Programming Languages — Ruby

IPA Ruby Standardization WG Draft

August 25, 2010

Contents

Page

Introduction

This document specifies the Ruby programming language.

Ruby is an object-oriented scripting language, which has been developed by Yukihiro Matsumoto and his contributors since 1993, and has several implementations distributed as open source software. Ruby has both enough features as an object-oriented language and simplicity as a scripting language, and advanced applications can be implemented with brief code in Ruby. These characteristics of Ruby enables high productivity of program development.

Ruby is thus used for many applications and network systems across the world at the present day, and has multiple implementations. Therefore, a standard specification which underlies compatibility among implementations has been demanded.

The biggest goal of Ruby is developer friendliness, and productivity of application development and intuitive description of program behaviors take precedence over brevity of the language specification itself and ease of implementation. This document is therefore complex as a language specification in order to specify the syntax and semantics of Ruby without ambiguity.

Information technology — Programming Languages — Ruby

1 Scope

This document 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 document does not specify

- the limit of size or complexity of a program text which is acceptable to a conforming processor,
- 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.*

  NOTE Corresponding JIS: JIS X 0201:1997 7-bit and 8-bit coded character sets for information interchange (MOD)

- IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems.*

3 Conformance

A strictly conforming Ruby program shall

- use only those features of the language specified in this document, and
- not produce output dependent on any unspecified or implementation-defined behavior.

A conforming Ruby processor shall

- accept any strictly conforming programs and behave as specified in this document.

A conforming Ruby processor may

- evaluate a strictly conforming program in a different way from the one described in this document, if it does not change the behavior of the program; however, if the program redefines any method of a built-in class or module (see Clause 15), the behavior of the program may be different from the one described in this document, and

  NOTE 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.

- support syntax not described in this document, and accept any programs which use features not specified in this document.

A conforming Ruby program is one that is acceptable to a conforming Ruby processor.

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
A procedure which is passed to a method invocation.

4.2
class
An 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
A variable whose value is shared by all the instances of a class.

4.4
constant
A 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 document does not force it.

4.5
exception
An object which represents an exceptional event.

4.6
global variable
A variable which is accessible everywhere in a program.

4.7
implementation-defined
Possibly differing between implementations, but defined for every implementation.

4.8
instance method
A method which can be invoked on all the instances of a class.

4.9
instance variable
A variable that exists in a set of variable bindings which every object has.

4.10
local variable
A 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 toplevel of a program.

4.11
method
A procedure which, when invoked on an object, performs a set of computations on the object.

4.12
method visibility
An attribute of a method which determines the conditions under which a method invocation is allowed.

4.13
module
An object which provides features to be included into a class or another module.

4.14
object
A computational entity which has states 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
An 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
An instance method of a singleton class.

4.17
unspecified
Possibly differing between implementations, and not necessarily defined for any particular implementation.

4.18
variable
A computational entity that refers to an object, which is called the value of the variable.

4.19
variable binding
An 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, $\ldots$, A_n $(n \geq 0)$ are of the same kind A. 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, $\ldots$, A_n $(n \geq 0)$ are of the same kind A and whose adjacent elements are separated by B_1, B_2, $\ldots$, B_n of the same kind B as follows: A_0, B_1, A_1, B_2, $\ldots$, B_n, A_n.

5.2 Syntax

5.2.1 General description

In this document, 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 elmement 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 elmement 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" consist 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 Ps 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 $T_1\ T_2 \ldots T_n$ ($n \geq 1$), where T_i ($1 \leq i \leq n$) is a syntactic term, represents a set of all sequences of characters of the form $t_1\ t_2 \ldots t_n$, 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-terminator*s (see 8.3), *whitespace* (see 8.4), and *comment*s (see 8.5) are used to separate *token*s (see 8.7), and are ordinarily ignored. *Line-terminator*s, *whitespace*, and *comment*s 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-terminator*s, *whitespace* characters, and/or *comment*s 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 Ts shall occur there.

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

  [*line-terminator* here]

---

4) other special term
The notation of an other speical 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-clause*s.

---

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-clause*s.

---

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 exmaple 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 document, 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 document, 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., $operator\text{-}expression_1$), 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 11.2.4) 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 states and a behavior. An object has a set of bindings of instance variables (see 6.2.2) as one of its states. Besides the set of bindings of instance variables, an object can have some attributes as its states, 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. For example, 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);
- constants (see 6.5.2);
- class variables (see 6.5.2);
- local variables (see 9.2);
- global variables (see 9.3).

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 sence, 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 15.2.3.3.3), 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 instance***s 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-oriendted 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 ***superclass***es of the class. All the descendants of a class in the tree are called ***subclass***es 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 constants, a set 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
```

```ruby
class ReadWriteStream < ReadStream, WriteStream
end
```

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

```ruby
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 document 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 document.

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*** visibility. 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.

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*
  | *comment*
  | *end-of-program-marker*
  | *token*

The program text of a program is first converted into a sequence of *input-element*s, which are either *line-terminator*s, *whitespace*, *comment*s, *end-of-program-marker*s, or *token*s. 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-character*s. 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.

Terminal symbols are sequences of those characters in ISO/IEC 646:1991 IRV. Control characters 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-terminator*s 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 *statement*s are separated by *separator*s (see 10.2). The syntax of the *separator*s is as follows:

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

The source

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

is therefore separated into the two *statement*s "`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, *comment*s 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-terminator*s.

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-comment*s, 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-character*s. However, *line-terminator*s 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*

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 ::
local-variable-identifier
| *global-variable-identifier*
| *class-variable-identifier*
| *instance-variable-identifier*
| *constant-identifier*
| *method-only-identifier*
| *assignment-like-method-identifier*

local-variable-identifier ::
(*lowercase-character* | `_`) *identifier-character**

