to any similar-looking characters from other Unicode ranges.

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:

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

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

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

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

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 $Initialiser_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**
  Identifier
  Identifier Initialiser

and that:

IterationStatement **:**
  `for` **`(`** $ExpressionNoIn_{opt}$ **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*

is a convenient abbreviation for:

IterationStatement **:**
  `for` **`(`** **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*
  `for` **`(`** *ExpressionNoIn* **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
  `for` **`(`** **`;`** **`;`** $Expression_{opt}$ **`)`** *Statement*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

which in turn is an abbreviation for:

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

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

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

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

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

DecimalDigits **::**
DecimalDigit
DecimalDigits DecimalDigit

the definition

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

matches either the letter **n** followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

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

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

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

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 SES contains the production:

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

Mark S. Miller 1/19/09 4:55 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

which is merely a convenient abbreviation for:

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

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multi-character 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

The specification often uses a numbered list to specify steps in an algorithm. These algorithms are used to precisely specify the required semantics of SES language constructs. The algorithms are not intended to imply the use of any specific implementation technique. In practice, there may be more efficient algorithms available to implement a given feature.

Mark S. Miller 1/19/09 4:56 PM
Deleted: ECMAScript

In order to facilitate their use in multiple parts of this specification some algorithms, called *abstract operations*, are named and written in parameterized functional form so that they may be referenced by name from within other algorithms.

When an algorithm is to produce a value as a result, the directive "return *x*" is used to indicate that the result of the algorithm is the value of *x* and that the algorithm should terminate. The notation Result(*n*) is used as shorthand for "the result of step *n*". Type(*x*) is used as shorthand for "the type of *x*".

For clarity of expression, algorithm steps may be subdivided into sequential substeps. Substeps are indented and may themselves be further divided into indented substeps. Outline numbering conventions are used to identify substeps with the first level of substeps labeled with lower case alphabetic characters and the second level of substeps labelled with lower case roman numerals. If more than three levels are required these rules repeat with the fourth level using numeric labels. For example:

1. Top-level step
    a. Substep.
    b. Substep
        i. Subsubstep.
        ii. Subsubstep.
            1. Subsubsubstep
                a. Subsubsubsubstep

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

A step may specify the iterative application of its substeps.

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

The mathematical function 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 sign(x) yields 1 if x is positive and −1 if x is negative. The sign function is not used in this standard for cases when x is zero.

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

The mathematical function floor(x) yields the largest integer (closest to positive infinity) that is not larger than x.

NOTE

floor(x) = x−(x modulo 1).

If an algorithm is defined to "throw an exception", execution of the algorithm is terminated and no result is returned. The calling algorithms are also terminated, until an algorithm step is reached that explicitly deals with the exception, using terminology such as "If an exception was thrown…". Once such an algorithm step has been encountered the exception is no longer considered to have occurred.

6. Source Text

SES source text is represented as a sequence of characters in the Unicode character encoding, version 3.0 or later, using the UTF-16 transformation format. The text must be in Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming SES implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

SourceCharacter **::**
  any Unicode character

Throughout the rest of this document, the phrase "code unit" and the word "character" will be used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of UTF-16 text. The phrase "Unicode character" will be used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code unit). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual "Unicode characters," even though a user might think of the whole sequence as a single character.

In string literals, regular expression literals and identifiers, any character (code unit) may also be expressed as a Unicode escape sequence consisting of six 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 or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

NOTE 1
Although this document sometimes refers to a "transformation" between a "character" within a "string" and the 16-bit unsigned integer that is the UTF-16 encoding of that character, there is actually no transformation because a "character" within a "string" is actually represented using that 16-bit unsigned value.

NOTE 2

Mark S. Miller 1/19/09 4:56 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:28 PM
Deleted: is expected to have been

Mark S. Miller 1/19/09 5:28 PM
Deleted: normalised to

Mark S. Miller 1/19/09 4:56 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

SES differs from the Java programming language in the behaviour 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 SES 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 SES 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.*

Mark S. Miller 1/19/09 4:56 PM
Deleted: *ECMAScript*

Mark S. Miller 1/19/09 4:56 PM
Deleted: *ECMAScript*

Mark S. Miller 1/19/09 4:56 PM
Deleted: *ECMAScript*

7 Lexical Conventions

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

Mark S. Miller 1/19/09 4:56 PM
Deleted: ECMAScript

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

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

Syntax

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

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

7.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category "Cf" in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

The format control characters may be used in identifiers, within comments, and within string literals and regular expression literals.

7.2 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:

Code Unit Value	***Name***	***Formal Name***

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

`\u0009`	Tab	<TAB>
`\u000B`	Vertical Tab	<VT>
`\u000C`	Form Feed	<FF>
`\u0020`	Space	<SP>
`\u0085`	Next Line	<NEL>
`\u00A0`	No-break space	<NBSP>
`\u200B`	Zero width space	<ZWSP>
`\uFEFF`	Byte Order Mark	<BOM>
Other category "Zs"	Any other Unicode "space separator"	<USP>

SES implementations must recognize all of the white space characters defined in Unicode 3.0. Later editions of the Unicode Standard may define other white space characters. SES implementations may recognize white space characters from later editions of the Unicode Standard.

Mark S. Miller 1/19/09 4:56 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

Syntax

WhiteSpace **::**
 <TAB>
 <VT>
 <FF>
 <SP>
 <NEL>
 <NBSP>
 <ZWSP>
 <BOM>
 <USP>

7.3 Line Terminators

Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, except that line terminators that are preceded by an escape sequence may occur within a string literal token. Line terminators also affect the process of automatic semicolon insertion (7.9).

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

The following characters are considered to be line terminators:

Code Unit Value	***Name***	***Formal Name***
`\u000A`	Line Feed	<LF>
`\u000D`	Carriage Return	<CR>

Mark S. Miller 1/19/09 5:31 PM
Deleted: `\u2028` ... [11]

Only the characters in the above table are treated as line terminators. Other new line or line breaking characters are treated as white space but not as line terminators. The character sequence <CR><LF> is commonly used as a line terminator. It should be considered a single character for the purpose of reporting line numbers.

Syntax

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

Mark S. Miller 1/19/09 5:32 PM
Deleted:
<LS>
<PS>

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

LineTerminatorSequence **::**
    <LF>
    <CR> [lookahead ∉ *<LF>*]
    <CR> <LF>

Mark S. Miller 1/19/09 5:32 PM
Deleted:]
<LS>
<PS>

7.4 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 recognised 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 (7.9).

Comments behave like white space and are discarded except that, if a *MultiLineComment* contains a line terminator character, then the entire comment is considered to be a *LineTerminator* for purposes of parsing by the syntactic grammar.

Syntax

Comment **::**
    MultiLineComment
    SingleLineComment

MultiLineComment **::**
    /* *MultiLineCommentChars*$_{opt}$ ***/**

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

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

MultiLineNotAsteriskChar **::**
    SourceCharacter **but not** *asterisk* *****

MultiLineNotForwardSlashOrAsteriskChar **::**
    SourceCharacter **but not** *forward-slash* **/** **or** *asterisk* *****

SingleLineComment **::**
    // *SingleLineCommentChars*$_{opt}$

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

SingleLineCommentChar **::**
    SourceCharacter **but not** *LineTerminator*

7.5 Tokens

Syntax

Token **::**
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

7.5.1 Reserved Words

Description

Reserved words cannot be used as identifiers.

Syntax

ReservedWord **::**
  Keyword
  FutureReservedWord
  NullLiteral
  BooleanLiteral

7.5.2 Keywords

The following tokens are <u>SES</u> keywords and may not be used as identifiers in <u>SES</u> programs.

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

Syntax

Keyword **:: one of**

`break`	`else`	`new`	`var`
`case`	`finally`	`return`	`void`
`catch`	`for`	`switch`	`while`
`continue`	`function`	`this`	
`default`	`if`	`throw`	`debugger`
`delete`	`in`	`try`	<u>`eval`</u>
`do`	`instanceof`	`typeof`	<u>`arguments`</u>

Mark S. Miller 1/19/09 5:33 PM
Deleted: `with`

7.5.3 Reserved Words

The following words are used as keywords in <u>ES3.1 or in</u> proposed extensions and are therefore reserved to <u>avoid conflicts</u>.

Mark S. Miller 1/19/09 5:34 PM
Deleted: Future

Mark S. Miller 1/19/09 5:34 PM
Deleted: allow for the possibility of future adoption of those extensions

Syntax

FutureReservedWord **:: one of**

`abstract`	`enum`	`int`	`short`
`boolean`	`export`	`interface`	`static`
`byte`	`extends`	`long`	`super`
`char`	`final`	`native`	`synchronized`
`class`	`float`	`package`	`throws`
	`goto`	`private`	`transient`
	`implements`	`protected`	`volatile`
`double`	`import`	`public`	
<u>`with`</u>	<u>`yield`</u>	<u>`lambda`</u>	<u>`let`</u>

Mark S. Miller 1/19/09 5:36 PM
Deleted: `const`

pratapL 1/19/09 8:57 PM
Comment: This table needs to be repacked to get rid of the holes.

7.6 Identifiers

Mark S. Miller 1/19/09 5:37 PM
Deleted: Note
The identifiers 'const', 'let', and 'yield' are likely to be used in a future version of this standard.

Description

Identifiers are interpreted according to the grammar given in Section 5.16 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode Standard. The characters in the specified categories in version 3.0 of the Unicode standard must be treated as in those categories by all conforming <u>SES</u> implementations.

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

This standard specifies specific character additions: The dollar sign (**$**) and the underscore (**_**) are permitted anywhere in an identifier<u>, except that an identifier cannot end with two consecutive underscores</u>.

Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier, as computed by the CV of the *UnicodeEscapeSequence* (see 7.8.4). The **** preceding the *UnicodeEscapeSequence* does not contribute a character to the identifier. A *UnicodeEscapeSequence* cannot be used to put a character into an identifier that would otherwise be illegal. In other words, if a **** *UnicodeEscapeSequence* sequence were replaced by its *UnicodeEscapeSequence*'s CV, the result must still be a valid *Identifier* that has the exact same sequence of characters as the original *Identifier*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Two identifiers that are canonically equivalent according to the Unicode standard are *not* equal unless they are represented by the exact same sequence of code units (in other words, conforming SES implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

SES implementations may recognize identifier characters defined in later editions of the Unicode Standard. If portability is a concern, programmers should only employ identifier characters defined in Unicode 3.0.

Mark S. Miller 1/19/09 4:57 PM
Deleted: ECMAScript

Syntax

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

IdentifierName ::
IdentifierStart
IdentifierName IdentifierPart

IdentifierStart ::
UnicodeLetter
`$`
`\` *UnicodeEscapeSequence*

IdentifierPart ::
IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
`\` *UnicodeEscapeSequence*

UnicodeLetter
any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Nl)”.

UnicodeCombiningMark
any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

UnicodeDigit
any character in the Unicode category “Decimal number (Nd)”

UnicodeConnectorPunctuation
any character in the Unicode category “Connector punctuation (Pc)”

UnicodeEscapeSequence
see 7.8.4.

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`

7.7 Punctuators

Syntax

Punctuator **:: one of**

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

= **+=** **-=** ***=** **%=** **<<=**

>>= **>>>=** **&=** **|=** **^=**

DivPunctuator **:: one of**
/ **/=**

7.8 Literals

Syntax

Literal **::**
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral

7.8.1 Null Literals

Syntax

NullLiteral **::**
null

Semantics

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

7.8.2 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**.

7.8.3 Numeric Literals

Syntax

NumericLiteral **::**
DecimalLiteral
HexIntegerLiteral

DecimalLiteral **::**
DecimalIntegerLiteral **.** $DecimalDigits_{opt}$ $ExponentPart_{opt}$
. *DecimalDigits* $ExponentPart_{opt}$
DecimalIntegerLiteral $ExponentPart_{opt}$

DecimalIntegerLiteral **::**
0
NonZeroDigit $DecimalDigits_{opt}$

DecimalDigits **::**
DecimalDigit
DecimalDigits DecimalDigit

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

ExponentPart **::**
ExponentIndicator SignedInteger

ExponentIndicator **:: one of**
e E

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

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

The source character immediately following a *NumericLiteral* must not be an *IdentifierStart* or *DecimalDigit*.

NOTE
For example:

```
3in
```

is an error and not the two input elements ***3*** *and* ***in****.*

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 as described below.

The MV of *NumericLiteral* **::** *DecimalLiteral* is the MV of *DecimalLiteral*.
The MV of *NumericLiteral* **::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
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^{e-n}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
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** is 0.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The MV of *DecimalDigit* **::** `1` or of *NonZeroDigit* **::** `1` or of *HexDigit* **::** `1` is 1.
The MV of *DecimalDigit* **::** `2` or of *NonZeroDigit* **::** `2` or of *HexDigit* **::** `2` is 2.
The MV of *DecimalDigit* **::** `3` or of *NonZeroDigit* **::** `3` or of *HexDigit* **::** `3` is 3.
The MV of *DecimalDigit* **::** `4` or of *NonZeroDigit* **::** `4` or of *HexDigit* **::** `4` is 4.
The MV of *DecimalDigit* **::** `5` or of *NonZeroDigit* **::** `5` or of *HexDigit* **::** `5` is 5.
The MV of *DecimalDigit* **::** `6` or of *NonZeroDigit* **::** `6` or of *HexDigit* **::** `6` is 6.
The MV of *DecimalDigit* **::** `7` or of *NonZeroDigit* **::** `7` or of *HexDigit* **::** `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*.

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 8.5), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a `0` digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a `0` digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an *ExponentPart* and

it is not `0`; or

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

7.8.4 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. All Unicode characters may appear literally in a string literal except for the closing quote character, backslash, carriage return, line separator, paragraph separator, and line feed. Any character may appear in the form of an escape sequence.

Syntax

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

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

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

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

LineContinuation

LineContinuation **::**
`\` *LineTerminatorSequence*

EscapeSequence **::**
CharacterEscapeSequence
`0` [lookahead ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **::**
SingleEscapeCharacter
NonEscapeCharacter

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

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

EscapeCharacter **::**
SingleEscapeCharacter
DecimalDigit
`x`
`u`

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

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

The definitions of the nonterminal *HexDigit* is given in section 7.8.3. *SourceCharacter* is described in sections 2 and 6.

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 7.8.3.

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 SV of *LineContinuation* **::** `\` *LineTerminator* is the empty character sequence.

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*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The CV of *EscapeSequence* **::** **`0`** [lookahead ∉ *DecimalDigit*]is a <NUL> character (Unicode value 0000).

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 character whose code unit value is determined by the *SingleEscapeCharacter* according to the following table:

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

The CV of *CharacterEscapeSequence* **::** *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.

The CV of *NonEscapeCharacter* **::** *SourceCharacter* **but not** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.

The CV of *HexEscapeSequence* **::** **`x`** *HexDigit HexDigit* is the character whose code unit value is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.

The CV of *UnicodeEscapeSequence* **::** **`u`** *HexDigit HexDigit HexDigit HexDigit* is the character whose code unit value is (4096 (that is, 16^3) times the MV of the first *HexDigit*) plus (256 (that is, 16^2) times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

NOTE
A line terminator character cannot appear in a string literal, except when preceded by a backslash **`\`** *as a 'LineContinuation' to produce the empty character sequence. 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`**.

8 Types

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

An SES language type corresponds to values that are directly manipulated by an SES programmer using the SES language. The SES language types are Undefined, Null, Boolean, String, Number, and Object.

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

8.1 The Undefined Type

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

8.2 The Null Type

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

Mark S. Miller 1/19/09 11:05 PM
Deleted: 7.8.5 Regular Expression Literals
A regular expression literal is an input element that is converted to a RegExp object (section 15.10) each time the literal is evaluated. Two regular expression literals in a program evaluate to regular expression objects that never compare as **`===`** to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by **`new RegExp`** (section 15.10.4) or calling the **`RegExp`** constructor as a function (section 15.10.3).
The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it should not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions or the productions used by these productions.
Syntax
RegularExpressionLiteral **::**
`/` *RegularExpressionBody* **`/`** *RegularExpressionFlags*
RegularExpressionBody **::**
RegularExpressionFirstChar RegularExpressionChars
RegularExpressionChars **::**
[empty]
RegularExpressionChars RegularExpressionChar
RegularExpressionFirstChar **::**
NonTerminator **but not** **`*`** **or** **`\`** **or** **`/`** **or** **`[`**
BackslashSequence
RegularExpressionClass
RegularExpressionChar **::**
NonTerminator **but not** **`\`** **or** **`/`** **or** **`[`**
BackslashSequence
RegularExpressionClass
BackslashSequence **::**
`\` *NonTerminator*
NonTerminator **::**
SourceCharacter **but not** *LineTerminator*
RegularExpressionClass **::**
`[` *RegularExpressionClassPreamble RegularExpressionClassChars* **`]`**
RegularExpressionClassPreamble **::**
[empty]
... [12]

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 4:58 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

8.3 The Boolean Type

The Boolean type represents a logical entity having two values, called **true** and **false**.

8.4 The String Type

The String type is the set of all finite ordered sequences of zero or more 16-bit unsigned integer values ("elements"). The String type is generally used to represent textual data in a running SES program, in which case each element in the string is treated as a code unit value (see section 6). Each element is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at position 0, the next element (if any) at position 1, and so on. The length of a string is the number of elements (i.e., 16-bit values) within it. The empty string has length zero and therefore contains no elements.

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

When a string contains actual textual data, each element is considered to be a single UTF-16 unit. Whether or not this is the actual storage format of a String, the characters within a String are numbered as though they were represented using UTF-16. All operations on Strings (except as otherwise stated) treat them as sequences of undifferentiated 16-bit unsigned integers; they do not ensure the resulting string is in normalised form, nor do they ensure language-sensitive results.

NOTE
The rationale behind these decisions was to keep the implementation of Strings as simple and high-performing as possible. The intent is that textual data coming into the execution environment from outside (e.g., user input, text read from a file or received over the network, etc.) be converted to Unicode Normalised Form C before the running program sees it. Usually this would occur at the same time incoming text is converted from its original character encoding to Unicode (and would impose no additional overhead). Since it is required that SES source code be in Normalised Form C, string literals are guaranteed to be normalised, as long as they do not contain any Unicode escape sequences.

Mark S. Miller 1/19/09 5:54 PM
Deleted: recommended

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:54 PM
Deleted: (if source text is guaranteed to be normalised)

8.5 The Number Type

The Number type has exactly 18437736874454810627 (that is, $2^{64}-2^{53}+3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the 9007199254740990 (that is, $2^{53}-2$) distinct "Not-a-Number" values of the IEEE Standard are represented in SES as a single special **NaN** value. (Note that the **NaN** value is produced by the program expression **`NaN`**.) In some implementations, external code might be able to detect a difference between various Non-a-Number values, but such behaviour is implementation-dependent; to SES code, all NaN values are indistinguishable from each other.

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

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 **+∞** and **−∞**, respectively. (Note that these two infinite number values are produced by the program expressions **`+Infinity`** (or simply **`Infinity`**) and **`-Infinity`**.)

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

Note that there is both a **positive zero** and a **negative zero**. 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 18437736874454810622 (that is, $2^{64}-2^{53}-2$) finite nonzero values are of two kinds:

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

$$s \times m \times 2^{e}$$

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

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

$$s \times m \times 2^{e}$$

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

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

Some SES 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.

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

8.6 The Object Type

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

- A *named data property* associates a name with a value and a set of boolean attributes.
- A *named accessor property* associates a name with a get method, a set method, and a set of boolean attributes.
- An *internal property* has no name and is not directly accessible via the property accessor operators. Internal properties exist purely for specification purposes. How and when internal properties are used is specified by the language specification below.

There are two types of access for normal (non-internal) properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

8.6.1 Property Attributes

Attributes are used in this specification to define and explain the state of named properties. A named data property associates a name with the following attributes:

Table 1 Attributes of a Named Data Property

Attribute Name	*Value Domain*	*Description*
[[Value]]	Any SES language type	The value retrieved by reading the property.
[[Writable]]	Boolean	If **false**, attempts by SES code to assign the property’s value will fail.
[[Enumerable]]	Boolean	If **true**, the property will be enumerated by a for-in enumeration (section 12.6.4). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If **false**, attempts to delete the property, change the property to be an accessor property, or change its attributes will fail.

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:56 PM
Deleted: not succeed

A named accessor property associates a name with the following attributes:

Table 2 Attributes of a Named Accessor Property

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Attribute Name	***Value Type Domain***	***Description***
[[Get]]	Object *or* Undefined	If the value is an Object it must be a function. The function is called with no-arguments to return the property value each time the property is read.
[[Set]]	Object *or* Undefined	If the value is an Object it must be a function. The function is called with the assigned value as its sole argument each time the property is assigned. The effect of a property's [[Set]] method may, but it not required to, have an effect on the value returned by subsequent calls to the property's [[Get]] function.
[[Enumerable]]	Boolean	If **true**, the property is to be enumerated by a for-in enumeration (section 12.6.4). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If **false**, attempts to delete the property, change the property to be a data property, or change its attributes will fail.

If the value of an attribute is not explicitly specified for a named property, the default value as defined in the following table is used:

Table 3 Default Attribute Values

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

8.6.2 Object Internal Properties and Methods

This specification uses various internal properties and methods to define the semantics of object values. These internal properties and methods are not part of the SES language. They are defined by this specification purely for expository purposes. An implementation of SES must behave as if it produced and operated upon internal properties in the manner described here. For the purposes of this document, the names of internal properties are enclosed in double square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a **TypeError** exception is thrown.

The following table summarises the internal properties used by this specification that are applicable to all SES objects. The description indicates their behaviour for native SES objects, unless stated otherwise in this document for particular types of SES objects. In particular, Array objects have a slightly different definition of the [[ThrowingPut]] method (see 15.4.5.1) and String objects have a different definition of the [[GetOwnProperty]] method. Host objects may support these internal properties with any implementation-dependent behaviour, or it may be that a host object supports only some internal properties and not others.

The "Value Type Domain" column of the following tables define the types of values associated with internal properties. The type names refer to the types defined in section 8 augmented by the following additional names. "*any*" means the value may be any SES language type. "*primitive*" means Undefined, Null, Boolean, String, or Number. "*SpecOp*" means the internal property is an implementation provided procedure defined by an abstract operation specification. "SpecOp" is followed by a list of descriptive parameter names. If a parameter name is the same as a type name then the name describes the type of the

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:00 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

parameter. If a "SpecOp" returns a value its parameter list is followed by the symbol "→" and the type of the returned value.

Table 4 Internal Properties Common to All Objects

Internal Property	*Value Type Domain*	*Description*
[[Parent]]	Object *or* Null	The parent of this object.
[[Class]]	String	A string value indicating a specification defined classification of objects.
[[PrimitiveValue]]	primitive	Internal state information associated with this object.
[[Extensible]]	Boolean	If **true**, own properties may be added to the object.
[[Get]]	SpecOp(*propertyName*) → any	Returns the value of the named property.
[[GetOwnProperty]]	SpecOp (*propertyName*) → Undefined *or* Property Descriptor	Returns the Property Descriptor of the named own property of this object, or **undefined** if absent.
[[GetProperty]]	SpecOp (*propertyName*) → Undefined *or* Property Descriptor	Returns the fully populated Property Descriptor of the named property of this object, or **undefined** if absent.
[[Put]]	SpecOp (*propertyName, any*)	Sets the specified named property to the value of the second parameter.
[[CanPut]]	SpecOp (*propertyName*) → *Boolean*	Returns a Boolean value indicating whether a [[Put]] operation with *PropertyName* can be performed.
[[HasProperty]]	SpecOp (*propertyName*) → *boolean*	Returns a Boolean value indicating whether the object already has a property with the given name.
[[Delete]]	SpecOp (*PropertyName*)	Removes the specified named own property from the object.
[[DefaultValue]]	SpecOp (*Hint*) → *primitive*	Hint is a string. Returns a default value for the object.
[[DefineOwnProperty]]	SpecOp (*propertyName, PropertyDescriptor*)	Creates or alters the named own property to have the state described by a Property Descriptor.
[[ThrowingPut]]	SpecOp (*propertyName, any*)	Sets the specified named property to the value of the second parameter.

All SES objects have an internal property called [[Parent]]. The value of this property is either **null** or an object and is used for implementing inheritance. Named data properties of the [[Parent]] object are inherited (are visible as properties of the child object) for the purposes of get access, but not for put access. Named accessor properties are inherited for both get access and put access.

Every object (including host objects) must implement the [[Parent]], [[Class]], and [[Extensible]] internal data properties and the [[Get]], [[GetProperty]], [[GetOwnProperty]], [[DefineOwnProperty]], [[Put]], [[CanPut]], [[HasProperty]], [[Delete]], and [[DefaultValue]] internal methods. (Note, however, that the [[DefaultValue]] method may, for some objects, simply throw a **TypeError** exception.)

The value of the [[Parent]] property must be either an object or **null**, and every [[Parent]] chain must have finite length (that is, starting from any object, recursively accessing the [[Parent]] property must eventually lead to a **null** value). Whether or not a native object can have a host object as its [[Parent]] depends on the implementation.

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 String value, even a value used by a built-in object for its [[Class]] property. The value of a [[Class]] property is used internally to distinguish different kinds of built-in objects. Note that this specification does not provide any means for a program to access that value except through `Object.prototype.toString` (see 15.2.4.2).

For most native objects the common internal methods behave as described in described in 8.12, except that Array objects have a slightly different implementation of the [[ThrowingPut]] method (see 15.4.5.1) and String objects have a slightly different implementation of the [[GetOwnProperty]] method (see 15.5.5.2). Host objects may implement these methods in any manner unless specified otherwise; for

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: prototype

Mark S. Miller 1/19/09 6:02 PM
Deleted: , *Boolean*

Mark S. Miller 1/19/09 6:03 PM
Deleted: → *Boolean*

Mark S. Miller 1/19/09 6:03 PM
Deleted: . The flag controls failure handling

Mark S. Miller 1/19/09 6:04 PM
Deleted: The flag controls failure handling.

Mark S. Miller 1/19/09 6:04 PM
Deleted: , *Boolean*

Mark S. Miller 1/19/09 6:04 PM
Deleted: , *Boolean*

Mark S. Miller 1/19/09 6:04 PM
Deleted: The flag controls failure handling.

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 8:57 PM
Comment: Need to restrict, for ES3.1 as well

Mark S. Miller 1/19/09 6:09 PM
Deleted: Arguments objects (10.5) have different implementations of [[Get]], [[ThrowingPut]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]].

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

example, one possibility is that [[Get]] and [[Put]] for a particular host object indeed fetch and store property values but [[HasProperty]] always generates **false**.

Mark S. Miller 1/19/09 8:57 PM
Comment: Need to restrict for SES. Taming must enforce these restrictions

Table 5 Internal Properties Only Defined for Some Objects

Internal Property	*Value Type Domain*	*Description*
[[Construct]]	SpecOp(a List of any) → Object	Constructs an object. Invoked via the **new** operator. The arguments to the SpecOp are the arguments passed to the **new** operator. Objects that implement this internal method are called *constructors*.
[[Call]]	SpecOp(a List of any) → any *or* Reference	Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are the arguments passed to the function call expression. Objects that implement this internal method are *functions*. Only functions that are host objects may return Reference values.
[[HasInstance]]	SpecOp(*any*) → Boolean	Returns a Boolean value indicating whether the argument is an Object that delegates behaviour to this object. Of the standard built-in SES objects, only Objects that are instances of the standard built-in constructor Function implement [[HasInstance]].
[[Scope]]	Lexical Environment	A lexical environment that defines the environment in which a Function object is executed. Of the standard built-in SES objects, only objects that are instances of the standard built-in constructor Function implement [[Scope]].
[[FormalParameters]]	List of Strings	A possibly empty List containing the identifier strings of a Function's *FormalParameterList*. Of the standard built-in SES objects, only Objects that are instances of the standard built-in constructor Function implement [[FormalParameterList]].
[[Code]]	SES code	The SES code of a function. Of the standard built-in SES objects, only Objects that are instances of the standard built-in constructor Function implement [[Code]].
[[TargetFunction]]	Object	The target function of a function object created using the standard built-in Function.prototype.bind method. Only SES objects that are bound using Function.prototype.bind have a [[TargetFunction]] internal property.
[[BoundThis]]	*any*	The pre-bound this value of a function Object created using the standard built-in Function.prototype.bind method. Only SES objects that are bound using Function.prototype.bind have a [[BoundThis]] internal property.
[[BoundArguments]]	List of *any*	The pre-bound argument values of a function Object created using the standard built-in Function.prototype.bind method. Only SES objects bound using Function.prototype.bind have a [[BoundArguments]] internal property.
[[Match]]	SpecOp(*string*, *index*) → *MatchResult*	Tests for a regular expression match and returns a MatchResult value (see section 15.10.2.1). Of the standard built-in SES objects only objects that are instances of the standard built-in constructor RegExp implement [[Match]].

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 6:11 PM
Deleted: [[IsArray]] ... [13]

8.7 The Reference Specification Type

Mark S. Miller 1/19/09 8:57 PM
Comment: It would be good to get rid of this completely for SES.

The Reference type is used to explain the behaviour of such operators as **`delete`**, **`typeof`**, and the assignment operators. For example, the left-hand operand of an assignment is expected to produce a reference. The behaviour of assignment could, instead, be explained entirely in terms of a case analysis on

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

the syntactic form of the left-hand operand of an assignment operator. (A possible reason not to use a syntactic case analysis is that it would be lengthy and awkward, affecting many parts of the specification.)

A **Reference** is a reference to a resolved name binding. A Reference consists of two components, the *base* value and the *referenced name*. The base value is either null, an Object, a Boolean, a String, a Number, or an environment record (10.2.1). A base value of null indicates that the reference could not be resolved to a binding. The referenced name is a String.

The following abstract operations are used in this specification to access the components of references:

- GetBase(V). Returns the base value component of the reference V.
- GetReferencedName(V). Returns the referenced name component of the reference V.
- HasPrimitiveBase(V). Returns **true** if the base value is a Boolean, String, or Number.
- IsPropertyReference(V). Returns **true** if the base value is an object and **false** if the base value is an environment record.
- IsUnresolvableReference(V). Returns **true** if the base value is **null** and **false** otherwise.

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

8.7.1 GetValue (V)

1. If Type(*V*) is not Reference, return *V*.
2. Let *base* be the result of calling GetBase(*V*).
3. If UnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsPropertyReference(*V*), then
   a. If HasPrimitiveBase(*V*), then let base be ToObject(*base*) (9.9)
   b. Return the result of calling the [[Get]] method of *base*, passing GetReferencedName(*V*) for the argument.
5. Else, *base* must be an environment record.
   a. Return the result of calling the GetBindingValue(*N*, *S*) concrete method of Result(2) passing GetReferencedName(*V*) and IsStrictReference(*V*) as arguments.

NOTE
The object that may be created in step 4a is immediately discarded after its use in that step. An implementation might choose to avoid the actual creation of the object.

8.7.2 PutValue (V, W)

1. If Type(*V*) is not Reference, throw a **ReferenceError** exception.
2. Let *base* be the result of calling GetBase(*V*).
3. If UnresolvableReference(*V*), then throw a **ReferenceError** exception.
4. Else if IsPropertyReference(*V*), then
   a. If HasPrimitiveBase(*V*) is false, then let *put* be the [[ThrowingPut]] method of *base*, otherwise let *put* be the special [[ThrowingPut]] method defined below.
   b. Call the *put* method using *base* as its **this** object, and passing GetReferencedName(*V*) for the property name and *W* for the value.
5. Else *base* must be a reference whose base is an environment record. So,
   a. Call the SetMutableBinding(*N*, *V*) concrete method of *base*, passing GetReferencedName(*V*) for N and *W* for V.
6. Return.

The following [[ThrowingPut]] internal method is used by PutValue when when *V* is a property reference with a primitive base value. It is called using *Base* as its **this** value and with property *P* and value *V*. The following steps are taken:

1. Throw a **TypeError** exception.

8.8 The List Specification Type

The List type is used to explain the evaluation of argument lists (see 11.2.4) in **`new`** expressions, in function calls, and in other algorithms where a simple list of values is needed. Values of the List type are simply ordered sequences of values. These sequences may be of any length.

8.9 The Completion Specification Type

The Completion type is used to explain the behaviour of statements (**`break`**, **`continue`**, **`return`** and **`throw`**) that perform nonlocal transfers of control. Values of the Completion type are triples of the form

Mark S. Miller 1/19/09 6:13 PM
Deleted: , 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

Mark S. Miller 1/19/09 6:14 PM
Deleted: Another

Mark S. Miller 1/19/09 6:14 PM
Deleted: three

Mark S. Miller 1/19/09 6:14 PM
Deleted: ,

Mark S. Miller 1/19/09 6:14 PM
Deleted: and the Boolean valued *strict reference* flag

Mark S. Miller 1/19/09 6:15 PM
Deleted: <#>IsStrictReference(V). Returns the strict reference component of the reference V.

Mark S. Miller 1/19/09 6:17 PM
Deleted:
If IsStrictReference(*V*) is **true**, then

Mark S. Miller 1/19/09 6:17 PM
Deleted: <#>Call the [[ThrowingPut]] method for the global object, passing GetReferencedName(*V*) for the property name, *W* for the value, and **false** for the *Throw* flag.

Mark S. Miller 1/19/09 6:18 PM
Deleted: ,

Mark S. Miller 1/19/09 6:18 PM
Deleted: , and IsStrictReference(*V*) for the *Throw* flag

Mark S. Miller 1/19/09 6:19 PM
Deleted: , *S*

Mark S. Miller 1/19/09 6:19 PM
Deleted: ,

Mark S. Miller 1/19/09 6:19 PM
Deleted: , and IsStrictReference(*V*) for *S*

Mark S. Miller 1/19/09 6:20 PM
Deleted: ,

Mark S. Miller 1/19/09 6:21 PM
Deleted: , and boolean flag *Throw* as arguments

Mark S. Miller 1/19/09 6:25 PM
Deleted: <#>Let *O* be ToObject(*Base*).
<#>If the result of calling the [[CanPut]] internal method of *O* with argument *P* is **false**, then
<#>If *Throw* is **true**, then throw a **TypeError** exception.
<#>Else return.
<#>Let *ownDesc* be the result of calling the [[GetOwnProperty]] method of *O* with argument *P*.
<#>If IsDataDescriptor(*ownDesc*) is **true**, then
<#>If *Throw* is **true**, then throw a **TypeError** exception.
<#>Else Return.
<#>Let *desc* be the result of calling the [[GetProperty]] method of *O* with argument *P*. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
<#>If IsAccessorDescriptor(*desc*) is **true**, then ... [14]

Mark S. Miller 1/19/09 6:25 PM
Deleted: <#>Return.
NOTE ... [15]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

(*type*, *value*, *target*), where *type* is one of **normal**, **break**, **continue**, **return**, or **throw**, *value* is any SES language value or **empty**, and *target* is any SES identifier or **empty**.

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

The term "abrupt completion" refers to any completion with a type other than **normal**.

8.10 The Property Descriptor and Property Identifier Specification Types

The Property Descriptor type is used to explain the manipulation and reification of named property attributes. Values of the Property Descriptor type are records composed of named fields where each field's name is an attribute name and its value is a corresponding attribute value. In addition, any field may be present or absent.

Property Descriptor values may be further classified as data property descriptors and accessor property descriptors based upon the existence or use of certain fields. A data property descriptor is one that includes any fields named either [[Value]], or [[Writable]]. An accessor property descriptor is one that includes any fields named either [[Get]], or [[Set]]. Any property descriptor may have fields named [[Enumerable]], and [[Configurable]]. A Property Descriptor value may not be both a data property descriptor and an accessor property descriptor however it may be neither. A generic property descriptor is a Property Descriptor value that is neither a data property descriptor nor an accessor property descriptor.

For notational convenience within this specification, an object literal-like syntax can be used to define a property descriptor value. For example, Property Descriptor {value: 42, writable: false, configurable: true} defines a data property descriptor. The order of listing field names is not significant. Any fields that are not explicitly listed are considered to be absent.

In specification text and algorithms, dot notation may be used to refer to a specific field of a Property Descriptor. For example, if D is a property descriptor then D.[[Value]] is short hand for "the field of D named *value*".

The Property Identifier type is used to associate a property name with a Property Descriptor. Values of the Property Identifier type are pairs of the form (name, descriptor), where name is a string and descriptor is a Property Descriptor value.

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

8.10.1 IsAccessorDescriptor (Desc)

When the abstract operation IsAccessorDescriptor is called with property descriptor *Desc* the following steps are taken:

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

8.10.2 IsDataDescriptor (Desc)

When the abstract operation IsDataDescriptor is called with property descriptor *Desc* the following steps are taken:

1. If *Desc* is **undefined**, then return **false**.
2. If both *Desc*.[[Value]] and *Desc*.[[Writable]] are absent, then return **false**.
3. Return **true**.

8.10.3 IsGenericDescriptor (Desc)

When the abstract operation IsGenericDescriptor is called with property descriptor *Desc* the following steps are taken:

1. If *Desc* is **undefined**, then return **false**.
2. If IsAccessorDescriptor(*Desc*) and IsDataDescriptor(*Desc*) are both **false**, then return **true**.
3. Return **false**.

8.10.4 FromPropertyDescriptor (Desc)

When the abstract operation FromPropertyDescriptor is called with property descriptor *Desc* the following steps are taken:, the following steps are taken:

The following algorithm assumes that *Desc* is a fully populated Property Descriptor, such as that returned from [[GetOwnProperty]].

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. If *Desc* is **undefined**, then return **undefined**.
2. Let *obj* be the result of creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
3. If IsDataDescriptor(*Desc*) is **true**, then
   a. Call the [[Put]] method of *obj* with arguments "value" and Desc.[[Value]].
   b. Call the [[Put]] method of *obj* with arguments "writable" and *Desc*.[[Writable]].
4. Else, IsAccessorDescriptor(*Desc*) must be **true**, so
   a. Call the [[Put]] method of *obj* with arguments "get" and Desc.[[Get]].
   b. Call the [[Put]] method of *obj* with arguments "set" and Desc.[[Set]].
5. Call the [[Put]] method of *obj* with arguments "enumerable" and *Desc*.[[Enumerable]].
6. Call the [[Put]] method of *obj* with arguments "configurable" and *Desc*.[[Configurable]].
7. Return *obj*.

8.10.5 ToPropertyDescriptor (Obj)

When the abstract operation ToPropertyDescriptor is called with object *Desc*, the following steps are taken:

1. If Type(*Obj*) is not Object throw a **TypeError** exception.
2. Let *desc* be the result of creating a new Property Descriptor that initially has no fields.
3. If the result of calling the [[HasProperty]] method of *Obj* with argument "enumerable" is **true**, then
   a. Let *enum* be the result of calling the [[Get]] method of *Obj* with "enumerable".
   b. Set the [[Enumerable]] field of *desc* to ToBoolean(*enum*).
4. If the result of calling the [[HasProperty]] method of *Obj* with argument "configurable" is **true**, then
   a. Let *conf* be the result of calling the [[Get]] method of *Obj* with argument "configurable".
   b. Set the [[Configurable]] field of *desc* to ToBoolean(*conf*).
5. If the result of calling the [[HasProperty]] method of *Obj* with argument "value" is **true**, then
   a. Let *value* be the result of calling the [[Get]] method of *Obj* with argument "value".
   b. Set the [[Value]] field of *desc* to value.
6. If the result of calling the [[HasProperty]] method of *Obj* with argument "writable" is **true**, then
   a. Let *writable* be the result of calling the [[Get]] method of *Obj* with argument "writable".
   b. Set the [[Writable]] field of *desc* to ToBoolean(*writable*).
7. If the result of calling the [[HasProperty]] method of *Obj* with argument "get" is **true**, then
   a. Let *getter* be the result of calling the [[Get]] method of *Obj* with argument "get".
   b. If IsCallable(*getter*) is **false** and *getter* is not **undefined**, then throw a **TypeError** exception.
   c. Set the [[Get]] field of *desc* to *getter*.
8. If the result of calling the [[HasProperty]] method of *Obj* with argument "set" is **true**, then
   a. Let *setter* be the result of calling the [[Get]] method of *Obj* with argument "set".
   b. If IsCallable(*setter*) is **false** and *setter* is not **undefined**, then throw a **TypeError** exception.
   c. Set the [[Set]] field of *desc* to *setter*.
9. If either *desc*.[[Get]] or *desc*.[[Set]] are present, then
   a. If either *desc*.[[Value]] or *desc*.[[Writable]] are present, then throw a **TypeError** exception.
10. Return *desc*.

8.11 The Lexical Environment and Environment Record Specification Types

The Lexical Environment and Environment Record types are used to explain the behaviour of name resolution in nested functions and blocks. These types and the operations upon them are defined in section 10.

8.12 Algorithms for Object Internal Methods

In the following algorithm descriptions, assume *O* is a native SES object, *P* is a string, and *Desc* is a Property Description record.

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 6:55 PM
Deleted: , and *Throw* is a Boolean flag

8.12.1 [[GetOwnProperty]] (P)

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

1. If *O* doesn't have an own property with name *P*, return **undefined**.
2. Let *D* be a newly created Property Descriptor (Section 8.10) with no fields.
3. Let *X* be *O*'s own property named *P*.
4. If *X* is a data property, then

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

    a. Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.
    b. Set *D*.[[Writable]] to the value of *X*'s [[Writable]] attribute
5. Else *X* is an accessor property, so
    a. Set *D*.[[Get]] to the value of *X*'s [[Get]] attribute.
    b. Set *D*.[[Set]] to the value of *X*'s [[Set]] attribute.
6. Set *D*.[[Enumerable]] to the value of *X*'s [[Enumerable]] attribute.
7. Set *D*.[[Configurable]] to the value of *X*'s [[Configurable]] attribute.
8. Return *D*.

Note, however, that if *O* is a String object it has a more elaborate [[GetOwnProperty]] method (15.5.5.2).

8.12.2 [[GetProperty]] (P)

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

1. Let *prop* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. If *prop* is not **undefined**, return Result(1).
3. If the [[Parent]] internal property of *O* is **null**, return **undefined**.
4. Call the [[GetProperty]] internal method of [[Parent]] with property name *P*.
5. Return Result(4).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

8.12.3 [[Get]] (P)

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

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
6. Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and providing no arguments.

8.12.4 [[CanPut]] (P)

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

1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
2. If *desc* is not **undefined**, then
    a. If IsAccessorDescriptor(*desc*) is **true**, then
        i. If *desc*.[[Set]] is undefined, then return **false**.
        ii. Else return **true**.
    b. Else, *desc* must be a DataDescriptor so return the value of *desc*.[[Writable]].
3. Let *proto* be the internal [[Parent]] internal property of *O*.
4. If *proto* is **null**, then return the value of the [[Extensible]] internal property of *O*.
5. Let *inherited* be the result of calling the [[GetProperty]] internal method of *proto* with property name *P*.
6. If *inherited* is **undefined**, return the value of the [[Extensible]] internal property of *O*.
7. If IsAccessorDescriptor(*inherited*) is **true**, then
    a. If *inherited*.[[Set]] is **undefined**, then return **false**.
    b. Else return **true**.
8. Else, *inherited* must be a DataDescriptor
    a. If the [[Extensible]] internal property of *O* is **false**, return **false**.
    b. Else return the value of *inherited*.[[Writable]].

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

NOTE
Host objects may define additional constraints upon [[Put]] operations. If possible, host objects should not allow [[Put]] operations in situations where this definition of [[CanPut]] returns false.

8.12.5 [[ThrowingPut]] (P, V)

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

Mark S. Miller 1/19/09 6:28 PM
Deleted: , **Throw**

Mark S. Miller 1/19/09 6:28 PM
Deleted: ,

Mark S. Miller 1/19/09 6:29 PM
Deleted: , and boolean flag *Throw*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. If the result of calling the [[CanPut]] internal method of *O* with argument *P* is false, then throw a **TypeError** exception.
2. Let *ownDesc* be the result of calling the [[GetOwnProperty]] method of *O* with argument *P*.
3. If IsDataDescriptor(*ownDesc*) is **true**, then
   a. Set the [[Value]] attribute of property *P* of *O* to *V*.
   b. Return.
4. Let *desc* be the result of calling the [[GetProperty]] method of *O* with argument *P*. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
5. If IsAccessorDescriptor(*desc*) is **true**, then
   c. Let *setter* be *desc*.[[Set]] which cannot be **undefined**.
   d. Call the [[Call]] method of *setter* providing *O* as the this value and providing *V* as the sole argument.
6. Else, create a named data property named *P* on object *O* whose attributes are:
   e. [[Value]]: V,
   f. [[Writable]]: true,
   g. [[Enumerable]]: true,
   h. [[Configurable]]: true.
7. Return.

