
Information technology — Programming languages — Ruby

Technologies de l'information — Langages de programmation — Ruby



Reference number
ISO/IEC 30170:2012(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2012



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2012

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

1	Scope	1
2	Normative references	1
3	Conformance	1
4	Terms and definitions	2
5	Notational conventions	4
5.1	General description	4
5.2	Syntax	4
5.2.1	General description	4
5.2.2	Productions	5
5.2.3	Syntactic term sequences	6
5.2.4	Syntactic terms	7
5.2.5	Conceptual names	10
5.3	Semantics	10
5.4	Attributes of execution contexts	11
6	Fundamental concepts	12
6.1	Objects	12
6.2	Variables	12
6.2.1	General description	12
6.2.2	Instance variables	13
6.3	Methods	13
6.4	Blocks	14
6.5	Classes, singleton classes, and modules	14
6.5.1	General description	14
6.5.2	Classes	14
6.5.3	Singleton classes	15
6.5.4	Inheritance	16
6.5.5	Modules	17
6.6	Boolean values	18
7	Execution contexts	18
7.1	General description	18
7.2	The initial state	19
8	Lexical structure	19
8.1	General description	19
8.2	Program text	20
8.3	Line terminators	20
8.4	Whitespace	21
8.5	Comments	22
8.6	End-of-program markers	23
8.7	Tokens	23
8.7.1	General description	23
8.7.2	Keywords	23
8.7.3	Identifiers	24
8.7.4	Punctuators	25
8.7.5	Operators	25

8.7.6	Literals	26
8.7.6.1	General description	26
8.7.6.2	Numeric literals	26
8.7.6.3	String literals	29
8.7.6.3.1	General description	29
8.7.6.3.2	Single quoted strings	29
8.7.6.3.3	Double quoted strings	30
8.7.6.3.4	Quoted non-expanded literal strings	33
8.7.6.3.5	Quoted expanded literal strings	35
8.7.6.3.6	Here documents	36
8.7.6.3.7	External command execution	38
8.7.6.4	Array literals	39
8.7.6.5	Regular expression literals	42
8.7.6.6	Symbol literals	43
9	Scope of variables	44
9.1	General description	44
9.2	Scope of local variables	44
9.3	Scope of global variables	45
10	Program structure	45
10.1	Program	45
10.2	Compound statement	46
11	Expressions	47
11.1	General description	47
11.2	Logical expressions	47
11.2.1	General description	47
11.2.2	Logical NOT expressions	48
11.2.3	Logical AND expressions	49
11.2.4	Logical OR expressions	49
11.3	Method invocation expressions	50
11.3.1	General description	50
11.3.2	Method arguments	55
11.3.3	Blocks	58
11.3.4	The super expression	61
11.3.5	The yield expression	64
11.4	Operator expressions	65
11.4.1	General description	65
11.4.2	Assignments	66
11.4.2.1	General description	66
11.4.2.2	Single assignments	66
11.4.2.2.1	General description	66
11.4.2.2.2	Single variable assignments	67
11.4.2.2.3	Scoped constant assignments	69
11.4.2.2.4	Single indexing assignments	69
11.4.2.2.5	Single method assignments	70
11.4.2.3	Abbreviated assignments	71
11.4.2.3.1	General description	71
11.4.2.3.2	Abbreviated variable assignments	71
11.4.2.3.3	Abbreviated indexing assignments	72
11.4.2.3.4	Abbreviated method assignments	73
11.4.2.4	Multiple assignments	74

11.4.2.5	Assignments with rescue modifiers	78
11.4.3	Unary operator expressions	78
11.4.3.1	General description	78
11.4.3.2	The defined? expression	79
11.4.4	Binary operator expressions	81
11.5	Primary expressions	84
11.5.1	General description	84
11.5.2	Control structures	85
11.5.2.1	General description	85
11.5.2.2	Conditional expressions	85
11.5.2.2.1	General description	85
11.5.2.2.2	The if expression	85
11.5.2.2.3	The unless expression	87
11.5.2.2.4	The case expression	87
11.5.2.2.5	Conditional operator expression	88
11.5.2.3	Iteration expressions	89
11.5.2.3.1	General description	89
11.5.2.3.2	The while expression	90
11.5.2.3.3	The until expression	91
11.5.2.3.4	The for expression	91
11.5.2.4	Jump expressions	92
11.5.2.4.1	General description	92
11.5.2.4.2	The return expression	93
11.5.2.4.3	The break expression	94
11.5.2.4.4	The next expression	95
11.5.2.4.5	The redo expression	96
11.5.2.4.6	The retry expression	97
11.5.2.5	The begin expression	97
11.5.3	Grouping expression	99
11.5.4	Variable references	99
11.5.4.1	General description	99
11.5.4.2	Constants	100
11.5.4.3	Scoped constants	101
11.5.4.4	Global variables	101
11.5.4.5	Class variables	102
11.5.4.6	Instance variables	102
11.5.4.7	Local variables or method invocations	102
11.5.4.7.1	General description	102
11.5.4.7.2	Determination of the type of local variable identifiers	103
11.5.4.7.3	Local variables	103
11.5.4.7.4	Method invocations	104
11.5.4.8	Pseudo variables	104
11.5.4.8.1	General description	104
11.5.4.8.2	The nil expression	104
11.5.4.8.3	The true expression and the false expression	104
11.5.4.8.4	The self expression	105
11.5.5	Object constructors	105
11.5.5.1	Array constructor	105
11.5.5.2	Hash constructor	105
11.5.5.3	Range expression	106
12	Statements	107
12.1	General description	107

12.2	Expression statement	107
12.3	The if modifier statement	108
12.4	The unless modifier statement	108
12.5	The while modifier statement	108
12.6	The until modifier statement	109
12.7	The rescue modifier statement	109
13	Classes and modules	110
13.1	Modules	110
13.1.1	General description	110
13.1.2	Module definition	111
13.1.3	Module inclusion	112
13.2	Classes	112
13.2.1	General description	112
13.2.2	Class definition	112
13.2.3	Inheritance	114
13.2.4	Instance creation	114
13.3	Methods	115
13.3.1	Method definition	115
13.3.2	Method parameters	116
13.3.3	Method invocation	118
13.3.4	Method lookup	120
13.3.5	Method visibility	121
13.3.5.1	General description	121
13.3.5.2	Public methods	121
13.3.5.3	Private methods	121
13.3.5.4	Protected methods	121
13.3.5.5	Visibility change	122
13.3.6	The alias statement	122
13.3.7	The undef statement	123
13.4	Singleton classes	124
13.4.1	General description	124
13.4.2	Singleton class definition	125
13.4.3	Singleton method definition	126
14	Exceptions	127
14.1	General description	127
14.2	Cause of exceptions	127
14.3	Exception handling	127
15	Built-in classes and modules	128
15.1	General description	128
15.2	Built-in classes	131
15.2.1	Object	131
15.2.1.1	General description	131
15.2.1.2	Direct superclass	131
15.2.1.3	Included modules	131
15.2.1.4	Constants	131
15.2.1.5	Instance methods	132
15.2.1.5.1	Object#initialize	132
15.2.2	Module	132
15.2.2.1	General description	132
15.2.2.2	Direct superclass	132

15.2.2.3	Singleton methods	132
15.2.2.3.1	Module.constants	132
15.2.2.3.2	Module.nesting	133
15.2.2.4	Instance methods	133
15.2.2.4.1	Module#<=>	133
15.2.2.4.2	Module#<	134
15.2.2.4.3	Module#<=	134
15.2.2.4.4	Module#>	134
15.2.2.4.5	Module#>=	135
15.2.2.4.6	Module#==	135
15.2.2.4.7	Module#===	135
15.2.2.4.8	Module#alias_method	135
15.2.2.4.9	Module#ancestors	136
15.2.2.4.10	Module#append_features	136
15.2.2.4.11	Module#attr	136
15.2.2.4.12	Module#attr_accessor	137
15.2.2.4.13	Module#attr_reader	137
15.2.2.4.14	Module#attr_writer	138
15.2.2.4.15	Module#class_eval	138
15.2.2.4.16	Module#class_variable_defined?	139
15.2.2.4.17	Module#class_variable_get	140
15.2.2.4.18	Module#class_variable_set	140
15.2.2.4.19	Module#class_variables	140
15.2.2.4.20	Module#const_defined?	141
15.2.2.4.21	Module#const_get	142
15.2.2.4.22	Module#const_missing	142
15.2.2.4.23	Module#const_set	143
15.2.2.4.24	Module#constants	143
15.2.2.4.25	Module#extend_object	144
15.2.2.4.26	Module#extended	144
15.2.2.4.27	Module#include	144
15.2.2.4.28	Module#include?	145
15.2.2.4.29	Module#included	145
15.2.2.4.30	Module#included_modules	145
15.2.2.4.31	Module#initialize	145
15.2.2.4.32	Module#initialize_copy	146
15.2.2.4.33	Module#instance_methods	147
15.2.2.4.34	Module#method_defined?	147
15.2.2.4.35	Module#module_eval	148
15.2.2.4.36	Module#private	148
15.2.2.4.37	Module#protected	148
15.2.2.4.38	Module#public	148
15.2.2.4.39	Module#remove_class_variable	149
15.2.2.4.40	Module#remove_const	150
15.2.2.4.41	Module#remove_method	150
15.2.2.4.42	Module#undef_method	151
15.2.3	Class	151
15.2.3.1	General description	151
15.2.3.2	Direct superclass	151
15.2.3.3	Instance methods	151
15.2.3.3.1	Class#initialize	151
15.2.3.3.2	Class#initialize_copy	152