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**

method-only-identifier ::
(*constant-identifier* | *local-variable-identifier*) (`!` | `?`)

```
assignment-like-method-identifier ::
    ( constant-identifier | local-variable-identifier ) =
```

```
identifier-character ::
    lowercase-character
  | uppercase-character
  | decimal-digit
  | _
```

```
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 between 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-digit*s, 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-digit*s, ignoring interleaving "_"s, computed using base 10.

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

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

The value of a *hexadecimal-integer-literal* is the value of the sequence of *hexadecimal-digit*s, ignoring interleaving “_”s, computed using base 16. The values of *hexadecimal-digit*s `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-digit*s 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), not only 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-character*s 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-sequence*s. 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'\”.

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 $\notin$ { `$`, `@`, `{` }]

| *double-escape-sequence*
| *interpolated-character-sequence*

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

simple-escape-sequence ::
`\` *double-escaped-character*

double-escaped-character ::
`\` | `n` | `t` | `r` | `f` | `v` | `a` | `e` | `b` | `s`

non-escaped-sequence ::
`\` *non-escaped-double-quoted-string-character*

non-escaped-double-quoted-string-character ::
source-character **but not** (*alpha-numeric-character* | *line-terminator*)

octal-escape-sequence ::
`\` *octal-digit* *octal-digit*$^?$ *octal-digit*$^?$

hexadecimal-escape-sequence ::
`\` `x` *hexadecimal-digit* *hexadecimal-digit*$^?$

control-escape-sequence ::
`\` (`C-` | `c`) *control-escaped-character*

control-escaped-character ::
double-escape-sequence
| `?`
| *source-character* **but not** (`\` | `?`)

interpolated-character-sequence ::
`#` *global-variable-identifier*
| `#` *class-variable-identifier*
| `#` *instance-variable-identifier*
| `#` `{` *compound-statement* `}`

alpha-numeric-character ::
uppercase-character
| *lowercase-character*

| *decimal-digit*

Semantics

A *double-quoted-string* consists of zero or more characters enclosed by double quotes. The sequence of *double-quoted-string-character*s within the pair of double quotes represents the content of a string.

Except for a *double-escape-sequence* and an *interpolated-character-sequence*, a *double-quoted-string-character* represents a character as it occurs in a program text.

A *simple-escape-sequence* represents a character as shown in Table 1.

Table 1 – Simple escape sequences

Escape sequence	Character code
`\\`	0x5c
`\n`	0x0a
`\t`	0x09
`\r`	0x0d
`\f`	0x0c
`\v`	0x0b
`\a`	0x07
`\e`	0x1b
`\b`	0x08
`\s`	0x20

An *octal-escape-sequence* represents a character the code of which is the value of the sequence of *octal-digit*s computed using base 8.

A *hexadecimal-escape-sequence* represents a character the code of which is the value of the sequence of *hexadecimal-digit*s computed using base 16.

A *non-escaped-sequence* represents an implementation-defined character.

A *line-terminator-escape-sequence* is used to break the content of a string into separate lines in a program text without inserting a *line-terminator* into the string. A *line-terminator-escape-sequence* does not count as a character of the string.

A *control-escape-sequence* represents a character the code of which is computed by performing a bitwise AND operation between 0x9f and the code of the character represented by the *control-escaped-character*, except when the *control-escaped-character* is `?`, in which case, the *control-escape-sequence* represents a character the code of which is 127.

An *interpolated-character-sequence* is a part of a *string-literal* which is dynamically evaluated when the *string-literal* in which it is embedded is evaluated. The value of a *string-literal* which contains *interpolated-character-sequence*s is a direct instance of the class `String` the content of which is made from the *string-literal* where each occurrence of *interpolated-character-sequence* is replaced by the content of an instance of the class `String` which is the dynamically evaluated value of the *interpolated-character-sequence*.

An *interpolated-character-sequence* is evaluated as follows:

a) If it is of the form `#` *global-variable-identifier*, evaluate the *global-variable-identifier* (see 11.5.4.4). Let V be the resulting value.

b) If it is of the form `#` *class-variable-identifier*, evaluate the *class-variable-identifier* (see 11.5.4.5). Let V be the resulting value.

c) If it is of the form `#` *instance-variable-identifier*, evaluate the *instance-variable-identifier* (see 11.5.4.6). Let V be the resulting value.

d) If it is of the form `#{` *compound-statement* `}`, evaluate the *compound-statement* (see 10.2). Let V be the resulting value.

e) If V is an instance of the class `String`, the value of *interpolated-character-sequence* is V.

f) Otherwise, invoke the method `to_s` on V with no arguments. Let S be the resulting value.

g) If S is an instance of the class `String`, the value of *interpolated-character-sequence* is S.

h) Otherwise, the behavior is unspecified.

EXAMPLE `"1 + 1 = #{1 + 1}"` represents a string whose content is "1 + 1 = 2".

8.7.6.3.4 Quoted non-expanded literal strings

Syntax

---

quoted-non-expanded-literal-string ::
  `%q` *non-expanded-delimited-string*

non-expanded-delimited-string ::
  literal-beginning-delimiter non-expanded-literal-string literal-ending-delimiter*

non-expanded-literal-string ::
  non-expanded-literal-character
  | *non-expanded-delimited-string*

non-expanded-literal-character ::
  non-escaped-literal-character
  | *non-expanded-literal-escape-sequence*

non-escaped-literal-character ::
  source-character **but not** *quoted-literal-escape-character*

non-expanded-literal-escape-sequence ::
    non-expanded-literal-escape-character-sequence
  | *non-escaped-non-expanded-literal-character-sequence*

non-expanded-literal-escape-character-sequence ::
    \ *non-expanded-literal-escaped-character*

non-expanded-literal-escaped-character ::
    literal-beginning-delimiter
  | *literal-ending-delimiter*
  | \

quoted-literal-escape-character ::
    non-expanded-literal-escaped-character

non-escaped-non-expanded-literal-character-sequence ::
    \ *non-escaped-non-expanded-literal-character*

non-escaped-non-expanded-literal-character ::
    source-character **but not** *non-expanded-literal-escaped-character*

literal-beginning-delimiter ::
    source-character **but not** *alpha-numeric-character*

literal-ending-delimiter ::
    source-character **but not** *alpha-numeric-character*

All *literal-beginning-delimiter*s in a *non-expanded-delimited-string* shall be the same character. All *literal-ending-delimiter*s in a *non-expanded-delimited-string* shall be the same character.

If a *literal-beginning-delimiter* is one of the characters on the left in Table 2, the corresponding *literal-beginning-delimiter* shall be the corresponding character on the right in Table 2. Otherwise, the *literal-ending-delimiter* shall be the same character as the *literal-beginning-delimiter*.

Table 2 – Matching *literal-beginning-delimiter* and *literal-ending-delimiter*

literal-beginning-delimiter	*literal-ending-delimiter*
`{`	`}`
`(`	`)`
`[`	`]`
`<`	`>`

The *non-expanded-delimited-string* of a *non-expanded-literal-string* in a *quoted-non-expanded-literal-string* applies only when its *literal-beginning-delimiter* is one of the characters on the left in Table 2.

NOTE 1 A *quoted-non-expanded-literal-string* can have nested brackets in regard to the *literal-beginning-delimiter* and the corresponding *literal-ending-delimiter* (e.g., `%q[[abc] [def]]`). Different brackets than these two brackets and any escaped brackets are ignored in this nesting. For example, `%q[\[abc\)def(]` represents a direct instance of the class `String` whose content is "[abc\)def(". In this case, only `[`, `]`, and `\` can be *non-expanded-literal-escaped-character*s because the *literal-beginning-delimiter* and the corresponding *literal-beginning-delimiter* are `[` and `]` respectively.

Semantics

The value of a *quoted-non-expanded-literal-string* represents a string whose content is the concatenation of the contents represented by the *non-expanded-literal-string*s of the *non-expanded-delimited-string* of the *quoted-non-expanded-literal-string*.

The value of a *non-expanded-literal-string* represents the content of a string as it occurs in a program text literally, except for *non-expanded-literal-escape-character-sequence*s.

NOTE 1 The content of a string represented by a *non-expanded-literal-string* contains the *literal-beginning-delimiter* and the *literal-ending-delimiter* of a *non-expanded-delimited-string* in the *non-expanded-literal-string*. For example, `%q((abc))` represents a direct instance of the class `String` whose content is "(abc)".

The value of a *non-expanded-literal-escape-character-sequence* represents a character as follows. The sequence "\\" represents "\"; the sequence "\" *literal-beginning-delimiter*, a *literal-beginning-delimiter*; the sequence "\" *literal-ending-delimiter*, a *literal-ending-delimiter*.

8.7.6.3.5 Quoted expanded literal strings

Syntax

---

quoted-expanded-literal-string ::
  `%` `Q`$^?$ *expanded-delimited-string*

expanded-delimited-string ::
  literal-beginning-delimiter *expanded-literal-string** *literal-ending-delimiter*

expanded-literal-string ::
  expanded-literal-character
  | *expanded-delimited-string*

expanded-literal-character ::
  non-escaped-literal-character **but not** `#`
  | `#` [lookahead $\notin$ { `$`, `@`, `{` }]
  | *double-escape-sequence*
  | *interpolated-character-sequence*

All *literal-beginning-delimiter*s in a *expanded-delimited-string* shall be the same character. All *literal-ending-delimiter*s in a *expanded-delimited-string* shall be the same character.

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

The *expanded-delimited-string* of a *expanded-literal-string* in a *quoted-expanded-literal-string* applies only when its *literal-beginning-delimiter* is one of the characters on the left in 8.7.6.3.4 Table 2.

Semantics

The value of a *quoted-expanded-literal-string* represents a string whose content is the concatenation of the contents represented by the *expanded-literal-string*s of the *expanded-delimited-string* of the *quoted-expanded-literal-string*.

A character in an *expanded-literal-string* other than a *double-escape-sequence* or an *interpolated-character-sequence* represents a character as it occurs in a program text. A *double-escape-sequence* and an *interpolated-character-sequence* represent characters as described in 8.7.6.3.3.

NOTE The content of a string represented by a *expanded-literal-string* contains the *literal-beginning-delimiter* and the *literal-ending-delimiter* of a *expanded-delimited-string* in the *expanded-literal-string*. For example, "`%Q((#{1 + 2}))`" represents a string whose content is "(3)".

8.7.6.3.6 Here documents

Syntax

---

here-document ::
  heredoc-start-line *heredoc-body* *heredoc-end-line*

heredoc-start-line ::
  heredoc-signifier *rest-of-line*

heredoc-signifier ::
  `<<` *heredoc-delimiter-specifier*

rest-of-line ::
  line-content$^?$ *line-terminator*

heredoc-body ::
  heredoc-body-line*

heredoc-body-line ::
  comment-line **but not** *heredoc-end-line*

heredoc-delimiter-specifier ::
  `-`$^?$ *heredoc-delimiter*

heredoc-delimiter ::
  non-quoted-delimiter
  | *single-quoted-delimiter*
  | *double-quoted-delimiter*
  | *command-quoted-delimiter*

non-quoted-delimiter ::
  non-quoted-delimiter-identifier

non-quoted-delimiter-identifier ::
  identifier-character*

single-quoted-delimiter ::
  ' *single-quoted-delimiter-identifier* '

single-quoted-delimiter-identifier ::
  ((*source-character* *source-character*$^{?}$) **but not** (' | *line-terminator*))*

double-quoted-delimiter ::
  " *double-quoted-delimiter-identifier* "

double-quoted-delimiter-identifier ::
  ((*source-character* *source-character*$^{?}$) **but not** (" | *line-terminator*))*

command-quoted-delimiter ::
  ` *command-quoted-delimiter-identifier* `

command-quoted-delimiter-identifier ::
  ((*source-character* *source-character*$^{?}$) **but not** (` | *line-terminator*))*

heredoc-end-line ::
  indented-heredoc-end-line
  | *non-indented-heredoc-end-line*

indented-heredoc-end-line ::
  [beginning of a line] *whitespace** *heredoc-delimiter-identifier* *line-terminator*

non-indented-heredoc-end-line ::
  [beginning of a line] *heredoc-delimiter-identifier* *line-terminator*

heredoc-delimiter-identifier ::
  non-quoted-delimiter-identifier
  | *single-quoted-delimiter-identifier*
  | *double-quoted-delimiter-identifier*
  | *command-quoted-delimiter-identifier*

The *heredoc-signifier*, the *heredoc-body*, and the *heredoc-end-line* in a *here-document* are treated as a unit and considered to be a single token occurring at the place where the *heredoc-signifier* occurs. The first character of the *rest-of-line* becomes the head of the input after the *here-document* has been processed.

The form of a *heredoc-end-line* depends on the presence or absence of the beginning "-" of the *heredoc-delimiter-specifier*.

If the *heredoc-delimiter-specifier* begins with "-", a line of the form *indented-heredoc-end-line* is treated as the *heredoc-end-line*, otherwise, a line of the form *non-indented-heredoc-end-line* is treated as the *heredoc-end-line*. In both forms, the *heredoc-delimiter-identifier* shall be the same sequence of characters as it occurs in the corresponding part of *heredoc-delimiter*.

If the *heredoc-delimiter* is of the form *non-quoted-delimiter*, the *heredoc-delimiter-identifier* shall be the same sequence of characters as the *non-quoted-delimiter-identifier*; if it is of the form *single-quoted-delimiter*, the *single-quoted-delimiter-identifier*; if it is of the form of *double-quoted-delimiter*, the *double-quoted-delimiter-identifier*; if it is of the form of *command-quoted-delimiter*, the *command-quoted-delimiter-identifier*.

Semantics

A *here-document* evaluates to a direct instance of the class `String` or the value of the invocation of the method `` ` ``.

The object to which a *here-document* evaluates is created as follows:

a) Create a direct instance S of the class `String` from the *heredoc-body*, the content of which depends on the form of the *heredoc-delimiter* as follows:

- If *heredoc-delimiter* is of the form *single-quoted-delimiter*, the content of S is the sequence of *source-character*s of the *heredoc-body*.

- If *heredoc-delimiter* is in any of the forms *non-quoted-delimiter*, *double-quoted-delimiter*, or *command-quoted-delimiter*, the content of S is the sequence of characters which is represented by the *heredoc-body* as a sequence of *double-quoted-string-character*s (see 8.7.6.3.3).

b) If the *heredoc-delimiter* is not of the form *command-quoted-delimiter*, let V be S.

c) Otherwise, invoke the method `` ` `` on the current self with the list of arguments which has only one element S. Let V be the resulting value of the method invocation.

d) V is the object to which the *here-document* evaluates.

8.7.6.3.7 External command execution

Syntax

---

external-command-execution ::
  backquoted-external-command-execution
  | *quoted-external-command-execution*

backquoted-external-command-execution ::
  `` ` `` *backquoted-external-command-execution-character** `` ` ``

backquoted-external-command-execution-character ::
  source-character **but not** (` | # | \ )
  | # [lookahead ∉ { $, @, { }]
  | *double-escape-sequence*
  | *interpolated-character-sequence*

quoted-external-command-execution ::
  `%x` *expanded-delimited-string*

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

Semantics

An *external-command-execution* is a form to invoke the method `.

An *external-command-execution* is evaluated as follows:

a) If the *external-command-execution* is of the form *backquoted-external-command-execution*, construct a direct instance *S* of the class `String` whose content is a sequence of characters represented by *backquoted-external-command-execution-character*s. A *backquoted-external-command-execution-character* other than a *double-escape-sequence* or an *interpolated-character-sequence* represents a character as it occurs in a program text. A *double-escape-sequence* and an *interpolated-character-sequence* represent characters as described in 8.7.6.3.3.

b) If the *external-command-execution* is of the form *quoted-external-command-execution*, construct a direct instance *S* of the class `String` by replacing "`%x`" with "`%Q`" and evaluating the resulting *quoted-expanded-literal-string* as described in 8.7.6.3.5.

c) Invoke the method ` on the current self with a list of arguments which has only one element *S*.

d) The value of the *external-command-execution* is the resulting value.

8.7.6.4 Array literals

Syntax

---

array-literal ::
  quoted-non-expanded-array-constructor
  | *quoted-expanded-array-constructor*

quoted-non-expanded-array-constructor ::
  `%w` *literal-beginning-delimiter* *non-expanded-array-content* *literal-ending-delimiter*

non-expanded-array-content ::
  quoted-array-item-separator-list? *non-expanded-array-item-list*?
    quoted-array-item-separator-list?

non-expanded-array-item-list ::
    non-expanded-array-item (*quoted-array-item-separator-list* *non-expanded-array-item*)*

quoted-array-item-separator-list ::
    quoted-array-item-separator^{+}

quoted-array-item-separator ::
    whitespace
  | *line-terminator*

non-expanded-array-item ::
    non-expanded-array-item-character^{+}

non-expanded-array-item-character ::
    non-escaped-array-character
  | *non-expanded-array-escape-sequence*

non-escaped-array-character ::
    non-escaped-literal-character **but not** *quoted-array-item-separator*

non-expanded-array-escape-sequence ::
    non-expanded-literal-escape-sequence
  | `\` *quoted-array-item-separator*

quoted-expanded-array-constructor ::
    `%W` *literal-beginning-delimiter* *expanded-array-content* *literal-ending-delimiter*

expanded-array-content ::
    quoted-array-item-separator-list$^{?}$ *expanded-array-item-list*$^{?}$
      quoted-array-item-separator-list$^{?}$

expanded-array-item-list ::
    expanded-array-item (*quoted-array-item-separator-list* *expanded-array-item*)*

expanded-array-item ::
    expanded-array-item-character^{+}

expanded-array-item-character ::
    non-escaped-array-item-character
  | `#` [lookahead $\notin$ { `$`, `@`, `{` }]
  | *expanded-array-escape-sequence*
  | *interpolated-character-sequence*

non-escaped-array-item-character ::
    source-character **but not** (*quoted-array-item-separator* | `\` | `#`)

expanded-array-escape-sequence ::
double-escape-sequence
| \ *quoted-array-item-separator*

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

If the *literal-beginning-delimiter* is none of the characters on the left in 8.7.6.3.4 Table 2, the *non-escaped-array-item-character* shall not be the *literal-beginning-delimiter*.

If the *literal-beginning-delimiter* is one of the characters on the left in 8.7.6.3.4 Table 2, the *quoted-non-expanded-array-constructor* or *quoted-expanded-array-constructor* shall satisfy the following conditions, where C is the *quoted-non-expanded-array-constructor* or *quoted-expanded-array-constructor*, B is the *literal-beginning-delimiter*, and E is the *literal-ending-delimiter* which corresponds to B in 8.7.6.3.4 Table 2, and "the number of x in y" means the number of x to appear in y except appearances in *non-expanded-array-escape-sequence*s or *expanded-array-escape-sequence*s:

- The number of B in C and the number of E in C are the same.
- For any substring S of C which starts from the first B and ends before the last E, the number of B in S is larger than the number of E in S.

NOTE The above conditions are for nested brackets in an *array-literal*. Matching of brackets is irrelevant to the structure of the value of an *array-literal*. For example, `%w[[ab cd][ef]]` represents `["[ab", "cd][ef]"]`.

Semantics

An *array-literal* evaluates to a direct instance of the class `Array` as follows:

a) A *quoted-non-expanded-array-constructor* is evaluated as follows:

   1) Create an empty direct instance of the class `Array`. Let A be the instance.

   2) If *non-expanded-array-item-list* is present, for each *non-expanded-array-item* of the *non-expanded-array-item-list*, take the following steps:

      i) Create a direct instance S of the class `String`, the content of which is represented by the sequence of *non-expanded-array-item-character*s.

         A *non-expanded-array-item-character* represents itself, except in the case of a *non-expanded-array-escape-sequence*. A *non-expanded-array-escape-sequence* represents a character represented by the *non-expanded-literal-escape-sequence* as described in 8.7.6.3.4, except when the *non-expanded-array-escape-sequence* is of the form \ *quoted-array-item-separator*. A *non-expanded-array-escape-sequence* of the form \ *quoted-array-item-separator* represents the *quoted-array-item-separator* as it occurs in a program text literally.

      ii) Append S to A.

   3) The value of the *quoted-non-expanded-array-constructor* is A.

b) A *quoted-expanded-array-constructor* is evaluated as follows:

1) Create an empty direct instance of the class `Array`. Let A be the instance.

2) If *expanded-array-item-list* is present, process each *expanded-array-item* of the *expanded-array-item-list* as follows:

i) Create a direct instance S of the class `String`, the content of which is represented by the sequence of *expanded-array-item-character*s.

An *expanded-array-item-character* represents itself, except in the case of an *expanded-array-escape-sequence* and an *interpolated-character-sequence*. An *expanded-array-escape-sequence* represents a character represented by the *double-escape-sequence* as described in 8.7.6.3.3, except when the *expanded-array-escape-sequence* is of the form `\` *quoted-array-item-separator*. An *expanded-array-escape-sequence* of the form `\` *quoted-array-item-separator* represents the *quoted-array-item-separator* as it occurs in a program text literally. An *interpolated-character-sequence* represents a sequence of characters as described in 8.7.6.3.3.

ii) Append S to A.

3) The value of the *quoted-expanded-array-constructor* is A.

8.7.6.5 Regular expression literals

Syntax

---

regular-expression-literal ::
    `/` *regular-expression-body* `/` *regular-expression-option**
  | `%r` *literal-beginning-delimiter* *expanded-literal-string**
      literal-ending-delimiter *regular-expression-option**

regular-expression-body ::
    *regular-expression-character**

regular-expression-character ::
    source-character **but not** (`/` | `#` | `\`)
  | `#` [lookahead $\notin$ { `$`, `@`, `{` }]
  | *regular-expression-unescaped-sequence*
  | *regular-expression-escape-sequence*
  | *line-terminator-escape-sequence*
  | *interpolated-character-sequence*

regular-expression-unescaped-sequence ::
    `\` *regular-expression-unescaped-character*

regular-expression-unescaped-character ::
    source-character **but not** (0x0d | 0x0a)
  | 0x0d [lookahead $\notin$ { 0x0a }]

regular-expression-escape-sequence ::
`\ /`

regular-expression-option ::
`i` | `m`

Within an *expanded-literal-string* of a *regular-expression-literal*, a *literal-beginning-delimiter* shall be the same character as the *literal-beginning-delimiter* of the *regular-expression-literal*.

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

Semantics

A *regular-expression-literal* evaluates to a direct instance of the class `Regexp`.

The pattern attribute of an instance of the class `Regexp` (see 15.2.15.1) resulting from a *regular-expression-literal* is the string represented by *regular-expression-character*s or *expanded-literal-string*s. The string shall be of the form *pattern* (see 15.2.15.4).

A *regular-expression-character* other than a *regular-expression-escape-sequence*, *line-terminator-escape-sequence*, or *interpolated-character-sequence* represents itself as it occurs in a program text literally. An *expanded-literal-string* other than a *line-terminator-escape-sequence* or *interpolated-character-sequence* represents itself as it occurs in a program text literally.

A *regular-expression-escape-sequence* represents the character `/`.

A *line-terminator-escape-sequence* in a *regular-expression-character* and an *expanded-literal-string* is ignored in the resulting pattern of an instance of the class `Regexp`.

An *interpolated-character-sequence* in a *regular-expression-literal* and an *expanded-literal-string* is evaluated as described in 8.7.6.3.3, and represents a string which is the content of the resulting instance of the class `String`.

A *regular-expression-option* specifies the ignorecase-flag and the multiline-flag attributes of an instance of the class `Regexp` resulting from a *regular-expression-literal*. If `i` is present in a *regular-expression-option*, the ignorecase-flag attribute of the resulting instance of the class `Regexp` is set to true. If `m` is present in a *regular-expression-option*, the multiline-flag attribute of the resulting instance of the class `Regexp` is set to true.

The grammar for a pattern of an instance of the class `Regexp` created from a *regular-expression-literal* is described in 15.2.15.4.

8.7.6.6 Symbol literals

Syntax

---

symbol ::
symbol-literal
| *dynamic-symbol*

symbol-literal ::
  `:` *symbol-name*

dynamic-symbol ::
  `:` *single-quoted-string*
  | `:` *double-quoted-string*
  | `%s` *literal-beginning-delimiter non-expanded-literal-string* literal-ending-delimiter*

symbol-name ::
  instance-variable-identifier
  | *global-variable-identifier*
  | *class-variable-identifier*
  | *constant-identifier*
  | *local-variable-identifier*
  | *method-only-identifier*
  | *assignment-like-method-identifier*
  | *operator-method-name*
  | *keyword*

The *single-quoted-string*, *double-quoted-string*, or *non-expanded-literal-string* of the *dynamic-symbol* shall not contain any sequence which represents the character 0x00 in the resulting value of the *single-quoted-string*, *double-quoted-string*, or *non-expanded-literal-string* as described in 8.7.6.3.2, 8.7.6.3.3, or 8.7.6.3.4.

Within a *non-expanded-literal-string*, *literal-beginning-delimiter* shall be the same character as the *literal-beginning-delimiter* of the *dynamic-symbol*.

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

Semantics

A *symbol* evaluates to a direct instance of the class `Symbol`. A *symbol-literal* evaluates to a direct instance of the class `Symbol` whose name is the *symbol-name*. A *dynamic-symbol* evaluates to a direct instance of the class `Symbol` whose name is the content of an instance of the class `String` which is the value of the *single-quoted-string* (see 8.7.6.3.2), *double-quoted-string* (see 8.7.6.3.3), or *non-expanded-literal-string* (see 8.7.6.3.4). If the content of the instance of the class `String` contains the character 0x00, a direct instance of the class `ArgumentError` may be raised.

9 Scope of variables

9.1 General description

The ***scope*** of a variable is a set of regions of a program text with which a set of bindings of variables is associated.

Instance variables, constants, and class variables have no scope, and their bindings are searched depending on values of attirubtes of execution contexts (see 11.5.4.2, 11.5.4.5, and 11.5.4.6).

9.2 Scope of local variables

A local variable is referred to by a *local-variable-identifier*.

Scopes for local variables are introduced by the following program constructs:

- *program* (see 10.1)
- *class-body* (see 13.2.2)
- *module-body* (see 13.1.2)
- *singleton-class-body* (see 13.4.2)
- *method-definition* (see 13.3.1) and *singleton-method-definition* (see 13.4.3), for both of which the scope starts with the *method-parameter-part* and continues up to and including the *method-body*.
- *block* (see 11.3.3)

Let P be any of the above program constructs. Let S be the region of P excluding all the regions of any of the above program constructs (except *block*) nested within P. Then, S is the ***local variable scope*** which corresponds to the program construct P.

The scope of a local variable is the local variable scope whose set of local variable bindings contains the binding of the local variable, which is resolved as described below.

Given a *local-variable-identifier* which is a reference to a local variable, the binding of the local variable is resolved as follows:

a) Let N be the *local-variable-identifier*. Let B be the current set of local variable bindings.

b) Let S be the scope of B.

c) If a binding with name N exists in B, that binding is the resolved binding.

d) If a binding with name N does not exist in B:

   1) If S is a local variable scope which corresponds to a *block*:

      i) If the *local-variable-identifier* occurs as a *left-hand-side* of a *block-parameter-list*, whether to proceed to the next step or not is implementation-defined.

      ii) Let new B be the element immediately below the current B on ⟦local-variable-bindings⟧, and continue searching for a binding with name N from Step b).

   2) Otherwise, a binding is considered not resolved.

9.3 Scope of global variables

The scope of global variables is global in the sense that they are accessible everywhere in a program. Global variable bindings are created in ⟦global-variable-bindings⟧.

10 Program structure

10.1 Program

Syntax

---

program ::
    compound-statement

The program text of a strictly conforming program shall be an element of the set of sequences of characters represented by the nonterminal symbol *program*. A conforming processor need not accept programs that include program constructs which cannot be evaluated.

Semantics

A *program* is evaluated as follows:

a) Push an empty set onto ⟦local-variable-bindings⟧ .

b) Evaluate the *compound-statement*.

c) The value of the *program* is the resulting value.

d) Restore the execution context by removing the element from the top of ⟦local-variable-bindings⟧ .

10.2 Compound statement

Syntax

---

compound-statement ::
    statement-list$^?$ *separator-list*$^?$

statement-list ::
    statement (*separator-list* *statement*)*

separator-list ::
    separator$^+$

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

Semantics

A *compound-statement* is evaluated as follows:

a) If the *statement-list* of the *compound-statement* is omitted, the value of the *compound-statement* is **nil**.

b) If the *statement-list* of the *compound-statement* is present, evaluate each *statement* of the *statement-list* in the order it appears in the program text. The value of the *compound-statement* is the value of the last *statement* of the *statement-list*.

11 Expressions

11.1 General description

Syntax

---

expression ::
    keyword-logical-expression

An *expression* is a program construct which make up a *statement* (see 12). A single *expression* can be a *statement* as an *expression-statement* (see 12.2).

NOTE A difference between an *expression* and a *statement* is that an *expression* is ordinarily used where its value is required, but a *statement* is ordinarily used where its value is not necessarily required. However, there are some exceptions. For example, a *jump-expression* (see 11.5.2.4) does not have a value, and the value of the last *statement* of a *compound-statement* can be used.

Semantics

See 11.2.2 for *keyword-logical-expressions*.

11.2 Logical expressions

11.2.1 General description

Syntax

---

logical-expression ::=
    logical-NOT-expression
    | *logical-AND-expression*
    | *logical-OR-expression*

Any of *logical-NOT-expression*, *logical-AND-expression*, and *logical-OR-expression* is a conceptual name, which is used to organize that of the form using a keyword (e.g., "`not x`") and that of the form using an operator (e.g, "`!x`"), because they are syntactically away from each other.

See 11.2.3 for *logical-NOT-expressions*. See 11.2.4 for *logical-AND-expressions*. See 11.2.5 for *logical-OR-expressions*.

11.2.2 Keyword logical expressions

Syntax

---

keyword-logical-expression ::
    keyword-NOT-expression
  | *keyword-AND-expression*
  | *keyword-OR-expression*

See 11.2.3 for *keyword-NOT-expression*s. See 11.2.4 for *keyword-AND-expression*s. See 11.2.5 for *keyword-OR-expression*s.

11.2.3 Logical NOT expressions

Syntax

---

logical-NOT-expression ::=
    keyword-NOT-expression
  | *operator-NOT-expression*

keyword-NOT-expression ::
    method-invocation-without-parentheses
  | *operator-expression*
  | `!` *method-invocation-without-parentheses*
  | `not` *keyword-NOT-expression*

operator-NOT-expression ::=
    `!` (*method-invocation-without-parentheses* | *unary-expression*)

Semantics

A *logical-NOT-expression* is evaluated as follows:

a) If it is of the form *method-invocation-without-parentheses*, evaluate it as described in 11.3.

b) If it is of the form *operator-expression*, evaluate it as described in 11.4.

c) Otherwise:

   1) If it is of the form `not` *keyword-NOT-expression*, evaluate the *keyword-NOT-expression*. Let X be the resulting value.

   2) If it is an *operator-NOT-expression*, evaluate its *method-invocation-without-parentheses* or *unary-expression*. Let X be the resulting value.

   3) If X is a trueish object, the value of the *keyword-NOT-expression* or the *operator-NOT-expression* is **false**.

4) Otherwise, the value of the *keyword-NOT-expression* or the *operator-NOT-expression* is **true**.

d) If it is a *operator-NOT-expression*, instead of Step c), the *operator-NOT-expression* may be evaluated as follows:

1) Evaluate the *method-invocation-without-parentheses* or the *unary-expression*. Let V be the resulting value.

2) Create an empty list of arguments L. Invoke the method `!@` on V with L as the list of arguments. The value of the *operator-NOT-expression* is the resulting value.

In this case, the processor shall:

- include the operator `!@` in *operator-method-name*.
- define an instance method `!@` in the class `Object`, one of its superclasses (see 6.5.4), or a module included in the class `Object`. The method `!@` shall not take any arguments and shall return **true** if the receiver is **false** or **nil**, and shall return **false** otherwise.

11.2.4 Logical AND expressions

Syntax

---

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

keyword-AND-expression ::
    expression [no *line-terminator* here] `and` *keyword-NOT-expression*

operator-AND-expression ::
    equality-expression
  | *operator-AND-expression* [no *line-terminator* here] `&&` *equality-expression*

Semantics

A *logical-AND-expression* is evaluated as follows:

a) If the *logical-AND-expression* is a *equality-expression*, evaluate the *equality-expression* as described in 11.4.4.

b) Otherwise:

1) Evaluate the *expression* or the *operator-AND-expression*. Let X be the resulting value.

2) If X is a trueish object, evaluate the *keyword-NOT-expression* or *equality-expression*. Let Y be the resulting value. The value of the *keyword-AND-expression* or the *operator-AND-expression* is Y.

3) Otherwise, the value of the *keyword-AND-expression* or the *operator-AND-expression* is X.

11.2.5 Logical OR expressions

Syntax

---

logical-OR-expression ::=
    keyword-OR-expression
    | *operator-OR-expression*

keyword-OR-expression ::
    expression [no *line-terminator* here] `or` *keyword-NOT-expression*

operator-OR-expression ::
    operator-AND-expression
    | *operator-OR-expression* [no *line-terminator* here] `||` *operator-AND-expression*

Semantics

A *logical-OR-expression* is evaluated as follows:

a) If the *logical-OR-expression* is a *operator-AND-expression*, evaluate the *operator-AND-expression* as described in 11.2.4.

b) Otherwise:

1) Evaluate the *expression* or the *operator-OR-expression*. Let X be the resulting value.

2) If X is a falseish object, evaluate the *keyword-NOT-expression* or the *operator-AND-expression*. Let Y be the resulting value. The value of the *keyword-OR-expression* or *operator-OR-expression* is Y.

3) Otherwise, the value of the *keyword-OR-expression* or *operator-OR-expression* is X.

11.3 Method invocation expressions

11.3.1 General description

Syntax

---

method-invocation-expression ::=
    primary-method-invocation
    | *method-invocation-without-parentheses*
    | *local-variable-identifier*

primary-method-invocation ::
    super-with-optional-argument

| *indexing-method-invocation*
| *method-only-identifier*
| *method-identifier* *block*
| *method-identifier*
[no *line-terminator* here] [no *whitespace* here] *argument-with-parentheses*
block$^?$
| *primary-expression* [no *line-terminator* here] `.` *method-name*
([no *line-terminator* here] [no *whitespace* here] *argument-with-parentheses*)$^?$
block$^?$
| *primary-expression* [no *line-terminator* here] `::` *method-name*
[no *line-terminator* here] [no *whitespace* here] *argument-with-parentheses*
block$^?$
| *primary-expression* [no *line-terminator* here] `::` *method-name-except-constant*
block$^?$

method-identifier ::
local-variable-identifier
| *constant-identifier*
| *method-only-identifier*

method-name ::
method-identifier
| *operator-method-name*
| *keyword*

indexing-method-invocation ::
primary-expression [no *line-terminator* here] [no *whitespace* here]
`[` *indexing-argument-list*$^?$ `]`

method-name-except-constant ::
method-name **but not** *constant-identifier*

method-invocation-without-parentheses ::
command
| *chained-command-with-do-block*
| *chained-command-with-do-block* (`.` | `::`) *method-name*
argument-without-parentheses
| *return-with-argument*
| *break-with-argument*
| *next-with-argument*

command ::
super-with-argument
| *yield-with-argument*
| *method-identifier* *argument-without-parentheses*
| *primary-expression* [no *line-terminator* here] (`.` | `::`) *method-name*
argument-without-parentheses

chained-command-with-do-block ::
  command-with-do-block *chained-method-invocation**

chained-method-invocation ::
  (. | ::) *method-name*
  | (. | ::) *method-name* [no *line-terminator* here] [no *whitespace* here]
    argument-with-parentheses

command-with-do-block ::
  super-with-argument-and-do-block
  | *method-identifier* *argument-without-parentheses* *do-block*
  | *primary-expression* [no *line-terminator* here]
    (. | ::) *method-name* *argument-without-parentheses* *do-block*

See 11.5.4.7 for *method-invocation-expression*s of the form *local-variable-identifier*.

If the *argument-with-parentheses* (see 11.3.2) of a *primary-method-invocation* is present, and the *argument-list* of the *argument-with-parentheses* is a *block-argument*, the *block* of the *primary-method-invocation* shall be omitted.

If the *argument-without-parentheses* of a *command-with-do-block* is present, and the *block-argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) is present, the *do-block* of the *command-with-do-block* shall be omitted.

Semantics

A *method-invocation-expression* is evaluated as follows:

a) A *primary-method-invocation* is evaluated as follows:

 1) If the *primary-method-invocation* is a *super-with-optional-argument* (see 11.3.4) or an *indexing-method-invocation*, evaluate it. The value of the *primary-method-invocation* is the resulting value.

 2) i) If the *primary-method-invocation* is a *method-only-identifier*, let O be the current self and let M be the *method-only-identifier*. Create an empty list of arguments L.

  ii) If the *method-identifier* of the *primary-method-invocation* is present:

   I) Let O be the current self and let M be the *method-identifier*.

   II) If the *argument-with-parentheses* is present, construct a list of arguments and a block from the *argument-with-parentheses* as described in 11.3.2. Let L be the resulting list. Let B be the resulting block, if any.

   If the *argument-with-parentheses* is omitted, create an empty list of arguments L.

   III) If the *block* is present, let B be the *block*.

iii) If "`.`" of the *primary-method-invocation* is present:

I) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name*.

II) If the *argument-with-parentheses* is present, construct a list of arguments and a block from the *argument-with-parentheses* as described in 11.3.2. Let L be the resulting list. Let B be the resulting block, if any.

If the *argument-with-parentheses* is omitted, create an empty list of arguments L.

III) If the *block* is present, let B be the *block*.

iv) If the `::` and *method-name* of the *primary-method-invocation* are present:

I) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name*.

II) Construct a list of arguments and a block from the *argument-with-parentheses* as described in 11.3.2. Let L be the resulting list. Let B be the resulting block, if any.

III) If the *block* is present, let B be the *block*.

v) If the `::` and *method-name-except-constant* of the *primary-method-invocation* are present:

I) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name-except-constant*.

II) Create an empty list of arguments L.

III) If the *block* is present, let B be the *block*.

3) Invoke the method M on O with L as the list of arguments and B, if any, as the block. (see 13.3.3). The value of the *primary-method-invocation* is the resulting value.

b) An *indexing-method-invocation* is evaluated as follows:

1) Evaluate the *primary-expression*. Let O be the resulting value.

2) If the *indexing-argument-list* is present, construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let L be the resulting list.

3) If the *indexing-argument-list* is omitted, Create an empty list of arguments L.

4) Invoke the method `[]` on O with L as the list of arguments. The value of the *indexing-method-invocation* is the resulting value.

c) A *method-invocation-without-parentheses* is evaluated as follows:

1) If the *method-invocation-without-parentheses* is a *command*, evaluate it. The value of the *method-invocation-without-parentheses* is the resulting value.

2) If the *method-invocation-without-parentheses* is a *return-with-argument*, *break-with-argument* or *next-with-argument*, evaluate it (see 11.5.2.4). The value of the *method-invocation-without-parentheses* is the resulting value.

3) If the *chained-command-with-do-block* of the *method-invocation-without-parentheses* is present:

i) Evaluate the *chained-command-with-do-block*. Let V be the resulting value.

ii) If the *method-name* and the *argument-without-parentheses* of the *method-invocation-without-parentheses* are present:

I) Let M be the *method-name*.

II) Construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2 and let L be the resulting list. If the *block-argument* of the *argument-list* of the *argument-without-parentheses* is present, let B be the *block* to which the *block-argument* corresponds [see 11.3.2 e) 6)].

III) Invoke the method M on V with L as the list of arguments and B, if any, as the block.

IV) Replace V with the resulting value.

iii) The value of the *method-invocation-without-parentheses* is V.

d) A *command* is evaluated as follows:

1) If the *command* is a *super-with-argument*(see 11.3.4) or a *yield-with-argument* (see 11.3.5), evaluate it. The value of the *command* is the resulting value.

2) Otherwise:

i) If the *method-identifier* of the *command* is present:

I) If the *method-identifier* is a *local-variable-identifier*, and if the *local-variable-identifier* is considered as a reference to a local variable by the steps in 11.5.4.7.2), and if the *argument-without-parentheses* starts with any of `&`, `<<`, `+`, `-`, `*`, `/`, and `%`, the behavior is unspecified.

NOTE 1 For example, if `x` is a reference to a local variable, the behavior of "`x -1`" is unspecified. The behavior of "`x -1`" may be the same as an *additive-expression* (see 11.4.4) of the form "`x - 1`".

II) Let O be the current self and let M be the *method-identifier*.

III) Construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2 and let L be the resulting list.

If the *block-argument* of the *argument-list* of the *argument-without-parentheses* is present, let B be the *block* to which the *block-argument* corresponds.

ii) If the *primary-expression*(see 11.5), *method-name*, and *argument-without-parentheses* of the *command* are present:

I) Evaluate the *primary-expression*. Let O be the resulting value. Let M be the *method-name*.

II) Construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2 and let L be the resulting list.

If the *block-argument* of the *argument-list* of the *argument-without-parentheses* is present, let B be the *block* to which the *block-argument* corresponds.

iii) Invoke the method M on O with L as the list of arguments and B, if any, as the block. The value of the *command* is the resulting value.

e) A *chained-command-with-do-block* is evaluated as follows:

1) Evaluate the *command-with-do-block* and let V be the resulting value.

2) For each *chained-method-invocation*, in the order they appear in the program text, take the following steps:

i) Let M be the *method-name* of the *chained-method-invocation*.

ii) If the *argument-with-parentheses* is present, construct a list of arguments and a block from the *argument-with-parentheses* as described in 11.3.2 and let L be the resulting list. Let B be the resulting block, if any.

If the *argument-with-parentheses* is omitted, create an empty list of arguments L.

iii) Invoke the method M on V with L as the list of arguments and B, if any, as the block.

iv) Replace V with the resulting value.

3) The value of the *chained-command-with-do-block* is V.

f) A *command-with-do-block* is evaluated as follows:

1) If the *command-with-do-block* is a *super-with-argument-and-do-block*, evaluate it. The value of the *command-with-do-block* is the resulting value.

2) Otherwise:

i) If the *method-identifier* of the *command-with-do-block* is present:

I) If the *method-identifier* is a *local-variable-identifier*, and if the *local-variable-identifier* is considered as a reference to a local variable by the steps in

11.5.4.7.2), and if the *argument-without-parentheses* starts with any of `&`, `<<`, `+`, `-`, `*`, `/`, and `%`, the behavior is unspecified.

NOTE 2 For example, if `x` is a reference to a local variable, the behavior of "`x -1 do end`" is unspecified.

II) Otherwise, let O be the current self and let M be the *method-identifier*.

ii) If the *primary-expression* of the *command-with-do-block* is present, evaluate the *primary-expression*, and let O be the resulting value and let M be the *method-name*.

iii) Construct a list of arguments from the *argument-without-parentheses* of the *command-with-do-block* and let L be the resulting list.

iv) Invoke the method M on O with L as the list of arguments and the *do-block* as the block. The value of the *command-with-do-block* is the resulting value.

11.3.2 Method arguments

Syntax

---

method-argument ::=
    indexing-argument-list
  | *argument-with-parentheses*
  | *argument-without-parentheses*

indexing-argument-list ::
    command
  | *operator-expression-list* ([no *line-terminator* here] `,`)?
  | *operator-expression-list* [no *line-terminator* here] `,` *splatting-argument*
  | *association-list* ([no *line-terminator* here] `,`)?
  | *splatting-argument*

splatting-argument ::
    `*` *operator-expression*

operator-expression-list ::
    operator-expression ([no *line-terminator* here] `,` *operator-expression*)*

argument-with-parentheses ::
    `( )`
  | `(` *argument-list* `)`
  | `(` *operator-expression-list* [no *line-terminator* here] `,` *chained-command-with-do-block* `)`
  | `(` *chained-command-with-do-block* `)`

argument-without-parentheses ::
    [lookahead $\notin$ { `{` }] [no *line-terminator* here] *argument-list*

argument-list ::
    block-argument
    | *splatting-argument* (, *block-argument*)?
    | *operator-expression-list* [no *line-terminator* here] , *association-list* ([no *line-terminator* here] , *splatting-argument*)? ([no *line-terminator* here] , *block-argument*)?
    | (*operator-expression-list* | *association-list*) ([no *line-terminator* here] , *splatting-argument*)? ([no *line-terminator* here] , *block-argument*)?
    | *command*

block-argument ::
    `&` *operator-expression*

If an *argument-without-parentheses* starts with a sequence of characters which is any of `&`, `<<`, `+`, `-`, `*`, `/`, and `%`:

- One or more *whitespace* characters shall be present just before the *argument-without-parentheses*.
- No *whitespace* shall be present just after the sequence of characters.

NOTE These constraints are necessary to distinguish the sequence of characters from binary operators (see 11.4.4). For example, "`x -y`" is considered as a *command*. However, "`x-y`" and "`x - y`" are not considered as *commands*, but as *additive-expressions*. That is, if `x` is not a reference to a local variable, the behaviors of "`x-y`" and "`x - y`" are the same as "`x() - y`".

Semantics

A *method-argument* evaluates to two values: an argument list, and a *block*. These two values are used when the method is invoked. However, a *method-argument* does not have a *block* value depending on evlauation steps.

A *method-argument* is evaluated as follows:

a) An *indexing-argument-list* is evaluated as follows:

   1) Create an empty list of arguments L.

   2) Evaluate the *command*, *operator-expressions* of *operator-expression-lists*, or the *association-list* and append their values to L in the order they appear in the program text.

   3) If the *splatting-argument* is present, evaluate it, and concatenate the resulting list of arguments to L.

   4) The argument list value of *indexing-argument-list* is L.

b) A *splatting-argument* is evaluated as follows:

   1) Create an empty list of arguments L.

2) Evaluate the *operator-expression*. Let V be the resulting value.

3) If V is not an instance of the class `Array`, the behavior is unspecified.

4) Append each element of V, in the indexing order, to L.

5) The argument list value of *splatting-argument* is L.

c) An *argument-with-parentheses* is evaluated as follows:

1) Create an empty list of arguments L.

2) If the *argument-list* is present, evaluate it as described in Step e), and concatenate the resulting list of arguments to L. If the *block-argument* of the *argument-list* is present, the *block* value of the *argument-with-parentheses* is the *block* value of the *argument-list*.

3) If the *operator-expression-list* is present, for each *operator-expression* of the *operator-expression-list*, in the order they appear in the program text, take the following steps:

i) Evaluate the *operator-expression*. Let V be the resulting value.

ii) Append V to L.

4) If the *chained-command-with-do-block* is present, evaluate it. Append the resulting value to L.

5) The argument list value of *argument-with-parentheses* is L.

d) An *argument-without-parentheses* is evaluated as follows:

1) If the first character of the *argument-without-parentheses* is `(`, the behavior is unspecified.

2) Evaluate the *argument-list* as described in Step e).

3) Let L be the resulting list.

e) An *argument-list* is evaluated as follows:

1) Create an empty list of arguments L.

2) If the *command* is present, evaluate it, and append the resulting value to L.

3) If the *operator-expression-list* is present, for each *operator-expression* of the *operator-expression-list*, in the order they appear in the program text, take the following steps:

i) Evaluate the *operator-expression*. Let V be the resulting value.

ii) Append V to L.

4) If the *association-list* is present, evaluate it. Append the resulting value to L.

5) If the *splatting-argument* is present, construct a list of arguments from it and concatenate the resulting list to L.

6) If the *block-argument* is present:

   i) Evaluate the *operator-expression* of the *block-argument*. Let P be the resulting value.

   ii) If P is not an instance of the class `Proc`, the behavior is unspecified.

   iii) Otherwise, the *block* value of *argument-list* is the block which P represents.

7) The argument list value of *argument-list* is L.

11.3.3 Blocks

Syntax

---

block ::
    brace-block
  | *do-block*

brace-block ::
    `{` *block-parameter*$^?$ *block-body* `}`

do-block ::
    `do` *block-parameter*$^?$ *block-body* `end`

block-parameter ::
    `| |`
  | `||`
  | `|` *block-parameter-list* `|`

block-parameter-list ::
    left-hand-side
  | *multiple-left-hand-side*

block-body ::
    compound-statement

Whether the *left-hand-side* (see 11.4.2.4) in the *block-parameter-list* is allowed to be of the following forms is implementation-defined.

- *constant-identifier*
- *global-variable-identifier*

- *instance-variable-identifier*
- *class-variable-identifier*
- *primary-expression* `[` *indexing-argument-list*? `]`
- *primary-expression* (`.` | `::`) (*local-variable-identifier* | *constant-identifier*)
- `::` *constant-identifier*

NOTE Some existing implementations allow some syntactic constructs such as *constant-identifier*s in a *block-parameter*. Whether they are allowed is therefore implementation-defined. Future implementations should not allow them.

Whether the *grouped-left-hand-side* (see 11.4.2.4) of the *multiple-left-hand-side* of the *block-parameter-list* is allowed to be of the following form is implementation-defined.

- `(` (*multiple-left-hand-side-item* `,`)$^{+}$ `)`

Semantics

A *block* is a procedure which is passed to a method invocation.

A *block* can be called either by a *yield-expression* (see 11.3.5) or by invoking the method `call` on an instance of the class `Proc` which is created by an invocation of the method `new` on the class `Proc` to which the block is passed (see 15.2.17.4.3).

A *block* can be called with arguments. If a *block* is called by a *yield-expression*, the arguments to the *yield-expression* are used as the arguments to the *block* call. If a *block* is called by an invocation of the method `call`, the arguments to the method invocation is used as the arguments to the *block* call.

A *block* is evaluated within the execution context as it exists just before the method invocation to which the *block* is passed. However, the changes of variable bindings in ⟦local-variable-bindings⟧ after the *block* is passed to the method invocation affect the execution context. Let E_b be the possibly affected execution context.

When a *block* is called, the *block* is evaluated as follows:

a) Let E_o be the current execution context. Let L be the list of arguments passed to the block.

b) Set the execution context to E_b.

c) Push an empty set of local variable bindings onto ⟦local-variable-bindings⟧ .

d) If the *block-parameter-list* in the *do-block* or the *brace-block* is present:

   1) If the *block-parameter-list* is of the form *left-hand-side* or *grouped-left-hand-side*:

      i) If the length of L is 0, let X be **nil**.

      ii) If the length of L is 1, let X be the only element of L.

iii) If the length of L is larger than 1, the result of this step is unspecified.

iv) If the *block-parameter-list* is of the form *left-hand-side*, evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) E, where the *variable* of E is the *left-hand-side* and the value of the *operator-expression* of E is X.

v) If the *block-parameter-list* is of the form *grouped-left-hand-side*, evaluate a *many-to-many-assignment-statement* (see 11.4.2.4) E, where the *multiple-left-hand-side* of E is the *grouped-left-hand-side* and the value of the *method-invocation-without-parentheses* or *operator-expression* of E is X.

2) If the *block-parameter-list* is of the form *multiple-left-hand-side* and the *multiple-left-hand-side* is not a *grouped-left-hand-side*:

i) If the length of L is 1:

I) If the only element of L is not an instance of the class `Array`, the result of this step is unspecified.

II) Create a list of arguments Y which contains the elements of L, preserving their order.

ii) If the length of L is 0 or larger than 1, let Y be L.

iii) Evaluate the *many-to-many-assignment-statement* E as described in 11.4.2.4, where the *multiple-left-hand-side* of E is the *block-parameter-list* and the list of arguments constructed from the *multiple-right-hand-side* of E is Y.

e) Evaluate the *block-body*. If the evaluation of the *block-body*:

1) is terminated by a *break-expression*:

i) If the method invocation with which *block* is passed has already terminated when the *block* is called:

I) Let S be an instance of the class `Symbol` with name `break`.

II) If the *jump-argument* of the *break-expression* is present, let V be the value of the *jump-argument*. Otherwise, let V be **nil**.

III) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value V.

ii) Otherwise, restore the execution context to E_o and terminate Step 13.3.3 i) and take Step 13.3.3 j) of the current method invocation.

If the *jump-argument* of the *break-expression* is present, the value of the current method invocation is the value of the *jump-argument*. Otherwise, the value of the current method invocation is **nil**.

2) is terminated by a *redo-expression*, repeat Step e).

3) is terminated by a *next-expression*:

i) If the *jump-argument* of the *next-expression* is present, let V be the value of the *jump-argument*.

ii) Otherwise, let V be **nil**.

4) is terminated by a *return-expression*, remove the element from the top of ⟦local-variable-bindings⟧ .

5) is terminates otherwise, let V be the resulting value of the evaluation of the *block-body*.

f) Unless Step e) is terminated by a *return-expression*, restore the execution context to E_o.

g) The value of calling the *do-block* or the *brace-block* is V.

11.3.4 The `super` expression

Syntax

---

super-expression ::=
  super-with-optional-argument
  | *super-with-argument*
  | *super-with-argument-and-do-block*

super-with-optional-argument ::
  `super` ([no *line-terminator* here] [no *whitespace* here] *argument-with-parentheses*)$^?$
  block$^?$

super-with-argument ::
  `super` *argument-without-parentheses*

super-with-argument-and-do-block ::
  `super` *argument-without-parentheses* *do-block*

The *block-argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) of a *super-with-argument-and-do-block* shall be omitted.

Semantics

A *super-expression* is evaluated as follows:

a) If the current self is pushed by a *singleton-class-definition* (see 13.4.2), or an invocation of one of the following methods, the behavior is unspecified:

- the method `class_eval` of the class `Module` (see 15.2.2.4.15)
- the method `module_eval` of the class `Module` (see 15.2.2.4.35)

- the method `instance_eval` of the class `Kernel` (see 15.3.1.3.18)

b) Let A be an empty list. Let B be the top of ⟦block⟧ .

   1) If the *super-expression* is a *super-with-optional-argument*, and neither the *argument-with-parentheses* nor the *block* is present, construct a list of arguments as follows:

      i) Let M be the method which correspond to the current method invocation. Let L be the *parameter-list* of the *method-parameter-part* of M. Let S be the set of local variable bindings in ⟦local-variable-bindings⟧ which corresponds to the current method invocation.

      ii) If the *mandatory-parameter-list* is present in L, for each *mandatory-parameter* p, take the following steps:

         I) Let v be the value of the binding with name p in S.

         II) Append v to A.

      iii) If the *optional-parameter-list* is present in L, for each *optional-parameter* p, take the following steps:

         I) Let n be the *optional-parameter-name* of p.

         II) Let v be the value of the binding with name n in S.

         III) Append v to A.

      iv) If the *array-parameter* is present in L:

         I) Let n be the *array-parameter-name* of the *array-parameter*.

         II) Let v be the value of the binding with name n in S. Append each element of v, in the indexing order, to A.

   2) If the *super-expression* is a *super-with-optional-argument* with either or both of the *argument-with-parentheses* and the *block*:

      i) If the *argument-with-parentheses* is present, construct a list of arguments and a block as described in 11.3.2. Let A be the resulting list. Let B be the resulting block, if any.

      ii) If the *block* is present, let B be the *block*.

   3) If the *super-expression* is a *super-with-argument*, construct the list of arguments from the *argument-without-parentheses* as described in 11.3.2. Let A be the resulting list. If *block-argument* of the *argument-list* of *argument-without-parentheses* is present, let B be the *block* constructed from the *block-argument*.

   4) If the *super-expression* is a *super-with-argument-and-do-block*, construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2. Let A be the resulting list. Let B be the *do-block*.

c) Determine the method to be invoked as follows:

1) Let C be the current class or module. Let N be the top of ⟦defined-method-name⟧.

2) If C is an instance of the class `Class`:

i) Search for a method binding with name N from Step b) in 13.3.4, assuming that C in 13.3.4 to be C.

ii) If a binding is found and its value is not undef (see 13.1.1), let V be the value of the binding.

iii) Otherwise:

I) Add a direct instance of the class `Symbol` with name N to the head of A.

II) Invoke the method `method_missing` (see 15.3.1.3.30) on the current self with A as arguments and B as the block.

III) Terminate the evaluation of the *super-expression*. The value of the *super-expression* is the resulting value of the method invocation.

3) If C is an instance of the class `Module` and not an instance of the class `Class`:

i) Let M be C and let new C be the class of the current self.

ii) Let L_m be the included module list of C. Search for M in L_m.

iii) If M is found in L_m:

I) Search for a method binding with name N in the set of bindings of instance methods of each module in L_m. Examine modules in L_m, in reverse order, from the module just before M to the first module in L_m.

II) If a binding is found and its value is not undef, let V be the value of the binding.

III) If a binding is found and its value is undef (see 13.1.1), take the steps from c) 2) iii) I) to c) 2) iii) III).

IV) If a binding is not found and C has a direct superclass, let S be the superclass. Take Step c) 2), assuming that C in c) 2) to be S.

V) If a binding is not found and C does not have a direct superclass, take the steps from c) 2) iii) I) to c) 2) iii) III).

iv) Otherwise, let new C be the direct superclass of C and repeat from Step c) 3) ii). If C does not have a direct superclass, the behavior is unspecified.

d) Take steps g), h), i), and j) of 13.3.3, assuming that A, B, M, R, and V in 13.3.3 to be A, B, N, the current self, and V in this subclause respectively. The value of the *super-expression* is the resulting value.

11.3.5 The `yield` expression

Syntax

---

yield-expression ::=
  yield-with-optional-argument
  | *yield-with-argument*

yield-with-optional-argument ::
  yield-with-parentheses-and-argument
  | *yield-with-parentheses-without-argument*
  | `yield`

yield-with-parentheses-and-argument ::
  `yield` [no *line-terminator* here] [no *whitespace* here] `(` *argument-list* `)`

yield-with-parentheses-without-argument ::
  `yield` [no *line-terminator* here] [no *whitespace* here] `(` `)`

yield-with-argument ::
  `yield` *argument-without-parentheses*

The *block-argument* of the *argument-list* (see 11.3.2) of a *yield-with-parentheses-and-argument* shall be omitted.

The *block-argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) of a *yield-with-argument* shall be omitted.

Semantics

A *yield-expression* is evaluated as follows:

a) Let B be the top of $[\![$block$]\!]$. If B is block-not-given:

  1) Let S be a direct instance of the class `Symbol` with name `noreason`.

  2) Let V be an implementation-defined value.

  3) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value V.

b) A *yield-with-optional-argument* is evaluated as follows:

  1) If the *yield-with-optional-argument* is of the form *yield-with-parentheses-and-argument*, create a list of arguments from the *argument-without-parentheses* as described in 11.3.2. Let L be the list.

2) If the *yield-with-optional-argument* is of the form *yield-with-parentheses-without-argument* or `yield`, create an empty list of argument L.

3) Call B with L as described in 11.3.3.

4) The value of the *yield-with-optional-argument* is the value of the block call.

c) A *yield-with-argument* is evaluated as follows:

1) Create a list of arguments from the *argument-without-parentheses* as described in 11.3.2. Let L be the list.

2) Call B with L as described in 11.3.3.

3) The value of the *yield-with-argument* is the value of the block call.

11.4 Operator expressions

11.4.1 General description

Syntax

---

operator-expression ::
  assignment-expression
  | *defined?-without-parentheses*
  | *conditional-operator-expression*

See 11.4.2 for *assignment-expression*s.

NOTE *assignment-statement* is not an *operator-expression* but a *statement*(see 12.1).

See 11.4.3.2 for *defined?-without-parentheses*s.

NOTE *defined?-with-parentheses* is not an *operator-expression* but a *primary-expression*(see 11.5.1).

See 11.5.2.2.5 for *conditional-operator-expression*s.

11.4.2 Assignments

11.4.2.1 General description

Syntax

---

assignment ::=
  assignment-expression
  | *assignment-statement*

assignment-expression ::
  single-assignment-expression

| *abbreviated-assignment-expression*
| *assignment-with-rescue-modifier*

assignment-statement ::
single-assignment-statement
| *abbreviated-assignment-statement*
| *multiple-assignment-statement*

Semantics

An *assignment* creates or updates variable bindings, or invokes a method whose name ends with `=`.

Evaluations of *assignment-expression*s and *assignment-statement*s are described in the clauses from 11.4.2.2 to 11.4.2.5.

11.4.2.2 Single assignments

11.4.2.2.1 General description

Syntax

---

single-assignment ::=
single-assignment-expression
| *single-assignment-statement*

single-assignment-expression ::
single-variable-assignment-expression
| *scoped-constant-assignment-expression*
| *single-indexing-assignment-expression*
| *single-method-assignment-expression*

single-assignment-statement ::
single-variable-assignment-statement
| *scoped-constant-assignment-statement*
| *single-indexing-assignment-statement*
| *single-method-assignment-statement*

11.4.2.2.2 Single variable assignments

Syntax

---

single-variable-assignment ::=
single-variable-assignment-expression
| *single-variable-assignment-statement*

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

single-variable-assignment-statement ::
    variable [no *line-terminator* here] = *method-invocation-without-parentheses*

Semantics

A *single-variable-assignment* is evaluated as follows:

a) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let V be the resulting value.

b) 1) If the *variable*(see 11.5.4) is a *constant-identifier*:

i) Let N be the *constant-identifier*.

ii) If a binding with name N exists in the set of bindings of constants of the current class or module, replace the value of the binding with V.

iii) Otherwise, create a variable binding with name N and value V in the set of bindings of constants of the current class or module.

2) If the *variable* is a *global-variable-identifier*:

i) Let N be the *global-variable-identifier*.

ii) If a binding with name N exists in ⟦global-variable-bindings⟧, replace the value of the binding with V.

iii) Otherwise, create a variable binding with name N and value V in ⟦global-variable-bindings⟧.

3) If the *variable* is a *class-variable-identifier*:

i) Let C be the first class or module in the list at the top of ⟦class-module-list⟧ which is not a singleton class.

Let CS be the set of classes which consists of C and all the superclasses of C. Let MS be the set of modules which consists of all the modules in the included module lists of all classes in CS. Let CM be the union of CS and MS.

Let N be the *class-variable-identifier*.

ii) If exactly one of the classes or modules in CM has a binding with name N in the set of bindings of class variables, let B be that binding.

If more than one class or module in CM has bindings with name N in the set of bindings of class variables, choose a binding B from those bindings in an implementation-defined way.

Replace the value of B with V.

iii) If none of the classes or modules in CM has a binding with name N in the set of bindings of class variables, create a variable binding with name N and value V in the set of bindings of class variables of C.

4) If the *variable* is an *instance-variable-identifier*:

i) Let N be the *instance-variable-identifier*.

ii) If a binding with name N exists in the set of bindings of instance variables of the current self, replace the value of the binding with V.

iii) Otherwise, create a variable binding with name N and value V in the set of bindings of instance variables of the current self.

5) If the *variable* is a *local-variable-identifier*:

i) Let N be the *local-variable-identifier*.

ii) Search for a binding of a local variable with name N as described in 9.2.

iii) If a binding is found, replace the value of the binding with V.

iv) Otherwise, create a variable binding with name N and value V in the current set of local variable bindings.

c) The value of the *single-variable-assignment* is V.

11.4.2.2.3 Scoped constant assignments

Syntax

---

scoped-constant-assignment ::=
    scoped-constant-assignment-expression
    | *scoped-constant-assignment-statement*

scoped-constant-assignment-expression ::
    primary-expression [no *line-terminator* here] [no *whitespace* here] `::` *constant-identifier*
        [no *line-terminator* here] `=` *operator-expression*
    | `::` *constant-identifier* [no *line-terminator* here] `=` *operator-expression*

scoped-constant-assignment-statement ::
    primary-expression [no *line-terminator* here] [no *whitespace* here] `::` *constant-identifier*
        [no *line-terminator* here] `=` *method-invocation-without-parentheses*
    | `::` *constant-identifier* [no *line-terminator* here] `=` *method-invocation-without-parentheses*

Semantics

A *scoped-constant-assignment* is evaluated as follows:

a) If the *primary-expression* is present, evaluate it and let M be the resulting value. Otherwise, let M be the class `Object`.

b) If M is an instance of the class `Module`:

   1) Let N be the *constant-identifier*.

   2) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let V be the resulting value.

   3) If a binding with name N exists in the set of bindings of constants of M, replace the value of the binding with V.

   4) Otherwise, create a variable binding with name N and value V in the set of bindings of constants of M.

   5) The value of the *scoped-constant-assignment* is V.

c) If M is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

11.4.2.2.4 Single indexing assignments

Syntax

---

single-indexing-assignment ::=
  single-indexing-assignment-expression
  | *single-indexing-assignment-statement*

single-indexing-assignment-expression ::
  primary-expression [no *line-terminator* here] [no *whitespace* here] `[` *indexing-argument-list*? `]`
  [no *line-terminator* here] `=` *operator-expression*

single-indexing-assignment-statement ::
  primary-expression [no *line-terminator* here] [no *whitespace* here] `[` *indexing-argument-list*? `]`
  [no *line-terminator* here] `=` *method-invocation-without-parentheses*

Semantics

A *single-indexing-assignment* is evaluated as follows:

a) Evaluate the *primary-expression*. Let O be the resulting value.

b) Construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let L be the resulting list.

c) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let V be the resulting value.

d) Append V to L.

e) Invoke the method `[]=` on O with L as the list of arguments.

f) The value of the *single-indexing-assignment* is V.

11.4.2.2.5 Single method assignments

Syntax

---

single-method-assignment ::=
  single-method-assignment-expression
  | *single-method-assignment-statement*

single-method-assignment-expression ::
  primary-expression [no *line-terminator* here] (`.` | `::`) *local-variable-identifier*
    [no *line-terminator* here] `=` *operator-expression*
  | *primary-expression* [no *line-terminator* here] `.` *constant-identifier*
    [no *line-terminator* here] `=` *operator-expression*

single-method-assignment-statement ::
  primary-expression [no *line-terminator* here] (`.` | `::`) *local-variable-identifier*
    [no *line-terminator* here] `=` *method-invocation-without-parentheses*
  | *primary-expression* [no *line-terminator* here] `.` *constant-identifier*
    [no *line-terminator* here] `=` *method-invocation-without-parentheses*

Semantics

A *single-method-assignment* is evaluated as follows:

a) Evaluate the *primary-expression*. Let O be the resulting value.

b) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let V be the resulting value.

c) Let M be the *local-variable-identifier* or *constant-identifier*. Let N be the concatenation of M and `=`.

d) Invoke the method whose name is N on O with a list of arguments which contains only one value V.

e) The value of the *single-method-assignment* is V.

11.4.2.3 Abbreviated assignments

11.4.2.3.1 General description

Syntax

---

abbreviated-assignment ::=
    abbreviated-assignment-expression
  | *abbreviated-assignment-statement*

abbreviated-assignment-expression ::
    abbreviated-variable-assignment-expression
  | *abbreviated-indexing-assignment-expression*
  | *abbreviated-method-assignment-expression*

abbreviated-assignment-statement ::
    abbreviated-variable-assignment-statement
  | *abbreviated-indexing-assignment-statement*
  | *abbreviated-method-assignment-statement*

11.4.2.3.2 Abbreviated variable assignments

Syntax

---

abbreviated-variable-assignment ::=
    abbreviated-variable-assignment-expression
  | *abbreviated-variable-assignment-statement*

abbreviated-variable-assignment-expression ::
    variable [no *line-terminator* here] *assignment-operator* *operator-expression*

abbreviated-variable-assignment-statement ::
    variable [no *line-terminator* here] *assignment-operator*
      method-invocation-without-parentheses

Semantics

An *abbreviated-variable-assignment* is evaluated as follows:

a) Evaluate the *variable* as a variable reference (see 11.5.4). Let V be the resulting value.

b) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let W be the resulting value.

c) Let OP be the *assignment-operator-name* of the *assignment-operator*.

d) Let X be the *operator-expression* of the form V OP W.

e) Let I be the *variable* of the *abbreviated-variable-assignment-expression* or the *abbreviated-variable-assignment-statement*.

f) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is I and the *operator-expression* is X.

g) The value of the *abbreviated-variable-assignment* is the resulting value of the evaluation.

11.4.2.3.3 Abbreviated indexing assignments

Syntax

---

abbreviated-indexing-assignment ::=
    abbreviated-indexing-assignment-expression
  | *abbreviated-indexing-assignment-statement*

abbreviated-indexing-assignment-expression ::
    primary-expression [no *line-terminator* here] [no *whitespace* here] `[` *indexing-argument-list*? `]`
      [no *line-terminator* here] *assignment-operator* *operator-expression*

abbreviated-indexing-assignment-statement ::
    primary-expression [no *line-terminator* here] [no *whitespace* here] `[` *indexing-argument-list*? `]`
      [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*

Semantics

An *abbreviated-indexing-assignment* is evaluated as follows:

a) Evaluate the *primary-expression*. Let O be the resulting value.

b) Construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let L be the resulting list.

c) Invoke the method `[]` on O with L as the list of arguments. Let V be the resulting value.

d) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let W be the resulting value.

e) Let OP be the *assignment-operator-name* of the *assignment-operator*.

f) Evaluate the *operator-expression* of the form V OP W. Let X be the resulting value.

g) Append X to L.

h) Invoke the method `[]=` on O with L as the list of arguments.

i) The value of the *abbreviated-indexing-assignment* is X.

11.4.2.3.4 Abbreviated method assignments

Syntax

---

abbreviated-method-assignment ::=
    abbreviated-method-assignment-expression
  | *abbreviated-method-assignment-statement*

abbreviated-method-assignment-expression ::
    primary-expression [no *line-terminator* here] (`.` | `::`) *local-variable-identifier*
      [no *line-terminator* here] *assignment-operator* *operator-expression*
  | *primary-expression* [no *line-terminator* here] `.` *constant-identifier*
      [no *line-terminator* here] *assignment-operator* *operator-expression*

abbreviated-method-assignment-statement ::
    primary-expression [no *line-terminator* here] (`.` | `::`) *local-variable-identifier*
      [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*
  | *primary-expression* [no *line-terminator* here] `.` *constant-identifier*
      [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*

Semantics

An *abbreviated-method-assignment* is evaluated as follows:

a) Evaluate the *primary-expression*. Let O be the resulting value.

b) Create an empty list of arguments L. Invoke the method whose name is the *local-variable-identifier* or the *constant-identifier* on O with L as the list of arguments. Let V be the resulting value.

c) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let W be the resulting value.

d) Let OP be the *assignment-operator-name* of the *assignment-operator*.

e) Evaluate the *operator-expression* of the form V OP W. Let X be the resulting value.

f) Let M be the *local-variable-identifier* or the *constant-identifier*. Let N be the concatenation of M and `=`.

g) Invoke the method whose name is N on O with a list of arguments which contains only one value X.

h) The value of the *abbreviated-method-assignment* is X.

11.4.2.4 Multiple assignments

Syntax

multiple-assignment-statement ::
many-to-one-assignment-statement
| *one-to-packing-assignment-statement*
| *many-to-many-assignment-statement*

many-to-one-assignment-statement ::
left-hand-side [no *line-terminator* here] `=` *multiple-right-hand-side*

one-to-packing-assignment-statement ::
packing-left-hand-side [no *line-terminator* here] `=`
(*method-invocation-without-parentheses* | *operator-expression*)

many-to-many-assignment-statement ::
multiple-left-hand-side [no *line-terminator* here] `=` *multiple-right-hand-side*
| (*multiple-left-hand-side* **but not** *packing-left-hand-side*)
[no *line-terminator* here] `=`
(*method-invocation-without-parentheses* | *operator-expression*)

left-hand-side ::
variable
| *primary-expression* [no *line-terminator* here] [no *whitespace* here] `[` *indexing-argument-list*? `]`
| *primary-expression* [no *line-terminator* here]
(`.` | `::`) (*local-variable-identifier* | *constant-identifier*)
| `::` *constant-identifier*

multiple-left-hand-side ::
(*multiple-left-hand-side-item* [no *line-terminator* here] `,`)+ *multiple-left-hand-side-item*?
| (*multiple-left-hand-side-item* [no *line-terminator* here] `,`)+ *packing-left-hand-side*?
| *packing-left-hand-side*
| *grouped-left-hand-side*

packing-left-hand-side ::
`*` *left-hand-side*?

grouped-left-hand-side ::
`(` *multiple-left-hand-side* `)`

multiple-left-hand-side-item ::
left-hand-side
| *grouped-left-hand-side*

multiple-right-hand-side ::
operator-expression-list ([no *line-terminator* here] `,` *splatting-right-hand-side*)?

| *splatting-right-hand-side*

splatting-right-hand-side ::
splatting-argument

Semantics

A *multiple-assignment-statement* is evaluated as follows:

a) A *many-to-one-assignment-statement* is evaluated as follows:

1) Construct a list of values *L* from the *multiple-right-hand-side* as described below.

i) If the *operator-expression-list* is present, evaluate its *operator-expression*s in the order they appear in the program text. Let *L1* be a list which contains the resulting values, preserving their order.

ii) If the *operator-expression-list* is omitted, create an empty list of values *L1*.

iii) If the *splatting-right-hand-side* is present, construct a list of values from its *splatting-argument* as described in 11.3.2 and let *L2* be the resulting list.

iv) If the *splatting-right-hand-side* is omitted, create an empty list of values *L2*.

v) The result is the concatenation of *L1* and *L2*.

2) If the length of *L* is 0 or 1, let *A* be an implementation-defined value.

3) If the length of *L* is larger than 1, create a direct instance of the class `Array` and store the elements of *L* in it, preserving their order. Let *A* be the instance of the class `Array`.

4) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is the *left-hand-side* and the value of its *operator-expression* is *A*.

5) The value of the *many-to-one-assignment-statement* is *A*.

b) A *one-to-packing-assignment-statement* is evaluated as follows:

1) Evaluate the *method-invocation-without-parentheses* or the *operator-expression*. Let *V* be the resulting value.

2) If *V* is an instance of the class `Array`, let *A* be a a new direct instance of the class `Array` which contains only one element *V* itself, or all the elements of *V* in the same order in *V*. Which is chosen is implementation-defined.

3) If *V* is not an instance of the class `Array`, create a direct instance *A* of the class `Array` which contains only one value *V*.

4) If the *left-hand-side* of the *packing-left-hand-side* is present, evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is the *left-hand-side* and the value of the *operator-expression* is *A*. Otherwise, skip this step.

5) The value of the *one-to-packing-assignment-statement* is A.

c) A *many-to-many-assignment-statement* is evaluated as follows:

1) If the *multiple-right-hand-side* is present, construct a list of values from it [see a) 1)] and let R be the resulting list.

2) If the *multiple-right-hand-side* is omitted:

i) Evaluate the *method-invocation-without-parentheses* or the *operator-expression*. Let V be the resulting value.

ii) If V is not an instance of the class `Array`, the behavior is unspecified.

iii) Create a list of arguments R which contains all the elements of V, preserving their order.

3) i) Create an empty list of variables L.

ii) For each *multiple-left-hand-side-item*, in the order they appear in the program text, append the *left-hand-side* or the *grouped-left-hand-side* of the *multiple-left-hand-side-item* to L.

iii) If the *packing-left-hand-side* of the *multiple-left-hand-side* is present, append it to L.

iv) If the *multiple-left-hand-side* is a *grouped-left-hand-side*, append the *grouped-left-hand-side* to L.

4) For each element L_i of L, in the same order in L, take the following steps:

i) Let i be the index of L_i within L. Let N_R be the number of elements of R.

ii) If L_i is a *left-hand-side*:

I) If i is larger than N_R, let V be **nil**.

II) Otherwise, let V be the ith element of R.

III) Evaluate the *single-variable-assignment* of the form L_i `=` V.

iii) If L_i is a *packing-left-hand-side* and its *left-hand-side* is present:

I) If i is larger than N_R, create an empty direct instance of the class `Array`. Let A be the instance.

II) Otherwise, create a direct instance of the class `Array` which contains elements in R whose index is equal to, or larger than i, in the same order they are stored in R. Let A be the instance.

III) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is the *left-hand-side* and the value of the *operator-expression* is A.

iv) If L_i is a *grouped-left-hand-side*:

I) If i is larger than N_R, let V be **nil**.

II) Otherwise, let V be the ith element of R.

III) Evaluate a *many-to-many-assignment-statement* where its *multiple-left-hand-side* is the *multiple-left-hand-side* of the *grouped-left-hand-side* and its *multiple-right-hand-side* is V.

11.4.2.5 Assignments with `rescue` modifiers

Syntax

---

assignment-with-rescue-modifier ::
  left-hand-side [no *line-terminator* here] `=`
   operator-expression$_1$ [no *line-terminator* here] `rescue` *operator-expression*$_2$

Semantics

An *assignment-with-rescue-modifier* is evaluated as follows:

a) Evaluate the *operator-expression*$_1$. Let V be the resulting value.

b) If an exception is raised and not handled during the evaluation of the *operator-expression*$_1$, and if the exception is an instance of the class `StandardError`, evaluate the *operator-expression*$_2$ and replace V with the resulting value.

c) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is the *left-hand-side* and the value of the *operator-expression* is V. The value of the *assignment-with-rescue-modifier* is the resulting value of the evaluation.

11.4.3 Unary operator expressions

11.4.3.1 General description

Syntax

---

unary-operator-expression ::=
  unary-minus-expression
  | *unary-expression*

unary-minus-expression ::
  power-expression
  | `-` *power-expression*

unary-expression ::
  primary-expression

| `~` *unary-expression* $_1$
| `+` *unary-expression* $_2$
| `!` *unary-expression* $_3$

Semantics

A *unary-operator-expression* is evaluated as follows:

a) A *unary-minus-expression* of the form *power-expression* is evaluated as described in 11.4.4 e).

b) A *unary-minus-expression* of the form `-` *power-expression* is evaluated as follows:

   1) Evaluate the *power-expression*. Let X be the resulting value.

   2) Create an empty list of arguments L. Invoke the method `-@` on X with L as the list of arguments. The value of the *unary-expression* is the resulting value of the invocation.

c) A *unary-expression* of the form `~` *unary-expression*$_1$ is evaluated as follows:

   1) Evaluate the *unary-expression*$_1$. Let X be the resulting value.

   2) Create an empty list of arguments L. Invoke the method `~` on X with L as the list of arguments. The value of the *unary-expression* is the resulting value of the invocation.

d) A *unary-expression* of the form `+` *unary-expression*$_2$ is evaluated as follows:

   1) Evaluate the *unary-expression*$_2$. Let X be the resulting value.

   2) Create an empty list of arguments L. Invoke the method `+@` on X with L as the list of arguments. The value of the *unary-expression*$_2$ is the resulting value of the invocation.

e) A *unary-expression* of the form `!` *unary-expression*$_3$ is evaluated as described in 11.2.

11.4.3.2 The `defined?` expression

Syntax

---

defined?-expression ::=
    defined?-with-parentheses
  | *defined?-without-parentheses*

defined?-with-parentheses ::
    `defined?` `(` *expression* `)`

defined?-without-parentheses ::
    `defined?` *operator-expression*

Semantics

A *defined?-expression* is evaluated as follows:

a) Let E be the *expression* of the *defined?-with-parentheses* or the *operator-expression* of the *defined?-without-parentheses*.

b) If E is a *constant-identifier*:

   1) Search for a binding of a constant with name E with the same evaluation steps for *constant-identifier* as described in 11.5.4.2. However, a direct instance of the class `NameError` shall not be raised when a binding is not found.

   2) If a binding is found, the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

   3) Otherwise, the value of the *defined?-expression* is **nil**.

c) If E is a *global-variable-identifier*:

   1) If a binding with name E exists in ⟦global-variable-bindings⟧ , the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

   2) Otherwise, the value of the *defined?-expression* is **nil**.

d) If E is a *class-variable-identifier*:

   1) Let C be the current class or module. Let *CS* be the set of classes which consists of C and all the superclasses of C. Let *MS* be the set of modules which consists of all the modules in the included module lists of all classes in *CS*. Let *CM* be the union of *CS* and *MS*.

   2) If any of the classes or modules in *CM* has a binding with name E in the set of bindings of class variables, the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

   3) Otherwise, the value of the *defined?-expression* is **nil**.

e) If E is an *instance-variable-identifier*:

   1) If a binding with name E exists in the set of bindings of instance variables of the current self, the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

   2) Otherwise, the value of the *defined?-expression* is **nil**.

f) If E is a *local-variable-identifier*:

   1) If the *local-variable-identifier* is a reference to a local variable (see 11.5.4.7.2), the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

2) Otherwise, search for a method binding with name E, starting from the current class or module as described in 13.3.4.

i) If the binding is found and its value is not undef, the value of the *defined?-expression* is an implementation-defined value, which shall be a trueish object.

ii) Otherwise, the value of the *defined?-expression* is **nil**.

g) Otherwise, the value of the *defined?-expression* is implementation-defined.

11.4.4 Binary operator expressions

Syntax

---

binary-operator-expression ::=
    equality-expression

equality-expression ::
    relational-expression
    | *relational-expression* [no *line-terminator* here] `<=>` *relational-expression*
    | *relational-expression* [no *line-terminator* here] `==` *relational-expression*
    | *relational-expression* [no *line-terminator* here] `===` *relational-expression*
    | *relational-expression* [no *line-terminator* here] `!=` *relational-expression*
    | *relational-expression* [no *line-terminator* here] `=~` *relational-expression*
    | *relational-expression* [no *line-terminator* here] `!~` *relational-expression*

relational-expression ::
    bitwise-OR-expression
    | *relational-expression* [no *line-terminator* here] `>` *bitwise-OR-expression*
    | *relational-expression* [no *line-terminator* here] `>=` *bitwise-OR-expression*
    | *relational-expression* [no *line-terminator* here] `<` *bitwise-OR-expression*
    | *relational-expression* [no *line-terminator* here] `<=` *bitwise-OR-expression*

bitwise-OR-expression ::
    bitwise-AND-expression
    | *bitwise-OR-expression* [no *line-terminator* here] `|` *bitwise-AND-expression*
    | *bitwise-OR-expression* [no *line-terminator* here] `^` *bitwise-AND-expression*

bitwise-AND-expression ::
    bitwise-shift-expression
    | *bitwise-AND-expression* [no *line-terminator* here] `&` *bitwise-shift-expression*

bitwise-shift-expression ::
    additive-expression
    | *bitwise-shift-expression* [no *line-terminator* here] `<<` *additive-expression*
    | *bitwise-shift-expression* [no *line-terminator* here] `>>` *additive-expression*

additive-expression ::
    multiplicative-expression
  | *additive-expression* [no *line-terminator* here] `+` *multiplicative-expression*
  | *additive-expression* [no *line-terminator* here] `-` *multiplicative-expression*

multiplicative-expression ::
    unary-minus-expression
  | *multiplicative-expression* [no *line-terminator* here] `*` *unary-minus-expression*
  | *multiplicative-expression* [no *line-terminator* here] `/` *unary-minus-expression*
  | *multiplicative-expression* [no *line-terminator* here] `%` *unary-minus-expression*

power-expression ::
    unary-expression
  | *unary-expression* [no *line-terminator* here] `**` *power-expression*

binary-operator ::=
    `<=>` | `==` | `!=` | `===` | `=~` | `!~` | `>` | `>=` | `<` | `<=`
  | `|` | `^` | `&` | `<<` | `>>` | `+` | `-` | `*` | `/` | `%` | `**`

If there is a *whitespace* character just before any of the following operators, there shall be one or more *whitespace* characters just after the operator.

- `&` of a *bitwise-AND-expression*
- `<<` of a *bitwise-shift-expression*
- `+` of a *additive-expression*
- `-` of a *additive-expression*
- `*` of a *multiplicative-expression*
- `/` of a *multiplicative-expression*
- `%` of a *multiplicative-expression*

NOTE This constraint is necessary to distinguish binary operators from leading sequences of characters of *argument-without-parentheses*es (see 11.3.2).

Semantics

An *equality-expression* is evaluated as follows:

a) If the *equality-expression* is of the form x `!=` y, take the following steps:

  1) Evaluate x. Let X be the resulting value.

  2) Evaluate y. Let Y be the resulting value.

3) Invoke the method `==` on X with Y as an argument. If the resulting value is a trueish object, the value of the *equality-expression* is **false**. Otherwise, the value of the *equality-expression* is **true**.

b) The steps in Step f) may be taken instead of Step a).

In this case, the following conditions shall be satisfied:

- The operator `!=` is included in *operator-method-name*.
- An instance method `!=` is defined in the class `Object`, one of its superclasses, or a module included in the class `Object`. The method `!=` shall take one argument and shall return the value of the *equality-expression* in Step a) 3), where let X and Y be the receiver and the argument, respectively.

c) If the *equality-expression* is of the form x `!~` y, take the following steps:

1) Evaluate x. Let X be the resulting value.

2) Evaluate y. Let Y be the resulting value.

3) Invoke the method `=~` on X with Y as an argument. If the resulting value is a trueish object, the value of the *equality-expression* is **false**. Otherwise, the value of the *equality-expression* is **true**.

d) The steps in Step f) may be taken instead of Step c). In this case, the following conditions shall be satisfied:

- The operator `!~` is included in *operator-method-name*.
- An instance method `!~` is defined in the class `Object`, one of its superclasses, or a module included in the class `Object`. The method `!~` shall take one argument and shall return the value of the *equality-expression* in Step c) 3), where let X and Y be the receiver and the argument, respectively.

e) If the *equality-expression* is an *unary-minus-expression* and not a *power-expression*, evaluate it as described in 11.4.3. If the *equality-expression* is an *unary-minus-expression* and a *power-expression*, evaluate the *power-expression* by taking the following steps and the resulting value is the value of the *equality-expression*.

1) If the *power-expression* is a *unary-expression*, evaluate it as described in 11.4.3 and the resulting value is the value of the *power-expression*.

2) If the *power-expression* is a *power-expression* of the form *unary-expression* `**` *power-expression*:

i) If the *unary-expression* is of the form `-` *unsigned-number*:

I) Evaluate the *unsigned-number* and let X be the resulting value.

II) Evaluate the *power-expression*$_2$ and let Y be the resulting value.

III) Invoke the method whose name is "**" on X with Y as an argument. Let Z be the resulting value.

IV) Invoke the method whose name is "-@" on Z with no arguments. The value of the *equality-expression* is the resulting value of the invocation.

ii) Otherwise:

I) Evaluate the *unary-expression* and let X be the resulting value.

II) Evaluate the *power-expression* and let Y be the resulting value.

III) Invoke the method whose name is "**" on X with Y as an argument. The value of the *power-expression* is the resulting value.

f) Otherwise, for the *equality-expression* of the form *x binary-operator y*, take the following steps:

1) Evaluate *x*. Let X be the resulting value.

2) Evaluate *y*. Let Y be the resulting value.

3) Invoke the method whose name is the *binary-operator* on X with Y as an argument. The value of the *equality-expression* is the resulting value of the invocation.

11.5 Primary expressions

11.5.1 General description

Syntax

---

primary-expression ::
  class-definition
  | *singleton-class-definition*
  | *module-definition*
  | *method-definition*
  | *singleton-method-definition*
  | *yield-with-optional-argument*
  | *if-expression*
  | *unless-expression*
  | *case-expression*
  | *while-expression*
  | *until-expression*
  | *for-expression*
  | *return-without-argument*
  | *break-without-argument*
  | *next-without-argument*
  | *redo-expression*
  | *retry-expression*
  | *begin-expression*
  | *grouping-expression*

| *variable-reference*
| *scoped-constant-reference*
| *array-constructor*
| *hash-constructor*
| *literal*
| *defined?-with-parentheses*
| *primary-method-invocation*

Semantics

See 13.2.2 for *class-definition*s.

See 13.4.2 for *singleton-class-definition*s.

See 13.1.2 for *module-definition*s.

See 13.3.1 for *method-definition*s.

See 13.4.3 for *singleton-method-definition*s.

See 11.3.5 for *yield-with-optional-argument*s.

See 8.7.6 for *literal*s.

See 11.4.3.2 for *defined?-with-parentheses*s.

See 11.3 for *primary-method-invocation*s.

11.5.2 Control structures

11.5.2.1 General description

Syntax

---

control-structure ::=
conditional-expression
| *iteration-expression*
| *jump-expression*
| *begin-expression*

11.5.2.2 Conditional expressions

11.5.2.2.1 General description

Syntax

---

conditional-expression ::=
if-expression

| *unless-expression*
| *case-expression*
| *conditional-operator-expression*

11.5.2.2.2 The `if` expression

Syntax

---

if-expression ::
  `if` *expression* *then-clause* *elsif-clause** *else-clause*? `end`

then-clause ::
  separator *compound-statement*
  | *separator*? `then` *compound-statement*

else-clause ::
  `else` *compound-statement*

elsif-clause ::
  `elsif` *expression* *then-clause*

Semantics

The *if-expression* is evaluated as follows:

a) Evaluate *expression*. Let V be the resulting value.

b) If V is a trueish object, evaluate the *compound-statement* of the *then-clause*. The value of the *if-expression* is the resulting value. In this case, *elsif-clause*s and the *else-clause*, if any, are not evaluated.

c) If V is a falseish object, and if there is no *elsif-clause* and no *else-clause*, then the value of the *if-expression* is **nil**.

d) If V is a falseish object, and if there is no *elsif-clause* but there is an *else-clause*, then evaluate the *compound-statement* of the *else-clause*. The value of the *if-expression* is the resulting value.

e) If V is a falseish object, and if there are one or more *elsif-clause*s, evaluate the sequence of *elsif-clause*s as follows:

  1) Evaluate the *expression* of each *elsif-clause* in the order they appear in the program text, until there is an *elsif-clause* for which *expression* evaluates to a trueish object. Let T be this *elsif-clause*.

2) If T exists, evaluate the *compound-statement* of its *then-clause*. The value of the *if-expression* is the resulting value. Other *elsif-clauses* and an *else-clause* following T, if any, are not evaluated.

3) If T does not exist, and if there is an *else-clause*, then evaluate the *compound-statement* of the *else-clause*. The value of the *if-expression* is the resulting value.

4) If T does not exist, and if there is no *else-clause*, then the value of the *if-expression* is **nil**.

11.5.2.2.3 The `unless` expression

Syntax

---

unless-expression ::
  `unless` *expression* *then-clause* *else-clause*$^?$ `end`

Semantics

The *unless-expression* is evaluated as follows:

a) Evaluate the *expression*. Let V be the resulting value.

b) If V is a falseish object, evaluate the *compound-statement* of the *then-clause*. The value of the *unless-expression* is the resulting value. In this case, the *else-clause*, if any, is not evaluated.

c) If V is a trueish object, and if there is no *else-clause*, then the value of the *unless-expression* is **nil**.

d) If V is a trueish object, and if there is an *else-clause*, then evaluate the *compound-statement* of the *else-clause*. The value of the *unless-expression* is the resulting value.

11.5.2.2.4 The `case` expression

Syntax

---

case-expression ::
  case-expression-with-expression
  | *case-expression-without-expression*

case-expression-with-expression ::
  `case` *expression* *separator-list*$^?$ *when-clause*$^+$ *else-clause*$^?$ `end`

case-expression-without-expression ::
  `case` *separator-list*$^?$ *when-clause*$^+$ *else-clause*$^?$ `end`

when-clause ::
  `when` *when-argument* *then-clause*

when-argument ::
  operator-expression-list ([no *line-terminator* here] **,** *splatting-argument*)?
  | *splatting-argument*

Semantics

A *case-expression* is evaluated as follows:

a) If the *case-expression* is a *case-expression-with-expression*, evaluate the *expression*. Let V be the resulting value.

b) The meaning of the phrase "O is matching" in Step c) is defined as follows:

   1) If the *case-expression* is a *case-expression-with-expression*, invoke the method `===` on O with a list of arguments which contains only one value V. O is matching if and only if the resulting value is a trueish object.

   2) If the *case-expression* is a *case-expression-without-expression*, O is matching if and only if O is a trueish object.

c) Take the following steps:

   1) Search the *when-clause*s in the order they appear in the program text for a matching *when-clause* as follows:

      i) If the *operator-expression-list* of the *when-argument* is present:

         I) For each of its *operator-expression*s, evaluate it and test if the resulting value is matching.

         II) If a matching value is found, other *operator-expression*s, if any, are not evaluated.

      ii) If no matching value is found, and the *splatting-argument*(see 11.3.2) is present:

         I) Construct a list of values from it as described in 11.3.2. For each element of the resulting list, in the same order in the list, test if it is matching.

         II) If a matching value is found, other values, if any, are not evaluated.

      iii) A *when-clause* is considered to be matching if and only if a matching value is found in its *when-argument*. Later *when-clause*s, if any, are not tested in this case.

   2) If one of the *when-clause*s is matching, evaluate the *compound-statement* of the *then-clause* of this *when-clause*. The value of the *case-expression* is the resulting value.

3) If none of the *when-clause*s is matching, and if there is an *else-clause*, then evaluate the *compound-statement* of the *else-clause*. The value of the *case-expression* is the resulting value.

4) Otherwise, the value of the *case-expression* is **nil**.

11.5.2.2.5 Conditional operator expression

Syntax

---

conditional-operator-expression ::
    range-constructor
   | *range-constructor* [no *line-terminator* here] `?` *operator-expression*$_1$ [no *line-terminator* here] `:` *operator-expression*$_2$

Semantics

A *conditional-operator-expression* of the form *range-constructor* `?` *operator-expression*$_1$ `:` *operator-expression*$_2$ is evaluated as follows:

a) Evaluate the *range-constructor*.

b) If the resulting value is a trueish object, evaluate the *operator-expression*$_1$. The value of the *conditional-operator-expression* is the resulting value of the evaluation.

c) Otherwise, evaluate the *operator-expression*$_2$. The value of the *conditional-operator-expression* is the resulting value of the evaluation.

11.5.2.3 Iteration expressions

11.5.2.3.1 General description

Syntax

---

iteration-expression ::=
    while-expression
   | *until-expression*
   | *for-expression*
   | *while-modifier-statement*
   | *until-modifier-statement*

Each *iteration-expression* has a ***condition expression*** and a ***body***.

The condition expression of an *iteration-expression* is the *iteration-expression*'s part evaluated to determine the condition of the iteration of the *iteration-expression*. The condition expression of a *while-expression* (see 11.5.2.3.2), *until-expression* (see 11.5.2.3.3), *for-expression* (see 11.5.2.3.4), *while-modifier-statement* (see 12.5) or *until-modifier-statement* (see 12.6) is its *expression*.

The body of an *iteration-expression* is the *iteration-expression*'s part evaluated iteratively. The body of a *while-expression*, *until-expression*, or *for-expression* is its *compound-statement*. The body of a *while-modifier-statement* or *until-modifier-statement* is its *statement*.

See 12.5 for *while-modifier-statement*s.

See 12.6 for *until-modifier-statement*s.

11.5.2.3.2 The `while` expression

Syntax

---

while-expression ::
    `while` *expression* *do-clause* `end`

do-clause ::
    separator *compound-statement*
  | [no *line-terminator* here] `do` *compound-statement*

Semantics

A *while-expression* is evaluated as follows:

a) Evaluate the *expression*, and take the following steps:

   1) If the evaluation of the *expression* is terminated by a *break-expression* (see 11.5.2.4.3), terminate the evaluation of the *while-expression*.

      If the *jump-argument* of the *break-expression* is present, the value of the *while-expression* is the value of the *jump-argument*. Otherwise, the value of the *while-expression* is **nil**.

   2) If the evaluation of the *expression* is terminated by a *next-expression* (see 11.5.2.4.4) or *redo-expression* (see 11.5.2.4.5), continue processing from the beginning of Step a).

   3) Otherwise, let V be the resulting value of the *expression*.

b) If V is a falseish object, terminate the evaluation of the *while-expression*. The value of the *while-expression* is **nil**.

c) If V is a trueish object, evaluate the *compound-statement* of the *do-clause*, and take the following steps:

   1) If the evaluation of the *compound-statement* is terminated by a *break-expression*, terminate the evaluation of the *while-expression*.

      If the *jump-argument* of the *break-expression* is present, the value of the *while-expression* is the value of the *jump-argument*. Otherwise, the value of the *while-expression* is **nil**.

   2) If the evaluation of the *compound-statement* is terminated by a *next-expression*, continue processing from Step a).

3) If the evaluation of the *compound-statement* is terminated by a *redo-expression*, continue processing from Step c).

4) Otherwise, continue processing from Step a).

11.5.2.3.3 The `until` expression

Syntax

---

until-expression ::
    `until` *expression* *do-clause* `end`

Semantics

An *until-expression* is evaluated as follows:

a) Evaluate the *expression*, and take the following steps:

1) If the evaluation of the *expression* is terminated by a *break-expression* (see 11.5.2.4.3), terminate the evaluation of the *until-expression*.

If the *jump-argument* of the *break-expression* is present, the value of the *until-expression* is the value of the *jump-argument*. Otherwise, the value of the *until-expression* is **nil**.

2) If the evaluation of the *expression* is terminated by a *next-expression* (see 11.5.2.4.4) or *redo-expression* (see 11.5.2.4.5), continue processing from the beginning of Step a).

3) Otherwise, let V be the resulting value of the *expression*.

b) If V is a trueish object, terminate the evaluation of the *until-expression*. The value of the *until-expression* is **nil**.

c) If V is a falseish object, evaluate the *compound-statement* of the *do-clause*, and take the following steps:

1) If the evaluation of the *compound-statement* is terminated by a *break-expression*, terminate the evaluation of the *until-expression*.

If the *jump-argument* of the *break-expression* is present, the value of the *until-expression* is the value of the *jump-argument*. Otherwise, the value of the *until-expression* is **nil**.

2) If the evaluation of the *compound-statement* is terminated by a *next-expression*, continue processing from Step a).

3) If the evaluation of the *compound-statement* is terminated by a *redo-expression*, continue processing from Step c).

4) Otherwise, continue processing from Step a).

11.5.2.3.4 The `for` expression

Syntax

---

for-expression ::
    `for` *for-variable* [no *line-terminator* here] `in` *expression* *do-clause* `end`

for-variable ::
    left-hand-side
  | *multiple-left-hand-side*

Semantics

A *for-expression* is evaluated as follows:

a) Evaluate the *expression*. If the evaluation of the *expression* is terminated by a *break-expression*, *next-expression*, or *redo-expression*, the behavior is unspecified. Otherwise, let O be the resulting value.

b) Let E be the *primary-method-invocation* of the form *primary-expression* [no *line-terminator* here] `. each do |` *block-parameter-list* `|` *block-body* `end`, where the value of the *primary-expression* is O, the *block-parameter-list* is the *for-variable*, the *block-body* is the *compound-statement* of the *do-clause*.

   Evaluate E; however, if a block whose *block-body* is the *compound-statement* of the *do-clause* of the *for-expression* is called during this evaluation, the steps in 11.3.3 except the Step c) and the Step e) 4) shall be taken for the evaluation of this call.

c) The value of the *for-expression* is the resulting value of the evaluation.

11.5.2.4 Jump expressions

11.5.2.4.1 General description

Syntax

---

jump-expression ::=
    return-expression
  | *break-expression*
  | *next-expression*
  | *redo-expression*
  | *retry-expression*

Semantics

*jump-expression*s are used to terminate the evaluation of a *method-body*, a *block-body*, the body of an *iteration-expression*, or the *compound-statement*$_2$ of a *rescue-clause*. The evaluation of the program construct terminated by a *jump-expression* and the evaluations of program constructs in the program construct which are under evaluation when the evaluation of the *jump-expression* has been started are terminated in the middle of the evaluation steps, and have no resulting values.

In this document, the ***current block*** or the ***current iteration-expression*** refers to the following:

a) If the current method invocation does not exist, the *block* or *iteration-expression* whose evaluation has been started most recently among *block*s and *iteration-expression*s which are under evaluation.

b) If the current method invocation exists, the *block* or *iteration-expression* whose evaluation has been started most recently among *block*s and *iteration-expression*s which are under evaluation and whose evaluation has been started during the evaluation of the current method invocation.

In the both cases, the current *block* or the current *iteration-expression* does not exist if such a *block* or *iteration-expression* does not exist.

11.5.2.4.2 The `return` expression

Syntax

---

return-expression ::=
  return-without-argument
  | *return-with-argument*

return-without-argument ::
  `return`

return-with-argument ::
  `return` *jump-argument*

jump-argument ::
  [no *line-terminator* here] *argument-list*

The *block-argument* of the *argument-list* (see 11.3.2) of a *jump-argument* shall be omitted.

Semantics

*return-expression*s and *jump-argument*s are evaluated as follows:

a) A *return-expression* is evaluated as follows:

   1) Let M be the *method-body* which corresponds to the current method invocation. If such an invocation does not exist, or has already terminated:

      i) Let S be a direct instance of the class `Symbol` with name `return`.

ii) If the *jump-argument* of the *return-expression* is present, let V be the value of the *jump-argument*. Otherwise, let V be **nil**.

iii) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value V.

2) Evaluate the *jump-argument*, if any, as described in Step b).

3) If there are *block-body*s which include the *return-expression* and are included in M, terminate the evaluations of such *block-body*s, from innermost to outermost (see 11.3.3).

4) Terminate the evaluation of M (see 13.3.3).

b) A *jump-argument* is evaluated as follows:

1) If the *jump-argument* is a *splatting-argument*:

i) Construct a list of values from the *splatting-argument* as described in 11.3.2 and let L be the resulting list.

ii) If the length of L is 0 or 1, the value of the *jump-argument* is an implementation-defined value.

iii) If the length of L is larger than 1, create a direct instance of the class `Array` and store the elements of L in it, preserving their order. The value of the *jump-argument* is the instance.

2) Otherwise:

i) Construct a list of values from the *argument-list* as described in 11.3.2 and let L be the resulting list.

ii) If the length of L is 1, the value of the *jump-argument* is the only element of L.

iii) If the length of L is larger than 1, create a direct instance of the class `Array` and store the elements of L in it, preserving their order. The value of the *jump-argument* is the instance of the class `Array`.

11.5.2.4.3 The `break` expression

Syntax

---

break-expression ::=
  break-without-argument
  | *break-with-argument*

break-without-argument ::
  `break`

break-with-argument ::
    `break` *jump-argument*

Semantics

A *break-expression* is evaluated as follows:

a) Evaluate the *jump-argument*, if any, as described in 11.5.2.4.2 b).

b) If the current *block* is present, terminate the evaluation of the *block-body* of the current *block* (see 11.3.3).

c) If the current *iteration-expression* is present, terminate the evaluation of the condition expression of the current *iteration-expression* (see 11.5.2.3) when the *break-expression* is in the condition expression, or terminate the body of the current *iteration-expression* when the *break-expression* is in the body.

d) If the current *block* or the current *iteration-expression* is not present:

   1) Let S be a direct instance of the class `Symbol` with name `break`.

   2) If the *jump-argument* of the *break-expression* is present, let V be the value of the *jump-argument*. Otherwise, let V be **nil**.

   3) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value V.

11.5.2.4.4 The `next` expression

Syntax

---

next-expression ::=
    next-without-argument
  | *next-with-argument*

next-without-argument ::
    `next`

next-with-argument ::
    `next` *jump-argument*

Semantics

A *next-expression* is evaluated as follows:

a) Evaluate the *jump-argument*, if any, as described in 11.5.2.4.2 b).

b) If the current *block* is present, terminate the evaluation of the *block-body* of the current *block* (see 11.3.3).

c) If the current *iteration-expression* is present, terminate the evaluation of the condition expression of the current *iteration-expression* (see 11.5.2.3) when the *next-expression* is in the condition expression, or terminate the body of the current *iteration-expression* when the *next-expression* is in the body.

d) If the current *block* or the current *iteration-expression* is not present:

   1) Let S be a direct instance of the class `Symbol` with name `next`.

   2) If the *jump-argument* of the *next-expression* is present, let V be the value of the *jump-argument*. Otherwise, let V be **nil**.

   3) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value V.