Note, however, that if *O* is an Array object, it has a more elaborate [[ThrowingPut]] method (15.4.5.1).

8.12.6 [[Put]] (P, V)

[[Put]] is primarily used in the specification of built-in methods. Algorithms that require explicit control over the handling of invalid property stores should call [[ThrowingPut]] directly.

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

1. Call the [[ThrowingPut]] internal method of *O* with arguments *P* and *V*.
2. Return.

8.12.7 [[HasProperty]] (P)

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

Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
If *desc* is **undefined**, then return **false**.
Else return **true**.

8.12.8 [[Delete]] (P)

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

1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, then return **true**.
3. If *desc*.[[Configurable]] is **true**, then remove the own property with name P from O.
4. Else throw a **TypeError** exception.

8.12.9 [[DefaultValue]] (hint)

When the [[DefaultValue]] internal method of *O* is called with hint String, the following steps are taken:

1. Let *toString* be the result of calling the [[Get]] internal method of object *O* with argument "toString".
2. If *toString* is an object then,
3. Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as the this value and an empty argument list.
4. If *str* is a primitive value, return *str*.
5. Let *valueOf* be the result of calling the [[Get]] internal method of object *O* with argument "valueOf".
6. If *valueOf* is an object then,
7. Let *val* be the result of calling the [[Call]] internal method of *valueOf*, with *O* as the this value and an empty argument list.
8. If *val* is a primitive value, return *val*.
9. Throw a **TypeError** exception.

When the [[DefaultValue]] method of *O* is called with hint Number, the following steps are taken:

Mark S. Miller 1/19/09 6:29 PM
Deleted: If *Throw* is **true**, then

Mark S. Miller 1/19/09 6:29 PM
Deleted: <#>Else return.

Mark S. Miller 1/19/09 8:57 PM
Comment: Redundant in SES, except that it is a specification error for this to fail in certain ways.

Mark S. Miller 1/19/09 6:31 PM
Deleted: ,

Mark S. Miller 1/19/09 6:31 PM
Deleted: , *and* **false**

Mark S. Miller 1/19/09 6:32 PM
Deleted: , **Throw**

Mark S. Miller 1/19/09 6:32 PM
Deleted: and the boolean flag *Throw*

Mark S. Miller 1/19/09 6:34 PM
Deleted:

Mark S. Miller 1/19/09 6:34 PM
Deleted: Remove

Mark S. Miller 1/19/09 6:33 PM
Deleted: <#>Return **true**.

Mark S. Miller 1/19/09 6:33 PM
Deleted: if *Throw*, then

Mark S. Miller 1/19/09 6:33 PM
Deleted: <#>Return **false**.

Mark S. Miller 1/19/09 8:57 PM
Comment: How do we specify SES's restrictions on valueOf and toString?

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. Let *valueOf* be the result of calling the [[Get]] internal method of object *O* with argument "valueOf".
2. If *valueOf* is an object then,
    a. Let *val* be the result of calling the [[Call]] internal method of *valueOf* , with *O* as the this value and an empty argument list.
    b. If *val* is a primitive value, return *val*.
3. Let *toString* be the result of calling the [[Get]] internal method of object *O* with argument "toString".
4. If *toString* is an object then,
    a. Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as the this value and an empty argument list.
    b. If *str* is a primitive value, return *str*.
5. Throw a **TypeError** exception.

When the [[DefaultValue]] internal method of *O* is called with no hint, then it behaves as if the hint were Number, unless *O* is a Date object (see 15.9), in which case it behaves as if the hint were String.

The above specification of [[DefaultValue]] for native objects can return only primitive values. If a host object implements its own [[DefaultValue]] method, it must ensure that its [[DefaultValue]] method can return only primitive values.

8.12.10 [[DefineOwnProperty]] (P, Desc)

In the following algorithm, the term "Reject" means "Throw a **TypeError** exception."

When the [[DefineOwnProperty]] internal method of *O* is called with property name *P* and property descriptor *Desc*, the following steps are taken:

1. Let *current* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. Let *extensible* be the value of the [[Extensible]] internal property of *O*.
3. If *current* is **undefined** and *extensible* is **false**, then Reject.
4. If *current* is **undefined** and *extensible* is **true**, then
    a. If IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
        i. Create an own data property named *P* of object *O* whose [[Value]], [[Writable]], [[Enumerable]] and [[Configurable]] attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its default value.
    b. Else, *Desc* must be an accessor Property Descriptor so,
        i. Create an own accessor property named *P* of object *O* whose [[Get]], [[Set]], [[Enumerable]] and [[Configurable]] attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its default value.
    c. Return.
5. Return, if every field in *Desc* is absent.
6. Return, if every field in *Desc* also occurs in *current* and the value of every field in *Desc* is the same value as the corresponding field in *current*.
7. If the [[Configurable]] field of *current* is **false** then
    a. Reject, if the [[Configurable]] field of *Desc* is **true**.
    b. Reject, if the [[Enumerable]] field of *current* and *Desc* are the Boolean negation of each other.
8. If IsGenericDescriptor(*Desc*) is **true**, then no further validation is required.
9. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) have different results, then
    a. Reject, if the [[Configurable]] field of *current* is **false**.
    b. If IsDataDescriptor(*current*) is **true**, then
        i. Convert the property named *P* of object *O* from a data property to an accessor property. Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes and set the rest of the property's attributes to their default values.
    c. Else,
        i. Convert the property named *P* of object *O* from an accessor property to a data property. Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes and set the rest of the property's attributes to their default values.

Mark S. Miller 1/19/09 6:34 PM
Deleted: **, Throw**

Mark S. Miller 1/19/09 6:34 PM
Deleted: If *Throw* is **true**, then t

Mark S. Miller 1/19/09 6:34 PM
Deleted: , otherwise return **false**

Mark S. Miller 1/19/09 6:35 PM
Deleted: ,

Mark S. Miller 1/19/09 6:35 PM
Deleted: , and boolean flag *Throw*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

10. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) are both **true**, then
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Reject, if the [[Writable]] field of *current* is **false** and the [[Writable]] field of *Desc* is **true**.
        ii. If the [[Writable]] field of *current* is **false**, then
            1. Reject, if the [[Value]] field of *Desc* is present and SameValue(*Desc*.[[Value]], *current*.[[Value]]) is **false**.
    b. else, the [[Configurable]] field of *current* is **true**, so any change is acceptable.
11. Else, IsAccessorDescriptor(*current*) and IsAccessorDescriptor(*Desc*) are both **true** so,
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Reject, if the [[Set]] field of *Desc* is present and SameValue(*Desc*.[[Set]], *current*.[[Set]] is **false**.
        ii. Reject, if the [[Get]] field of *Desc* is present and SameValue(*Desc*.[[Get]], *current*.[[Get]]) is **false**.
12. For each attribute field of *Desc* that is present, set the correspondingly named attribute of the property named *P* of object *O* to the value of the field.
13. Return **true**.

9 Type Conversion and Testing

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

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

9.1 ToPrimitive

The abstract operation ToPrimitive takes a Value argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts 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 favour 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 a default value for the Object. The default value of an object is retrieved by calling the internal [[DefaultValue]] method of the object, passing the optional hint *PreferredType*. The behaviour of the [[DefaultValue]] method is defined by this specification for all native SES objects (8.6.2.6).

Mark S. Miller 1/19/09 5:01 PM
Deleted: ECMAScript

9.2 ToBoolean

The abstract operation ToBoolean converts 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**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 abstract operation ToNumber converts its argument to a value of type Number according to the following table:

Input Type	*Result*
Undefined	**NaN**
Null	**+0**
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 note below.
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint Number). 2. Call ToNumber(Result(1)). 3. 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$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

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

StrWhiteSpaceChar **:::**
WhiteSpace
LineTerminator

StrNumericLiteral **:::**
StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral **:::**
StrUnsignedDecimalLiteral
`+` *StrUnsignedDecimalLiteral*
`-` *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral **:::**
`Infinity`
DecimalDigits `.` *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
`.` *DecimalDigits ExponentPart*$_{opt}$
DecimalDigits ExponentPart$_{opt}$

DecimalDigits **:::**
DecimalDigit
DecimalDigits DecimalDigit

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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* (see 7.8.3):

- A *StringNumericLiteral* may be preceded and/or followed by white space and/or line terminators.
- 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 white space 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 (see 7.8.3), 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 as described below.

- The MV of *StringNumericLiteral* **:::** [empty] is 0.
- The MV of *StringNumericLiteral* **:::** *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* **:::** $StrWhiteSpace_{opt}$ *StrNumericLiteral* $StrWhiteSpace_{opt}$ is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *StrNumericLiteral* **:::** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* **:::** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral***:::** **`+`** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral***:::** **`-`** *StrUnsignedDecimalLiteral* is the negative of the MV of *StrUnsignedDecimalLiteral*. (Note that if the MV of *StrUnsignedDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this sign less mathematical zero to a floating-point **+0** or **−0** as appropriate.)
- The MV of *StrUnsignedDecimalLiteral***:::** **`Infinity`** is 10^{10000} (a value so large that it will round to $+\infty$).
- The MV of *StrUnsignedDecimalLiteral***:::** *DecimalDigits* **`.`** is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral***:::** *DecimalDigits* **`.`** *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n}), where n is the number of characters in the second *DecimalDigits*.

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

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

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 white space character in the string numeric literal is '**-**', in which case the rounded value is −0. Otherwise, the rounded value must be the number value for the MV (in the sense defined in 8.5), unless the literal includes a *StrUnsignedDecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is *significant* if it is not part of an *ExponentPart* and

it is not **0**; or

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

9.4 ToInteger

The abstract operation ToInteger converts its argument to an integral numeric value. This abstract operation functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, return **+0**.
3. If Result(1) is **+0**, **−0**, **+∞**, or **−∞**, return Result(1).
4. Compute sign(Result(1)) * floor(abs(Result(1))).
5. Return Result(4).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

9.5 ToInt32: (Signed 32 Bit Integer)

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

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, **+0**, **−0**, **+∞**, or **−∞**, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. 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}.
5. If Result(4) is greater than or equal to 2^{31}, return Result(4)– 2^{32}, otherwise return Result(4).

NOTE
Given the above definition of ToInt32:

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

ToInt32(ToUint32(x)) is equal to ToInt32(x) for all values of x. (It is to preserve this latter property that +∞ and −∞ are mapped to +0.)

ToInt32 maps −0 to +0.

9.6 ToUint32: (Unsigned 32 Bit Integer)

The abstract operation ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through $2^{32}-1$, inclusive. This abstraction operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, +0, −0, +∞, or −∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. 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}.
5. Return Result(4).

NOTE
Given the above definition of ToUInt32:

Step 5 is the only difference between ToUint32 and ToInt32.

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

ToUint32(ToInt32(x)) is equal to ToUint32(x) for all values of x. (It is to preserve this latter property that +∞ and −∞ are mapped to +0.)

ToUint32 maps −0 to +0.

9.7 ToUint16: (Unsigned 16 Bit Integer)

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

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, +0, −0, +∞, or −∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. 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}.
5. Return Result(4).

NOTE
Given the above definition of ToUint16:

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

ToUint16 maps −0 to +0.

9.8 ToString

The abstract operation ToString converts its argument to a value of type String according to the following table:

Input Type	*Result*
Undefined	`"undefined"`
Null	`"null"`
Boolean	If the argument is **true**, then the result is `"true"`. If the argument is **false**, then the result is `"false"`.
Number	See note below.
String	Return the input argument (no conversion)
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint String). 2. Call ToString(Result(1)). 3. Return Result(2).

9.8.1 ToString Applied to the Number Type

The abstract operation ToString converts a number *m* to string format as follows:

1. If *m* is **NaN**, return the string `"NaN"`.
2. If *m* is **+0** or **−0**, return the string `"0"`.
3. If *m* is less than zero, return the string concatenation of the string `"-"` and ToString(−*m*).
4. If *m* is infinity, return the string `"Infinity"`.
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the string consisting of the *k* digits of the decimal representation of s (in order, with no leading zeroes), followed by *n*−*k* occurrences of the character '`0`'.
7. If $0 < n \leq 21$, return the string consisting of the most significant *n* digits of the decimal representation of *s*, followed by a decimal point '`.`', followed by the remaining *k*−*n* digits of the decimal representation of *s*.
8. If $-6 < n \leq 0$, return the string consisting of the character '`0`', followed by a decimal point '`.`', followed by −*n* occurrences of the character '`0`', followed by the *k* digits of the decimal representation of *s*.
9. Otherwise, if *k* = 1, return the string consisting of the single digit of *s*, followed by lowercase character '`e`', followed by a plus sign '`+`' or minus sign '`−`' according to whether *n*−1 is positive or negative, followed by the decimal representation of the integer abs(*n*−1) (with no leading zeros).
10. Return the string consisting of the most significant digit of the decimal representation of s, followed by a decimal point '.', followed by the remaining k−1 digits of the decimal representation of s, followed by the lowercase character 'e', followed by a plus sign '+' or minus sign '−' according to whether n−1 is positive or negative, followed by the decimal representation of the integer abs(n−1) (with no leading zeros).

NOTE
The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard:

If x is any number value other than ***−0****, then ToNumber(ToString(x)) is exactly the same number value as x.*

The least significant digit of s is not always uniquely determined by the requirements listed in step 5.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 5 be used as a guideline:

Otherwise, let n, k, and s be integers such that k ≥ 1, $10^{k-1} \leq s < 10^k$, the number value for s × 10^{n-k} is m, and k is as small as possible. If there are multiple possibilities for s, choose the value of s for which s × 10^{n-k} is closest in value to m. If there are two such possible values of s, choose the one that is even. Note that k is the number of digits in the decimal representation of s and that s is not divisible by 10.

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

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.*

Mark S. Miller 1/19/09 5:02 PM
Deleted: *ECMAScript*

9.9 ToObject

The abstract operation ToObject converts its argument to a value of type Object according to the following table:

Input Type	***Result***
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Create a new Boolean object whose [[PrimitiveValue]] property is set to the value of the boolean. See 15.6 for a description of Boolean objects.
Number	Create a new Number object whose [[PrimitiveValue]] property is set to the value of the number. See 15.7 for a description of Number objects.
String	Create a new String object whose [[PrimitiveValue]] property is set to the value of the string. See 15.5 for a description of String objects.
Object	The result is the input argument (no conversion).

9.10 CheckObjectCoercible

The abstract operation CheckObjectCoercible throws an error if its argument is a value that can not be converted to an Object using ToObject. It is defined by the following table:

Input Type	***Result***
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return
Number	Return
String	Return
Object	Return

9.11 IsCallable

The abstract operation IsCallable determines if its argument, which must be an SES language value, is a callable function Object according to the following table:

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Input Type	***Result***
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return **false**.
Number	Return **false**.
String	Return **false**.
Object	If the argument object has an internal [[Call]] method, then return **true**, otherwise

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

	return **false**.

9.12 The SameValue Algorithm

The internal comparison abstract operation SameValue(*x*, *y*), where *x* and *y* are SES language values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is Number, then.
   a. If *x* is NaN and *y* is NaN, return **true**.
   b. If *x* is +0 and *y* is -0, return **false**.
   c. If *x* is -0 and *y* is +0, return **false**.
   d. If *x* is the same number value as *y*, return **true**.
   e. Return **false**.
5. 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**.
6. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
7. Return true if *x* and *y* refer to the same object. Otherwise, return **false**.

10 Executable Code and Execution Contexts

10.1 Types of Executable Code

There are three types of SES executable code:

Global code is source text that is treated as an SES *Program*. The global code of a particular *Program* does not include any source text that is parsed as part of a *FunctionBody*.

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 SES *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 parsed as part of a *FunctionBody*. The *function code* of a particular *FunctionBody* does not include any source text that is parsed as part of a nested *FunctionBody*.

10.1.1 N/A

10.2 Lexical Environments

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

An *Environment Record* records the identifier bindings that are created within the scope of its associated Lexical Environment.

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

Lexical Environments and Environment Record values are purely specification mechanisms and need not correspond to any particular artefact of an SES implementation. It is impossible for an SES program to directly access or manipulate such values.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:08 PM
Deleted: *Function code* also denotes the source text supplied when using the built-in **Function** object as a constructor. More precisely, the last parameter provided to the **Function** constructor is converted to a string and treated as the *FunctionBody*. If more than one parameter is provided to the **Function** constructor, all parameters except the last one are converted to strings and concatenated together, separated by commas. The resulting string is interpreted as the *FormalParameterList* for the *FunctionBody* defined by the last parameter. The function code for a particular instantiation of a **Function** does not include any source text that is parsed as part of a nested *FunctionBody*.

Mark S. Miller 1/19/09 7:08 PM
Deleted: Strict Mode Code

Mark S. Miller 1/19/09 7:08 PM
Deleted: As described in section 4.2.2, an ECMAScript *Program* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as strict global code, strict eval code, and strict function code. Code is interpreted in strict mode code in the following situations:
<#>Global code is strict global code if the *Program* that defines the global code includes a Use Strict Directive *(14.1)*.
<#>Eval code is strict eval code if the *Program* that defines the eval code includes a Use Strict Directive (14.1) or if the call to eval is a direct call (see section 15.1.2.1) to the eval function that is contained in strict mode code.
<#>Function code that is part of a *FunctionDeclaration* or *FunctionExpression* is strict function code if its *FunctionDeclaration* or *FunctionExpression* is contained in strict mode code or if its *FunctionBody* includes a Use Strict Directive (14.1).
<#>Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a string that when processed as a *FunctionBody* includes a Use Strict Directive (14.1).

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

10.2.1 Environment Records

Declarative environment records are used to define the effect of SES language syntactic elements such as *FunctionDeclarations*, *VariableDeclarations*, and *Catch* clauses that directly associate **identifier** bindings with SES language values or variables.

For specification purposes Declarative Environment Record values can be thought of as implemeting the following methods:

Method	*Purpose*
HasBinding(N)	Determine if an environment record has a binding for an identifier. Return **true** if it does and **false** if it does not. The string value N is the text of the identifier.
CreateMutableBinding(N)	Create a new mutable binding in an environment record. The string value N is the text of the bound name.
GetBindingValue(N)	Returns the value of an already existing binding from an environment record. The string value N is the text of the bound name. If the binding is an uninitialized immutable binding throw a ReferenceError exception.
SetMutableBinding(N,V)	Set the value of an already existing mutable binding in an environment record. The string value N is the text of the bound name. V is the value for the binding and may be a value of any SES language type. If the binding cannot be set throw a TypeError exception.

10.2.1.1 Declarative Environment Records

Each declarative environment record is associated with a ECMAScirpt program scope containing variable, and or function declarations. A declarative environment record binds the set of identifiers defined by the declarations contained within its scope.

In addition to the mutable binds supported by all Environment Records, declarative environment records also provide for immutable bindings. An immutable binding is one where the association between an identifier and a value may not be modified once it has been established. Declarative environment records support the following methods in addition to the Environment Record abstract specification methods:

Method	*Purpose*
CreateImmutableBinding(N)	Create a new but uninitialized immutable binding in an environment record. The string value N is the text of the bound name.
InitializeImmutableBinding(N,V)	Set the value of an already existing but uninitialized immutable binding in an environment record. The string value N is the text of the bound name. V is the value for the binding and is a value of any SES language type.

The behaviour of the concrete specification methods for Declarative Environment Records are defined by the following algorithms.

10.2.1.1.1 HasBinding(*N*)

The concrete environment record method HasBinding for declarative environment records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. If *envRec* has a binding for the name that is the value of *N*, return **true**.
3. If it does not have such a binding, return **false**

Mark S. Miller 1/19/09 7:12 PM
Deleted: There are two kinds of Environment Record values used in this specification: *declarative environment records* and *object environment records*.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:11 PM
Deleted: Object environment records are used to define the effect of ECMAScript elements such as *Program* and *WithStatement* that associate **identifier** bindings with the properties of some object.

Mark S. Miller 1/19/09 7:13 PM
Deleted: existing in a simple object-oriented hierarchy where Environment Record is an abstract class with two concrete subclasses, declarative environment record and object environment record. The abstract class defines the following abstract specification methods that have distinct concrete algorithms for each of its subclasses

Mark S. Miller 1/19/09 7:14 PM
Deleted: ,S

Mark S. Miller 1/19/09 7:14 PM
Deleted: S is **true** and

Mark S. Miller 1/19/09 7:15 PM
Deleted: S is used to identify strict mode references.

Mark S. Miller 1/19/09 7:15 PM
Deleted: , S

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:15 PM
Deleted: S is a Boolean flag.

Mark S. Miller 1/19/09 7:15 PM
Deleted: S is **true** and

Mark S. Miller 1/19/09 7:16 PM
Deleted: S is used to identify strict mode references.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

10.2.1.1.2 CreateMutableBinding (N)

The concrete Environment Record method CreateMutableBinding for declarative environment records creates a new mutable binding for the name *N* that is initialized to the value **undefined**. A binding must not already exist in this Environment Record for *N*.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create a mutable binding in *envRec* for *N* and set its bound value to **undefined**.

10.2.1.1.3 SetMutableBinding (N,V)

The concrete Environment Record method SetMutableValue for declarative environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. A binding for *N* must already exist. If the binding is an immutable binding, a TypeError is always thrown.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* must have a binding for *N*.
3. If the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
4. Else this must be an attempt to change the value of an immutable binding so throw a **TypeError** exception.

10.2.1.1.4 GetBindingValue(N)

The concrete Environment Record method GetBindingValue for declarative environment records simply returns the value of its bound identifier whose name is the value of the argument *N*. The binding must already exist. If the binding is an uninitialized immutable binding throw a ReferenceError exception.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* has a binding for *N*.
3. If the binding for *N* in *envRec* is an uninitialized immutable binding, then throw a **ReferenceError** exception.
4. Else, return the value currently bound to *N* in *envRec*.

10.2.1.1.5 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name *N* that is initialized to the value undefined. A binding must not already exist in this environment record for *N*.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create an immutable binding in *envRec* for *N* and record that it is uninitialized.

10.2.1.1.6 InitializeImmutableBinding (N,V)

The concrete Environment Record method InitializeImmutableBinding for declarative environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. An uninitialized immutable binding for *N* must already exist.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* must have an uninitialized immutable binding for N.
3. Set the bound value for *N* in *envRec* to *V*.
4. Record that the immutable binding for *N* in *envRec* has been initialized.

10.2.1.2 N/A

10.2.2 Lexical Environment Operations

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

10.2.2.1 GetIdentifierReference (lex, name, strict)

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

Mark S. Miller 1/19/09 7:17 PM
Deleted: ,S

Mark S. Miller 1/19/09 7:17 PM
Deleted:

Mark S. Miller 1/19/09 7:17 PM
Deleted: The *S* argument is ignored because strict mode does not change the meaning of setting bindings in declarative environment records have.

Mark S. Miller 1/19/09 7:18 PM
Deleted: ,S

Mark S. Miller 1/19/09 7:18 PM
Deleted: *S* is **true** and

Mark S. Miller 1/19/09 7:18 PM
Deleted:

Mark S. Miller 1/19/09 7:19 PM
Deleted: If *S* is **false**, return the value **undefined**, otherwise

Mark S. Miller 1/19/09 7:19 PM
Deleted: **Object Environment Records**

Mark S. Miller 1/19/09 7:20 PM
Deleted: Each object environment record is associated with an object called its *binding object*. An environment record binds the set of identifiers that directly correspond to the property names of its binding object. Property names that are not identifiers are not included in the set of bound identifiers. Because properties can be dynamically added and deleted from objects, the set of identifiers bound by an object environment record may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a result of such a side-effect is considered to be a mutable binding even if the Writable attribute of the corresponding property has the value **false**. Immutable bindings do not exist for object environment records.
The behaviour of the concrete specification methods for Object Environment Records is defined by the following algorithms.
10.2.1.2.1 HasBinding(N)
The concrete Environment Record method HasBinding for object environment records determines if its associated binding object has a property whose name is the value of the argument *N*:
<#>Let *envRec* be the object environment record for which the method was invoked.
<#>Let *bindings* be the binding object for *envRec*.
<#>Return the result of calling the [[HasProperty]] method of *bindings*, passing *N* as the property name.
10.2.1.2.2 CreateMutableBinding (N)
The concrete Environment Record method CreateMutableBinding for object environment records creates a property whose name is the string value *N* in the environment record and initializes it to the value **undefined**. A property named *N* must not already exist in the binding object.
<#>Let *envRec* be the declarative environment record for which the method was invoked.
<#>Let *bindings* be the binding object for *envRec*.
<#>Assert: The result of calling the [[HasProperty]] method of *bindings*, passing *N* as the property name is **false**.
<#>Call the [[Put]] method of *bindings*, passing *N* and **undefined** for the arguments.
10.2.1.2.3 SetMutableBinding (N,V,S)
The concrete Environment Record method ... [16]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. If *lex* is the value **null**, then
   a. Return a value of type Reference whose base value is **null**, whose referenced name is *name*, and whose strict mode flag is *strict.*
2. Let *envRec* be *lex*'s environment record.
3. Let *exists* be the result of calling the HasBinding(*N*) concrete method of *envRec* passing *name* as the argument *N.*
4. If *exists* is **true**, then
   a. Return a value of type Reference whose base value is *envRec*, whose referenced name is *name*, and whose strict mode flag is *strict.*
5. Else
   a. Let *outer* be the value of *lex's* outer environment reference.
   b. Return the result of calling GetIdentifierReference passing *outer*, *name*, and *strict* as arguments..

10.2.2.2 NewDeclarativeEnvironmentRecord(E)

When the abstract operation NewDeclarativeEnvironmentRecord is called with either a Lexical Environment or **null** as argument *E* the following steps are performed:

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

10.2.2.3 NewObjectEnvironmentRecord(O, E)

When the abstract operation NewObjectEnvironmentRecord is called with an Object *O* and a Lexical Environment *E* (or **null)** as arguments*,* the following steps are performed:

1. Let *env* be a new Lexical Environment.
2. Let *envRec* be a new ObjectEnvironmentRecord containing using *O* as the binding object.
3. Set *env's* environment record to be *envRec.*
4. Set the outer lexical environment reference of *env* to *E.*
5. Return *env*.

10.2.3 The Global Environment

The *global environment* is a unique Lexical Environment which is created before any SES code is executed. The global environment's Environment Record is an declarative environment record whose binding object is the global object (15.1). The global environments's outer environment reference is **null**.

The global environment is *immutable*. It, and all values reachable from it are frozen, and so are safely sharable without violating isoation.

10.3 Execution Contexts

When control is transferred to SES executable code, 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. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context.

An execution context contains whatever state is necessary to tract the execution progress of its associated code. In addition, each execution context has the following state components:

Component	***Purpose***
LexicalEnvironment	Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.
VariableEnvironmnet	Identifies the Lexical Environment whose environment record holds bindings created by *VariableStatements* and *FunctionDeclarations* within this execution context.
ThisBinding	The value associated with the **`this`** keyword within tamed code associated

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:21 PM
Deleted: object

Mark S. Miller 1/19/09 7:22 PM
Deleted: As ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modifie

Mark S. Miller 1/19/09 7:22 PM
Deleted: d.

Mark S. Miller 1/19/09 8:57 PM
Comment: SES is statically scoped, so it would be good to simplify away this dynamic concept

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 8:57 PM
Comment: In the current spec architecture, this is needed to explain the behavior of certain tamed methods. But it would be good to explain this behavior without using a ThisBinding concept.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

	with this execution context.

The LexicalEnvironment and VariableEnvironment components of an execution context are always Lexical Environments. When an execution context is created its LexicalEnvironment and VariableEnvironment components initially have the same value. The value of the VariableEnvironment component never changes while the value of the LexicalEnvironment component may change during execution of code within an execution context.

In most situations only the running execution context (the top of the execution context stack) is directly manipulated by algorithms within this specification. Hence when the terms "LexicalEnvironment", "VariableEnvironment" and "ThisBinding" are used without qualification they are in reference to those components of the running execution context.

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an SES implementation. It is impossible for an SES program to access an execution context.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

10.3.1 Identifier Resolution

Identifier resolution is the process of determining the binding of an *Identifer* using the LexicalEnvironment of the running execution context. During execution of SES code, the syntactic production *PrimaryExpression* : *Identifier* is evaluated using the following algorithm:

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

1. Let *env* be the running execution context's LexicalEnvironment.
2. If the syntactic production that is being evaluated is contained in a strict mode code, then let *strict* be **true** else let *strict* be **false**.
3. Return the result of calling GetIdentifierReference function passing *env*, *Identifier*, and *strict* as arguments.

The result of evaluating an identifier is always a value of type Reference with its referenced name component equal to the *Identifier* string.

10.4 Establishing an Execution Context

Evaluation of global code or code using the eval function (15.1.2.1) establishes and enters a new execution context. Every invocation of an SES code function (13.2.1) also establishes and enters a new execution context, even if a function is calling itself recursively. Every return exits an execution context. A thrown exception may also exit one or more execution contexts.

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

When control enters an execution context, the execution context's ThisBinding is set, its VariableEnvironment and initial LexicalEnvironment are defined, and declaration binding instantiation is performed. The exact manner in which these actions occur depend on the type of code being entered.

10.4.1 Global Code

The following steps are performed when control enters the execution context for global code:

1. Initialize the execution context using the global code as described in 10.4.1.1.
2. Perform Declaration Binding Instantiation as described in 10.6 using the global code.

10.4.1.1 Initial Global Execution Context

The following steps are performed to initialize an execution context for SES code *C*:

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

1. Set the VariableEnvironment to the Global Environment.
2. Set the LexicalEnvironment to the Global Environment.
3. Set the ThisBinding to **undefined**.

Mark S. Miller 1/19/09 7:29 PM
Deleted: If *C* is strict code, s

Mark S. Miller 1/19/09 7:29 PM
Deleted: , otherwise set the ThisBinding to the global object

10.4.2 Eval Code

The following sets are performed when control enters the execution context for eval code:

1. If there is no calling context or if the eval code is not being evaluated by a direct call (15.1.2.1) then,
   a. Initialize the execution context as if it was a global execution context using the eval code as *C* as described in 10.4.1.1.
2. Else,
   a. Set the ThisBinding to the same value as the ThisBinding of the calling execution context.
   b. Set the LexicalEnvironment to the LexicalEnvironment of the calling execution context.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

c. Let *strictVarEnv* be the result of calling NewDeclarativeEnvironmentRecord(E) passing the LexicalEnvironment as the argument.

d. Set the VariableEnvironment to *strictVarEnv*.

3. Perform Declaration Binding Instantiation as described in 10.6 using the eval code.

The eval code cannot instantiate variable or function bindings in the variable environment of the calling context that invoked the eval. Instead such bindings are instantiated in a new VariableEnvironment that is only accessible to the eval code.

10.4.3 Function Code

The following sets are performed when control enters the execution context for function code contained in function object *F*, a caller provided *thisArg*, and a caller provided *argumentsList*:

1. Set the ThisBinding to *thisArg*.
2. Let *localEnv* be the result of calling NewDeclarativeEnvironmentRecord(E) passing the value of the [[Scope]] property of *F* as the argument.
3. Set the LexicalEnvironment to *localEnv*.
4. Set the VariableEnvironment to *localEnv*.
5. Let *code* be the value of F's [[Code]] internal property.
6. Perform Declaration Binding Instantiation using the function code *code* and *argumentList* as described in 10.6.

10.5 Arguments Object

When control enters an execution context for function code, an arguments object is created.

The arguments object is created by calling the abstract operation CreateArgumentsObject with arguments *args* the actual arguments passed to the [[Call]] method. When CreateArgumentsObject is called the following steps are performed:

1. Let *len* be the number of elements in *args*.
2. Let *obj* be the result of creating a new SES object.
3. Set the [[Class]] internal property of *obj* to **"Object"**.
4. Set the "constructor" property of *obj* to the standard built-in Object constructor (15.2.3).
5. Set the [[Parent]] internal property of *obj* to the standard built-in Array prototype object (15.4.4).
6. Call the [[DefineOwnProperty]] internal method on *obj* passing **"length"** and the Property Descriptor {[[Value]]: *len*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**} as arguments.
7. Let *indx* = *len* - 1.
8. Repeat while *indx* >= 0,
   a. Let *val* be the element of *args* at 0-origined list position *indx*.
   b. Call the [[DefineOwnProperty]] method on *obj* passing ToString(*indx*) and the property descriptor {[[Value]]: *val*, [[Writable]]: **false**, [[Enumerable]]: **true**, [[Configurable]]: **false**} as arguments.
   c. Let *indx* = *indx* - 1
9. Let *f* be a function which when evaluated throws a **TypeError** exception and performs no other actions.
   a. Call the [[DefineOwnProperty]] internal method of *obj* passing **"callee"** and the Property Descriptor {[[Get]]: *f*, [[Put]]: *f*, [[Enummerable]]: **false**, [[Configurable]]: **false**} as arguments.
   b. Call the [[DefineOwnProperty]] internal method of *obj* passing **"caller"** and the Property Descriptor {[[Get]]: *f*, [[Put]]: *f*, [[Enummerable]]: **false**, [[Configurable]]: **false**} as arguments.
10. Set the internal [[Extensible]] property of *obj* to **false**.

11. Return *obj*

NOTE
Arguments objects define non-configurable data properties named "caller" and "callee" whose values are ***undefined****. The "callee" property has a more specific meaning for non strict mode ECMAScript functions and a "callee" property has historically been provided as an implementation-defined extension by some ECMAScript implementations. The SES definition of these properties exists to ensure that neither of them is defined in any other manner by conforming SES implementations.*

Mark S. Miller 1/19/09 7:30 PM
Deleted: <#>Set the VariableEnvironment to the same value as the VariableEnvironment of the calling execution context.
<#>If the eval code is strict code, then

Mark S. Miller 1/19/09 7:31 PM
Deleted: 10.4.2.1 Strict Mode Restrictions
if either the code of the calling context or the eval code is strict code ... [17]

Mark S. Miller 1/19/09 7:32 PM
Deleted: If the function code is strict code, s

Mark S. Miller 1/19/09 7:32 PM
Deleted: <#>Else if *thisArg* is **null** or **undefined**, set the ThisBinding to the global object.
<#>Else if *thisArg* is not an Object, set the ThisBinding to ToObject(*thisArg*).
<#>Else set the ThisBinding to *thisArg*.

Mark S. Miller 1/19/09 7:33 PM
Deleted: may be

Mark S. Miller 1/19/09 7:34 PM
Deleted: *func* the function object whose code is to be evaluated, *names* a List containing the function's formal parameter names,..., *env* the variable environment for the function code, and *strict* a Boolean that indicates whether or not the function code is strict code ... [18]

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:35 PM
Deleted: [[Constructor]] internal

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 7:35 PM
Deleted: ", ...**true**...**true**..., and **false** ... [19]

Mark S. Miller 1/19/09 7:36 PM
Deleted: <#>Let *map* be be the result of creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name
<#>Let *mappedNames* be an empty List.

Mark S. Miller 1/19/09 7:37 PM
Deleted: ,...**true**...**true**...,...and **false** ... [20]

Mark S. Miller 1/19/09 7:38 PM
Deleted: <#>If *indx* is less than the number of elements in *names*, then
<#>Let *name* be the element of *names* at 0-origined list position *indx*.
<#>If *name* is not an element of *mappedNames*, th ... [21]

Mark S. Miller 1/19/09 7:39 PM
Deleted: <#>If *mappedNames* is not empty, then
<#>Set the [[ParameterMap]] internal property of *o* ... [22]

Mark S. Miller 1/19/09 7:41 PM
Deleted: ", ..., and **true** ... [23]

Mark S. Miller 1/19/09 7:41 PM
Deleted: ", ..., and **true** ... [24]

Mark S. Miller 1/19/09 7:43 PM
Deleted: The function *MakeArgGetter* called with string *name* and environment record *env* creates a functio ... [25]

Mark S. Miller 1/19/09 7:45 PM
Deleted: *Numerically named data properties of an arguments object whose non-negative numeric nam* ... [26]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

10.6 Declaration Binding Instantiation

Every execution context has associated with a VariableEnviornment. Variables and functions declared in SES code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.

Which Environment Record is used to bind declaration and its kind depends upon the type of SES code executed by the execution context, but the remainder of the behaviour is generic. On entering an execution context, bindings are created in the VariableEnvironment environment record as follows using the caller provided *code* and (if it is function code) a function object *func* and argument list *args*:

1. Let *env* be the environment record component of the running execution context's VariableEnvironment.
2. If *code* is eval code, let *eval* be **true**, otherwise let *eval* be **false**.
3. If *code* is function code, then
   a. Let *func* be the function whose [[Call]] internal method initiated execution of *code*. Let *names* be the value of *func*'s [[FormalParameters]] internal property.
   b. Let *argCount* be the number of elements in *args*.
   c. Let *n* be the number 0.
   d. For each string *argName* in *names*, in list order do
      i. Let *n* be the current value of *n* plus 1.
      ii. If *n* is greater than *argCount*, let *v* be **undefined** otherwise let *v* be the value of the *n*'th element of *args*.
      iii. Call *env's* CreateMutableBind(*N*) concrete method passing *argName* as the argument.
      iv. Call *env's* SetMutableBinding(*N*,*V*) concrete method passing *argName* and *v* as the arguments.
4. For each *FunctionDeclaration f* in the execution context's code, in source text order do
   a. Let *fn* be the *Identifier* in *FunctionDeclaration f.*
   b. Let *fo* be the result of evaluating *FunctionDeclaration* for *f* as described in 13.
   c. Let *funcAlreadyDeclared* be the result of calling *env's* HasBinding(*N*) concrete method passing *fn* as the argument.
   d. If *funcAlreadyDeclared* is **false**, call *env's* CreateMutableBind(*N*) concrete method passing *fn* as the argument.
   e. Else if *eval* is **true** throw an **EvalError** exception.
   f. Call *env's* SetMutableBinding(*N*,*V*) concrete method passing *fn* and *fo* as the arguments.
5. For each *VariableDeclaration* and *VariableDeclarationNoIn d* in the execution context's code, in source text order do
   a. Let *dn* be the *Identifier* in *d*.
   b. Let *varAlreadyDeclared* be the result of calling *env's* HasBinding(*N*) concrete method passing *dn* as the argument.
   c. If *varAlreadyDeclared* is **false**, then
      i. Call *env*'s CreateMutableBind(*N*) concrete method passing *dn* as the argument.
      ii. Call *env*'s SetMutableBinding(*N*,*V*,*S*) concrete method passing *dn*, **undefined**, and *strict* as the arguments.
   d. Else if *eval* is **true** throw an **EvalError** exception.
6. Let *argumentsAlreadyDeclared* be the result of calling *env's* HasBinding(*N*) concrete method passing `"arguments"` as the argument
7. If the code is function code and *argumentsAlreadyDeclared* is **false**, then
   a. Let *argsObj* be the result of calling the abstract operation CreateArgumentsObject passing *func, names, args, env* and *strict* as arguments.
      i. Call *env*'s CreateImmutableBinding(*N*) concrete method passing the string `"arguments"` as the argument.
      ii. Call *env*'s InitializeImmutableBind(*N*,*V*) concrete method passing `"arguments"` and *argsObj* as arguments.

11 Expressions

11.1 Primary Expressions

Syntax

Mark S. Miller 1/19/09 5:02 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 7:47 PM
Deleted:

Mark S. Miller 1/19/09 7:48 PM
Deleted: <#>If *code* is strict mode code, then let *strict* be **true** else let *strict* be **false**.

Mark S. Miller 1/19/09 7:49 PM
Deleted: <#>Let *argAlreadyDeclared* be the result of calling *env's* HasBinding(*N*) concrete method passing *argName* as the argument.
If *argAlreadyDeclared* is **false**, c

Mark S. Miller 1/19/09 7:49 PM
Deleted: ,*S*

Mark S. Miller 1/19/09 7:49 PM
Deleted: ,

Mark S. Miller 1/19/09 7:49 PM
Deleted: , and *strict*

Mark S. Miller 1/19/09 7:50 PM
Deleted: *strict* is **true** and

Mark S. Miller 1/19/09 7:50 PM
Deleted: ,*S*

Mark S. Miller 1/19/09 7:50 PM
Deleted: ,

Mark S. Miller 1/19/09 7:51 PM
Deleted: , and *strict*

Mark S. Miller 1/19/09 7:52 PM
Deleted: *strict* is **true** and

Mark S. Miller 1/19/09 7:53 PM
Deleted: <#>If *strict* is **true**, then

Mark S. Miller 1/19/09 7:53 PM
Deleted: <#>Else,
<#>Call *env*'s CreateImmutableBinding(*N*,*D*) concrete method passing the string `"arguments"` and **false** as the arguments.
<#>Call *env*'s SetMutableBind(*N*,*V*,*S*) concrete method passing `"arguments"`, *argsObj*, and *strict* as arguments.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

PrimaryExpression **:**
Identifier
Literal
ArrayLiteral
ObjectLiteral
(*Expression* **)**

Mark S. Miller 1/19/09 7:54 PM
Deleted: `this`

11.1.1 N/A

11.1.2 Identifier Reference

Mark S. Miller 1/19/09 7:54 PM
Deleted: The `this` Keyword

Mark S. Miller 1/19/09 7:55 PM
Deleted: The `this` keyword evaluates to the **this** value of the execution context.

An *Identifier* is evaluated using the scoping rules stated in 10.3.1. The result of evaluating an *Identifier* is always a value of type Reference.

Mark S. Miller 1/19/09 8:57 PM
Comment: To fix this, we need to account for lvalues in assignments and ThisBindings in invoking some tamed methods.

11.1.3 Literal Reference

A *Literal* is evaluated as described in 7.8.

11.1.4 Array Initialiser

An array initialiser is an expression describing the initialisation of an Array object, written in a form of a literal. It is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initialiser is evaluated.

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an *AssignmentExpression* (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.

Syntax

ArrayLiteral **:**
[$Elision_{opt}$ **]**
[*ElementList* **]**
[*ElementList* **,** $Elision_{opt}$ **]**

ElementList **:**
$Elision_{opt}$ *AssignmentExpression*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*

Elision **:**
,
Elision **,**

Semantics

The production *ArrayLiteral* **: [** $Elision_{opt}$ **]** is evaluated as follows:

1. Let *array* be the result of creating a new object as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Call the [[Put]] internal method of *array* with arguments **`"length"`** and *pad*.
4. Return *array*.

Mark S. Miller 1/19/09 8:57 PM
Comment: Is this safe? Should it be [[DefineOwnProperty]]?

The production *ArrayLiteral* **: [** *ElementList* **]** is evaluated as follows:

1. Return the result of evaluating *ElementList*.

The production *ArrayLiteral* **: [** *ElementList* **,** $Elision_{opt}$ **]** is evaluated as follows:

1. Let *array* be the result of evaluating *ElementList*.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *len* be the result of calling the [[Get]] internal method of *array* with argument **`"length"`**.
4. Call the [[Put]] internal method of *array* with arguments **`"length"`** and ToUint32(*pad*+*len*).
5. Return *array*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The production *ElementList* **:** *Elision*$_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Let *array* be the result of creating a new object as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
2. Let *firstIndex* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *initResult* be the result of evaluating *AssignmentExpression*.
4. Let *initValue* be GetValue(*initResult*).
5. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*firstIndex*) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
6. Return *array*.

Mark S. Miller 1/19/09 8:02 PM
Deleted:),

Mark S. Miller 1/19/09 8:02 PM
Deleted: , and **false**

The production *ElementList* **:** *ElementList* **,** *Elision*$_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Let *array* be the result of evaluating *ElementList*.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *initResult* be the result of evaluating *AssignmentExpression*.
4. Let *initValue* be GetValue(*initResult*).
5. Let *len* be the result of calling the [[Get]] internal method of *array* with argument **`"length"`**.
6. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32((*pad*+*len*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
7. Return *array*.

Mark S. Miller 1/19/09 8:02 PM
Deleted: , and **false**

The production *Elision* **:** **,** is *evaluated* as follows:

1. Return the numeric value **1**.

The production *Elision* **:** *Elision* **,** is evaluated as follows:

1. Let *preceeding* be the result of evaluating Elision.
2. Return (*preceeding*+1).

NOTE
The use of [[Put]] rather than [[ThrowingPut]] in this section is intentional as there are no situations where these [[Put]] operations may fail.

Mark S. Miller 1/19/09 8:04 PM
Deleted: *[[DefineOwnProperty]] is used to ensure that own properties are defined for the array even if the standard built-in Array prototype object has been modified in a manner that would preclude the creation of new own properties using [[Put]].*

11.1.5 Object Initialiser

An object initialiser is an expression describing the initialisation of an Object, written in a form resembling a literal. It is a list of zero or more pairs of property names and associated values, enclosed in curly braces. The values need not be literals; they are evaluated each time the object initialiser is evaluated.