15.2.3.3.3	Class#new	152
15.2.3.3.4	Class#superclass	153
15.2.4	NilClass	153
15.2.4.1	General description	153
15.2.4.2	Direct superclass	153
15.2.4.3	Instance methods	153
15.2.4.3.1	NilClass#&	153
15.2.4.3.2	NilClass# 	154
15.2.4.3.3	NilClass#^	154
15.2.4.3.4	NilClass#nil?	154
15.2.4.3.5	NilClass#to_s	154
15.2.5	TrueClass	154
15.2.5.1	General description	154
15.2.5.2	Direct superclass	155
15.2.5.3	Instance methods	155
15.2.5.3.1	TrueClass#&	155
15.2.5.3.2	TrueClass# 	155
15.2.5.3.3	TrueClass#^	155
15.2.5.3.4	TrueClass#to_s	155
15.2.6	FalseClass	156
15.2.6.1	General description	156
15.2.6.2	Direct superclass	156
15.2.6.3	Instance methods	156
15.2.6.3.1	FalseClass#&	156
15.2.6.3.2	FalseClass# 	156
15.2.6.3.3	FalseClass#^	156
15.2.6.3.4	FalseClass#to_s	157
15.2.7	Numeric	157
15.2.7.1	General description	157
15.2.7.2	Direct superclass	157
15.2.7.3	Included modules	157
15.2.7.4	Instance methods	157
15.2.7.4.1	Numeric#+@	157
15.2.7.4.2	Numeric#-@	158
15.2.7.4.3	Numeric#abs	158
15.2.7.4.4	Numeric#coerce	158
15.2.8	Integer	159
15.2.8.1	General description	159
15.2.8.2	Direct superclass	160
15.2.8.3	Instance methods	160
15.2.8.3.1	Integer#<=>	160
15.2.8.3.2	Integer#==	160
15.2.8.3.3	Integer#+	161
15.2.8.3.4	Integer#-	161
15.2.8.3.5	Integer#*	162
15.2.8.3.6	Integer#/	162
15.2.8.3.7	Integer#%	163
15.2.8.3.8	Integer#~	164
15.2.8.3.9	Integer#&	164
15.2.8.3.10	Integer# 	164
15.2.8.3.11	Integer#^	165
15.2.8.3.12	Integer#<<	165

15.2.8.3.13	Integer#>>	165
15.2.8.3.14	Integer#ceil	165
15.2.8.3.15	Integer#downto	166
15.2.8.3.16	Integer#eql?	166
15.2.8.3.17	Integer#floor	166
15.2.8.3.18	Integer#hash	167
15.2.8.3.19	Integer#next	167
15.2.8.3.20	Integer#round	167
15.2.8.3.21	Integer#succ	167
15.2.8.3.22	Integer#times	167
15.2.8.3.23	Integer#to_f	168
15.2.8.3.24	Integer#to_i	168
15.2.8.3.25	Integer#to_s	168
15.2.8.3.26	Integer#truncate	169
15.2.8.3.27	Integer#upto	169
15.2.9	Float	169
15.2.9.1	General description	169
15.2.9.2	Direct superclass	170
15.2.9.3	Instance methods	170
15.2.9.3.1	Float#<=>	170
15.2.9.3.2	Float#==	170
15.2.9.3.3	Float#+	171
15.2.9.3.4	Float#-	172
15.2.9.3.5	Float#*	172
15.2.9.3.6	Float#/	173
15.2.9.3.7	Float#%	173
15.2.9.3.8	Float#ceil	174
15.2.9.3.9	Float#finite?	174
15.2.9.3.10	Float#floor	175
15.2.9.3.11	Float#infinite?	175
15.2.9.3.12	Float#round	175
15.2.9.3.13	Float#to_f	175
15.2.9.3.14	Float#to_i	175
15.2.9.3.15	Float#truncate	176
15.2.10	String	176
15.2.10.1	General description	176
15.2.10.2	Direct superclass	176
15.2.10.3	Included modules	176
15.2.10.4	Upper-case and lower-case characters	177
15.2.10.5	Instance methods	177
15.2.10.5.1	String#<=>	177
15.2.10.5.2	String#==	178
15.2.10.5.3	String#=~	178
15.2.10.5.4	String#+	179
15.2.10.5.5	String#*	179
15.2.10.5.6	String#[]	179
15.2.10.5.7	String#capitalize	181
15.2.10.5.8	String#capitalize!	181
15.2.10.5.9	String#chomp	181
15.2.10.5.10	String#chomp!	182
15.2.10.5.11	String#chop	182
15.2.10.5.12	String#chop!	182

15.2.10.5.13	String#downcase	183
15.2.10.5.14	String#downcase!	183
15.2.10.5.15	String#each_line	183
15.2.10.5.16	String#empty?	184
15.2.10.5.17	String#eq?	184
15.2.10.5.18	String#gsub	184
15.2.10.5.19	String#gsub!	186
15.2.10.5.20	String#hash	186
15.2.10.5.21	String#include?	186
15.2.10.5.22	String#index	187
15.2.10.5.23	String#initialize	187
15.2.10.5.24	String#initialize_copy	188
15.2.10.5.25	String#intern	188
15.2.10.5.26	String#length	188
15.2.10.5.27	String#match	188
15.2.10.5.28	String#replace	189
15.2.10.5.29	String#reverse	189
15.2.10.5.30	String#reverse!	189
15.2.10.5.31	String#rindex	189
15.2.10.5.32	String#scan	190
15.2.10.5.33	String#size	191
15.2.10.5.34	String#slice	191
15.2.10.5.35	String#split	191
15.2.10.5.36	String#sub	193
15.2.10.5.37	String#sub!	193
15.2.10.5.38	String#to_f	194
15.2.10.5.39	String#to_i	194
15.2.10.5.40	String#to_s	195
15.2.10.5.41	String#to_sym	195
15.2.10.5.42	String#upcase	195
15.2.10.5.43	String#upcase!	196
15.2.11	Symbol	196
15.2.11.1	General description	196
15.2.11.2	Direct superclass	196
15.2.11.3	Instance methods	196
15.2.11.3.1	Symbol#===	196
15.2.11.3.2	Symbol#id2name	197
15.2.11.3.3	Symbol#to_s	197
15.2.11.3.4	Symbol#to_sym	197
15.2.12	Array	197
15.2.12.1	General description	197
15.2.12.2	Direct superclass	198
15.2.12.3	Included modules	198
15.2.12.4	Singleton methods	198
15.2.12.4.1	Array.[]	198
15.2.12.5	Instance methods	198
15.2.12.5.1	Array#+	198
15.2.12.5.2	Array#*	199
15.2.12.5.3	Array#<<	199
15.2.12.5.4	Array#[]	199
15.2.12.5.5	Array#[]=	200
15.2.12.5.6	Array#clear	201

15.2.12.5.7	Array#collect!	201
15.2.12.5.8	Array#concat	201
15.2.12.5.9	Array#delete_at	202
15.2.12.5.10	Array#each	202
15.2.12.5.11	Array#each_index	202
15.2.12.5.12	Array#empty?	203
15.2.12.5.13	Array#first	203
15.2.12.5.14	Array#index	204
15.2.12.5.15	Array#initialize	204
15.2.12.5.16	Array#initialize_copy	205
15.2.12.5.17	Array#join	205
15.2.12.5.18	Array#last	206
15.2.12.5.19	Array#length	206
15.2.12.5.20	Array#map!	207
15.2.12.5.21	Array#pop	207
15.2.12.5.22	Array#push	207
15.2.12.5.23	Array#replace	207
15.2.12.5.24	Array#reverse	207
15.2.12.5.25	Array#reverse!	208
15.2.12.5.26	Array#rindex	208
15.2.12.5.27	Array#shift	208
15.2.12.5.28	Array#size	209
15.2.12.5.29	Array#slice	209
15.2.12.5.30	Array#unshift	209
15.2.13	Hash	209
15.2.13.1	General description	209
15.2.13.2	Direct superclass	210
15.2.13.3	Included modules	210
15.2.13.4	Instance methods	210
15.2.13.4.1	Hash#==	210
15.2.13.4.2	Hash#[]	211
15.2.13.4.3	Hash#[]=	211
15.2.13.4.4	Hash#clear	212
15.2.13.4.5	Hash#default	212
15.2.13.4.6	Hash#default=	212
15.2.13.4.7	Hash#default_proc	213
15.2.13.4.8	Hash#delete	213
15.2.13.4.9	Hash#each	213
15.2.13.4.10	Hash#each_key	214
15.2.13.4.11	Hash#each_value	214
15.2.13.4.12	Hash#empty?	214
15.2.13.4.13	Hash#has_key?	215
15.2.13.4.14	Hash#has_value?	215
15.2.13.4.15	Hash#include?	215
15.2.13.4.16	Hash#initialize	215
15.2.13.4.17	Hash#initialize_copy	216
15.2.13.4.18	Hash#key?	216
15.2.13.4.19	Hash#keys	216
15.2.13.4.20	Hash#length	217
15.2.13.4.21	Hash#member?	217
15.2.13.4.22	Hash#merge	217
15.2.13.4.23	Hash#replace	218

15.2.13.4.24	Hash#shift	218
15.2.13.4.25	Hash#size	218
15.2.13.4.26	Hash#store	219
15.2.13.4.27	Hash#value?	219
15.2.13.4.28	Hash#values	219
15.2.14	Range	219
15.2.14.1	General description	219
15.2.14.2	Direct superclass	219
15.2.14.3	Included modules	220
15.2.14.4	Instance methods	220
15.2.14.4.1	Range#==	220
15.2.14.4.2	Range#===	220
15.2.14.4.3	Range#begin	221
15.2.14.4.4	Range#each	221
15.2.14.4.5	Range#end	222
15.2.14.4.6	Range#exclude_end?	222
15.2.14.4.7	Range#first	222
15.2.14.4.8	Range#include?	222
15.2.14.4.9	Range#initialize	222
15.2.14.4.10	Range#last	223
15.2.14.4.11	Range#member?	223
15.2.15	Regexp	223
15.2.15.1	General description	223
15.2.15.2	Direct superclass	224
15.2.15.3	Constants	224
15.2.15.4	Patterns	224
15.2.15.5	Matching process	228
15.2.15.6	Singleton methods	229
15.2.15.6.1	Regexp.compile	229
15.2.15.6.2	Regexp.escape	229
15.2.15.6.3	Regexp.last_match	230
15.2.15.6.4	Regexp.quote	231
15.2.15.7	Instance methods	231
15.2.15.7.1	Regexp#==	231
15.2.15.7.2	Regexp#===	231
15.2.15.7.3	Regexp#=~	232
15.2.15.7.4	Regexp#casefold?	232
15.2.15.7.5	Regexp#initialize	233
15.2.15.7.6	Regexp#initialize_copy	233
15.2.15.7.7	Regexp#match	234
15.2.15.7.8	Regexp#source	234
15.2.16	MatchData	234
15.2.16.1	General description	234
15.2.16.2	Direct superclass	235
15.2.16.3	Instance methods	235
15.2.16.3.1	MatchData#[]	235
15.2.16.3.2	MatchData#begin	235
15.2.16.3.3	MatchData#captures	235
15.2.16.3.4	MatchData#end	236
15.2.16.3.5	MatchData#initialize_copy	236
15.2.16.3.6	MatchData#length	237
15.2.16.3.7	MatchData#offset	237