11.5.2.4.5 The `redo` expression

Syntax

---

redo-expression ::
  `redo`

Semantics

A *redo-expression* is evaluated as follows:

a) If the current *block* is present, terminate the evaluation of the *block-body* of the current *block* (see 11.3.3).

b) If the current *iteration-expression* is present, terminate the evaluation of the condition expression of the current *iteration-expression* (see 11.5.2.3) when the *redo-expression* is in the condition expression, or terminate the body of the current *iteration-expression* when the *redo-expression* is in the body.

c) If the current *block* or the current *iteration-expression* is not present:

   1) Let S be a direct instance of the class `Symbol` with name `redo`.

   2) Raise a direct instance of the class `LocalJumpError` which has two instance variable bindings, one named `@reason` with the value S and the other named `@exit_value` with the value **nil**.

11.5.2.4.6 The `retry` expression

Syntax

retry-expression ::
    `retry`

Semantics

A *retry-expression* is evaluated as follows:

a) If the current method invocation (see 13.3.3) exists, let M be the *method-body* which corresponds to the current method invocation. Otherwise, let M be the *program* (see 10.1).

b) Let E be the innermost *rescue-clause* in M which encloses the *retry-expression*. If such a *rescue-clause* does not exist, the behavior is unspecified.

c) Terminate the evaluation of the *compound-statement* of the *then-clause* of E (see 11.5.2.5).

11.5.2.5 The `begin` expression

Syntax

begin-expression ::
    `begin` *body-statement* `end`

body-statement ::
    compound-statement rescue-clause* *else-clause*$^?$ *ensure-clause*$^?$

rescue-clause ::
    `rescue` [no *line-terminator* here] *exception-class-list*$^?$
      exception-variable-assignment$^?$ *then-clause*

exception-class-list ::
    operator-expression
  | *multiple-right-hand-side*

exception-variable-assignment ::
    `=>` *left-hand-side*

ensure-clause ::
    `ensure` *compound-statement*

Semantics

The value of a *begin-expression* is the resulting value of the *body-statement*.

A *body-statement* is evaluated as follows:

a) Evaluate the *compound-statement* of the *body-statement*.

b) If no exception is raised, or all the raised exceptions are handled during Step a):

   1) If the *else-clause* is present, evaluate the *else-clause* as described in 11.5.2.2.2.

   2) If the *ensure-clause* is present, evaluate its *compound-statement*. The value of the *ensure-clause* is the value of this evaluation.

   3) If both the *else-clause* and the *ensure-clause* are present, the value of the *body-statement* is the value of the *ensure-clause*. If only one of these clauses is present, the value of the *body-statement* is the value of the clause.

   4) If neither the *else-clause* nor the *ensure-clause* is present, the value of the *body-statement* is the value of its *compound-statement*.

c) If an exception is raised and not handled during Step a), test each *rescue-clause*, if any, in the order it occurs in the program text. The test determines whether the *rescue-clause* can handle the exception as follows:

   1) Let E be the exception raised.

   2) If the *exception-class-list* is omitted in the *rescue-clause*, and if E is an instance of the class `StandardError`, the *rescue-clause* handles E.

   3) If the *exception-class-list* of the *rescue-clause* is present:

      i) If the *exception-class-list* is of the form *operator-expression*, evaluate the *operator-expression*. Create an empty list of values, and append the value of the *operator-expression* to the list.

      ii) If the *exception-class-list* is of the form *multiple-right-hand-side*, construct a list of values from the *multiple-right-hand-side* (see 11.4.2.4).

      iii) Let L be the list created by evaluating the *exception-class-list* as above. For each element D of L:

         I) If D is neither the class `Exception` nor a subclass of the class `Exception`, raise a direct instance of the class `TypeError`.

         II) If E is an instance of D, the *rescue-clause* handles E. In this case, any remaining *rescue-clauses* in the *body-statement* are not tested.

d) If a *rescue-clause* R which can handle E is found:

   1) If the *exception-variable-assignment* of R is present, evaluate it in the same way as a *multiple-assignment-statement* of the form *left-hand-side* = *multiple-right-hand-side* where the value of *multiple-right-hand-side* is E.

   2) Evaluate the *compound-statement* of the *then-clause* of R. If this evaluation is terminated by a *retry-expression*, continue processing from Step a). Otherwise, let V be the value of this evaluation.

3) If the *ensure-clause* is present, evaluate it. The value of the *body-statement* is the value of the *ensure-clause*.

4) If the *ensure-clause* is omitted, the value of the *body-statement* is V.

e) If no *rescue-clause* is present or if a *rescue-clause* which can handle E is not found:

1) If the *ensure-clause* is present, evaluate it.

2) The value of the *body-statement* is unspecified.

The *ensure-clause* of a *body-statement*, if any, is always evaluated, even when the evaluation of *body-statement* is terminated by a *jump-expression*.

11.5.3 Grouping expression

Syntax

---

grouping-expression ::
    (*expression*)
  | (*compound-statement*)

Semantics

A *grouping-expression* is evaluated as follows:

a) Evaluate the *expression* or the *compound-statement*.

b) The value of the *grouping-expression* is the resulting value.

11.5.4 Variable references

11.5.4.1 General description

Syntax

---

variable-reference ::
    variable
  | *pseudo-variable*

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

scoped-constant-reference ::
    primary-expression [no *line-terminator* here] [no *whitespace* here] :: *constant-*

identifier
| :: *constant-identifier*

See from 11.5.4.2 to 11.5.4.7 for *variable* and *scoped-constant-reference*s.

See 11.5.4.8 for *pseudo-variable*s.

11.5.4.2 Constants

A *constant-identifier* is evaluated as follows:

a) Let N be the *constant-identifier*.

b) Search for a binding of a constant with name N as described below.

   As soon as the binding is found in any of the following steps, the evaluation of the *constant-identifier* is terminated and the value of the *constant-identifier* is the value of the binding found.

c) Let L be the top of ⟦class-module-list⟧ . Search for a binding of a constant with name N in each element of L from start to end, including the first element, which is the current class or module, but except for the last element, which is the class `Object`.

d) If a binding is not found, let C be the current class or module.

   Let L be the included module list of C. Search each element of L in the reverse order for a binding of a constant with name N.

e) If the binding is not found:

   1) If C is an instance of the class `Class`:

      i) If C does not have a direct superclass, create a direct instance of the class `Symbol` with name N, and let R be that instance. Invoke the method `const_missing` on the current class or module with R as the only argument.

      ii) Let S be the direct superclass of C.

      iii) Search for a binding of a constant with name N in S.

      iv) If the binding is not found, let L be the included module list of S and search each element of L in the reverse order for a binding of a constant with name N.

      v) If the binding is not found, let C be the direct superclass of S. Continue processing from Step e) 1) i).

   2) If C is not an instance of the class `Class`:

      i) Search for a binding of a constant with name N in the class `Object`.

ii) If the binding is not found, let L be the included module list of the class `Object` and search each element of L in the reverse order for a binding of a constant with name N.

iii) If the binding is not found, create a direct instance of the class `Symbol` with name N, and let R be that instance. Invoke the method `const_missing` on the current class or module with R as the only argument.

11.5.4.3 Scoped constants

A *scoped-constant-reference* is evaluated as follows:

a) If the *primary-expression* is present, evaluate it and let C be the resulting value. Otherwise, let C be the class `Object`.

b) If C is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

c) Otherwise:

1) Let N be the *constant-identifier*.

2) If a binding with name N exists in the set of bindings of constants of C, the value of the *scoped-constant-reference* is the value of the binding.

3) Otherwise:

i) Let L be the included module list of C. Search each element of L in the reverse order for a binding of a constant with name N.

ii) If the binding is found, the value of the *scoped-constant-reference* is the value of the binding.

iii) Otherwise, if C is an instance of the class `Class`, search for a binding of a constant with name N from Step e) of 11.5.4.2.

iv) Otherwise, create a direct instance of the class `Symbol` with name N, and let R be that instance. Invoke the method `const_missing` on C with R as the only argument.

11.5.4.4 Global variables

A *global-variable-identifier* is evaluated as follows:

a) Let N be the *global-variable-identifier*.

b) If a binding with name N exists in ⟦global-variable-bindings⟧ , the value of *global-variable-identifier* is the value of the binding.

c) Otherwise, the value of *global-variable-identifier* is **nil**.

11.5.4.5 Class variables

A *class-variable-identifier* is evaluated as follows:

a) Let N be the *class-variable-identifier*. Let C be the first class or module in the list at the top of ⟦class-module-list⟧ which is not a singleton class.

b) Let *CS* be the set of classes which consists of C and all the superclasses of C. Let *MS* be the set of modules which consists of all the modules in the included module list of all classes in *CS*. Let *CM* be the union of *CS* and *MS*.

c) If a binding with name N exists in the set of bindings of class variables of only one of the classes or modules in *CM*, let V be the value of the binding.

d) If more than two classes or modules in *CM* have a binding with name N in the set of bindings of class variables, let V be the value of one of these bindings. Which binding is selected is implementation-defined.

e) If none of the classes or modules in *CM* has a binding with name N in the set of bindings of class variables, let S be a direct instance of the class `Symbol` with name N and raise a direct instance of the class `NameError` which has S as its name attribute.

f) The value of the *class-variable-identifier* is V.

11.5.4.6 Instance variables

An *instance-variable-identifier* is evaluated as follows:

a) Let N be the *instance-variable-identifier*.

b) If a binding with name N exists in the set of bindings of instance variables of the current self, the value of the *instance-variable-identifier* is the value of the binding.

c) Otherwise, the value of the *instance-variable-identifier* is **nil**.

11.5.4.7 Local variables or method invocations

11.5.4.7.1 General description

An occurrence of a *local-variable-identifier* in a *variable-reference* is evaluated as either a reference to a local variable or a method invocation.

11.5.4.7.2 Determination of the type of local variable identifiers

Whether the occurrence of a *local-variable-identifier* I is a reference to a local variable or a method invocation is determined as follows:

a) Let P be the point of the program text where I occurs.

b) Let S be the innermost local variable scope which encloses P and which does not correspond to a *block* (see 9.2).

c) Let R be the region of the program text between the beginning of S and P.

d) If the same identifier as I occurs as a reference to a local variable in *variable-reference* in R, then I is a reference to a local variable.

e) If the same identifier as I occurs in one of the the forms below in R, then I is a reference to a local variable.

- *mandatory-parameter*
- *optional-parameter-name*
- *array-parameter-name*
- *proc-parameter-name*
- *variable* of *left-hand-side*
- *variable* of *single-variable-assignment-expression*
- *variable* of *single-variable-assignment-statement*
- *variable* of *abbreviated-variable-assignment-expression*
- *variable* of *abbreviated-variable-assignment-statement*

f) Otherwise, I is a method invocation.

11.5.4.7.3 Local variables

If a *local-variable-identifier* is a reference to a local variable, it is evaluated as follows:

a) Let N be the *local-variable-identifier*.

b) Search for a binding of a local variable with name N as described in 9.2.

c) If a binding is found, the value of *local-variable-identifier* is the value of the binding.

d) Otherwise, the value of *local-variable-identifier* is **nil**.

11.5.4.7.4 Method invocations

If a *local-variable-identifier* is a method invocation, it is evaluated as follows:

a) Let N be the *local-variable-identifier*.

b) Create an empty list of arguments L, and invoke the method N on the current self with L as the list of arguments (see 13.3.3).

11.5.4.8 Pseudo variables

11.5.4.8.1 General description

Syntax

pseudo-variable ::
  nil-expression
  | *true-expression*
  | *false-expression*
  | *self-expression*

NOTE A *pseudo-variable* has a similar form to a *local-variable-identifier*, but is not a variable.

11.5.4.8.2 The `nil` expression

Syntax

nil-expression ::
  `nil`

Semantics

A *nil-expression* evaluates to **nil**, which is the only instance of the class `NilClass` (see 6.6).

11.5.4.8.3 The `true` expression and the `false` expression

Syntax

true-expression ::
  `true`

false-expression ::
  `false`

Semantics

A *true-expression* evaluates to **true**, which is the only instance of the class `TrueClass`. A *false-expression* evaluates to **false**, which is the only instance of the class `FalseClass` (see 6.6).

11.5.4.8.4 The `self` expression

Syntax

self-expression ::
  `self`

Semantics

A *self-expression* evaluates to the value of the current self.

11.5.5 Object constructors

11.5.5.1 Array constructor

Syntax

---

array-constructor ::
    `[` *indexing-argument-list*$^?$ `]`

Semantics

An *array-constructor* is evaluated as follows:

a) If there is an *indexing-argument-list*, construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let L be the resulting list.

b) Otherwise, create an empty list of values L.

c) Create a direct instance of the class `Array` (see 15.2.12) which stores the values in L in the same order they are stored in L. Let O be the instance.

d) The value of the *array-constructor* is O.

11.5.5.2 Hash constructor

Syntax

---

hash-constructor ::
    `{` (*association-list* [no *line-terminator* here] `,`$^?$)$^?$ `}`

association-list ::
    association ([no *line-terminator* here] `,` *association*)*

association ::
    association-key [no *line-terminator* here] `=>` *association-value*

association-key ::
    operator-expression

association-value ::
    operator-expression

Semantics

a) A *hash-constructor* is evaluated as follows:

   1) If there is an *association-list*, evaluate the *association-list*. The value of the *hash-constructor* is the resulting value.

   2) Otherwise, create an empty direct instance of the class `Hash`. The value of the *hash-constructor* is the resulting instance.

b) An *association-list* is evaluated as follows:

   1) Create an empty direct instance H of the class `Hash`.

   2) For each *association* A_i, in the order it appears in the program text, take the following steps:

      i) Evaluate the *operator-expression* of the *association-key* of A_i. Let K_i be the resulting value.

      ii) Evaluate the *operator-expression* of the *association-value*. Let V_i be the resulting value.

      iii) Store a pair of K_i and V_i in H by invoking the method `[]=` on H with K_i and V_i as the arguments.

   3) The value of the *association-list* is H.

11.5.5.3 Range constructor

Syntax

---

range-constructor ::
    operator-OR-expression
    | *operator-OR-expression*$_1$ [no *line-terminator* here] *range-operator* *operator-OR-expression*$_2$

range-operator ::
    `..`
    | `...`

Semantics

A *range-constructor* of the form *operator-OR-expression*$_1$ *range-operator* *operator-OR-expression*$_2$ is evaluated as follows:

a) Evaluate the *operator-OR-expression*$_1$. Let A be the resulting value.

b) Evaluate the *operator-OR-expression*$_2$. Let B be the resulting value.

c) If the *range-operator* is the terminal "`..`", construct a list L which contains three arguments: A, B, and **false**.

If the *range-operator* is the terminal "`...`", construct a list L which contains three arguments: A, B, and **true**.

d) Invoke the method `new` on the class `Range` (see 15.2.14) with L as the list of arguments. The value of the *range-constructor* is the resulting value.

12 Statements

12.1 General description

Syntax

---

statement ::
  expression-statement
  | *alias-statement*
  | *undef-statement*
  | *if-modifier-statement*
  | *unless-modifier-statement*
  | *while-modifier-statement*
  | *until-modifier-statement*
  | *rescue-modifier-statement*
  | *assignment-statement*

Semantics

See 13.3.6 for *alias-statement*s.

See 13.3.7 for *undef-statement*s.

See 11.4.2 for *assignment-statement*s.

12.2 The expression statement

Syntax

---

expression-statement ::
  expression

Semantics

An *expression-statement* is evaluated as follows:

a) Evaluate the *expression*.

b) The value of the *expression-statement* is the resulting value.

12.3 The `if` modifier statement

Syntax

---

if-modifier-statement ::
    statement [no *line-terminator* here] `if` *expression*

Semantics

An *if-modifier-statement* of the form S `if` E, where S is the *statement* and E is the *expression*, is evaluated as follows:

a) Evaluate the *if-expression* of the form `if` E `then` S `end`.

b) The value of the *if-modifier-statement* is the resulting value.

12.4 The `unless` modifier statement

Syntax

---

unless-modifier-statement ::
    statement [no *line-terminator* here] `unless` *expression*

Semantics

An *unless-modifier-statement* of the form S `unless` E, where S is the *statement* and E is the *expression*, is evaluated as follows:

a) Evaluate the *unless-expression* of the form `unless` E `then` S `end`.

b) The value of the *unless-modifier-statement* is the resulting value.

12.5 The `while` modifier statement

Syntax

---

while-modifier-statement ::
    statement [no *line-terminator* here] `while` *expression*

Semantics

A *while-modifier-statement* of the form S `while` E, where S is the *statement* and E is the *expression*, is evaluated as follows:

a) If S is a *begin-expression*, the behavior is implementation-defined.

b) Evaluate the *while-expression* of the form `while` E `do` S `end`.

c) The value of the *while-modifier-statement* is the resulting value.

12.6 The `until` modifier statement

Syntax

---

until-modifier-statement ::
    statement [no *line-terminator* here] `until` *expression*

Semantics

An *until-modifier-statement* of the form S `until` E, where S is the *statement* and E is the *expression*, is evaluated as follows:

a) If S is a *begin-expression*, the behavior is implementation-defined.

b) Evaluate the *until-expression* of the form `until` E `do` S `end`.

c) The value of the *until-modifier-statement* is the resulting value.

12.7 The `rescue` modifier statement

Syntax

---

rescue-modifier-statement ::
    main-statement-of-rescue-modifier-statement [no *line-terminator* here]
      `rescue` *fallback-statement-of-rescue-modifier-statement*

main-statement-of-rescue-modifier-statement ::
    statement

fallback-statement-of-rescue-modifier-statement ::
    statement **but not** *statement-not-allowed-in-fallback-statement*

statement-not-allowed-in-fallback-statement ::
    keyword-AND-expression
    | *keyword-OR-expression*
    | *if-modifier-statement*
    | *unless-modifier-statement*
    | *while-modifier-statement*
    | *until-modifier-statement*
    | *rescue-modifier-statement*

Semantics

A *rescue-modifier-statement* is evaluated as follows:

a) Evaluate the *main-statement-of-rescue-modifier-statement*. Let V be the resulting value.

b) If a direct instance of the class `StandardError` is raised and not handled in Step a), evaluate *fallback-statement-of-rescue-modifier-statement*. The value of the *rescue-modifier-statement* is the resulting value.

c) If no instances of the class `Exception` are raised in Step a), or all the instances of the class `Exception` raised in Step a) are handled in Step a), the value of the *rescue-modifier-statement* is V.

13 Classes and modules

13.1 Modules

13.1.1 General description

Every module is an instance of the class `Module` (see 15.2.2). However, not every instance of the class `Module` is a module because the class `Module` is a superclass of the class `Class`, an instance of which is not a module, but a class.

Modules have the following attributes:

Included module list: A list of modules included in the module. Module inclusion is described in 13.1.3.

Constants: A set of bindings of constants.

A binding of a constant is created by the following program constructs:

- *Assignment*s (see 11.4.2)
- *Module-definition*s (see 13.1.2)
- *Class-definition*s (see 13.2.2)

Class variables: A set of bindings of class variables. A binding of a class variable is created by an *assignment* (see 11.4.2).

Instance methods: A set of method bindings. A method binding is created by a *method-definition* (see 13.3.1), a *singleton-method-definition* (see 13.4.3), an *alias-statement* (see 13.3.6) or an *undef-statement* (see 13.3.7). The value of a method binding may be ***undef***, which is the flag indicating that a method cannot be invoked (see 13.3.7).

13.1.2 Module definition

Syntax

module-definition ::
    `module` *module-path* *module-body* `end`

module-path ::
    top-module-path
    | *module-name*
    | *nested-module-path*

module-name ::
    constant-identifier

top-module-path ::
    `::` *module-name*

nested-module-path ::
    primary-expression [no *line-terminator* here] `::` *module-name*

module-body ::
    body-statement

Semantics

A *module-definition* is evaluated as follows:

a) Determine the class or module C in which a binding with name *module-name* is to be created or modified as follows:

   1) If the *module-path* is of the form *top-module-path*, let C be the class `Object`.

   2) If the *module-path* is of the form *module-name*, let C be the current class or module.

   3) If the *module-path* is of the form *nested-module-path*, evaluate the *primary-expression*. If the resulting value is an instance of the class `Module`, let C be the instace. Otherwise, raise a direct instance of the class `TypeError`.

b) Let N be the *module-name*.

   1) If a binding with name N exists in the set of bindings of constants of C, let B be this binding. If the value of B is a module, let M be that module. Otherwise, raise a direct instance of the class `TypeError`.

   2) Otherwise, create a direct instance M of the class `Module`. Create a variable binding with name N and value M in the set of bindings of constants of C.

c) Modify the execution context as follows:

1) Create a new list which has the same members as that of the list at the top of ⟦class-module-list⟧ , and add M to the head of the newly created list. Push the list onto ⟦class-module-list⟧ .

2) Push M onto ⟦self⟧ .

3) Push the public visibility onto ⟦default-method-visibility⟧ .

4) Push an empty set of bindings onto ⟦local-variable-bindings⟧ .

d) Evaluate the *body-statement* (see 11.5.2.5) of the *module-body*. The value of the *module-definition* is the resulting value of the *body-statement*.

e) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧ , ⟦self⟧ , ⟦default-method-visibility⟧ , and ⟦local-variable-bindings⟧ .

13.1.3 Module inclusion

Modules and classes can be extended by including other modules into them. When a module is included, the instance methods, the class variables, and the constants of the included module are available to the including class or module (see 11.5.4.5, 13.3.3, and 11.5.4.2).

Modules and classes can include other modules by invoking the method `include` (see 15.2.2.4.27) or the method `extend` (see 15.3.1.3.13).

A module M is included in another module N if and only if M is an element of the included module list of N. A module M is included in a class C if and only if M is an element of the included module list of C, or M is included in one of the superclasses of C.

13.2 Classes

13.2.1 General description

Every class is an instance of the class `Class` (see 15.2.3), which is a direct subclass of the class `Module`.

Classes have the same set of attributes as modules. In addition, every class has at most one single direct superclass.

13.2.2 Class definition

Syntax

---

class-definition ::
    `class` *class-path* [no *line-terminator* here] (`<` *superclass*)$^?$ *separator*
      class-body `end`

class-path ::
    top-class-path
  | *class-name*
  | *nested-class-path*

class-name ::
  constant-identifier

top-class-path ::
  `::` *class-name*

nested-class-path ::
  primary-expression [no *line-terminator* here] `::` *class-name*

superclass ::
  expression

class-body ::
  body-statement

Semantics

A *class-definition* is evaluated as follows:

a) Determine the class or module M in which the binding with name *class-name* is to be created or modified as follows:

   1) If the *class-path* is of the form *top-class-path*, let M be the class `Object`.

   2) If the *class-path* is of the form *class-name*, let M be the current class or module.

   3) If the *class-path* is of the form *nested-class-path*, evaluate the *primary-expression*. If the resulting value is an instance of the class `Module`, let M be the instance. Otherwise, raise a direct instance of the class `TypeError`.

b) Let N be the *class-name*.

   1) If a binding with name N exists in the set of bindings of constants of M, let B be that binding.

      i) If the value of B is an instance of the class `Class`, let C be the instance. Otherwise, raise a direct instance of the class `TypeError`.

      ii) If the *superclass* is present, evaluate it. If the resulting value does not correspond to the direct superclass of C, raise a direct instance of the class `TypeError`.

   2) Otherwise, create a direct instance of the class `Class`. Let C be that class.

      i) If the *superclass* is present, evaluate it. If the resulting value is not an instance of the class `Class`, raise a direct instance of the class `TypeError`. If the value of the *superclass* is a singleton class or the class `Class`, the behavior is unspecified. Otherwise, set the direct superclass of C to the value of the *superclass*.

ii) If the *superclass* of the *class-definition* is omitted, set the direct superclass of C to the class `Object`.

iii) Create a singleton class, and associate it with C. It shall have the singleton class of the direct superclass of C as one of its superclasses.

iv) Create a variable binding with name N and value C in the set of bindings of constants of M.

c) Modify the execution context as follows:

1) Create a new list which has the same members as that of the list at the top of ⟦class-module-list⟧, and add C to the head of the newly created list. Push the list onto ⟦class-module-list⟧.

2) Push C onto ⟦self⟧.

3) Push the public visibility onto ⟦default-method-visibility⟧.

4) Push an empty set of bindings onto ⟦local-variable-bindings⟧.

d) Evaluate the *body-statement* (see 11.5.2.5) of the *class-body*. The value of the *class-definition* is the resulting value of the *body-statement*.

e) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧, ⟦self⟧, ⟦default-method-visibility⟧, and ⟦local-variable-bindings⟧.

13.2.3 Inheritance

A class inherits attributes of its superclasses. Inheritance means that a class implicitly contains all attributes of its superclasses, as described below:

- Constants and class variables of superclasses can be referred to (see 11.5.4.2 and 11.5.4.5).
- Singleton methods of superclasses can be invoked (see 13.4).
- Instance methods defined in superclasses can be invoked on an instance of their subclasses (see 13.3.3).

13.2.4 Instance creation

A direct instance of a class can be created by invoking the method `new` (see 15.2.3.3.3) on the class.

13.3 Methods

13.3.1 Method definition

Syntax

method-definition ::
  `def` *defined-method-name* [no *line-terminator* here] *method-parameter-part*
    method-body `end`

defined-method-name ::
  method-name
  | *assignment-like-method-identifier*

method-body ::
  body-statement

The following constructs shall not be present in the *method-parameter-part* or the *method-body*:

- A *class-definition*.
- A *module-definition*.
- A *single-variable-assignment*, where its *variable* is a *constant-identifier*.
- A *scoped-constant-assignment*.
- A *multiple-assignment-statement* in which there exists a *left-hand-side* of any of the following forms:
  - — *constant-identifier*;
  - — *primary-expression* [no *line-terminator* here] (`.` | `::`) (*local-variable-identifier* | *constant-identifier*);
  - — `::` *constant-identifier*.

However, those constructs may occur within a *singleton-class-definition* in the *method-parameter-part* or the *method-body*.

Semantics

A method is defined by a *method-definition* or a *singleton-method-definition* (see 13.4.3), and has the *method-parameter-part* and the *method-body* of the *method-definition* or *singleton-method-definition*. The *method-body* is evaluated when the method is invoked (see 13.3.3). The evaluation of the *method-body* is the evaluation of its *body-statement* (see 11.5.2.5). In addition, a method has the following attributes:

Class module list: The list of classes and modules which is the top element of ⟦class-module-list⟧ when the method is defined.

Defined name: The name with which the method is defined.

Visibility: The visibility of the method (see 13.3.5).

A class or a module can define a new method with the same name as the name of a method in one of its superclasses or included modules of the class or module. In that case, the new method is said to ***override*** the method in the superclass or the included module.

A *method-definition* is evaluated as follows:

a) Let N be the *defined-method-name*.

b) Create a method U defined by the *method-definition*. Initialize the attributes of U as follows:

- The class module list is the element at the top of ⟦class-module-list⟧.
- The defined name is N.
- The visibility is:
  - — If the current class or module is a singleton class, then the current visibility.
  - — Otherwise, if N is `initialize` or `initialize_copy`, then the private visibility.
  - — Otherwise, the current visibility.

c) If a method binding with name N exists in the set of bindings of instance methods of the current class or module, let V be the value of that binding.

   1) If V is undef, the evaluation of the *method-definition* is implementation-defined.

   2) Replace the value of the binding with U.

d) Otherwise, create a method binding with name N and value U in the set of bindings of instance methods of the current class or module.

e) The value of the *method-definition* is implementation-defined.

13.3.2 Method parameters

Syntax

---

method-parameter-part ::
  (*parameter-list*$^?$)
  | *parameter-list*$^?$ *separator*

parameter-list ::
  mandatory-parameter-list (, *optional-parameter-list*)$^?$
   (, *array-parameter*)$^?$ (, *proc-parameter*)$^?$
  | *optional-parameter-list* (, *array-parameter*)$^?$ (, *proc-parameter*)$^?$
  | *array-parameter* (, *proc-parameter*)$^?$
  | *proc-parameter*

mandatory-parameter-list ::
  mandatory-parameter
  | *mandatory-parameter-list* **,** *mandatory-parameter*

mandatory-parameter ::
  local-variable-identifier

optional-parameter-list ::
  optional-parameter
  | *optional-parameter-list* **,** *optional-parameter*

optional-parameter ::
  optional-parameter-name **=** *default-parameter-expression*

optional-parameter-name ::
  local-variable-identifier

default-parameter-expression ::
  operator-expression

array-parameter ::
  ***** *array-parameter-name*
  | *****

array-parameter-name ::
  local-variable-identifier

proc-parameter ::
  & *proc-parameter-name*

proc-parameter-name ::
  local-variable-identifier

All the *local-variable-identifier*s of *mandatory-parameter*s, *optional-parameter-name*s, the *array-parameter-name*, and the *proc-parameter-name* in a *parameter-list* shall be different.

Semantics

There are four kinds of parameters as described below. How those parameters are bound to the actual arguments is described in 13.3.3.

Mandatory parameters: These parameters are represented by *mandatory-parameter*s. For each mandatory parameter, a corresponding actual argument shall be given when the method is invoked.

Optional parameters: These parameters are represented by *optional-parameter*s. Each optional parameter consists of a parameter name represented by *optional-parameter-name* and an expression represented by *default-parameter-expression*. For each optional parameter, when there is no corresponding argument in the list of arguments given to the method invocation, the value of the *default-parameter-expression* is used as the value of the argument.

An array parameter: This parameter is represented by *array-parameter-name*. Let N be the number of arguments, excluding a *block-argument*, given to a method invocation. If N is more than the sum of the number of mandatory parameters and optional parameters, this parameter is bound to a direct instance of the class `Array` containing the extra arguments excluding a *block-argument*. Otherwise, the parameter is bound to an empty direct instance of the class `Array`. If an *array-parameter* is of the form "*", those extra arguments are ignored.

A proc parameter: This parameter is represented by *proc-parameter-name*. The parameter is bound to a direct instance of the class `Proc` which represents the block passed to the method invocation.

13.3.3 Method invocation

The way in which a list of arguments is created is described in 11.3.

Given the receiver R, the method name M, and the list of arguments A, take the following steps:

a) If the method is invoked with a block, let B be the block. Otherwise, let B be block-not-given.

b) Let C be the singleton class of R if R has a singleton class. Otherwise, let C be the class of R.

c) Search for a method binding with name M, starting from C as described in 13.3.4.

d) If a binding is found and its value is not undef, let V be the value of the binding.

e) Otherwise, if M is `method_missing`, the behavior is unspecified. If M is not `method_missing`, add a direct instance of the class `Symbol` with name M to the head of A, and invoke the method `method_missing` (see 15.3.1.3.30) on R with A as arguments and B as the block. Let O be the resulting value, and go to Step j).

f) Check the visibility of V to see whether the method can be invoked (see 13.3.5). If the method cannot be invoked, add a direct instance of the class `Symbol` with name M to the head of A, and invoke the method `method_missing` on R with A as arguments and B as the block. Let O be the resulting value, and go to Step j).

g) Modify the execution context as follows:

    1) Push the class module list of V onto ⟦class-module-list⟧.

    2) Push R onto ⟦self⟧.

    3) Push M onto ⟦invoked-method-name⟧.

    4) Push the public visibility to ⟦default-method-visibility⟧.

5) Push the defined name of V onto ⟦defined-method-name⟧ .

6) Push B onto ⟦block⟧ .

7) Push an empty set of local variable bindings onto ⟦local-variable-bindings⟧ .

h) Evaluate the *method-parameter-part* of V as follows:

1) Let L be the *parameter-list* of the *method-parameter-part*.

2) Let P_m, P_o, and P_a be the *mandatory-parameter*s of the *mandatory-parameter-list*, the *optional-parameter*s of the *optional-parameter-list*, and the *array-parameter* of L, respectively. Let N_A, N_{Pm}, and N_{Po} be the number of elements of A, P_m, and P_o respectively. If there are no *mandatory-parameter*s or *optional-parameter*s, let N_{Pm} and N_{Po} be 0. Let S_b be the current set of local variable bindings.

3) If N_A is smaller than N_{Pm}, raise a direct instance of the class `ArgumentError`.

4) If the method does not have P_a and N_A is larger than the sum of N_{Pm} and N_{Po}, raise a direct instance of the class `ArgumentError`.

5) Otherwise, for each ith argument A_i in A, in the same order in A, take the following steps:

i) Let P_i be the ith *mandatory-parameter* or *optional-parameter* in the order it appears in L.

ii) If such P_i does not exist, go to Step h) 6).

iii) If P_i is a mandatory parameter, let n be the *mandatory-parameter*. If P_i is an optional parameter, let n be the *optional-parameter-name*. Create a variable binding with name n and value A_i in S_b.

6) If N_A is larger than the sum of N_{Pm} and N_{Po}, and P_a exists:

i) Create a direct instance X of the class `Array` whose length is the number of extra arguments.

ii) Store each extra arguments into X, preserving the order in which they occur in the list of arguments.

iii) Let n be the *array-parameter-name* of P_a.

iv) Create a variable binding with name n and value X in S_b.

7) If N_A is smaller than the sum of N_{Pm} and N_{Po}:

i) For each optional parameter P_{Oi} to which no argument corresponds, evaluate the *default-parameter-expression* of P_{Oi}, and let X be the resulting value.

ii) Let n be the *optional-parameter-name* of P_{Oi}.

        iii) Create a variable binding with name n and value X in S_b.

    8) If N_A is smaller than or equal to the sum of N_{Pm} and N_{Po}, and P_a exists:

        i) Create an empty direct instance X of the class `Array`.

        ii) Let n be the *array-parameter-name* of P_a.

        iii) Create a variable binding with name n and value X in S_b.

    9) If the *proc-parameter* of L is present, let D be the top of ⟦block⟧ .

        i) If D is block-not-given, let X be **nil**.

        ii) Otherwise, invoke the method `new` on the class `Proc` with an empty list of arguments and D as the block. Let X be the resulting value of the method invocation.

        iii) Let n be the *proc-parameter-name* of *proc-parameter*.

        iv) Create a variable binding with name n and value X in S_b.

i) Evaluate the *method-body* of V.

    1) If the evaluation of the *method-body* is terminated by a *return-expression*:

        i) If the *jump-argument* of the *return-expression* is present, let O be the value of the *jump-argument*.

        ii) Otherwise, let O be **nil**.

    2) Otherwise, let O be the resulting value of the evaluation.

j) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧ , ⟦self⟧ , ⟦invoked-method-name⟧ , ⟦default-method-visibility⟧ , ⟦defined-method-name⟧ , ⟦block⟧ , and ⟦local-variable-bindings⟧ .

k) The value of the method invocation is O.

The method invocation or the *super-expression* [see 11.3.4 d)] which corresponds to the set of items on the tops of all the attributes of the execution context modified in Step g), except ⟦local-variable-bindings⟧ , is called the ***current method invocation***.

13.3.4 Method lookup

Method lookup is the process by which a binding of an instance method is resolved.

Given a method name M and a class or a module C which is initially searched for the binding of the method, the method binding is resolved as follows:

a) If a method binding with name M exists in the set of bindings of instance methods of C, let B be that binding.

b) Otherwise, let L_m be the included module list of C. Search for a method binding with name M in the set of bindings of instance methods of each module in L_m. Examine modules in L_m in reverse order.

   1) If a binding is found, let B be that binding.

   2) Otherwise:

      i) If C does not have a direct superclass, the binding is considered not resolved.

      ii) Otherwise, let new C be the direct superclass of C, and continue processing from Step a).

   B is the resolved method binding.

13.3.5 Method visibility

13.3.5.1 General description

Methods are categorized into one of public, private, or protected methods according to the conditions under which the method invocation is allowed. The attribute of a method which determines these conditions is called the ***visibility*** of the method.

13.3.5.2 Public methods

A public method is a method whose visibility attribute is set to the public visibility.

A public method can be invoked on an object anywhere within a program.

13.3.5.3 Private methods

A private method is a method whose visibility attribute is set to the private visibility.

A private method cannot be invoked with an explicit receiver, i.e., a private method cannot be invoked if a *primary-expression* or a *chained-method-invocation* occurs at the position which corresponds to the method receiver in the method invocation, except for a method invocation of any of the following forms where the *primary-expression* is a *self-expression*.

•) *single-method-assignment*

• *abbreviated-method-assignment*

• *single-indexing-assignment*

• *abbreviated-indexing-assignment*

13.3.5.4 Protected methods

A protected method is a method whose visibility attribute is set to the protected visibility.

A protected method can be invoked if and only if the following condition holds:

- Let M be an instance of the class `Module` in which the binding of the method exists.

  M is included in the current self, or M is the class of the current self or one of its superclasses.

If M is a singleton class, whether the method can be invoked or not may be determined in a implementation-defined way.

13.3.5.5 Visibility change

The visibility of methods can be changed with the methods `public` (see 15.2.2.4.38), `private` (see 15.2.2.4.36), and `protected` (see 15.2.2.4.37), which are defined in the class `Module`.

13.3.6 The `alias` statement

Syntax

---

alias-statement ::
  `alias` *new-name* *aliased-name*

new-name ::
  defined-method-name
  | *symbol*

aliased-name ::
  defined-method-name
  | *symbol*

Semantics

An *alias-statement* is evaluated as follows:

a) Evaluate the *new-name* as follows:

   1) If the *new-name* is of the form *defined-method-name*, let N be the *defined-method-name*.

   2) If the *new-name* is of the form *symbol*, evaluate it. Let N be the name of the resulting instance of the class `Symbol`.

b) Evaluate the *aliased-name* as follows:

   1) If *aliased-name* is of the form *defined-method-name*, let A be the *defined-method-name*.

   2) If *aliased-name* is of the form *symbol*, evaluate it. Let A be the name of the resulting instance of the class `Symbol`.

c) Let C be the current class or module.

d) Search for a method binding with name A, starting from C as described in 13.3.4.

e) If a binding is found and its value is not undef, let V be the value of the binding.

f) Otherwise, let S be a direct instance of the class `Symbol` with name A and raise a direct instance of the class `NameError` which has S as its name attribute.

g) If a method binding with name N exists in the set of bindings of instance methods of the current class or module, replace the value of the binding with V.

h) Otherwise, create a method binding with name N and value V in the set of bindings of instance methods of the current class or module.

i) The value of the *alias-statement* is **nil**.

13.3.7 The `undef` statement

Syntax

---

undef-statement ::
  `undef` *undef-list*

undef-list ::
  method-name-or-symbol (`,` *method-name-or-symbol*)*

method-name-or-symbol ::
  defined-method-name
  | *symbol*

Semantics

An *undef-statement* is evaluated as follows:

a) For each *method-name-or-symbol* of the *undef-list*, take the following steps:

  1) Let C be the current class or module.

  2) If the *method-name-or-symbol* is of the form *defined-method-name*, let N be the *defined-method-name*. Otherwise, evaluate the *symbol*. Let N be the name of the resulting instance of the class `Symbol`.

  3) Search for a method binding with name N, starting from C as described in 13.3.4.

  4) If a binding is found and its value is not undef:

    i) If the binding is found in C, replace the value of the binding with undef.

    ii) Otherwise, create a method binding with name N and value undef in the set of bindings of instance methods of C.

  5) Otherwise, let S be a direct instance of the class `Symbol` with name N and raise a direct instance of the class `NameError` which has S as its name attribute.

b) The value of the *undef-statement* is **nil**.

13.4 Singleton classes

13.4.1 General description

A singleton class is an object which is associated with another object. A singleton class modifies the behavior of an object when associated with it. When such an association is made, the singleton class is called the singleton class of the object, and the object is called the primary associated object of the singleton class.

An object has at most one singleton class. When an object is created, it shall not be associated with any singleton classes unless the object is an instance of the class `Class`. Singleton classes are associated with an object by evaluation of a program construct such as a *singleton-method-definition* or a *singleton-class-definition*. However, when an instance of the class `Class` is created, it shall already have been associated with its singleton class.

Normally, a singleton class shall be associated with only its primary associated object; however, the singleton class of an instance of the class `Class` may be associated with some additional instances of the class `Class` which are not the primary associated objects of any other singleton classes, in an implementation-defined way. Once associated, the primary associated object of a singleton class shall not be dissociated from its singleton class; however the aforementioned additional associated instances of the class `Class` are dissociated from their singleton class when they become the primary associated object of another singleton class [see 13.4.2 e) and 13.4.3 e)].

Every singleton class is an instance of the class `Class` (see 15.2.3), and has the same set of attributes as classes.

The direct superclass of a singleton class is implementation-defined. However, a singleton class shall be a subclass of the class of the object with which it is associated.

NOTE 1 For example, the singleton class of the class `Object` is a subclass of the class `Class` because the class `Object` is a direct instance of the class `Class`. Therefore, the instance methods of the class `Class` can be invoked on the class `Object`.

The singleton class of a class which has a direct superclass shall satisfy the following condition:

- Let E_c be the singleton class of a class C, and let S be the direct superclass of C, and let E_s be the singleton class of S. Then, E_c have E_s as one of its superclasses.

NOTE 2 This requirement enables classes to inherit singleton methods from its superclasses. For example, the singleton class of the class `File` has the singleton class of the class `IO` as its superclass. Thereby, the class `File` inherits the singleton method `open` of the class `IO`.

Although singleton classes are instances of the class `Class`, they cannot create an instance of themselves. When the method `new` is invoked on a singleton class, a direct instance of the class `TypeError` shall be raised [see 15.2.3.3.3 a)].

Whether a singleton class can be a superclass of other classes is unspecified [see 13.2.2 b) 2) i) and 15.2.3.3.1 c)].

Whether a singleton class can have class variables or not is implementation-defined.

13.4.2 Singleton class definition

Syntax

---

singleton-class-definition ::
  `class <<` *expression separator singleton-class-body* `end`

singleton-class-body ::
  body-statement

Semantics

A *singleton-class-definition* is evaluated as follows:

a) Evaluate the *expression*. Let O be the resulting value. If O is an instance of the class `Integer` or the class `Symbol`, a direct instance of the class `TypeError` may be raised.

b) If O is one of **nil**, **true**, or **false**, let E be the class of O and go to Step f).

c) If O is not associated with a singleton class, create a new singleton class. Let E be the newly created singleton class, and associate O with E as its primary associated object.

d) If O is associated with a singleton class as its primary associated object, let E be that singleton class.

e) If O is associated with a singleton class not as its primary associated object, dissociate O from the singleton class, and create a new singleton class. Let E be the newly created singleton class, and associate O with E as its primary associated object.

f) Modify the execution context as follows:

   1) Create a new list which consists of the same elements as the list at the top of ⟦class-module-list⟧ and add E to the head of the newly created list. Push the list onto ⟦class-module-list⟧.

   2) Push E onto ⟦self⟧.

   3) Push the public visibility onto ⟦default-method-visibility⟧.

   4) Push an empty set of bindings onto ⟦local-variable-bindings⟧.

g) Evaluate the *singleton-class-body*. The value of the *singleton-class-definition* is the value of the *singleton-class-body*.

h) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧, ⟦self⟧, ⟦default-method-visibility⟧, and ⟦local-variable-bindings⟧.

13.4.3 Singleton method definition

Syntax

singleton-method-definition ::
    `def` *singleton* (`.` | `::`) *defined-method-name* [no *line-terminator* here]
      method-parameter-part *method-body* `end`

singleton ::
    variable-reference
  | `(` *expression* `)`

Semantics

A *singleton-method-definition* is evaluated as follows:

a) Evaluate the *singleton*. Let S be the resulting value. If S is an instance of the class `Integer` or the class `Symbol`, a direct instance of the class `TypeError` may be raised.

b) If S is one of **nil**, **true**, or **false**, let E be the class of O and go to Step f).

c) If S is not associated with a singleton class, create a new singleton class. Let E be the newly created singleton class, and associate S with E as its primary associated object.

d) If S is associated with a singleton class as its primary associated object, let E be that singleton class.

e) If S is associated with a singleton class not as its primary associated object, dissociate S from the singleton class, and create a new singleton class. Let E be the newly created singleton class, and associate S with E as its primary associated object.

f) Let N be the *defined-method-name*.

g) Create a method U defined by the *singleton-method-definition*. U has the *method-parameter-part* and the *method-body* of the *singleton-method-definition* as described in 13.3.1. Initialize the attributes of U as follows:

- The class module list is the element at the top of ⟦class-module-list⟧.
- The defined name is N.
- The visibility is the public visibility.

h) If a method binding with name N exists in the set of bindings of instance methods of E, let V be the value of that binding.

   1) If V is undef, the evaluation of the *singleton-method-definition* is implementation-defined.

   2) Replace the value of the binding with U.

i) Otherwise, create a method binding with name N and value U in the set of bindings of instance methods of E.

j) The value of the *singleton-method-definition* is implementation-defined.

14 Exceptions

14.1 General description

If an instance of the class `Exception` is raised, the current evaluation process stops, and control is transferred to a program construct that can handle this exception.

14.2 Cause of exceptions

An exception is raised when:

- the method `raise` (see 15.3.1.2.12) is invoked.
- an exceptional condition occurs as described in various parts of this document.

Only instances of the class `Exception` shall be raised.

14.3 Exception handling

Exceptions are handled by a *body-statement*, an *assignment-with-rescue-modifier*, or a *rescue-modifier-statement*. These program constructs are called ***exception handler***s. When an exception handler is handling an exception, the exception being handled is called the ***current exception***.

When an exception is raised, it is handled by an exception handler. This exception handler is determined as follows:

a) Let S be the innermost local variable scope which lexically encloses the location where the exception is raised, and which corresponds to one of a *program*, a *method-definition*, a *singleton-method-definition*, or a *block*.

b) Test each exception handler in S which lexically encloses the location where the exception is raised from the innermost to the outermost.

   - An *assignment-with-rescue-modifier* is considered to handle the exception if the exception is an instance of the class `StandardError` (see 11.4.2.5), except when the exception is raised in its *operator-expression*$_2$. In this case, *assignment-with-rescue-modifier* does not handle the exception.
   - A *rescue-modifier-statement* is considered to handle the exception if the exception is an instance of the class `StandardError` (see 12.7), except when the exception is raised in its *fallback-statement-of-rescue-modifier-statement*. In this case, *rescue-modifier-statement* does not handle the exception.
   - A *body-statement* is considered to handle the exception if one of its *rescue-clause*s is considered to handle the exception (see 11.5.2.5), except when the exception is raised in one of its *rescue-clause*s, *else-clause*, or *ensure-clause*. In this case, *body-statement* does not handle the exception. If an *ensure-clause* of a *body-statement* is present, it is evaluated even if the handler does not handle the exception (see 11.5.2.5).

c) If an exception handler which can handle the exception is found in S, terminate the search for the exception handler. Continue evaluating the program as defined for the relevant construct (see 11.4.2.5, 11.5.2.5, and 12.7).

d) If none of the exception handlers in S can handle the exception:

1) If S corresponds to a *method-definition* or a *singleton-method-definition*, terminate Step h) or Step i) of 13.3.3, and take Step j) of the current method invocation (see 13.3.3). Continue the search from Step a), under the assumption that the exception is raised at the location where the method is invoked.

2) If S corresponds to a *block*, terminate the evaluation of the current *block*, and take Step f) of 11.3.3. Continue the search from Step a), under the assumption that the exception is raised at the location where the block is called.

3) If S corresponds to a *program*, terminate the evaluation of the *program*, take Step d) of 10.1, and print the information of the exception in an implementation-defined way.

15 Built-in classes and modules

15.1 General description

Built-in classes and modules are classes and modules which are already created before execution of a program (see 7.2).

Built-in classes and modules are respectively specified in 15.2 and 15.3. A built-in class or module is specified by describing the following attributes (see 13.1.1 and 13.2.1):

- The direct superclass (for built-in classes only).
- The include module list.
- Constants.
- Singleton methods, i.e. instance methods of the singleton class of the built-in class or module. The class module list of a singleton method of the built-in class or module consists of two elements: the first is the singleton class of the built-in class or module; the second is the class `Object`.
- Instance methods. The class module list (see 13.3.1) of an instance method of the built-in class or module consists of two elements: the first is the built-in class or module; the second is the class `Object`.

The set of bindings of class variables of a built-in class or module is an empty set.

NOTE A built-in class or module is not created by a *class-definition* or *module-definition* in a program text, but is created as a class or module whose attributes are described in 15.2 or 15.3 in advance prior to an execution of a program.

A conforming processor may provide the following additional attributes and/or values.

- A specific initial value for an attribute defined in this document whose initial value is not specified in this document;
- Constants, singleton methods, instance methods;
- Additional optional parameters or array parameters for methods specified in this document;
- Additional inclusion of modules into built-in classes/modules.

In 15.2 and 15.3, the following notations are used:

- Each subclause of 15.2 and 15.3 (e.g., 15.2.1) specifies a built-in class or module. The title of the subclause is the name of the built-in class or module. The name is used as the name of a constant binding in the class `Object` (see 15.2.1.4).
- A built-in class except the class `Object` (see 15.2.1) has, as its direct superclass, the class described in the subclause titled "Direct superclass" in the subclause specifying the built-in class.
- When a subclause specifying a built-in class or module contains a subclause titled "Included modules", the built-in class or module includes (see 13.1.3) the modules listed in that subclause in the order of that listing.
- Each subclause in a subclause titled "Singleton methods" with a title of the form *C.m* specifies the singleton method *m* of the class *C*.
- Each subclause in a subclause titled "Instance methods" with a title of the form *C#m* specifies the instance method *m* of the class *C*.
- The parameter specification of a method is described in the form of *method-parameter-part* (see 13.3.2).

  EXAMPLE 1 The following example defines the parameter specification of a method `sample`.