Syntax

ObjectLiteral **:**
  { }
  { *PropertyNameAndValueList* **}**
  { *PropertyNameAndValueList* **, }**

PropertyNameAndValueList **:**
  PropertyAssignment
  PropertyNameAndValueList **,** *PropertyAssignment*

PropertyAssignment **:**
  PropertyName **:** *AssignmentExpression*
  get *PropertyName* **( ) {** *FunctionBody* **}**
  set *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

PropertyName **:**
  IdentifierName
  StringLiteral
  NumericLiteral

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

PropertySetParameterList **:**
Identifier

Semantics

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

1. Create a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
2. Return Result(1).

The productions *ObjectLiteral* **: {** *PropertyNameAndValueList* **}** and

ObjectLiteral **: {** *PropertyNameAndValueList* **, }** are evaluated as follows:

1. Return the result of evaluating *PropertyNameAndValueList*.

The production
PropertyNameAndValueList **:** *PropertyAssignment*
is evaluated as follows:

1. Let *obj* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
2. Let *propId* be the result of evaluating *PropertyAssignment*.
3. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name and *propId*.descriptor.
4. Return *obj*.

Mark S. Miller 1/19/09 8:01 PM
Deleted: ,

Mark S. Miller 1/19/09 8:01 PM
Deleted: , and **false**

The production
PropertyNameAndValueList **:** *PropertyNameAndValueList* **,** *PropertyAssignment*
is evaluated as follows:

1. Let *obj* be the result of evaluating *PropertyNameAndValueList*.
2. Let *propId* be the result of evaluating *PropertyAssignment*.
3. Let *previous* be the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *propId*.name.
4. If *previous* is not **undefined** then throw a **SyntaxError** exception if any of the following conditions are true
   a. IsAccessorDescriptor(*previous*) is **true** and IsAccessorDescriptor(*propId*.descriptor) is **true**.
   b. IsPropertyDescriptor(*previous*) is **true** and IsAccessorDescriptor(*propId*.descriptor) is **true.**
   c. IsAccessDescriptor(*previous*) is **true** and IsPropertyDescriptor(*propId*.descriptor) is **true**.
   d. IsPropertyDescriptor(*previous*) is **true** and IsPropertyDescriptor(*propId*.descriptor) is **true** and either both *previous* and *propId*.descriptor have [[Get]] fields or both *previous* and *propId*.descriptor have [[Set]] fields
5. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name and *propId*.descriptor.
6. Return ***obj***.

Mark S. Miller 1/19/09 8:00 PM
Deleted: This production is contained in strict code and

Mark S. Miller 1/19/09 8:01 PM
Deleted: ,

Mark S. Miller 1/19/09 8:01 PM
Deleted: , and **false**

If the above steps would throw a SyntaxError then an implementation must report the error immediately when scanning the program.

The production *PropertyAssignment* **:** *PropertyName* **:** *AssignmentExpression* is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *exprValue* be the result of evaluating *AssignmentExpression*.
3. Let *propValue* be GetValue(*exprValue*).
4. Let *desc* be the Property Descriptor{[[Value]]: propValue, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
5. Return Property Identifer (*propName*, *desc*).

The production *PropertyAssignment* **: *get*** *PropertyName* ***( ) {*** *FunctionBody* ***}*** is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *descriptiveName* be the result of concatenating the String `"get "` and *propName*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

3. Let *closure* be the reault of creating a new Function object as specified in 13.2 with an empty parameter list and body specified by *FunctionBody*. Pass in the Lexical Environment of the running execution context as the *Scope*, and *descriptiveName* as the *Name*.
4. Let *desc* be the Property Descriptor{[[Get]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
5. Return Property Identifier (*propName*, *desc*).

The production *PropertyAssignment* **:** ***set*** *PropertyName* **(** *PropertySetParameterList* **)** ***{*** *FunctionBody* ***}*** is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *descriptiveName* be the result of concatenating the String **"set "** and *propName*.
3. Let *closure* be the result of creating a new Function object as specified in 13.2 with parameters specified by *PropertySetParameterList* and body specified by *FunctionBody*. Pass in the Lexical Environment of the running execution context as the *Scope*, and *descriptiveName* as the *Name*.
4. Let *desc* be the Property Descriptor{[[Set]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
5. Return Property Identifier (*propName*, *desc*).

The production *PropertyName* **:** *IdentifierName* is evaluated as follows:

1. Return the string value containing the same sequence of characters as the *IdentifierName*.

The production *PropertyName* **:** *StringLiteral* is evaluated as follows:

1. Return the value of the *StringLiteral*.

The production *PropertyName* **:** *NumericLiteral* is evaluated as follows:

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.
2. Return ToString(*nbr*).

11.1.6 The Grouping Operator

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

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

NOTE
This algorithm does not apply GetValue to Result(1). The principal motivation for this is so that operators such as **delete** *and* **typeof** *may be applied to parenthesised expressions.*

11.2 Left-Hand-Side Expressions

Syntax

MemberExpression **:**
PrimaryExpression
FunctionExpression
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
new *MemberExpression Arguments*

NewExpression **:**
MemberExpression
new *NewExpression*

CallExpression **:**
MemberExpression Arguments
CallExpression Arguments
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 **.** *IdentifierName*
CallExpression **.** *IdentifierName*

or the bracket notation:

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

The dot notation is explained by the following syntactic conversion:

MemberExpression **.** *IdentifierName*

is identical in its behaviour to

MemberExpression **[** *<identifier-name-string>* **]**

and similarly

CallExpression **.** *IdentifierName*

is identical in its behaviour to

CallExpression **[** *<identifier-name-string>* **]**

where <identifier-name-string> is a string literal containing the same sequence of characters after processing of Unicode escape sequences as the *IdentifierName*.

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

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be GetValue(*baseReference*).
3. Let *propertyNameReference* be the result of evaluating *Expression*.
4. Let *propertyNameValue* be GetValue(*propertyNameReference*).
5. Call CheckObjectCoercible(*baseValue*).
6. Let *propertyNameString* be ToString(*propertyNameValue*).
7. Return a value of type Reference whose base value is *baseValue* and whose referenced name is *propertyNameString*.

Mark S. Miller 1/19/09 8:09 PM
Deleted: <#>If the syntactic production that is being evaluated is contained in strict mode code, let *strict* be **true**, else let *strict* be **false**.

Mark S. Miller 1/19/09 8:09 PM
Deleted: , and whose strict mode flag is *strict*

The production *CallExpression* **:** *CallExpression* **[** *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:

1. Evaluate *NewExpression*.
2. Call GetValue(Result(1)).
3. If Type(Result(2)) is not Object, throw a **TypeError** exception.
4. If Result(2) does not implement the internal [[Construct]] method, throw a **TypeError** exception.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

5. Call the [[Construct]] method on Result(2), providing no arguments (that is, an empty list of arguments).
6. Return GetValue(Result(5)).

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

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *Arguments*, producing an internal list of argument values (11.2.4).
4. If Type(Result(2)) is not Object, throw a **TypeError** exception.
5. If Result(2) does not implement the internal [[Construct]] method, throw a **TypeError** exception.
6. Call the [[Construct]] method on Result(2), providing the list Result(3) as the argument values.
7. Return GetValue(Result(6)).

11.2.3 Function Calls

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

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *Arguments*, producing an internal list of argument values (see 11.2.4).
4. If Type(Result(2)) is not Object, throw a **TypeError** exception.
5. If IsCallable(Result(2)) is **false**,, throw a **TypeError** exception.
6. If Type(Result(1)) is Reference, and IsPropertyReference(Result(1)) is **true**, Result(6) is GetBase(Result(1)). Otherwise, Result(6) is **null**.
7. Call the [[Call]] method on Result(2), providing Result(6) as the **this** value and providing the list Result(3) as the argument values.
8. Return GetValue(Result(7)).

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

11.2.4 Argument Lists

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

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

1. Return an empty internal list of values.

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

1. Evaluate *ArgumentList*.
2. Return Result(1).

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

1. Evaluate *AssignmentExpression*.
2. Call GetValue(Result(1)).
3. Return an internal list whose sole item is Result(2).

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

1. Evaluate *ArgumentList*.
2. Evaluate *AssignmentExpression*.
3. Call GetValue(Result(2)).
4. 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.

11.2.5 Function Expressions

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

1. Evaluate *FunctionExpression*.

Mark S. Miller 1/19/09 8:30 PM
Deleted: *NOTE*
Result(8) will never be of type Reference if Result(2) is a native ECMAScript object. Whether calling a host object can return a value of type Reference is implementation-dependent. If a value of type Reference is returned, it must be a non-strict Property Reference.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. Return Result(1).

11.3 Postfix Expressions

Syntax

PostfixExpression **:**
  LeftHandSideExpression
  LeftHandSideExpression [no *LineTerminator* here] `++`
  LeftHandSideExpression [no *LineTerminator* here] `--`

11.3.1 Postfix Increment Operator

The production *PostfixExpression* **:** *LeftHandSideExpression* [no *LineTerminator* here] `++` is evaluated as follows:

1. Evaluate LeftHandSideExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Add the value `1` to Result(3), using the same rules as for the `+` operator (see 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(3).

11.3.2 Postfix Decrement Operator

The production *PostfixExpression* **:** *LeftHandSideExpression* [no *LineTerminator* here] `--` is evaluated as follows:

1. Evaluate LeftHandSideExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Subtract the value `1` from Result(3), using the same rules as for the `-` operator (11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. 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:

1. Let *ref* be the result of evaluating UnaryExpression.
2. If Type(*ref*) is not Reference, return **true**.
3. If UnresolvableReference(*ref*) retrun **true**.
4. If IsPropertyReference(*ref*) is **true**, then
   a. Return the result of calling the [[Delete]] internal method on ToObject(GetBase(*ref*)) providing GetReferencedName(*ref*) as the argument.
5. Else throw a **ReferenceError** exception.

NOTE

*If the property to be deleted has the attribute { [[Configurable]]: **false** }, a **TypeError** exception is thrown.*

Mark S. Miller 1/19/09 8:32 PM
Deleted: and IsStrictReference(*ref*)

Mark S. Miller 1/19/09 8:32 PM
Deleted: s

Mark S. Miller 1/19/09 8:32 PM
Deleted: , *ref* is a Reference to an Environment Record binding, so
If IsStrictReference(*ref*) is **true**

Mark S. Miller 1/19/09 8:57 PM
Comment: After verifying that this is the behavior of [[Delete]], remove this note.

Mark S. Miller 1/19/09 8:33 PM
Deleted: <#>If GetBase(*ref*) is a declarative environment record, return **false**.
<#>Let *obj* be the binding object of the object environment record that is the value of GetBase(*ref*).
<#>Return the result of calling the [[Delete]] internal method on *obj*, providing GetReferencedName(*ref*) and **false** as the arguments.

Mark S. Miller 1/19/09 8:34 PM
Deleted: *When a* `delete` *operator occurs within strict mode code, a* ***ReferenceError*** *exception is thrown if its UnaryExpression is a direct reference to a variable, function argument, or function name. In addition, i*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

11.4.2 The `void` Operator

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

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

11.4.3 The `typeof` Operator

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

1. Let *val* be the result of evaluating *UnaryExpression*.
2. If Type(*val*) is not Reference, then
    a. If IsUnresolvableReference(*val*) is **true**, return `"undefined"`.
    b. Let *val* be GetValue(*val*).
3. Return a string determined by Type(*val*) 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 or host and implements [[Call]])	`"function"`
Object (host and doesn't implement [[Call]])	Implementation-defined

Mark S. Miller 1/19/09 8:57 PM
Comment: tighten the spec, probably to "object"

11.4.4 Prefix Increment Operator

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Add the value `1` to Result(3), using the same rules as for the **+** operator (see 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(4).

11.4.5 Prefix Decrement Operator

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Subtract the value `1` from Result(3), using the same rules as for the **-** operator (see 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. 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:

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

3. Call ToNumber(Result(2)).
4. 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:

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. If Result(3) is **NaN**, return **NaN**.
5. Negate Result(3); that is, compute a number with the same magnitude but opposite sign.
6. Return Result(5).

11.4.8 Bitwise NOT Operator (~)

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

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

11.4.9 Logical NOT Operator (!)

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

1. Evaluate *UnaryExpression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **true**, return **false**.
5. Return **true**.

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:

1. Evaluate MultiplicativeExpression.
2. Call GetValue(Result(1)).
3. Evaluate UnaryExpression.
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the specified operation (*, /, or %) to Result(5) and Result(6). See the notes below (11.5.1, 11.5.2, 11.5.3).
8. 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 SES, because of finite precision.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 SES language requires support of gradual underflow as defined by IEEE 754.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

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. SES 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:

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

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. The sign is determined by the rule already stated above.

Division of a zero by a zero results in **NaN**; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.

Division of a 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, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The SES language requires support of gradual underflow as defined by IEEE 754.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

11.5.3 Applying the % Operator

The % operator yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

NOTE
In C and C++, the remainder operator accepts only integral operands; in SES, it also accepts floating-point operands.

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

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 behaviour is not analogous to that of the usual integer remainder operator. Instead the SES language defines % on floating-point

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 SES floating-point remainder operation is determined by the rules of IEEE arithmetic:

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

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:

1. Evaluate AdditiveExpression.
2. Call GetValue(Result(1)).
3. Evaluate MultiplicativeExpression.
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. If Type(Result(5)) is String or Type(Result(6)) is String, then
    a. Call ToString(Result(5)).
    b. Call ToString(Result(6)).
    c. Concatenate Result(7a) followed by Result(7b).
    d. Return Result(7c).
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply the addition operation to Result(8) and Result(9). See the note below (11.6.3).
11. Return Result(10).

NOTE
No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native SES objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

11.6.2 The Subtraction Operator (-)

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

1. Evaluate AdditiveExpression.
2. Call GetValue(Result(1)).
3. Evaluate MultiplicativeExpression.
4. Call GetValue(Result(3)).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the subtraction operation to Result(5) and Result(6). See the note below (11.6.3).
8. 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. The SES language requires support of gradual underflow as defined by IEEE 754.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

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*

11.7.1 The Left Shift Operator (<<)

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

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

1. Evaluate *ShiftExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *AdditiveExpression*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToUint32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Left shift Result(5) by Result(7) bits. The result is a signed 32 bit integer.
9. Return Result(8).

11.7.2 The Signed Right Shift Operator (>>)

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. Evaluate *ShiftExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *AdditiveExpression*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToUint32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform sign-extending right shift of Result(5) by Result(7) bits. The most significant bit is propagated. The result is a signed 32 bit integer.
9. Return Result(8).

11.7.3 The Unsigned Right Shift Operator (>>>)

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

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

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

11.8 Relational Operators

Syntax

RelationalExpression **:**
ShiftExpression
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*
RelationalExpression **`instanceof`** *ShiftExpression*
RelationalExpression **`in`** *ShiftExpression*

RelationalExpressionNoIn **:**
ShiftExpression
RelationalExpressionNoIn **<** *ShiftExpression*
RelationalExpressionNoIn **>** *ShiftExpression*
RelationalExpressionNoIn **<=** *ShiftExpression*
RelationalExpressionNoIn **>=** *ShiftExpression*
RelationalExpressionNoIn **`instanceof`** *ShiftExpression*

NOTE
The 'NoIn' variants are needed to avoid confusing the `in` operator in a relational expression with the `in` operator in a `for` statement.

Semantics

The result of evaluating a relational operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

The *RelationalExpressionNoIn* productions are evaluated in the same manner as the *RelationalExpression* productions except that the contained *RelationalExpressionNoIn* is evaluated instead of the contained *RelationalExpression*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

11.8.1 The Less-than Operator (<)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(2) < Result(4). (see 11.8.5)
6. If Result(6) is **undefined**, return **false**. Otherwise, return Result(6).

11.8.2 The Greater-than Operator (>)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with LeftFirst equal to **false**. (see 11.8.5).
6. If Result(6) is **undefined**, return **false**. Otherwise, return Result(6).

11.8.3 The Less-than-or-equal Operator (<=)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with LeftFirst equal to **false**. (see 11.8.5).
6. If Result(6) is **true** or **undefined**, return **false**. Otherwise, return **true**.

11.8.4 The Greater-than-or-equal Operator (>=)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(2) < Result(4). (see 11.8.5).
6. If Result(6) is **true** or **undefined**, return **false**. Otherwise, return **true**.

11.8.5 The Abstract Relational Comparison Algorithm

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

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

1. If the *LeftFirst* flag is true, then
    a. Let *px* be the result of calling ToPrimitive(x, hint Number).
    b. Let *py* be the result of calling ToPrimitive(y, hint Number).
2. Else the order of evaluation needs to be reversed to preserve left to right evaluation
    a. Let *py* be the result of calling ToPrimitive(y, hint Number).
    b. Let *px* be the result of calling ToPrimitive(x, hint Number).
3. If Type(*px*) is String or Type(*py*) 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*.)
4. Let *nx* be the result of calling ToNumber(*px*). Because of *px* and *py* are primitive values evaluation order is not important.
5. Let ny be the result of calling ToNumber(*py*).
6. If *nx* is **NaN**, return **undefined**.

pratapL 1/19/09 8:57 PM
Comment: See Deviations doc item 2.8

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

7. If *ny* is **NaN**, return **undefined**.
8. If *nx* and *ny* are the same number value, return **false**.
9. If *nx* is **+0** and *ny* is **−0**, return **false**.
10. If *nx* is **−0** and *ny* is **+0**, return **false**.
11. If *nx* is +∞, return **false**.
12. If *ny* is +∞, return **true**.
13. If *ny* is **−∞**, return **false**.
14. If *nx* is **−∞**, return **true**.
15. If the mathematical value of *nx* is less than the mathematical value of *ny* —note that these mathematical values are both finite and not both zero—return **true**. Otherwise, return **false**.
16. If *py* is a prefix of *px*, 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.)
17. If *px* is a prefix of *py*, return **true**.
18. Let *k* be the smallest nonnegative integer such that the character at position *k* within *px* is different from the character at position *k* within *py*. (There must be such a *k*, for neither string is a prefix of the other.)
19. Let *m* be the integer that is the code unit value for the character at position *k* within *px*.
20. Let *n* be the integer that is the code unit value for the character at position *k* within *py*.
21. If *m* < *n*, return **true**. Otherwise, return **false**.

NOTE
The comparison of strings uses a simple lexicographic ordering on sequences of code unit value values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore strings that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both strings are already in normalised form.

11.8.6 The instanceof operator

The production *RelationalExpression*: *RelationalExpression* `instanceof` *ShiftExpression* is evaluated as follows:

1. Evaluate *RelationalExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *ShiftExpression*.
4. Call GetValue(Result(3)).
5. If Result(4) is not an object, throw a **TypeError** exception.
6. If Result(4) does not have a [[HasInstance]] method, throw a **TypeError** exception.
7. Call the [[HasInstance]] method of Result(4) with parameter Result(2).
8. Return Result(7).

11.8.7 The in operator

The production *RelationalExpression* **:** *RelationalExpression* `in` *ShiftExpression* is evaluated as follows:

1. Evaluate *RelationalExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *ShiftExpression*.
4. Call GetValue(Result(3)).
5. If Result(4) is not an object, throw a **TypeError** exception.
6. Call ToString(Result(2)).
7. Call the [[HasProperty]] method of Result(4) with parameter Result(6).
8. Return Result(7).

11.9 Equality Operators

Syntax

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

EqualityExpression **:**
RelationalExpression
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*

EqualityExpressionNoIn **:**
RelationalExpressionNoIn
EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

Semantics

The result of evaluating an equality operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

The *EqualityExpressionNoIn* productions are evaluated in the same manner as the *EqualityExpression* productions except that the contained *EqualityExpressionNoIn* and *RelationalExpressionNoIn* are evaluated instead of the contained *EqualityExpression* and *RelationalExpression*, respectively.

11.9.1 The Equals Operator (==)

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

1. Evaluate *EqualityExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *RelationalExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (see 11.9.3).
6. Return Result(5).

11.9.2 The Does-not-equals Operator (!=)

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

1. Evaluate *EqualityExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *RelationalExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (see 11.9.3).
6. If Result(5) is **true**, return **false**. Otherwise, return **true**.

11.9.3 The Abstract Equality Comparison Algorithm

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), go to step 14.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is not Number, go to step 11.
5. If *x* is **NaN**, return **false**.
6. If *y* is **NaN**, return **false**.
7. If *x* is the same number value as *y*, return **true**.
8. If *x* is **+0** and *y* is **−0**, return **true**.
9. If *x* is **−0** and *y* is **+0**, return **true**.
10. Return **false**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

11. 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**.
12. If Type(x) is Boolean, return **true** if x and y are both **true** or both **false**. Otherwise, return **false**.
13. Return **true** if x and y refer to the same object. Otherwise, return **false**.
14. If x is **null** and y is **undefined**, return **true**.
15. If x is **undefined** and y is **null**, return **true**.
16. If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y).
17. If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y.
18. If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y.
19. If Type(y) is Boolean, return the result of the comparison x == ToNumber(y).
20. If Type(x) is either String or Number and Type(y) is Object, return the result of the comparison x == ToPrimitive(y).
21. If Type(x) is Object and Type(y) is either String or Number, return the result of the comparison ToPrimitive(x) == y.
22. Return **false**.

NOTE
Given the above definition of equality:

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:

`A != B` *is equivalent to* `!(A == B)`.

`A == B` *is equivalent to* `B == A`*, except in the order of evaluation of* `A` *and* `B`.

The equality operator is not always transitive. For example, there might be two distinct String objects, each representing the same string value; each String object would be considered equal to the string value by the `==` *operator, but the two String objects would not be equal to each other.*

Comparison of strings uses a simple equality test on sequences of code unit value values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode 2.0 specification. Therefore strings that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both strings are already in normalised form.

11.9.4 The Strict Equals Operator (===)

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

1. Evaluate *EqualityExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *RelationalExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) === Result(2). (See below.)
6. Return Result(5).

11.9.5 The Strict Does-not-equal Operator (!==)

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

1. Evaluate *EqualityExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *RelationalExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) === Result(2). (See below.)

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

6. If Result(5) is **true**, return **false**. Otherwise, return **true**.

11.9.6 The Strict Equality Comparison Algorithm

The comparison *x* === *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is not Number, go to step 11.
5. If *x* is **NaN**, return **false**.
6. If *y* is **NaN**, return **false**.
7. If *x* is the same number value as *y*, return **true**.
8. If *x* is +0 and *y* is −0, return **true**.
9. If *x* is −0 and *y* is +0, return **true**.
10. Return **false**.
11. 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**.
12. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
13. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.

11.10 Binary Bitwise Operators

Syntax

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

BitwiseANDExpressionNoIn **:**
    EqualityExpressionNoIn
    BitwiseANDExpressionNoIn **&** *EqualityExpressionNoIn*

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

BitwiseXORExpressionNoIn **:**
    BitwiseANDExpressionNoIn
    BitwiseXORExpressionNoIn **^** *BitwiseANDExpressionNoIn*

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

BitwiseORExpressionNoIn **:**
    BitwiseXORExpressionNoIn
    BitwiseORExpressionNoIn **|** *BitwiseXORExpressionNoIn*

Semantics

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

11.11 Binary Logical Operators

Syntax

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

LogicalANDExpressionNoIn **:**
  BitwiseORExpressionNoIn
  LogicalANDExpressionNoIn `&&` *BitwiseORExpressionNoIn*

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

LogicalORExpressionNoIn **:**
  LogicalANDExpressionNoIn
  LogicalORExpressionNoIn `||` *LogicalANDExpressionNoIn*

Semantics

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

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

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

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

The *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are evaluated in the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that the contained *LogicalANDExpressionNoIn*, *BitwiseORExpressionNoIn* and *LogicalORExpressionNoIn* are evaluated instead of the contained *LogicalANDExpression*, *BitwiseORExpression* and *LogicalORExpression*, respectively.

NOTE
The value produced by a `&&` *or* `||` *operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.*

11.12 Conditional Operator (? :)

Syntax

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

ConditionalExpressionNoIn **:**
LogicalORExpressionNoIn
LogicalORExpressionNoIn **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

Semantics

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

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

The *ConditionalExpressionNoIn* production is evaluated in the same manner as the *ConditionalExpression* production except that the contained *LogicalORExpressionNoIn*, *AssignmentExpression* and *AssignmentExpressionNoIn* are evaluated instead of the contained *LogicalORExpression*, first *AssignmentExpression* and second *AssignmentExpression*, respectively.

NOTE
The grammar for a ConditionalExpression in SES is a little bit different from that in C and Java, which each allow the second subexpression to be an Expression but restrict the third expression to be a ConditionalExpression. The motivation for this difference in SES is to allow an assignment expression to be governed by either arm of a conditional and to eliminate the confusing and fairly useless case of a comma expression as the centre expression.

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

11.13 Assignment Operators

Syntax

AssignmentExpression **:**
ConditionalExpression
LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn **:**
ConditionalExpressionNoIn
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

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

Semantics

The *AssignmentExpressionNoIn* productions are evaluated in the same manner as the *AssignmentExpression* productions except that the contained *ConditionalExpressionNoIn and AssignmentExpressionNoIn* are evaluated instead of the contained *ConditionalExpression* and *AssignmentExpression*, respectively.

11.13.1 Simple Assignment (=)

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

NOTE
*If the LeftHandSide evaluates to an unresolvable reference, a **ReferenceError** exception is thrown upon assignment. The LeftHandSide also may not be a reference to a property with the attribute value {[[Writable]]:**false**} nor to a non-existent property of an object whose [[Extensible]] property has the value **false**. In these cases a **TypeError** exception is thrown.*

Mark S. Miller 1/19/09 8:40 PM
Deleted: *When an assignment occurs within strict mode code, its*

Mark S. Miller 1/19/09 8:40 PM
Deleted: *must not*

Mark S. Miller 1/19/09 8:40 PM
Deleted: *. If it does*

11.13.2 Compound Assignment (`op=`)

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

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

NOTE
See NOTE 11.13.1.

11.14 Comma Operator (,)

Syntax

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

ExpressionNoIn **:**
    AssignmentExpressionNoIn
    ExpressionNoIn **,** *AssignmentExpressionNoIn*

Semantics

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

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

The *ExpressionNoIn* production is evaluated in the same manner as the *Expression* production except that the contained *ExpressionNoIn* and *AssignmentExpressionNoIn* are evaluated instead of the contained *Expression* and *AssignmentExpression*, respectively.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

12 Statements

Syntax

Statement **:**
    Block
    VariableStatement
    EmptyStatement
    ExpressionStatement
    IfStatement
    IterationStatement
    ContinueStatement
    BreakStatement
    ReturnStatement
    LabelledStatement
    SwitchStatement
    ThrowStatement
    TryStatement
    DebuggerStatement

Mark S. Miller 1/19/09 8:41 PM
Deleted: *WithStatement*

Semantics

A *Statement* can be part of a *LabelledStatement*, which itself can be part of a *LabelledStatement*, and so on. The labels introduced this way are collectively referred to as the "current label set" when describing the semantics of individual statements. A *LabelledStatement* has no semantic meaning other than the introduction of a label to a *label set*. The label set of an *IterationStatement* or a *SwitchStatement* initially contains the single element **empty**. The label set of any other statement is initially empty.

Note:
TBD: Implementations have been known to support *FunctionDeclaration* in a *Statement*; however there is no uniform support. It is impossible to reconcile their differing semantics, and hence this specification excludes their possibility.

12.1 Block

Syntax

Block **:**
    { $StatementList_{opt}$ **}**

StatementList **:**
    Statement
    StatementList Statement

Semantics

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

1. Return (**normal**, **empty**, **empty**).

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

1. Evaluate *StatementList*.
2. Return Result(1).

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

1. Evaluate *Statement*.
2. If an exception was thrown, return (**throw**, *V*, empty) where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
3. Return Result(1).

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

1. Evaluate *StatementList*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. If Result(1) is an abrupt completion, return Result(1).
3. Evaluate *Statement*.
4. If an exception was thrown, return (**throw**, *V*, empty) where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
5. If Result(3).value is **empty**, let *V* = Result(1).value, otherwise let V = Result(3).value.
6. Return (Result(3).type, *V*, Result(3).target).

12.2 Variable statement

Syntax

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

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

VariableDeclarationListNoIn **:**
    VariableDeclarationNoIn
    VariableDeclarationListNoIn **`,`** *VariableDeclarationNoIn*

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

VariableDeclarationNoIn **:**
    Identifier InitialiserNoIn$_{opt}$

Initialiser **:**
    = *AssignmentExpression*

InitialiserNoIn **:**
    = *AssignmentExpressionNoIn*

Description

A variable statement declares variables that are created as defined in section 10.6. Variables are initialised to **undefined** when created. A variable with an *Initialiser* is assigned the value of its *AssignmentExpression* when the *VariableStatement* is executed, not when the variable is created.

Semantics

The production *VariableStatement* **:** **`var`** *VariableDeclarationList* **`;`** is evaluated as follows:

1. Evaluate *VariableDeclarationList*.
2. Return (**normal**, **empty**, **empty**).

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

1. Evaluate *VariableDeclaration*.

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

1. Evaluate *VariableDeclarationList*.
2. Evaluate *VariableDeclaration*.

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

1. Return a string value containing the same sequence of characters as in the *Identifier*.

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

1. Let *rhs* be the result of evaluating *Initialiser*.

Mark S. Miller 1/19/09 8:43 PM
Deleted: <#>If the *VariableDeclaration* occurs in strict mode code, let *strict* be **true**, otherwise let *strict* be **false**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. Let *value* be GetValue(*rhs*).
3. Call the SetMutableBinding(*N*,*V*) concrete method of the execution context's VariableEnvironment passing the *Identifer* and *value* as arguments.
4. Return a string value containing the same sequence of characters as in the *Identifier*.

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

1. Evaluate *AssignmentExpression*.
2. Return Result(1).

The *VariableDeclarationListNoIn*, *VariableDeclarationNoIn* and *InitialiserNoIn* productions are evaluated in the same manner as the *VariableDeclarationList*, *VariableDeclaration* and *Initialiser* productions except that the contained *VariableDeclarationListNoIn*, *VariableDeclarationNoIn*, *InitialiserNoIn* and *AssignmentExpressionNoIn* are evaluated instead of the contained *VariableDeclarationList*, *VariableDeclaration*, *Initialiser* and *AssignmentExpression*, respectively.

Mark S. Miller 1/19/09 8:43 PM
Deleted: ,*S*

Mark S. Miller 1/19/09 8:43 PM
Deleted: ,

Mark S. Miller 1/19/09 8:43 PM
Deleted: , and *strict*

Mark S. Miller 1/19/09 8:57 PM
Comment: Need at least a comment explaining why. Better would be to remove this wart.

12.3 Empty Statement

Syntax

EmptyStatement **:**
  ;

Semantics

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

1. Return (**normal**, **empty**, **empty**).

12.4 Expression Statement

Syntax

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

Note that an *ExpressionStatement* cannot start with an opening curly brace because that might make it ambiguous with a *Block*. Also, an *ExpressionStatement* cannot start with the **function** keyword because that might make it ambiguous with a *FunctionDeclaration*.

Semantics

The production *ExpressionStatement* **:** [lookahead ∉ {**{**, **function**}] *Expression* **;** is evaluated as follows:

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

12.5 The `if` Statement

Syntax

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

Each **else** for which the choice of associated **if** is ambiguous shall be associated with the nearest possible **if** that would otherwise have no corresponding **else**.

Semantics

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

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

4. If Result(3) is **false**, go to step 7.
5. Evaluate the first *Statement*.
6. Return Result(5).
7. Evaluate the second *Statement*.
8. Return Result(7).

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

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, return (**normal**, **empty**, **empty**).
5. Evaluate *Statement*.
6. Return Result(5).

12.6 Iteration Statements

Mark S. Miller 1/19/09 8:45 PM
Deleted: 12.5.1 Strict Mode Restrictions
In strict mode code a *Statement* that is part of an *IfStatement* production may not be a *VariableStatement* nor may it be a *LabelledStatment* whose *Statement* production is a *VariableStatment*. The *LabelledStatement* restriction also applies if such an *VariableStatment* is preceeded by multiple labels.

An iteration statement consists of a *header* (which consists of a keyword and a parenthesised control construct) and a *body* (which consists of a *Statement*).

Syntax

IterationStatement **:**
    `do` *Statement* **`while (`** *Expression* **`);`**
    `while (` *Expression* **`)`** *Statement*
    `for (`*ExpressionNoIn*$_{opt}$**`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for ( var` *VariableDeclarationListNoIn* **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for (` *LeftHandSideExpression* **`in`** *Expression* **`)`** *Statement*
    `for ( var` *VariableDeclarationNoIn* **`in`** *Expression* **`)`** *Statement*

12.6.1 The `do-while` Statement

Mark S. Miller 1/19/09 8:45 PM
Deleted: Strict Mode Restrictions
A *Statement* that is an element of an *IterationStatement* production may not be a *VariableStatment* nor may it be a *LabelledStatment* whose *Statement* production is a *VariableStatment*. The *LabelledStatement* restriction also applies if such an *VariableStatment* is preceeded by multiple labels.

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

1. Let *V* = **empty**.
2. Evaluate *Statement*.
3. If Result(2).value is not **empty**, let *V* = Result(2).value.
4. If Result(2).type is **continue** and Result(2).target is in the current label set, go to step 7.
5. If Result(2).type is **break** and Result(2).target is in the current label set, return (**normal**, *V*, **empty**).
6. If Result(2) is an abrupt completion, return Result(2).
7. Evaluate *Expression*.
8. Call GetValue(Result(7)).
9. Call ToBoolean(Result(8)).
10. If Result(9) is **true**, go to step 2.
11. Return (**normal**, *V*, **empty**);

12.6.2 The `while` statement

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

1. Let *V* = **empty**.
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. Call ToBoolean(Result(3)).
5. If Result(4) is **false**, return (**normal**, *V*, **empty**).
6. Evaluate *Statement*.
7. If Result(6).value is not **empty**, let *V* = Result(6).value.
8. If Result(6).type is **continue** and Result(6).target is in the current label set, go to 2.
9. If Result(6).type is **break** and Result(6).target is in the current label set, return (**normal**, *V*, **empty**).
10. If Result(6) is an abrupt completion, return Result(6).
11. Go to step 2.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

12.6.3 The `for` Statement

The production *IterationStatement* **:** **`for (`***ExpressionNoIn$_{opt}$* **`;`** *Expression$_{opt}$* **`;`** *Expression$_{opt}$***`)`** *Statement* is evaluated as follows:

1. If *ExpressionNoIn* is not present, go to step 4.
2. Evaluate *ExpressionNoIn*.
3. Call GetValue(Result(2)). (This value is not used.)
4. Let *V* = **empty**.
5. If the first *Expression* is not present, go to step 10.
6. Evaluate the first *Expression*.
7. Call GetValue(Result(6)).
8. Call ToBoolean(Result(7)).
9. If Result(8) is **false**, go to step 19.
10. Evaluate *Statement*.
11. If Result(10).value is not **empty**, let *V* = Result(10).value
12. If Result(10).type is **break** and Result(10).target is in the current label set, go to step 19.
13. If Result(10).type is **continue** and Result(10).target is in the current label set, go to step 15.
14. If Result(10) is an abrupt completion, return Result(10).
15. If the second *Expression* is not present, go to step 5.
16. Evaluate the second *Expression*.
17. Call GetValue(Result(16). (This value is not used.)
18. Go to step 5.
19. Return (**normal**, *V*, **empty**).

The production *IterationStatement* **:** **`for ( var`** *VariableDeclarationListNoIn* **`;`** *Expression$_{opt}$* **`;`** *Expression$_{opt}$* **`)`** *Statement* is evaluated as follows:

1. Evaluate *VariableDeclarationListNoIn*.
2. Let V = **empty**.
3. If the first *Expression* is not present, go to step 8.
4. Evaluate the first *Expression*.
5. Call GetValue(Result(4)).
6. Call ToBoolean(Result(5)).
7. If Result(6) is **false**, go to step 17.
8. Evaluate *Statement*.
9. If Result(8).value is not **empty**, let V = Result(8).value.
10. If Result(8).type is **break** and Result(8).target is in the current label set, go to step 17.
11. If Result(8).type is **continue** and Result(8).target is in the current label set, go to step 13.
12. If Result(8) is an abrupt completion, return Result(8).
13. If the second *Expression* is not present, go to step 3.
14. Evaluate the second *Expression*.
15. Call GetValue(Result(14)). (This value is not used.)
16. Go to step 3.
17. Return (**normal**, V, **empty**).

12.6.4 The `for-in` Statement

Mark S. Miller 1/19/09 8:57 PM
Comment: When we can, tighten this with a deterministic enumeration order.

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

1. Evaluate the *Expression*.
2. Call GetValue(Result(1)).
3. If Result(2) is **null** or **undefined**, return (**normal**, V, **empty**).
4. Call ToObject(Result(2)).
5. Let *V* = **empty**.
6. Get the name of the next property of Result(4) whose [[Enumerable]] attribute is **true**. If there is no such property, go to step 15.
7. Evaluate the *LeftHandSideExpression* (it may be evaluated repeatedly).
8. Call PutValue(Result(7), Result(6)).
9. Evaluate *Statement*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

10. If Result(9).value is not **empty**, let *V* = Result(9).value.
11. If Result(9).type is **break** and Result(9).target is in the current label set, go to step 15.
12. If Result(9).type is **continue** and Result(9).target is in the current label set, go to step 6.
13. If Result(9) is an abrupt completion, return Result(9).
14. Go to step 6.
15. Return (**normal**, *V*, **empty**).

The production *IterationStatement* **:** `for ( var` *VariableDeclarationNoIn* `in` *Expression* `)` *Statement* is evaluated as follows:

1. Evaluate *VariableDeclarationNoIn*.
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. If Result(3) is **null** or **undefined**, return (**normal**, V, **empty**).
5. Call ToObject(Result(3)).
6. Let V = **empty**.
7. Get the name of the next property of Result(5) whose [[Enumerable]] attribute is **true**. If there is no such property, go to step 16.
8. Evaluate Result(1) as if it were an Identifier; see step 7 from the previous algorithm (it may be evaluated repeatedly).
9. Call PutValue(Result(7), Result(8)).
10. Evaluate *Statement*.
11. If Result(10).value is not **empty**, let V = Result(10).value.
12. If Result(10).type is **break** and Result(10).target is in the current label set, go to step 16.
13. If Result(10).type is **continue** and Result(10).target is in the current label set, go to step 7.
14. If Result(9) is an abrupt completion, return Result(9).
15. Go to step 7.
16. Return (**normal**, V, **empty**).

pratapL 1/19/09 8:57 PM
Comment: From AWB:

I don't see what value this phrase adds. It just adds a spot where step numbering can get out of whack.

The mechanics of enumerating the properties (step 5 in the first algorithm, step 6 in the second) is not specified. 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 guaranteed not to be visited in the active enumeration.

Enumerating the properties of an object includes enumerating properties of its parent, and the parent of the parent, and so on, recursively; but a property of a parent is not enumerated if it is "shadowed" because some previous object in the inheritance chain has a property with the same name.

Mark S. Miller 1/19/09 8:47 PM
Deleted: prototype

Mark S. Miller 1/19/09 8:47 PM
Deleted: prototype

Mark S. Miller 1/19/09 8:47 PM
Deleted: prototype

Mark S. Miller 1/19/09 8:47 PM
Deleted: prototype

Mark S. Miller 1/19/09 8:47 PM
Deleted: prototype

NOTE
See NOTE 11.13.1.

12.7 The `continue` Statement

Syntax

ContinueStatement **:**
`continue` [no *LineTerminator* here] *Identifier$_{opt}$* `;`

Semantics

A program is considered syntactically incorrect if either of the following are true:

The program contains a `continue` statement without the optional *Identifier*, which is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.

The program contains a `continue` statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing (but not crossing function boundaries) *IterationStatement*.

A *ContinueStatement* without an *Identifier* is evaluated as follows:

1. Return (**continue**, **empty**, **empty**).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

A *ContinueStatement* with the optional *Identifier* is evaluated as follows:

1. Return (**continue**, **empty**, *Identifier*).

12.8 The `break` Statement

Syntax

BreakStatement **:**
    `break` [no *LineTerminator* here] *Identifier*$_{opt}$ **`;`**

Semantics

A program is considered syntactically incorrect if either of the following are true:

The program contains a **`break`** statement without the optional *Identifier*, which is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement* or a *SwitchStatement*.

The program contains a **`break`** statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing (but not crossing function boundaries) *Statement*.

A *BreakStatement* without an *Identifier* is evaluated as follows:

1. Return (**break**, **empty**, **empty**).

A *BreakStatement* with an *Identifier* is evaluated as follows:

1. Return (**break**, **empty**, *Identifier*).

12.9 The `return` Statement

Syntax

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

Semantics

An SES program is considered syntactically incorrect if it contains a **`return`** statement that is not within a *FunctionBody*. A **`return`** statement causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is **undefined**. Otherwise, the return value is the value of *Expression*.

The production *ReturnStatement* **:** **`return`** [no *LineTerminator* here] *Expression*$_{opt}$ **`;`** is evaluated as:

1. If the *Expression* is not present, return (**return**, **undefined**, **empty**).
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. Return (**return**, Result(3), **empty**).

12.10 N/A

12.11 The `switch` Statement

Syntax

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

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

CaseClauses **:**
    CaseClause
    CaseClauses *CaseClause*

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 8:48 PM
Deleted: The `with` Statement

Mark S. Miller 1/19/09 8:48 PM
Deleted: Syntax
WithStatement **:**
`with (` *Expression* **`)`** *Statement*
Description
The **`with`** statement adds a computed object environment record to the lexical environment of the current execution context, then executes a statement with this augmented scope chain, then restores the lexical environment.
Semantics
The production *WithStatement* **:** **`with (`** *Expression* **`)`** *Statement* is evaluated as follows:
<#>Let *val* be the result of evaluating *Expression*.
<#>Let *obj* be ToObject(GetValue(*val*).
<#>Let *oldEnv* be the running execution context's LexicalEnvironment.
<#>Let *newEnv* be the result of calling NewObjectEnvironmentRecord(O,E) passing *obj* and *oldEnv* as the arguments
<#>Set the running execution context's LexicalEnvironment to *newEnv*.
<#>Let *C* be the result of evaluating *Statement* but if an exception is thrown during the evaluation, let *C* be (**throw**, *V*, **empty**), where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
<#>Set the running execution context's Lexical Environment to *oldEnv*.
<#>Return *C*.
NOTE
No matter how control leaves the embedded 'Statement', whether normally or by some form of abrupt completion or exception, the LexicalEnvironment is always restored to its former state.
12.10.1 Strict Mode Restrictions
Strict mode code may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context is treated as a syntax error.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

Semantics

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

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Evaluate *CaseBlock*, passing it Result(2) as a parameter.
4. If Result(3).type is **break** and Result(3).target is in the current label set, return (**normal**, Result(3).value, **empty**).
5. Return Result(3).

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

1. Let *V* = **empty**.
2. Let *A* be the list of CaseClause items in source text order.
3. Let *C* be the next CaseClause in *A*. If there is no such CaseClause, then go to step 16.
4. Evaluate *C*.
5. If *input* is not equal to Result(4) as defined by the `!==` operator, then go to step 3.
6. If *C* does not have a StatementList, then go to step 10.
7. Evaluate *C*'s StatementList and let *R* be the result.
8. If *R* is an abrupt completion, then return *R*.
9. Let *V* = *R*.value.
10. Let *C* be the next CaseClause in *A*. If there is no such CaseClause, then go to step 16.
11. If *C* does not have a StatementList, then go to step 10.
12. Evaluate *C*'s StatementList and let *R* be the result.
13. If *R*.value is not **empty**, then let *V* = *R*.value.
14. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
15. Go to step 10.
16. Return (**normal**, *V*, **empty**).