15.2.16.3.8	MatchData#post_match	237
15.2.16.3.9	MatchData#pre_match	237
15.2.16.3.10	MatchData#size	238
15.2.16.3.11	MatchData#string	238
15.2.16.3.12	MatchData#to_a	238
15.2.16.3.13	MatchData#to_s	238
15.2.17	Proc	239
15.2.17.1	General description	239
15.2.17.2	Direct superclass	239
15.2.17.3	Singleton methods	239
15.2.17.3.1	Proc.new	239
15.2.17.4	Instance methods	239
15.2.17.4.1	Proc#[]	239
15.2.17.4.2	Proc#arity	239
15.2.17.4.3	Proc#call	240
15.2.17.4.4	Proc#clone	241
15.2.17.4.5	Proc#dup	241
15.2.18	Struct	242
15.2.18.1	General description	242
15.2.18.2	Direct superclass	242
15.2.18.3	Singleton methods	242
15.2.18.3.1	Struct.new	242
15.2.18.4	Instance methods	244
15.2.18.4.1	Struct#==	244
15.2.18.4.2	Struct#[]	244
15.2.18.4.3	Struct#[]=	245
15.2.18.4.4	Struct#each	246
15.2.18.4.5	Struct#each_pair	246
15.2.18.4.6	Struct#initialize	246
15.2.18.4.7	Struct#initialize_copy	247
15.2.18.4.8	Struct#members	247
15.2.18.4.9	Struct#select	247
15.2.19	Time	248
15.2.19.1	General description	248
15.2.19.2	Direct superclass	248
15.2.19.3	Time computation	248
15.2.19.3.1	Day	248
15.2.19.3.2	Year	249
15.2.19.3.3	Month	249
15.2.19.3.4	Days of month	250
15.2.19.3.5	Hours, Minutes, and Seconds	250
15.2.19.4	Time zone and Local time	251
15.2.19.5	Daylight saving time	251
15.2.19.6	Singleton methods	251
15.2.19.6.1	Time.at	251
15.2.19.6.2	Time.gm	252
15.2.19.6.3	Time.local	254
15.2.19.6.4	Time.mktime	254
15.2.19.6.5	Time.now	254
15.2.19.6.6	Time.utc	254
15.2.19.7	Instance methods	255
15.2.19.7.1	Time#<=>	255

15.2.19.7.2	Time#+	255
15.2.19.7.3	Time#-	256
15.2.19.7.4	Time#asctime	256
15.2.19.7.5	Time#ctime	257
15.2.19.7.6	Time#day	257
15.2.19.7.7	Time#dst?	257
15.2.19.7.8	Time#getgm	258
15.2.19.7.9	Time#getlocal	258
15.2.19.7.10	Time#getutc	258
15.2.19.7.11	Time#gmt?	258
15.2.19.7.12	Time#gmt_offset	258
15.2.19.7.13	Time#gmtime	259
15.2.19.7.14	Time#gmtoff	259
15.2.19.7.15	Time#hour	259
15.2.19.7.16	Time#initialize	259
15.2.19.7.17	Time#initialize_copy	260
15.2.19.7.18	Time#localtime	260
15.2.19.7.19	Time#mday	260
15.2.19.7.20	Time#min	261
15.2.19.7.21	Time#mon	261
15.2.19.7.22	Time#month	261
15.2.19.7.23	Time#sec	261
15.2.19.7.24	Time#to_f	262
15.2.19.7.25	Time#to_i	262
15.2.19.7.26	Time#usec	262
15.2.19.7.27	Time#utc	262
15.2.19.7.28	Time#utc?	263
15.2.19.7.29	Time#utc_offset	263
15.2.19.7.30	Time#wday	263
15.2.19.7.31	Time#yday	263
15.2.19.7.32	Time#year	264
15.2.19.7.33	Time#zone	264
15.2.20	IO	264
15.2.20.1	General description	264
15.2.20.2	Direct superclass	265
15.2.20.3	Included modules	265
15.2.20.4	Singleton methods	265
15.2.20.4.1	IO.open	265
15.2.20.5	Instance methods	266
15.2.20.5.1	IO#close	266
15.2.20.5.2	IO#closed?	266
15.2.20.5.3	IO#each	267
15.2.20.5.4	IO#each_byte	267
15.2.20.5.5	IO#each_line	268
15.2.20.5.6	IO#eof?	268
15.2.20.5.7	IO#flush	268
15.2.20.5.8	IO#getc	268
15.2.20.5.9	IO#gets	269
15.2.20.5.10	IO#initialize_copy	269
15.2.20.5.11	IO#print	269
15.2.20.5.12	IO#putc	270
15.2.20.5.13	IO#puts	270

15.2.20.5.14	IO#read	271
15.2.20.5.15	IO#readchar	271
15.2.20.5.16	IO#readline	272
15.2.20.5.17	IO#readlines	272
15.2.20.5.18	IO#sync	273
15.2.20.5.19	IO#sync=	273
15.2.20.5.20	IO#write	273
15.2.21	File	274
15.2.21.1	General description	274
15.2.21.2	Direct superclass	274
15.2.21.3	Singleton methods	274
15.2.21.3.1	File.exist?	274
15.2.21.4	Instance methods	274
15.2.21.4.1	File#initialize	274
15.2.21.4.2	File#path	275
15.2.22	Exception	275
15.2.22.1	General description	275
15.2.22.2	Direct superclass	275
15.2.22.3	Singleton methods	275
15.2.22.3.1	Exception.exception	275
15.2.22.4	Instance methods	276
15.2.22.4.1	Exception#exception	276
15.2.22.4.2	Exception#initialize	276
15.2.22.4.3	Exception#message	276
15.2.22.4.4	Exception#to_s	277
15.2.23	StandardError	277
15.2.23.1	General description	277
15.2.23.2	Direct superclass	277
15.2.24	ArgumentError	277
15.2.24.1	General description	277
15.2.24.2	Direct superclass	277
15.2.25	LocalJumpError	277
15.2.25.1	Direct superclass	278
15.2.25.2	Instance methods	278
15.2.25.2.1	LocalJumpError#exit_value	278
15.2.25.2.2	LocalJumpError#reason	278
15.2.26	RangeError	278
15.2.26.1	General description	278
15.2.26.2	Direct superclass	278
15.2.27	RegexpError	278
15.2.27.1	General description	278
15.2.27.2	Direct superclass	278
15.2.28	RuntimeError	278
15.2.28.1	General description	278
15.2.28.2	Direct superclass	279
15.2.29	TypeError	279
15.2.29.1	General description	279
15.2.29.2	Direct superclass	279
15.2.30	ZeroDivisionError	279
15.2.30.1	General description	279
15.2.30.2	Direct superclass	279
15.2.31	NameError	279

15.2.31.1	Direct superclass	279
15.2.31.2	Instance methods	279
15.2.31.2.1	NameError#initialize	279
15.2.31.2.2	NameError#name	280
15.2.32	NoMethodError	280
15.2.32.1	Direct superclass	280
15.2.32.2	Instance methods	280
15.2.32.2.1	NoMethodError#args	280
15.2.32.2.2	NoMethodError#initialize	280
15.2.33	IndexError	281
15.2.33.1	General description	281
15.2.33.2	Direct superclass	281
15.2.34	IOError	281
15.2.34.1	General description	281
15.2.34.2	Direct superclass	281
15.2.35	EOFError	281
15.2.35.1	General description	281
15.2.35.2	Direct superclass	281
15.2.36	SystemCallError	281
15.2.36.1	General description	281
15.2.36.2	Direct superclass	281
15.2.37	ScriptError	281
15.2.37.1	General description	281
15.2.37.2	Direct superclass	282
15.2.38	SyntaxError	282
15.2.38.1	General description	282
15.2.38.2	Direct superclass	282
15.2.39	LoadError	282
15.2.39.1	General description	282
15.2.39.2	Direct superclass	282
15.3	Built-in modules	282
15.3.1	Kernel	282
15.3.1.1	General description	282
15.3.1.2	Singleton methods	282
15.3.1.2.1	Kernel.`	282
15.3.1.2.2	Kernel.block_given?	283
15.3.1.2.3	Kernel.eval	283
15.3.1.2.4	Kernel.global_variables	283
15.3.1.2.5	Kernel.iterator?	284
15.3.1.2.6	Kernel.lambda	284
15.3.1.2.7	Kernel.local_variables	285
15.3.1.2.8	Kernel.loop	285
15.3.1.2.9	Kernel.p	285
15.3.1.2.10	Kernel.print	286
15.3.1.2.11	Kernel.puts	286
15.3.1.2.12	Kernel.raise	286
15.3.1.2.13	Kernel.require	287
15.3.1.3	Instance methods	288
15.3.1.3.1	Kernel#==	288
15.3.1.3.2	Kernel#===	288
15.3.1.3.3	Kernel#`	289
15.3.1.3.4	Kernel#_id__	289