  ```
  sample( arg1, arg2, opt=expr, *ary, &blk)
  ```

- A singleton method name is prefixed by the name of the class or the module, and a dot (`.`).

  EXAMPLE 2 The following example defines the parameter specification of a singleton method `sample` of a class `SampleClass`:

  ```
  SampleClass.sample( arg1, arg2, opt=expr, *ary, &blk)
  ```

- Next to the parameter specification, the visibility and the behavior of the method are specified.

  The visibility, which is any one of public, protected, or private, is specified after the label named "Visibility:".

  The behavior, which is the steps which shall be taken while evaluating the *method-body* of the method [see 13.3.3 i)], is specified after the label named "Behavior:".

In these steps, a reference to the name of an argument in the parameter specification is considered to be the object bound to the local variables of the same name.

- The phrase "call *block* with X as the argument" indicates that the block corresponding to the proc parameter *block* is called as described in 11.3.3 with X as the argument to the block call.

- The phrase "return X" indicates that the evaluation of the *method-body* is terminated at that point, and X is the value of the *method-body*.

- The phrase "the name designated by N" means the result of the following steps:

  a) If N is an instance of the class `Symbol`, the name of N.

  b) If N is an instance of the class `String`, the content (see 15.2.10.1) of N.

  c) Otherwise, the behavior of the method is unspecified.

15.2 Built-in classes

15.2.1 Object

15.2.1.1 General description

The class `Object` is an implicit direct superclass for other classes. That is, if the direct superclass of a class is not specified explicitly in the class definition, the direct superclass of the class is the class `Object` (see 13.2.2).

All built-in classes and modules can be referred to through constants of the class `Object` (see 15.2.1.4).

15.2.1.2 Direct superclass

The class `Object` does not have a direct superclass, or may have an implementation-defined superclass.

15.2.1.3 Included modules

The following module is included in the class `Object`.

- `Kernel`

15.2.1.4 Constants

The following constants are defined in the class `Object`.

STDIN: An implementation-defined readable instance of the class `IO`, which is used for reading conventional input.

STDOUT: An implementation-defined writable instance of the class `IO`, which is used for writing conventional output.

STDERR: An implementation-defined writable instance of the class `IO`, which is used for writing diagnostic output.

Other than these constants, for each built-in class or module, including the class `Object` itself, a conforming processor shall define a constant in the class `Object`, whose name is the name of the class or module, and whose value is the class or module itself.

15.2.1.5 Instance methods

15.2.1.5.1 Object#initialize

`initialize(` `*`*args* `)`

Visibility: private

Behavior: The method `initialize` is the default object initialization method, which is invoked when an instance is created (see 13.2.4). It returns an implementation-defined value.

If the class `Object` is not the root of the class inheritance tree, the method `initialize` shall be defined in the class which is the root of the class inheritance tree instead of in the class `Object`.

15.2.2 Module

15.2.2.1 General description

All modules are instances of the class `Module`. Therefore, behaviors defined in the class `Module` are shared by all modules.

The binary relation on the instances of the class `Module` denoted $A \sqsubset B$ is defined as follows:

- B is a module, and B is included in A (see 13.1.3) or
- Both A and B are instances of the class `Class`, and B is a superclass of A.

15.2.2.2 Direct superclass

The class `Object`

15.2.2.3 Singleton methods

15.2.2.3.1 Module.constants

`Module.constants`

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array`. Let A be the instance.

b) Let C be the current class or module. Let L be the list which consists of the same elements as the list at the second element from the top of ⟦class-module-list⟧, except the last element, which is the class `Object`.

Let *CS* be the set of classes which consists of C and all the superclasses of C except the class `Object`, but when C is the class `Object`, it shall be included in *CS*. Let *MS* be the set of modules which consists of all the modules in the included module list of all classes in *CS*. Let *CM* be the union of L, *CS* and *MS*.

c) For each class or module c in *CM*, and for each name N of a constant defined in c, take the following steps:

1) Let S be either a new direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which is chosen as the value of S is implementation-defined.

2) Unless A contains the element of the same name as S, when S is an instance of the class `Symbol`, or the same content as S, when S is an instance of the class `String`, insert S to A. The position where S is inserted is implementation-defined.

d) Return A.

15.2.2.3.2 Module.nesting

```
Module.nesting
```

Visibility: public

Behavior: The method returns a new direct instance of the class `Array` which contains all but the last element of the list at the second element from the top of the ⟦class-module-list⟧ in the same order.

15.2.2.4 Instance methods

15.2.2.4.1 Module#<

```
<( other )
```

Visibility: public

Behavior: Let A be *other*. Let R be the receiver of the method.

a) If A is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

b) If A and R are the same object, return **false**.

c) If $R \sqsubset A$, return **true**.

d) If $A \sqsubset R$, return **false**.

e) Otherwise, return **nil**.

15.2.2.4.2 Module#<=

`<=( other )`

Visibility: public

Behavior:

a) If *other* and the receiver are the same object, return **true**.

b) Otherwise, the behavior is the same as the method `<` (see 15.2.2.4.1).

15.2.2.4.3 Module#<=>

`<=>( other )`

Visibility: public

Behavior: Let A be *other*. Let R be the receiver of the method.

a) If A is not an instance of the class `Module`, return **nil**.

b) If A and R are the same object, return an instance of the class `Integer` whose value is 0.

c) If $R \sqsubset A$, return an instance of the class `Integer` whose value is -1.

d) If $A \sqsubset R$, return an instance of the class `Integer` whose value is 1.

e) Otherwise, return **nil**.

15.2.2.4.4 Module#==

`==( other )`

Visibility: public

Behavior: Same as the method `==` of the module `Kernel` (see 15.3.1.3.1).

15.2.2.4.5 Module#===

`===(` *object* `)`

Visibility: public

Behavior: Invoke the method `kind_of?` (see 15.3.1.3.26) of the module `Kernel` on *object* with the receiver as the only argument, and return the resulting value.

15.2.2.4.6 Module#>

`>(` *other* `)`

Visibility: public

Behavior: Let A be *other*. Let R be the receiver of the method.

a) If A is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

b) If A and R are the same object, return **false**.

c) If $R \sqsubset A$, return **false**.

d) If $A \sqsubset R$, return **true**.

e) Otherwise, return **nil**.

15.2.2.4.7 Module#>=

`>=(` *other* `)`

Visibility: public

Behavior:

a) If *other* and the receiver are the same object, return **true**.

b) Otherwise, the behavior is the same as the method `>` (see 15.2.2.4.6).

15.2.2.4.8 Module#alias_method

`alias_method(` *new_name, aliased_name* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *new_name*. Let A be the name designated by *aliased_name*.

b) Take steps d) through h) of 13.3.6, assuming that A, C, and N in 13.3.6 to be A, C, and N in the above steps.

c) Return C.

15.2.2.4.9 Module#ancestors

`ancestors`

Visibility: public

Behavior:

a) Create an empty direct instance A of the class `Array`.

b) Let C be the receiver of the method.

c) If C is a singleton class, the behavior is implementation-defined.

d) Otherwise, append C to the end of A.

e) Append each element of the included module list of C to A in the reverse order.

f) If C has a direct superclass, let new C be the direct superclass of the current C, and repeat from Step c).

g) Return A.

15.2.2.4.10 Module#append_features

`append_features(` *module* `)`

Visibility: private

Behavior: Let *L1* and *L2* be the included module list of the receiver and *module* respectively.

a) If *module* and the receiver are the same object, the behavior is unspecified.

b) If the receiver is an element of *L2*, the behavior is implementation-defined.

c) Otherwise, for each module M in *L1*, in the same order in *L1*, take the following steps:

   1) If M and *module* are the same object, the behavior is unspecified.

   2) If M is not in *L2*, append M to the end of *L2*.

d) Append the receiver to *L2*.

e) Return an implementation-defined value.

15.2.2.4.11 Module#attr

`attr(` *name* `)`

Visibility: private

Behavior: Invoke the method `attr_reader` of the class `Module` (see 15.2.2.4.13) on the receiver with *name* as the only argument, and return the resulting value.

15.2.2.4.12 Module#attr_accessor

`attr_accessor(` `*`*name_list* `)`

Visibility: private

Behavior:

Let C be the method receiver.

a) For each element E of *name_list*, take the following steps:

   1) Let N be the name designated by E.

   2) If N is not of the form *local-variable-identifier* or *constant-identifier*, raise a direct instance of the class `NameError` which has E as its name attribute.

   3) Define an instance method in C as if by evaluating the following method definition at the location of the invocation. In the following method definition, `N` is N, and `@N` is the name which is N prefixed by "`@`".

```
def N
  @N
end
```

   4) Define an instance method in C as if by evaluating the following method definition at the location of the invocation. In the following method definition, `N=` is the name N postfixed by `=`, and `@N` is the name which is N prefixed by "`@`". The choice of the parameter name is arbitrary, and `val` is chosen only for the expository purpose.

```
def N=(val)
  @N = val
end
```

b) Return an implementation-defined value.

15.2.2.4.13 Module#attr_reader

`attr_reader(` **name_list* `)`

Visibility: private

Behavior: The method takes the same steps as the method `attr_accessor` [see 15.2.2.4.12, except Step a) 4)] of the class `Module`.

15.2.2.4.14 Module#attr_writer

`attr_writer(` **name_list* `)`

Visibility: private

Behavior: The method takes the same steps as the method `attr_accessor` of the class `Module` (see 15.2.2.4.12), except Step a) 3).

15.2.2.4.15 Module#class_eval

`class_eval(` *string*`=nil,` *&block* `)`

Visibility: public

Behavior:

a) Let M be the receiver.

b) If *block* is given:

 1) If *string* is given, raise a direct instance of the class `ArgumentError`.

 2) Call *block* with implementation-defined arguments as described in 11.3.3, and let V be the resulting value. A conforming processor shall modify the execution context just before 11.3.3 d) as follows:

  - Create a new list which has the same members as those of the list at the top of ⟦class-module-list⟧, and add M to the head of the newly created list. Push the list onto ⟦class-module-list⟧.

  - Push the receiver onto ⟦self⟧.

  - Push the public visibility onto ⟦default-method-visibility⟧.

 In 11.3.3 d) and e), a conforming processor shall ignore M which is added to the head of the top of ⟦class-module-list⟧ as described above, except when referring to the current class or module in a *method-definition* (see 13.3.1), an *alias-statement* (see 13.3.6), or an *undef-statement* (see 13.3.7).

3) Return *V*.

c) If *block* is not given:

1) If *string* is not an instance of the class `String`, the behavior is unspecified.

2) Let *E* be the execution context as it exists just before this method invoked.

3) Modify *E* as follows:

- Create a new list which has the same members as those of the list at the top of ⟦class-module-list⟧, and add *M* to the head of the newly created list. Push the list onto ⟦class-module-list⟧.

- Push the receiver onto ⟦self⟧.

- Push the public visibility onto ⟦default-method-visibility⟧.

4) Parse the content of *string* as a *program* (see 10.1). If it fails, raise a direct instance of the class `SyntaxError`.

5) Evaluate the *program* within the execution context *E*. Let *V* be the resulting value of the evaluation.

6) Restore the execution context *E* by removing the elements from the tops of ⟦class-module-list⟧, ⟦self⟧, and ⟦default-method-visibility⟧, even when an exception is raised and not handled in c) 4) or c) 5).

7) Return *V*.

In Step c) 5), a local variable scope which corresponds to the *program* is considered as a local variable scope which corresponds to a *block* in 9.2 d) 1).

15.2.2.4.16 Module#class_variable_defined?

`class_variable_defined?(` *symbol* `)`

Visibility: public

Behavior: Let *C* be the receiver of the method.

a) Let *N* be the name designated by *symbol*.

b) If *N* is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) Search for a binding of the class variable with name *N* by taking steps b) through d) of 11.5.4.5, assuming that *C* and *N* in 11.5.4.5 to be *C* and *N* in the above steps.

d) If a binding is found, return **true**.

e) Otherwise, return **false**.

15.2.2.4.17 Module#class_variable_get

`class_variable_get(` *symbol* `)`

Visibility: implementation-defined

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) Search for a binding of the class variable with name N by taking steps b) through d) of 11.5.4.5, assuming that C and N in 11.5.4.5 to be C and N in the above steps.

d) If a binding is found, return the value of the binding.

e) Otherwise, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

15.2.2.4.18 Module#class_variable_set

`class_variable_set(` *symbol, obj* `)`

Visibility: implementation-defined

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) Search for a binding of the class variable with name N by taking steps b) through d) of 11.5.4.5, assuming that C and N in 11.5.4.5 to be C and N in the above steps.

d) If a binding is found, replace the value of the binding with *obj*.

e) Otherwise, create a variable binding with name N and value *obj* in the set of bindings of class variables of C.

f) Return *obj*.

15.2.2.4.19 Module#class_variables

`class_variables`

Visibility: public

Behavior: The method returns a new direct instance of the class `Array` which consists of names of all class variables of the receiver. These names are represented by direct instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-defined.

15.2.2.4.20 Module#const_defined?

`const_defined?(` *symbol* `)`

Visibility: public

Behavior:

a) Let C be the receiver of the method.

b) Let N be the name designated by *symbol*.

c) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

d) If a binding with name N exists in the set of bindings of constants of C, return **true**.

e) Otherwise, return **false**.

15.2.2.4.21 Module#const_get

`const_get(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) Search for a binding of a constant with name N from Step e) of 11.5.4.2, assuming that C in 11.5.4.2 to be the receiver of the method.

d) If a binding is found, return the value of the binding.

e) Otherwise, return the value of the invocation of the method `const_missing` [see 11.5.4.2 e) 1) i)].

15.2.2.4.22 Module#const_missing

`const_missing(` *symbol* `)`

Visibility: public

Behavior: The method `const_missing` is invoked when a binding of a constant does not exist on a constant reference (see 11.5.4.2).

When the method is invoked, take the following steps:

a) Take steps a) through c) of 15.2.2.4.20.

b) Raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

15.2.2.4.23 Module#const_set

`const_set(` *symbol, obj* `)`

Visibility: public

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding with name N exists in the set of bindings of constants of C, replace the value of the binding with *obj*.

d) Otherwise, create a variable binding with N and value *obj* in the set of bindings of constants of C.

e) Return *obj*.

15.2.2.4.24 Module#constants

`constants`

Visibility: public

Behavior:

The method returns a new direct instance of the class `Array` which consists of names of all constants defined in the receiver. These names are represented by direct instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-defined.

15.2.2.4.25 Module#extend_object

`extend_object(` *object* `)`

Visibility: private

Behavior: Let S be the singleton class of *object*. Invoke the method `append_features` (see 15.2.2.4.10) on the receiver with S as the only argument, and return the resulting value.

15.2.2.4.26 Module#extended

`extended(` *object* `)`

Visibility: private

Behavior: The method returns **nil**.

NOTE The method `extended` is invoked in the method `extend` of the module `Kernel` (see 15.3.1.3.13). The method `extended` can be overriden to hook an invocation of the method `extend`.

15.2.2.4.27 Module#include

`include(` **module_list* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) For each element A of *module_list*, in the reverse order in *module_list*, take the following steps:

   1) If A is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

   2) If A is an instance of the class `Class`, raise a direct instance of the class `TypeError`.

   3) Invoke the method `append_features` (see 15.2.2.4.10) on A with C as the only argument.

   4) Invoke the method `included` (see 15.2.2.4.29) on A with C as the only argument.

b) Return C.

15.2.2.4.28 Module#include?

`include?( module )`

Visibility: public

Behavior: Let C be the receiver of the method.

a) If *module* is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

b) If *module* is an element of the included module list of C, return **true**.

c) Otherwise, if C is an instance of the class `Class`, and if *module* is an element of the included module list of one of the superclasses of C, then return **true**.

d) Otherwise, return **false**.

15.2.2.4.29 Module#included

`included( module )`

Visibility: private

Behavior: The method returns **nil**.

NOTE The method `included` is invoked in the method `include` of the class `Module` (see 15.2.2.4.27). The method `included` can be overriden to hook an invocation of the method `include`.

15.2.2.4.30 Module#included_modules

`included_modules`

Visibility: public

Behavior: Let C be the receiver of the method.

a) Create an empty direct instance A of the class `Array`.

b) Append each element of the included module list of C, in the reverse order, to A.

c) If C is an instance of the class `Class`, and if C has a direct superclass, then let new C be the direct superclass of the current C, and repeat from Step b).

d) Otherwise, return A.

15.2.2.4.31 Module#initialize

`initialize( &block )`

Visibility: private

Behavior:

a) If *block* is given, take step b) of the method `class_eval` of the class `Module` (see 15.2.2.4.15), assuming that *block* in 15.2.2.4.15 to be *block* given to this method.

b) Return an implementation-defined value.

15.2.2.4.32 Module#initialize_copy

`initialize_copy( original )`

Visibility: private

Behavior:

a) Invoke the instance method `initialize_copy` defined in the module `Kernel` on the receiver with *original* as the argument.

b) If the receiver is associated with a singleton class, let E_o be the singleton class, and take the following steps:

    1) Create a singleton class whose direct superclass is the direct superclass of E_o. Let E_n be the singleton class.

    2) For each binding B_{v1} of the constants of E_o, create a variable binding with the same name and value as B_{v1} in the set of bindings of constants of E_n.

    3) For each binding B_{v2} of the class variables of E_o, create a variable binding with the same name and value as B_{v2} in the set of bindings of class variables of E_n.

    4) For each binding B_m of the instance methods of E_o, create a method binding with the same name and value as B_m in the set of bindings of instance methods of E_n.

    5) Associate the receiver with E_n.

c) If the receiver is an instance of the class `Class`:

    1) If *original* has a direct superclass, set the direct superclass of the receiver to the direct superclass of *original*.

    2) Otherwise, the behavior is unspecified.

d) Append each element of the included module list of *original*, in the same order, to the included module list of the receiver.

e) For each binding B_{v3} of the constants of *original*, create a variable binding with the same name and value as B_{v3} in the set of bindings of constants of the receiver.

f) For each binding B_{v4} of the class variables of *original*, create a variable binding with the same name and value as B_{v4} in the set of bindings of class variables of the receiver.

g) For each binding B_{m2} of the instance methods of *original*, create a method binding with the same name and value as B_{m2} in the set of bindings of instance methods of the receiver.

h) Return an implementation-defined value.

15.2.2.4.33 Module#instance_methods

`instance_methods(` *include_super* `=true )`

Visibility: public

Behavior: Let C be the receiver of the method.

a) Create an empty direct instance A of the class `Array`.

b) Let I be the set of bindings of instance methods of C. For each binding B of I, let N be the name of B, and let V be the value of B, and take the following steps:

   1) If V is undef, or the visibility of V is private, skip the next two steps.

   2) Let S be either a new direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which is chosen as the value of S is implementation-defined.

   3) Unless A contains the element of the same name (if S is an instance of the class `Symbol`) or the same content (if S is an instance of the class `String`) as S, append S to A.

c) If *include_super* is a trueish object:

   1) For each module M in included module list of C, take step b), assuming that C in that step to be M.

   2) If C does not have a direct superclass, return A.

   3) Let new C be the direct superclass of C.

   4) Repeat from Step b).

d) Return A.

15.2.2.4.34 Module#method_defined?

`method_defined?(` *symbol* `)`

Visibility: public

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) Search for a binding of an instance method named N starting from C as described in 13.3.4.

c) If a binding is found and its value is not undef, return **true**.

d) Otherwise, return **false**.

15.2.2.4.35 Module#module_eval

`module_eval(` *string* `=nil,` *&block* `)`

Visibility: public

Behavior: Same as the method `class_eval` (see 15.2.2.4.15).

15.2.2.4.36 Module#private

`private(` **symbol_list* `)`

Visibility: private

Behavior: Same as the method `public` (see 15.2.2.4.38), except to let NV be the private visibility in 15.2.2.4.38 a).

15.2.2.4.37 Module#protected

`protected(` **symbol_list* `)`

Visibility: private

Behavior: Same as the method `public` (see 15.2.2.4.38), except to let NV be the protected visibility in 15.2.2.4.38 a).

15.2.2.4.38 Module#public

```
public( *symbol_list )
```

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let NV be the public visibility.

b) If the length of *symbol_list* is 0, change the current visibility to NV and return C.

c) Otherwise, for each element S of *symbol_list*, take the following steps:

   1) Let N be the name designated by S.

   2) Search for a method binding with name N starting from C as described in 13.3.4.

   3) If a binding is found and its value is not undef, let V the value of the binding.

   4) Otherwise, raise a direct instance of the class `NameError` which has S as its name attribute.

   5) If C is the class or module in which the binding is found, change the visibility of V to NV.

   6) Otherwise, define an instance method in C as if by evaluating the following method definition. In the definition, `N` is N. The choice of the parameter name is arbitrary, and `args` is chosen only for the expository purpose.