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

1. Let *V* = **empty**.
2. Let *A* be the list of CaseClause items in the first CaseClauses, in source text order.
3. Let *C* be the next CaseClause in *A*. If there is no such CaseClause, then go to step 11.
4. Evaluate *C*.
5. If *input* is not equal to Result(4) as defined by the `!==` operator, then go to step 3.
6. If *C* does not have a StatementList, then go to step 20.
7. Evaluate *C*'s StatementList and let *R* be the result.
8. If *R* is an abrupt completion, then return *R*.
9. Let *V* = *R*.value.
10. Go to step 20.
11. Let *B* be the list of CaseClause items in the second CaseClauses, in source text order.
12. Let *C* be the next CaseClause in *B*. If there is no such CaseClause, then go to step 26.
13. Evaluate *C*.
14. If *input* is not equal to Result(13) as defined by the `!==` operator, then go to step 12.
15. If *C* does not have a StatementList, then go to step 31.
16. Evaluate *C*'s StatementList and let *R* be the result.
17. If *R* is an abrupt completion, then return *R*.
18. Let *V* = *R*.value.
19. Go to step 31.
20. Let *C* be the next CaseClause in *A*. If there is no such CaseClause, then go to step 26.
21. If *C* does not have a StatementList, then go to step 20.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

22. Evaluate *C*'s StatementList and let *R* be the result.
23. If *R*.value is not **empty**, then let *V* = *R*.value.
24. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
25. Go to step 20.
26. If the DefaultClause does not have a StatementList, then go to step 30.
27. Evaluate the DefaultClause's StatementList and let *R* be the result.
28. If *R*.value is not **empty**, then let *V* = *R*.value.
29. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
30. Let *B* be the list of CaseClause items in the second CaseClauses, in source text order.
31. Let *C* be the next CaseClause in *B*. If there is no such CaseClause, then go to step 37.
32. If *C* does not have a StatementList, then go to step 31.
33. Evaluate *C*'s StatementList and let *R* be the result.
34. If *R*.value is not **empty**, then let *V* = *R*.value.
35. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
36. Go to step 31.
37. Return (**normal**, *V*, **empty**).

The production *CaseClause* **:** `case` *Expression* **:** *StatementList*$_{opt}$ is evaluated as follows:

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

NOTE
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.

12.12 Labelled Statements

Mark S. Miller 1/19/09 8:49 PM
Deleted: **12.11.1 Strict Mode Restrictions**
A *Statement* that is an element of an *StatementList* that is part of a *CauseClause* or *DefaultClause* may not be a *VariableStatment* nor may it be a *LabelledStatment* whose *Statement* production is a *VariableStatment*. The *LabelledStatement* restriction also applies if such a *VariableStatment* is preceeded by multiple labels.

Syntax

LabelledStatement **:**
  Identifier **:** *Statement*

Semantics

A *Statement* may be prefixed by a label. Labelled statements are only used in conjunction with labelled `break` and `continue` statements. SES has no `goto` statement.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

An SES program is considered syntactically incorrect if it contains a *LabelledStatement* that is enclosed by a *LabelledStatement* with the same *Identifier* as label. This does not apply to labels appearing within the body of a *FunctionDeclaration* that is nested, directly or indirectly, within a labelled statement.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

The production *Identifier* **:** *Statement* is evaluated by adding *Identifier* to the label set of *Statement* and then evaluating *Statement*. If the *LabelledStatement* itself has a non-empty label set, these labels are also added to the label set of *Statement* before evaluating it. If the result of evaluating *Statement* is (**break**, *V*, *L*) where *L* is equal to *Identifier*, the production results in (**normal**, *V*, **empty**).

Prior to the evaluation of a *LabelledStatement*, the contained *Statement* is regarded as possessing an empty label set, except if it is an *IterationStatement* or a *SwitchStatement*, in which case it is regarded as possessing a label set consisting of the single element, **empty**.

12.13 The `throw` statement

Syntax

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

Semantics

The production *ThrowStatement* **:** `throw` [no *LineTerminator* here] *Expression* *;* is evaluated as:

1. Evaluate *Expression*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. Call GetValue(Result(1)).
3. Return (**throw**, Result(2), **empty**).

12.14 The `try` statement

Syntax

TryStatement **:**
  `try` *Block Catch*
  `try` *Block Finally*
  `try` *Block Catch Finally*

Catch **:**
  `catch (`*Identifier* **`)`** *Block*

Finally **:**
  `finally` *Block*

Description

The **`try`** statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a **`throw`** statement. The **`catch`** clause provides the exception-handling code. When a catch clause catches an exception, its *Identifier* is bound to a safe value representing that exception.

Mark S. Miller 1/19/09 8:57 PM
Comment: Need to define all the machinery for safe catching.

Semantics

The production *TryStatement* **: `try`** *Block Catch* is evaluated as follows:

1. Evaluate *Block*.
2. If Result(1).type is not **throw**, return Result(1).
3. Evaluate *Catch* with parameter Result(1).
4. Return Result(3).

The production *TryStatement* **: `try`** *Block Finally* is evaluated as follows:

1. Evaluate *Block*.
2. Evaluate *Finally*.
3. If Result(2) .*type* is **normal**, return Result(1).
4. Return Result(2).

The production *TryStatement* **: `try`** *Block Catch Finally* is evaluated as follows:

1. Evaluate *Block*.
2. Let *C* = Result(1).
3. If Result(1).type is not **throw**, go to step 6.
4. Evaluate *Catch* with parameter Result(1).
5. Let *C* = Result(4).
6. Evaluate *Finally*.
7. If Result(6).type is **normal**, return *C*.
8. Return Result(6).

The production *Catch* **: `catch (`***Identifier* **`)`** *Block* is evaluated as follows:

1. Let *C* be a safe value derived from the parameter that has been passed to this production.
2. Let *oldEnv* be the running execution context's LexicalEnvironment.
3. Let *catchEnv* be the result of calling NewDeclarativeEnvironmentRecord(E) passing *oldEnv* as the argument.
4. Call the CreateMutableBinding(*N*) concrete method of *catchEnv* passing the *Identifier* String value as the argument.
5. Call the SetMutableBinding(*N,V*) concrete method of *catchEnv* passing the *Identifer* and *C* as arguments.
6. Set the running execution context's LexicalEnvironment to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.

Mark S. Miller 1/19/09 8:57 PM
Comment: See above note.

Mark S. Miller 1/19/09 8:54 PM
Deleted: ,*S*

Mark S. Miller 1/19/09 8:54 PM
Deleted: ,

Mark S. Miller 1/19/09 8:54 PM
Deleted: , and **false**

Mark S. Miller 1/19/09 8:54 PM
Deleted: Note that the last argument is immaterial in this situation.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

8. Set the running execution context's LexicalEnvironment to *oldEnv*.
9. Return *B*.

The production *Finally* **:** **`finally`** *Block* is evaluated as follows:

1. Evaluate *Block*.
2. Return Result(1).

Mark S. Miller 1/19/09 8:57 PM
Comment: Consider adopting the following rule into SES: If Block terminates abruptly, abort further execution of all SES programs within this SES environment.

12.15 Debugger statement

Syntax

DebuggerStatement **:**
  `debugger ;`

Semantics

Evaluating the *DebuggerStatement* production may allow an implementation to cause a breakpoint when run under a debugger.

13 Function Definition

Syntax

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

FunctionExpression **:**
  `function` *Identifier$_{opt}$* **`(`** *FormalParameterList$_{opt}$* **`)`** **`{`** *FunctionBody* **`}`**

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

FunctionBody **:**
  SourceElements

Semantics

The production *FunctionDeclaration* **:** **`function`** *Identifier* **`(`** *FormalParameterList$_{opt}$* **`)`** **`{`** *FunctionBody* **`}`** is processed for function declarations as follows during Declaration Binding instantiation (10.3.3):

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList$_{opt}$*, and body specified by *FunctionBody*. Pass in the VariableEnvironment of the running execution context as the *Scope* and the string value of *Identifier* as *Name*.

Mark S. Miller 1/19/09 8:58 PM
Deleted: Pass in **true** as the *Strict* flag if the *FunctionDeclaration* is contained in strict code or if its *FunctionBody* is strict code.

The production *FunctionExpression* **:** **`function`** **`(`** *FormalParameterList$_{opt}$* **`)`** **`{`** *FunctionBody* **`}`** is evaluated as follows:

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList$_{opt}$* and body specified by *FunctionBody*. Pass in the LexicalEnvironment of the running execution context as the *Scope* and an empty string as *Name*.

Mark S. Miller 1/19/09 8:59 PM
Deleted: Pass in **true** as the *Strict* flag if the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code.

The production FunctionExpression **:** **`function`** Identifier **`(`** FormalParameterList$_{opt}$ **`)`** **`{`** FunctionBody **`}`** is evaluated as follows:

1. Let *funcEnv* be the result of calling NewDeclarativEnvironmentRecord(*E*) passing the running execution context's Lexical Environment as the argument
2. Let *envRec* be *funcEnv's* environment record.
3. Call the CreateImmutableBinding(N) concrete method of *envRec* passing the string value of *Identifier* as the argument.
4. Let *closure* be the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList$_{opt}$* and body specified by *FunctionBody*. Pass in *funcEnv* as the *Scope* and the string value of *Identifier* as *Name*.

Mark S. Miller 1/19/09 9:00 PM
Deleted: Pass in **true** as the *Strict* flag if the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

5. Call the InitializeImmutableBinding(*N,V*) concrete method of *envRec* passing the string value of *Identifier* and *closure* as the arguments.
6. Return *closure*.

It is a **SyntaxError** if any single *Identifier* value appears more than once within a *FormalParameterList*.

NOTE
The Identifier in a FunctionExpression can be referenced from inside the FunctionExpression's FunctionBody to allow the function to call itself recursively. However, unlike in a FunctionDeclaration, the Identifier in a FunctionExpression cannot be referenced from and does not affect the scope enclosing the FunctionExpression.

The production *FunctionBody* **:** *SourceElements* is evaluated as follows:

1. Process *SourceElements* for function declarations.
2. Evaluate *SourceElements*.
3. Return Result(3).

13.1 N/A

13.2 Creating Function Objects

Given an optional parameter list specified by *FormalParameterList*, a body specified by *FunctionBody*, a Lexical Environment specified by *Scope*, and a possibly empty string *Name*, a Function object is constructed as follows:

1. Create a new native SES object and let *F* be that object.
2. Set the [[Class]] internal property of *F* to **"Function"**.
3. Set the [[Parent]] internal property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
4. Set the [[Call]] internal property of *F* as described in 13.2.1.
5. Set the [[Construct]] internal property of *F* as described in 13.2.2.
6. Set the [[Scope]] internal property of *F* to the value of *Scope*.
7. Let *names* be a List containing, in left to right textual order, the strings corresponding to the identifiers of *FormalParameterList*.
8. Set the [[FormalParameters]] internal property of *F* to *names*.
9. Set the [[Code]] internal property of *F* to *FunctionBody*.
10. Set the **length** property of *F* to the number of formal parameters specified in *FormalParameterList*. If no parameters are specified, set the **length** property of *F* to 0. This property is given attributes as specified in 15.3.5.1.
11. Set the [[Extensible]] internal property of *F* to **true**.
12. Set the **name** property of *F* to *Name*. This property is given attributes as specified in 15.3.5.4.
13. Set the **arguments** property of *F* using the PropertyDescriptor {[[Value: **null**, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
14. Set the **caller** property of *F* using the PropertyDescriptor {[[Value: **null**, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
15. Return *F*.

13.2.1 [[Call]]

When the [[Call]] property for a Function object *F* is called with a this value and a list of arguments, the following steps are taken:

1. Let *funcCtx* be the result of establishing a new execution context for function code using *F*'s *FormalParameterList* and the passed arguments list *args*.
2. Let *result* be the result of evaluating *F*'s *FunctionBody*.
3. Exit the execution context *funcCtx*, restoring the previous execution context.
4. If *result*.type is **throw** then throw *result*.value.
5. If *result*.type is **return** then return *result*.value.
6. Otherwise *result*.type must be **normal**. Return **undefined**.

Mark S. Miller 1/19/09 9:00 PM
Deleted: of a strict mode *FunctionDeclaration* or *FunctionExpression*

Mark S. Miller 1/19/09 9:01 PM
Deleted: <#>The code of this *FunctionBody* is strict mode code if the first *SourceElement* within *SourceElements* is a *Statement* production that is an *ExpressionStatment* whose *Expression* consists solely of a *StringLiteral* whose value is a Use Strict Directive (14.1) or if any of the conditions in 10.1.1 apply. If the code of this *FunctionBody* is strict mode code *SourceElements* is processed and evaluated in the following steps as non-strict mode code.

Mark S. Miller 1/19/09 9:02 PM
Deleted: Definitions

Mark S. Miller 1/19/09 9:02 PM
Deleted: This section is no longer used.

Mark S. Miller 1/19/09 9:02 PM
Deleted: , a Boolean flag *Strict*

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:04 PM
Comment: Cajita freezes the function on first use or potentially escaping occurrence. Can SES do better? In any case, must remember to specify at least this.

Mark S. Miller 1/19/09 9:05 PM
Deleted: <#>Let *proto* be the result of creating a new object as would be constructed by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
<#>Set the **constructor** property of *proto* to *F*. This property has attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.
<#>Set the **prototype** property of *F* to *proto*. This property is given attributes as specified in 15.3.5.2.

Mark S. Miller 1/19/09 9:06 PM
Deleted: *NOTE*
A ***prototype*** *property is automatically created for every function, to allow for the possibility that the function will be used as a constructor.*

Mark S. Miller 1/19/09 9:06 PM
Deleted: ,

Mark S. Miller 1/19/09 9:06 PM
Deleted: , and the **this** value as described in 10.4.3

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

13.2.2 [[Construct]]

When the [[Construct]] property for a Function object *F* is called with a possibly empty list of arguments, the following steps are taken:

1. Let *result* be the result of calling the [[Call]] internal property of *F*, providing the argument list passed into [[Construct]] as *args*.
2. If Type(*result*) is Object then return *result*.
3. Else throw a **TypeError**.

Mark S. Miller 1/19/09 9:07 PM
Deleted: <#>Let *obj* be a newly created native ECMAScript object.
<#>Set the [[Class]] internal property of *obj* to `"Object"`.
<#>Set the [[Extensible]] internal property of *obj* to **true**.
<#>Let *proto* be the value of the `prototype` property of *F*.
<#>If *proto* is an Object, set the [[Prototype]] internal property of *obj* to *proto*.
<#>If *proto* is not an Object, set the [[Prototype]] internal property of *obj* to the standard built-in Object prototype object as described in 15.2.4.

Mark S. Miller 1/19/09 9:08 PM
Deleted: *obj* as the **this** value and providing

Mark S. Miller 1/19/09 9:09 PM
Deleted: Return *obj*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

14 Program

Syntax

Program **:**
SourceElements

SourceElements **:**
SourceElement
SourceElements SourceElement

SourceElement **:**
Statement
FunctionDeclaration

Semantics

The production *Program* **:** *SourceElements* is evaluated as follows:

1. Process *SourceElements* for function declarations.
2. Return the result of evaluating *SourceElements*.

The production *SourceElements* **:** *SourceElement* is processed for function declarations as follows:

1. Process *SourceElement* for function declarations.

The production *SourceElements* **:** *SourceElement* is evaluated as follows:

1. Evaluate *SourceElement*.
2. Return Result(1).

The production *SourceElements* **:** *SourceElements SourceElement* is processed for function declarations as follows:

1. Process *SourceElements* for function declarations.
2. Process *SourceElement* for function declarations.

The production *SourceElements* **:** *SourceElements SourceElement* is evaluated as follows:

1. Evaluate *SourceElements*.
2. If Result(1) is an abrupt completion, return Result(1)
3. Evaluate *SourceElement*.
4. Return Result(3).

The production *SourceElement* **:** *Statement* is processed for function declarations by taking no action.

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

1. Evaluate *Statement*.
2. Return Result(1).

The production *SourceElement* **:** *FunctionDeclaration* is processed for function declarations as follows:

1. Process *FunctionDeclaration* for function declarations (see clause 13).

The production *SourceElement* **:** *FunctionDeclaration* is evaluated as follows:

1. Return (**normal**, **empty**, **empty**).

Mark S. Miller 1/19/09 9:10 PM
Deleted: <#>The code of this *Program* is strict mode code if the first *SourceElement* within *SourceElements* is a *Statement* production that is an *ExpressionStatement* whose *Expression* consists solely of a *StringLiteral* whose value is the Use Strict Directive (14.1) or if any of the conditions of 10.1.1 apply. If the code of this *Program* is strict mode code, *SourceElements* is processed and evaluated in the following steps as strict mode code. Otherwise *SourceElements* is processed and evaluated in the following steps as non-strict mode code.

Mark S. Miller 1/19/09 9:12 PM
Deleted: 14.1 Use Strict Directive
A strict mode *Program* may be explicitly designated by the occurrence of a Use Strict Directive as the first *SourceElement* of the *Program*. A Use Strict Directive is coded as a *ExpressionStatement* that consists entirely of a *StringLiteral* whose value conforms to the *UseStrictDirective* grammar below.
To determine whether or not *StringLiteral* is a Use Strict Directive, the string value of the literal is converted to a sequence of lexical input elements (7) as if it was the source text of an ECMAScript program. If the sequence of input elements conform to the following grammar, the StringLiteral is a Use Strict Directive.

Syntax
UseStrictDirective **:**
[no *LineTerminator* here] **`use`** [no *LineTerminator* here] **`strict`** [no *LineTerminator* here] *useExtension*$_{opt}$
useExtension **:**
, *ArbitraryInputElements*$_{opt}$
ArbitraryInputElements **:**
[any sequence of lexical input elements that does not include any LineTerminator elements]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15 Native SES Objects

There are certain built-in objects available whenever an SES program begins execution. One, the global object, is part of the lexical environment of the executing program. Others are accessible as initial properties of the global object.

Unless specified otherwise, the [[Class]] property of a built-in object is **"Function"** if that built-in object has a [[Call]] property, or **"Object"** if that built-in object does not have a [[Call]] property. The global object, and all values reachable from the global object are frozen.

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. 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.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this section is given fewer arguments than the function is specified to require, the function or constructor shall behave exactly as if it had been given sufficient additional arguments, each such argument being the **undefined** value.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this section is given more arguments than the function is specified to allow, the behaviour of the function or constructor is to ignore the extra arguments.

NOTE
Implementations that add additional abilities to the set of built-in functions must do so by adding new functions rather than adding new parameters to existing functions.

Every built-in function and every built-in constructor has the Function prototype object, which is the initial value of the expression **Function.prototype** (15.3.2.1), as the value of its internal [[Parent]] property.

Every built-in prototype object has the Object prototype object as the value of its internal [[Parent]] property, except the Object prototype object itself.

None of the built-in functions described in this section shall implement the internal [[Construct]] method unless otherwise specified in the description of a particular function. 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 largest number of named arguments shown in the section headings for the function description, including optional parameters.

NOTE
For example, the Function object that is the initial value of the `slice` *property of the String prototype object is described under the section heading "String.prototype.slice (start , end)" which shows the two named arguments start and end; therefore the value of the* `length` *property of that Function object is 2.*

In every case, the **length** property of a built-in Function object described in this section has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** } (and no others). Every other property described in this section has the attribute { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** } unless otherwise specified.

15.1 The Global Object

The unique *global object* is created before control enters any execution context.

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.

The global object does not have a [[Call]] property; it is not possible to invoke the global object as a function.

The values of the [[Parent]] and [[Class]] properties of the global object are the Object prototype object, and "Object", respectively.

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 9:15 PM
Deleted: Unless specified otherwise, the [[Extensible]] property of a built-in

Mark S. Miller 1/19/09 9:15 PM
Deleted: object has the value **true**.

Mark S. Miller 1/19/09 9:16 PM
Deleted: undefined

Mark S. Miller 1/19/09 9:17 PM
Deleted: *cap*

Mark S. Miller 1/19/09 9:17 PM
Deleted: *are encouraged to*

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:18 PM
Deleted: ,

Mark S. Miller 1/19/09 9:18 PM
Deleted: which is the initial value of the expression **Object.prototype** (15.3.2.1),

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:20 PM
Deleted: None of the built-in functions described in this section shall initially have a **prototype** property unless otherwise specified in the description of a particular function.

Mark S. Miller 1/19/09 9:21 PM
Deleted: **true**

Mark S. Miller 1/19/09 9:21 PM
Deleted: **true**

Mark S. Miller 1/19/09 9:22 PM
Deleted: Unless other wise specified, the properties of the global object have attributes { [[Enumerable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:22 PM
Deleted: implementation-dependent

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself. However, all such additional properties must be frozen, and all values reachable from them must be frozen as well.

Mark S. Miller 1/19/09 9:23 PM
Deleted: ; for example, in the HTML document object model the **window** property of the global object is the global object itself

15.1.1 Value Properties of the Global Object

15.1.1.1 NaN

The initial value of **NaN** is **NaN** (8.5).

Mark S. Miller 1/19/09 9:24 PM
Deleted: . This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }

15.1.1.2 Infinity

The initial value of **Infinity** is +∞ (8.5).

Mark S. Miller 1/19/09 9:24 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.1.1.3 undefined

The initial value of **undefined** is **undefined** (8.1).

Mark S. Miller 1/19/09 9:24 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.1.2 Function Properties of the Global Object

15.1.2.1 eval (x)

Mark S. Miller 1/19/09 9:27 PM
Comment: Redo around the assumption that "eval" is a keyword. Still need to distinguish direct eval operator from indirect eval function.

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

1. If *x* is not a string value, return *x*.
2. Let *prog* be the SES code that is the result of parsing *x* as a *Program*. If the parse fails, throw a **SyntaxError** exception (but see also clause 16).
3. Let *evalCtx* be the result of establishing a new execution context (10.4.2) for the eval code *prog*.
4. Let *result* be the result of evaluating the program *prog*.
5. Exit the running execution context evalCtx, restoring the previous execution context.
6. If *result*.type is **normal** and its completion value is a value *V*, then return the value *V*.
7. If *result*.type is **normal** and its completion value is **empty**, then return the value **undefined**.
8. Otherwise, *result*.type must be **throw**. Throw *result*.value as an exception.

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

15.1.2.2 parseInt (string , radix)

Mark S. Miller 1/19/09 9:27 PM
Deleted: If the value of the **eval** property is used in any way other than a direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **eval** property is assigned to, an **EvalError** exception may be thrown.
15.1.2.1.1 Strict Mode Restrictions
If strict mode code uses the value of the **eval** property any way other than as a direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **eval** property is assigned to, an **EvalError** exception is thrown.

The **parseInt** function produces an integer value dictated by interpretation of the contents of the *string* argument according to the specified *radix*. Leading white space in the string is ignored. If *radix* is **undefined** or 0, it is assumed to be 10 except when the number begins with the character pairs **0x** or **0X**, in which case a radix of 16 is assumed. Any radix-16 number may also optionally begin with the character pairs **0x** or **0X**.

When the **parseInt** function is called, the following steps are taken:

1. Call ToString(*string*).
2. Let *S* be a newly created substring of Result(1) consisting of the first character that is not a *StrWhiteSpaceChar* and all characters following that character. (In other words, remove leading white space.)
3. Let *sign* be 1.
4. If *S* is not empty and the first character of *S* is a minus sign **-**, let *sign* be −1.
5. If *S* is not empty and the first character of *S* is a plus sign **+** or a minus sign **-**, then remove the first character from *S*.
6. Let *R* = ToInt32(*radix*).
7. If *R* = 0, go to step 11.
8. If *R* < 2 or *R* > 36, then return **NaN**.
9. If *R* = 16, go to step 13.
10. Go to step 14.
11. Let *R* = 10.
12. If the length of *S* is at least 1 and the first character of *S* is "**0**", then at the implementation's discretion either let *R* = 8 or leave *R* unchanged.
13. If the length of *S* is at least 2 and the first two characters of *S* are either "**0x**" or "**0X**", then remove the first two characters from *S* and let *R* = 16.
14. If *S* contains any character that is not a radix-*R* digit, then let *Z* be the substring of *S* consisting of all characters before the first such character; otherwise, let *Z* be *S*.
15. If *Z* is empty, return **NaN**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

16. Compute the mathematical integer value that is represented by *Z* in radix-*R* notation, using the letters **`A-Z`** and **`a-z`** for digits with values 10 through 35. (However, if *R* is 10 and *Z* contains more than 20 significant digits, every significant digit after the 20th may be replaced by a **`0`** digit, at the option of the implementation; and if *R* is not 2, 4, 8, 10, 16, or 32, then Result(16) may be an implementation-dependent approximation to the mathematical integer value that is represented by *Z* in radix-*R* notation.)
17. Compute the number value for Result(16).
18. Return *sign* × Result(17).

NOTE
`parseInt` *may interpret only a leading portion of the string as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.*

When radix is 0 or ***undefined*** *and the string's number begins with a* ***`0`*** *digit not followed by an* ***`x`*** *or* ***`X`****, then the implementation may, at its discretion, interpret the number either as being octal or as being decimal. Implementations are encouraged to interpret numbers in this case as being decimal.*

15.1.2.3 parseFloat (string)

The **`parseFloat`** function produces a number value dictated by interpretation of the contents of the *string* argument as a decimal literal.

When the **`parseFloat`** function is called, the following steps are taken:

1. Call ToString(*string*).
2. Compute a substring of Result(1) consisting of the leftmost character that is not a *StrWhiteSpaceChar* and all characters to the right of that character.(In other words, remove leading white space.)
3. If neither Result(2) nor any prefix of Result(2) satisfies the syntax of a *StrDecimalLiteral* (see 9.3.1), return **NaN**.
4. Compute the longest prefix of Result(2), which might be Result(2) itself, which satisfies the syntax of a *StrDecimalLiteral*.
5. Return the number value for the MV of Result(4).

NOTE
`parseFloat` *may interpret only a leading portion of the string as a number value; it ignores any characters that cannot be interpreted as part of the notation of an decimal literal, and no indication is given that any such characters were ignored.*

15.1.2.4 isNaN (number)

Returns **true** if the result is **NaN**, and otherwise returns **false**.

1. Call GetValue(number).
2. Call ToNumber(Result(1)).
3. If Result(2) is **NaN**, return **true**.
4. Return **false**.

15.1.2.5 isFinite (number)

Returns **false** if the result is **NaN**, +∞, or −∞, and otherwise returns **true**.

1. Call GetValue(number).
2. Call ToNumber(Result(1)).
3. If Result(2) is **NaN**, +∞, or −∞, return **false**.
4. Return **true**.

15.1.3 URI Handling Function Properties

Uniform Resource Identifiers, or URIs, are strings that identify resources (e.g. web pages or files) and transport protocols by which to access them (e.g. HTTP or FTP) on the Internet. The SES language itself does not provide any support for using URIs except for functions that encode and decode URIs as described in 15.1.3.1, 15.1.3.2, 15.1.3.3 and 15.1.3.4.

NOTE

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Many implementations of SES provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

A URI is composed of a sequence of components separated by component separators. The general form is:

Scheme **`:`** *First* **`/`** *Second* **`;`** *Third* **`?`** *Fourth*

where the italicised names represent components and the "**`:`**", "**`/`**", "**`;`**" and "**`?`**" are reserved characters used as separators. The **`encodeURI`** and **`decodeURI`** functions are intended to work with complete URIs; they assume that any reserved characters in the URI are intended to have special meaning and so are not encoded. The **`encodeURIComponent`** and **`decodeURIComponent`** functions are intended to work with the individual component parts of a URI; they assume that any reserved characters represent text and so must be encoded so that they are not interpreted as reserved characters when the component is part of a complete URI.

The following lexical grammar specifies the form of encoded URIs.

uri **:::**
  uriCharacters$_{opt}$

uriCharacters **:::**
  uriCharacter uriCharacters$_{opt}$

uriCharacter **:::**
  uriReserved
  uriUnescaped
  uriEscaped

uriReserved **::: one of**
  `; / ? : @ & = + $ ,`

uriUnescaped **:::**
  uriAlpha
  DecimalDigit
  uriMark

uriEscaped **:::**
  `%` *HexDigit HexDigit*

uriAlpha **::: 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`

uriMark **::: one of**
  `- _ . ! ~ * ' ( )`

When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is first transformed into a sequence of octets using the UTF-8 transformation, with surrogate pairs first transformed from their UCS-2 to UCS-4 encodings. (Note that for code units in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a string with each octet represented by an escape sequence of the form "`%xx`".

The encoding and escaping process is described by the hidden function Encode taking two string arguments *string* and *unescapedSet*. This function is defined for expository purpose only.

1. Compute the number of characters in *string*.
2. Let *R* be the empty string.
3. Let *k* be 0.
4. If *k* equals Result(1), return *R*.
5. Let *C* be the character at position *k* within *string*.
6. If *C* is not in *unescapedSet*, go to step 9.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

7. Let S be a string containing only the character C.
8. Go to step 24.
9. If the code unit value of C is not less than 0xDC00 and not greater than 0xDFFF, throw a **URIError** exception.
10. If the code unit value of C is less than 0xD800 or greater than 0xDBFF, let V be the code unit value of C and go to step 16.
11. Increase k by 1.
12. If k equals Result(1), throw a **URIError** exception.
13. Get the code unit value of the character at position k within *string*.
14. If Result(13) is less than 0xDC00 or greater than 0xDFFF, throw a **URIError** exception.
15. Let V be (((the code unit value of C) – 0xD800) * 0x400 + (Result(13) – 0xDC00) + 0x10000).
16. Let *Octets* be the array of octets resulting by applying the UTF-8 transformation to V, and let L be the array size.
17. Let j be 0.
18. Get the value at position j within *Octets*.
19. Let S be a string containing three characters "`%XY`" where XY are two uppercase hexadecimal digits encoding the value of Result(18).
20. Let R be a new string value computed by concatenating the previous value of R and S.
21. Increase j by 1.
22. If j is equal to L, go to step 25.
23. Go to step 18.
24. Let R be a new string value computed by concatenating the previous value of R and S.
25. Increase k by 1.
26. Go to step 4.

The unescaping and decoding process is described by the hidden function Decode taking two string arguments *string* and *reservedSet*. This function is defined for expository purpose only.

1. Compute the number of characters in *string*.
2. Let R be the empty string.
3. Let k be 0.
4. If k equals Result(1), return R.
5. Let C be the character at position k within *string*.
6. If C is not '%', go to step 40.
7. Let *start* be k.
8. If $k + 2$ is greater than or equal to Result(1), throw a **URIError** exception.
9. If the characters at position (k+1) and (k + 2) within *string* do not represent hexadecimal digits, throw a **URIError** exception.
10. Let B be the 8-bit value represented by the two hexadecimal digits at position (k + 1) and (k + 2).
11. Increment k by 2.
12. If the most significant bit in B is 0, let C be the character with code unit value B and go to step 37.
13. Let n be the smallest non-negative number such that (B << n) & 0x80 is equal to 0.
14. If n equals 1 or n is greater than 4, throw a **URIError** exception.
15. Let *Octets* be an array of 8-bit integers of size n.
16. Put B into *Octets* at position 0.
17. If k + (3 * (n – 1)) is greater than or equal to Result(1), throw a **URIError** exception.
18. Let j be 1.
19. If j equals n, go to step 29.
20. Increment k by 1.
21. If the character at position k is not '%', throw a **URIError** exception.
22. If the characters at position (k +1) and (k + 2) within *string* do not represent hexadecimal digits, throw a **URIError** exception.
23. Let B be the 8-bit value represented by the two hexadecimal digits at position (k + 1) and (k + 2).
24. If the two most significant bits in B are not 10, throw a **URIError** exception.
25. Increment k by 2.
26. Put B into *Octets* at position j.
27. Increment j by 1.
28. Go to step 19.
29. Let V be the value obtained by applying the UTF-8 transformation to *Octets*, that is, from an array of octets into a 32-bit value.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

30. If *V* is less than 0x10000, go to step 36.
31. If *V* is greater than 0x10FFFF, throw a **URIError** exception.
32. Let *L* be (((*V* – 0x10000) & 0x3FF) + 0xDC00).
33. Let *H* be ((((*V* – 0x10000) >> 10) & 0x3FF) + 0xD800).
34. Let *S* be the string containing the two characters with code unit values *H* and *L*.
35. Go to step 41.
36. Let *C* be the character with code unit value *V*.
37. If *C* is not in *reservedSet*, go to step 40.
38. Let *S* be the substring of *string* from position *start* to position *k* included.
39. Go to step 41.
40. Let *S* be the string containing only the character *C*.
41. Let *R* be a new string value computed by concatenating the previous value of *R* and *S*.
42. Increase *k* by 1.
43. Go to step 4.

NOTE 1
The syntax of Uniform Resource Identifiers is given in RFC2396.

NOTE 2
A formal description and implementation of UTF-8 is given in the Unicode Standard, Version 2.0, Appendix A.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of n octets, n>1, the initial octet has the n higher-order bits set to 1, followed by a bit set to 0. The remaining bits of that octet contain bits from the value of the character to be encoded. The following octets all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each contain bits from the character to be encoded. The possible UTF-8 encodings of SES characters are:

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

Code Unit Value	*Representation*	*1st Octet*	*2nd Octet*	*3rd Octet*	*4th Octet*
`0x0000 - 0x007F`	`00000000 0zzzzzzz`	`0zzzzzzz`			
`0x0080 - 0x07FF`	`00000yyy yyzzzzzz`	`110yyyyy`	`10zzzzzz`		
`0x0800 - 0xD7FF`	`xxxxyyyy yyzzzzzz`	`1110xxxx`	`10yyyyyy`	`10zzzzzz`	
`0xD800 - 0xDBFF` followed by `0xDC00 – 0xDFFF`	`110110vv vvwwwwxx` followed by `110111yy yyzzzzzz`	`11110uuu`	`10uuwwww`	`10xxyyyy`	`10zzzzzz`
`0xD800 - 0xDBFF` not followed by `0xDC00 – 0xDFFF`	causes `URIError`				
`0xDC00 – 0xDFFF`	causes `URIError`				
`0xE000 - 0xFFFF`	`xxxxyyyy yyzzzzzz`	`1110xxxx`	`10yyyyyy`	`10zzzzzz`	

Where
`uuuuu = vvvv + 1`

to account for the addition of 0x10000 as in 3.7, Surrogates of the Unicode Standard version 2.0.

The range of code unit values 0xD800-0xDFFF is used to encode surrogate pairs; the above transformation combines a UCS-2 surrogate pair into a UCS-4 representation and encodes the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

15.1.3.1 decodeURI (encodedURI)

The **`decodeURI`** function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the **`encodeURI`** function is replaced with the

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

character that it represents. Escape sequences that could not have been introduced by **`encodeURI`** are not replaced.

When the **`decodeURI`** function is called with one argument *encodedURI*, the following steps are taken:

1. Call ToString(*encodedURI*).
2. Let *reservedURISet* be a string containing one instance of each character valid in *uriReserved* plus "#".
3. Call Decode(Result(1), *reservedURISet*)
4. Return Result(3).

NOTE
The character "#" is not decoded from escape sequences even though it is not a reserved URI character.

15.1.3.2 decodeURIComponent (encodedURIComponent)

The **`decodeURIComponent`** function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the **`encodeURIComponent`** function is replaced with the character that it represents.

When the **`decodeURIComponent`** function is called with one argument *encodedURIComponent*, the following steps are taken:

1. Call ToString(*encodedURIComponent*).
2. Let *reservedURIComponentSet* be the empty string.
3. Call Decode(Result(1), *reservedURIComponentSet*)
4. Return Result(3).

15.1.3.3 encodeURI (uri)

The **`encodeURI`** function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

When the **`encodeURI`** function is called with one argument *uri*, the following steps are taken:

1. Call ToString(*uri*).
2. Let *unescapedURISet* be a string containing one instance of each character valid in *uriReserved* and *uriUnescaped* plus "#".
3. Call Encode(Result(1), *unescapedURISet*)
4. Return Result(3).

NOTE
The character "#" is not encoded to an escape sequence even though it is not a reserved or unescaped URI character.

15.1.3.4 encodeURIComponent (uriComponent)

The **`encodeURIComponent`** function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

When the **`encodeURIComponent`** function is called with one argument *uriComponent*, the following steps are taken:

1. Call ToString(*uriComponent*).
2. Let *unescapedURIComponentSet* be a string containing one instance of each character valid in *uriUnescaped*.
3. Call Encode(Result(1), *unescapedURIComponentSet*)
4. Return Result(3).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.1.4 Constructor Properties of the Global Object

15.1.4.1 Object (. . .)

See 15.2.1 and 15.2.2.

15.1.4.2 Function (. . .)

See 15.3.1 and 15.3.2.

15.1.4.3 Array (. . .)

See 15.4.1 and 15.4.2.

15.1.4.4 String (. . .)

See 15.5.1 and 15.5.2.

15.1.4.5 Boolean (. . .)

See 15.6.1 and 15.6.2.

15.1.4.6 Number (. . .)

See 15.7.1 and 15.7.2.

15.1.4.7 Date (. . .)

See 15.9.2.

15.1.4.8 RegExp (. . .)

See 15.10.3 and 15.10.4.

15.1.4.9 Error (. . .)

See 15.11.1 and 15.11.2.

15.1.4.10 EvalError (. . .)

See 15.11.6.1.

15.1.4.11 RangeError (. . .)

See 15.11.6.2.

15.1.4.12 ReferenceError (. . .)

See 15.11.6.3.

15.1.4.13 SyntaxError (. . .)

See 15.11.6.4.

15.1.4.14 TypeError (. . .)

See 15.11.6.5.

15.1.4.15 URIError (. . .)

See 15.11.6.6.

15.1.5 Other Properties of the Global Object

15.1.5.1 Math

See 15.8.

15.1.5.2 JSON

See 15.12.

15.2 Object Objects

15.2.1 The Object Constructor Called as a Function

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

15.2.1.1 Object ([value])

When the `Object` function is called with no arguments or with one argument *value*, the following steps are taken:

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. If *value* is **null**, **undefined** or not supplied, create and return a new Object object exactly if the object constructor had been called with the same arguments (15.2.2.1).
2. If *value* is an object, return value.
3. Else throw a **TypeError**.

15.2.2 The Object Constructor

When `Object` is called as part of a `new` expression, it will create a new object.

15.2.2.1 new Object ([value])

When the `Object` constructor is called with no arguments or with one argument *value*, the following steps are taken:

1. If *value* is supplied, throw a **TypeError**.
2. Else create a new native SES object.
   The [[Parent]] 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 [[Extensible]] property of the newly constructed object is set to **true**.
   The newly constructed object has no [[PrimitiveValue]] property.
3. Return the newly created native object.

15.2.3 Properties of the Object Constructor

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

Besides the internal properties and the `length` property (whose value is **1**), the Object constructor has the following properties:

15.2.3.1 N/A

15.2.3.2 Object.getPrototypeOf (O)

When the `getPrototypeOf` function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Return the value of the [[Parent]] internal property of *O*.

15.2.3.3 Object.getOwnPropertyDescriptor (O, P)

When the **getOwnPropertyDescriptor** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. If *P* is **undefined** or **null**, throw a **TypeError**, otherwise let *name* be ToString(*P*).
3. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *name*.
4. Return the result of calling FromPropertyDescriptor(*desc*).

15.2.3.4 Object.getOwnPropertyNames (O)

When the **getOwnPropertyNames** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *array* be the result of creating a new object as if by the expression `new Array ()` where `Array` is the standard built-in constructor with that name.
3. For each named own property *P* of *O*
   a. Let *name* be the string value that is the name of *P*.
   b. Append a property to *array* as if by calling *array*.push(*name*).
4. Return *array*.

NOTE
If O is a String instance, the set of own properties processed in step 3a does not include the implicit properties defined in 15.5.5.2 that correspond to character positions of the object's [[PrimitiveValue]] string.

Mark S. Miller 1/19/09 9:30 PM
Deleted:

Mark S. Miller 1/19/09 9:31 PM
Deleted: Return ToObject(*value*)

Mark S. Miller 1/19/09 9:31 PM
Deleted: is a constructor that may

Mark S. Miller 1/19/09 9:31 PM
Deleted: n

pratapL 1/19/09 8:57 PM
Comment: Deviations doc item 3.12 suggests removing this phrase.

Mark S. Miller 1/19/09 9:34 PM
Deleted: not

Mark S. Miller 1/19/09 9:34 PM
Deleted: go to step 8

Mark S. Miller 1/19/09 9:34 PM
Deleted:
<#>If the type of *value* is not Object, go to step 5.
<#>If the *value* is a native ECMAScript object, do not create a new object but simply return *value.*
<#>If the *value* is a host object, then actions are taken and a result is returned in an implementation-dependent manner that may depend on the host object.
<#>If the type of *value* is String, return ToObject(*value*).
<#>If the type of *value* is Boolean, return ToObject(*value*).
If the type of *value* is Number, return ToObject(*value*).

Mark S. Miller 1/19/09 9:34 PM
Deleted: (The argument *value* was not supplied or its type was Null or Undefined.)

Mark S. Miller 1/19/09 9:34 PM
Deleted: C

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:35 PM
Deleted: Object.prototype

Mark S. Miller 1/19/09 9:35 PM
Deleted: The initial value of `Object.prototype` is the Object prototype object (15.2.4).
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:38 PM
Comment: Also fix ES3.1

Mark S. Miller 1/19/09 9:37 PM
Deleted: let *name* be the empty string

Mark S. Miller 1/19/09 9:40 PM
Deleted: the standard built-in method Array.prototype.push with *array* as the **this** ... [27]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.2.3.5 Object.create (O [, Properties])

The **create** function creates a new object with a specified parent. When the **create** function is called, the following steps are taken:

Mark S. Miller 1/19/09 9:40 PM
Deleted: prototype

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *obj* be the result of creating a new object as if by the expression new Object() where Object is the standard built-in constructor with that name
3. Set the [[Parent]] internal property of *obj* to *O*.
4. Add own properties to *obj* as if by calling the standard built-in function Object.defineProperties with arguments *obj* and *Properties*.
5. Return *obj*.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

15.2.3.6 Object.defineProperty (O, P, Attributes)

The **defineProperty** function is used to add an own properties and/or update the attributes of existing own property of an object. When the **defineProperty** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *name* be ToString(*P*).
3. Let *desc* be the result of calling ToPropertyDescriptor with *Attributes* as the argument.
4. Call the [[DefineOwnProperty]] internal method of *O* with arguments *name* and *desc*.
5. Return *O*.

Mark S. Miller 1/19/09 9:41 PM
Deleted: ,

Mark S. Miller 1/19/09 9:42 PM
Deleted: , and **true**

15.2.3.7 Object.defineProperties (O, Properties)

The **defineProperties** function is used to add own properties and/or update the attributes of existing own properties of an object. When the **defineProperties** function is called, the following steps are taken atomically:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *props* be ToObject(*Properties*).
3. For each named own property name P of *props*,
    a. Let *descObj* be the result of calling the [[GetOwnProperty]] internal method of *props* with *P* as the argument.
    b. Let *desc* be the result of calling ToPropertyDescriptor with *descObj* as the argument.
4. Call the [[DefineOwnProperty]] internal method of *O* with arguments *P* and *desc*.
5. Return *O*.

The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of O, this function must be implemented such that it either atomically completes all property updates successfully or fails without making any update to the properties of object *O*.

Mark S. Miller 1/19/09 9:42 PM
Deleted: ,

Mark S. Miller 1/19/09 9:42 PM
Deleted: , and **true**

15.2.3.8 Object.seal (O)

When the **seal** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] method of *O* with *P*.
    b. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**
    c. Call the [[DefineOwnProperty]] internal method of *O* with *P* and *desc* as arguments.
3. Set the internal [[Extensible]] internal property of *O* to **false**.
4. Return *O*.

Mark S. Miller 1/19/09 9:43 PM
Deleted: ,

Mark S. Miller 1/19/09 9:43 PM
Deleted: , and **true**

The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of O, this function must be implemented such that it either atomically completes all property updates successfully or fails without making any update to the properties of object O.

15.2.3.9 Object.freeze (O)

When the **freeze** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] method of *O* with *P*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

b. If IsDataDescriptor(*desc*) then
    i. If *desc*.[[Writable]] is **true**, set *desc*.[[Writable]] to **false**.
c. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**.
d. Call the [[DefineOwnProperty]] internal method of *O* with *P* and *desc* as arguments.

3. Set the [[Extensible]] internal property of *O* to **false**.
4. Return *O*.

The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of *O*, this function must be implemented such that it either atomically completes all updates successfully or fails without making any update to object *O*.

15.2.3.10 Object.preventExtensions (O)

When the **preventExtensions** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Set the [[Extensible]] internal property of *O* to **false**.
3. Return *O*.

15.2.3.11 Object.isSealed (O)

When the **isSealed** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own data property *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
    b. If the *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **false**, then return **true**.
4. Otherwise, return **false**.

15.2.3.12 Object.isFrozen (O)

When the **isFrozen** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own data property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] method of *O* with *P*.
    b. If IsDataDescriptor(*desc*) then
        i. If *desc*.[[Writable]] is **true**, return **false**.
    c. If *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **true**, then return **false**.
4. Otherwise, return **true**.

15.2.3.13 Object.isExtensible (O)

When the **isExtensible** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Return the Boolean value of the [[Extensible]] internal property of *O*.

15.2.3.14 Object.keys (O)

When the **keys** function is called with argument O, the following steps are taken:

1. If the Type(*O*) is not Object, throw a **TypeError** exception.
2. Let *array* be the result of creating a new Object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
3. For each own enumerable property of O, append the key string of the property to *array*.
4. Return *array*.