15.3.1.3.5	Kernel#_send_	289
15.3.1.3.6	Kernel#block_given?	289
15.3.1.3.7	Kernel#class	289
15.3.1.3.8	Kernel#clone	290
15.3.1.3.9	Kernel#dup	290
15.3.1.3.10	Kernel#eq?	291
15.3.1.3.11	Kernel#equal?	291
15.3.1.3.12	Kernel#eval	291
15.3.1.3.13	Kernel#extend	292
15.3.1.3.14	Kernel#global_variables	292
15.3.1.3.15	Kernel#hash	292
15.3.1.3.16	Kernel#initialize_copy	293
15.3.1.3.17	Kernel#inspect	293
15.3.1.3.18	Kernel#instance_eval	293
15.3.1.3.19	Kernel#instance_of?	294
15.3.1.3.20	Kernel#instance_variable_defined?	294
15.3.1.3.21	Kernel#instance_variable_get	294
15.3.1.3.22	Kernel#instance_variable_set	295
15.3.1.3.23	Kernel#instance_variables	295
15.3.1.3.24	Kernel#is_a?	295
15.3.1.3.25	Kernel#iterator?	296
15.3.1.3.26	Kernel#kind_of?	296
15.3.1.3.27	Kernel#lambda	296
15.3.1.3.28	Kernel#local_variables	296
15.3.1.3.29	Kernel#loop	297
15.3.1.3.30	Kernel#method_missing	297
15.3.1.3.31	Kernel#methods	297
15.3.1.3.32	Kernel#nil?	297
15.3.1.3.33	Kernel#object_id	298
15.3.1.3.34	Kernel#p	298
15.3.1.3.35	Kernel#print	298
15.3.1.3.36	Kernel#private_methods	298
15.3.1.3.37	Kernel#protected_methods	299
15.3.1.3.38	Kernel#public_methods	299
15.3.1.3.39	Kernel#puts	300
15.3.1.3.40	Kernel#raise	300
15.3.1.3.41	Kernel#remove_instance_variable	300
15.3.1.3.42	Kernel#require	301
15.3.1.3.43	Kernel#respond_to?	301
15.3.1.3.44	Kernel#send	301
15.3.1.3.45	Kernel#singleton_methods	302
15.3.1.3.46	Kernel#to_s	302
15.3.2	Enumerable	302
15.3.2.1	General description	302
15.3.2.2	Instance methods	303
15.3.2.2.1	Enumerable#all?	303
15.3.2.2.2	Enumerable#any?	303
15.3.2.2.3	Enumerable#collect	303
15.3.2.2.4	Enumerable#detect	304
15.3.2.2.5	Enumerable#each_with_index	304
15.3.2.2.6	Enumerable#entries	305
15.3.2.2.7	Enumerable#find	305

15.3.2.2.8	Enumerable#find_all	305
15.3.2.2.9	Enumerable#grep	305
15.3.2.2.10	Enumerable#include?	306
15.3.2.2.11	Enumerable#inject	306
15.3.2.2.12	Enumerable#map	307
15.3.2.2.13	Enumerable#max	307
15.3.2.2.14	Enumerable#member?	308
15.3.2.2.15	Enumerable#min	308
15.3.2.2.16	Enumerable#partition	309
15.3.2.2.17	Enumerable#reject	309
15.3.2.2.18	Enumerable#select	310
15.3.2.2.19	Enumerable#sort	310
15.3.2.2.20	Enumerable#to_a	311
15.3.3	Comparable	311
15.3.3.1	General description	311
15.3.3.2	Instance methods	311
15.3.3.2.1	Comparable#<	311
15.3.3.2.2	Comparable#<=	311
15.3.3.2.3	Comparable#>	312
15.3.3.2.4	Comparable#>=	312
15.3.3.2.5	Comparable#==	312
15.3.3.2.6	Comparable#between?	313

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 30170 was prepared by the Japanese Industrial Standards Committee (as JIS X3017) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by the national bodies of ISO and IEC.

Introduction

This International Standard is based upon a submission from the Japanese National Standards Body called JIS X3017 “Programming Language Ruby”, registered and published in 2011.

Ruby is an object-oriented scripting language designed to be developer-friendly, productive and intuitive. There is a continuing growth of interest in Ruby around the world, especially among web application developers, while its use spans from web applications to private tools.

As the Ruby language grows and spreads, there is no globally agreed upon documented Ruby specification. In order to avoid confusion as a result of diversification of usage and incompatibility among implementations, the Japan Industry Standard is proposed as an international standard.

There are multiple Ruby implementations available. Many of them are distributed as open source software. The implementation called “Matz Ruby Implementation (MRI)” has been treated as a reference implementation insofar as virtually all implementers check compatibility of their implementations by comparing them to MRI. Therefore, this specification of Ruby is codified as a strict subset of MRI.

This International Standard specifies only core language features and core libraries which are stable enough and common between MRI versions and compatible between existing implementations. There are two versions of MRI currently distributed and maintained: MRI 1.8, which has been available since 2003 and MRI 1.9, which was released in 2010. Currently, MRI 1.8 is more widely used than MRI 1.9. Use of MRI 1.9 will likely spread in the next several years. To avoid future divergence, features which are planned or prospected to be changed are excluded from this version of the specification, or it is clearly stated that the behavior of the features are not specified. For example, this specification does not specify the handling of character type in detail because it is planned to be changed in MRI 1.9 for full support of ISO/IEC 10646. The full support of ISO/IEC 10646 is going to be standardized in a future version of this standard. The library defined in this specification is limited to that which is commonly used or necessary to write simple programs.

This International Standard introduces special notations and a concept called “Execution context” in order to specify flexible syntax and dynamic semantics of the Ruby language as simple as possible.

Information technology — Programming languages — Ruby

1 Scope

This International Standard specifies the syntax and semantics of the computer programming language Ruby, and the requirements for conforming Ruby processors, strictly conforming Ruby programs, and conforming Ruby programs.

This International Standard does not specify,

- the limit of size or complexity of a program text which a conforming processor evaluates,
- the minimal requirements of a data processing system that is capable of supporting a conforming processor,
- the method for activating the execution of programs on a data processing system, and
- the method for reporting syntactic and runtime errors.

NOTE Execution of a Ruby program is to evaluate the *program* (see 10) by a Ruby processor.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.
- IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*.
- ISO/IEC 2382-1:1993, *Information technology – Vocabulary – Part 1: Fundamental terms*.

3 Conformance

A strictly conforming Ruby program shall

- use only those features of the language specified in this International Standard, and

©ISO/IEC 2012 — All rights reserved

- not produce output dependent on any unspecified or implementation-defined behavior.

A conforming Ruby processor shall

- evaluate any strictly conforming programs as specified in this International Standard.

A conforming Ruby processor may

- evaluate a strictly conforming program in a different way from the one described in this International Standard, if it does not change the behavior of the program; however, if the program redefines any method or constant of a built-in class or module (see Clause 15), the behavior of the program may be different from the one described in this International Standard (see NOTE 2), and
- support syntax not described in this International Standard, and evaluate any programs which use features not specified in this International Standard.

A conforming processor shall be accompanied by a document that defines all implementation-defined behavior and all extensions not specified in this International Standard.

A conforming Ruby program is one that a conforming Ruby processor can evaluate.

A conforming program shall be accompanied by a document that defines expected behavior of each implementation-defined behavior and extensions used in the program and not specified in this International Standard, if these behaviors affect the output of the program.

NOTE 1 The description of expected behaviors can be replaced by the name of a conforming processor which supports the expected behaviors.

NOTE 2 For example, a conforming processor may omit an invocation of a method of a built-in class or module for optimization purpose, and do the same calculation as the method instead. In this case, even if a program redefines the method, the behavior of the program might not change because the redefined method might not actually be invoked.

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply. Other terms are defined where they appear in ***bold slant face*** or on the left side of a syntax rule.

4.1

block

procedure which is passed to a method invocation

4.2

class

object which defines the behavior of a set of other objects, called its instances

NOTE The behavior is a set of methods which can be invoked on an instance.

4.3

class variable

variable whose value is shared by all the instances of a class

4.4**constant**

variable which is defined in a class or a module, and is accessible both inside and outside the class or module

NOTE The value of a constant is ordinarily expected to remain unchanged during the execution of a program, but this International Standard does not enforce this expectation.

4.5**exception**

object which represents an exceptional event

4.6**global variable**

variable which is accessible everywhere in a program

4.7**implementation-defined**

behavior that possibly differs between implementations, but is defined for every implementation

4.8**instance method**

method which can be invoked on all the instances of a class

4.9**instance variable**

variable that exists in a set of variable bindings which every instance of an object has

4.10**local variable**

variable which is accessible only in a certain scope introduced by a program construct such as a method definition, a block, a class definition, a module definition, a singleton class definition, or the top level of a program

4.11**method**

procedure which, when invoked on an object, performs a set of computations on the object

4.12**method visibility**

attribute of a method which determines the conditions under which a method invocation is allowed

4.13**module**

object which provides features to be included into a class or another module

4.14**object**

computational entity which has a state and a behavior

NOTE The behavior of an object is a set of methods which can be invoked on the object.

4.15

singleton class

object which can modify the behavior of its associated object

NOTE A singleton class is ordinarily associated with a single object. However, a conforming processor may associate a singleton class with multiple objects as described in 13.4.1.

4.16

singleton method

instance method of a singleton class

4.17

unspecified

behavior that possibly differs between implementations, and is not necessarily defined for every implementation

4.18

variable

computational entity that refers to an object, which is called the value of the variable

4.19

variable binding

association between a variable and an object which is referred to by the variable

5 Notational conventions

5.1 General description

In this clause, the following terms are used:

a) sequence of A

A “sequence of A ”, whose length is n , indicates a sequence whose n elements $A_1, A_2, \dots, A_n$ ($n \geq 0$) are of the same kind A as follows: $A_1 A_2 \dots A_n$. A sequence whose length is 0 is called an empty sequence.

b) sequence of A separated by B

A “sequence of A separated by B ”, whose length is $n + 1$, indicates a sequence whose $n + 1$ elements $A_0, A_1, A_2, \dots, A_n$ ($n \geq 0$) are of the same kind A and whose adjacent elements are separated by $B_1, B_2, \dots, B_n$ of the same kind B as follows: $A_0 B_1 A_1 B_2 \dots B_n A_n$.

5.2 Syntax

5.2.1 General description

In this International Standard, the syntax of the Ruby language is specified by syntactic rules which are a series of productions (see 5.2.2), and constraints of syntax written in a natural language. Syntactic rules are given in some subclauses, and are entitled “Syntax”.

5.2.2 Productions

Each production is of the following form, where X is a nonterminal symbol [see 5.2.4 b)], and Y is a sequence of syntactic term sequences (see 5.2.3) separated by a vertical line ($|$), and where whitespace and newlines are used for the sake of readability:

$$X :: Y$$

A production defines a set of sequences of characters represented by a nonterminal symbol X as a union of sets represented by syntactic term sequences in Y . The production $X :: Y$ is therefore called “the production of X ” or “the X production”. X is called the left hand side of the production, and Y is called the right hand side of the production. The nonterminal symbol X is said to directly refer to nonterminal symbols which appear in Y . A relationship that a nonterminal symbol A refers to a nonterminal symbol B is defined recursively as follows:

- If A directly refers to B , then A refers to B ;
- If A refers to a nonterminal symbol C , and if C refers to B , then A refers to B .