```
def N(*args)
  super
end
```

   The attributes of the method created by the above definition are initialized as follows:

      i) The class module list is the element at the top of ⟦class-module-list⟧.

      ii) The defined name is the defined name of V.

      iii) The visibility is NV.

d) Return C.

15.2.2.4.39 Module#remove_class_variable

```
remove_class_variable( symbol )
```

Visibility: implementation-defined

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding with name N exists in the set of bindings of class variables of C, let V be the value of the binding.

   1) Remove the binding from the set of bindings of class variables of C.

   2) Return V.

d) Otherwise, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

15.2.2.4.40 Module#remove_const

`remove_const(` *symbol* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding with name N exists in the set of bindings of constants of C, let V be the value of the binding.

   1) Remove the binding from the set of bindings of constants of C.

   2) Return V.

d) Otherwise, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

15.2.2.4.41 Module#remove_method

`remove_method(` **symbol_list* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) For each element *S* of *symbol_list*, in the order in the list, take the following steps:

1) Let *N* be the name designated by *S*.

2) If a binding with name *N* exists in the set of bindings of instance methods of *C*, and if the value of the binding is not undef, then remove the binding from the set.

3) Otherwise, raise a direct instance of the class `NameError` which has *S* as its name attribute. In this case, the remaining elements of *symbol_list* are not processed.

b) Return *C*.

15.2.2.4.42 Module#undef_method

`undef_method(` *`*symbol_list`* `)`

Visibility: private

Behavior: Let *C* be the receiver of the method.

a) For each element *S* of *symbol_list*, in the order in the list, take the following steps:

1) Let *N* be the name designated by *S*.

2) Take steps a) 3) and a) 4) of 13.3.7, assuming that *C* and *N* in 13.3.7 to be *C* and *N* in the above steps, respectively.

b) Return *C*.

15.2.3 Class

15.2.3.1 General description

All classes are instances of the class `Class`. Therefore, behaviors defined in the class `Class` are shared by all classes.

The instance methods `append_features` and `extend_object` of the class `Class` shall be undefined by invoking the method `undef_method` (see 15.2.2.4.42) on the class `Class` with instances of the class `Symbol` whoses names are "append_features" and "extend_object" as the arguments.

NOTE The instance methods `append_features` and `extend_object` are methods for modules. These methods are therefore undefined in the class `Class`, whose instances do not represent modules, but classes.

15.2.3.2 Direct superclass

The class `Module`

15.2.3.3 Instance methods

15.2.3.3.1 Class#initialize

`initialize(` *superclass*`=Object,` `&`*block*`)`

Visibility: private

Behavior:

a) If the receiver has its direct superclass, or is the root of the class inheritance tree, then raise a direct instance of the class `TypeError`.

b) If *superclass* is not an instance of the class `Class`, raise a direct instance of the class `TypeError`.

c) If *superclass* is a singleton class or the class `Class`, the behavior is unspecified.

d) Set the direct superclass of the receiver to *superclass*.

e) Create a singleton class, and associate it with the receiver. The singleton class shall have the singleton class of *superclass* as one of its superclasses.

f) If *block* is given, take step b) of the method `class_eval` of the class `Module` (see 15.2.2.4.15), assuming that *block* in 15.2.2.4.15 to be *block* given to this method.

g) Return an implementation-defined value.

15.2.3.3.2 Class#initialize_copy

`initialize_copy(` *original*`)`

Visibility: private

Behavior:

a) If the direct superclass of the receiver has already been set, or if the receiver is the root of the class inheritance tree, then raise a direct instance of the class `TypeError`.

b) If the receiver is a singleton class, raise a direct instance of the class `TypeError`.

c) Invoke the instance method `initialize_copy` defined in the class `Module` on the receiver with *original* as the argument.

d) Return an implementation-defined value.

15.2.3.3.3 Class#new

`new(` `*`*args*`,` `&`*block*`)`

Visibility: public

Behavior:

a) If the receiver is a singleton class, raise a direct instance of the class `TypeError`.

b) Create a direct instance of the receiver which has no bindings of instance variables. Let O be the newly created instance.

c) Invoke the method `initialize` on O with all the elements of *args* as arguments and *block* as the block.

d) Return O.

15.2.3.3.4 Class#superclass

`superclass`

Visibility: public

Behavior: Let C be the receiver of the method.

a) If C is a singleton class, return an implementation-defined value.

b) If C does not have a direct superclass, return **nil**.

c) Otherwise, return the direct superclass of C.

15.2.4 NilClass

15.2.4.1 General description

The class `NilClass` has only one instance **nil** (see 6.6).

Instances of the class `NilClass` shall not be created by the method `new` of the class `NilClass`. Therefore, the singleton method `new` of the class `NilClass` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `NilClass` with a direct instance of the class `Symbol` whose name is "new" as the argument.

15.2.4.2 Direct superclass

The class `Object`

15.2.4.3 Instance methods

15.2.4.3.1 NilClass#&

`&(` *other* `)`

Visibility: public

Behavior: The method returns **false**.

15.2.4.3.2 NilClass#^

`^( *other* )`

Visibility: public

Behavior:

a) If *other* is a falseish object, return **false**.

b) Otherwise, return **true**.

15.2.4.3.3 NilClass#|

`|( *other* )`

Visibility: public

Behavior:

a) If *other* is a falseish object, return **false**.

b) Otherwise, return **true**.

15.2.4.3.4 NilClass#nil?

`nil?`

Visibility: public

Behavior: The method returns **true**.

15.2.4.3.5 NilClass#to_s

`to_s`

Visibility: public

Behavior: The method creates an empty direct instance of the class `String`, and returns this instance.

15.2.5 TrueClass

15.2.5.1 General description

The class `TrueClass` has only one instance **true** (see 6.6).

Instances of the class `TrueClass` shall not be created by the method `new` of the class `TrueClass`. Therefore, the singleton method `new` of the class `TrueClass` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `TrueClass` with a direct instance of the class `Symbol` whose name is “new” as the argument.

15.2.5.2 Direct superclass

The class `Object`

15.2.5.3 Instance methods

15.2.5.3.1 TrueClass#&

`&(` *other* `)`

Visibility: public

Behavior:

a) If *other* is a falseish object, return **false**.

b) Otherwise, return **true**.

15.2.5.3.2 TrueClass#^

`^(` *other* `)`

Visibility: public

Behavior:

a) If *other* is a falseish object, return **true**.

b) Otherwise, return **false**.

15.2.5.3.3 TrueClass#to_s

`to_s`

Visibility: public

Behavior: The method creates a direct instance of the class `String`, the content of which is “true”, and returns this instance.

15.2.5.3.4 TrueClass#|

```
|( other )
```

Visibility: public

Behavior: The method returns **true**.

15.2.6 FalseClass

15.2.6.1 General description

The class `FalseClass` has only one instance **false** (see 6.6).

Instances of the class `FalseClass` shall not be created by the method `new` of the class `FalseClass`. Therefore, the singleton method `new` of the class `FalseClass` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `FalseClass` with a direct instance of the class `Symbol` whose name is "new" as the argument.

15.2.6.2 Direct superclass

The class `Object`

15.2.6.3 Instance methods

15.2.6.3.1 FalseClass#&

```
&( other )
```

Visibility: public

Behavior: The method returns **false**.

15.2.6.3.2 FalseClass#^

```
^( other )
```

Visibility: public

Behavior:

a) If *other* is a falseish object, return **false**.

b) Otherwise, return **true**.

15.2.6.3.3 FalseClass#to_s

`to_s`

Visibility: public

Behavior: The method creates a direct instance of the class `String`, the content of which is "false", and returns this instance.

15.2.6.3.4 FalseClass#|

`|(` *other* `)`

Visibility: public

Behavior:

a) If *other* is a falseish object, return **false**.

b) Otherwise, return **true**.

15.2.7 Numeric

15.2.7.1 General description

Instances of the class `Numeric` represent numbers. The class `Numeric` is the superclass of all the other built-in classes which represent numbers.

The notation "the value of the instance N of the class `Numeric`" means the number represented by N.

15.2.7.2 Direct superclass

The class `Object`

15.2.7.3 Included modules

The following module is included in the class `Numeric`.

- `Comparable`

15.2.7.4 Instance methods

15.2.7.4.1 Numeric#+@

`+@`

Visibility: public

Behavior: The method returns the receiver.

15.2.7.4.2 Numeric#−@

```
-@
```

Visibility: public

Behavior:

a) Invoke the method `coerce` on the receiver with an instance of the class `Integer` whose value is 0 as the only argument. Let V be the resulting value.

   1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

      i) Invoke the method `-` on F with S as the only argument.

      ii) Return the resulting value.

   2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.7.4.3 Numeric#abs

```
abs
```

Visibility: public

Behavior:

a) Invoke the method `<` on the receiver with an instance of the class `Integer` whose value is 0 as an argument.

b) If this invocation results in a trueish object, invoke the method `-@` on the receiver and return the resulting value.

c) Otherwise, return the receiver.

15.2.7.4.4 Numeric#coerce

coerce(*other*)

Visibility: public

Behavior:

a) If the class of the receiver and the class of *other* are the same class, let X and Y be *other* and the receiver, respectively.

b) Otherwise, let X and Y be instances of the class `Float` which are converted from *other* and the receiver, respectively. *other* and the receiver are converted as follows:

1) Let O be *other* or the receiver.

2) If O is an instance of the class `Float`, let F be O.

3) Otherwise:

i) If an invocation of the method `respond_to?` on O with a direct instance of the class `Symbol` whose name is `to_f` as the argument results in a falseish object, raise a direct instance of the class `TypeError`.

ii) Invoke the method `to_f` on O with no arguments, and let F be the resulting value.

iii) If F is not an instance of the class `Float`, raise a direct instance of the class `TypeError`.

4) If the value of F is NaN, the behavior is unspecified.

5) The converted value of O is F.

c) Create a direct instance of the class `Array` which consists of two elements: the first is X; the second is Y.

d) Return the instance of the class `Array`.

15.2.8 Integer

15.2.8.1 General description

Instances of the class `Integer` represent integers. The ranges of these integers are unbounded. However the actual values computable depend on resource limitations, and the behavior when the resource limits are exceeded is implementation-defined.

Instances of the class `Integer` shall not be created by the method `new` of the class `Integer`. Therefore, the singleton method `new` of the class `Integer` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `Integer` with a direct instance of the class `Symbol` whose name is "new" as the argument.

Subclasses of the class `Integer` may be defined as built-in classes. In this case:

- The class `Integer` shall not have its direct instances. Instead of a direct instance of the class `Integer`, a direct instance of a subclass of the class `Integer` shall be created.

- Instance methods of the class `Integer` need not be defined in the class `Integer` itself if the instance methods are defined in all subclasses of the class `Integer`.

- For each subclass of the class `Integer`, the ranges of the values of its instances may be bounded.

15.2.8.2 Direct superclass

The class `Numeric`

15.2.8.3 Instance methods

15.2.8.3.1 Integer#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer` whose value is the sum of the values of the receiver and *other*.

b) If *other* is an instance of the class `Float`, let R be the value of the receiver as a floating-point number.

Return a direct instance of the class `Float` whose value is the sum of R and the value of *other*.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `+` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.8.3.2 Integer#−

`-(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer` whose value is the result of subtracting the value of *other* from the value of the receiver.

b) If *other* is an instance of the class `Float`, let R be the value of the receiver as a floating-point number.

Return a direct instance of the class `Float` whose value is the result of subtracting the value of *other* from R.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `-` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.8.3.3 Integer#*

`*(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer` whose value is the result of multiplication of the values of the receiver and *other*.

b) If *other* is an instance of the class `Float`, let R be the value of the receiver as a floating-point number.

Return a direct instance of the class `Float` whose value is the result of multiplication of R and the value of *other*.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `*` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.8.3.4 Integer#/

`/(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`:

1) If the value of *other* is 0, raise a direct instance of the class `ZeroDivisionError`.

2) Otherwise, let n be the value of the receiver divided by the value of *other*. Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to n.

NOTE The behavior is the same even if the receiver has a negative value. For exampple, `-5 / 2` returns `-3`.

b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `/` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.8.3.5 Integer#%

`%(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`:

1) If the value of *other* is 0, raise a direct instance of the class `ZeroDivisionError`.

2) Otherwise, let x and y be the values of the receiver and *other*.

i) Let t be the largest integer smaller than or equal to x divided by y.

ii) Let m be $x - t \times y$.

iii) Otherwise, return an instance of the class `Integer` whose value is m.

b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `%` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.8.3.6 Integer#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`:

1) If the value of the receiver is larger than the value of *other*, return an instance of the class `Integer` whose value is 1.

2) If the values of the receiver and *other* are the same integer, return an instance of the class `Integer` whose value is 0.

3) If the value of the receiver is smaller than the value of *other*, return an instance of the class `Integer` whose value is -1.

b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `<=>` on F with S as the only argument.

ii) If this invocation does not result in an instance of the class `Integer`, the behavior is unspecified.

iii) Otherwise, return the value of this invocation.

2) Otherwise, return **nil**.

15.2.8.3.7 Integer#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer`:

1) If the values of the receiver and *other* are the same integer, return **true**.

2) Otherwise, return **false**.

b) Otherwise, invoke the method `==` on *other* with the receiver as the argument. Return the resulting value of this invocation.

15.2.8.3.8 Integer#~

`~`

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose two's complement representation is the one's complement of the two's complement representation of the receiver.

15.2.8.3.9 Integer#&

`&(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, return an instance of the class `Integer` whose two's complement representation is the bitwise AND of the two's complement representations of the receiver and *other*.

15.2.8.3.10 Integer#|

`|(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, return an instance of the class `Integer` whose two's complement representation is the bitwise inclusive OR of the two's complement representations of the receiver and *other*.

15.2.8.3.11 Integer#^

`^( other )`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, return an instance of the class `Integer` whose two's complement representation is the bitwise exclusive OR of the two's complement representations of the receiver and *other*.

15.2.8.3.12 Integer#<<

`<<( other )`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, let x and y be the values of the receiver and *other*.

c) Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to $x \times 2^{y}$.

15.2.8.3.13 Integer#>>

`>>( other )`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, let x and y be the values of the receiver and *other*.

c) Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to $x \times 2^{-y}$.

15.2.8.3.14 Integer#ceil

`ceil`

Visibility: public

Behavior: The method returns the receiver.

15.2.8.3.15 Integer#downto

`downto(`*num*, `&`*block*`)`

Visibility: public

Behavior:

a) If *num* is not an instance of the class `Integer`, or *block* is not given, the behavior is unspecified.

b) Let i be the value of the receiver.

c) If i is smaller than the value of *num*, return the receiver.

d) Call *block* with an instance of the class `Integer` whose value is i.

e) Decrement i by 1 and continue processing from Step c).

15.2.8.3.16 Integer#eql?

`eql?(`*other*`)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Integer`, return **false**.

b) Otherwise, invoke the method `==` on *other* with the receiver as the argument.

c) If this invocation results in a trueish object, return **true**. Otherwise, return **false**.

15.2.8.3.17 Integer#floor

`floor`

Visibility: public

Behavior: The method returns the receiver.

15.2.8.3.18 Integer#hash

```
hash
```

Visibility: public

Behavior: The method returns an implementation-defined instance of the class `Integer`, which satisfies the following condition:

a) Let I_1 and I_2 be instances of the class `Integer`.

b) Let H_1 and H_2 be the resulting values of invocations of the method `hash` on I_1 and I_2, respectively.

c) The values of H_1 and H_2 shall be the same integer, if the values of I_1 and I_2 are the same integer.

15.2.8.3.19 Integer#next

```
next
```

Visibility: public

Behavior: The method returns an instance of the class `Integer`, whose value is the value of the receiver plus 1.

15.2.8.3.20 Integer#round

```
round
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.3.21 Integer#succ

```
succ
```

Visibility: public

Behavior: Same as the method `next` (see 15.2.8.3.19).

15.2.8.3.22 Integer#times

`times(&block)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Let i be 0.

c) If i is larger than or equal to the value of the receiver, return the receiver.

d) Call *block* with an instance of the class `Integer` whose value is i as an argument.

e) Increment i by 1 and continue processing from Step c).

15.2.8.3.23 Integer#to_f

`to_f`

Visibility: public

Behavior: The method returns a direct instance of the class `Float` whose value is the value of the receiver as a floating-point number.

15.2.8.3.24 Integer#to_i

`to_i`

Visibility: public

Behavior: The method returns the receiver.

15.2.8.3.25 Integer#to_s

`to_s`

Visibility: public

Behavior: The method returns a direct instance of the class `String` whose content satisfy the following conditions:

- If the value of the receiver is negative, the first character is the character "-" (0x2d).
- The sequence R of the rest of characters represents the magnitude M of the value of the receiver in base 10. If M is 0, R is a single "0". Otherwise, the first character of R is not "0".

EXAMPLE 1 `123.to_s` returns `"123"`.

EXAMPLE 2 `-123.to_s` returns `"-123"`.

15.2.8.3.26 Integer#truncate

`truncate`

Visibility: public

Behavior: The method returns the receiver.

15.2.8.3.27 Integer#upto

`upto(` *num*, `&`*block* `)`

Visibility: public

Behavior:

a) If *num* is not an instance of the class `Integer`, or *block* is not given, the behavior is unspecified.

b) Let i be the value of the receiver.

c) If i is larger than the value of *num*, return the receiver.

d) Call *block* with an instance of the class `Integer` whose value is i.

e) Increment i by 1 and continue processing from Step c).

15.2.9 Float

15.2.9.1 General description

Instances of the class `Float` represent floating-point numbers.

The precision of the value of an instance of the class `Float` is implementation-defined; however, if the underlying system of a conforming processor supports IEC 60559, the representation of an instance of the class `Float` shall be the 64-bit double format as specified in IEC 60559, 3.2.2.

When an arithmetic operation involving floating-point numbers results in a value which cannot be represented exactly as an instance of the class `Float`, the result is rounded to the nearest representable value. If the two nearest representable values are equally near, which is chosen is implementation-defined.

If the underlying system of a conforming processor supports IEC 60559:

- If an arithmetic operation involving floating-point numbers results in NaN while invoking a method of the class `Float`, the behavior of the method is unspecified.

Instances of the class `Float` shall not be created by the method `new` of the class `Float`. Therefore, the singleton method `new` of the class `Float` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `Float` with a direct instance of the class `Symbol` whose name is “new” as the argument.

15.2.9.2 Direct superclass

The class `Numeric`

15.2.9.3 Instance methods

15.2.9.3.1 Float#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float` whose value is the sum of the values of the receiver and *other*.

b) If *other* is an instance of the class `Integer`, let R be the value of *other* as a floating-point number.

Return a direct instance of the class `Float` whose value is the sum of R and the value of the receiver.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `+` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.9.3.2 Float#−

`-(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float` whose value is the result of subtracting the value of *other* from the value of the receiver.

b) If *other* is an instance of the class `Integer`, let R be the value of *other* as a floating-point number.

   Return a direct instance of the class `Float` whose value is the result of subtracting R from the value of the receiver.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

   1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

      i) Invoke the method `-` on F with S as the only argument.

      ii) Return the resulting value.

   2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.9.3.3 Float#*

`*(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float` whose value is the result of multiplication of the values of the receiver and *other*.

b) If *other* is an instance of the class `Integer`, let R be the value of *other* as a floating-point number.

   Return a direct instance of the class `Float` whose value is the result of multiplication of R and the value of the receiver.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

   1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

      i) Invoke the method `*` on F with S as the only argument.

      ii) Return the resulting value.

   2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.9.3.4 Float#/

`/( other )`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float` whose value is the value of the receiver divided by the value of *other*.

b) If *other* is an instance of the class `Integer`, let R be the value of *other* as a floating-point number.

   Return a direct instance of the class `Float` whose value is the value of the receiver divided by R.

c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

   1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

      i) Invoke the method `/` on F with S as the only argument.

      ii) Return the resulting value.

   2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.9.3.5 Float#%

`%( other )`

Visibility: public

Behavior: In the following steps, binary operators $+$, $-$, and $*$ represent floating-point arithmetic operations addition, subtraction, and multiplication which are used in the instance methods `+`, `-`, and `*` of the class `Float`, respectively. The operator $*$ has a higher precedence than the operators $+$ and $-$.

a) If *other* is an instance of the class `Integer` or the class `Float`:

   Let x be the value of the receiver.

   1) If *other* is an instance of the class `Float`, let y be the value of *other*. If *other* is an instance of the class `Integer`, let y be the value of *other* as a floating-point number.

      i) Let t be the largest integer smaller than or equal to x divided by y.

      ii) Let m be $x - t * y$.

iii) If $m * y < 0$, return a direct instance of the class `Float` whose value is $m + y$.

iv) Otherwise, return a direct instance of the class `Float` whose value is m.

b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `%` on F with S as the only argument.

ii) Return the resulting value.

2) Otherwise, raise a direct instance of the class `TypeError`.

15.2.9.3.6 Float#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Integer` or the class `Float`:

1) Let a be the value of the receiver. If *other* is an instance of the class `Float`, let b be the value of *other*. Otherwise, let b be the value of *other* as a floating-point number.

2) If a conforming processor supports IEC 60559, and if a or b is NaN, then return an implementation-defined value.

3) If $a > b$, return an instance of the class `Integer` whose value is 1.

4) If $a = b$, return an instance of the class `Integer` whose value is 0.

5) If $a < b$, return an instance of the class `Integer` whose value is -1.

b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument. Let V be the resulting value.

1) If V is an instance of the class `Array` which contains two elements, let F and S be the first and the second element of V respectively.

i) Invoke the method `<=>` on F with S as the only argument.

ii) If this invocation does not result in an instance of the class `Integer`, the behavior is unspecified.

iii) Otherwise, return the value of this invocation.

2) Otherwise, return **nil**.

15.2.9.3.7 Float#==

`==( other )`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Float`:

1) If a conforming processor supports IEC 60559, and if the value of the receiver is NaN, then return **false**.

2) If the values of the receiver and *other* are the same number, return **true**.

3) Otherwise, return **false**.

b) If *other* is an instance of the class `Integer`:

1) If the values of the receiver and *other* are the mathematically the same, return **true**.

2) Otherwise, return **false**.

c) Otherwise, invoke the method **==** on *other* with the receiver as the argument and return the resulting value of this invocation.

15.2.9.3.8 Float#ceil

`ceil`

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the smallest integer larger than or equal to the value of the receiver.

15.2.9.3.9 Float#finite?

`finite?`

Visibility: public

Behavior:

a) If the value of the receiver is a finite number, return **true**.

b) Otherwise, return **false**.

15.2.9.3.10 Float#floor

```
floor
```

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the largest integer smaller than or equal to the value of the receiver.

15.2.9.3.11 Float#infinite?

```
infinite?
```

Visibility: public

Behavior:

a) If the value of the receiver is the positive infinite, return an instance of the class `Integer` whose value is 1.

b) If the value of the receiver is the negative infinite, return an instance of the class `Integer` whose value is -1.

c) Otherwise, return **nil**.

15.2.9.3.12 Float#round

```
round
```

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the nearest integer to the value of the receiver. If there are two integers equally distant from the value of the receiver, the one which has the larger absolute value is chosen.

15.2.9.3.13 Float#to_f

```
to_f
```

Visibility: public

Behavior: The method returns the receiver.

15.2.9.3.14 Float#to_i

```
to_i
```

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the integer part of the receiver.

15.2.9.3.15 Float#truncate

```
truncate
```

Visibility: public

Behavior: Same as the method `to_i` (see 15.2.9.3.14).

15.2.10 String

15.2.10.1 General description

Instances of the class `String` represent sequences of characters. The sequence of characters represented by an instance of the class `String` is called the ***content*** of that instance.

An instance of the class `String` which does not contain any character is said to be ***empty***. An instance of the class `String` shall be empty when it is created by Step b) of the method `new` of the class `Class`.

The notation "an instance of the class `Object` which represents the character C" means either of the following:

- An instance of the class `Integer` whose value is the character code of C.
- An instance of the class `String` whose content is the single character C.

A conforming processor shall choose one of the above representations and use the same representation wherever this notation is used.

Characters of an instance of the class `String` have their indices counted up from 0. The notation "the nth character of an instance of the class `String`" means the character of the instance whose index is n.

15.2.10.2 Direct superclass

The class `Object`

15.2.10.3 Included modules

The following modules are included in the class `String`.

- `Comparable`

15.2.10.4 Upper-case and lower-case characters

Some methods of the class `String` handle upper-case and lower-case characters. The correspondence between upper-case and lower-case characters is given in Table 3.

Table 3 – The correspondence between upper-case and lower-case characters

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

15.2.10.5 Instance methods

15.2.10.5.1 String#*

`*(` *num* `)`

Visibility: public

Behavior:

a) If *num* is not an instance of the class `Integer`, the behavior is unspecified.

b) Let n be the value of the *num*.

c) If n is smaller than 0, raise a direct instance of the class `ArgumentError`.

d) Otherwise, let C be the content of the receiver.

e) Create a direct instance S of the class `String` the content of which is C repeated n times.

f) Return S.

15.2.10.5.2 String#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `String`, the behavior is unspecified.

b) Let S and O be the contents of the receiver and the *other* respectively.

c) Return a new direct instance of the class `String` the content of which is the concatenation of S and O.

15.2.10.5.3 String#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `String`, the behavior is unspecified.

b) Let *S1* and *S2* be the contents of the receiver and the *other* respectively.

c) If both *S1* and *S2* are empty, return an instance of the class `Integer` whose value is 0.

d) Otherwise, if *S1* is empty, return an instance of the class `Integer` whose value is -1.

e) Otherwise, if *S2* is empty, return an instance of the class `Integer` whose value is 1.

f) Let a, b be the character codes of the first characters of *S1* and *S2* respectively.

1) If $a > b$, return an instance of the class `Integer` whose value is 1.

2) If $a < b$, return an instance of the class `Integer` whose value is -1.

3) Otherwise, let new *S1* and *S2* be *S1* and *S2* excluding their first characters, respectively. Continue processing from Step c).

15.2.10.5.4 String#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `String`, the behavior is unspecified.

b) If *other* is an instance of the class `String`:

1) If the contents of the receiver and *other* are the same, return **true**.

2) Otherwise, return **false**.

15.2.10.5.5 String#=~

`=~(` *regexp* `)`

Visibility: public

Behavior:

a) If *regexp* is not an instance of the class `Regexp`, the behavior is unspecified.

b) Otherwise, invoke the method `match` on *regexp* with the receiver as the argument (see 15.2.15.7.7), and return the resulting value.

15.2.10.5.6 String#[]

`[](` `*`*args* `)`

Visibility: public

Behavior:

a) If the length of *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.

b) Let P be the first element of *args*. Let n be the length of the receiver.

c) If P is an instance of the class `Integer`, let b be the value of P.

1) If the length of *args* is 1:

i) If b is smaller than 0, increment b by n. If b is still smaller than 0, return **nil**.

ii) If $b \geq n$, return **nil**.

iii) Create an instance of the class `Object` which represents the bth character of the receiver and return this instance.

2) If the length of *args* is 2:

i) If the last element of *args* is an instance of the class `Integer`, let l be the value of the instance. Otherwise, the behavior is unspecified.

ii) If l is smaller than 0, or b is larger than n, return **nil**.

iii) If b is smaller than 0, increment b by n. If b is still smaller than 0, return **nil**.

iv) If $b + l$ is larger than n, let l be $n - b$.

v) If l is smaller than or equal to 0, create an empty direct instance of the class `String` and return the instance.

vi) Otherwise, create a direct instance of the class `String` whose content is the $(n-l)$ characters of the receiver, from the bth index, preserving their order. Return the instance.

d) If P is an instance of the class `Regexp`:

1) If the length of *args* is 1, let i be 0.

2) If the length of *args* is 2, and the last element of *args* is an instance of the class `Integer`, let i be the value of the instance. Otherwise, the behavior is unspecified.

3) Test if the pattern of P matches the content of the receiver. (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

4) If M is **nil**, return **nil**.

5) If i is larger than the length of the match result attribute of M, return **nil**.

6) If i is smaller than 0, increment i by the length of the match result attribute of M. If i is still smaller than or equal to 0, return **nil**.

7) Let m be the ith element of the match result attribute of M. Create a direct instance of the class `String` whose content is the substring of m and return the instance.

e) If P is an instance of the class `String`:

1) If the length of *args* is 2, the behavior is unspecified.

2) If the receiver includes the content of P as a substring, create a direct instance of the class `String` whose content is equal to the content of P and return the instance.

3) Otherwise, return **nil**.

f) Otherwise, the behavior is unspecified.

15.2.10.5.7 String#capitalize

```
capitalize
```

Visibility: public

Behavior: The method returns a new direct instance of the class `String` which contains all the characters of the receiver, except:

- If the first character of the receiver is a lower-case character, the first character of the resulting instance is the corresponding upper-case character.
- If the ith character of the receiver (where $i > 0$) is an upper case character, the ith character of the resulting instance is the corresponding lower-case character.

15.2.10.5.8 String#capitalize!

```
capitalize!
```

Visibility: public

Behavior:

a) Let s be the content of the instance of the class `String` returned when the method `capitalize` is invoked on the receiver.

b) If the content of the receiver and s are the same, return **nil**. Otherwise, change the content of the receiver to s, and return the receiver.

15.2.10.5.9 String#chomp

```
chomp( rs="\n" )
```

Visibility: public

Behavior:

a) If *rs* is **nil**, return a new direct instance of the class `String` whose content is the same as the receiver.

b) If the receiver is empty, return a new empty direct instance of the class `String`.

c) If *rs* is not an instance of the class `String`, the behavior is unspecified.

d) Otherwise, return a new direct instance of the class `String` whose content is the same as the receiver, except the following characters:

1) If *rs* consists of only one character 0x0a, the *line-terminator* on the end, if any, is excluded.

2) If *rs* is empty, a sequence of *line-terminator*s on the end, if any, is excluded.

3) Otherwise, if the receiver ends with the content of *rs*, this sequence of characters at the end of the receiver is excluded.

15.2.10.5.10 String#chomp!

`chomp!(` *rs* `="\n")`

Visibility: public

Behavior:

a) Let *s* be the content of the instance of the class `String` returned when the method `chomp` is invoked on the receiver with *rs* as the argument.

b) If the content of the receiver and *s* are the same, return **nil**. Otherwise, change the content of the receiver to *s*, and return the receiver.

15.2.10.5.11 String#chop

`chop`

Visibility: public

Behavior:

a) If the receiver is empty, return a new empty direct instance of the class `String`.

b) Otherwise, create a new direct instance of the class `String` whose content is the receiver without the last character and return this instance. If the last character is 0x0a, and the character just before the 0x0a is 0x0d, the 0x0d is also dropped.

15.2.10.5.12 String#chop!

```
chop!
```

Visibility: public

Behavior:

a) Let s be the content of the instance of the class `String` returned when the method `chop` is invoked on the receiver.

b) If the content of the receiver and s are the same, return **nil**. Otherwise, change the content of the receiver to s, and return the receiver.

15.2.10.5.13 String#downcase

```
downcase
```

Visibility: public

Behavior: The method returns a new direct instance of the class `String` which contains all the characters of the receiver, with the upper-case characters replaced with the corresponding lower-case characters.