NOTE
If an implementation defines a specific order of enumeration for the for-in statement, Object.keys must return that same order.

15.2.4 Properties of the Object Prototype Object

The value of the internal [[Parent]] property of the Object prototype object is **null**, the value of the internal [[Class]] property is `"Object"`, and the value of the internal [[Extensible]] property is **false**.

Mark S. Miller 1/19/09 9:44 PM
Deleted: ,

Mark S. Miller 1/19/09 9:44 PM
Deleted: , and **true**

Mark S. Miller 1/19/09 9:46 PM
Comment: If we can, SES should specify a deterministic order.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:46 PM
Deleted: **true**

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.2.4.1 N/A

Mark S. Miller 1/19/09 9:46 PM
Deleted: Object.prototype.constructor

15.2.4.2 Object.prototype.toString ( )

Mark S. Miller 1/19/09 9:46 PM
Deleted: The initial value of **Object.prototype.constructor** is the built-in **Object** constructor.

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

1. Let *O* be the result of calling ToObject passing the **this** object as the argument.
2. Get the [[Class]] property of *O*.
3. Compute a string value by concatenating the three strings **"[object "**, Result(2), and **"]"**.
4. Return Result(3).

15.2.4.3 Object.prototype.toLocaleString ( )

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

1. Let *O* be the result of calling ToObject passing the **this** object as the argument.
2. Call the [[Get]] internal method of *O* passing **"toString"** as the argument.
3. If IsCallable(Result(2)) is **false**, throw a **TypeError** exception.
4. Call the [[Call]] internal method of Result(2) passing *O* as the **this** value and no arguments.
5. Return Result(4).

NOTE 1
This function is provided to give all Objects a generic **toLocaleString** *interface, even though not all may use it. Currently,* **Array**, **Number**, *and* **Date** *provide their own locale-sensitive* **toLocaleString** *methods.*

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.2.4.4 Object.prototype.valueOf ( )

The **valueOf** method returns its **this** value.

Mark S. Miller 1/19/09 9:48 PM
Deleted: . If the object is the result of calling the Object constructor with a host object (15.2.2.1), it is implementation-defined whether **valueOf** returns its **this** value or another value such as the host object originally passed to the constructor

15.2.4.5 Object.prototype.hasOwnProperty (V)

When the **hasOwnProperty** method is called with argument *V*, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** object as the argument.
2. Call ToString(*V*).
3. Call the [[GetOwnProperty]] internal method of *O* passing Result(2) as the argument.
4. If Result(3) is **undefined**, return **false**.
5. Return **true**.

NOTE
Unlike [[HasProperty]] (8.6.2.4), this method does not consider objects in the prototype chain.

15.2.4.6 Object.prototype.isPrototypeOf (V)

When the **isPrototypeOf** method is called with argument *V*, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** object as the argument.
2. If *V* is not an object, return **false**.
3. Let *V* be the value of the [[Parent]] property of *V*.
4. if *V* is **null**, return **false**
5. If *O* and *V* refer to the same object, return **true**.
6. Go to step 3.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

15.2.4.7 Object.prototype.propertyIsEnumerable (V)

When the **propertyIsEnumerable** method is called with argument *V*, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** object as the argument.
2. Call ToString(*V*).
3. Call the [[GetOwnProperty]] internal method of O passing Result(2) as the argument.
4. If Result(3) is **undefined**, return **false**.
5. Return the value of Result(3).[[Enumerable]].

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

NOTE
This method does not consider objects in the prototype chain.

15.2.5 Properties of Object Instances

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

15.3 Function Objects

15.3.1 The Function Constructor Called as a Function

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

15.3.2 The Function Constructor

When **`Function`** is called as part of a **`new`** expression, it throws a **TypeError**.

15.3.3 Properties of the Function Constructor

The value of the internal [[Parent]] property of the Function constructor is the Function prototype object (15.3.4).

The value of the internal [[Extensible]] property of the Function constructor is **false**.

The Function constructor has properties named “caller” and “arguments” whose value is **null**. These properties have attributes: attribute {[[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}. An SES implementation must not associate any implementation specific behaviour with access of these properties.

The Function constructor has the following properties:

15.3.3.1 N/A

15.3.3.2 Function.length

The is a data property with an initial value of 1.

15.3.4 Properties of the Function Prototype Object

The Function prototype object is itself a Function object (its [[Class]] is **`"Function"`**) that, when invoked, accepts any arguments and returns **undefined**.

The value of the internal [[Parent]] property of the Function prototype object is the Object prototype object (15.3.2.1).

It is a function with an “empty body”; if it is invoked, it merely returns **undefined**.

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

15.3.4.1 N/A

15.3.4.2 Function.prototype.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 white space, line terminators, and semicolons within the representation string is implementation-dependent.

The **`toString`** function is not generic; it throws a **TypeError** exception if its **this** value is not a Function object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.3.4.3 Function.prototype.apply (ignored, argArray)

The **`apply`** method takes two arguments, *ignored* and *argArray*, and performs a function call using the [[Call]] property of the object. If the object does not have a [[Call]] property, a **TypeError** exception is thrown.

If *argArray* is **null** or **undefined**, the called function is passed no arguments. Otherwise, if *argArray* is neither an array nor an arguments object (see 10.3.2), a **TypeError** exception is thrown. If *argArray* is either an array or an arguments object, the function is passed the (ToUint32(*argArray*.length)) arguments *argArray*[0], *argArray*[1], …, *argArray*[ToUint32(*argArray*.length)–1].

Mark S. Miller 1/19/09 9:50 PM
Deleted: it creates and initialises a new Function object

Mark S. Miller 1/19/09 9:50 PM
Deleted: **15.3.1.1 Function (p1, p2, … , pn, body)**
When the **`Function`** function is called with some arguments *p1*, *p2*, … , *pn*, *body* (where *n* might be 0, that is, there are no “*p*” arguments, and where *body* might also not be provided), the following steps are taken:
<#>Create and return a new Function object as if the standard built-in constructor Function was used in a **new** expression with the same arguments (15.3.2.1).

Mark S. Miller 1/19/09 9:51 PM
Deleted: is a constructor: it initialises the newly created object.

Mark S. Miller 1/19/09 9:51 PM
Deleted: **15.3.2.1 new Function (p1, p2, … , pn, 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 *p1*, *p2*, … , *pn*, *body* (wher ... [28]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:52 PM
Deleted: **true**

Mark S. Miller 1/19/09 9:52 PM
Deleted: initial

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 9:52 PM
Deleted: **Function.prototype**

Mark S. Miller 1/19/09 9:52 PM
Deleted: The initial value of **`Function.prototype`** is the Function p ... [29]

Mark S. Miller 1/19/09 9:53 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, ... [30]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 9:54 PM
Deleted: The initial value of the internal [[Extensible]] property of the Function prot ... [31]

Mark S. Miller 1/19/09 9:54 PM
Deleted: **Function.prototype.constructor**

Mark S. Miller 1/19/09 9:54 PM
Deleted: The initial value of **`Function.prototype.constructo`** ... [32]

Mark S. Miller 1/19/09 9:59 PM
Deleted: **thisArg**

Mark S. Miller 1/19/09 9:59 PM
Deleted: *thisArg*

Mark S. Miller 1/19/09 9:59 PM
Deleted: The called function is passed *thisArg* as the **this** value.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The **length** property of the **apply** method is **2**.

15.3.4.4 Function.prototype.call (ignored [, arg1 [, arg2, …]])

The **call** method takes one or more arguments, *ignored* and (optionally) *arg1*, *arg2* etc, and performs a function call using the [[Call]] property of the object. If the object does not have a [[Call]] property, a **TypeError** exception is thrown. The called function is passed *arg1*, *arg2*, etc. as the arguments.

1. The **length** property of the **call** method is **1**.

15.3.4.5 Function.prototype.bind (ignored [, arg1 [, arg2, …]])

The bind method takes one or more arguments, *ignored* and (optionally) *arg1*, *arg2*, etc, and returns a new function object by performing the following steps:

1. Let *Target* be the **this** object.
2. If IsCallable(*Target*) is **false**, throw a **TypeError** exception.
3. Let *A* be a new (possibly empty) internal list of all of the argument values provided after *ignored* (*arg1*, *arg2* etc), in order.
4. Create a new native SES object and let *F* be that object.
5. Set the [[TargetFunction]] internal property of *F* to *Target*.
6. Set the [[BoundArgs]] internal property of *F* to *A*.
7. Set the [[Class]] internal property of F to **"Function"**.
8. Set the [[Parent]] internal property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
9. Set the [[Call]] internal property of *F* as described in 15.3.4.5.1.
10. The [[Scope]] internal property of *F* is unused and need not exist.
11. If the [[Class]] internal property of *Target* is **"Function"**, then
    a. Let *L* be the **length** property of *Target* minus the length of *A*.
    b. Set the **length** own property of *F* to either 0 or *L*, whichever is larger.
12. Else set the **length** own property of *F* to 0.
13. The length own property of F is given attributes as specified in 15.3.5.1.
14. Set the [[Extensible]] property of F to **false**.
15. Return F.

15.3.4.5.1 [[Call]]

When the [[Call]] internal method of a function object, *F*, that was created using the bind function is called with a **this** value and a list of arguments *ExtraArgs* the following steps are taken:

1. Let *boundArgs* be the value of *F's* [[BoundArgs]] internal property.
2. Let *target* be the value of *F's* [[TargetFunction]] internal property.
3. Let *args* be a new list containing the same values as the list *boundArgs* in the same order followed by the same values as the list *ExtraArgs* in the same order.
4. Return the result of calling the [[Call]] internal method of *target* providing *args* as the arguments.

15.3.5 Properties of Function Instances

In addition to the required internal properties, every function instance has a [[Call]] property, a [[Construct]] property, a [[FormalParameters]] property, a [[Code]] property, and a [[Scope]] property (see 8.6.2 and 13.2). The value of the [[Class]] property is **"Function"**.

Function instances have properties named "caller" and "arguments" whose value is **null**. These properties have attributes: attribute { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }. An SES implementation must not associate any implementation specific behaviour with accesses of these properties.

15.3.5.1 length

The value of the **length** property is 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 behaviour of a function when invoked on a number of arguments other than the number specified by its **length** property depends on the function. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 9:59 PM
Deleted: **thisArg**

Mark S. Miller 1/19/09 9:59 PM
Deleted: *thisArg*

Mark S. Miller 1/19/09 10:00 PM
Deleted: The called function is passed *thisArg* as the **this** value.

Mark S. Miller 1/19/09 10:00 PM
Deleted: **thisArg**

Mark S. Miller 1/19/09 10:00 PM
Deleted: *thisArg*

Mark S. Miller 1/19/09 10:00 PM
Deleted: *thisArg*

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 10:01 PM
Deleted: <#>Set the [[BoundThis]] internal property of F to the value of *thisArg*.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:01 PM
Deleted: <#>Set the [[Construct]] internal property of *F* as described in 15.3.4.5.2.

Mark S. Miller 1/19/09 10:02 PM
Deleted: **true**

Mark S. Miller 1/19/09 10:03 PM
Deleted: <#>If the [[Class]] property of *Target* is **"Function"**, then
<#>Let *boundProto* be the value of the prototype property of *Target*
<#>Else
<#>Let *boundProto* be the result of creating a new object as would be constructed by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
<#>Set the **constructor own** property of *boundProto* to *F*. This property has attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.
<#>Set the **prototype** property of *F* to *boundProto*.
<#>The **prototype** property of F is given attributes as specified in 15.3.5.2.

Mark S. Miller 1/19/09 10:04 PM
Deleted: <#>Let *boundThis* be the value of *F's* [[BoundThis]] internal property.

Mark S. Miller 1/19/09 10:04 PM
Deleted: providing *boundThis* as the **this** value and

Mark S. Miller 1/19/09 10:04 PM
Deleted: **15.3.4.5.2 [[Construct]]**
When the [[Construct]] internal method of a function object, *F*, that was created using the bind function is called ... [33]

Mark S. Miller 1/19/09 10:05 PM
Deleted: that correspond to strict mode functions

Mark S. Miller 1/19/09 10:05 PM
Deleted: initial

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 10:05 PM
Deleted: y from strict mode function code

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.3.5.3 [[HasInstance]] (V)

Assume *F* is a Function object.

When the [[HasInstance]] method of *F* is called with value *V*, the following steps are taken:

1. If *V* is not an object, return **false**.
2. Call the [[Get]] method of *F* with property name **"prototype"**.
3. Let *O* be Result(2).
4. If *O* is not an object, throw a **TypeError** exception.
5. Let *V* be the value of the [[Parent]] property of *V*.
6. If *V* is **null**, return **false**.
7. If *O* and *V* refer to the same object, return **true**.
8. Go to step 5.

Mark S. Miller 1/19/09 10:06 PM
Deleted: **15.3.5.2 prototype**
The value of the **prototype** property is used to initialise the internal [[Prototype]] property of a newly created object before the Function object is invoked as a constructor for that newly created object. This property has the attribute { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:07 PM
Comment: Replace this with an internal property, but don't call it [[Prototype]]!

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

15.3.5.4 name

The value of the name property is the name of the function, or an empty string if the function is anonymous. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

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 a nonnegative integer less than 2^{32}. The value of 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 Called as a Function

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

15.4.1.1 Array ([item1 [, item2 [, …]]])

When the **Array** function is called the following steps are taken:

1. Create and return a new Array object exactly as if the standard built-in constructor **Array** was used in a **new** expression with the same arguments (15.4.2).

15.4.2 The Array Constructor

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

15.4.2.1 new Array ([item0 [, item1 [, …]]])

This description applies if and only if the Array constructor is given no arguments or at least two arguments.

The [[Parent]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **Array.prototype** (15.4.3.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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

The [[Extensible]] property of the newly constructed object is set to **true**.

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* (if supplied); the **1** property of the newly constructed object is set to *item1* (if supplied); and, in general, for as many arguments as there

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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`.

15.4.2.2 new Array (len)

The [[Parent]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] property of the newly constructed object is set to **`"Array"`**. The [[Extensible]] property of the newly constructed object is set to **true**.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

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*.

15.4.3 Properties of the Array Constructor

The value of the internal [[Parent]] property of the Array constructor is the Function prototype object (15.3.4).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Besides the internal properties and the **`length`** property (whose value is **1**), the Array constructor has the following properties:

15.4.3.1 N/A

Mark S. Miller 1/19/09 10:09 PM
Deleted: **Array.prototype**

15.4.3.2 Array.isArray (arg)

Mark S. Miller 1/19/09 10:10 PM
Comment: This should just test [[Class]] === "Array", as it should in ES3.1.

Mark S. Miller 1/19/09 10:09 PM
Deleted: The initial value of **`Array.prototype`** is the Array prototype object (15.4.4).
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

The isArray function takes one argument *arg*, and returns the Boolean value **true** if the argument is an object behaves as an Array; otherwise it return **false**. The following steps are taken:

1. If Type(*arg*) is not Object, return **false**.
2. If *arg* has the internal [[IsArray]] property and the value of the property is **true**, then return **true**.
3. Return **false**.

15.4.4 Properties of the Array Prototype Object

The value of the internal [[Parent]] property of the Array prototype object is the Object prototype object (15.2.3.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

The Array prototype object is itself an array; its [[Class]] is **`"Array"`**, and it has a **`length`** property (whose initial value is **+0**) and the special internal [[ThrowingPut]] method described in 15.4.5.1.

Mark S. Miller 1/19/09 10:11 PM
Deleted: It also has the internal property [[IsArray]] whose value is **true**.

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 the **this** to be an object for which the value of the internal [[Class]] property is not **`"Array"`**.

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

15.4.4.1 N/A

Mark S. Miller 1/19/09 10:11 PM
Deleted: **Array.prototype.constructor**

15.4.4.2 Array.prototype.toString ( )

Mark S. Miller 1/19/09 10:11 PM
Deleted: The initial value of **`Array.prototype.constructor`** is the built-in **`Array`** constructor.

The result of calling this function is the same as if the standard built-in method **`Array.prototype.join`** were invoked for this object with no argument.

The **`toString`** function is not generic; it throws a **TypeError** exception if its **this** value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.4.4.3 Array.prototype.toLocaleString ( )

The elements of the array are converted to strings using their **`toLocaleString`** methods, and these strings are then concatenated, separated by occurrences of a separator string that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of **`toString`**, except that the result of this function is intended to be locale-specific.

The result is calculated as follows:

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Let *separator* be the list-separator string appropriate for the host environment's current locale (this is derived in an implementation-defined way).
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument **"0"**.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(6)).toLocaleString().
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the [[Get]] method of this object with argument ToString(*k*).
13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(12)).toLocaleString().
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

The **toLocaleString** function is not generic; it throws a **TypeError** exception if its **this** value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is required that implementations of this version do not use this parameter position for anything else.

Mark S. Miller 1/19/09 10:12 PM
Deleted: *recommended*

15.4.4.4 Array.prototype.concat ([item1 [, item2 [, …]]])

When the **concat** method is called with zero or more arguments *item1*, *item2*, etc., it returns an array containing the array elements of the object followed by the array elements of each argument in order.

The following steps are taken:

1. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
2. Let *n* be 0.
3. Let *E* be this object.
4. If *E* is not an Array object, go to step 16.
5. Let *k* be 0.
6. Call the [[Get]] method of *E* with argument **"length"**.
7. If *k* equals Result(6) go to step 19.
8. Call ToString(*k*).
9. If *E* has a property named by Result(8), go to step 10, but if *E* has no property named by Result(8), go to step 13.
10. Call ToString(n).
11. Call the [[Get]] method of *E* with argument Result(8).
12. Call the [[Put]] method of *A* with arguments Result(10) and Result(11).
13. Increase *n* by 1.
14. Increase *k* by 1.
15. Go to step 7.
16. Call ToString(*n*).
17. Call the [[Put]] method of *A* with arguments Result(16) and *E*.
18. Increase *n* by 1.
19. Get the next argument in the argument list; if there are no more arguments, go to step 22.
20. Let *E* be Result(19).
21. Go to step 4.
22. Call the [[Put]] method of *A* with arguments **"length"** and *n*.
23. Return *A*.

Mark S. Miller 1/19/09 10:14 PM
Comment: Example of an unnecessary ambiguity. Should this be based on [[Class]] or [[IsArray]]?

The **length** property of the **concat** method is **1**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

NOTE
*The **concat** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **concat** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.5 Array.prototype.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.

The **join** method takes one argument, *separator*, and performs the following steps:

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If *separator* is **undefined**, let *separator* be the single-character string **","**.
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument **"0"**.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(6)).
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the [[Get]] method of this object with argument ToString(*k*).
13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(12)).
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

The **length** property of the **join** method is **1**.

NOTE
*The **join** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **join** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.6 Array.prototype.pop ( )

The last element of the array is removed from the array and returned.

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the [[Put]] method of this object with arguments **"length"** and Result(2).
5. Return **undefined**.
6. Call ToString(Result(2)–1).
7. Call the [[Get]] method of this object with argument Result(6).
8. Call the [[Delete]] method of this object with argument Result(6).
9. Call the [[Put]] method of this object with arguments **"length"** and (Result(2)–1).
10. Return Result(7).

NOTE
*The **pop** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **pop** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.7 Array.prototype.push ([item1 [, item2 [, …]]])

The arguments are appended to the end of the array, in the order in which they appear. The new length of the array is returned as the result of the call.

When the **push** method is called with zero or more arguments *item1, item2*, etc., the following steps are taken:

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

1. Call the [[Get]] method of this object with argument **`"length"`**.
2. Let *n* be the result of calling ToUint32(Result(1)).
3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
4. Call the [[Put]] method of this object with arguments ToString(*n*) and Result(3).
5. Increase *n* by 1.
6. Go to step 3.
7. Call the [[Put]] method of this object with arguments **`"length"`** and *n*.
8. Return *n*.

The **`length`** property of the **`push`** method is **1**.

NOTE
*The **`push`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`push`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.8 Array.prototype.reverse ( )

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

1. Call the [[Get]] method of this object with argument **`"length"`**.
2. Call ToUint32(Result(1)).
3. Compute floor(Result(2)/2).
4. Let *k* be **`0`**.
5. If *k* equals Result(3), return this object.
6. Compute Result(2)−*k*−1.
7. Call ToString(*k*).
8. Call ToString(Result(6)).
9. Call the [[Get]] method of this object with argument Result(7).
10. Call the [[Get]] method of this object with argument Result(8).
11. If this object does not have a property named by Result(8), go to step 19.
12. If this object does not have a property named by Result(7), go to step 16.
13. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
14. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
15. Go to step 25.
16. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
17. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
18. Go to step 25.
19. If this object does not have a property named by Result(7), go to step 23.
20. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete..
21. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
22. Go to step 25.
23. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete.
24. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
25. Increase *k* by 1.
26. Go to step 5.

NOTE
*The **`reverse`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **`reverse`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.9 Array.prototype.shift ( )

The first element of the array is removed from the array and returned.

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the [[Put]] method of this object with arguments **"length"** and Result(2).
5. Return **undefined**.
6. Call the [[Get]] method of this object with argument **0**.
7. Let *k* be 1.
8. If *k* equals Result(2), go to step 18.
9. Call ToString(*k*).
10. Call ToString(*k*–1).
11. If this object has a property named by Result(9), go to step 12; but if this object has no property named by Result(9), then go to step 15.
12. Call the [[Get]] method of this object with argument Result(9).
13. Call the [[Put]] method of this object with arguments Result(10) and Result(12).
14. Go to step 16.
15. Call the [[Delete]] method of this object with argument Result(10).
16. Increase *k* by 1.
17. Go to step 8.
18. Call the [[Delete]] method of this object with argument ToString(Result(2)–1).
19. Call the [[Put]] method of this object with arguments **"length"** and (Result(2)–1).
20. Return Result(6).

NOTE
The ***shift*** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the* ***shift*** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.10 Array.prototype.slice (start, end)

The **slice** method takes two arguments, *start* and *end*, and returns an array containing the elements of the array from element *start* up to, but not including, element *end* (or through the end of the array if *end* is **undefined**). If *start* is negative, it is treated as (*length*+*start*) where *length* is the length of the array. If *end* is negative, it is treated as (*length*+*end*) where *length* is the length of the array. The following steps are taken:

1. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
2. Call the [[Get]] method of this object with argument **"length"**.
3. Call ToUint32(Result(2)).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Let *k* be Result(5).
7. If *end* is **undefined**, use Result(3); else use ToInteger(*end*).
8. If Result(7) is negative, use max((Result(3)+Result(7)),0); else use min(Result(7),Result(3)).
9. Let *n* be 0.
10. If *k* is greater than or equal to Result(8), go to step 19.
11. Call ToString(*k*).
12. If this object has a property named by Result(11), go to step 13; but if this object has no property named by Result(11), then go to step 16.
13. Call ToString(*n*).
14. Call the [[Get]] method of this object with argument Result(11).
15. Call the [[Put]] method of *A* with arguments Result(13) and Result(14).
16. Increase *k* by 1.
17. Increase *n* by 1.
18. Go to step 10.
19. Call the [[Put]] method of *A* with arguments **"length"** and *n*.
20. Return *A*.

The **length** property of the **slice** method is **2**.

NOTE

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

*The **`slice`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`slice`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.11 Array.prototype.sort (comparefn)

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If *comparefn* is not **undefined**, it should be a function that accepts two arguments x and y and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

If *comparefn* is not **undefined** and is not a consistent comparison function for the elements of this array (see below), the behaviour of **`sort`** is implementation-defined. Let *len* be ToUint32(**`this.length`**). If there exist integers i and j and an object P such that all of the conditions below are satisfied then the behaviour of **`sort`** is implementation-defined:

$0 \le i < len$

$0 \le j < len$

`this` does not have a property with name ToString(i)

P is obtained by following one or more [[Parent]] properties starting at **`this`**

P has a property with name ToString(j)

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

The behaviour of sort is also implementation defined if any of the following conditions are true:

The [[Extensible]] internal property of this array is **false**.

Any array index property of this array whose name has a numeric value that is a nonnegative integer less than *len* is a data property whose [[Writable]] attribute is **false**.

If any property of this array whose property name is a nonnegative integer less than *len* is an accessor property.

Otherwise the following steps are taken.

1. Let *obj* be this object.
2. Let *getLen* be the result of calling the [[Get]] internal method of obj with argument "**`length`**".
3. Let *len* be ToUint32(*getLen*).
4. Perform an implementation-dependent sequence of calls to the [[Get]] , [[Put]], and [[Delete]] methods of this object and to SortCompare (described below), where the first argument for each call to [[Get]], [[Put]], or [[Delete]] is a nonnegative integer less than *len* and where the arguments for calls to SortCompare are results of previous calls to the [[Get]] method.
5. Return this object.

The returned object must have the following two properties.

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), if property **`old[`***j***`]`** existed, then **`new[`**$\pi(j)$**`]`** is exactly the same value as **`old[`***j***`]`**,. But if property **`old[`***j***`]`** did not exist, then **`new[`**$\pi(j)$**`]`** does not exist.

Then for all nonnegative integers j and k, each less than Result(2), if SortCompare(j,k) < 0 (see SortCompare below), then $\pi(j) < \pi(k)$.

Here the notation **`old[`***j***`]`** is used to refer to the hypothetical result of calling the [[Get]] method of this object with argument j before this function 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 function has been executed.

A function *comparefn* is a consistent comparison function for a set of values S if all of the requirements below are met for all values a, b, and c (possibly the same value) in the set S: The notation $a <_{CF} b$ means *comparefn*(a,b) < 0; $a =_{CF} b$ means *comparefn*(a,b) = 0 (of either sign); and $a >_{CF} b$ means *comparefn*(a,b) > 0.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Calling *comparefn*(a,b) always returns the same value v when given a specific pair of values a and b as its two arguments. Furthermore, v has type Number, and v is not NaN. Note that this implies that exactly one of $a <_{CF} b$, $a =_{CF} b$, and $a >_{CF} b$ will be true for a given pair of a and b.

$a =_{CF} a$ (reflexivity)

If $a =_{CF} b$, then $b =_{CF} a$ (symmetry)

If $a =_{CF} b$ and $b =_{CF} c$, then $a =_{CF} c$ (transitivity of $=_{CF}$)

If $a <_{CF} b$ and $b <_{CF} c$, then $a <_{CF} c$ (transitivity of $<_{CF}$)

If $a >_{CF} b$ and $b >_{CF} c$, then $a >_{CF} c$ (transitivity of $>_{CF}$)

NOTE
The above conditions are necessary and sufficient to ensure that comparefn divides the set S into equivalence classes and that these equivalence classes are totally ordered.

NOTE 2
Calling comparefn(a, b) does not modify the ***this*** *object.*

When the SortCompare abstract operation is called with two arguments *j* and *k*, the following steps are taken:

1. Let *jString* be ToString(j).
2. Let *kString* ToString(k).
3. If this object does not have a property named by *jString*, and this object does not have a property named by *kString*, return **+0**.
4. If this object does not have a property named by *jString*, return 1.
5. If this object does not have a property named by *kString*, return –1.
6. Let *x* be the result of calling the [[Get]] internal method of this object with argument *jString*.
7. Let *y* be the result of calling the [[Get]] internal method of this object with argument *kString*.
8. If *x* and *y* are both undefined, return **+0**.
9. If *x* is undefined, return 1.
10. If *y* is undefined, return −1.
11. If the argument *comparefn* is not **undefined**, then
    a. Return the result of calling the [[Call]] internal method of *comparefn* passing **undefined** as the **this** value and with arguments *x* and *y*.
12. Let *xString* be ToString(*x*).
13. Let *yString* be ToString(*y*).
14. If *xString* < *yString*, return −1.
15. If *xString* > *yString*, return 1.
16. Return **+0**.

NOTE 1
Because non-existent property values always compare greater than ***undefined*** *property values, and* ***undefined*** *always compares greater than any other value, undefined property values always sort to the end of the result, followed by non-existent property values.*

NOTE 2
The **`sort`** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the* **`sort`** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.12 Array.prototype.splice (start, deleteCount [, item1 [, item2 [, …]]])

When the **`splice`** method is called with two or more arguments *start*, *deleteCount* and (optionally) *item1*, *item2*, etc., the *deleteCount* elements of the array starting at array index *start* are replaced by the arguments *item1*, *item2*, etc. The following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
2. Call the [[Get]] method of this object with argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

6. Compute min(max(ToInteger(*deleteCount*),0),Result(3)–Result(5)).
7. Let *k* be 0.
8. If *k* equals Result(6), go to step 16.
9. Call ToString(Result(5)+*k*).
10. If this object has a property named by Result(9), go to step 11; but if this object has no property named by Result(9), then go to step 14.
11. Call ToString(*k*).
12. Call the [[Get]] method of this object with argument Result(9).
13. Call the [[Put]] method of *A* with arguments Result(11) and Result(12).
14. Increment *k* by 1.
15. Go to step 8.
16. Call the [[Put]] method of *A* with arguments **`"length"`** and Result(6).
17. Compute the number of additional arguments *item1*, *item2*, etc.
18. If Result(17) is equal to Result(6), go to step 48.
19. If Result(17) is greater than Result(6), go to step 37.
20. Let *k* be Result(5).
21. If *k* is equal to (Result(3)–Result(6)), go to step 31.
22. Call ToString(*k*+Result(6)).
23. Call ToString(*k*+Result(17)).
24. If this object has a property named by Result(22), go to step 25; but if this object has no property named by Result(22), then go to step 28.
25. Call the [[Get]] method of this object with argument Result(22).
26. Call the [[Put]] method of this object with arguments Result(23) and Result(25).
27. Go to step 29.
28. Call the [[Delete]] method of this object with argument Result(23).
29. Increase *k* by 1.
30. Go to step 21.
31. Let *k* be Result(3).
32. If *k* is equal to (Result(3)–Result(6)+Result(17)), go to step 48.
33. Call ToString(*k*–1).
34. Call the [[Delete]] method of this object with argument Result(33).
35. Decrease *k* by 1.
36. Go to step 32.
37. Let *k* be (Result(3)–Result(6)).
38. If *k* is equal to Result(5), go to step 48.
39. Call ToString(*k*+Result(6)–1).
40. Call ToString(*k*+Result(17)–1)
41. If this object has a property named by Result(39), go to step 42; but if this object has no property named by Result(39), then go to step 45.
42. Call the [[Get]] method of this object with argument Result(39).
43. Call the [[Put]] method of this object with arguments Result(40) and Result(42).
44. Go to step 46.
45. Call the [[Delete]] method of this object with argument Result(40).
46. Decrease *k* by 1.
47. Go to step 38.
48. Let *k* be Result(5).
49. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 53.
50. Call the [[Put]] method of this object with arguments ToString(*k*) and Result(49).
51. Increase *k* by 1.
52. Go to step 49.
53. Call the [[Put]] method of this object with arguments **`"length"`** and (Result(3)–Result(6)+Result(17)).
54. Return *A*.

The **`length`** property of the **`splice`** method is **2**.

NOTE

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

*The **`splice`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`splice`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.13 Array.prototype.unshift ([item1 [, item2 [, …]]])

The arguments are prepended to the start of the array, such that their order within the array is the same as the order in which they appear in the argument list.

When the **`unshift`** method is called with zero or more arguments *item1, item2*, etc., the following steps are taken:

1. Call the [[Get]] method of this object with argument **`"length"`**.
2. Call ToUint32(Result(1)).
3. Compute the number of arguments.
4. Let *k* be Result(2).
5. If *k* is zero, go to step 15.
6. Call ToString(*k*–1).
7. Call ToString(*k*+Result(3)–1).
8. If this object has a property named by Result(6), go to step 9; but if this object has no property named by Result(6), then go to step 12.
9. Call the [[Get]] method of this object with argument Result(6).
10. Call the [[Put]] method of this object with arguments Result(7) and Result(9).
11. Go to step 13.
12. Call the [[Delete]] method of this object with argument Result(7).
13. Decrease *k* by 1.
14. Go to step 5.
15. Let *k* be 0.
16. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 21.
17. Call ToString(*k*).
18. Call the [[Put]] method of this object with arguments Result(17) and Result(16).
19. Increase *k* by 1.
20. Go to step 16.
21. Call the [[Put]] method of this object with arguments **`"length"`** and (Result(2)+Result(3)).
22. Return (Result(2)+Result(3)).

The **`length`** property of the **`unshift`** method is **1**.

NOTE
*The **`unshift`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`unshift`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.14 Array.prototype.indexOf (searchElement [, fromIndex])

`indexOf` compares *searchElement* to the elements of the array, in ascending order, using strict equality, and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, the whole array will be searched.

When the **`indexOf`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method of *E* with the argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. Call ToInt32(*fromIndex*) (if *fromIndex* is **undefined** this step produces 0).
6. Let *n* be Result(5).

pratapL 1/19/09 8:57 PM
Comment: From Lars:
I am reasonably confident that this algorithm is not consistent with the one published as part of the JS1.6 spec:

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Objects:Array:indexOf

In particular, note how the latter algorithm only performs the ===
comparison if the index is present in the array, whereas steps 12-14
simply call [[Get]] and use the result. I am also not confident that
the bounds computations are equivalent.

(As it happens I'm not sure that the code on the mozilla site is 100%
what we want either, in particular, unlike array methods in ES3 it
does not appear to bound the length above at 2^32-1, for better or
worse).

Anyway, anything that isn't essentially 100% compatible with the
published 1.6 spec will also not be compatible with what's in ES4.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

7. If *n* is greater than or equal to Result(3) go to step 18.
8. If *n* is greater than or equal to 0, let *k* be *n*, and go to step 11.
9. Let *k* be Result(3) - abs(n).
10. If *k* is less than 0, let k be 0.
11. Call ToString(*k*).
12. Call the [[Get]] method of *E* with the argument Result(11).
13. Perform the comparison SameValue(*searchElement*, Result(12)).
14. If Result(13) is **false** go to step 16.
15. Return *k*.
16. Increase *k* by 1.
17. If *k* is less than Result(3) go to step 11.
18. Return -1.

NOTE
The **`indexOf`** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the* **`indexOf`** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.15 Array.prototype.lastIndexOf (searchElement [, fromIndex])

`lastIndexOf` compares *searchElement* to the elements of the array in descending order using strict equality, and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to the array's length (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, -1 is returned.

When the **`lastIndexOf`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method of *E* with the argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. Call ToInt32(*fromIndex*) (if *fromIndex* is undefined this step produces the same values as Result(3)).
6. Let *n* be Result(5).
7. If *n* is greater than or equal to Result(3), let *k* be Result(3) - 1, and go to step 11.
8. If *n* is greater than or equal to 0, let *k* be *n*, and go to step 11.
9. Let *k* be Result(3) - abs(*n*).
10. If *k* is less than 0 go to step 18.
11. Call ToString(*k*).
12. Call the [[Get]] method of *E* with the argument Result(11).
13. Perform the comparison SameValue(*searchElement*, Result(12)).
14. If Result(13) is false go to step 16.
15. Return *k*.
16. Decrease *k* by 1.
17. If *k* is greater than or equal to 0 go to step 11.
18. Return -1.

NOTE

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

15.4.4.16 Array.prototype.every (callbackfn [, mustBeAbsent])

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **`every`** calls the provided callback, as a function, once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **false**. If such an element is found, **`every`** immediately returns **false**. Otherwise, if *callbackfn* returned **true** for all elements, **`every`** will return

Mark S. Miller 1/19/09 10:24 PM
Deleted: *Whether the* **`lastIndexOf`** *function can be applied successfully to a host object is implementation-dependent.*

Mark S. Miller 1/19/09 10:18 PM
Deleted: thisArg

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

true. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a *mustBeAbsent* parameter is provided, a **TypeError** is thrown.

Mark S. Miller 1/19/09 10:18 PM
Deleted: *thisArg*

Mark S. Miller 1/19/09 10:19 PM
Deleted: it will be used as the **this** value for each invocation of the callback. If it is not provided, **undefined** is used instead

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

`every` does not mutate the array on which it is called.

The range of elements processed by **`every`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`every`** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time **`every`** visits them; elements that are deleted are not visited. **`every`** acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns **true**.

When the **`every`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method of *E* with the argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. N/A
8. Let *k* be 0.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 16.
11. Call the [[Get]] method of *E* with argument Result(9).
12. Call the [[Call]] method of *callbackfn* with arguments Result(11), *k*, and *E*.
13. Call ToBoolean(Result(12)).
14. If Result(13) is **true** go to step 16.
15. Return **false**.
16. Increase *k* by 1.
17. If *k* is less than Result(3) go to step 9.
18. Return **true**.

Mark S. Miller 1/19/09 10:20 PM
Deleted: Let *O* be *thisArg*.

Mark S. Miller 1/19/09 10:21 PM
Deleted: *O* as the **this** value and

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

Mark S. Miller 1/19/09 10:24 PM
Deleted: *. Whether the **`every`** function can be applied successfully to a host object is implementation-dependent*

15.4.4.17 Array.prototype.some (callbackfn [, mustBeAbsent])

Mark S. Miller 1/19/09 10:21 PM
Deleted: **thisArg**

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **`some`** calls the callback, as a function, once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **true**. If such an element is found, **`some`** immediately returns **true**. Otherwise, **`some`** returns **false**. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a *mustBeAbsent* parameter is provided, a **TypeError** is thrown.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

Mark S. Miller 1/19/09 10:21 PM
Deleted: If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of the callback. If it is not provided, **undefined** is used instead.

`some` does not mutate the array on which it is called.

The range of elements processed by **`some`** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **`some`** begins will not be visited by *callbackfn*. If an existing, unvisited element of the array is changed by *callbackfn*, their value as passed to *callbackfn* will be the value at the time that **`some`** visits them; elements that are deleted are not visited.

When the **`some`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. Call the [[Get]] method of *E* with the argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. N/A
8. Let *k* be 0.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 16.
11. Call the [[Get]] method of *E* with argument Result(9).
12. Call the [[Call]] method of *callbackfn* with arguments Result(11), *k*, and *E*.
13. Call ToBoolean(Result(12)).
14. If Result(13) is **false** go to step 16.
15. Return **true**.
16. Increase *k* by 1.
17. If *k* is less than Result(3) go to step 9.
18. Return **false**.

Mark S. Miller 1/19/09 10:22 PM
Deleted: Let *O* be *thisArg*.

Mark S. Miller 1/19/09 10:22 PM
Deleted: *O* as the **this** value and

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

Mark S. Miller 1/19/09 10:24 PM
Deleted: *. Whether the* **`some`** *function can be applied successfully to a host object is implementation-dependent*

15.4.4.18 Array.prototype.forEach (callbackfn [, mustBeAbsent])

Mark S. Miller 1/19/09 10:23 PM
Deleted: **thisArg**

callbackfn should be a function that accepts three arguments. **`forEach`** calls the provided callback, as a function, once for each element present in the array, in ascending order. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a *mustBeAbsent* parameter is provided, a **TypeError** is thrown.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

Mark S. Miller 1/19/09 10:22 PM
Deleted: If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of the callback. If it is not provided, **undefined** is used instead.

`forEach` does not mutate the array on which it is called.

The range of elements processed by **`forEach`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`forEach`** begins will not be visited by *callbackfn*. If existing elements of the array are changed, or deleted, their value as passed to callback will be the value at the time **`forEach`** visits them; elements that are deleted are not visited.

When the **`forEach`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method of E with the argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 14.
5. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. N/A
8. Let *k* be 0.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 13.
11. Call the [[Get]] method of *E* with argument Result(9).
12. Call the [[Call]] method of *callbackfn* with arguments Result(11), *k*, and *E*.
13. Increase *k* by 1.
14. If *k* is less than Result(3) go to step 9.
15. Return.

Mark S. Miller 1/19/09 10:23 PM
Deleted: Let *O* be *thisArg*.

Mark S. Miller 1/19/09 10:23 PM
Deleted: *O* as **this** value and

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

Mark S. Miller 1/19/09 10:24 PM
Deleted: *Whether the* **`forEach`** *function can be applied successfully to a host object is implementation-dependent.*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.4.4.19 Array.prototype.map (callbackfn [, mustBeAbsent])

Mark S. Miller 1/19/09 10:25 PM
Deleted: thisArg

callbackfn should be a function that accepts three arguments. **`map`** calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array from the results. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a *mustBeAbsent* parameter is provided, a **TypeError** is thrown.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

Mark S. Miller 1/19/09 10:25 PM
Deleted: If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of the callback. If it is not provided, **undefined** is used instead.

`map` does not mutate the array on which it is called.

The range of elements processed by **`map`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`map`** begins will not be visited by *callbackfn*. If existing elements of the array are changed, or deleted, their value as passed to *callbackfn* will be the value at the time **`map`** visits them; elements that are deleted are not visited.

When the **`map`** method is called with one or two arguments, the following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
2. Let *n* be 0.
3. Let *E* be this object
4. Call the [[Get]] method of *E* with the argument **`"length"`**.
5. Call ToUint32(Result(4)).
6. If Result(5) is 0 go to step 20.
7. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
8. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
9. N/A
10. Let *k* be 0.
11. Call ToString(*k*).
12. If *E* does not have a property named by Result(11), go to step 19.
13. Call the [[Get]] method of *E* with argument Result(11).
14. Call the [[Call]] method of *callbackfn* with arguments Result(13), *k*, and *E*.
15. Call ToString(*n*).
16. Call the [[Put]] method of *A* with the argument Result(14) and Result(15).
17. Increase *n* by 1.
18. Increase *k* by 1.
19. If *k* is less than Result(5) go to step 11.
20. Return *A*.

Mark S. Miller 1/19/09 10:25 PM
Deleted: Let *O* be *thisArg*.

Mark S. Miller 1/19/09 10:25 PM
Deleted: *O* as the **this** value and

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

Mark S. Miller 1/19/09 10:25 PM
Deleted: *Whether the **map** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.20 Array.prototype.filter (callbackfn [, mustBeAbsent])

Mark S. Miller 1/19/09 10:25 PM
Deleted: thisArg

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **`filter`** calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array of all the values for which *callbackfn* returns **true**. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a *mustBeAbsent* parameter is provided, a **TypeError** is thrown.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

Mark S. Miller 1/19/09 10:26 PM
Deleted: If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of the callback. If it is not provided, **undefined** is used instead.

`filter` does not mutate the array on which it is called.

The range of elements processed by **`filter`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`filter`** begins will not be visited by *callbackfn*. If existing

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

elements of the array are changed, or deleted, their value as passed to *callbackfn* will be the value at the time **`filter`** visits them; elements that are deleted are not visited.

When the **`filter`** method is called with one or two arguments, the following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
2. Let *n* be 0.
3. Let *E* be this object
4. Call the [[Get]] method of *E* with the argument **`"length"`**.
5. Call ToUint32(Result(4)).
6. If Result(5) is 0 go to step 22.
7. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
8. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
9. N/A
10. Let *k* be 0.
11. Call ToString(*k*).
12. If *E* does not have a property named by Result(11), go to step 20.
13. Call the [[Get]] method of *E* with argument Result(11).
14. Call the [[Call]] method of *callbackfn* with arguments Result(13), *k*, and *E*.
15. Call ToBoolean(Result(14)).
16. If Result(15) is **false** go to step 20.
17. Call ToString(*n*).
18. Call the [[Put]] method of *A* with the argument Result(13) and Result(17).
19. Increase *n* by 1.
20. Increase *k* by 1.
21. If *k* is less than Result(5) go to step 11.
22. Return *A*.

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

Mark S. Miller 1/19/09 10:26 PM
Deleted: Let *O* this *thisArg*.

Mark S. Miller 1/19/09 10:26 PM
Deleted: *O* as the **this** value and

Mark S. Miller 1/19/09 10:26 PM
Deleted: *. Whether the **`filter`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.21 Array.prototype.reduce (callbackfn [, initialValue])

callbackfn should be a function that takes four arguments. **`reduce`** calls the callback, as a function, once for each element present in the array, in ascending order.

callbackfn is called with four arguments: the previousValue (or value from the previous call to *callbackfn*), the currentValue (value of the current element), the currentIndex, and the Array object being traversed. The first time that callback is called, the previousValue and currentValue can be one of two values. If an *initialValue* was provided in the call to **`reduce`**, then previousValue will be equal to *initialValue* and currentValue will be equal to the first value in the array. If no *initialValue* was provided, then previousValue will be equal to the first value in the array and currentValue will be equal to the second.

`reduce` does not mutate the array on which it is called.

The range of elements processed by **`reduce`** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **`reduce`** begins will not be visited by *callbackfn*. If an existing, unvisisted element is changed by *callbackfn*, their value as passed to *callbackfn* will be the value at the time **`reduce`** visits them; elements that are deleted are not visited.