NOTE 1 A syntactic term represents a set of sequences of characters as described in 5.2.3.

In a constraint written in a natural language in a syntactic rule, or in a semantic rule (see 5.3), “ X ”, where X is a syntactic term sequence, indicates an element of the set of sequences of characters represented by the syntactic term sequence X . Especially in the case that X is a nonterminal symbol Y , “ Y ” indicates an element of the set of sequences of characters represented by the nonterminal symbol, and “the nonterminal symbol Y ” indicates the nonterminal symbol itself. A sequence of characters represented by “ Y ” is also called “of the form Y ”.

When a nonterminal symbol Y directly refers to a nonterminal symbol Z , “ Z of Y ” indicates a part of a sequence of characters represented by Y , which is represented by such Z .

NOTE 2 For example, a sequence x of characters represented by X whose production is “ $X :: Y Z$ ” consists of a sequence y of characters represented by Y and a sequence z of characters represented by Z , and $x = yz$. In this case, “ Z of X ” indicates z .

“ Z in Y ” indicates a part of a sequence of characters represented by Y , which is represented by Z referred to by the nonterminal symbol Y .

“Each Z of Y ” indicates a sequence of characters defined by the following a) to c):

- a) This notation is used when Z appears in a primary term P (see 5.2.4), and the right hand side of the production of Y contains zero or more repetitions of P [see 5.2.4 f)] (i.e., P^*).
- b) Let Y_n ($n \geq 0$) be the right hand side of the production of Y , where P^* is replaced with a sequence of P s whose length is n . For any sequence y of characters represented by Y , there exists i such that a sequence of characters represented by Y_i is y .
- c) “Each Z of Y ” indicates a part of y represented by Z which appears repeatedly in Y_i .

If the number of Z referred to by Y in productions in a subclause is only one, “ Z ” is used as a short form of “ Z of Y ” or “ Z in Y ”.

The nonterminal symbols *input-element* (see 8.1), *program* (see 10.1), and *pattern* (see 15.2.15.4) are called start symbols.

EXAMPLE 1 The following example is the *input-element* production. This production means an *input-element* is any of a *line-terminator*, *whitespace*, *comment*, *end-of-program-marker*, or *token*.

```

input-element ::
    line-terminator
    | whitespace
    | comment
    | end-of-program-marker
    | token

```

EXAMPLE 2 *Y* and *Z* are defined as follows:

```

Y ::
    Z ( # Z ) *

Z ::
    a | b | ( Y )

```

In this case, for each following sequence of characters represented by *Y*, “each *Z* of *Y*” indicates each underlined part.

```

a
a#b
a#b#a
(a#b)
a#(a#b)#a

```

5.2.3 Syntactic term sequences

A syntactic term sequence is a sequence of syntactic terms (see 5.2.4). A syntactic term sequence *S*, which is a sequence $T_1 T_2 \dots T_n$ ($n \geq 1$), where T_i ($1 \leq i \leq n$) is a syntactic term, represents a set of all sequences of characters of the form $t_1 t_2 \dots t_n$, where t_i is any element of the set of sequences of characters represented by T_i . However, if T_i is a special term, the meaning of t_i is defined in 5.2.4 d).

Line-terminators (see 8.3), *whitespace* (see 8.4), and *comments* (see 8.5) are used to separate *tokens* (see 8.7), and are ordinarily ignored. *Line-terminators*, *whitespace*, and *comments* are therefore omitted in the right hand side of productions except in Clause 8 and 15.2.15.4. That is, in the right hand side of productions, the following syntactic term is omitted before and after terms.

```

( line-terminator | whitespace | comment ) *

```

However, a location where a *line-terminator* or *whitespace* shall not occur, or a location where a *line-terminator* or *whitespace* shall occur is indicated by special terms: a forbidden term [see 5.2.4 d) 2)] or a mandatory term [see 5.2.4 d) 3)], respectively.

EXAMPLE The following example represents a sequence of characters: **alias** [a terminal symbol, see 5.2.4 a)], *new-name*, and *aliased-name*, in this order. However, there might be any number of *line-terminators*, *whitespace* characters, and/or *comments* between these elements.

alias *new-name aliased-name*

5.2.4 Syntactic terms

A syntactic term represents a sequence of characters, or a constraint to a sequence of characters represented by a syntactic term sequence which includes the syntactic term. A syntactic term is any of the following a) to h). In particular, syntactic terms a) to c) are called primary terms.

NOTE Note that a syntactic term is specified recursively.

a) terminal symbol

A terminal symbol is shown in **typewriter face**. A terminal symbol represents a set whose only element is a sequence of characters shown in **typewriter face**.

EXAMPLE 1 **+** represents a sequence of one character “+”. **def** represents a sequence of three characters “def”.

b) nonterminal symbol

A nonterminal symbol is shown in *italic face*. A nonterminal symbol represents a set of sequences of characters defined by the production of the nonterminal symbol.

EXAMPLE 2 A *binary-digit* defined by the following production represents “0” or “1”.

binary-digit ::
0 | 1

c) grouping term

A grouping term is a sequence of syntactic term sequences separated by a vertical line (|) and enclosed by parentheses [()]. A grouping term represents a union of sets of sequences of characters represented by syntactic term sequences in the grouping term.

EXAMPLE 3 The following example represents an *alpha-numeric-character* or a *line-terminator*.

(*alpha-numeric-character* | *line-terminator*)

d) special term

A special term is a text enclosed by square brackets ([]). A special term is any of the following:

1) negative lookahead

The notation of a negative lookahead is [lookahead $\notin$ S], where S is a sequence of terminal symbols separated by a comma (,) enclosed by curly brackets ({ }). A negative lookahead represents a constraint that any sequence of characters in S shall not occur just after the negative lookahead.

EXAMPLE 4 The following example means that an *argument-without-parentheses* shall not begin with “{”:

argument-without-parentheses ::
[lookahead $\notin$ { { } }] *argument-list*

2) forbidden term

The notation of a forbidden term is [no T here], where T is a primary term. A forbidden term represents a constraint that no T shall occur there.

EXAMPLE 5 The following example means no *line-terminator* shall occur there.

[no *line-terminator* here]

3) mandatory term

The notation of a mandatory term is [T here], where T is a primary term. A mandatory term represents a constraint that one or more T s shall occur there.

EXAMPLE 6 The following example means one or more *line-terminators* shall occur there.

[*line-terminator* here]

4) other special term

The notation of an other special term is [U], where U is a text which does not match any of d) 1) to d) 3). This special term represents a set of sequences of characters represented by U , or a constraint represented by U to a sequence of characters represented by a syntactic term sequence which includes this special term.

EXAMPLE 7 The following example means that a *source-character* is any character specified in ISO/IEC 646:1991 IRV:

source-character ::
[any character in ISO/IEC 646:1991 IRV]

EXAMPLE 8 The following example means **=begin** shall occur at the beginning of a line.

[beginning of a line] **=begin**

e) optional term

An optional term is a primary term postfixed with a superscripted question mark (?).

An optional term represents a superset of the set represented by the primary term, which has an empty sequence of characters as the only additional element.

EXAMPLE 9 The following example means that the *block* is optional.

block[?]

f) zero or more repetitions

A primary term postfixed with a superscripted asterisk (*) indicates zero or more repetitions of the primary term. Zero or more repetitions represent a set of sequences of characters whose elements are all sequences of any zero or more elements of the set represented by the primary term.

EXAMPLE 10 The following example means a sequence of characters which consists of zero or more *elsif-clauses*.

elsif-clause^{*}

g) one or more repetitions

A primary term postfixed with a superscripted plus sign (+) indicates one or more repetitions of the primary term. One or more repetitions represent a set of sequences of characters whose elements are all sequences of any one or more elements of the set represented by the primary term.

EXAMPLE 11 The following example means a sequence of characters which consists of one or more *when-clauses*.

when-clause⁺

h) exception term

An exception term is a sequences of a primary term P_1 , the phrase **but not**, and another primary term P_2 . An exception term represents a set of sequences of characters whose elements are all elements of P_1 excluding all elements of P_2 .

EXAMPLE 12 The following example represents a *source-character* but not a *single-quoted-string-meta-character*.

source-character **but not** *single-quoted-string-meta-character*

5.2.5 Conceptual names

A nonterminal symbol (except start symbols) which is not referred to by any start symbol is called a conceptual name. In the production of a conceptual name, ::= is used instead of :: to distinguish conceptual names from other nonterminal symbols.

NOTE 1 In this International Standard, some semantically related nonterminal symbols are syntactically away from each other. Conceptual names are used to define names which organize such nonterminal symbols [e.g., *assignment* (see 11.4.2)]. Conceptual names are also used to define nonterminal symbols used only in semantic rules [e.g., *binary-operator* (see 11.4.4)].

EXAMPLE 1 The following example defines the conceptual name *assignment*, which can be used to mention either *assignment-expression* or *assignment-statement*.

```
assignment ::=
    assignment-expression
    | assignment-statement
```

5.3 Semantics

For syntactic rules, corresponding semantic rules are given in some subclauses, and are entitled “Semantics”. In this International Standard, the behaviors of programs are specified by processes evaluating the programs. The evaluation of a program construct, which is a sequence of characters represented by a nonterminal symbol, usually results in a value, which is called the (resulting) value of the program construct. Semantic rules specify the ways of evaluating program constructs specified in corresponding syntactic rules, and the resulting values of the evaluations.

The start of evaluation steps of a program construct described in semantic rules is called the start of the evaluation of the program construct. The time when there is no evaluation step to be taken for the program construct is called the end of the evaluation of the program construct. If the evaluation of a program construct has started, and if the evaluation has not ended, the program construct is said to be under evaluation.

If there is no semantic rule corresponding to a nonterminal symbol *X*, and if the right hand side of the production of *X* is a sequence of other nonterminal symbols separated by a vertical line (|), the semantic rule of *X* is defined by the semantic rules of other nonterminal symbols referred to by *X*.