15.2.10.5.14 String#downcase!

```
downcase!
```

Visibility: public

Behavior:

a) Let s be the content of the instance of the class `String` returned when the method `downcase` is invoked on the receiver.

b) If the content of the receiver and s are the same, return **nil**. Otherwise, change the content of the receiver to s, and return the receiver.

15.2.10.5.15 String#each_line

`each_line(&`*block*`)`

Visibility: public

Behavior: Let s be the content of the receiver. Let c be the first character of s.

a) If *block* is not given, the behavior is unspecified.

b) Find the first 0x0a in s from c. If there is such a 0x0a:

1) Let d be that 0x0a.

2) Create a direct instance S of the class `String` whose content is a sequence of characters from c to d.

3) Call *block* with S as the argument.

4) If d is the last character of s, return the receiver. Otherwise, let new c be the character just after d and continue processing from Step b).

c) If there is not such a 0x0a, create a direct instance of the class `String` whose content is a sequence of characters from c to the last character of s. Call *block* with this instance as the argument.

d) Return the receiver.

15.2.10.5.16 String#empty?

`empty?`

Visibility: public

Behavior:

a) If the receiver is empty, return **true**.

b) Otherwise, return **false**.

15.2.10.5.17 String#eql?

`eql?(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `String`:

1) If the contents of the receiver and *other* are the same, return **true**.

2) Otherwise, return **false**.

b) If *other* is not an instance of the class `String`, return **false**.

15.2.10.5.18 String#gsub

`gsub(*`*args*`, &`*block*`)`

Visibility: public

Behavior:

a) If the length of *args* is 0 or larger than 2, or the length of *args* is 1 and *block* is not given, raise a direct instance of the class `ArgumentError`.

b) Let P be the first element of *args*. If P is not an instance of the class `Regexp`, or the length of *args* is 2 and the last element of *args* is not an instance of the class `String`, the behavior is unspecified.

c) Let S be the content of the receiver, and let l be the length of S.

d) Let L be an empty list and let n be an integer 0.

e) Test if the pattern of P matches S from the index n (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

f) If M is **nil**, append to L the substring of S beginning at the nth character up to the last character of S.

g) Otherwise:

1) If the length of *args* is 1:

i) Call *block* with a new direct instance of the class `String` whose content is the matched substring of M as the argument.

ii) Let V be the resulting value of this call. If V is not an instance of the class `String`, the behavior is unspecified.

2) Let *pre* be the pre-match (see 15.2.16.1) of M. Append to L the substring of *pre* beginning at the nth character up to the last character of *pre*, unless n is larger than the index of the last character of *pre*.

3) If the length of *args* is 1, append the content of V to L. If the length of *args* is 2, append to L the content of the last element of *args*.

4) Let *post* be the post-match (see 15.2.16.1) of M. Let i be the index of the first character of *post* within S.

i) If i is equal to n, i.e. if P matched an empty string:

I) Append to L a new direct instance of the class `String` whose content is the ith character of S.

II) Increment n by 1.

ii) Otherwise, let new n be i.

5) If $n < l$, continue processing from Step e).

h) Create a direct instance of the class `String` whose content is the concatenation of all the elements of L, and return the instance.

15.2.10.5.19 String#gsub!

`gsub!(` `*`*args*, `&`*block* `)`

Visibility: public

Behavior:

a) Let s be the content of the instance of the class `String` returned when the method `gsub` is invoked on the receiver with the same arguments.

b) If the content of the receiver and s are the same, return **nil**. Otherwise, change the content of the receiver to s, and return the receiver.

15.2.10.5.20 String#hash

`hash`

Visibility: public

Behavior: The method returns an implementation-defined instance of the class `Integer` which satisfies the following condition:

a) Let S_1 and S_2 be two distinct instances of the class `String`.

b) Let H_1 and H_2 be the resulting values of the invocations of the method `hash` on S_1 and S_2 respectively.

c) If S_1 and S_2 has the same content, the values of H_1 and H_2 shall be the same integer.

15.2.10.5.21 String#include?

`include?(` *obj* `)`

Visibility: public

Behavior:

a) If *obj* is an instance of the class `Integer`:

If the receiver includes the character whose character code is the value of *obj*, return **true**. Otherwise, return **false**.

b) If *obj* is an instance of the class `String`:

If there exists a substring of the receiver whose sequence of characters is the same as the content of *obj*, return **true**. Otherwise, return **false**.

c) Otherwise, the behavior is unspecified.

15.2.10.5.22 String#index

`index(` *substring, offset*`=`*0* `)`

Visibility: public

Behavior:

a) If *substring* is not an instance of the class `String`, the behavior is unspecified.

b) Let R and S be the contents of the receiver and *substring*, respectively.

c) If *offset* is not an instance of the class `Integer`, the behavior is unspecified.

d) Let n be the value of *offset*.

e) If n is larger than or equal to 0, let O be n.

f) Otherwise, let O be $l + n$, where l is the length of S.

g) If O is smaller than 0, return **nil**.

h) If S appears as a substring of R at one or more positions whose index is larger than or equal to O, return an instance of the class `Integer` whose value is the index of the first such position.

i) Otherwise, return **nil**.

15.2.10.5.23 String#initialize

`initialize(` *str*`=""` `)`

Visibility: private

Behavior:

a) If *str* is not an instance of the class `String`, the behavior is unspecified.

b) Otherwise, initialize the content of the receiver to the same sequence of characters as the content of *str*.

c) Return an implementation-defined value.

15.2.10.5.24 String#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class `String`, the behavior is unspecified.

b) If *original* is an instance of the class `String`, change the content of the receiver to the content of *original*.

c) Return an implementation-defined value.

15.2.10.5.25 String#intern

`intern`

Visibility: public

Behavior:

a) If the length of the receiver is 0, or if the receiver contains 0x00, then the behavior is unspecified.

b) Otherwise, return a direct instance of the class `Symbol` whose name is the content of the receiver.

15.2.10.5.26 String#length

`length`

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the number of characters of the content of the receiver.

15.2.10.5.27 String#match

`match(` *regexp* `)`

Visibility: public

Behavior:

a) If *regexp* is an instance of the class `Regexp`, let R be *regexp*.

b) Otherwise, if *regexp* is an instance of the class `String`, create a direct instance of the class `Regexp` by invoking the method `new` on the class `Regexp` with *regexp* as the argument. Let R be the instance of the class `Regexp`.

c) Otherwise, the behavior is unspecified.

d) Invoke the method `match` on R with the receiver as the argument.

e) Return the resulting value of the invocation.

15.2.10.5.28 String#replace

`replace(` *other* `)`

Visibility: public

Behavior: Same as the method `initialize_copy` (see 15.2.10.5.24).

15.2.10.5.29 String#reverse

`reverse`

Visibility: public

Behavior: The method returns a new direct instance of the class `String` which contains all the characters of the content of the receiver in the reverse order.

15.2.10.5.30 String#reverse!

`reverse!`

Visibility: public

Behavior:

a) Change the content of the receiver to the content of the resulting instance of the class `String` when the method `reverse` is invoked on the receiver.

b) Return the receiver.

15.2.10.5.31 String#rindex

`rindex(` *substring, offset*`=nil )`

Visibility: public

Behavior:

a) If *substring* is not an instance of the class `String`, the behavior is unspecified.

b) Let R and S be the contents of the receiver and *substring*, respectively.

c) If *offset* is given:

   1) If *offset* is not an instance of the class `Integer`, the behavior is unspecified.

   2) Let n be the value of *offset*.

   3) If n is larger than or equal to 0, let O be n.

   4) Otherwise, let O be $l + n$, where l is the length of S.

   5) If O is smaller than 0, return **nil**.

d) Otherwise, let O be 0.

e) If S appears as a substring of R at one or more positions whose index is smaller than or equal to O, return an instance of the class `Integer` whose value is the index of the last such position.

f) Otherwise, return **nil**.

15.2.10.5.32 String#scan

`scan(` *reg,* `&`*block* `)`

Visibility: public

Behavior:

a) If *reg* is not an instance of the class `Regexp`, the behavior is unspecified.

b) If *block* is not given, create an empty direct instance A of the class `Array`.

c) Let S be the content of the receiver, and let l be the length of S.

d) Let n be an integer 0.

e) Test if the pattern of *reg* matches S from the index n (see 15.2.15.4 and 15.2.15.5). Let M be the result attribute of the matching process.

f) If M is not **nil**:

1) Let L be the match result attribute of M.

2) If the length of L is 1, create a direct instance V of the class `String` whose content is the matched substring of M.

3) If the length of L is larger than 1:

i) Create an empty direct instance V of the class `Array`.

ii) Except for the first element, for each element e of L, in the same order in the list, append to V a new direct instance of the class `String` whose content is the substring of e.

4) If *block* is given, call *block* with V as the argument. Otherwise, append V to A.

5) Let *post* be the post-match of M. Let i be the index of the first character of *post* within S.

i) If i and n are the same, i.e. if *reg* matches the empty string, increment n by 1.

ii) Otherwise, let new n be i.

6) If $n < l$, continue processing from Step e).

g) If *block* is given, return the receiver. Otherwise, return A.

15.2.10.5.33 String#size

```
size
```

Visibility: public

Behavior: Same as the method `length` (see 15.2.10.5.26).

15.2.10.5.34 String#slice

```
slice( *args )
```

Visibility: public

Behavior: Same as the method `[]` (see 15.2.10.5.6).

15.2.10.5.35 String#split

`split(` *sep* `)`

Visibility: public

Behavior:

a) If *sep* is not an instance of the class `Regexp`, the behavior is unspecified.

b) Create an empty direct instance A of the class `Array`.

c) Let S be the content of the receiver, and let l be the length of S.

d) Let both *sp* and *bp* be 0, and let *was-empty* be false.

e) Test if the pattern of *sep* matches S from the index *sp* (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

f) If M is **nil**, append to A a new direct instance of the class `String` whose content is the substring of S beginning at the *sp*th character up to the last character of S.

g) Otherwise:

   1) If the matched substring of M is an empty string:

      i) If *was-empty* is true, append to A a new direct instance of the class `String` whose content is the *bp*th character of S.

      ii) Otherwise, increment *sp* by 1. If $sp < l$, let new *was-empty* be true and continue processing from Step e).

   2) Otherwise, let new *was-empty* be false. Let *pre* be the pre-match of M. Append to A a new direct instance of the class `String` whose content is the substring of *pre* beginning at the *bp*th character up to the last character of *pre*, unless *bp* is larger than the index of the last character of *pre*.

   3) Let L be the match result attribute of M.

   4) If the length of L is larger than 1, except for the first element, for each element e of L, in the same order in the list, take the following steps:

      i) Let c be the substring of e.

      ii) If c is not **nil**, append to A a new direct instance of the class `String` whose content is c.

   5) Let *post* be the post-match of M, and replace both *sp* and *bp* with the index of the first character of *post*.

   6) If $sp > l$, continue processing from Step e).

h) If the last element of A is an instance of the class `String` whose content is empty, remove the element. Repeat this step until this condition does not hold.

i) Return A.

15.2.10.5.36 String#sub

`sub(*`*args*`, &`*block*`)`

Visibility: public

Behavior:

a) If the length of *args* is 1 and *block* is given, or the length of *args* is 2:

   1) If the first element of *args* is not an instance of the class `Regexp`, the behavior is unspecified.

   2) Test if the pattern of the first element of *args* matches the content of the receiver (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

   3) If M is **nil**, create a direct instance of the class `String` whose content is the same as the receiver and return the instance.

   4) Otherwise:

      i) If the length of *args* is 1, call *block* with a new direct instance of the class `String` whose content is the matched substring of M as the argument. Let S be the resulting value of this call. If S is not an instance of the class `String`, the behavior is unspecified.

      ii) If the length of *args* is 2, let S be the last element of *args*. If S is not an instance of the class `String`, the behavior is unspecified.

      iii) Create a direct instance of the class `String` whose content is the concatenation of pre-match of M, the content of S, and post-match of M, and return the instance.

b) Otherwise, raise a direct instance of the class `ArgumentError`.

15.2.10.5.37 String#sub!

`sub!(*`*args*`, &`*block*`)`

Visibility: public

Behavior:

a) Let s be the content of the instance of the class `String` returned when the method `sub` is invoked on the receiver with the same arguments.

b) If the content of the receiver and s are the same, return **nil**. Otherwise, change the content of the receiver to s, and return the receiver.

15.2.10.5.38 String#to_i

---

to_i(*base* =10)

Visibility: public

Behavior:

a) If *base* is not an instance of the class `Integer` whose value is 2, 8, 10, nor 16, the behavior is unspecified. Otherwise, let b be the value of *base*.

b) If the receiver is empty, return an instance of the class `Integer` whose value is 0.

c) Let i be 0. Increment i by 1 while the ith character of the receiver is a *whitespace* character.

d) If the ith character of the receiver is "+" or "−", increment i by 1.

e) If the ith character of the receiver is "0", and any of the following conditions holds, increment i by 2:

Let c be the character of the receiver whose index is i plus 1.

- b is 2, and c is "b" or "B".
- b is 8, and c is "o" or "O".
- b is 10, and c is "d" or "D".
- b is 16, and c is "x" or "X".

f) Let s be a sequence of the following characters of the receiver from the ith index:

- If b is 2, *binary-digit* and "_".
- If b is 8, *octal-digit* and "_".
- If b is 10, *decimal-digit* and "_".
- If b is 16, *hexadecimal-digit* and "_".

g) If the length of s is 0, return an instance of the class `Integer` whose value is 0.

h) If s starts with "_", or s contains successive "_"s, the behavior is unspecified.

i) Let n be the value of s, ignoring interleaving "_"s, computed in base b.

If the "$-$" occurs in Step d), return an instance of the class `Integer` whose value is $-n$. Otherwise, return an instance of the class `Integer` whose value is n.

15.2.10.5.39 String#to_f

`to_f`

Visibility: public

Behavior:

a) If the receiver is empty, return a direct instance of the class `Float` whose value is 0.0.

b) If the receiver starts with a sequence of characters which is a *float-literal*, return a direct instance of the class `Float` whose value is the value of the *float-literal* (see 8.7.6.2).

c) If the receiver starts with a sequence of characters which is a *unprefixed-decimal-integer-literal*, return a direct instance of the class `Float` whose value is the value of the *unprefixed-decimal-integer-literal* as a floating-point number (see 8.7.6.2).

d) Otherwise, return a direct instance of the class `Float` whose value is implementation-defined.

15.2.10.5.40 String#to_s

`to_s`

Visibility: public

Behavior:

a) If the receiver is a direct instance of the class `String`, return the receiver.

b) Otherwise, create a new direct instance of the class `String` whose content is the same as the content of the receiver and return this instance.

15.2.10.5.41 String#to_sym

`to_sym`

Visibility: public

Behavior: Same as the method `intern` (see 15.2.10.5.25).

15.2.10.5.42 String#upcase

```
upcase
```

Visibility: public

Behavior: The method returns a new direct instance of the class `String` which contains all the characters of the receiver, with all the lower-case characters replaced with the corresponding upper-case characters.

15.2.10.5.43 String#upcase!

```
upcase!
```

Visibility: public

Behavior:

a) Let *s* be the content of the instance of the class `String` returned when the method `upcase` is invoked on the receiver.

b) If the content of the receiver and *s* are the same, return **nil**. Otherwise, change the content of the receiver to *s*, and return the receiver.

15.2.11 Symbol

15.2.11.1 General description

Instances of the class `Symbol` represent names (see 8.7.6.6). No two instances of the class `Symbol` shall represent the same name.

Instances of the class `Symbol` shall not be created by the method `new` of the class `Symbol`. Therefore, the singleton method `new` of the class `Symbol` shall be undefined, by invoking the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `Symbol` with a direct instance of the class `Symbol` whose name is "new" as the argument.

15.2.11.2 Direct superclass

The class `Object`

15.2.11.3 Instance methods

15.2.11.3.1 Symbol#===

```
===( other )
```

Visibility: public

Behavior: Same as the method `==` of the module `Kernel` (see 15.3.1.3.1).

15.2.11.3.2 Symbol#id2name

```
id2name
```

Visibility: public

Behavior: The method creates a direct instance of the class `String`, the content of which represents the name of the receiver, and returns this instance.

15.2.11.3.3 Symbol#to_s

```
to_s
```

Visibility: public

Behavior: Same as the method `id2name` (see 15.2.11.3.2).

15.2.11.3.4 Symbol#to_sym

```
to_sym
```

Visibility: public

Behavior: The method returns the receiver.

15.2.12 Array

15.2.12.1 General description

Instances of the class `Array` represent arrays, which are unbounded. An instance of the class `Array` which has no element is said to be ***empty***. The number of elements in an instance of the class `Array` is called its ***length***.

Instances of the class `Array` shall be empty when they are created by Step b) of the method `new` of the class `Class`.

Elements of an instance of the class `Array` have their indices counted up from 0.

Given an instance A of the class `Array`, operations ***append***, ***prepend***, and ***remove*** are defined as follows:

append: To append an object O to A is defined as follows:

Insert O after the last element of A.

Appending an object to A increases its length by 1.

prepend: To prepend an object O to A is defined as follows:

Insert O to the first index of A. Original elements of A are moved toward the end of A by one position.

Prepending an object to A increases its length by 1.

remove: To remove an element X from A is defined as follows:

a) Remove X from A.

b) If X is not the last element of A, move the elements after X toward the head of A by one position.

Removing an object to A decreases its length by 1.

15.2.12.2 Direct superclass

The class `Object`

15.2.12.3 Included modules

The following module is included in the class `Array`.

- `Enumerable`

15.2.12.4 Singleton methods

15.2.12.4.1 Array.[]

`Array.[]( *`*items*`)`

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` which contains the elements of *items*, preserving their order.

15.2.12.5 Instance methods

15.2.12.5.1 Array#*

`*(` *num* `)`

Visibility: public

Behavior:

a) If *num* is not an instance of the class `Integer`, the behavior is unspecified.

b) If the value of *num* is smaller than 0, raise a direct instance of the class `ArgumentError`.

c) If the value of *num* is 0, return an empty direct instance of the class `Array`.

d) Otherwise, create an empty direct instance A of the class `Array` and repeat the following for *num* times:

- Append all the elements of the receiver to A, preserving their order.

e) Return A.

15.2.12.5.2 Array#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If *other* is an instance of the class `Array`, let A be *other*. Otherwise, the behavior is unspecified.

b) Create an empty direct instance R of the class `Array`.

c) For each element of the receiver, in the indexing order, append the element to R. Then, for each element of A, in the indexing order, append the element to R.

d) Return R.

15.2.12.5.3 Array#<<

`<<(` *obj* `)`

Visibility: public

Behavior: The method appends *obj* to the receiver and return the receiver.

15.2.12.5.4 Array#[]

`[](` **args* `)`

Visibility: public

Behavior:

a) Let n be the length of the receiver.

b) If the length of *args* is 0, raise a direct instance of the class `ArgumentError`.

c) If the length of *args* is 1:

1) If the only argument is an instance of the class `Integer`, let k be the value of the only argument. Otherwise, the behavior is unspecified.

2) If $k < 0$, increment k by n. If k is still smaller than 0, return **nil**.

3) If $k \geq n$, return **nil**.

4) Otherwise, return the kth element of the receiver.

d) If the length of *args* is 2:

1) If the elements of *args* are instances of the class `Integer`, let b and l be the values of the first and the last element of *args*, respectively. Otherwise, the behavior is unspecified.

2) If $b < 0$, increment b by n. If b is still smaller than 0, return **nil**.

3) If $b > n$ or $l < 0$, return **nil**.

4) If $b = n$, create an empty direct instance of the class `Array` and return this instance.

5) If $l > n - b$, let new l be $n - b$.

6) Create an empty direct instance A of the class `Array`. Append the l elements of the receiver to A, from the bth index, preserving their order. Return A.

e) If the length of *args* is larger than 2, raise a direct instance of the class `ArgumentError`.

15.2.12.5.5 Array#[]=

`[]=(` `*`*args* `)`

Visibility: public

Behavior:

a) Let n be the length of the receiver.

b) If the length of *args* is smaller than 2, raise a direct instance of the class `ArgumentError`.

c) If the length of *args* is 2:

1) If the first element of *args* is an instance of the class `Integer`, let k be the value of the element and let V be the last element of *args*. Otherwise, the behavior is unspecified.

2) If $k < 0$, increment k by n. If k is still smaller than 0, raise a direct instance of the class `IndexError`.

3) If $k < n$, replace the kth element of the receiver with V.

4) Otherwise, expand the length of the receiver to $k + 1$. The last element of the receiver is V. If $k > n$, the elements whose index is from n to $k - 1$ is **nil**.

5) Return V.

d) If the length of *args* is 3, the behavior is unspecified.

e) If the length of *args* is larger than 3, raise a direct instance of the class `ArgumentError`.

15.2.12.5.6 Array#clear

`clear`

Visibility: public

Behavior: The method removes all the elements from the receiver and return the receiver.

15.2.12.5.7 Array#collect!

`collect!(&`*block*`)`

Visibility: public

Behavior:

a) If *block* is given:

1) For each element of the receiver in the indexing order, call *block* with the element as the only argument and replace the element with the resulting value.

2) Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.12.5.8 Array#concat

`concat(` *other*`)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Array`, the behavior is unspecified.

b) Otherwise, append all the elements of *other* to the receiver, preserving their order.

c) Return the receiver.

15.2.12.5.9 Array#delete_at

`delete_at(` *index* `)`

Visibility: public

Behavior:

a) If the *index* is not an instance of the class `Integer`, the behavior is unspecified.

b) Otherwise, let i be the value of the *index*.

c) Let n be the length of the receiver.

d) If i is smaller than 0, increment i by n. If i is still smaller than 0, return **nil**.

e) If i is larger than or equal to n, return **nil**.

f) Otherwise, remove the ith element of the receiver, and return the removed element.

15.2.12.5.10 Array#each

`each(` *&block* `)`

Visibility: public

Behavior:

a) If *block* is given:

   1) For each element of the receiver in the indexing order, call *block* with the element as the only argument.

   2) Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.12.5.11 Array#each_index

```
each_index(&block)
```

Visibility: public

Behavior:

a) If *block* is given:

1) For each element of the receiver in the indexing order, call *block* with an argument, which is an instance of the class `Integer` whose value is the index of the element.

2) Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.12.5.12 Array#empty?

```
empty?
```

Visibility: public

Behavior:

a) If the receiver is empty, return **true**.

b) Otherwise, return **false**.

15.2.12.5.13 Array#first

```
first(*args)
```

Visibility: public

Behavior:

a) If the length of *args* is 0:

1) If the receiver is empty, return **nil**.

2) Otherwise, return the first element of the receiver.

b) If the length of *args* is 1:

1) If the only argument is not an instance of the class `Integer`, the behavior is unspecified. Otherwise, let n be the value of the only argument.

2) If n is smaller than 0, raise a direct instance of the class `ArgumentError`.

3) Otherwise, let N be the smaller of n and the length of the receiver.

4) Return a newly created instance of the class `Array` which contains the first N elements of the receiver, preserving their order.

c) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.

15.2.12.5.14 Array#index

`index(` *object* `=nil )`

Visibility: public

Behavior:

a) If *object* is given:

1) For each element E of the receiver in the indexing order, take the following steps:

i) Invoke the method `==` on E with *object* as the argument.

ii) If the resulting value is a trueish object, return an instance of the class `Integer` whose value is the index of E.

2) If an instance of the class `Integer` is not returned in Step a) 1) ii), return **nil**.

b) Otherwise, the behavior is unspecified.

15.2.12.5.15 Array#initialize

`initialize(` *size* `=0,` *obj* `=nil, &`*block* `)`

Visibility: private

Behavior:

a) If *size* is not an instance of the class `Integer`, the behavior is unspecified. Otherwise, let n be the value of *size*.

b) If n is smaller than 0, raise a direct instance of the class `ArgumentError`.

c) Remove all the elements from the receiver.

d) If n is 0, return an implementation-defined value.

e) If n is larger than 0:

1) If *block* is given:

i) Let k be 0.

ii) Call *block* with an argument, which is an instance of the class `Integer` whose value is k. Append the resulting value of this call to the receiver.

iii) Increase k by 1. If k is equal to n, terminate this process. Otherwise, repeat from Step e) 1) ii).

2) Otherwise, append *obj* to the receiver n times.

3) Return an implementation-defined value.

15.2.12.5.16 Array#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class `Array`, the behavior is unspecified.

b) Remove all the elements from the receiver.

c) Append all the elements of *original* to the receiver, preserving their order.

d) Return an implementation-defined value.

15.2.12.5.17 Array#join

`join(` *sep* `=nil )`

Visibility: public

Behavior:

a) If *sep* is neither **nil** nor an instance of the class `String`, the behavior is unspecified.

b) Create an empty direct instance S of the class `String`.

c) For each element X of the receiver, in the indexing order:

1) If *sep* is not **nil**, and X is not the first element of the receiver, append the content of *sep* to S.

2) If X is an instance of the class `String`, append the content of X to S.

3) If X is an instance of the class `Array`:

i) If X is the receiver, i.e. if the receiver contains itself, append an implementation-defined sequence of characters to S.

ii) Otherwise, append to S the content of the instance of the class `String` returned by the invocation of the method `join` on X with *sep* as the argument.

4) Otherwise, the behavior is unspecified.

d) Return S.

15.2.12.5.18 Array#last

`last(*`*args*`)`

Visibility: public

Behavior:

a) If the length of *args* is 0:

1) If the receiver is empty, return **nil**.

2) Otherwise, return the last element of the receiver.

b) If the length of *args* is 1:

1) If the only argument is not an instance of the class `Integer`, the behavior is unspecified. Otherwise, let n be the value of the only argument.

2) If n is smaller than 0, raise a direct instance of the class `ArgumentError`.

3) Otherwise, let N be the smaller of n and the length of the receiver.

Return a newly created instance of the class `Array` which contains the last N elements of the receiver, preserving their order.

c) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.

15.2.12.5.19 Array#length

`length`

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the number of elements of the receiver.

15.2.12.5.20 Array#map!

`map!(` *&block* `)`

Visibility: public

Behavior: Same as the method `collect!` (see 15.2.12.5.7).

15.2.12.5.21 Array#pop

`pop`

Visibility: public

Behavior:

a) If the receiver is empty, return **nil**.

b) Otherwise, remove the last element from the receiver and return that element.

15.2.12.5.22 Array#push

`push(` **items* `)`

Visibility: public

Behavior:

a) For each element of *items*, in the indexing order, append it to the receiver.

b) Return the receiver.

15.2.12.5.23 Array#replace

`replace(` *other* `)`

Visibility: public

Behavior: Same as the method `initialize_copy` (see 15.2.12.5.16).

15.2.12.5.24 Array#reverse

`reverse`

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` which contains all the elements of the receiver in the reverse order.

15.2.12.5.25 Array#reverse!

`reverse!`

Visibility: public

Behavior: The method reverses the order of the elements of the receiver and return the receiver.

15.2.12.5.26 Array#rindex

`rindex(` *object*`=nil )`

Visibility: public

Behavior:

a) If *object* is given:

   1) For each element E of the receiver in the reverse indexing order, take the following steps:

      i) Invoke the method `==` on E with *object* as the argument.

      ii) If the resulting value is a trueish object, return an instance of the class `Integer` whose value is the index of E.

   2) If an instance of the class `Integer` is not returned in Step a) 1) ii), return **nil**.

b) Otherwise, the behavior is unspecified.

15.2.12.5.27 Array#shift

`shift`

Visibility: public

Behavior:

a) If the receiver is empty, return **nil**.

b) Otherwise, remove the first element from the receiver and return that element.

15.2.12.5.28 Array#size

---

size

Visibility: public

Behavior: Same as the method `length` (see 15.2.12.5.19).

15.2.12.5.29 Array#slice

---

slice(**args*)

Visibility: public

Behavior: Same as the method `[]` (see 15.2.12.5.4).

15.2.12.5.30 Array#unshift

---

unshift(**items*)

Visibility: public

Behavior:

a) For each element of *items*, in the reverse indexing order, prepend it to the receiver.

b) Return the receiver.

15.2.13 Hash

15.2.13.1 General description

Instances of the class `Hash` represent hashes, which are sets of key/value pairs.

An instance of the class `Hash` which has no key/value pair is said to be ***empty***. Instances of the class `Hash` shall be empty when they are created by Step b) of the method `new` of the class `Class`.

An instance of the class `Hash` cannot contain more than one key/value pair for each key.

An instance of the class `Hash` has the following attribute:

default value or proc: Either of the followings:

- A default value, which is returned by the method `[]` when the specified key is not found in the instance of the class `Hash`.

- A default proc, which is an instance of the class `Proc` and used to generate the return value of the method `[]` when the specified key is not found in the instance of the class `Hash`.

An instance of the class `Hash` shall not have both a default value and a default proc simultaneously.

Given two keys K_1 and K_2, the notation "$K_1 \equiv K_2$" means that the keys are equivalent, i.e. all of the following conditions hold:

- An invocation of the method `eql?` on K_1 with K_2 as the only argument evaluates to a trueish object.

- Let H_1 and H_2 be the results of invocations of the method `hash` on K_1 and K_2, respectively.

  H_1 and H_2 are the instances of the class `Integer` which represents the same integer.

  A conforming processor may define a certain range of integers, and when the values of H_1 or H_2 lies outside of this range, the processor may convert H_1 or H_2 to another instance of the class `Integer` whose value is within the range. Let I_1 and I_2 be each of the resulting instances respectively.

  The values of I_1 and I_2 are the same integer.

  If H_1 or H_2 is not an instance of the class `Integer`, whether $K_1 \equiv K_2$ is unspecified.

NOTE $K_1 \equiv K_2$ is not equivalent to $K_2 \equiv K_1$.

15.2.13.2 Direct superclass

The class `Object`

15.2.13.3 Included modules

The following module is included in the class `Hash`.

- `Enumerable`

15.2.13.4 Instance methods

15.2.13.4.1 Hash#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Hash`, the behavior is unspecified.

b) If all of the following conditions hold, return **true**:

- The receiver and *other* have the same number of key/value pairs.
- For each key/value pair P in the receiver, *other* has a corresponding key/value pair Q which satisfies the following conditions:
  - — The key of $P \equiv$ the key of Q.
  - — An invocation of the method `==` on the value of P with the value of Q as an argument results in a trueish object.

c) Otherwise, return **false**.

15.2.13.4.2 Hash#[]

`[]` (*key*)

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where *key* $\equiv$ the key of P, return the value of P.

b) Otherwise, invoke the method `default` on the receiver with *key* as the argument and return the resulting value.

15.2.13.4.3 Hash#[]=

`[]=` (*key*, *value*)

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where *key* $\equiv$ the key of P, replace the value of P with *value*.

b) Otherwise:

1) If *key* is a direct instance of the class `String`, create a copy of *key*, i.e. create a direct instance K of the class `String` whose content is the same as the *key*.

2) If *key* is not an instance of the class `String`, let K be *key*.

3) If *key* is an instance of a subclass of the class `String`, whether to create a copy or not is implementation-defined.

4) Store a pair of K and *value* into the receiver.

c) Return *value*.

15.2.13.4.4 Hash#clear

`clear`

Visibility: public

Behavior:

a) Remove all the key/value pairs from the receiver.

b) Return the receiver.

15.2.13.4.5 Hash#default

`default(` *`*args`* `)`

Visibility: public

Behavior:

a) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.

b) If the receiver has the default value, return the value.

c) If the receiver has the default proc:

1) If the length of *args* is 0, return **nil**.

2) If the length of *args* is 1, invoke the method `call` on the default proc of the receiver with two arguments, the receiver and the only element of *args*. Return the resulting value of this invocation.

d) Otherwise, return **nil**.

15.2.13.4.6 Hash#default=

`default=(` *value* `)`

Visibility: public

Behavior:

a) If the receiver has the default proc, remove the default proc.

b) Set the default value of the receiver to *value*.

c) Return *value*.

15.2.13.4.7 Hash#default_proc

`default_proc`

Visibility: public

Behavior:

a) If the receiver has the default proc, return the default proc.

b) Otherwise, return **nil**.

15.2.13.4.8 Hash#delete

`delete(` *key*, `&`*block* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where *key* $\equiv$ the key of P, remove P from the receiver and return the value of P.

b) Otherwise:

1) If *block* is given, call *block* with *key* as the argument. Return the resulting value of this call.

2) Otherwise, return **nil**.

15.2.13.4.9 Hash#each

`each(` `&`*block* `)`

Visibility: public

Behavior:

a) If *block* is given, for each key/value pair of the receiver in an implementation-defined order:

1) Create a direct instance of the class `Array` which contains two elements, the key and the value of the pair.

2) Call *block* with the instance as an argument.

Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.13.4.10 Hash#each_key

`each_key(&block)`

Visibility: public

Behavior:

a) If *block* is given, for each key/value pair of the receiver, in an implementation-defined order, call *block* with the key of the pair as the argument. Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.13.4.11 Hash#each_value

`each_value(&block)`

Visibility: public

Behavior:

a) If *block* is given, call *block* for each key/value pair of the receiver, with the value as the argument, in an implementation-defined order. Return the receiver.

b) If *block* is not given, the behavior is unspecified.

15.2.13.4.12 Hash#empty?

`empty?`

Visibility: public

Behavior:

a) If the receiver is empty, return **true**.

b) Otherwise, return **false**.

15.2.13.4.13 Hash#has_key?

`has_key?(` *key* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where *key* $\equiv$ the key of P, return **true**.

b) Otherwise, return **false**.

15.2.13.4.14 Hash#has_value?

`has_value?(` *value* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair whose value holds the following condition, return **true**.

- An invocation of the method `==` on the value with *value* as the argument result in a trueish object.

b) Otherwise, return **false**.

15.2.13.4.15 Hash#include?

`include?(` *key* `)`

Visibility: public

Behavior: Same as the method `has_key?` (see 15.2.13.4.13).

15.2.13.4.16 Hash#initialize

`initialize(` `*`*args*, `&`*block* `)`

Visibility: private

Behavior:

a) If *block* is given, and the length of *args* is not 0, raise a direct instance of the class `ArgumentError`.

b) If *block* is given and the length of *args* is 0, create a direct instance of the class `Proc` which represents *block* and set the default proc of the receiver to this instance.

c) If *block* is not given:

1) If the length of *args* is 0, let D be **nil**.

2) If the length of *args* is 1, let D be the only argument.

3) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.

4) Set the default value of the receiver to D.

d) Return an implementation-defined value.

15.2.13.4.17 Hash#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class `Hash`, the behavior is unspecified.

b) Remove all the key/value pairs from the receiver.

c) For each key/value pair P of *original*, in an implementation-defined order, add or update a key/value pair of the receiver by invoking the method `[]=` (see 15.2.13.4.3) on the receiver with the key of P and the value of P as the arguments.

d) Remove the default value or the default proc from the receiver.

e) If *orignal* has a default value, set the default value of the receiver to that value.

f) If *orignal* has a default proc, set the default proc of the receiver to that proc.

g) Return an implementation-defined value.

15.2.13.4.18 Hash#key?

`key?(` *key* `)`

Visibility: public

Behavior: Same as the method `has_key?` (see 15.2.13.4.13).

15.2.13.4.19 Hash#keys

```
keys
```

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` whose content is all the keys of the receiver. The order of the keys stored is implementation-defined.