When the **`reduce`** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method on *E* with argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. If Type(*callbackfn*) is not Object, throw a **TypeError** exception.
5. If IsCallable(*callbackfn)* is **false**, throw a **TypeError** exception.
6. If Result(3) is 0 and *initialValue* is not supplied throw a **TypeError** exception.
7. Let *k* be 0.
8. If *initialValue* is supplied let *P* be *initialValue* and go to step 17.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 14.
11. Call the [[Get]] method on *E* with the argument Result(9).
12. Increase *k* by 1.
13. Let *P* be Result(11) and go to step 17.
14. Increase *k* by 1.
15. If *k* < Result(3) go to step 9.
16. Throw a **TypeError** exception.
17. Call ToString(*k*).
18. If *E* does not have a property named by Result(17), go to step 22.
19. Call the [[Get]] method of *E* with the argument Result(17).
20. Call the [[Call]] method on *callbackfn* with **null** as the this value and arguments *P*, Result(19), *k*, *E*.
21. Let *P* be Result(20).
22. Increase *k* by 1.
23. If *k* < Result(3) go to step 17.
24. Return *P*.

NOTE
The ***reduce*** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.*

Mark S. Miller 1/19/09 10:27 PM
Deleted: *Whether the* ***reduce*** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.22 Array.prototype.reduceRight (callbackfn [, initialValue])

callbackfn should be a function that takes four arguments. **reduceRight** calls the callback, as a function, once for each element present in the array, in descending order.

callbackfn is called with four arguments: the previousValue (or value from the previous call to *callbackfn*), the currentValue (value of the current element), the currentIndex, and the Array object being traversed. The first time the function is called, the previousValue and currentValue can be one of two values. If an *initialValue* was provided in the call to **reduceRight**, then previousValue will be equal to *initialValue* and currentValue will be equal to the last value in the array. If no *initialValue* was provided, then previousValue will be equal to the last value in the array and currentValue will be equal to the second-to-last value.

reduceRight does not mutate the array on which it is called.

The range of elements processed by **reduceRight** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **reduceRight** begins will not be visited by *callbackfn*. If an existing, unvisited element is changed by *callbackfn*, their value as passed to *callbackfn* will be the value at the time **reduceRight** visits them; elements that are deleted are not visited.

When the **reduceRight** method is called with one or two arguments, the following steps are taken:

1. Let *E* be this object.
2. Call the [[Get]] method on *E* with argument **"length"**.
3. Call ToUint32(Result(2)).
4. If Type(*callbackfn*) is not Object throw a **TypeError** exception.
5. If IsCallable(*callbackfn)* is **false**, throw a **TypeError** exception.
6. If Result(3) is 0 and *initialValue* is not supplied throw a **TypeError** exception.
7. Let *k* be Result(3) – 1.
8. If *initialValue* is supplied let *P* be *initialValue* and go to step 17.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 14.
11. Call the [[Get]] method on *E* with the argument Result(9).
12. Decrease *k* by 1.
13. Let *P* be Result(11) and go to step 17.
14. Decrease *k* by 1.
15. If *k* is greater than or equal to 0 go to step 9.
16. Throw a **TypeError** exception.
17. Call ToString(*k*).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

18. If *E* does not have a property named by Result(17), go to step 22.
19. Call the [[Get]] method of *E* with the argument Result(17).
20. Call the [[Call]] method on *callbackfn* with **null** as the **this** value and arguments *P*, Result(19), *k*, *E*.
21. Let *P* be Result(20).
22. Decrease *k* by 1.
23. If *k* is greater than or equal to 0 go to step 17.
24. Return *P*.

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

15.4.5 Properties of Array Instances

Array instances inherit properties from the Array prototype object, their [[Class]] property value is `"Array"`, and they have the internal property [[IsArray]] value true. Array instances also also have the following properties.

15.4.5.1 [[ThrowingPut]] (P, V)

Array objects use a variation of the [[ThrowingPut]] method used for other native SES objects (8.6.2.10).

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

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

1. Call the [[CanPut]] method of A with name P.
2. If Result(1) is **false**, then throw a **TypeError** exception.
3. If A doesn't have a property with name P, go to step 7.
4. If P is "length", go to step 12.
5. Set the value of property P of A to V.
6. Go to step 8.
7. Create a property with name P, set its value to V and give it empty attributes.
8. If P is not an array index, return.
9. If ToUint32(P) is less than the value of the length property of A, then return.
10. Change (or set) the value of the length property of A to ToUint32(P)+1.
11. Return.
12. Compute ToUint32(V).
13. If Result(12) is not equal to ToNumber(V), throw a RangeError exception.
14. 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 an own property (a non-inherited property) named ToString(k), then delete that property.
15. Set the value of property P of A to Result(12).
16. Return.

15.4.5.2 length

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 attributes { [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.5 String Objects

15.5.1 The String Constructor Called as a Function

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

15.5.1.1 String ([value])

Returns a string value (not a String object) computed by ToString(*value*). If *value* is not supplied, the empty string `""` is returned.

Mark S. Miller 1/19/09 10:27 PM
Deleted: . *Whether the* `reduceRight` *function can be applied successfully to a host object is implementation-dependent*

Mark S. Miller 1/19/09 10:28 PM
Deleted: ,

Mark S. Miller 1/19/09 10:28 PM
Deleted: Throw

Mark S. Miller 1/19/09 5:03 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 10:28 PM
Deleted: ,

Mark S. Miller 1/19/09 10:28 PM
Deleted: , and *Throw* is a boolean flag

Mark S. Miller 1/19/09 10:29 PM
Deleted: ,

Mark S. Miller 1/19/09 10:29 PM
Deleted: , and Boolean flag *Throw*

Mark S. Miller 1/19/09 10:29 PM
Deleted:
If *Throw* is **true**,

Mark S. Miller 1/19/09 10:29 PM
Deleted:

Mark S. Miller 1/19/09 10:29 PM
Deleted: <#>Else return.

Mark S. Miller 1/19/09 10:31 PM
Comment: Should call [[Delete]] or something, so that if the properties in question are not deletable, this fails without deleting them. Likewise for ES3.1.

pratapL 1/19/09 8:57 PM
Comment: From AWB:
This probably should be rewritten using the structured notation

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.5.2 The String Constructor

When **`String`** is called as part of a **`new`** expression, it throws a **TypeError**.

15.5.3 Properties of the String Constructor

The value of the internal [[Parent]] property of the String constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **1**), the String constructor has the following properties:

15.5.3.1 N/A

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 (9.7) and regarding the resulting 16-bit integer as the code unit value of a character. If no arguments are supplied, the result is the empty string.

The **`length`** property of the **`fromCharCode`** function is **1**.

15.5.4 Properties of the String Prototype Object

The String prototype object is itself a String object (its [[Class]] is **`"String"`**) whose value is an empty string.

The value of the internal [[Parent]] property of the String prototype object is the Object prototype object (15.2.3.1).

15.5.4.1 N/A

15.5.4.2 String.prototype.toString ( )

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

The **`toString`** function is not generic; it throws a **TypeError** exception if its **this** value is not a String object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.5.4.3 String.prototype.valueOf ( )

Returns this string value.

The **`valueOf`** function is not generic; it throws a **TypeError** exception if its **this** value is not a String object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.5.4.4 String.prototype.charAt (pos)

Returns a string containing the character at position *pos* in the string resulting from converting this object to a 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 that is an integer, then the result of **`x.charAt(`***pos***`)`** is equal to the result of **`x.substring(`***pos***`,`** *pos***`+1)`**.

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

1. Call ToString, giving it the **this** value as its argument.
2. Call ToInteger(*pos*).
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return the empty string.
5. 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

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

Mark S. Miller 1/19/09 10:32 PM
Deleted: is a constructor: it initialises the newly created object.
15.5.2.1 new String ([value])
The [[Prototype]] property of the newly constructed object is set to the original String prototype object, the one that is the initial value of **`String.prototype`** (15.5.3.1).
The [[Class]] property of the newly constructed object is set to **`"String"`**. The [[Extensible]] property of the newly constructed object is set to **true**.
The [[PrimitiveValue]] property of the newly constructed object is set to ToString(*value*), or to the empty string if *value* is no

Mark S. Miller 1/19/09 10:32 PM
Deleted: t supplied

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:33 PM
Deleted: **String.prototype**

Mark S. Miller 1/19/09 10:33 PM
Deleted: The initial value of **`String.prototype`** is the String prototype object (15.5.4).
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:33 PM
Deleted: **String.prototype.constructor**

Mark S. Miller 1/19/09 10:33 PM
Deleted: The initial value of **`String.prototype.constructor`** is the built-in **`String`** constructor.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.5.4.5 String.prototype.charCodeAt (pos)

Returns a number (a nonnegative integer less than 2^{16}) representing the code unit value of the character at position *pos* in the string resulting from converting this object to a 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:

1. Call ToString, giving it the **this** value as its argument.
2. Call ToInteger(*pos*).
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return **NaN**.
5. Return a value of Number type, whose value is the code unit value of the character at position Result(2) in the string Result(1), 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
The `charCodeAt` *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.*

15.5.4.6 String.prototype.concat ([string1 [, string2 [, …]]])

When the `concat` method is called with zero or more arguments *string1*, *string2*, etc., it returns a string consisting of the characters of this object (converted to a string) followed by the characters of each of *string1*, *string2*, etc. (where each argument is converted to a string). The result is a string value, not a String object. The following steps are taken:

1. Call ToString, giving it the **this** value as its argument.
2. Let *R* be Result(1).
3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
4. Call ToString(Result(3)).
5. Let *R* be the string value consisting of the characters in the previous value of *R* followed by the characters Result(4).
6. Go to step 3.
7. Return *R*.

The `length` property of the `concat` method is **1**.

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

15.5.4.7 String.prototype.indexOf (searchString, position)

If *searchString* appears as a substring of the result of converting this object to a string, at one or more positions that are greater than or equal to *position*, then the index of the smallest such position is returned; otherwise, `-1` is returned. If *position* is **undefined**, 0 is assumed, so as to search all of the string.

The `indexOf` method takes two arguments, *searchString* and *position*, and performs the following steps:

1. Call ToString, giving it the **this** value as its argument.
2. Call ToString(*searchString*).
3. Call ToInteger(*position*). (If *position* is **undefined**, this step produces the value `0`).
4. Compute the number of characters in Result(1).
5. Compute min(max(Result(3), 0), Result(4)).
6. Compute the number of characters in the string that is Result(2).
7. 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`.
8. Return Result(7).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The `length` property of the `indexOf` method is **1**.

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

15.5.4.8 String.prototype.lastIndexOf (searchString, position)

If *searchString* appears as a substring of the result of converting this object to a string at one or more positions that are smaller than or equal to *position*, then the index of the greatest such position is returned; otherwise, **`-1`** is returned. If *position* is **undefined**, the length of the string value is assumed, so as to search all of the string.

The **`lastIndexOf`** method takes two arguments, *searchString* and *position*, and performs the following steps:

1. Call ToString, giving it the **this** value as its argument.
2. Call ToString(*searchString*).
3. Call ToNumber(*position*). (If *position* is **undefined**, this step produces the value **NaN**).
4. If Result(3) is **NaN**, use **+∞**; otherwise, call ToInteger(Result(3)).
5. Compute the number of characters in Result(1).
6. Compute min(max(Result(4), 0), Result(5)).
7. Compute the number of characters in the string that is Result(2).
8. Compute the largest possible nonnegative 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`**.
9. Return Result(8).

The `length` property of the `lastIndexOf` method is **1**.

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

15.5.4.9 String.prototype.localeCompare (that)

When the **`localeCompare`** method is called with one argument *that*, it returns a number other than **NaN** that represents the result of a locale-sensitive string comparison of this object (converted to a string) with *that* (converted to a string). The two strings are compared in an implementation-defined fashion. The result is intended to order strings in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether **this** comes before *that* in the sort order, the strings are equal, or **this** comes after *that* in the sort order, respectively.

The **`localeCompare`** method, if considered as a function of two arguments **this** and *that*, is a consistent comparison function (as defined in 15.4.4.11) on the set of all strings. Furthermore, **`localeCompare`** returns **`0`** or **`–0`** when comparing two strings that are considered canonically equivalent by the Unicode standard.

The actual return values are left implementation-defined to permit implementers to encode additional information in the result value, but the function is required to define a total ordering on all strings and to return **`0`** when comparing two strings that are considered canonically equivalent by the Unicode standard.

NOTE 1
*The **`localeCompare`** method itself is not directly suitable as an argument to **`Array.prototype.sort`** because the latter requires a function of two arguments.*

NOTE 2
This function is intended to rely on whatever language-sensitive comparison functionality is available to the SES environment from the host environment, and to compare according to the rules of the host environment's current locale. It is strongly recommended that this function treat strings that are canonically equivalent according to the Unicode standard as identical (in other words, compare the

Mark S. Miller 1/19/09 5:03 PM
Deleted: *ECMAScript*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

strings as if they had both been converted to Normalised Form C or D first). It is also recommended that this function not honour Unicode compatibility equivalences or decompositions.

If no language-sensitive comparison at all is available from the host environment, this function may perform a bitwise comparison.

NOTE 3
The **`localeCompare`** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

NOTE 4
The second parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.5.4.10 String.prototype.match (regexp)

If *regexp* is not an object whose [[Class]] property is **`"RegExp"`**, it is replaced with the result of the expression **`new RegExp(`***regexp***`)`**. Let *string* denote the result of converting the **this** value to a string. Then do one of the following:

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

If *regexp*.**`global`** is **true**: Set the *regexp*.**`lastIndex`** property to 0 and invoke **`RegExp.prototype.exec`** repeatedly until there is no match. If there is a match with an empty string (in other words, if the value of *regexp*.**`lastIndex`** is left unchanged), increment *regexp*.**`lastIndex`** by 1. Let *n* be the number of matches. If *n*=0, then the value returned is `null`; otherwise, the value returned is an array with the **`length`** property set to *n* and properties 0 through *n*–1 corresponding to the first elements of the results of all matching invocations of **`RegExp.prototype.exec`**.

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

15.5.4.11 String.prototype.replace (searchValue, replaceValue)

Let *string* denote the result of converting the **this** value to a string.

If *searchValue* is a regular expression (an object whose [[Class]] property is **`"RegExp"`**), do the following: If *searchValue*.global is **false**, then search *string* for the first match of the regular expression *searchValue*. If *searchValue*.global is **true**, then search *string* for all matches of the regular expression *searchValue*. Do the search in the same manner as in **`String.prototype.match`**, including the update of *searchValue*.**`lastIndex`**. Let *m* be the number of left capturing parentheses in *searchValue* (*NcapturingParens* as specified in 15.10.2.1).

If *searchValue* is not a regular expression, let *searchString* be ToString(*searchValue*) and search *string* for the first occurrence of *searchString*. Let *m* be 0.

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Characters	***Replacement text***
`$$`	**`$`**
`$&`	The matched substring.
`` $` ``	The portion of *string* that precedes the matched substring.
`$'`	The portion of *string* that follows the matched substring.
`$`*n*	The *n*th capture, where *n* is a single digit 1-9 and **`$`***n* is not followed by a decimal digit. If *n*≤*m* and the *n*th capture is **undefined**, use the empty string instead. If *n*>*m*, the result is implementation-defined.
`$`*nn*	The *nn*th capture, where *nn* is a two-digit decimal number 01-99. If *nn*≤*m* and the *nn*th capture is **undefined**, use the empty string instead. If *nn*>*m*, the result is implementation-defined.

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

15.5.4.12 String.prototype.search (regexp)

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

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

NOTE 1
This method ignores the ***`lastIndex`*** *and* ***`global`*** *properties of regexp. The* ***`lastIndex`*** *property of regexp is left unchanged.*

NOTE 2
The ***`search`*** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

15.5.4.13 String.prototype.slice (start, end)

The **`slice`** method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a string, starting from character position *start* and running to, but not including, character position *end* (or through the end of the string if *end* is **undefined**). If *start* is negative, it is treated as (*sourceLength*+*start*) where *sourceLength* is the length of the string. If *end* is negative, it is treated as (*sourceLength*+*end*) where *sourceLength* is the length of the string. The result is a string value, not a String object. The following steps are taken:

1. Call ToString, giving it the **this** value as its argument.
2. Compute the number of characters in Result(1).
3. Call ToInteger(*start*).
4. If *end* is **undefined**, use Result(2); else use ToInteger(*end*).
5. If Result(3) is negative, use max(Result(2)+Result(3),0); else use min(Result(3),Result(2)).
6. If Result(4) is negative, use max(Result(2)+Result(4),0); else use min(Result(4),Result(2)).
7. Compute max(Result(6)–Result(5),0).
8. Return a string containing Result(7) consecutive characters from Result(1) beginning with the character at position Result(5).

The **`length`** property of the **`slice`** method is **2**.

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.5.4.14 String.prototype.split (separator, limit)

Returns an Array object into which substrings of the result of converting this object to a string have been stored. The substrings are determined by searching from left to right for occurrences of *separator*; these occurrences are not part of any substring in the returned array, but serve to divide up the string value. The value of *separator* may be a string of any length or it may be a RegExp object (i.e., an object whose [[Class]] property is **"RegExp"**; see 15.10).

The value of *separator* may be an empty string, an empty regular expression, or a regular expression that can match an empty string. In this case, *separator* does not match the empty substring at the beginning or end of the input string, nor does it match the empty substring at the end of the previous separator match. (For example, if *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 *separator* is a regular expression, only the first match at a given position of the **this** string is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, **"ab".split(/a*?/)** evaluates to the array **["a","b"]**, while **"ab".split(/a*/)** evaluates to the array **["","b"]**.)

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

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

If *separator* is **undefined**, then the result array contains just one string, which is the **this** value (converted to a string). If *limit* is not **undefined**, then the output array is truncated so that it contains no more than *limit* elements.

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

1. Let *S* = ToString(**this**).
2. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
3. If *limit* is **undefined**, let *lim* = 2^{32}–1; else let *lim* = ToUint32(*limit*).
4. Let *s* be the number of characters in *S*.
5. Let *p* = 0.
6. If *separator* is a RegExp object (its [[Class]] is **"RegExp"**), let *R* = *separator*; otherwise let *R* = ToString(*separator*).
7. If *lim* = 0, return *A*.
8. If *separator* is **undefined**, go to step 33.
9. If *s* = 0, go to step 31.
10. Let *q* = *p*.
11. If *q* = *s*, go to step 28.
12. Call *SplitMatch*(*R*, *S*, *q*) and let *z* be its MatchResult result.
13. If *z* is **failure**, go to step 26.
14. *z* must be a State. Let *e* be *z*'s *endIndex* and let *cap* be *z*'s *captures* array.
15. If *e* = *p*, go to step 26.
16. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *q* (exclusive).
17. Call the [[Put]] method of A with arguments A.length and T.
18. If A.length = lim, return A.
19. Let p = e.
20. Let *i* = 0.
21. If *i* is equal to the number of elements in *cap*, go to step 10.
22. Let *i* = *i*+1.
23. Call the [[Put]] method of *A* with arguments *A*.**length** and *cap*[*i*].

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

24. If *A*.**`length`** = *lim*, return *A*.
25. Go to step 21.
26. Let *q* = *q*+1.
27. Go to step 11.
28. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *s* (exclusive).
29. Call the [[Put]] method of *A* with arguments *A*.**`length`** and *T*.
30. Return *A*.
31. Call *SplitMatch*(*R*, *S*, 0) and let *z* be its MatchResult result.
32. If *z* is not **failure**, return *A*.
33. Call the [[Put]] method of *A* with arguments **`"0"`** and *S*.
34. Return *A*.

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

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

The **`length`** property of the **`split`** method is **2**.

NOTE 1
*The **`split`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

NOTE 2
*The **`split`** method ignores the value of separator**`.global`** for separators that are RegExp objects.*

15.5.4.15 String.prototype.substring (start, end)

The **substring** method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a string, starting from character position *start* and running to, but not including, character position *end* of the string (or through the end of the string is *end* is **undefined**). 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.

The following steps are taken:

1. Call ToString, giving it the **this** value as its argument.
2. Compute the number of characters in Result(1).
3. Call ToInteger(*start*).
4. If *end* is **undefined**, use Result(2); else use ToInteger(*end*).
5. Compute min(max(Result(3), 0), Result(2)).
6. Compute min(max(Result(4), 0), Result(2)).
7. Compute min(Result(5), Result(6)).
8. Compute max(Result(5), Result(6)).
9. 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.

The **`length`** property of the **`substring`** method is **2**.

NOTE

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

15.5.4.16 String.prototype.toLowerCase ( )

If this object is not already a string, it is converted to a string. The characters in that string are converted one by one to lower case. The result is a string value, not a String object.

The characters are converted one by one. The result of each conversion is the original character, unless that character has a Unicode lowercase equivalent, in which case the lowercase equivalent is used instead.

NOTE 1
The result should be derived according to the case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file, but also the SpecialCasings.txt file that accompanies it in Unicode 2.1.8 and later).

NOTE 2
The ***`toLowerCase`*** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

15.5.4.17 String.prototype.toLocaleLowerCase ( )

This function works exactly the same as **`toLowerCase`** except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1
The ***`toLocaleLowerCase`*** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.5.4.18 String.prototype.toUpperCase ( )

This function behaves in exactly the same way as **`String.prototype.toLowerCase`**, except that characters are mapped to their *uppercase* equivalents as specified in the Unicode Character Database.

NOTE 1
Because both ***`toUpperCase`*** *and* ***`toLowerCase`*** *have context-sensitive behaviour, the functions are not symmetrical. In other words,* ***`s.toUpperCase().toLowerCase()`*** *is not necessarily equal to* ***`s.toLowerCase()`****.*

NOTE 2
The ***`toUpperCase`*** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

15.5.4.19 String.prototype.toLocaleUpperCase ( )

This function works exactly the same as **`toUpperCase`** except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1
The ***`toLocaleUpperCase`*** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.5.4.20 String.prototype.trim ( )

If this object is not already a string, it is converted to a string. The result is a copy of the string with both leading and trailing white space removed. The definition of white space is the union of *WhiteSpace* and *LineTerminator*. The result is a string value, not a String object.

NOTE

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

15.5.4.21 String.prototype.toJSON (key)

When the toJSON method is called with argument key, the following steps are taken:

1. Let *O* be this object.
2. Call the [[Get]] method of *O* with argument "**`valueOf`**".
3. If IsCallable(Result(2)) is **false**, go to step 6
4. Call the [[Call]] method of Result(2) with *O* as the **this** value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a **TypeError** exception.

NOTE
The **`toJSON`** *function is intentionally generic; it does not require that its* ***this*** *value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.*
An object is free to use the argument 'key' that is passed in to filter its stringification.

15.5.5 N/A

15.6 Boolean Objects

15.6.1 The Boolean Constructor Called as a 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*).

15.6.2 The Boolean Constructor

When **`Boolean`** is called as part of a **`new`** expression it is a constructor: it throws a TypeError.

15.6.3 Properties of the Boolean Constructor

The value of the internal [[Parent]] property of the Boolean constructor is the Function prototype object (15.3.4).

15.6.4 Properties of the Boolean Prototype Object

The Boolean prototype object is itself a Boolean object (its [[Class]] is **`"Boolean"`**) whose value is **false**.

The value of the internal [[Parent]] property of the Boolean prototype object is the Object prototype object (15.2.3.1).

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; a **TypeError** exception is thrown if the **this** value is not an object for which the value of the internal [[Class]] property is **`"Boolean"`**. Also, the phrase "this boolean value" refers to the boolean value represented by this Boolean object, that is, the value of the internal [[PrimitiveValue]] property of this Boolean object.

15.6.4.1 N/A

15.6.4.2 Boolean.prototype.toString ( )

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

The **`toString`** function is not generic; it throws a **TypeError** exception if its **this** value is not a Boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

Mark S. Miller 1/19/09 10:39 PM
Comment: This documents both what the toJSON contract is as well as how strings obey that contract. Should simplify here for just string behavior, and elsewhere define the general contract.

Mark S. Miller 1/19/09 10:39 PM
Deleted: **Properties of String Instances**

Mark S. Miller 1/19/09 10:39 PM
Deleted: String instances inherit properties from the String prototype object and also have a [[PrimitiveValue]] property and a **`length`** property.
The [[PrimitiveValue]] property is the string value represented by this String object.
15.5.5.1 length
The number of characters in the String value represented by this String object.
Once a String object is created, this property is unchanging. It has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.
15.5.5.2 [[GetOwnProperty]] (P)
String objects use a variation of the [[GetOwnProperty]] method used for other native ECMAScript objects (8.6.2.8).
Assume *S* is a String object and *P* is a string.
When the [[GetOwnProperty]] method of *S* is called with property name *P*, the following steps are taken:
<#>Call the default [[GetOwnProperty]] method (8.6.2.8) with *S* as the **this** value and argument *P*.
<#>If Result(1) is not **undefined** return Result(1).
<#>If P is not an array index (15.4), return **undefined**.
<#>Call ToString, giving *S* as its argument.
<#>Call ToInteger(*P*).
<#>Compute the number of characters in Result(4).
<#>If Result(5) is less than 0 or is not less than Result(6), return **undefined**.
<#>Create a string of length 1, containing one character from Result(4), namely the character at position Result(5), where the first (leftmost) character in Result(4) is considered to be at position 0, the next one at position 1, and so on.
<#>Return a Property Descriptor { [[Value]]: Result(8), [[Enumerable]]: **false**, [[Writable]]: **false**, [[Configurable]]: **false** }

Mark S. Miller 1/19/09 10:40 PM
Deleted: initialises the newly created object

Mark S. Miller 1/19/09 10:41 PM
Deleted: **15.6.2.1 new Boolean (value)**
The [[Prototype]] property of the newly constructed object is set to the original Boolean prototype object, the one that is the initial value of **`Boolean.prototype`** (15. ... [34]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:42 PM
Deleted: Besides the internal properties and the **`length`** property (whose value is **1**), the Boolean constructor has the following property:
15.6.3.1 Boolean.prototype ... [35]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:43 PM
Deleted: **Boolean.prototype.constructor**

Mark S. Miller 1/19/09 10:43 PM
Deleted: The initial value of **`Boolean.prototype.constructor`** is the built-in **`Boolean`** constructor.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.6.4.3 Boolean.prototype.valueOf ( )

Returns this boolean value.

The **`valueOf`** function is not generic; it throws a **TypeError** exception if its **this** value is not a Boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.6.4.4 Boolean.prototype.toJSON (key)

When the toJSON method is called with argument key, the following steps are taken:

1. Let *O* be this object.
2. Call the [[Get]] method of *O* with argument "**`valueOf`**".
3. If IsCallable(Result(2)) is **false**, go to step 6
4. Call the [[Call]] method of Result(2) with *O* as the **this** value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a **TypeError** exception.

NOTE
*The **`toJSON`** function is intentionally generic; it does not require that its **this** value be a Boolean object. Therefore, it can be transferred to other kinds of objects for use as a method.*
An object is free to use the argument 'key' that is passed in to filter its stringification.

15.7 Number Objects

15.7.1 The Number Constructor Called as a 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*) if *value* was supplied, else returns **+0**.

15.7.2 The Number Constructor

When **`Number`** is called as part of a **`new`** expression it is a constructor: it throws a **TypeError**.

15.7.3 Properties of the Number Constructor

The value of the internal [[Parent]] property of the Number constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **1**), the Number constructor has the following property:

15.7.3.1 N/A

15.7.3.2 Number.MAX_VALUE

The value of **`Number.MAX_VALUE`** is the largest positive finite value of the number type, which is approximately $1.7976931348623157 \times 10^{308}$.

15.7.3.3 Number.MIN_VALUE

The value of **`Number.MIN_VALUE`** is the smallest positive value of the number type, which is approximately 5×10^{-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

The Number prototype object is itself a Number object (its [[Class]] is **`"Number"`**) whose value is +0.

The value of the internal [[Parent]] property of the Number prototype object is the Object prototype object (15.2.3.1).

Mark S. Miller 1/19/09 10:44 PM
Deleted: **15.6.5 Properties of Boolean Instances**
Boolean instances have no special properties beyond those inherited from the Boolean prototype object.

Mark S. Miller 1/19/09 10:44 PM
Deleted: initialises the newly created object.

Mark S. Miller 1/19/09 10:44 PM
Deleted: **15.7.2.1 new Number ([value])**
The [[Prototype]] property of the newly constructed object is set to the original Number prototype object, the one that is the initial value of **`Number.prototype`** (15.7.3.1).
The [[Class]] property of the newly constructed object is set to **`"Number"`**.
The [[PrimitiveValue]] property of the newly constructed object is set to ToNumber(*value*) if *value* was supplied, else to +0.
The [[Extensible]] property of the newly constructed object is set to **true**.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:45 PM
Deleted: **Number.prototype**

Mark S. Miller 1/19/09 10:45 PM
Deleted: The initial value of **`Number.prototype`** is the Number prototype object (15.7.4).
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:45 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:45 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:45 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:45 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 10:45 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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; a **TypeError** exception is thrown if the **this** value is not an object for which the value of the internal [[Class]] property is **"Number"**. Also, the phrase "this number value" refers to the number value represented by this Number object, that is, the value of the internal [[PrimitiveValue]] property of this Number object.

15.7.4.1 N/A

Mark S. Miller 1/19/09 10:46 PM
Deleted: Number.prototype.constructor

15.7.4.2 Number.prototype.toString (radix)

Mark S. Miller 1/19/09 10:46 PM
Deleted: The initial value of `Number.prototype.constructor` is the built-in `Number` constructor.

If *radix* is the number 10 or **undefined**, then this number value is given as an argument to the ToString operator; the resulting string value is returned.

If *radix* is an integer from 2 to 36, but not 10, the result is a string, the choice of which is implementation-dependent.

The **toString** function is not generic; it throws a **TypeError** exception if its **this** value is not a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.4.3 Number.prototype.toLocaleString()

Produces a string value that represents the value of the Number formatted according to the conventions of the host environment's current locale. This function is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as **toString**.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.7.4.4 Number.prototype.valueOf ( )

Returns this number value.

The **valueOf** function is not generic; it throws a **TypeError** exception if its **this** value is not a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.4.5 Number.prototype.toFixed (fractionDigits)

Return a string containing the number represented in fixed-point notation with *fractionDigits* digits after the decimal point. If *fractionDigits* is **undefined**, 0 is assumed. Specifically, perform the following steps:

1. Let *f* be ToInteger(*fractionDigits*). (If *fractionDigits* is **undefined**, this step produces the value **0**).
2. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
3. Let *x* be this number value.
4. If *x* is **NaN**, return the string **"NaN"**.
5. Let *s* be the empty string.
6. If $x \geq 0$, go to step 9.
7. Let *s* be **"-"**.
8. Let $x = -x$.
9. If $x \geq 10^{21}$, let *m* = ToString(*x*) and go to step 20.
10. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such *n*, pick the larger *n*.
11. If $n = 0$, let *m* be the string **"0"**. Otherwise, let *m* be the string consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
12. If $f = 0$, go to step 20.
13. Let *k* be the number of characters in *m*.
14. If $k > f$, go to step 18.
15. Let *z* be the string consisting of $f+1-k$ occurrences of the character '0'.
16. Let *m* be the concatenation of strings *z* and *m*.
17. Let $k = f + 1$.
18. Let a be the first $k-f$ characters of *m*, and let *b* be the remaining *f* characters of *m*.
19. Let *m* be the concatenation of the three strings *a*, **"."**, and *b*.
20. Return the concatenation of the strings *s* and *m*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The **length** property of the **toFixed** method is **1**.

If the **toFixed** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **toFixed** for values of *fractionDigits* less than 0 or greater than 20. In this case **toFixed** would not necessarily throw **RangeError** for such values.

NOTE
The output of **toFixed** *may be more precise than* **toString** *for some values because toString only prints enough significant digits to distinguish the number from adjacent number values. For example,* ***(1000000000000000128).toString()*** *returns* ***"1000000000000000100"****, while* ***(1000000000000000128).toFixed(0)*** *returns* ***"1000000000000000128"****.*

15.7.4.6 Number.prototype.toExponential (fractionDigits)

Return a string containing the number represented in exponential notation with one digit before the significand's decimal point and *fractionDigits* digits after the significand's decimal point. If *fractionDigits* is **undefined**, include as many significand digits as necessary to uniquely specify the number (just like in ToString except that in this case the number is always output in exponential notation). Specifically, perform the following steps:

1. Let x be this number value.
2. Let f be ToInteger(*fractionDigits*).
3. If x is **NaN**, return the string **"NaN"**.
4. Let s be the empty string.
5. If $x \geq 0$, go to step 8.
6. Let s be **"-"**.
7. Let $x = -x$.
8. If $x = +\infty$, let m = **"Infinity"** and go to step 30.
9. If *fractionDigits* is **undefined**, go to step 14.
10. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
11. If $x = 0$, go to step 16.
12. Let e and n be integers such that $10^f \leq n < 10^{f+1}$ and for which the exact mathematical value of $n \times 10^{e-f} - x$ is as close to zero as possible. If there are two such sets of e and n, pick the e and n for which $n \times 10^{e-f}$ is larger.
13. Go to step 20.
14. If $x \neq 0$, go to step 19.
15. Let $f = 0$.
16. Let m be the string consisting of f+1 occurrences of the character '0'.
17. Let $e = 0$.
18. Go to step 21.
19. Let e, n, and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is x, and f is as small as possible. Note that the decimal representation of n has f+1 digits, n is not divisible by 10, and the least significant digit of n is not necessarily uniquely determined by these criteria.
20. Let m be the string consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
21. If $f = 0$, go to step 24.
22. Let a be the first character of m, and let b be the remaining f characters of m.
23. Let m be the concatenation of the three strings a, **"."**, and b.
24. If $e = 0$, let c = **"+"** and d = **"0"** and go to step 29.
25. If $e > 0$, let c = **"+"** and go to step 28.
26. Let c = **"-"**.
27. Let $e = -e$.
28. Let d be the string consisting of the digits of the decimal representation of e (in order, with no leading zeroes).
29. Let m be the concatenation of the four strings m, **"e"**, c, and d.
30. Return the concatenation of the strings s and m.

The **length** property of the **toExponential** method is **1**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

If the **toExponential** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **toExponential** for values of *fractionDigits* less than 0 or greater than 20. In this case **toExponential** would not necessarily throw **RangeError** for such values.

NOTE
For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 19 be used as a guideline:

Let e, n, and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, *the number value for* $n \times 10^{e-f}$ *is x, and f is as small as possible. If there are multiple possibilities for n, choose the value of n for which* $n \times 10^{e-f}$ *is closest in value to x. If there are two such possible values of n, choose the one that is even.*

15.7.4.7 Number.prototype.toPrecision (precision)

Return a string containing the number represented either in exponential notation with one digit before the significand's decimal point and *precision*–1 digits after the significand's decimal point or in fixed notation with *precision* significant digits. If *precision* is **undefined**, call ToString (9.8.1) instead. Specifically, perform the following steps:

1. Let *x* be this number value.
2. If *precision* is **undefined**, return ToString(*x*).
3. Let *p* be ToInteger(*precision*).
4. If *x* is **NaN**, return the string **"NaN"**.
5. Let *s* be the empty string.
6. If $x \geq 0$, go to step 9.
7. Let *s* be **"-"**.
8. Let $x = -x$.
9. If $x = +\infty$, let *m* = **"Infinity"** and go to step 30.
10. If $p < 1$ or $p > 21$, throw a **RangeError** exception.
11. If $x \neq 0$, go to step 15.
12. Let *m* be the string consisting of *p* occurrences of the character ‘0’.
13. Let $e = 0$.
14. Go to step 18.
15. Let *e* and *n* be integers such that $10^{p-1} \leq n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-p+1}$ is larger.
16. Let *m* be the string consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
17. If $e < -6$ or $e \geq p$, go to step 22.
18. If $e = p-1$, go to step 30.
19. If $e \geq 0$, let *m* be the concatenation of the first $e+1$ characters of *m*, the character ‘**.**’, and the remaining $p-(e+1)$ characters of *m* and go to step 30.
20. Let *m* be the concatenation of the string **"0."**, $-(e+1)$ occurrences of the character ‘0’, and the string *m*.
21. Go to step 30.
22. Let *a* be the first character of *m*, and let *b* be the remaining $p-1$ characters of *m*.
23. Let *m* be the concatenation of the three strings *a*, **"."**, and *b*.
24. If $e = 0$, let *c* = **"+"** and *d* = **"0"** and go to step 29.
25. If $e > 0$, let *c* = **"+"** and go to step 28.
26. Let *c* = **"-"**.
27. Let $e = -e$.
28. Let *d* be the string consisting of the digits of the decimal representation of *e* (in order, with no leading zeroes).
29. Let *m* be the concatenation of the four strings *m*, **"e"**, *c*, and *d*.
30. Return the concatenation of the strings *s* and *m*.

The **length** property of the **toPrecision** method is **1**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

If the **toPrecision** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **toPrecision** for values of *precision* less than 1 or greater than 21. In this case **toPrecision** would not necessarily throw **RangeError** for such values.

15.7.4.8 Number.prototype.toJSON (key)

When the toJSON method is called with argument key, the following steps are taken:

1. Let *O* be this object.
2. Call the [[Get]] method of *O* with argument "**valueOf**".
3. If IsCallable(Result(2)) is **false**, go to step 6
4. Call the [[Call]] method of Result(2) with *O* as the **this** value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a **TypeError** exception.

NOTE
*The **toJSON** function is intentionally generic; it does not require that its **this** value be a Number object. Therefore, it can be transferred to other kinds of objects for use as a method.*
An object is free to use the argument 'key' that is passed in to filter its stringification.

15.8 The Math Object

Mark S. Miller 1/19/09 10:47 PM
Deleted: 15.7.5 Properties of Number Instances
Number instances have no special properties beyond those inherited from the Number prototype object.

The Math object is a single object that has some named properties, some of which are functions.

The value of the internal [[Parent]] property of the Math object is the Object prototype object (15.2.3.1). The value of the internal [[Class]] property of the Math object is **"Math"**.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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.

The Math object does not have a [[Call]] property; it is not possible to invoke the Math object as a function.

NOTE
In this specification, the phrase "the number value for x" has a technical meaning defined in 8.5.

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.

Mark S. Miller 1/19/09 10:47 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.8.1.2 LN10

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

Mark S. Miller 1/19/09 10:47 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.8.1.3 LN2

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

Mark S. Miller 1/19/09 10:47 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

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
*The value of **Math.LOG2E** is approximately the reciprocal of the value of **Math.LN2**.*

Mark S. Miller 1/19/09 10:47 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

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
*The value of **Math.LOG10E** is approximately the reciprocal of the value of **Math.LN10**.*

Mark S. Miller 1/19/09 10:48 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.8.1.6 PI

The number value for π, the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.8.1.7 SQRT1_2

The number value for the square root of 1/2, which is approximately 0.7071067811865476.

Mark S. Miller 1/19/09 10:48 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE
The value of ***`Math.SQRT1_2`*** *is approximately the reciprocal of the value of* ***`Math.SQRT2`***.

Mark S. Miller 1/19/09 10:48 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

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

Mark S. Miller 1/19/09 10:48 PM
Deleted: This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

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).

In the function descriptions below, the symbols NaN, −0, +0, −∞ and +∞ refer to the number values described in 8.5.

NOTE
The behaviour of the functions ***`acos`***, ***`asin`***, ***`atan`***, ***`atan2`***, ***`cos`***, ***`exp`***, ***`log`***, ***`pow`***, ***`sin`***, *and* ***`sqrt`*** *is not precisely specified here except to require specific results for certain argument values that represent boundary cases of interest. For other argument values, these functions 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 implementer should be able to use the same mathematical library for SES on a given hardware platform that is available to C programmers on that platform.*

Mark S. Miller 1/19/09 5:04 PM
Deleted: *ECMAScript*

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in ***`fdlibm`***, *the freely distributable mathematical library from Sun Microsystems (`fdlibm-comment@sunpro.eng.sun.com`). This specification also requires specific results for certain argument values that represent boundary cases of interest*

15.8.2.1 abs (x)

Returns the absolute value of *x*; the result has the same magnitude as *x* but has positive sign.

If *x* is NaN, the result is NaN.
If *x* is −0, the result is +0.
If *x* is −∞, the result is +∞.

15.8.2.2 acos (x)

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

If *x* is NaN, the result is NaN.
If *x* is greater than 1, the result is NaN.
If *x* is less than −1, the result is NaN.
If *x* is exactly 1, the result is +0.

15.8.2.3 asin (x)

Returns an implementation-dependent approximation to the arc sine of *x*. The result is expressed in radians and ranges from −π/2 to +π/2.

If *x* is NaN, the result is NaN.
If *x* is greater than 1, the result is NaN.
If *x* is less than −1, the result is NaN.
If *x* is +0, the result is +0.
If *x* is −0, the result is −0.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.8.2.4 atan (x)

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

If x is NaN, the result is NaN.
If x is +0, the result is +0.
If x is −0, the result is −0.
If x is $+\infty$, the result is an implementation-dependent approximation to $+\pi/2$.
If x is $-\infty$, the result is an implementation-dependent approximation to $-\pi/2$.

15.8.2.5 atan2 (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 y and x 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 x or y 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 if 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 x and is equal to a mathematical integer. If x is already an integer, the result is x.

If x is NaN, the result is NaN.
If x is +0, the result is +0.
If x is −0, the result is −0.
If x is $+\infty$, the result is $+\infty$.
If x is $-\infty$, the result is $-\infty$.
If x is less than `0` but greater than `-1`, the result is −0.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 *x*. The argument is expressed in radians.

If *x* is NaN, the result is NaN.
If *x* is +0, the result is 1.
If *x* is −0, the result is 1.
If *x* is +∞, the result is NaN.
If *x* is −∞, the result is NaN.

15.8.2.8 exp (x)

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

If *x* is NaN, the result is NaN.
If *x* is +0, the result is 1.
If *x* is −0, the result is 1.
If *x* is +∞, the result is +∞.
If *x* is −∞, the result is +0.

15.8.2.9 floor (x)

Returns the greatest (closest to **+∞**) number value that is not greater than *x* and is equal to a mathematical integer. If *x* is already an integer, the result is *x*.

If *x* is NaN, the result is NaN.
If *x* is +0, the result is +0.
If *x* is −0, the result is −0.
If *x* is +∞, the result is +∞.
If *x* is −∞, the result is −∞.
If *x* is greater than `0` but less than `1`, the result is `+0`.

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

15.8.2.10 log (x)

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

If *x* is NaN, the result is NaN.
If *x* is less than 0, the result is NaN.
If *x* is +0 or −0, the result is −∞.
If *x* is 1, the result is +0.
If *x* is +∞, the result is +∞.

15.8.2.11 max ([value1 [, value2 [, …]]])

Given zero or more arguments, calls ToNumber on each of the arguments and returns the largest of the resulting values.

If no arguments are given, the result is −∞.
If any value is NaN, the result is NaN.
The comparison of values to determine the largest value is done as in 11.8.5 except that +0 is considered to be larger than −0.

The **`length`** property of the **`max`** method is **2**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.8.2.12 min ([value1 [, value2 [, …]]])

Given zero or more arguments, calls ToNumber on each of the arguments and returns the smallest of the resulting values.

- If no arguments are given, the result is +∞.
- If any value is NaN, the result is NaN.
- The comparison of values to determine the smallest value is done as in 11.8.5 except that +0 is considered to be larger than −0.

The **`length`** property of the **`min`** method is **2**.

15.8.2.13 pow (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 +∞.
- 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 pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

To preserve isolation, the randomness or pseudorandomness must be sufficiently unguessable by its clients that no one client can guess better than chance how many times other clients have called it.

Mark S. Miller 1/19/09 10:53 PM
Comment: Consider adopting this language, at least as advisory, into ES3.1.

15.8.2.15 round (x)

Returns the number value that is closest to *x* and is equal to a mathematical integer. If two integer number values are equally close to *x*, then the result is the number value that is closer to +∞. If *x* is already an integer, the result is *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

If *x* is +∞, the result is +∞.

If *x* is −∞, the result is −∞.

If *x* is greater than `0` but less than `0.5`, the result is +0.

If *x* is less than `0` but greater than or equal to `-0.5`, the result is −0.

NOTE 1
Math.round(3.5) returns 4, but Math.round(–3.5) returns –3.

NOTE 2
The value of **`Math.round(x)`** *is the same as the value of* **`Math.floor(x+0.5)`***, except when* **`x`** *is −0 or is less than* `0` *but greater than or equal to* `-0.5`*; for these cases* **`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 *x*. The argument is expressed in radians.

If *x* is NaN, the result is NaN.

If *x* is +0, the result is +0.

If *x* is −0, the result is −0.

If *x* is +∞ or −∞, the result is NaN.

15.8.2.17 sqrt (x)

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

If *x* is NaN, the result is NaN.