EXAMPLE 1 A *variable* (see 11.5.4) has the following production, and has no description of semantic rules.

```

variable ::
    constant-identifier
    | global-variable-identifier
    | class-variable-identifier
    | instance-variable-identifier
    | local-variable-identifier

```

In this case, the semantic rule of *variable* is defined by the semantic rule of *constant-identifier*, *global-variable-identifier*, *class-variable-identifier*, *instance-variable-identifier*, or *local-variable-identifier*.

If there is more than one same nonterminal symbol in the right hand side of a production, the nonterminal symbols have a subscript to distinguish them in semantic rules (e.g., *operator-expression*₁), if necessary.

The semantic rule of a conceptual name describes the semantic rule of program constructs which are elements of the set of sequences of characters represented by the conceptual name. In semantic rules, “*X*”, where *X* is a conceptual name, indicates a program construct which is an element of the set of sequences of characters represented by the nonterminal symbol *X*.

EXAMPLE 2 *logical-AND-expression* (see 11.2.3) has the following production.

```

logical-AND-expression ::=
    keyword-AND-expression
    | operator-AND-expression

```

Since *logical-AND-expression* is a conceptual name, a sequence of characters represented by a *keyword-AND-expression* or *operator-AND-expression* never be recognized as a *logical-AND-expression* under parsing process of a program text. However, *keyword-AND-expression* and *operator-AND-expression* have similar semantic rules and they are described as the semantic rule of *logical-AND-expression*. In semantic rules, “*logical-AND-expression*” indicates a program construct represented by a *keyword-AND-expression* or *operator-AND-expression*.

5.4 Attributes of execution contexts

The names of the attributes of execution contexts (see 7.1) are enclosed in double square brackets ([[]]).

EXAMPLE [[self]] is one of the attributes of execution contexts.

6 Fundamental concepts

6.1 Objects

An object has a state and a behavior. The state of an object is represented by the attributes of the object. Every object has a set of bindings of instance variables (see 6.2.2) as one of its attributes. Besides the set of bindings of instance variables, an object can have some other attributes, depending on the class of the object. The behavior of an object is defined by a set of methods (see 6.3) which can be invoked on that object. A method is defined in a class, a singleton class, or a module (see 6.5).

Every value directly manipulated by a program is an object. All of the following values are objects:

- A value which is referred to by a variable (see 6.2);
- A value which is passed to a method as an argument;
- A value which is returned by a method;
- A value which is returned as the result of evaluating an *expression* (see Clause 11), a *statement* (see Clause 12), a *compound-statement* (see 10.2), or a *program* (see 10.1).

Other values are not objects, unless explicitly specified as objects.

NOTE Primitive values such as integers are also objects. For example, an integer literal (see 8.7.6.2) evaluates to an object.

6.2 Variables

6.2.1 General description

A variable is denoted by a name, and refers to an object, which is called the value of the variable. A variable itself is not an object. While a variable can refer to only one object at a time, an object can be referred to by more than one variable at a time.

A variable is said to be **bound** to an object if the variable refers to the object. This association of a variable with an object is called a **variable binding**. When a variable with name *N* is bound to an object *O*, *N* is called the name of the binding, and *O* is called the value of the binding.

There are five kinds of variables:

- instance variables (see 6.2.2), whose names are prefixed with single “@” (e.g., “@var”);
- constants (see 6.5.2), whose names begin with an uppercase character (e.g., “Const”);
- class variables (see 6.5.2), whose names are prefixed with “@@” (e.g., “@@var”);
- local variables (see 9.2), whose names begin with a lowercase character or “_” (e.g., “var”);
- global variables (see 9.3), whose names are prefixed with “\$” (e.g., “\$var”).

Any variable can be bound to any kind of object.

EXAMPLE In the following program, first, the local variable `x` refers to an integer, then it refers to a string, finally it refers to an array.

```
x = 123
x = "abc"
x = [1, 2, 3]
```

6.2.2 Instance variables

An object has a set of variable bindings. A variable whose binding is in this set is an instance variable of that object. This set of bindings of instance variables represents a state of that object.

An instance variable of an object is not directly accessible outside the object. An instance variable is ordinarily accessed through methods called accessors outside the object. In this sense, a set of bindings of instance variables is encapsulated in an object.

EXAMPLE In the following program, the value of the instance variable `@value` of an instance of the class `ValueHolder` is initialized by the method `initialize` (see 15.2.3.3.1), and is accessed through the accessor method `value`, and printed by the method `puts` of the module `Kernel` (see 15.3.1.2.11). Text after `#` is a comment (see 8.5).

```
class ValueHolder
  def initialize(value)
    @value = value
  end

  def value
    return @value
  end
end

vh = ValueHolder.new(10)  # initialize(10) is invoked.
puts vh.value
```

6.3 Methods

A method is a procedure which, when invoked on an object, performs a set of computations on the object. A method itself is not an object. The behavior of an object is defined by a set of methods which can be invoked on that object. A method has one or more (when aliased) names associated with it. An association between a name and a method is called a **method binding**. When a name *N* is bound to a method *M*, *N* is called the name of the binding, and *M* is called the value of the binding. A name bound to a method is called the **method name**. A method can be invoked on an object by specifying one of its names. The object on which the method is invoked is called the **receiver** of the method invocation.

EXAMPLE In a method invocation `obj.method(arg1, arg2)`, `obj` is called the receiver, and `method` is called the method name. See 11.3 for method invocation expressions.

Methods are described further in 13.3.

6.4 Blocks

A block is a procedure which is passed to a method invocation. The block passed to a method invocation is called zero or more times in the method invocation.

A block itself is not an object. However, a block can be represented by an object which is an instance of the class `Proc` (see 15.2.17).

EXAMPLE 1 In the following program, for each element of an array, the block “{ |i| puts i }” is called by the method `each` of the class `Array` (see 15.2.12.5.10).

```
a = [1, 2, 3]
a.each { |i| puts i }
```

EXAMPLE 2 In the following program, an instance of the class `Proc` which represents the block “{ puts "abc" }” is created, and is called by the method `call` of the class `Proc` (see 15.2.17.4.3).

```
x = Proc.new { puts "abc" }
x.call
```

Blocks are described further in 11.3.3.

6.5 Classes, singleton classes, and modules

6.5.1 General description

Behaviors of objects are defined by classes, singleton classes, and modules. A class defines methods shared by objects of the same class. A singleton class is associated to an object, and can modify the behavior of that object. A module defines, and provides methods to be included into classes and other modules. Classes, singleton classes, and modules are themselves objects, which are dynamically created and modified at run-time.

6.5.2 Classes

A class is itself an object, and creates other objects. The created objects are called **direct instances** of the class (see 13.2.4).

A class defines a set of methods which, unless overridden (see 13.3.1), can be invoked on all the instances of the class. These methods are instance methods of the class.

A class is itself an object, and created by evaluation of a program construct such as a *class-definition* (see 13.2.2). A class has two sets of variable bindings besides a set of bindings of instance variables. The one is a set of bindings of constants. The other is a set of bindings of class variables, which represents the state shared by all the instances of the class.

The constants, class variables, singleton methods and instance methods of a class are called the **features** of the class.

EXAMPLE 1 The class `Array` (see 15.2.12) is itself an object, and can be the receiver of a method invocation. An invocation of the method `new` on the class `Array` creates an object called a direct instance of the class `Array`.

EXAMPLE 2 In the following program, the instance method `push` of the class `Array` (see 15.2.12.5.22) is invoked on an instance of the class `Array`.

```
a = Array.new
a.push(1, 2, 3)  # The value of a is changed to [1, 2, 3].
```

EXAMPLE 3 In the following program, the class `X` is defined by a class definition. The class variable `@@a` is shared by instances of the class `X`.

```
class X
  @@a = "abc"

  def print_a
    puts @@a
  end

  def set_a(value)
    @@a = value
  end
end
x1 = X.new
x1.print_a      # prints abc
x2 = X.new
x2.set_a("def")
x2.print_a      # prints def
x1.print_a      # prints def
```

Classes are described further in 13.2.

6.5.3 Singleton classes

Every object, including classes, can be associated with at most one singleton class. The singleton class defines methods which can be invoked on that object. Those methods are singleton methods of the object.

- If the object is not a class, the singleton methods of the object can be invoked only on that object.
- If the object is a class, singleton methods of the class can be invoked only on that class and its subclasses (see 6.5.4).

A singleton class is created, and associated with an object by a singleton class definition (see 13.4.2) or a singleton method definition (see 13.4.3).

EXAMPLE 1 In the following program, the singleton class of `x` is created by a singleton class definition. The method `show` is called a singleton method of `x`, and can be invoked only on `x`.

```
x = "abc"
y = "def"

# The definition of the singleton class of x
class << x
  def show
    puts self  # prints the receiver
  end
end

x.show  # prints abc
y.show  # raises an exception
```

EXAMPLE 2 In the following program, the same singleton method `show` as EXAMPLE 1 is defined by a singleton method definition. The singleton class of `x` is created implicitly by the singleton method definition.

```
x = "abc"

# The definition of a singleton method of x
def x.show
  puts self  # prints the receiver
end

x.show
```

EXAMPLE 3 In the following program, the singleton method `a` of the class `X` is defined by a singleton method definition.

```
class X
  # The definition of a singleton method of the class X
  def X.a
    puts "The method a is invoked."
  end
end
X.a
```

NOTE Singleton methods of a class is similar to so-called class methods in other object-oriented languages because they can be invoked on that class.

Singleton classes are described further in 13.4.

6.5.4 Inheritance

A class has at most one single class as its **direct superclass**. If a class *A* has a class *B* as its direct superclass, *A* is called a **direct subclass** of *B*.

All the classes in a program, including built-in classes, form a rooted tree called a **class inheritance tree**, where the parent of a class is its direct superclass, and the children of a class are all its direct subclasses. There is only one class which does not have a superclass. It is the root of the tree. All the ancestors of a class in the tree are called **superclasses** of the class. All the descendants of a class in the tree are called **subclasses** of the class.

A class inherits constants, class variables, singleton methods, and instance methods from its superclasses, if any (see 13.2.3). If an object *C* is a direct instance of a class *D*, *C* is called an instance of *D* and all its superclasses.