If *x* less than 0, the result is NaN.

If *x* is +0, the result is +0.

If *x* is −0, the result is −0.

If *x* is +∞, the result is +∞.

15.8.2.18 tan (x)

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

If *x* is NaN, the result is NaN.

If *x* is +0, the result is +0.

If *x* is −0, the result is −0.

If *x* is +∞ or −∞, the result is NaN.

15.9 Date Objects

15.9.1 Overview of Date Objects and Definitions of Internal Operators

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 SES in milliseconds since 01 January, 1970 UTC. Leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. SES number values can represent all integers from –9,007,199,254,740,991 to 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 SES Date objects is slightly smaller: exactly –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 8,640,000,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value **+0**.

15.9.1.2 Day Number and Time within Day

A given time value *t* belongs to day number

Day(*t*) = floor(*t* / msPerDay)

where the number of milliseconds per day is

msPerDay = 86400000

The remainder is called the time within the day:

TimeWithinDay(*t*) = *t* modulo msPerDay

15.9.1.3 Year Number

SES 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

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

DaysInYear(*y*) = 365 if (*y* modulo 4) ≠ 0
= 366 if (*y* modulo 4) = 0 and (y modulo 100) ≠ 0
= 365 if (*y* modulo 100) = 0 and (y modulo 400) ≠ 0
= 366 if (*y* modulo 400) = 0

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:

DayFromYear(y) = 365 × (y−1970) + floor((y−1969)/4) − floor((y−1901)/100) + floor((y−1601)/400)

The time value of the start of a year is:

TimeFromYear(*y*) = msPerDay × DayFromYear(*y*)

A time value determines a year by:

YearFromTime(*t*) = the largest integer *y* (closest to positive infinity) such that TimeFromYear(*y*) ≤ *t*

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

InLeapYear(*t*) = 0 if DaysInYear(YearFromTime(*t*)) = 365
= 1 if DaysInYear(YearFromTime(*t*)) = 366

15.9.1.4 Month Number

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

MonthFromTime(*t*)	= 0	if	0	≤ DayWithinYear(*t*) < 31
	= 1	if	31	≤ DayWithinYear (*t*) < 59+InLeapYear(*t*)
	= 2	if	59+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 90+InLeapYear(*t*)
	= 3	if	90+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 120+InLeapYear(*t*)
	= 4	if	120+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 151+InLeapYear(*t*)
	= 5	if	151+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 181+InLeapYear(*t*)
	= 6	if	181+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 212+InLeapYear(*t*)
	= 7	if	212+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 243+InLeapYear(*t*)
	= 8	if	243+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 273+InLeapYear(*t*)
	= 9	if	273+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 304+InLeapYear(*t*)
	= 10	if	304+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 334+InLeapYear(*t*)
	= 11	if	334+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 365+InLeapYear(*t*)

where

DayWithinYear(*t*) = Day(*t*)−DayFromYear(YearFromTime(*t*))

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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 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 DateFromTime(*t*) from a time value *t* to a month number is defined by:

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

15.9.1.6 Week Day

The weekday for a particular time value *t* is defined as

WeekDay(*t*) = (Day(*t*) + 4) 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 WeekDay(0) = 4, corresponding to Thursday, 01 January, 1970.

15.9.1.7 Local Time Zone Adjustment

An implementation of SES is expected to determine the local time zone adjustment. 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.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

pratapL 1/19/09 8:57 PM
Comment: This assertion is incorrect. It assumes time zone boundaries are fixed for eternity. It is not, and is subject to politics (as seen by the recent DST change that has happened in US.

The wording in this section needs to change.

15.9.1.8 Daylight Saving Time Adjustment

An implementation of SES is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment DaylightSavingTA(*t*), measured in milliseconds, must depend only on four things:

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

pratapL 1/19/09 8:57 PM
Comment: Same as the earlier comment. This assertion about DST is incorrect. The wording needs to be changed.

(1) the time since the beginning of the year

t – TimeFromYear(YearFromTime(*t*))

(2) whether t is in a leap year

InLeapYear(*t*)

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

WeekDay(TimeFromYear(YearFromTime(*t*))

and (4) the geographic location.

The implementation of SES 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 locale observed daylight saving time year round.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

If the host environment provides functionality for determining daylight saving time, the implementation of SES 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 host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

LocalTime(t) = t + LocalTZA + DaylightSavingTA(t)

Conversion from local time to UTC is defined by

UTC(t) = t – LocalTZA – DaylightSavingTA(t – LocalTZA)

Note that UTC(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:

HourFromTime(t) = floor(t / msPerHour) modulo HoursPerDay

MinFromTime(t) = floor(t / msPerMinute) modulo MinutesPerHour

SecFromTime(t) = floor(t / msPerSecond) modulo SecondsPerMinute

msFromTime(t) = t modulo msPerSecond

where

HoursPerDay = 24

MinutesPerHour = 60

SecondsPerMinute = 60

msPerSecond = 1000

msPerMinute = msPerSecond × SecondsPerMinute = 60000

msPerHour = msPerMinute × 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 SES number values. This operator functions as follows:

1. If *hour* is not finite or *min* is not finite or *sec* is not finite or *ms* is not finite, return **NaN**.
2. Call ToInteger(*hour*).
3. Call ToInteger(*min*).
4. Call ToInteger(*sec*).
5. Call ToInteger(*ms*).
6. Compute Result(2) ***** msPerHour **+** Result(3) ***** msPerMinute **+** Result(4) ***** msPerSecond **+** Result(5), performing the arithmetic according to IEEE 754 rules (that is, as if using the SES operators ***** and **+**).
7. 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 SES number values. This operator functions as follows:

1. If *year* is not finite or *month* is not finite or *date* is not finite, return **NaN**.
2. Call ToInteger(*year*).
3. Call ToInteger(*month*).
4. Call ToInteger(*date*).
5. Compute Result(2) + floor(Result(3)/12).
6. Compute Result(3) modulo 12.
7. Find a value t such that YearFromTime(t) **==** Result(5) and MonthFromTime(t) **==** Result(6) and DateFromTime(t) **==** 1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Compute Day(Result(7)) + Result(4) − 1.
9. Return Result(8).

pratapL 1/19/09 8:57 PM
Comment: From AWB:
Need to defiine TimeValue and make it clear that a TimeValue is what this function returns. Needed for Date.now().

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.9.1.13 MakeDate (day, time)

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

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

1. If *day* is not finite or *time* is not finite, return **NaN**.
2. Compute *day* × msPerDay + *time*.
3. Return Result(2).

15.9.1.14 TimeClip (time)

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

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

1. If *time* is not finite, return **NaN**.
2. If abs(Result(1)) > **8.64 x 10^{15}**, return **NaN**.
3. Return an implementation-dependent choice of either ToInteger(Result(2)) or ToInteger(Result(2)) + (**+0**). (Adding a positive zero converts **–0** to **+0**.)

NOTE
*The point of step 3 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish **–0** and **+0**.*

15.9.1.15 Date Time string format

The Simplified ISO 8601 format is as follows: **`YYYY-MM-DDTHH:mm:ss.sssTZ`**

Where the fields are as follows:

`YYYY` is the year in the Gregorian calendar

`MM` is the month of the year between 01 (January) and 12 (December)

`DD` is the day of the month between 01 and 31.

The “T” appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.

`HH` is the number of complete hours that have passed since midnight

`mm` is the number of complete minutes since the start of the hour

`ss` is the number of complete seconds since the start of the minute

The ‘**`.`**’ (dot)

`sss` is the number of complete milliseconds since the start of the second.

Both the ‘**`.`**’ And the milliseconds field are optional

`TZ` is the timezone specified as **`Z`** (for UTC) or either **`+`** or **`-`** followed by a time expression **`HH:MM`**

Extended years

SES requires the ability to specify 6 digit years (extended years); approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In thesimplified SES format such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or – sign with the convention that year 0 is positive.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

NOTE

This format includes date-only forms:

```
YYYY
YYYY-MM
YYYY-MM-DD
```

Time-only forms with an optional time zone appended:

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

THH:mm
THH:mm:ss
THH:mm:ss.sss

It also includes “date-times” which could be any combination of the above.

All numbers must be base 10.

Illegal values (out-of-bounds as well as syntax errors) in a format string means that the format string is not a valid instance of this format.

As every day both starts and ends with midnight, the two notations `00:00` and `24:00` are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: `1995-02-04T24:00` and `1995-02-05T00:00`

There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. For this reason, ISO 8601 and this format specifies *numeric* representations of date and time.

15.9.2 The Date Constructor Called as a Function

When **`Date`** is called as a function rather than as a constructor, it throws a **TypeError**.

Mark S. Miller 1/19/09 10:55 PM
Deleted: returns a string representing the current time (UTC).

NOTE
*The function call **`Date(…)`** is not equivalent to the object creation expression **`new Date(…)`** with the same arguments.*

15.9.3 The Date Constructor

Mark S. Miller 1/19/09 10:56 PM
Deleted: 15.9.2.1 Date ([year [, month [, date [, hours [, minutes [, seconds [, ms]]]]]]])
All of the arguments are optional; any arguments supplied are accepted but are completely ignored. A string is created and returned as if by the expression **`(new Date()).toString()`** where **`Date`** is the standard built-in constructor with that name and **`toString`** is the standard built-in method **`Date.prototype.toString`**.

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

15.9.3.1 new Date (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When **Date** is called with two to seven arguments, it computes the date from *year*, *month*, and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*.

The [[Parent]] property of the newly constructed object is set to the original Date prototype object.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:56 PM
Deleted: , the one that is the initial value of **`Date.prototype`** (15.9.4.1)

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

The [[Extensible]] property of the newly constructed object is set to **true**.

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

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. If *date* is supplied use ToNumber(*date*); else use **1**.
4. If *hours* is supplied use ToNumber(*hours*); else use **0**.
5. If *minutes* is supplied use ToNumber(*minutes*); else use **0**.
6. If *seconds* is supplied use ToNumber(*seconds*); else use **0**.
7. If *ms* is supplied use ToNumber(*ms*); else use **0**.
8. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Compute MakeDate(Result(9), Result(10)).
12. Set the [[PrimitiveValue]] property of the newly constructed object to TimeClip(UTC(Result(11))).

15.9.3.2 new Date (value)

The [[Parent]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **`Date.prototype`** (15.9.4.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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

The [[Extensible]] property of the newly constructed object is set to **true**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

1. Call ToPrimitive(*value*).
2. If Type(Result(1)) is String, then go to step 5.
3. Let *V* be ToNumber(Result(1)).
4. Set the [[PrimitiveValue]] property of the newly constructed object to TimeClip(*V*) and return.
5. Parse Result(1) as a date, in exactly the same manner as for the **`parse`** method (15.9.4.2); let *V* be the time value for this date.
6. Go to step 4.

15.9.3.3 new Date ( )

Throws a **TypeError**.

15.9.4 Properties of the Date Constructor

The value of the internal [[Parent]] property of the Date constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **`7`**), the Date constructor has the following properties:

15.9.4.1 N/A

15.9.4.2 Date.parse (string)

The **`parse`** function applies the ToString operator to its argument and interprets the resulting string as a date; it returns a number, the UTC time value corresponding to the date. The string may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the string. The function first attempts to parse the format of the string according to the rules called out in Date Time String Format (15.9.1.15). If the string does not conform to that format the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Unrecognizable strings or dates containing illegal element values in the format string shall cause **`Date.parse`** to return NaN.

If *x* is any Date object whose milliseconds amount is zero within a particular implementation of SES, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

```
x.valueOf()
```

```
Date.parse(x.toString())
```

```
Date.parse(x.toUTCString())
```

However, the expression

```
Date.parse(x.toLocaleString())
```

is not required to produce the same number value as the preceding three expressions and, in general, the value produced by **`Date.parse`** is implementation-dependent when given any string value that could not be produced in that implementation by the **`toString`** or **`toUTCString`** method.

15.9.4.3 Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When the **`UTC`** function is called with fewer than two arguments, the behaviour is implementation-dependent. When the **`UTC`** function is called with two to seven arguments, it computes the date from *year*, *month* and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*. The following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. If *date* is supplied use ToNumber(*date*); else use **1**.
4. If *hours* is supplied use ToNumber(*hours*); else use **0**.
5. If *minutes* is supplied use ToNumber(*minutes*); else use **0**.
6. If *seconds* is supplied use ToNumber(*seconds*); else use **0**.
7. If *ms* is supplied use ToNumber(*ms*); else use **0**.
8. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).

Mark S. Miller 1/19/09 10:57 PM
Deleted: The [[Prototype]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **`Date.prototype`** (15.9.4.1).
The [[Class]] property of the newly constructed object is set to **`"Date"`**.
The [[Extensible]] property of the newly constructed object is set to **true**.
The [[PrimitiveValue]] property of the newly constructed object is set to the current

Mark S. Miller 1/19/09 10:57 PM
Deleted: time (UTC)

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 10:57 PM
Deleted: **Date.prototype**

Mark S. Miller 1/19/09 10:57 PM
Deleted: The initial value of **`Date.prototype`** is the built-in Date prototype object (15.9.5).
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Return TimeClip(MakeDate(Result(9), Result(10))).

The **`length`** property of the **`UTC`** function is **7**.

NOTE
The UTC function differs from the Date constructor in two ways: it returns a time value as a number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

15.9.4.4 Date.now ( )

Throws a **TypeError**.

Mark S. Miller 1/19/09 10:58 PM
Deleted: The `now` method produces the time value at the time of the call

15.9.5 Properties of the Date Prototype Object

The Date prototype object is itself a Date object (its [[Class]] is **`"Date"`**) whose [[PrimitiveValue]] is **NaN**.

The value of the internal [[Parent]] property of the Date prototype object is the Object prototype object (15.2.3.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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. None of these functions are generic; a **TypeError** exception is thrown if the **this** value is not an object for which the value of the internal [[Class]] property is **`"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 [[PrimitiveValue]] property of this Date object.

15.9.5.1 N/A

Mark S. Miller 1/19/09 10:59 PM
Deleted: **Date.prototype.constructor**

Mark S. Miller 1/19/09 10:59 PM
Deleted: The initial value of **`Date.prototype.constructor`** is the built-in **`Date`** constructor.

15.9.5.2 Date.prototype.toString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form.

NOTE
For any Date value `d` whose milliseconds amount is zero, the result of `Date.parse(d.toString())` is equal to `d.valueOf()`. See section 15.9.4.2.

15.9.5.3 Date.prototype.toDateString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form.

15.9.5.4 Date.prototype.toTimeString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form.

15.9.5.5 Date.prototype.toLocaleString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.6 Date.prototype.toLocaleDateString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The first parameter to this function is likely to be used in a future version of this standard; it is required that implementations of this version do not use this parameter position for anything else.

Mark S. Miller 1/19/09 11:00 PM
Deleted: *recommended*

15.9.5.7 Date.prototype.toLocaleTimeString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is required that implementations of this version do not use this parameter position for anything else.

Mark S. Miller 1/19/09 11:00 PM
Deleted: *recommended that implementations*

15.9.5.8 Date.prototype.valueOf ( )

The **`valueOf`** function returns a number, which is this time value.

15.9.5.9 Date.prototype.getTime ( )

1. If the **this** value is not an object whose [[Class]] property is **`"Date"`**, throw a **TypeError** exception.
2. Return this time value.

15.9.5.10 Date.prototype.getFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*)).

15.9.5.11 Date.prototype.getUTCFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(*t*).

15.9.5.12 Date.prototype.getMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(LocalTime(*t*)).

15.9.5.13 Date.prototype.getUTCMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(*t*).

15.9.5.14 Date.prototype.getDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(LocalTime(*t*)).

15.9.5.15 Date.prototype.getUTCDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(*t*).

15.9.5.16 Date.prototype.getDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(LocalTime(*t*)).

15.9.5.17 Date.prototype.getUTCDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

3. Return WeekDay(*t*).

15.9.5.18 Date.prototype.getHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(LocalTime(*t*)).

15.9.5.19 Date.prototype.getUTCHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(*t*).

15.9.5.20 Date.prototype.getMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(LocalTime(*t*)).

15.9.5.21 Date.prototype.getUTCMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(*t*).

15.9.5.22 Date.prototype.getSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(LocalTime(*t*)).

15.9.5.23 Date.prototype.getUTCSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(*t*).

15.9.5.24 Date.prototype.getMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(LocalTime(*t*)).

15.9.5.25 Date.prototype.getUTCMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(*t*).

15.9.5.26 Date.prototype.getTimezoneOffset ( )

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

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return (*t* − LocalTime(*t*)) / msPerMinute.

15.9.5.27 Date.prototype.setTime (time)

1. If the **this** value is not a Date object, throw a **TypeError** exception.
2. Call ToNumber(*time*).
3. Call TimeClip(Result(1)).
4. Set the [[PrimitiveValue]] property of the **this** value to Result(2).
5. Return the value of the [[PrimitiveValue]] property of the **this** value.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.9.5.28 Date.prototype.setMilliseconds (ms)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Day(*t*), Result(3))).
5. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[PrimitiveValue]] property of the **this** value.

15.9.5.29 Date.prototype.setUTCMilliseconds (ms)

1. Let *t* be this time value.
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute MakeDate(Day(*t*), Result(3)).
5. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[PrimitiveValue]] property of the **this** value.

15.9.5.30 Date.prototype.setSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value getMilliseconds( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Day(*t*), Result(4))).
6. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setSeconds`** method is **2**.

15.9.5.31 Date.prototype.setUTCSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Day(*t*), Result(4)).
6. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setUTCSeconds`** method is **2**.

15.9.5.33 Date.prototype.setMinutes (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( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Day(*t*), Result(5))).
7. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setMinutes`** method is **3**.

15.9.5.34 Date.prototype.setUTCMinutes (min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value getUTCSeconds( ).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute MakeDate(Day(*t*), Result(5)).
7. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the **this** value.

The `length` property of the `setUTCMinutes` method is **3**.

15.9.5.35 Date.prototype.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( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(*t*), Result(6))).
8. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(7)).
9. Return the value of the [[PrimitiveValue]] property of the **this** value.

The `length` property of the `setHours` method is **4**.

15.9.5.36 Date.prototype.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( ).

1. Let *t* be this time value.
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute MakeDate(Day(*t*), Result(6)).
8. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(7)).
9. Return the value of the [[PrimitiveValue]] property of the **this** value.

The `length` property of the `setUTCHours` method is **4**.

15.9.5.36 Date.prototype.setDate (date)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Result(3), TimeWithinDay(*t*))).
5. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[PrimitiveValue]] property of the **this** value.

15.9.5.37 Date.prototype.setUTCDate (date)

1. Let *t* be this time value.
2. Call ToNumber(*date*).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute MakeDate(Result(3), TimeWithinDay(*t*)).
5. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[PrimitiveValue]] property of the **this** value.

15.9.5.38 Date.prototype.setMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getDate( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Result(4), TimeWithinDay(*t*))).
6. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setMonth`** method is **2**.

15.9.5.39 Date.prototype.setUTCMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate( ).

1. Let *t* be this time value.
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Result(4), TimeWithinDay(*t*)).
6. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setUTCMonth`** method is **2**.

15.9.5.40 Date.prototype.setFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value getMonth( ).

If *date* is not specified, this behaves as if *date* were specified with the value getDate( ).

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setFullYear`** method is **3**.

15.9.5.41 Date.prototype.setUTCFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value getUTCMonth( ).

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate( ).

1. Let *t* be this time value; but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute MakeDate(Result(5), TimeWithinDay(*t*)).
7. Set the [[PrimitiveValue]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the **this** value.

The **`length`** property of the **`setUTCFullYear`** method is **3**.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.9.5.42 Date.prototype.toUTCString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in a convenient, human-readable form in UTC.

NOTE
The intent is to produce a string representation of a date that is more readeable than the format specified in Section 15.9.1.15. It is not essential that the chosen format be unambiguous or easily machine parsable. If an implementation does not have a preferred human-readable format it is recommended to use the format called out in Section 15.9.1.15 but with a space rather than a "`T`" used to separate the date and time elements.

15.9.5.43 Date.prototype.toISOString ( )

This function returns a string value. The format of the string is as called out in Date Time string format (15.9.1.15). All fields are present in the string. The time zone is always UTC, denoted by the suffix Z.

pratapL 1/19/09 8:57 PM
Comment: From AWB:
Need to make it clear that the string is derived from this date object's internal date value.

15.9.5.44 Date.prototype.toJSON (key)

This function returns the same string as Date.prototype.toISOString().

When the toJSON method is called with argument key, the following steps are taken:

1. Let *O* be this object.
2. Call the [[Get]] method of *O* with argument "**`toISOString`**".
3. If IsCallable(Result(2)) is **false**, go to step 6
4. Call the [[Call]] method of Result(2) with *O* as the **this** value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a **TypeError** exception.

NOTE
*The **`toJSON`** function is intentionally generic; it does not require that its **this** value be a Date object. Therefore, it can be transferred to other kinds of objects for use as a method.*
An object is free to use the argument 'key' that is passed in to filter its stringification.

15.9.6 Properties of Date Instances

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

15.10 RegExp (Regular Expression) Objects

A RegExp object contains a regular expression and the associated flags.

NOTE
The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

15.10.1 Patterns

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

Syntax

Pattern **::**
  Disjunction

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Term **::**
Assertion
Atom
Atom Quantifier

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

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

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

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

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

AtomEscape **::**
DecimalEscape
CharacterEscape
CharacterClassEscape

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

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

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

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

DecimalEscape **::**
DecimalIntegerLiteral [lookahead ∉ *DecimalDigit*]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

ClassRanges **::**
  [empty]
  NonemptyClassRanges

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

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

ClassAtom **::**
  `-`
  ClassAtomNoDash

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

ClassEscape **::**
  DecimalEscape
  `b`
  CharacterEscape
  CharacterClassEscape

15.10.2 Pattern Semantics

A regular expression pattern is converted into an internal procedure using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same. The internal procedure is used as the value of a RegExp object's [[Match]] internal property.

15.10.2.1 Notation

The descriptions below use the following variables:

Input is the string being matched by the regular expression pattern. The notation *input*[*n*] means the *n*th character of *input*, where *n* can range between 0 (inclusive) and *InputLength* (exclusive).

InputLength is the number of characters in the *Input* string.

NcapturingParens is the total number of left capturing parentheses (i.e. the total number of times the *Atom* :: **`(`** *Disjunction* **`)`** production is expanded) in the pattern. A left capturing parenthesis is any **`(`** pattern character that is matched by the **`(`** terminal of the *Atom* :: **`(`** *Disjunction* **`)`** production.

IgnoreCase is the setting of the RegExp object's **`ignoreCase`** property.

Multiline is the setting of the RegExp object''s **`multiline`** property.

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

A *CharSet* is a mathematical set of characters.

A *State* is an ordered pair (*endIndex*, *captures*) where *endIndex* is an integer and *captures* is an internal array of *NcapturingParens* values. States are used to represent partial match states in the regular expression matching algorithms. The *endIndex* is one plus the index of the last input character matched so far by the pattern, while *captures* holds the results of capturing parentheses.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The *n*th element of *captures* is either a string that represents the value obtained by the *n*th set of capturing parentheses or **undefined** if the *n*th set of capturing parentheses hasn''t been reached yet. Due to backtracking, many states may be in use at any time during the matching process.

A *MatchResult* is either a State or the special token **failure** that indicates that the match failed.

A *Continuation* procedure is an internal closure (i.e. an internal procedure with some arguments already bound to values) that takes one State argument and returns a MatchResult result. If an internal closure references variables bound in the function that creates the closure, the closure uses the values that these variables had at the time the closure was created. The continuation attempts to match the remaining portion (specified by the closure's already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. If the match succeeds, the continuation returns the final State that it reached; if the match fails, the continuation returns **failure**.

A *Matcher* procedure is an internal closure that takes two arguments -- a State and a Continuation -- and returns a MatchResult result. The matcher attempts to match a middle subpattern (specified by the closure's already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. The Continuation argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new State, the matcher then calls Continuation on that state to test if the rest of the pattern can match as well. If it can, the matcher returns the state returned by the continuation; if not, the matcher may try different choices at its choice points, repeatedly calling Continuation until it either succeeds or all possibilities have been exhausted.

An *AssertionTester* procedure is an internal closure that takes a State argument and returns a boolean result. The assertion tester tests a specific condition (specified by the closure's already-bound arguments) against the current place in the input string and returns **true** if the condition matched or **false** if not.

An *EscapeValue* is either a character or an integer. An EscapeValue is used to denote the interpretation of a *DecimalEscape* escape sequence: a character *ch* means that the escape sequence is interpreted as the character *ch*, while an integer *n* means that the escape sequence is interpreted as a backreference to the *n*th set of capturing parentheses.

15.10.2.2 Pattern

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

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

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

15.10.2.3 Disjunction

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

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

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

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

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

returns the array

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

and not

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

15.10.2.4 Alternative

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

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

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

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

15.10.2.5 Term

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

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

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

1. If *max* is zero, then call *c*(*x*) and return its result.
2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:

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

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

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

Compare

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

which returns **"abcde"** with

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

which returns **"abc"**.

Consider also

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

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

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

and not any of:

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

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

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

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

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

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

which returns the array

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

and not

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

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

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

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

or the slightly more complicated:

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

which returns the array

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

15.10.2.6 Assertion

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

1. Let *e* be *x*'s *endIndex*.
2. If *e* is zero, return **true**.
3. If *Multiline* is **false**, return **false**.
4. If the character *Input*[*e*–1] is one of *LineTerminator*, return **true**.
5. Return **false**.

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

1. Let *e* be *x*'s *endIndex*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

2. If *e* is equal to *InputLength*, return **true**.
3. If *multiline* is **false**, return **false**.
4. If the character *Input*[*e*] is one of *LineTerminator*, return **true**.
5. Return **false**.

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

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

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

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

The abstract operation *IsWordChar* takes an integer parameter *e* and performs the following:

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

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

4. Return **false**.

15.10.2.7 Quantifier

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

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

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

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

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

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

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

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

1. Let *i* be the MV of *DecimalDigits* (see 7.8.3).
2. Return the two results *i* and *i*.

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

15.10.2.8 Atom

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

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

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

1. Let *A* be the set of all characters except *LineTerminator*.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

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

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

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

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

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this production's *Atom* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *Atom*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
   1. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:

      Let *cap* be a fresh copy of *y*'s *captures* internal array.
      Let *xe* be *x*'s *endIndex*.
      Let *ye* be *y*'s *endIndex*.
      Let *s* be a fresh string whose characters are the characters of *Input* at positions *xe* (inclusive) through *ye* (exclusive).
      Set *cap*[*parenIndex*+1] to *s*.
      Let *z* be the State (*ye*, *cap*).
      Call *c*(*z*) and return its result.
   2. Call *m*(*x*, *d*) and return its result.

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

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

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
   1. Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
   2. Call *m*(*x*, *d*) and let *r* be its result.
   3. If *r* is **failure**, return **failure**.
   4. Let *y* be *r*'s State.
   5. Let *cap* be *y*'s *captures* internal array.
   6. Let *xe* be *x*'s *endIndex*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

7. Let *z* be the State (*xe*, *cap*).
8. Call *c*(*z*) and return its result.

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

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

The abstract operation *CharacterSetMatcher* takes two arguments, a CharSet *A* and a boolean flag *invert*, and performs the following:

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

The abstract operation *Canonicalize* takes a character parameter *ch* and performs the following:

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

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

The form **(?=** *Disjunction* **)** specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside *Disjunction* must match at the current position, but the current position is not advanced before matching the sequel. If *Disjunction* can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a **(?=** form (this unusual behaviour is inherited from Perl). This only matters when the *Disjunction* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

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

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

To illustrate the lack of backtracking into the lookahead, consider:

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

This expression returns

`["aba", "a"]`

and not:

`["aaaba", "a"]`

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

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

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

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

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

15.10.2.9 AtomEscape

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

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

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

1. Evaluate *CharacterEscape* to obtain a character *ch*.
2. Let *A* be a one-element CharSet containing the character *ch*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

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

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

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

15.10.2.10 CharacterEscape

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

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

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

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

The production *CharacterEscape* **::** *HexEscapeSequence* evaluates by evaluating the CV of the *HexEscapeSequence* (see 7.8.4) and returning its character result.

The production *CharacterEscape* **::** *UnicodeEscapeSequence* evaluates by evaluating the CV of the *UnicodeEscapeSequence* (see 7.8.4) and returning its character result.

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

15.10.2.11 DecimalEscape

The production *DecimalEscape* **::** *DecimalIntegerLiteral* [lookahead ∉ *DecimalDigit*] evaluates as follows.

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

The definition of "the MV of *DecimalIntegerLiteral*" is in 7.8.3.

Informative comments: If `\` is followed by a decimal number *n* whose first digit is not `0`, then the escape sequence is considered to be a backreference. It is an error if *n* is greater than the total number of left capturing parentheses in the entire regular expression. `\0` represents the NUL character and cannot be followed by a decimal digit.

15.10.2.12 CharacterClassEscape

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The production *CharacterClassEscape* **:: s** evaluates by returning the set of characters containing the characters that are on the right-hand side of the *WhiteSpace* (7.2) or *LineTerminator* (7.3) productions.

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

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

pratapL 1/19/09 8:57 PM
Comment: From DEC:
This class is pretty close to useless in its current form. We could make it more useful by having it match ECMAScript identifier characters. It could at least them be used to process ECMAScript programs.

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

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

0 1 2 3 4 5 6 7 8 9 _

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

15.10.2.13 CharacterClass

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

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

15.10.2.14 ClassRanges

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

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

15.10.2.15 NonemptyClassRanges

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

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

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

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

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

The abstract operation *CharacterRange* takes two CharSet parameters *A* and *B* and performs the following:

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.10.2.16 NonemptyClassRangesNoDash

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

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

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

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

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

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

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

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

15.10.2.17 ClassAtom

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

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

15.10.2.18 ClassAtomNoDash

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

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

15.10.2.19 ClassEscape

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

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

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

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

Informative comments: A *ClassAtom* can use any of the escape sequences that are allowed in the rest of the regular expression except for **\b**, **\B**, and backreferences. Inside a *CharacterClass*, **\b** means the backspace character, while **\B** and backreferences raise errors. Using a backreference inside a ClassAtom causes an error.

15.10.3 The RegExp Constructor Called as a Function

15.10.3.1 RegExp(pattern, flags)

If *pattern* is an object *R* whose [[Class]] property is **"RegExp"** and *flags* is **undefined**, then return *R* unchanged. Otherwise call the standard built-in RegExp constructor (15.10.4.1), passing it the *pattern* and *flags* arguments and return the object constructed by that constructor.

15.10.4 The RegExp Constructor

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

15.10.4.1 new RegExp(pattern, flags)

If *pattern* is an object *R* whose [[Class]] property is **"RegExp"** and *flags* is **undefined**, then let *P* be the *pattern* used to construct *R* and let *F* be the flags used to construct *R*. If *pattern* is an object *R* whose [[Class]] property is **"RegExp"** and *flags* is not **undefined**, then throw a **TypeError** exception. Otherwise, let *P* be the empty string if *pattern* is **undefined** and ToString(*pattern*) otherwise, and let *F* be the empty string if *flags* is **undefined** and ToString(*flags*) otherwise.

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

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

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

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

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

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

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

The [[Parent]] property of the newly constructed object is set to the original RegExp prototype object.

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

15.10.5 Properties of the RegExp Constructor

The value of the internal [[Parent]] property of the RegExp constructor is the Function prototype object (15.3.4).

The **length** property of the **RegExp** constructor is 2.

15.10.6 Properties of the RegExp Prototype Object

The value of the internal [[Parent]] property of the RegExp prototype object is the Object prototype. The RegExp prototype object is itself a regular expression object; its [[Class]] is "RegExp"..

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:09 PM
Deleted: , the one that is the initial value of **RegExp.prototype**

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:10 PM
Deleted: Besides the internal properties and the

Mark S. Miller 1/19/09 11:10 PM
Deleted: (whose value is **2**), the RegExp constructor has the following properties

Mark S. Miller 1/19/09 11:11 PM
Deleted: :

Mark S. Miller 1/19/09 11:09 PM
Deleted: **15.10.5.1 RegExp.prototype**
The initial value of **RegExp.prototype** is the RegExp prototype object (15.10.6).
This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

pratapL 1/19/09 8:57 PM
Comment: From AWB:
Do any other properties, eg source, need to be specified?

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

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

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

15.10.6.1 N/A

Mark S. Miller 1/19/09 11:11 PM
Deleted: RegExp.prototype.constructor

15.10.6.2 RegExp.prototype.exec(string)

Mark S. Miller 1/19/09 11:11 PM
Deleted: The initial value of **RegExp.prototype.constructor** is the built-in **RegExp** constructor.

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

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

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

    The **index** property is set to the position of the matched substring within the complete string *S*.

    The **input** property is set to *S*.

    The **length** property is set to *n* + 1.

    The **0** property is set to the matched substring (i.e. the portion of *S* between offset *i* inclusive and offset *e* exclusive).

    For each integer *i* such that *I* > 0 and *I* ≤ *n*, set the property named ToString(*i*) to the i^{th} element of *r*'s *captures* array.

15.10.6.3 RegExp.prototype.test(string)

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

15.10.6.4 RegExp.prototype.toString()

Let *src* be a string in the form of a *Pattern* representing the current regular expression. *src* may or may not be identical to the **source** property or to the source code supplied to the RegExp constructor; however, if *src* were supplied to the RegExp constructor along with the current regular expression's flags, the resulting regular expression must behave identically to the current regular expression.

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

NOTE
An implementation may choose to take advantage of src being allowed to be different from the source passed to the RegExp constructor to escape special characters in src. For example, in the regular expression obtained from ***new RegExp("/")****, src could be, among other possibilities,* ***"/"*** *or* ***"\/"****. The latter would permit the entire result (****"/\//"****) of the* ***toString*** *call to have the form RegularExpressionLiteral.*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.10.7 Properties of RegExp Instances

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

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

15.10.7.1 source

The value of the **source** property is string in the form of a *Pattern* representing the current regular expression.

Mark S. Miller 1/19/09 11:12 PM
Deleted: This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.10.7.2 global

The value of the **global** property is a Boolean value indicating whether the flags contained the character "**g**".

Mark S. Miller 1/19/09 11:12 PM
Deleted: This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.10.7.3 ignoreCase

The value of the **ignoreCase** property is a Boolean value indicating whether the flags contained the character "**i**".

Mark S. Miller 1/19/09 11:12 PM
Deleted: This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.10.7.4 multiline

The value of the **multiline** property is a Boolean value indicating whether the flags contained the character "**m**".

Mark S. Miller 1/19/09 11:13 PM
Deleted: This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.10.7.5 lastIndex

The value of the **lastIndex** property is an integer that specifies the string position at which to start the next match. This property shall have the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.11 Error Objects

Instances of Error objects are thrown as exceptions when runtime errors occur. The Error objects may also serve as base objects for user-defined exception classes.

15.11.1 The Error Constructor Called as a Function

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

15.11.1.1 Error (message)

The [[Parent]] property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Error.prototype** (15.11.3.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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

The [[Extensible]] property of the newly constructed object is set to **true**.

If the argument *message* is not **undefined**, the **message** property of the newly constructed object is set to ToString(*message*). Otherwise, the message property is set to the empty string.

pratapL 1/19/09 8:57 PM
Comment: ALP: Should this just defer to 15.11.2.1?

15.11.2 The Error Constructor

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

15.11.2.1 new Error (message)

The [[Parent]] property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Error.prototype** (15.11.3.1).

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

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

The [[Extensible]] property of the newly constructed object is set to **true**.

If the argument *message* is not **undefined**, the **message** property of the newly constructed object is set to ToString(*message*).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.11.3 Properties of the Error Constructor

The value of the internal [[Parent]] property of the Error constructor is the Function prototype object (15.3.4).

The **`length`** property of the Error constructor is 1.

15.11.4 Properties of the Error Prototype Object

The Error prototype object is itself an Error object (its [[Class]] is **`"Error"`**).

The value of the internal [[Parent]] property of the Error prototype object is the Object prototype object (15.2.3.1).

15.11.4.1 N/A

15.11.4.2 Error.prototype.name

The initial value of **`Error.prototype.name`** is **`"Error"`**.

15.11.4.3 Error.prototype.message

The initial value of **`Error.prototype.message`** is an implementation-defined string.

15.11.4.4 Error.prototype.toString ( )

Returns an implementation defined string.

15.11.5 Properties of Error Instances

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

15.11.6 Native Error Types Used in This Standard

One of the *NativeError* objects below is thrown when a runtime error is detected. All of these objects share the same structure, as described in 15.11.7.

15.11.6.1 EvalError

Indicates that the global function **`eval`** was used in a way that is incompatible with its definition. See 15.1.2.1.

15.11.6.2 RangeError

Indicates a numeric value has exceeded the allowable range. See 15.4.2.2, 15.4.5.1, 15.7.4.5, 15.7.4.6, and 15.7.4.7.

15.11.6.3 ReferenceError

Indicate that an invalid reference value has been detected. See 8.7.1, and 8.7.2.

15.11.6.4 SyntaxError

Indicates that a parsing error has occurred. See 15.1.2.1, 15.3.2.1, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, and 15.10.4.1.

15.11.6.5 TypeError

Indicates the actual type of an operand is different than the expected type. See 8.6.2, 8.6.2.6, 9.9, 11.2.2, 11.2.3, 11.8.6, 11.8.7, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.5.3, 15.4.4.2, 15.4.4.3, 15.5.4.2, 15.5.4.3, 15.6.4, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.9, 15.9.5.27, 15.10.4.1, and 15.10.6.

15.11.6.6 URIError

Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition. See 15.1.3.

15.11.7 *NativeError* Object Structure

When an SES implementation detects a runtime error, it throws an instance of one of the *NativeError* objects defined in 15.11.6. Each of these objects has the structure described below, differing only in the name used as the constructor name instead of *NativeError*, in the **`name`** property of the prototype object, and in the implementation-defined **`message`** property of the prototype object.

For each error object, references to *NativeError* in the definition should be replaced with the appropriate error object name from 15.11.6.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:14 PM
Deleted: Besides the internal properties and the

Mark S. Miller 1/19/09 11:14 PM
Deleted: (whose value is **1**),

Mark S. Miller 1/19/09 11:14 PM
Deleted: has the following property:

pratapL 1/19/09 8:57 PM
Comment: Herman Venter says these do not have specified attributes.

Mark S. Miller 1/19/09 11:14 PM
Deleted: **15.11.3.1 Error.prototype**
The initial value of **`Error.prototype`** is the Error prototype object (15.11.4).
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:14 PM
Deleted: **Error.prototype.constructor**

Mark S. Miller 1/19/09 11:14 PM
Deleted: The initial value of **`Error.prototype.constructor`** is the built-in **`Error`** constructor.

pratapL 1/19/09 8:57 PM
Comment: Deviations doc item 3.37 suggests codifying this behaviour.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

15.11.7.1 *NativeError* Constructors Called as Functions

When a *NativeError* constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the object as a function is equivalent to calling it as a constructor with the same arguments.

15.11.7.2 *NativeError* (message)

The [[Parent]] property of the newly constructed object is set to the prototype object for this error constructor. The [[Class]] property of the newly constructed object is set to `"Error"`. The [[Extensible]] property of the newly constructed object is set to **true**.

If the argument *message* is not **undefined**, the **`message`** property of the newly constructed object is set to ToString(*message*).

15.11.7.3 The *NativeError* Constructors

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

15.11.7.4 New *NativeError* (message)

The [[Parent]] property of the newly constructed object is set to the prototype object for this *NativeError* constructor. The [[Class]] property of the newly constructed object is set to `"Error"`. The [[Extensible]] property of the newly constructed object is set to **true**.

If the argument *message* is not **undefined**, the **`message`** property of the newly constructed object is set to ToString(*message*).

15.11.7.5 Properties of the *NativeError* Constructors

The value of the internal [[Parent]] property of a *NativeError* constructor is the Function prototype object (15.3.4).

The **`length`** property of each *NativeError* constructor is 1.

15.11.7.6 *N/A*

15.11.7.7 Properties of the *NativeError* Prototype Objects

Each *NativeError* prototype object is an Error object (its [[Class]] is `"Error"`).

The value of the internal [[Parent]] property of each *NativeError* prototype object is the Error prototype object (15.11.4).

15.11.7.8 *N/A*

15.11.7.9 *NativeError*.prototype.name

The initial value of the **`name`** property of the prototype for a given *NativeError* constructor is the name of the constructor (the name used instead of *NativeError*).

15.11.7.10 *NativeError*.prototype.message

The initial value of the **`message`** property of the prototype for a given *NativeError* constructor is an implementation-defined string.

NOTE

The prototypes for the NativeError constructors do not themselves provide a **`toString`** *function, but instances of errors will inherit it from the Error prototype object.*

15.11.7.11 Properties of *NativeError* Instances

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

15.12 The JSON Object

The **JSON** object is a single object that contains two functions, **parse** and **stringify**, that are used to parse and construct JSON texts. The JSON Data Interchange Format is described in RFC 4627 <http://www.ietf.org/rfc/rfc4627.txt?number=4627>.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:17 PM
Deleted: Besides the internal properties and the

Mark S. Miller 1/19/09 11:17 PM
Deleted: (whose value is **1**),

Mark S. Miller 1/19/09 11:17 PM
Deleted: has the following property:

Mark S. Miller 1/19/09 11:17 PM
Deleted: ***NativeError*.prototype**

Mark S. Miller 1/19/09 11:17 PM
Deleted: The initial value of *NativeError***`.prototype`** is a *NativeError* prototype object (15.11.7.7). Each *NativeError* constructor has a separate prototype object.
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

Mark S. Miller 1/19/09 11:17 PM
Deleted: ***NativeError*.prototype.constructor**

Mark S. Miller 1/19/09 11:17 PM
Deleted: The initial value of the **`constructor`** property of the prototype for a given *NativeError* constructor is the *NativeError* constructor function itself (15.11.7).

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

The value of the internal [[Parent]] property of the JSON object is the Object prototype object (15.2.3.1). The value of the internal [[Class]] property of the JSON object is `"JSON"`. The value of the [[Extensible]] property of the JSON object is set to **false**.

Mark S. Miller 1/19/09 6:01 PM
Deleted: [[Prototype]]

pratapL 1/19/09 8:57 PM
Comment: From DEC:
Using Math as the model for the JSON object. I am not aware that [[Class]] is seen anywhere except in the useless text returned by object.toString. I see no harm in JSON being mutable. Caja can lock it down if it wants to.

Mark S. Miller 1/19/09 11:18 PM
Deleted: true

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

The JSON object does not have a [[Call]] property; it is not possible to invoke the JSON object as a function.

JSON.stringify produces a string that conforms to the following grammar. JSON.parse accepts a string that conform to the following grammar. JSON recognizes <SP>, <TAB>, <CR>, and <LF> as white space. Only those characters are recognized as white space.

JSONValue **:**
  NullLiteral
  BooleanLiteral
  JSONObject
  JSONArray
  JSONString
  JSONNumber

JSONObject **:**
  { }
  { JSONMemberList}

JSONMember **:**
  JSONString **:** *JSONValue*

JSONMemberList **:**
  JSONMember
  JSONMemberList **,** *JSONMember*

JSONArray **:**
  [ ]
  [JSONElementList]

JSONElementList **:**
  JSONValue
  JSONElementList , JSONValue

JSONString **::**
  "JSONCharacters$_{opt}$ "

JSONCharacters **::**
  JSONCharacter JSONCharacters$_{opt}$

JSONCharacter **::**

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

*Any Unicode character except U+0000 thru U+001F or double-quote " or backslash *
\ JSONEscapeSequence

JSONEscapeSequence **::**
JSONEscapeCharacter
u HexDigit HexDigit HexDigit HexDigit

JSONEscapeCharacter **::** *one of*
" / \ b f n r t

JSONNumber **::**
$-_{opt}$ *JSONInteger JSONFraction*$_{opt}$ *JSONExponent*$_{opt}$

JSONInteger **::**
DecimalDigit
JSONDigit DecimalDigits

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

JSONFraction **::**
. DecimalDigits

JSONExponent **::**
ExponentIndicator SignedInteger

15.12.1 parse (text [, reviver])

The **parse** function parses a JSON text (a JSON formatted string) and produces a SES value. The JSON format is a restricted form of SES literal. JSON objects are realized as SES objects. JSON Arrays are realized as SES arrays. JSON strings, numbers, booleans, and null are realized as SES strings, numbers, booleans, and null. JSON uses a more limited set of white space characters than *WhiteSpace*. The process of parsing is similar to 11.1.4 and 11.1.5 as constrained by the JSON grammar.

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

Mark S. Miller 1/19/09 5:04 PM
Deleted: ECMAScript

The optional **reviver** parameter is a function that takes two parameters, (key, value). It can filter and transform the results. It is called with each of the key/value pairs produced by the parse, and its return value is used instead of the orginal value. If it returns what it received, the structure is not modified. If it returns **undefined** then the member is deleted from the result.

1. ToString of *text*.
2. Parse Result(1) as a JSON value. Throw a **SyntaxError** exception if the *text* did not conform to the JSON grammar for JSON values. JSON objects will produce objects as if made by the original Object constructor. JSON arrays will produce arrays as if made by the original Array constructor. JSON numbers will produce number values. JSON strings will produce string values. JSON true, false, and null will produce the true, false, and null values. No other values are possible.
3. If IsCallable(*reviver*) is **true**
    a. Produce a new object as if by the original Object constructor.
    b. Call the [[Put]] method of Result(3.a) with an empty string and Result(2).
    c. Call the abstract operation *walk*, passing Result(3.b) and the empty string. The abstract operation *Walk* is described below.
    d. Return Result(3.c).
4. Else
    a. Return Result(2).

The abstract operation *Walk* is a recursive abstract operation that takes two parameters: a *holder* object and the *name* of a property in that object.

1. Call the [[Get]] method of the *holder* with *name*.
2. If Result(1) is an object, and IsCallable(Result(1)) is **false**, then
    a. If the [[Class]] property of Result(1) is "Array"
        i. Set *I* to 0.
        ii. While *I* is less than then **length** of Result(1)
            1. Call the internal *walk* function, passing Result(1) and *I*.
            2. Call the [[Put]] method of Result(1) with Result(2.a.ii.1).
            3. Add 1 to *I*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

b. Else
   i. Produce an array using the original Object.keys method with Result(1).
   ii. For each element in Result(2.b.i)
      1. Call the internal *walk* function, passing Result(1) and Result(2.b.ii).
      2. If Result(2.b.ii.1) is **undefined**
         a. Call the [[Delete]] method of Result(1) with Result(2.b.ii).
      3. Else
         a. Call the [[Put]] method of Result(1) with Result(2.b.ii.1).
3. Call *reviver* with *name* and Result(1).
4. Return Result(3).

Mark S. Miller 1/19/09 11:21 PM
Deleted: as a method of *holder*

NOTE
In the case where there are duplicate name strings within an object, lexically preceeding values for the same key shall be overwritten.

15.12.2 stringify (value [, replacer [, space]])

The stringify function produces a JSON formatted string that captures information from a JavaScript value. It can take three parameters. The first parameter is required. The *value* parameter is a JavaScript value is usually an object or array, although it can also be a string, boolean, number or **null**. The optional *replacer* parameter is either a function that alters the way objects and arrays are stringified, or an array of strings that acts as a whitelist for selecting the keys that will be stringified. The optional *space* parameter is a string or number that allows the result to have white space injected into it to improve human readability.

JSON structures are allowed to be nested to any depth, but they must be acyclic. If the value is a cyclic structure, then the stringify function must throw an Error. This is an example of a value that cannot be stringified:

```
a = [];
a[0] = a;
my_text = JSON.stringify(a); // This must throw an Error.
```

The **null** value is rendered in JSON text as the string `null`.
The **true** value is rendered in JSON text as the string `true`.
The **false** value is rendered in JSON text as the string `false`.

String values are wrapped in double quotes. The characters " and \ are escaped with \ prefixes. Control characters are replaced with escape sequences `\u`HHHH, or with the shorter forms, `\b` (backspace), `\f` (formfeed), `\n` (newline), `\r` (carriage return), `\t` (tab).

Finite numbers are stringifyed by **String(**number**)**. **NaN** and **Infinity** regardless of sign are represented as the string `null`.

Values that do not have a JSON representation (such as **undefined** and functions) do not produce a string. Instead they produce the undefined value. In arrays these values are represented as the string `null`. In objects an unrepresentable value causes the property to be excluded from stringification.

An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right brace. A property is a quoted string representing the key or property name, a colon, and then the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.

These are the steps in stringifying an object:

1. Create a new array by the original Array method.
2. Let *stack* be Result(1).
3. Let *indent* be the empty string.
4. If *space* is a number
   a. Set *gap* to a string containing space space characters. This will be the empty string if space is less than 1.
5. Else if *space* is a string
   a. Set *gap* to *space*.
6. Else

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

    a. Set *gap* to the empty string.
7. Create a new Object by the original Object method.
8. Call the [[Put]] method of Result(7) with the empty string and *value*.
9. Call the abstract operation *Str* with the empty string and Result(7).
10. Return Result(9).

The internal abstract operation *Str*(*key*, *holder*) has access to the *replacer* from the invocation of the stringify method. Its algorithm is as follows:

1. Call the [[Get]] method of *holder* with *key*.
2. Let *value* be Result(1).
3. If *value* is an object
    a. Call the [[Get]] method on *value* with "toJSON".
    b. If IsCallable(Result(3.a)) is **true**
        i. Call Result(3.a) as a method of *value* with *key*.
        ii. Let *value* be Result(3.b.i).
4. If IsCallable(replacer) is **true**
    a. Call *replacer* with *key* and *value*.
    b. Let *value* be Result(4.a).
5. If *value* is **null** then return "null".
6. If *value* is **true** then return "true".
7. If *value* is **false** then return "false".
8. If *value* is a string, then return the result of calling the abstract operation *Quote* with *value*.
9. If *value* is a number
    a. If *value* is finite then return *value*.
    b. Return "null".
10. If *value* is an object, and IsCallable(*value*) is **false**
    a. If the [[Class]] property of *value* is "Array" then
        i. Call the abstract operation *JA* with *value*.
        ii. Return Result(10.a.1).
    b. Call the abstract operation *JO* with *value*.
    c. Return Result(10.b).
11. Return **undefined**.

Mark S. Miller 1/19/09 11:22 PM
Deleted: as a method of *holder*

The abstract operation *Quote*(*value*) wraps a string value in double quotes and escapes characters within it.

1. Let *product* be the double quote character.
2. For each character in value
    a. If Result(2) is the double quote character or backslash character
        i. Let *product* be the concatenation of *product* and the backslash character.
        ii. Let *product* be the concatenation of *product* and Result(2).
    b. Else If Result(2) is backspace, formfeed, newline, carriage return, or tab
        i. Let *product* be the concatenation of *product* and the backslash character.
        ii. Let *product* be the concatenation of *product* and the lowercase letter b, f, n, r, or t.
    c. Else If Result(2) is a control character having a value less than the space character
        i. Let *product* be the concatenation of *product* and the backslash character.
        ii. Let *product* be the concatenation of *product* and the lowercase u character.
        iii. Convert the numeric value of Result(2) to a string of 4 base 16 digits.
        iv. Let *product* be the concatenation of *product* and Result(2.c.3).
    d. Else
        i. Let *product* be the concatenation of *product* and Result(2).
3. Let *product* be the concatenation of *product* and the double quote character.
4. Return *product*.

The abstract operation *JO*(*value*) serializes an object. It has access to the *stack*, *indent*, *gap*, *replacer*, and *space* of the invocation of the stringify method.

1. Call the original *indexOf* method on *stack* with *value*.
2. If Result(1) is not -1 then throw an Error because the structure is cyclical.
3. Push *value* onto *stack*.
4. Let *stepback* be *indent*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

5. Let *indent* be the concatenation of *indent* and *gap*.
6. If the [[Class]] property of *replacer* is "Array"
   a. Let *K* be the replacer parameter.
7. Else
   a. Call the original Object.keys method with *value*.
   b. Let *K* be Result(6.a).
8. Create a new array by the original Array method.
9. Let *partial* be Result(8).
10. For each element of *K*.
    a. Call the *str* function with Result(10) and *value*.
    b. If Result(10.a) is not undefined
       i. Call *Quote* with Result(10).
       ii. Let *member* be Result(10.b.i).
       iii. Let *member* be the concatenation of *member* and the colon character.
       iv. If *gap* is not empty string
           1. Let *member* be the concatenation of *member* and the *space* character.
       v. Else
           1. Let *member* be the concatenation of *member* and the Result(10.a).
       vi. Push Result(10.b.v) onto *partial*.
11. If Length(*partial*) is 0 then
    a. Let *final* be "{}".
12. Else
    a. If *gap* is the empty string
       i. Call the original join method of *partial* with the comma character.
       ii. Concatenate "{" and Result(12.a.i) and "}".
       iii. Set *final* to Result(12.a.ii).
    b. Else
       i. Concatenate the comma character and the line feed character and *indent*.
       ii. Call the original join method of *partial* with Result(12.b.i).
       iii. Concatenate "{" and the line feed character and *indent* and Result(12.b.ii) and the line feed character and *stepback* and "}".
       iv. Set *final* to Result(12.b.iii).
13. Pop the *stack*.
14. Let *indent* be *stepback*.
15. Return *final*.

The abstract operation *JA*(*value*) serializes an array. It has access to the *stack*, *indent*, *gap*, and *space* of the invocation of the stringify method. The representation of arrays includes only the elements between zero and array.length – 1. Named properties are excluded from the stringification. An array is stringified as an open left bracket, elements separated by comma, and a closing right bracket.

1. Call the original *indexOf* method on *stack* with *value*.
2. If Result(1) is not -1 then throw an Error because the structure is cyclical.
3. Push *value* onto *stack*.
4. Let *stepback* be *indent*.
5. Let *indent* be the concatenation of *indent* and *gap*.
6. Create a new array by the original Array method.
7. Let *partial* be Result(6).
8. For each *index* in *value*.
   a. Call the *str* function with Result(8) and *value*.
   b. If Result(8.a) is **undefined**
      i. Push null on *partial*.
   c. Else
      i. Push Result(8.a).
9. If Length(*partial*) is 0 then
   a. Let *final* be "[]".
10. Else
    a. If *gap* is the empty string
       i. Call the original join method of *partial* with the comma character.
       ii. Concatenate "[" and Result(12.a.i) and "]".

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

            iii. Set *final* to Result(12.a.ii).
    b. Else
            i. Concatenate the comma character and the line feed character and *indent*.
            ii. Call the original join method of *partial* with Result(10.b.i).
            iii. Concatenate "[" and the line feed character and *indent* and Result(10.b.ii) and the line feed character and *stepback* and "]"
            iv. Set *final* to Result(10.b.iii).
11. Pop the *stack*.
12. Let *indent* be *stepback*.
13. Return *final*.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

16 Errors

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

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

Attempts to define an *ObjectLiteral* that has multiple **get** property assignments with the same name or multiple **set** property assignments with the same name.

Attempts to define an *ObjectLiteral* that has both an accessor property assignment and a **get** or **set** property assignment with the same name.

Errors in regular expression literals.

Violation of strict mode restriction whose detection does not require program execution.

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

Improper uses of `return`, `break`, and `continue`.

Using the `eval` property other than via a direct call.

Errors in regular expression literals.

Attempts to call PutValue on a value that is not a reference (for example, executing the assignment statement `3=4`).

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

Mark S. Miller 1/19/09 11:25 PM
Comment: Tighten

Mark S. Miller 1/19/09 11:25 PM
Comment: oops. Lost the distinction.

Mark S. Miller 1/19/09 11:26 PM
Deleted: An implementation shall report all errors as specified, except for the following:
An implementation may extend program and regular expression syntax. To permit this, all operations (such as calling `eval`, using a regular expression literal, or using the `Function` or `RegExp` constructor) that are allowed to throw **SyntaxError** are permitted to exhibit implementation-defined behaviour instead of throwing **SyntaxError** when they encounter an implementation-defined extension to the program or regular expression syntax.
An implementation may provide additional types, values, objects, properties, and functions beyond those described in this specification. This may cause constructs (such as looking up a variable in the global scope) to have implementation-defined behaviour instead of throwing an error (such as **ReferenceError**).
An implementation is not required to detect **EvalError**. If it chooses not to detect **EvalError**, the implementation must allow `eval` to be used indirectly and/or allow assignments to `eval`.
An implementation may define behaviour other than throwing **RangeError** for `toFixed`, `toExponential`, and `toPrecision` when the *fractionDigits* or *precision* argument is outside the specified range.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Annex A
(informative)

Grammar Summary

A.1 Lexical Grammar

SourceCharacter **::** See clause 6
any Unicode character

InputElementDiv **::** See clause 7
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp **::** *See clause 7*
WhiteSpace
LineTerminator
Comment
Token

Mark S. Miller 1/19/09 11:27 PM
Deleted:
RegularExpressionLiteral

WhiteSpace **::** *See 7.2*
<*TAB*>
<*VT*>
<*FF*>
<*SP*>
<*NEL*>
<*NBSP*>
<*ZWSP*>
<*BOM*>
<*USP*>

LineTerminator **::** *See 7.3*
<*LF*>
<*CR*>

Mark S. Miller 1/19/09 11:27 PM
Deleted:
<*LS*>
<*PS*>

LineTerminatorSequence **::** *See 7.3*
<*LF*>
<*CR*> [lookahead ∉ <*LF*>]
<*CR*> <*LF*>

Mark S. Miller 1/19/09 11:27 PM
Deleted:
<*LS*>
<*PS*>

Comment **::** *See 7.4*
MultiLineComment
SingleLineComment

MultiLineComment **::** *See 7.4*
/* *MultiLineCommentChars*$_{opt}$ ***/**

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

MultiLineCommentChars **::** *See 7.4*
  MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
  `*` *PostAsteriskCommentChars$_{opt}$*

PostAsteriskCommentChars **::** *See 7.4*
  MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{opt}$
  `*` *PostAsteriskCommentChars$_{opt}$*

MultiLineNotAsteriskChar **::** *See 7.4*
  SourceCharacter **but not** *asterisk* `*`

MultiLineNotForwardSlashOrAsteriskChar **::** *See 7.4*
  SourceCharacter **but not** *forward-slash* `/` **or** *asterisk* `*`

SingleLineComment **::** *See 7.4*
  `//` *SingleLineCommentChars$_{opt}$*

SingleLineCommentChars **::** *See 7.4*
  SingleLineCommentChar SingleLineCommentChars$_{opt}$

SingleLineCommentChar **::** *See 7.4*
  SourceCharacter **but not** *LineTerminator*

Token **::** *See 7.5*
  IdentifierName
  Punctuator
  NumericLiteral
  StringLiteral

ReservedWord **::** See 7.5.1
  Keyword
  FutureReservedWord
  NullLiteral
  BooleanLiteral

Keyword **:: one of** See 7.5.2

break	**else**	**new**	**var**
case	**finally**	**return**	**void**
catch	**for**	**switch**	**while**
continue	**function**	**this**	
default	**if**	**throw**	**debugger**
delete	**in**	**try**	**eval**
do	**instanceof**	**typeof**	**arguments**

FutureReservedWord **:: one of** See 7.5.3

abstract	**enum**	**int**	**short**
boolean	**export**	**interface**	**static**
byte	**extends**	**long**	**super**
char	**final**	**native**	**synchronized**
class	**float**	**package**	**throws**
const	**goto**	**private**	**transient**

Mark S. Miller 1/19/09 11:29 PM
Deleted: with

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

	implements	**protected**	**volatile**
double	**import**	**public**	
with	**yield**	**lambda**	**let**

Identifier **::** *See 7.6*
IdentifierName **but not** *ReservedWord*

IdentifierName **::** *See 7.6*
IdentifierStart
IdentifierName IdentifierPart

IdentifierStart **::** *See 7.6*
UnicodeLetter
$

**** *UnicodeEscapeSequence*

IdentifierPart **::** *See 7.6*
IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
UnicodeEscapeSequence

UnicodeLetter See 7.6
any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Nl)”.

UnicodeCombiningMark See 7.6
any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

UnicodeDigit See 7.6
any character in the Unicode category “Decimal number (Nd)”

UnicodeConnectorPunctuation See 7.6
any character in the Unicode category “Connector punctuation (Pc)”

UnicodeEscapeSequence :: *See 7.6*

\u *HexDigit HexDigit HexDigit HexDigit*

HexDigit **:: one of** See 7.6
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

Punctuator **:: one of** See 7.7

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

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

{ **}** **(** **)** **[** **]**

DivPunctuator **:: one of** See 7.7
/ **/=**

Literal :: *See 7.8*
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral

NullLiteral :: *See 7.8.1*
null

BooleanLiteral :: *See 7.8.2*
true
false

NumericLiteral :: *See 7.8.3*
DecimalLiteral
HexIntegerLiteral

DecimalLiteral :: *See 7.8.3*
DecimalIntegerLiteral **.** *DecimalDigits$_{opt}$ ExponentPart$_{opt}$*
. *DecimalDigits ExponentPart$_{opt}$*
DecimalIntegerLiteral ExponentPart$_{opt}$

DecimalIntegerLiteral :: *See 7.8.3*
0
NonZeroDigit DecimalDigits$_{opt}$

DecimalDigits :: *See 7.8.3*
DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **:: one of** See 7.8.3
0 1 2 3 4 5 6 7 8 9

ExponentIndicator **:: one of** See 7.8.3
e E

SignedInteger **::** *See 7.8.3*
DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

HexIntegerLiteral **::** *See 7.8.3*
    0x *HexDigit*
    0X *HexDigit*
    HexIntegerLiteral HexDigit

StringLiteral **::** *See 7.8.4*
    " *DoubleStringCharacters$_{opt}$* **"**
    ' *SingleStringCharacters$_{opt}$* **'**

DoubleStringCharacters **::** *See 7.8.4*
    DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::** *See 7.8.4*
    SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **::** *See 7.8.4*
    SourceCharacter **but not** *double-quote* **"** **or** *backslash* **** **or** *LineTerminator*
    **** *EscapeSequence*
    LineContinuation

SingleStringCharacter **::** *See 7.8.4*
    SourceCharacter **but not** *single-quote* **'** **or** *backslash* **** **or** *LineTerminator*
    **** *EscapeSequence*
    LineContinuation

LineContinuation **::** *See 7.8.4*
    **** *LineTerminatorSequence*

EscapeSequence **::** *See 7.8.4*
    CharacterEscapeSequence
    0 [lookahead ∉ *DecimalDigit*]
    HexEscapeSequence
    UnicodeEscapeSequence

CharacterEscapeSequence **::** *See 7.8.4*
    SingleEscapeCharacter
    NonEscapeCharacter

SingleEscapeCharacter **::** **one of** *See 7.8.4*
    ' " \\ b f n r t v

NonEscapeCharacter **::** *See 7.8.4*
    SourceCharacter **but not** *EscapeCharacter* **or** *LineTerminator*

EscapeCharacter **::** *See 7.8.4*
    SingleEscapeCharacter
    DecimalDigit
    x
    u

HexEscapeSequence **::** *See 7.8.4*
    x *HexDigit HexDigit*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

UnicodeEscapeSequence **::** *See 7.8.4*
  u *HexDigit HexDigit HexDigit HexDigit*

BackslashSequence :: *See 7.8.5*
  **** *NonTerminator*

NonTerminator :: *See 7.8.5*
  SourceCharacter **but not** *LineTerminator*

Mark S. Miller 1/19/09 11:31 PM
Deleted: *RegularExpressionLiteral* **::** *See 7.8.5*
/ *RegularExpressionBody* **/** *RegularExpressionFlags*

RegularExpressionBody **::** *See 7.8.5*
RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars **::** *See 7.8.5*
[empty]
RegularExpressionChars RegularExpressionChar

RegularExpressionFirstChar **::** *See 7.8.5*
NonTerminator **but not * or \\ or / or [**
BackslashSequence
RegularExpressionClass

RegularExpressionChar :: *See 7.8.5*
NonTerminator **but not \\ or / or [**
BackslashSequence
RegularExpressionClass

Mark S. Miller 1/19/09 11:31 PM
Deleted: *RegularExpressionClass* **::** *See 7.8.5*
[*RegularExpressionClassPreamble*
RegularExpressionClassChars **]**

RegularExpressionClassPreamble **::** *See 7.8.5*
[empty]
^
-
^-

RegularExpressionClassChars **::** *See 7.8.5*
[empty]
RegularExpressionClassChars
RegularExpressionClassChar

RegularExpressionClassChar **::** *See 7.8.5*
NonTerminator ***but not] or \\ or -***
- RegularExpressionClassChar
BackslashExpression

RegularExpressionFlags :: *See 7.8.5*
[empty]
RegularExpressionFlags IdentifierPart

A.2 Number Conversions

StringNumericLiteral **:::** See 9.3.1
  StrWhiteSpace$_{opt}$
  StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace ::: *See 9.3.1*
  StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar ::: *See 9.3.1*
  WhiteSpace
  LineTerminator

StrNumericLiteral ::: *See 9.3.1*
  StrDecimalLiteral
  HexIntegerLiteral

StrDecimalLiteral **:::** See 9.3.1
  StrUnsignedDecimalLiteral
  + *StrUnsignedDecimalLiteral*
  - *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral ::: *See 9.3.1*
  Infinity
  DecimalDigits **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
  . *DecimalDigits ExponentPart*$_{opt}$
  DecimalDigits ExponentPart$_{opt}$

DecimalDigits ::: *See 9.3.1*
  DecimalDigit
  DecimalDigits DecimalDigit

DecimalDigit **::: one of** See 9.3.1
  0 1 2 3 4 5 6 7 8 9

ExponentPart ::: *See 9.3.1*
  ExponentIndicator SignedInteger

ExponentIndicator **::: one of** See 9.3.1
  e E

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

SignedInteger ::: See 9.3.1
DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

HexIntegerLiteral ::: See 9.3.1
0x *HexDigit*
0X *HexDigit*
HexIntegerLiteral HexDigit

HexDigit **::: one of** See 9.3.1
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

A.3 Expressions

PrimaryExpression **:** See 11.1
Identifier
Literal
ArrayLiteral
ObjectLiteral
(*Expression* **)**

Mark S. Miller 1/19/09 11:31 PM
Deleted: this

ArrayLiteral : See 11.1.4
[$Elision_{opt}$ **]**
[*ElementList* **]**
[*ElementList* , $Elision_{opt}$ **]**

ElementList : See 11.1.4
$Elision_{opt}$ *AssignmentExpression*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*

Elision : See 11.1.4
,
Elision **,**

ObjectLiteral : See 11.1.5
{ }
{ *PropertyNameAndValueList* **}**
{ *PropertyNameAndValueList* **, }**

PropertyNameAndValueList : See 11.1.5
PropertyAssignment
PropertyNameAndValueList **,** *PropertyAssignment*

PropertyAssignment : See 11.1.5
PropertyName **:** *AssignmentExpression*
get *PropertyName* **( ) {** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

PropertyName : *See 11.1.5*
  IdentifierName
  StringLiteral
  NumericLiteral

PropertySetParameterList : *See 11.1.5*
  Identifier

MemberExpression : *See 11.2*
  PrimaryExpression
  FunctionExpression
  MemberExpression **[** *Expression* **]**
  MemberExpression **.** *IdentifierName*
  new *MemberExpression Arguments*

NewExpression : *See 11.2*
  MemberExpression
  new *NewExpression*

CallExpression : *See 11.2*
  MemberExpression Arguments
  CallExpression Arguments
  CallExpression **[** *Expression* **]**
  CallExpression **.** *IdentifierName*

Arguments : *See 11.2*
  ( )
  (*ArgumentList* **)**

ArgumentList : *See 11.2*
  AssignmentExpression
  ArgumentList **,** *AssignmentExpression*

LeftHandSideExpression : *See 11.2*
  NewExpression
  CallExpression

PostfixExpression : *See 11.3*
  LeftHandSideExpression
  LeftHandSideExpression [no *LineTerminator* here] **++**
  LeftHandSideExpression [no *LineTerminator* here] **--**

UnaryExpression : *See 11.4*
  PostfixExpression
  delete *MemberExpression*
  void *UnaryExpression*
  typeof *UnaryExpression*
  ++ *UnaryExpression*
  -- *UnaryExpression*
  + *UnaryExpression*
  - *UnaryExpression*
  ~ *UnaryExpression*
  ! *UnaryExpression*

Mark S. Miller 1/19/09 11:34 PM
Comment: Should apply to ES3.1 as well. SES still needs to ban when MemberExpression is PrimaryExpression.

Mark S. Miller 1/19/09 11:33 PM
Deleted: *UnaryExpression*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

MultiplicativeExpression : *See 11.5*
UnaryExpression
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

AdditiveExpression : *See 11.6*
MultiplicativeExpression
AdditiveExpression **+** *MultiplicativeExpression*
AdditiveExpression **-** *MultiplicativeExpression*

ShiftExpression : *See 11.7*
AdditiveExpression
ShiftExpression **<<** *AdditiveExpression*
ShiftExpression **>>** *AdditiveExpression*
ShiftExpression **>>>** *AdditiveExpression*

RelationalExpression : *See 11.8*
ShiftExpression
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*
RelationalExpression **instanceof** *ShiftExpression*
RelationalExpression **in** *ShiftExpression*

RelationalExpressionNoIn : *See 11.8*
ShiftExpression
RelationalExpressionNoIn **<** *ShiftExpression*
RelationalExpressionNoIn **>** *ShiftExpression*
RelationalExpressionNoIn **<=** *ShiftExpression*
RelationalExpressionNoIn **>=** *ShiftExpression*
RelationalExpressionNoIn **instanceof** *ShiftExpression*

EqualityExpression : *See 11.9*
RelationalExpression
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*

EqualityExpressionNoIn : *See 11.9*
RelationalExpressionNoIn
EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

BitwiseANDExpression : *See 11.10*
EqualityExpression
BitwiseANDExpression **&** *EqualityExpression*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

BitwiseANDExpressionNoIn : *See 11.10*
  EqualityExpressionNoIn
  BitwiseANDExpressionNoIn **&** *EqualityExpressionNoIn*

BitwiseXORExpression : *See 11.10*
  BitwiseANDExpression
  BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseXORExpressionNoIn : *See 11.10*
  BitwiseANDExpressionNoIn
  BitwiseXORExpressionNoIn **^** *BitwiseANDExpressionNoIn*

BitwiseORExpression : *See 11.10*
  BitwiseXORExpression
  BitwiseORExpression **|** *BitwiseXORExpression*

BitwiseORExpressionNoIn : *See 11.10*
  BitwiseXORExpressionNoIn
  BitwiseORExpressionNoIn **|** *BitwiseXORExpressionNoIn*

LogicalANDExpression : *See 11.11*
  BitwiseORExpression
  LogicalANDExpression **&&** *BitwiseORExpression*

LogicalANDExpressionNoIn : *See 11.11*
  BitwiseORExpressionNoIn
  LogicalANDExpressionNoIn **&&** *BitwiseORExpressionNoIn*

LogicalORExpression : *See 11.11*
  LogicalANDExpression
  LogicalORExpression **||** *LogicalANDExpression*

LogicalORExpressionNoIn : *See 11.11*
  LogicalANDExpressionNoIn
  LogicalORExpressionNoIn **||** *LogicalANDExpressionNoIn*

ConditionalExpression : *See 11.12*
  LogicalORExpression
  LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

ConditionalExpressionNoIn : *See 11.12*
  LogicalORExpressionNoIn
  LogicalORExpressionNoIn **?** *AssignmentExpressionNoIn* **:** *AssignmentExpressionNoIn*

AssignmentExpression : *See 11.13*
  ConditionalExpression
  LeftHandSideExpression AssignmentOperator AssignmentExpression

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

AssignmentExpressionNoIn : *See 11.13*
ConditionalExpressionNoIn
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

AssignmentOperator **: one of** See 11.13
`=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

Expression : *See 11.14*
AssignmentExpression
Expression **,** *AssignmentExpression*

ExpressionNoIn **:** See 11.14
AssignmentExpressionNoIn
ExpressionNoIn **,** *AssignmentExpressionNoIn*

A.4 Statements

Statement **:** See clause 12
Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement

Mark S. Miller 1/19/09 11:34 PM
Deleted: *WithStatement*

Block : *See 12.1*
{ *StatementList$_{opt}$* **}**

StatementList : *See 12.1*
Statement
StatementList Statement

VariableStatement : *See 12.2*
var *VariableDeclarationList* **;**

VariableDeclarationList : *See 12.2*
VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclarationListNoIn : *See 12.2*
VariableDeclarationNoIn
VariableDeclarationListNoIn **,** *VariableDeclarationNoIn*

VariableDeclaration : *See 12.2*
Identifier Initialiser$_{opt}$

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

VariableDeclarationNoIn : *See 12.2*
    Identifier InitialiserNoIn$_{opt}$

Initialiser : *See 12.2*
    = *AssignmentExpression*

InitialiserNoIn : *See 12.2*
    = *AssignmentExpressionNoIn*

EmptyStatement : *See 12.3*
    ;

ExpressionStatement : *See 12.4*
    [lookahead ∉ {**{**, **function**}] *Expression* **;**

IfStatement : *See 12.5*
    if (*Expression* **)** *Statement* **else** *Statement*
    if (*Expression* **)** *Statement*

IterationStatement : *See 12.6*
    do *Statement* **while (** *Expression* **);**
    while (*Expression* **)** *Statement*
    for (*ExpressionNoIn$_{opt}$* **;** *Expression$_{opt}$* **;** *Expression$_{opt}$* **)** *Statement*
    for (var *VariableDeclarationListNoIn* **;** *Expression$_{opt}$* **;** *Expression$_{opt}$* **)** *Statement*
    for (*LeftHandSideExpression* **in** *Expression* **)** *Statement*
    for (var *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*

ContinueStatement : *See 12.7*
    continue [no *LineTerminator* here] *Identifier$_{opt}$* **;**

BreakStatement : *See 12.8*
    break [no *LineTerminator* here] *Identifier$_{opt}$* **;**

ReturnStatement : *See 12.9*
    return [no *LineTerminator* here] *Expression$_{opt}$* **;**

SwitchStatement : *See 12.11*
    switch (*Expression* **)** *CaseBlock*

Mark S. Miller 1/19/09 11:35 PM
Deleted: *WithStatement :* *See 12.10*
with (*Expression* **)** *Statement*

CaseBlock : *See 12.11*
    { *CaseClauses$_{opt}$* **}**
    { *CaseClauses$_{opt}$* *DefaultClause CaseClauses$_{opt}$* **}**

CaseClauses : *See 12.11*
    CaseClause
    CaseClauses CaseClause

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

CaseClause : *See 12.11*
  `case` *Expression* **`:`** *StatementList$_{opt}$*

DefaultClause : *See 12.11*
  `default :` *StatementList$_{opt}$*

LabelledStatement : *See 12.12*
  Identifier **`:`** *Statement*

ThrowStatement : *See 12.13*
  `throw` [no *LineTerminator* here] *Expression ;*

TryStatement : *See 12.14*
  `try` *Block Catch*
  `try` *Block Finally*
  `try` *Block Catch Finally*

Catch : *See 12.14*
  `catch (`*Identifier* **`)`** *Block*

Finally : *See 12.14*
  `finally` *Block*

DebuggerStatement **:** *See 12.15*
  `debugger ;`

A.5 Functions and Programs

FunctionDeclaration **:** See clause 13
  `function` *Identifier* **`(`** *FormalParameterList$_{opt}$* **`)`** **`{`** *FunctionBody* **`}`**

FunctionExpression **:** *See clause 13*
  `function` *Identifier$_{opt}$* **`(`** *FormalParameterList$_{opt}$* **`)`** **`{`** *FunctionBody* **`}`**

FormalParameterList **:** *See clause 13*
  Identifier
  FormalParameterList **`,`** *Identifier*

FunctionBody **:** *See clause 13*
  SourceElements

Program **:** *See clause 14*
  SourceElements

SourceElements **:** *See clause 14*
  SourceElement
  SourceElements SourceElement

SourceElement **:** *See clause 14*
  Statement
  FunctionDeclaration

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

A.6 Universal Resource Identifier Character Classes

uri **:::** See 15.1.3
uriCharacters$_{opt}$

uriCharacters **:::** See 15.1.3
uriCharacter uriCharacters$_{opt}$

uriCharacter **:::** See 15.1.3
uriReserved
uriUnescaped
uriEscaped

uriReserved **::: one of** See 15.1.3
`;   /   ?   :   @   &   =   +   $   ,`

uriUnescaped **:::** See 15.1.3
uriAlpha
DecimalDigit
uriMark

uriEscaped **:::** See 15.1.3
`%` *HexDigit HexDigit*

uriAlpha **::: one of** See 15.1.3
`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`

uriMark **::: one of** See 15.1.3
`-   _   .   !   ~   *   '   (   )`

A.7 Regular Expressions

Pattern **::** See 15.10.1
Disjunction

Disjunction :: *See 15.10.1*
Alternative
Alternative **|** *Disjunction*

Alternative :: *See 15.10.1*
[empty]
Alternative Term

Term :: *See 15.10.1*
Assertion
Atom
Atom Quantifier

Mark S. Miller 1/19/09 11:36 PM
Deleted: *UseStrictDirective* **:** *See clause 14.1*
[no *LineTerminator* here] **`use`** [no *LineTerminator* here] **`strict`** [no *LineTerminator* here] *useExtension*$_{opt}$
useExtension **:** *See clause 14.1*
, *ArbitraryInputElements*$_{opt}$
ArbitraryInputElements **:** *See clause 14.1*
[any sequence of lexical input elements that does not include any *LineTerminator* elements]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Assertion :: *See 15.10.1*
^
$
\ b
\ B

Quantifier :: *See 15.10.1*
QuantifierPrefix
QuantifierPrefix **?**

QuantifierPrefix :: *See 15.10.1*
*****
+
?
{ *DecimalDigits* **}**
{ *DecimalDigits* **, }**
{ *DecimalDigits* **,** *DecimalDigits* **}**

Atom :: *See 15.10.1*
PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(? : *Disjunction* **)**
(? = *Disjunction* **)**
(? ! *Disjunction* **)**

PatternCharacter :: *SourceCharacter but not any of:* *See 15.10.1*
^ $ \ . * + ? ( ) [ ] { } |

AtomEscape :: *See 15.10.1*
DecimalEscape
CharacterEscape
CharacterClassEscape

CharacterEscape :: *See 15.10.1*
ControlEscape
c *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

ControlEscape **:: one of** See 15.10.1
f n r t v

ControlLetter **:: one of** See 15.10.1
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

IdentityEscape :: *See 15.10.1*
SourceCharacter **but not** *IdentifierPart*

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

DecimalEscape :: *See 15.10.1*
DecimalIntegerLiteral [lookahead ∉ *DecimalDigit*]

CharacterClass :: *See 15.10.1*
[[lookahead ∉ {**^**}] *ClassRanges* **]**
[**^** *ClassRanges* **]**

ClassRanges :: *See 15.10.1*
[empty]
NonemptyClassRanges

NonemptyClassRanges :: *See 15.10.1*
ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom **-** *ClassAtom ClassRanges*

NonemptyClassRangesNoDash :: *See 15.10.1*
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash **-** *ClassAtom ClassRanges*

ClassAtom :: *See 15.10.1*
-
ClassAtomNoDash

ClassAtomNoDash :: *See 15.10.1*
SourceCharacter **but not one of \ ] -**
**** *ClassEscape*

ClassEscape :: *See 15.10.1*
DecimalEscape
b
CharacterEscape
CharacterClassEscape

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Annex B

(informative)

Compatibility

B.1 Additional Syntax

Past editions of ECMAScript have included additional syntax and semantics for specifying octal literals and octal escape sequences. These have been removed from SES and may not be implemented by a conforming SES implementation.

B.2 Additional Properties

Some implementations of ECMAScript have included additional properties for some of the standard native objects. These have been removed from SES and may not be implemented by a conforming SES implementation.

Annex C

(informative)

Note
This annex needs to be updated according to the rest of the document.

Some SES Restrictions

C.1.1 Excluded Features

1. A function may not have two or more formal parameters that have the same name. An attempt to create a such a function will fail statically, if expressed as a *FunctionDeclaration* or *FunctionExpression*.
2. For functions, if an arguments object is created, the initial values of the arguments object's "caller" and "callee" properties are the value **undefined** and may not be modified (10.5).
3. For functions, if an arguments object is created the binding of the local identifier "arguments' to the arguments object is immutable (10.6).
4. Eval code cannot instantiate variables or functions in the lexical context of its eval operator. Instead, a new variable environment is created and that environment is used for declaration binding instantiation for the eval code (10.4.2).
5. When a postfix increment operator occurs within strict mode code, its *LeftHandSide* must not be a reference to a property with the attribute value {[[Writable]]: **false**} nor to a non-existent property of an object whose [[Extensible]] property has the value **false**. In these cases a `TypeError exception` is thrown (11.3.1).
6. The same restrictions as specified for 11.3.1 apply for the postfix decrement operator (11.3.2).
7. When a **`delete`** operator occurs, a ***SyntaxError*** is thrown if its *MemberExpression* is a *PrimaryExpression*. In addition, if the property to be deleted has the attribute { [[Configurable]]:**false** }, a **TypeError** exception is thrown (11.4.1).
8. The same restrictions as specified for 11.3.1 apply for the prefix increment operator (11.4.4).
9. The same restrictions as specified for 11.3.1 apply for the prefix decrement operator (11.4.5).
10. When a simple assignment occurs, its *LeftHandSide* must not evaluate to a Reference whose base is **null**. If it does a **ReferenceError** exception is thrown. The *LeftHandSide* also may not be a reference to a property with the attribute value {[[Writable]]:**false**} nor to a non-existent property of an object whose [[Extensible]] property has the value **false**. In these cases a **`TypeError`** exception is thrown (11.13.1).
11. Code may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context should be treated as a syntax error (12.10).
12. Code uses the value of the **`eval`** property in any way other than as direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **`eval`** property is assigned to, an **EvalError** exception is thrown (15.1.2.1.1).

Mark S. Miller 1/19/09 11:53 PM
Deleted:

Mark S. Miller 1/19/09 11:53 PM
Deleted:

Mark S. Miller 1/19/09 11:53 PM
Deleted:

Mark S. Miller 1/19/09 5:05 PM
Deleted: ECMAScript…l …this edition of ECMAScript…This non-normative annex presents uniform syntax and semantics for octal literals and octal escape sequences for compatibility with some older ECMAScript programs.
B.1.1 Numeric Literals
The syntax and semantics of 7.8.3 can be extended as follows:
Syntax
NumericLiteral ::
DecimalLiteral
HexIntegerLiteral
OctalIntegerLiteral
OctalIntegerLiteral ::
0 *OctalDigit*
OctalIntegerLiteral OctalDigit
OctalDigit :: ***one of***
0 1 2 3 4 5 6 7
Semantics
The MV of *NumericLiteral* **::** *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
The MV of *OctalDigit* **::** 0 is 0. ... [36]

Mark S. Miller 1/19/09 11:39 PM
Deleted: B.1.2 String Literals ... [37]

Mark S. Miller 1/19/09 5:05 PM
Deleted: ECMAScript… …This non-norm ... [38]

Mark S. Miller 1/19/09 11:41 PM
Deleted: B.2.1 escape (string) ... [39]

Mark S. Miller 1/19/09 11:42 PM
Deleted: The Strict variant of…ECMAS ... [40]

Mark S. Miller 1/19/09 11:42 PM
Deleted: C.1 The `strict` mode

Mark S. Miller 1/19/09 11:42 PM
Deleted: strict mode …,…either …, or ... [41]

Mark S. Miller 1/19/09 11:43 PM
Deleted: strict mode …a `callee` propert ... [42]

Mark S. Miller 1/19/09 11:43 PM
Deleted: strict mode

Mark S. Miller 1/19/09 11:44 PM
Deleted: Strict mode e

Mark S. Miller 1/19/09 11:44 PM
Deleted: <#>If **this** is evaluated within str ... [43]

Mark S. Miller 1/19/09 11:45 PM
Deleted: within strict mode ... [44]

Mark S. Miller 1/19/09 11:47 PM
Deleted: within strict mode code

Mark S. Miller 1/19/09 11:47 PM
Deleted: Strict mode code

Mark S. Miller 1/19/09 11:47 PM
Deleted: If strict mode code

Mark S. Miller 1/19/09 11:47 PM
Comment: Fix

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

13. An implementation may not associate special meanings with strict mode functions to properties named "caller" or "arguments" of function instances.

C.1.2 Additional Execution Exceptions

Annex D
(informative)

Note:
This annex needs to be updated according to the rest of the document.

Correction and Clarifications in SES with Possible Compatability Impact

Annex E
(informative)

Note:
This annex needs to be updated according to the rest of the document.

Additions and Changes in SES which Introduce Incompatabilities with Edition 3.1-strict.

Mark S. Miller 1/19/09 11:48 PM
Deleted: .

Mark S. Miller 1/19/09 11:52 PM
Deleted:

Mark S. Miller 1/19/09 11:49 PM
Deleted: Edition 3.1

Mark S. Miller 1/19/09 11:49 PM
Deleted: Through out: In the Edition 3 specification the meaning of phrases such as "as if by the expression `new Array()`" are subject to misinterpretation. For Edition 3.1 the specification text for all internal references and invocations of standard built-in objects and methods has been clarified by making it implicit that the intent is that the actual built-in object is to be used rather than the current dynamic value of the correspondingly name property.
11.8.2, 11.8.3,11.8.5 While ECMAScript generally uses a left to right evaluation order the Edition 3 specification language for the > and <= operators resulted in a partial right to left order. The specification has been corrected for these operators such that it now specifies a full left to right evaluation order. However, this change of order is potentially observable if user-defined valueOf or toString methods with side-effects are invoked during the evaluation process.
11. 2.3 Edition 3.1 reverses the order of steps 2 and 3 of the algorithm. The original order as specified in Editions 1 through 3was incorrectly specified such that side-effects of evaluating *Arguments* could affect the result of evaluating *MemberExpression*.
12.2 In Edition 3 the algorithm for evaluating the production *VariableDeclaration* **:** *Identifier Initialiser* was specified in a manner that is incorrect for situations where a *VariableDeclaration* is nested within a *WithStatement* for an object that has a property name that is identical to the *Identifier* in the *VariableDeclaration*. In this situation, the Edition 3 specificaiton causes the value of the *Initialiser* to be assigned to the object's property rather than the actual variable introduced by the declaration. For Edition 3.1 the algorithm has been revised such that the value of the *Initialiser* will be assigned to the assocated variable regardless of any such nesting. Existing ECMAScript code that depends up faithful implementation of this Edition 3 semantics will not operated as expected using an implemention that conforms to the Edition 3.1 specification.
15.10.6 RegExp.prototype is now a RegExp object rather than an instance of Object. The value of its [[Class]] internal property which is observable using Object.prototype.toString is now "RegExp" rather than "Object"

Mark S. Miller 1/19/09 11:49 PM
Deleted: Edition 3.1

Mark S. Miller 1/19/09 11:50 PM
Deleted: Section 7.1 Unicode format control characters are no longer stripped from ECMAScript source text before processing.
Section 7.2 Unicode characters <NEL>, <ZWSP>, and <BOM> are now treated as whitespace.
Section 7.3 Line terminator characters that are preceded by an escape sequence are now allowed within a string literal token.
Section 7.8.5 Regular expression literals now return an unique object each time the literal is evaluated. This change is detectable by any programs that test the object identity of such literal values or that are sensitive to the shared side effects.
Section 7.8.5 in Edition 3.1 requires scan time reporting of any possible RegExp constructor errors that would ... [45]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Free printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Interne.ch

Files of this Standard can be freely downloaded from our ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

ECMA

114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009

Revision History

15 Jan 2009	pratapL	8.6.2(Table 5), 15.4.3.2, 15.4.4, 15.4.5 - changed Array.isArray specification to use [[IsArray]] internal property. 8.6.2(Table 5), 15.3.4.5, 15.3.4.5.1, 15.3.4.5.2 - improved definition of Function.prototype.bind including various bugs. 8.6.2, 10.5 - rewrote CreateArgumentsObject and associated helpers. 8.7, 8.7.1, 8.7.2, 9.10, 9.11, 9.12, 11.2.1 - allowed primitive values to be used as the base value of References, eliminating transient wrapper object creation. 10.1.1, 13, 14, 14.1, A.5 - changed method of specification for Use Strict Directives. 11.1.5 - made the name of getter/setter functions defined using object initializers be prefixed with "get " and "set ". 12.5, 12.6, 12.11 - eliminated strict mode restrictions on var statements agreed to in Kona. 12.6.4 - fixed step 8 of the algorithm to refer to the right Results. 13.2, 15.3.3, 15.3.4 - restricted use of "arguments" and "caller" properties of the Function instances and Function instances, particularly relating to strict mode. 15.3.3 - added restriction of use of Function.caller and Function.arguments Annex C Preliminary cleanup Spell checked and corrected through sections 0 through 14. Regenerated TOC.

Mark S. Miller 1/19/09 5:21 PM
Deleted: 25 March 2008 ... [46]

Mark S. Miller 1/19/09 5:14 PM
Deleted: 15 January 2009