EXAMPLE The following program defines three classes: the class `X`, the class `Y`, and the class `Z`.

```
class X
end

class Y < X
end

class Z < Y
end
```

The class `X` is called the direct superclass of the class `Y`, and the class `Y` is called a direct subclass of the class `X`. The class `Y` inherits features from the class `X`. The class `X` is called a superclass of the class `Z`, and

the class Z is called a subclass of the class X. The class Z inherits features from the class X and the class Y. A direct instance of the class Z is called an instance of the class X, the class Y, and the class Z.

6.5.5 Modules

Multiple inheritance of classes is not permitted. That is, a class can have only one direct superclass. However, features can be appended to a class from multiple modules by using module inclusions.

A module is an object which has the same structure as a class except that it cannot create an instance of itself and cannot be inherited. As with classes, a module has a set of bindings of constants, a set of bindings of class variables, and a set of instance methods. Instance methods, constants, and class variables defined in a module can be used by other classes, modules, and singleton classes by including the module into them.

While a class can have only one direct superclass, a class, a module, or a singleton class can include multiple modules. Instance methods defined in a module can be invoked on an instance of a class which includes the module. A module is created by a module definition (see 13.1.2).

EXAMPLE The following example is not a strictly conforming Ruby program, because a class cannot have multiple direct superclasses.

```
class Stream
end

class ReadStream < Stream
  def read(n)
    # reads n bytes from a stream
  end
end

class WriteStream < Stream
  def write(str)
    # writes str to a stream
  end
end

class ReadWriteStream < ReadStream, WriteStream
end
```

Instead, a class can include multiple modules. The following example uses module inclusion instead of multiple inheritance.

```
class Stream
end

module Readable
  def read(n); end
end

module Writable
  def write(str); end
end

class ReadStream < Stream
  include Readable
end
```

```

class WriteStream < Stream
  include Writable
end

class ReadWriteStream
  include Readable
  include Writable
end

```

Modules are described further in 13.1.

6.6 Boolean values

An object is classified into either a *trueish object* or a *falseish object*.

Only **false** and **nil** are falseish objects. **false** is the only instance of the class **FalseClass** (see 15.2.6), to which a *false-expression* evaluates (see 11.5.4.8.3). **nil** is the only instance of the class **NilClass** (see 15.2.4), to which a *nil-expression* evaluates (see 11.5.4.8.2).

Objects other than **false** and **nil** are classified into trueish objects. **true** is the only instance of the class **TrueClass** (see 15.2.5), to which a *true-expression* evaluates (see 11.5.4.8.3).

7 Execution contexts

7.1 General description

An *execution context* is a set of attributes which affects evaluation of a program.

An execution context is not a part of the Ruby language. It is defined in this International Standard only for the description of the semantics of a program. A conforming processor shall evaluate a program producing the same result as if the processor acted within an execution context in the manner described in this International Standard.

An execution context consists of a set of attributes as described below. Each attribute of an execution context except `[[global-variable-bindings]]` forms a stack. Attributes of an execution context are changed when a program construct is evaluated.

The following are the attributes of an execution context:

`[[self]]`: A stack of objects. The object at the top of the stack is called the *current self*, to which a *self-expression* evaluates (see 11.5.4.8.4).

`[[class-module-list]]`: A stack of lists of classes, modules, or singleton classes. The class or module at the head of the list which is on the top of the stack is called the *current class or module*.

`[[default-method-visibility]]`: A stack of visibilities of methods, each of which is one of the **public**, **private**, and **protected** visibilities. The top of the stack is called the *current visibility*.

`[[local-variable-bindings]]`: A stack of sets of bindings of local variables. The element at the

top of the stack is called the **current set of local variable bindings**. A set of bindings is pushed onto the stack on every entry into a local variable scope (see 9.2), and the top element is removed from the stack on every exit from the scope. The scope with which an element in the stack is associated is called the **scope of the set of local variable bindings**.

[[invoked-method-name]] : A stack of names by which methods are invoked.

[[defined-method-name]] : A stack of names with which the invoked methods are defined.

NOTE The top elements of [[invoked-method-name]] and [[defined-method-name]] are usually the same. However, they can be different if an invoked method has an alias name.

[[block]] : A stack of blocks passed to method invocations. An element of the stack may be block-not-given. **block-not-given** is the special value which indicates that no block is passed to a method invocation.

[[global-variable-bindings]] : A set of bindings of global variables.

7.2 The initial state

Immediately prior to execution of a program, the attributes of the execution context is initialized as follows:

- a) Set [[global-variable-bindings]] to a newly created empty set. A conforming processor may add bindings of any global variables to [[global-variable-bindings]].
- b) Create built-in classes and modules as described in Clause 15.
- c) Create an empty stack for each attribute of the execution context except [[global-variable-bindings]].
- d) Create a direct instance of the class **Object** and push it onto [[self]].
- e) Create a list containing only element, the class **Object**, and push the list onto [[class-module-list]].
- f) Push the private visibility onto [[default-method-visibility]].
- g) Push block-not-given onto [[block]].

8 Lexical structure

8.1 General description

Syntax

```

input-element ::
    line-terminator
    | whitespace
  
```

	<i>comment</i>
	<i>end-of-program-marker</i>
	<i>token</i>

The program text of a program is first converted into a sequence of *input-elements*, which are either *line-terminators*, *whitespace*, *comments*, *end-of-program-markers*, or *tokens*. When several prefixes of the input under the converting process have matching productions, the production that matches the longest prefix is selected.

8.2 Program text

Syntax

source-character ::
 [any character in ISO/IEC 646:1991 IRV]

A program is represented as a **program text**. A program text is a sequence of *source-characters*. A *source-character* is a character in ISO/IEC 646:1991 IRV (the International Reference Version). The support for any other character sets and encodings is unspecified.

NOTE A conforming processor is required to support ISO/IEC 646:1991 IRV. A conforming processor may support other character sets and encodings. However, ways to handle characters other than those in ISO/IEC 646:1991 IRV and ways to handle coded character sets where characters have different codes from ISO/IEC 646:1991 are not specified in this International Standard.

Terminal symbols are sequences of those characters in ISO/IEC 646:1991 IRV. Control characters and the character SPACE in ISO/IEC 646:1991 IRV are represented by two digits in hexadecimal notation prefixed by “0x”, where the first and the second digits respectively represent x and y of the notations of the form x/y specified in ISO/IEC 646, 5.1.

EXAMPLE “0x0a” represents the character LF, whose bit combination specified in ISO/IEC 646 is 0/10.

8.3 Line terminators

Syntax

line-terminator ::
 0x0d[?] 0x0a

Except in Clause 8 and 15.2.15.4, *line-terminators* are omitted from productions as described in 5.2.3. However, a location where a *line-terminator* shall not occur, or a location where a *line-terminator* shall occur is indicated by special terms: a forbidden term [see 5.2.4 d) 2)] or a mandatory term [see 5.2.4 d) 3)], respectively.

EXAMPLE *statements* are separated by *separators* (see 10.2). The syntax of the *separators* is as follows:

```
separator ::
    ;
    | [ line-terminator here ]
```

The source

```
x = 1 + 2
puts x
```

is therefore separated into the two *statements* “x = 1 + 2” and “puts x” by a *line-terminator*.

The source

```
x =
  1 + 2
```

is parsed as the single *statement* “x = 1 + 2” because “x =” is not a *statement*. However, the source

```
x
  = 1 + 2
```

is not a strictly conforming Ruby program because a *line-terminator* shall not occur before = in a *single-variable-assignment-expression*, and “= 1 + 2” is not a *statement*. The fact that a *line-terminator* shall not occur before = is indicated in the syntax of the *single-variable-assignment-expression* as follows (see 11.4.2.2.2):

```
single-variable-assignment-expression ::
    variable [no line-terminator here] = operator-expression
```

8.4 Whitespace

Syntax

```
whitespace ::
    0x09 | 0x0b | 0x0c | 0x0d | 0x20 | line-terminator-escape-sequence
```

```
line-terminator-escape-sequence ::
    \ line-terminator
```

Except in Clause 8 and 15.2.15.4, *whitespace* is omitted from productions as described in 5.2.3. However, a location where *whitespace* shall not occur, or a location where *whitespace* shall occur is indicated by special terms: a forbidden term [see 5.2.4 d) 2)] or a mandatory term [see 5.2.4 d) 3)] , respectively.

8.5 Comments

Syntax

```

comment ::
    single-line-comment
    | multi-line-comment

single-line-comment ::
    # comment-content?

comment-content ::
    line-content

line-content ::
    ( source-character+ ) but not ( source-character* line-terminator source-character* )

multi-line-comment ::
    multi-line-comment-begin-line multi-line-comment-line?
    multi-line-comment-end-line

multi-line-comment-begin-line ::
    [ beginning of a line ] =begin rest-of-begin-end-line? line-terminator

multi-line-comment-end-line ::
    [ beginning of a line ] =end rest-of-begin-end-line?
    ( line-terminator | [ end of a program ] )

rest-of-begin-end-line ::
    whitespace+ comment-content

multi-line-comment-line ::
    comment-line but not multi-line-comment-end-line

comment-line ::
    comment-content line-terminator

```

The notation “[beginning of a line]” indicates the beginning of a program or the position immediately after a *line-terminator*.

A *comment* is either a *single-line-comment* or a *multi-line-comment*. Except in Clause 8 and 15.2.15.4, *comments* are omitted from productions as described in 5.2.3.

A *single-line-comment* begins with “#” and continues to the end of the line. A *line-terminator* at the end of the line is not considered to be a part of the comment. A *single-line-comment* can

contain any characters except *line-terminators*.

A *multi-line-comment* begins with a line beginning with **=begin**, and continues until and including a line that begins with **=end**. Unlike *single-line-comments*, a *line-terminator* of a *multi-line-comment-end-line*, if any, is considered to be part of the comment.

NOTE A *line-content* is a sequence of *source-characters*. However, *line-terminators* are not permitted within a *line-content* as specified in the *line-content* production.

8.6 End-of-program markers

Syntax

```
end-of-program-marker ::
    [ beginning of a line ] __END__ ( line-terminator | [ end of a program ] )
```

An *end-of-program-marker* indicates the end of a program. Any source characters after an *end-of-program-marker* are not treated as a program text.

NOTE __END__ is not a *keyword*, and can be a *local-variable-identifier*.

8.7 Tokens

8.7.1 General description

Syntax

```
token ::
    keyword
    | identifier
    | punctuator
    | operator
    | literal
```

punctuators and *operators* are symbols that have independent syntactic and semantic significance. The semantics of *punctuators* and *operators* are described in the clauses from Clause 9 to Clause 14.

8.7.2 Keywords

Syntax

```
keyword ::
    __LINE__ | __ENCODING__ | __FILE__ | BEGIN | END | alias | and | begin
    | break | case | class | def | defined? | do | else | elsif | end
    | ensure | for | false | if | in | module | next | nil | not | or | redo
```

	rescue		retry		return		self		super		then		true		undef		unless
	until		when		while		yield										

Keywords are case-sensitive.

NOTE __LINE__, __ENCODING__, __FILE__, BEGIN, and END are reserved for future use.

8.7.3 Identifiers

Syntax

identifier ::

	<i>local-variable-identifier</i>
	<i>global-variable-identifier</i>
	<i>class-variable-identifier</i>
	<i>instance-variable-identifier</i>
	<i>constant-identifier</i>
	<i>method-only-identifier</i>
	<i>assignment-like-method-identifier</i>

local-variable-identifier ::

(lowercase-character | _) identifier-character* **but not** keyword

global-variable-identifier ::

\$ identifier-start-character identifier-character*

class-variable-identifier ::

@@ identifier-start-character identifier-character*

instance-variable-identifier ::

@ identifier-start-character identifier-character*

constant-identifier ::

uppercase-character identifier-character* **but not** keyword

method-only-identifier ::

(constant-identifier | local-variable-identifier) (! | ?)

assignment-like-method-identifier ::

(constant-identifier | local-variable-identifier) =

identifier-character ::

	<i>lowercase-character</i>
	<i>uppercase-character</i>
	<i>decimal-digit</i>

| -

identifier-start-character ::

lowercase-character

| uppercase-character

| -

uppercase-character ::

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

lowercase-character ::

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

decimal-digit ::

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

8.7.4 Punctuators

Syntax

punctuator ::

[|] | (|) | { | } | :: | , | ; | .. | ... | ? | : | =>

8.7.5 Operators

Syntax

operator ::

! | != | !~ | && | ||

| operator-method-name

| =

| assignment-operator

operator-method-name ::

^ | & | | | <=> | == | === | =~ | > | >= | < | <= | << | >> | + | -

| * | / | % | ** | ~ | +@ | -@ | [] | []= | ' ,

assignment-operator ::

assignment-operator-name =

assignment-operator-name ::

&& | || | ^ | & | | | << | >> | + | - | * | / | % | **

8.7.6 Literals

8.7.6.1 General description

literal ::
 numeric-literal
 | *string-literal*
 | *array-literal*
 | *regular-expression-literal*
 | *symbol*

8.7.6.2 Numeric literals

Syntax

numeric-literal ::
 signed-number
 | *unsigned-number*

signed-number ::
 (+ | -) *unsigned-number*

unsigned-number ::
 integer-literal
 | *float-literal*

integer-literal ::
 decimal-integer-literal
 | *binary-integer-literal*
 | *octal-integer-literal*
 | *hexadecimal-integer-literal*

decimal-integer-literal ::
 unprefixed-decimal-integer-literal
 | *prefixed-decimal-integer-literal*

unprefixed-decimal-integer-literal ::
 0
 | *decimal-digit-except-zero* (_? *decimal-digit*)*

prefixed-decimal-integer-literal ::
 0 (d | D) *digit-decimal-part*

digit-decimal-part ::
decimal-digit (*_*? *decimal-digit*)*

binary-integer-literal ::
 0 (*b* | *B*) *binary-digit* (*_*? *binary-digit*)*

octal-integer-literal ::
 0 (*_* | *o* | *O*)? *octal-digit* (*_*? *octal-digit*)*

hexadecimal-integer-literal ::
 0 (*x* | *X*) *hexadecimal-digit* (*_*? *hexadecimal-digit*)*

float-literal ::
float-literal-without-exponent
 | *float-literal-with-exponent*

float-literal-without-exponent ::
unprefixed-decimal-integer-literal . *digit-decimal-part*

float-literal-with-exponent ::
significand-part *exponent-part*

significand-part ::
float-literal-without-exponent
 | *unprefixed-decimal-integer-literal*

exponent-part ::
 (*e* | *E*) (*+* | *-*)? *digit-decimal-part*

decimal-digit-except-zero ::
 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary-digit ::
 0 | 1

octal-digit ::
 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal-digit ::
decimal-digit | *a* | *b* | *c* | *d* | *e* | *f* | *A* | *B* | *C* | *D* | *E* | *F*

If the previous token of a *signed-number* is a *local-variable-identifier*, *constant-identifier*, or *method-only-identifier*, at least one *whitespace* character or *line-terminator* shall be present be-

tween the *local-variable-identifier*, *constant-identifier*, or *method-only-identifier*, and the *signed-number*.

EXAMPLE -123 in the following program is a *signed-number* because there is *whitespace* between **x** and -123.

```
x -123
```

In the above program, the method **x** is invoked with the value of -123 as the argument.

However, -123 in the following program is separated into the two tokens - and 123 because there is no *whitespace* between **x** and -123.

```
x-123
```

In the above program, the method - is invoked on the value of **x** with the value of 123 as the argument.

Semantics

A *numeric-literal* evaluates to either an instance of the class **Integer** or a direct instance of the class **Float**.

NOTE Subclasses of the class **Integer** may be defined as described in 15.2.8.1.

An *unsigned-number* of the form *integer-literal* evaluates to an instance of the class **Integer** whose value is the value of one of the syntactic term sequences in the *integer-literal* production.

An *unsigned-number* of the form *float-literal* evaluates to a direct instance of the class **Float** whose value is the value of one of the syntactic term sequences in the *float-literal* production.

A *signed-number* which begins with “+” evaluates to the resulting instance of the *unsigned-number*. A *signed-number* which begins with “-” evaluates to an instance of the class **Integer** or a direct instance of the class **Float** whose value is the negated value of the resulting instance of the *unsigned-number*.

The value of an *integer-literal*, a *decimal-integer-literal*, a *float-literal*, or a *significand-part* is the value of one of the syntactic term sequences in their production.

The value of a *unprefixed-decimal-integer-literal* is 0 if it is of the form “0”, otherwise the value of the *unprefixed-decimal-integer-literal* is the value of a sequence of characters, which consist of a *decimal-digit-except-zero* followed by a sequence of *decimal-digits*, ignoring interleaving “_”s, computed using base 10.

The value of a *prefixed-decimal-integer-literal* is the value of the *digit-decimal-part*.

The value of a *digit-decimal-part* is the value of the sequence of *decimal-digits*, ignoring interleaving “_”s, computed using base 10.

The value of a *binary-integer-literal* is the value of the sequence of *binary-digits*, ignoring interleaving “_”s, computed using base 2.

The value of an *octal-integer-literal* is the value of the sequence of *octal-digits*, ignoring interleaving “_”s, computed using base 8.

The value of a *hexadecimal-integer-literal* is the value of the sequence of *hexadecimal-digits*, ignoring interleaving “_”s, computed using base 16. The values of *hexadecimal-digits* **a** (or **A**) through **f** (or **F**) are 10 through 15, respectively.

The value of a *float-literal-without-exponent* is the value of the *unprefixed-decimal-integer-literal* plus the value of the *digit-decimal-part* times 10^{-n} where n is the number of *decimal-digits* of the *digit-decimal-part*.

The value of a *float-literal-with-exponent* is the value of the *significand-part* times 10^n where n is the value of the *exponent-part*.

The value of an *exponent-part* is the negative value of the *digit-decimal-part* if “-” occurs, otherwise, it is the value of the *digit-decimal-part*.

See 15.2.8.1 for the range of the value of an instance of the class **Integer**.

See 15.2.9.1 for the precision of the value of an instance of the class **Float**.

8.7.6.3 String literals

8.7.6.3.1 General description

Syntax

```
string-literal ::
    single-quoted-string
  | double-quoted-string
  | quoted-non-expanded-literal-string
  | quoted-expanded-literal-string
  | here-document
  | external-command-execution
```

Semantics

A *string-literal* evaluates to a direct instance of the class **String**.

NOTE Some of the *string-literals* represents a value of an expression (see 8.7.6.3.3), as well as the literal characters of the program text.

8.7.6.3.2 Single quoted strings

Syntax

```
single-quoted-string ::
    ' single-quoted-string-character* '

single-quoted-string-character ::
    single-quoted-string-non-escaped-character
  | single-quoted-escape-sequence
```

single-quoted-escape-sequence ::
 single-escape-character-sequence
 | *single-quoted-string-non-escaped-character-sequence*

single-escape-character-sequence ::
 \ *single-quoted-string-meta-character*

single-quoted-string-non-escaped-character-sequence ::
 \ *single-quoted-string-non-escaped-character*

single-quoted-string-meta-character ::
 ' | \

single-quoted-string-non-escaped-character ::
 source-character **but not** *single-quoted-string-meta-character*

Semantics

A *single-quoted-string* consists of zero or more characters enclosed by single quotes. The sequence of *single-quoted-string-characters* within the pair of single quotes represents the content of a string as it occurs in a program text literally, except for *single-escape-character-sequences*. The sequence “\\” represents “\”. The sequence “\’” represents “’”.

NOTE Unlike a *single-escape-character-sequence*, a *single-quoted-string-non-escaped-character-sequence* represents two characters as it occurs in a program text literally. For example, ‘\a’ represents two characters \ and a.

EXAMPLE ‘\a\’\’ represents a string whose content is “\a\’” without the double quotes.

8.7.6.3.3 Double quoted strings

Syntax

double-quoted-string ::
 " *double-quoted-string-character** "

double-quoted-string-character ::
 source-character **but not** (" | # | \)
 | # [lookahead ∉ { \$, @, { }]
 | *double-escape-sequence*
 | *interpolated-character-sequence*

double-escape-sequence ::
 simple-escape-sequence
 | *non-escaped-sequence*
 | *line-terminator-escape-sequence*
 | *octal-escape-sequence*