15.2.13.4.20 Hash#length

```
length
```

Visibility: public

Behavior: The method returns an instance of the class `Integer` whose value is the number of key/value pairs stored in the receiver.

15.2.13.4.21 Hash#member?

`member?(` *key* `)`

Visibility: public

Behavior: Same as the method `has_key?` (see 15.2.13.4.13).

15.2.13.4.22 Hash#merge

`merge(` *other*, `&`*block* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Hash`, the behavior is unspecified.

b) Otherwise, create a direct instance H of the class `Hash` which has the same key/value pairs as the receiver.

c) For each key/value pair P of *other*, in an implementation-defined order:

   1) If *block* is given:

      i) If H has the key/value pair Q where the key of $P \equiv$ the key of Q, call *block* with three arguments, the key of P, the value of Q, and the value of P. Let V be the resulting value. Add or update a key/value pair of the receiver by invoking the method `[]=` (see 15.2.13.4.3) on H with the key of P and V as the arguments.

ii) Otherwise, add or update a key/value pair of the receiver by invoking the method `[]=` (see 15.2.13.4.3) on H with the key of P and the value of P as the arguments.

2) If *block* is not given, add or update a key/value pair of the receiver by invoking the method `[]=` (see 15.2.13.4.3) on H with the key of P and the value of P as the arguments.

d) Return H.

15.2.13.4.23 Hash#replace

`replace(` *other* `)`

Visibility: public

Behavior: Same as the method `initialize_copy` (see 15.2.13.4.17).

15.2.13.4.24 Hash#shift

`shift`

Visibility: public

Behavior:

a) If the receiver is empty:

1) If the receiver has the default proc, invoke the method `call` on the default proc with two arguments, the receiver and **nil**. Return the resulting value of this call.

2) If the receiver has the default value, return the value.

3) Otherwise, return **nil**.

b) Otherwise, choose a key/value pair P and remove P from the receiver. Return a newly created instance of the class `Array` which contains two elements, the key and the value of P.

Which pair is chosen is implementation-defined.

15.2.13.4.25 Hash#size

`size`

Visibility: public

Behavior: Same as the method `length` (see 15.2.13.4.20).

15.2.13.4.26 Hash#store

`store(` *key, value* `)`

Visibility: public

Behavior: Same as the method `[]=` (see 15.2.13.4.3).

15.2.13.4.27 Hash#value?

`value?(` *value* `)`

Visibility: public

Behavior: Same as the method `has_value?` (see 15.2.13.4.14).

15.2.13.4.28 Hash#values

`values`

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` which contains all the values of the receiver. The order of the values stored is implementation-defined.

15.2.14 Range

15.2.14.1 General description

Instances of the class `Range` represent ranges between two values, the start and end points.

An instance of the class `Range` has the following attributes:

start point: The value at the start of the range.

end point: The value at the end of the range.

exclusive flag: If this is true, the end point is excluded from the range. Otherwise, the end point is included in the range.

When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel` is invoked on an instance of the class `Range`, those attributes shall be copied from the receiver to the resulting value.

15.2.14.2 Direct superclass

The class `Object`

15.2.14.3 Included modules

The following module is included in the class `Range`.

- `Enumerable`

15.2.14.4 Instance methods

15.2.14.4.1 Range#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If all of the following conditions hold, return **true**:

- *other* is an instance of the class `Range`.
- Let S be the start point of *other*. Invocation of the method `==` on the start point of the receiver with S as the argument results in a trueish object.
- Let E be the end point of *other*. Invocation of the method `==` on the end point of the receiver with E as the argument results in a trueish object.
- The exclusive flags of the receiver and *other* are the same boolean value.

b) Otherwise, return **false**.

15.2.14.4.2 Range#===

`===(` *obj* `)`

Visibility: public

Behavior:

a) If neither the start point of the receiver nor the end point of the receiver is an instance of the class `Numeric`, the behavior is unspecified.

b) Invoke the method `<=>` on the start point of the receiver with *obj* as the argument. Let S be the result of this invocation.

1) If S is not an instance of the class `Integer`, the behavior is unspecified.

2) If the value of S is larger than 0, return **false**.

c) Invoke the method `<=>` on *obj* with the end point of the receiver as the argument. Let E be the result of this invocation.

- If E is not an instance of the class `Integer`, the behavior is unspecified.
- If the exclusive flag of the receiver is true, and the value of E is smaller than 0, return **true**.
- If the exclusive flag of the receiver is false, and the value of E is smaller than or equal to 0, return **true**.
- Otherwise, return **false**.

15.2.14.4.3 Range#begin

`begin`

Visibility: public

Behavior: The method returns the start point of the receiver.

15.2.14.4.4 Range#each

`each(`&*block*`)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) If an invocation of the method `respond_to?` on the start point of the receiver with a direct instance of the class `Symbol` whose name is `succ` as the argument results in a falseish object, raise a direct instance of the class `TypeError`.

c) Let V be the start point of the receiver.

d) Invoke the method `<=>` on V with the end point of the receiver as the argument. Let C be the resulting value.

1) If C is not an instance of the class `Integer`, the behavior is unspecified.

2) If the value of C is larger than 0, return the receiver.

3) If the value of C is 0:

i) If the exclusive flag of the receiver is true, return the receiver.

ii) If the exclusive flag of the receiver is false, call *block* with V as the argument, then, return the receiver.

e) Call *block* with V as the argument.

f) Invoke the method `succ` on V with no argument, and let new V be the resulting value.

g) Continue processing from Step d).

15.2.14.4.5 Range#end

```
end
```

Visibility: public

Behavior: The method returns the end point of the receiver.

15.2.14.4.6 Range#exclude_end?

```
exclude_end?
```

Visibility: public

Behavior: If the exclusive flag of the receiver is true, return **true**. Otherwise, return **false**.

15.2.14.4.7 Range#first

```
first
```

Visibility: public

Behavior: Same as the method `begin` (see 15.2.14.4.3).

15.2.14.4.8 Range#include?

`include?(` *obj* `)`

Visibility: public

Behavior: Same as the method `===` (see 15.2.14.4.2).

15.2.14.4.9 Range#initialize

`initialize(` *left, right, exclusive* `=false )`

Visibility: private

Behavior:

a) Invoke the method `<=>` on *left* with *right* as the argument. If an exception is raised and not handled during this invocation, raise a direct instance of the class `ArgumentError`. If the result of this invocation is not an instance of the class `Integer`, the behavior is unspecified.

b) If *exclusive* is a trueish object, let f be true. Otherwise, let f be false.

c) Set the start point, end point, and exclusive flag of the receiver to *left*, *right*, and f, respectively.

d) Return an implementation-defined value.

15.2.14.4.10 Range#last

`last`

Visibility: public

Behavior: Same as the method `end` (see 15.2.14.4.5).

15.2.14.4.11 Range#member?

`member?(` *obj* `)`

Visibility: public

Behavior: Same as the method `===` (see 15.2.14.4.2).

15.2.15 Regexp

15.2.15.1 General description

Instances of the class `Regexp` represent regular expressions, and have the following attributes.

pattern: A *pattern* of the regular expression (see 15.2.15.4). The default value of this attribute is empty.

If the value of this attribute is empty when a method is invoked on an instance of the class `Regexp`, except for the invocation of the method `initialize`, the behavior of the invoked method is unspecified.

ignorecase-flag: A boolean value which indicates whether a match is performed in the case insensitive manner. The default value of this attribute is false.

multiline-flag: A boolean value which indicates whether the pattern "`.`" matches a *line-terminator* (see 15.2.15.4). The default value of this attribute is false.

15.2.15.2 Direct superclass

The class `Object`

15.2.15.3 Constants

The following constants are defined in the class `Regexp`.

IGNORECASE: An instance of the class `Integer` whose value is 2^n, where the integer n is an implementation-defined value. The value of this constant shall be different from that of `MULTILINE` described below.

MULTILINE: An instance of the class `Integer` whose value is 2^m, where the integer m is an implementation-defined value.

The above constants are used to set the ignorecase-flag and multiline-flag attributes of an instance of the class `Regexp` (see 15.2.15.7.1).

15.2.15.4 Patterns

Syntax

---

pattern ::
  alternative
  | *pattern*$_1$ `|` *alternative*$_2$

alternative ::
  [empty]
  | *alternative*$_3$ *term*

term ::
  anchor
  | *atom*$_1$
  | *atom*$_2$ *quantifier*

anchor ::
  left-anchor | *right-anchor*

left-anchor ::
  `\A` | `^`

right-anchor ::
  `\z` | `$`

quantifier ::
  `*` | `+` | `?`

atom ::
    pattern-character
  | *grouping*
  | `.`
  | *atom-escape-sequence*

pattern-character ::
    source-character **but not** *regexp-meta-character*

regexp-meta-character ::
    `|` | `.` | `*` | `+` | `^` | `?` | `(` | `)` | `#` | `\` | `$`
  | *future-reserved-meta-character*

future-reserved-meta-character ::
    `[` | `]` | `{` | `}`

grouping ::
    `(` *pattern* `)`

atom-escape-sequence ::
    decimal-escape-sequence
  | *regexp-character-escape-sequence*

decimal-escape-sequence ::
    `\` *decimal-digit-except-zero*

regexp-character-escape-sequence ::
    regexp-escape-sequence
  | *regexp-non-escaped-sequence*
  | *hexadecimal-escape-sequence*
  | *regexp-octal-escape-sequence*
  | *regexp-control-escape-sequence*

regexp-escape-sequence ::
    `\` *regexp-escaped-character*

regexp-escaped-character ::
    `n` | `t` | `r` | `f` | `v` | `a` | `e`

regexp-non-escaped-sequence ::
    `\` *regexp-meta-character*

regexp-octal-escape-sequence ::
    octal-escape-sequence **but not** *decimal-escape-sequence*

regexp-control-escape-sequence ::
    `\` (`C-` | `c`) *regexp-control-escaped-character*

regexp-control-escaped-character ::
    regexp-character-escape-sequence
  | `?`
  | *source-character* **but not** (`\` | `?`)

---

*future-reserved-meta-character*s are reserved for the extension of the pattern of regular expressions.

Semantics

A regular expression selects specific substrings from a string called a target string according to the *pattern* of the regular expression. If the *pattern* matches more than one substring, the substring which begins earliest in the target string is selected. If there is more than one such substring beginning at that point, the substring that has the highest priority, which is described below, is selected. Each component of the *pattern* matches a substring of the target string as follows:

a) A *pattern* matches the following substring:

  1) If the *pattern* is an *alternative*$_1$, it matches the string matched with the *alternative*$_1$.

  2) If the *pattern* is a *pattern*$_1$ `|` *alternative*$_2$, it matches the string matched with either the *pattern*$_1$ or the *alternative*$_2$. The one matched with the *pattern*$_1$ has a higher priority.

  EXAMPLE 1 `"ab".slice(/(a|ab)/)` returns `"a"`, not `"ab"`.

b) An *alternative* matches the following substring:

  1) If the *alternative* is [empty], it matches an empty string.

  2) If the *alternative* is an *alternative*$_3$ *term*, the *alternative* matches the substring whose first part is matched with the *alternative*$_3$ and whose rest part is matched with the *term*.

  If there is more than one such substring, the priority of the substrings is determined as follows:

    i) If there is more than one candidate which is matched with the *alternative*$_3$, a substring whose first part is a candidate with a higher priority has a higher priority.

    EXAMPLE 2 `"abc".slice(/(a|ab)(c|b)/)` returns `"ab"`, not `"abc"`. In this case, `(a|ab)` is prior to `(c|b)`.

    ii) If the first parts of substrings are the same, and if there is more than one candidate which is matched with the *term*, a substring whose rest part is a candidate with a higher priority has a higher priority.

    EXAMPLE 3 `"abc".slice(/a(b|bc)/)` returns `"ab"`, not `"abc"`.

c) A *term* matches the following substring:

1) If the *term* is an $atom_1$, it matches the string matched with the $atom_1$.

2) If the *term* is an $atom_2$ *quantifier*, it matches a string as follows:

i) If the *quantifier* is `*`, it matches a sequence of zero or more strings matched with the $atom_2$.

ii) If the *quantifier* is `+`, it matches a sequence of one or more strings matched with $atom_2$.

iii) If the *quantifier* is `?`, it matches a sequence of zero or one string matched with the $atom_2$.

A longer sequence has a higher priority.

EXAMPLE 4 `"aaa".slice(/a*/)` returns `"aaa"`, none of `""`, `"a"`, and `"aa"`.

3) If the *term* is an *anchor*, it matches the empty string at a specific position within the target string *S*, as follows:

i) If the *anchor* is `\A`, it matches an empty string at the beginning of *S*.

ii) If the *anchor* is `^`, it matches an empty string at the beginning of *S* or just after a *line-terminator* which is followed by at least one character.

iii) If the *anchor* is `\z`, it matches an empty string at the end of *S*.

iv) If the *anchor* is `$`, it matches an empty string at the end of *S* or just before a *line-terminator*.

d) An *atom* matches the following substring:

1) If the *atom* is a *pattern-character*, it matches a character *C* represented by the *pattern-character*. If the *atom* is present in the pattern of an instance of the class `Regexp` whose ignorecase-flag attribute is true, it also matches a corresponding upper-case character of *C*, if *C* is a lower-case character, or a corresponding lower-case character of *C*, if *C* is an upper-case character.

2) If the *atom* is a *grouping*, it matches the string matched with the *grouping*.

3) If the *atom* is "`.`", it matches any character except for a *line-terminator*. If the *atom* is present in the pattern of an instance of the class `Regexp` whose multiline-flag attribute is true, it also matches a *line-terminator*.

4) If the *atom* is an *atom-escape-sequence*, it matches the string matched with the *atom-escape-sequence*.

e) A *grouping* matches the substring matched with the *pattern*.

f) An *atom-escape-sequence* matches the following substring:

1) If the *atom-escape-sequence* is a *decimal-escape-sequence*, it matches the string matched with the *decimal-escape-sequence*.

2) If the *atom-escape-sequence* is a *regexp-character-escape-sequence*, it matches a string of length one, the content of which is the character represented by the *regexp-character-escape-sequence*.

g) A *decimal-escape-sequence* matches the following substring:

1) Let i be an integer represented by *decimal-digit-except-zero*.

2) Let G be the ith *grouping* in the *pattern*, counted from 1, in the order of the occurrence of "(" of *grouping*s from the left of the *pattern*.

3) If the *decimal-escape-sequence* is present before G within the *pattern*, it does not match any string.

4) If G matches any string, the *decimal-escape-sequence* matches the same string.

5) Otherwise, the *decimal-escape-sequence* does not match any string.

h) A *regexp-character-escape-sequence* represents a character as follows:

- A *regexp-escape-sequence* represents a character as shown in 8.7.6.3.3, Table 1.
- A *regexp-non-escaped-sequence* represents a *regexp-meta-character*.
- A *hexadecimal-escape-sequence* represents a character as described in 8.7.6.3.3.
- A *regexp-octal-escape-sequence* is interpreted in the same way as an *octal-escape-sequence* (see 8.7.6.3.3).
- A *regexp-control-escape-sequence* represents a character, the code of which is computed by taking bitwise AND of 0x9f and the code of the character represented by the *regexp-control-escaped-character*, except when the *regexp-control-escaped-character* is `?`, in which case, the *regexp-control-escape-sequence* represents a character whose code is 127.

15.2.15.5 Matching process

A *pattern* P is considered to successfully match the given string S, if there exists a substring of S (including S itself) matched with P.

a) When an index is specified, it is tested if P matches the part of S which begins at the index and ends at the end of S. However, if the match succeeds, the string attribute of the resulting instance of the class `MatchData` is S, not the part of S which begins at the index, as described below.

b) A matching process returns either an instance of the class `MatchData` (see 15.2.16) if the match succeeds or **nil** if the match fails.

c) An instance of the class `MatchData` is created as follows:

1) Let B be the substring of S which P matched.

2) Create a direct instance of the class `MatchData`, and let M be the instance.

3) Set the string attribute of M (see 15.2.16.1) to S.

4) Create a new empty list L.

5) Let O be a pair of the substring B and the index of the first character of B within S. Append O to L.

6) For each *grouping* G in P, in the order of the occurrence of its "`(`" within P, take the following steps:

   i) If G matches a substring of B under the matching process of P, let B_G be the substring. Let O be a pair of the substring B_G and the index of the first character of B_G within S. Append O to L.

   ii) Otherwise, append to L a pair whose substring and index of the substring are **nil**.

7) Set the match result attribute of M to L.

8) M is the instance of the class `MatchData` returned by the matching process.

d) A matching process creates or updates a local variable binding with name "~", which is specifically used by the method `Regexp.last_match` (see 15.2.15.6.3), as follows:

1) Let M be the value which the matching process returns.

2) If the binding for the name "~" can be resolved by the process described in 9.2 as if "~" were a *local-variable-identifier*, replace the value of the binding with M.

3) Otherwise, create a local variable binding with name "~" and value M in the uppermost non-block element of ⟦local-variable-bindings⟧ where the non-block element means the element which does not correspond to a *block*.

e) A conforming processor may name the binding other than "~"; however, it shall not be of the form *local-variable-identifier*.

15.2.15.6 Singleton methods

15.2.15.6.1 Regexp.compile

`Regexp.compile(` *`*args`* `)`

Visibility: public

Behavior: Same as the method `new` (see 15.2.3.3.3).

15.2.15.6.2 Regexp.escape

```
Regexp.escape( string )
```

Visibility: public

Behavior:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Let *S* be the content of *string*.

c) Return a a new direct instance of the class `String` whose content is the same as *S*, except that every occurrences of characters on the left of Table 4 are replaced with the corresponding sequences of characters on the right of Table 4.

Table 4 – Regexp escaped characters

<table>
<tr><th>Characters replaced</th><th>Escaped sequence</th></tr>
<tr><td>0x0a</td><td>\n</td></tr>
<tr><td>0x09</td><td>\t</td></tr>
<tr><td>0x0d</td><td>\r</td></tr>
<tr><td>0x0c</td><td>\f</td></tr>
<tr><td>0x20</td><td>\ 0x20</td></tr>
<tr><td>#</td><td>\#</td></tr>
<tr><td>$</td><td>\$</td></tr>
<tr><td>(</td><td>\(</td></tr>
<tr><td>)</td><td>\)</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>+</td><td>\+</td></tr>
<tr><td>-</td><td>\-</td></tr>
<tr><td>.</td><td>\.</td></tr>
<tr><td>?</td><td>\?</td></tr>
<tr><td>[</td><td>\[</td></tr>
<tr><td>\</td><td>\\</td></tr>
<tr><td>]</td><td>\]</td></tr>
<tr><td>^</td><td>\^</td></tr>
<tr><td>{</td><td>\{</td></tr>
<tr><td>|</td><td>\|</td></tr>
<tr><td>}</td><td>\}</td></tr>
</table>

15.2.15.6.3 Regexp.last_match

```
Regexp.last_match( *index )
```

Visibility: public

Behavior:

a) Search for a binding of a local variable with name "~" as described in 9.2 as if "~" were a *local-variable-identifier*.

b) If the binding is found and its value is an instance of the class `MatchData`, let M be the instance. Otherwise, return **nil**.

c) If the length of *index* is 0, return M.

d) If the length of *index* is larger than 1, raise a direct instance of the class `ArgumentError`.

e) If the length of *index* is 1, let A be the only argument.

f) If A is not an instance of the class `Integer`, the behavior of the method is unspecified.

g) Let R be the result returned by invoking the method `[]` (see 15.2.16.3.1) on M with A as the only argument.

h) Return R.

15.2.15.6.4 Regexp.quote

```
Regexp.quote
```

Visibility: public

Behavior: Same as the method `escape` (see 15.2.15.6.2).

15.2.15.7 Instance methods

15.2.15.7.1 Regexp#initialize

`initialize(` *source, flag* `=nil )`

Visibility: private

Behavior:

a) If *source* is an instance of the class `Regexp`, let S be the pattern attribute of *source*. If *source* is an instance of the class `String`, let S be the content of *source*. Otherwise, the behavior is unspecified.

b) If S is not of the form *pattern* (see 15.2.15.4), raise a direct instance of the class `RegexpError`.

c) Set the pattern attribute of the receiver to S.

d) If *flag* is an instance of the class `Integer`, let n be the value of the instance.

1) If computing bitwise AND of the value of the constant `IGNORECASE` of the class `Regexp` and n results in non-zero value, set the ignorecase-flag attribute of the receiver to true.

2) If computing bitwise AND of the value of the constant `MULTILINE` of the class `Regexp` and n results in non-zero value, set the multiline-flag attribute of the receiver to true.

e) If *flag* is not an instance of the class `Integer`, and if *flag* is a trueish object, then set the ignorecase-flag attribute of the receiver to true.

f) Return an implementation-defined value.

15.2.15.7.2 Regexp#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class of the receiver, raise a direct instance of the class `TypeError`.

b) Set the pattern attribute of the receiver to the pattern attribute of *original*.

c) Set the ignorecase-flag attribute of the receiver to the ignorecase-flag attribute of *original*.

d) Set the multiline-flag attribute of the receiver to the multiline-flag attribute of *original*.

e) Return an implementation-defined value.

15.2.15.7.3 Regexp#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Regexp`, return **false**.

b) If the corresponding attributes of the receiver and *other* are the same, return **true**.

c) Otherwise, return **false**.

15.2.15.7.4 Regexp#===

`===( string )`

Visibility: public

Behavior:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Let S be the content of *string*.

c) Test if the pattern of the receiver matches S (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

d) If M is an instance of the class `MatchData`, return **true**.

e) Otherwise, return **false**.

15.2.15.7.5 Regexp#=~

`=~( string )`

Visibility: public

Behavior:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Let S be the content of *string*.

c) Test if the pattern of the receiver matches S (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

d) If M is **nil** return **nil**.

e) If M is an instance of the class `MatchData`, let P be first element of the match result attribute of M, and let i be the index of the substring of P.

f) Return an instance of the class `Integer` whose value is i.

15.2.15.7.6 Regexp#casefold?

`casefold?`

Visibility: public

Behavior: The method returns the value of the ignorecase-flag attribute of the receiver.

15.2.15.7.7 Regexp#match

`match(` *string* `)`

Visibility: public

Behavior:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Let S be the content of *string*.

c) Test if the pattern of the receiver matches S (see 15.2.15.4 and 15.2.15.5). Let M be the result of the matching process.

d) Return M.

15.2.15.7.8 Regexp#source

`source`

Visibility: public

Behavior: The method returns a direct instance of the class `String` whose content is the pattern of the receiver.

15.2.16 MatchData

15.2.16.1 General description

Instances of the class `MatchData` represent results of successful matches of instances of the class `Regexp` against instances of the class `String`.

An instance of the class `MatchData` has the attributes called ***string*** and ***match result***, which are initialized as described in 15.2.15.5. The string attribute is the target string S of a matching process. The match result attribute is a list whose element is a pair of a substring B matched by the *pattern* of an instance of the class `Regexp` or a *grouping* in the *pattern*, and the index I of the first character of B within S. B is called the substring of the element, and I is called the index of the substring of the element. Elements of the match result attribute are indexed by integers starting from 0.

Given an instance M of the class `MatchData`, three values named ***matched substring***, ***pre-match*** and ***post-match*** of M, respectively, are defined as follows:

Let S be the string attribute of M. Let F be the first element of the match result attribute of M. Let B and O be the substring of F and the index of the substring of F. Let i be the sum of O and the length of B.

matched substring: The matched substring of M is B.

pre-match: The pre-match of M is a part of S, from the first up to, but not including the Oth character of S.

post-match: The post-match of M is a part of S, from the ith up to the last character of S.

15.2.16.2 Direct superclass

The class `Object`

15.2.16.3 Instance methods

15.2.16.3.1 MatchData#[]

`[]( *`*args*` )`

Visibility: public

Behavior: Invoke the method `to_a` on the receiver (see 15.2.16.3.12), and invoke the method `[]` on the resulting instance of the class `Array` with *args* as the arguments (see 15.2.12.5.4), and then, return the resulting value of the invocation of the method `[]`.

15.2.16.3.2 MatchData#begin

`begin(` *index*`)`

Visibility: public

Behavior:

a) If *index* is not an instance of the class `Integer`, the behavior is unspecified.

b) Let L be the match result attribute of the receiver, and let i be the value of *index*.

c) If i is smaller than 0, or larger than or equal to the number of elements of L, raise a direct instance of the class `IndexError`.

d) Otherwise, return the second portion of the ith element of L.

15.2.16.3.3 MatchData#captures

`captures`

Visibility: public

Behavior:

a) Let L be the match result attribute of the receiver.

b) Create an empty direct instance A of the class `Array`.

c) Except for the first element, for each element e of L, in the same order in the list, append to A a direct instance of the class `String` whose content is the substring of e.

d) Return A.

15.2.16.3.4 MatchData#end

`end(` *index* `)`

Visibility: public

Behavior:

a) If *index* is not an instance of the class `Integer`, the behavior is unspecified.

b) Let L be the match result attribute of the receiver, and let i be the value of *index*.

c) If i is smaller than 0, or larger than or equal to the number of elements of L, raise a direct instance of the class `IndexError`.

d) Let F and S be the substring and the index of the substring of the ith element of L, respectively.

e) If F is **nil**, return **nil**.

f) Otherwise, let f be the length of F. Return an instance of the class `Integer` whose value is the sum of S and f.

15.2.16.3.5 MatchData#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class of the receiver, raise a direct instance of the class `TypeError`.

b) Set the string attribute of the receiver to the string attribute of *original*.

c) Set the match result attribute of the receiver to the match result attribute of *original*.

d) Return an implementation-defined value.

15.2.16.3.6 MatchData#length

```
length
```

Visibility: public

Behavior:

The method returns the number of elements of the match result attribute of the receiver.

15.2.16.3.7 MatchData#offset

`offset(` *index* `)`

Visibility: public

Behavior:

a) If *index* is not an instance of the class `Integer`, the behavior is unspecified.

b) Let L be the match result attribute of the receiver, and let i be the value of *index*.

c) If i is smaller than 0, or larger than or equal to the number of elements of L, raise a direct instance of the class `IndexError`.

d) Let S and b be the substring and the index of the substring of the ith element of L, respectively. Let e be the sum of b and the length of S.

e) Return a new instance of the class `Array` which contains two instances of the class `Integer`, the one whose value is b and the other whose value is e, in this order.

15.2.16.3.8 MatchData#post_match

```
post_match
```

Visibility: public

Behavior: The method returns an instance of the class `String` the content of which is the post-match of the receiver.

15.2.16.3.9 MatchData#pre_match

```
pre_match
```

Visibility: public

Behavior: The method returns an instance of the class `String` the content of which is the pre-match of the receiver.

15.2.16.3.10 MatchData#size

```
size
```

Visibility: public

Behavior: Same as the method `length` (see 15.2.16.3.6).

15.2.16.3.11 MatchData#string

```
string
```

Visibility: public

Behavior:

The method returns an instance of the class `String` the content of which is the string attribute of the receiver.

15.2.16.3.12 MatchData#to_a

```
to_a
```

Visibility: public

Behavior:

a) Let L be the match result attribute of the receiver.

b) Create an empty direct instance A of the class `Array`.

c) For each element e of L, in the same order in the list, append to A an instance of the class `String` whose content is the substring of e.

d) Return A.

15.2.16.3.13 MatchData#to_s

```
to_s
```

Visibility: public

Behavior: The method returns an instance of the class `String` the content of which is the matched substring of the receiver.

15.2.17 Proc

15.2.17.1 General description

Instances of the class `Proc` represent *block*s.

An instance of the class `Proc` has the following attribute.

block: The block represented by the instance.

15.2.17.2 Direct superclass

The class `Object`

15.2.17.3 Singleton methods

15.2.17.3.1 Proc.new

`Proc.new(` `&`*block*`)`

Visibility: public

Behavior:

a) If *block* is given, let *B* be *block*.

b) Otherwise:

   1) If the top of ⟦block⟧ is block-not-given, then raise a direct instance of the class `ArgumentError`.

   2) Otherwise, let *B* be the top of ⟦block⟧ .

c) Create a new direct instance of the class `Proc` which has *B* as its block attribute.

d) Return the instance.

15.2.17.4 Instance methods

15.2.17.4.1 Proc#[]

`[](` `*`*args*`)`

Visibility: public

Behavior: Same as the method `call` (see 15.2.17.4.3).

15.2.17.4.2 Proc#arity

`arity`

Visibility: public

Behavior: Let B be the block represented by the receiver.

a) If a *block-parameter* is omitted in B, return an instance of the class `Integer` whose value is implementation-defined.

b) If a *block-parameter* is present in B:

   1) If a *block-parameter-list* is omitted in the *block-parameter*, return an instance of the class `Integer` whose value is 0.

   2) If a *block-parameter-list* is present in the *block-parameter*:

      i) If the *block-parameter-list* is of the form *left-hand-side*, return an instance of the class `Integer` whose value is 1.

      ii) If the *block-parameter-list* is of the form *multiple-left-hand-side*:

         I) If the *multiple-left-hand-side* is of the form *grouped-left-hand-side*, return an instance of the class `Integer` whose value is implementation-defined.

         II) If the *multiple-left-hand-side* is of the form *packing-left-hand-side*, return an instance of the class `Integer` whose value is -1.

         III) Otherwise, let n be the number of *multiple-left-hand-side-items* of the *multiple-left-hand-side*.

         IV) If the *multiple-left-hand-side* ends with a *packing-left-hand-side*, return an instance of the class `Integer` whose value is $-(n+1)$.

         V) Otherwise, return an instance of the class `Integer` whose value is n.

15.2.17.4.3 Proc#call

`call(*`*args*`)`

Visibility: public

Behavior: Let B be the block attribute of the receiver. Let L be an empty list.

a) Append each element of *args*, in the indexing order, to L.

b) Call B with L as the arguments (see 11.3.3). Let V be the result of the call.

c) Return V.

15.2.17.4.4 Proc#clone

```
clone
```

Visibility: public

Behavior:

a) Create a direct instance of the class of the receiver which has no bindings of instance variables. Let O be the newly created instance.

b) For each binding B of the instance variables of the receiver, create a variable binding with the same name and value as B in the set of bindings of instance variables of O.

c) If the receiver is associated with a singleton class, let E_o be the singleton class, and take the following steps:

   1) Create a singleton class whose direct superclass is the direct superclass of E_o. Let E_n be the singleton class.

   2) For each binding B_{v1} of the constants of E_o, create a variable binding with the same name and value as B_{v1} in the set of bindings of constants of E_n.

   3) For each binding B_{v2} of the class variables of E_o, create a variable binding with the same name and value as B_{v2} in the set of bindings of class variables of E_n.

   4) For each binding B_m of the instance methods of E_o, create a method binding with the same name and value as B_m in the set of bindings of instance methods of E_n.

   5) Associate O with E_n.

d) Set the block attribute of O to the block attribute of the receiver.

e) Return O.

15.2.17.4.5 Proc#dup

```
dup
```

Visibility: public

Behavior:

a) Create a direct instance of the class of the receiver which has no bindings of instance variables. Let O be the newly created instance.

b) Set the block attribute of O to the block attribute of the receiver.

c) Return O.

15.2.18 Struct

15.2.18.1 General description

The class `Struct` is a generator of a structure type which is a class defining a set of fields and methods for accessing these fields. Fields are indexed by integers starting from 0 (see 15.2.18.3.1). An instance of a generated class has values for the set of fields. Those values can be referred to and updated with accessor methods for their fields.

15.2.18.2 Direct superclass

The class `Object`

15.2.18.3 Singleton methods

15.2.18.3.1 Struct.new

`Struct.new(` *string, *symbol_list* `)`

Visibility: public

Behavior: The method creates a class defining a set of fields and accessor methods for these fields.

When the method is invoked, take the following steps:

a) Create a direct instance of the class `Class` which has the class `Struct` as its direct superclass. Let C be that class.

b) If *string* is not an instance of the class `String` or the class `Symbol`, the behavior is unspecified.

c) If *string* is an instance of the class `String`, let N be the content of the instance.

   1) If N is not of the form *constant-identifier*, raise a direct instance of the class `ArgumentError`.

   2) Otherwise,

      i) If the binding with name N exists in the set of bindings of constants in the class `Struct`, replace the value of the binding with C.

      ii) Otherwise, create a constant binding in the class `Struct` with name N and value C.

d) If *string* is an instance of the class `Symbol`, prepend the instance to *symbol_list*.

e) Let i be 0.

f) For each element S of *symbol_list*, take the following steps:

1) Let N be the name designated by S.

2) Define a field, which is named N and is indexed by i, in C.

3) If N is of the form *local-variable-identifier* or *constant-identifier*:

i) Define a method named N in C which takes no arguments, and when invoked, returns the value of the field named N.

ii) Define a method named N= (i.e. N postfixed by "`=`") in C which takes one argument, and when invoked, sets the field named N to the given argument and returns the argument.

4) Increment i by 1.

g) Return C.

Classes created by the method `Struct.new` are equipped with public singleton methods `new`, `[]`, and `members`. The following describes those methods, assuming that the name of a class created by the method `Struct.new` is C.

C`.new(`**args*`)`

Visibility: public

Behavior:

a) Create a direct instance of the class with the set of fields the receiver defines. Let I be the instance.

b) Invoke the method `initialize` on I with *args* as the list of arguments.

c) Return I.

C`.[](`**args*`)`

Visibility: public

Behavior: Same as the method `new` described above.

C`.members`

Visibility: public

Behavior:

a) Create a direct instance *A* of the class `Array`. For each field of the receiver, in the indexing order of the fields, create a direct instance of the class `String` whose content is the name of the field and append the instance to *A*.

b) Return *A*.

15.2.18.4 Instance methods

15.2.18.4.1 Struct#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If *other* and the receiver are the same object, return **true**.

b) If the class of *other* and that of the receiver are different, return **false**.

c) Otherwise, for each field named *f* of the receiver, take the following steps:

  1) Let *R* and *O* be the values of the fields named *f* of the receiver and *other* respectively.

  2) If *R* and *O* are not the same object,

    i) Invoke the method `==` on *R* with *O* as the only argument. Let *V* be the resulting value of the invocation.

    ii) If *V* is a falseish object, return **false**.

d) Return **true**.

15.2.18.4.2 Struct#[]

`[](` *name* `)`

Visibility: public

Behavior:

a) If *name* is an instance of the class `Symbol` or the class `String`:

  1) Let *N* be the name designated by *name*.

  2) If the receiver has the field named *N*, return the value of the field.

3) Otherwise, let S be an instance of the class `Symbol` with name N and raise a direct instance of the class `NameError` which has S as its name attribute.

b) If *name* is an instance of the class `Integer`, let i be the value of *name*. Let n be the number of the fields of the receiver.

1) If i is negative, let new i be $n + i$.

2) If i is still negative or i is larger than or equal to n, raise a direct instance of the class `IndexError`.

3) Otherwise, return the value of the field whose index is i.

c) Otherwise, the behavior of the method is unspecified.

15.2.18.4.3 Struct#[]=

`[] =(` *name, obj* `)`

Visibility: public

Behavior:

a) If *name* is an instance of the class `Symbol` or an instance of the class `String`:

1) Let N be the name designated by *name*.

2) If the receiver has the field named N,

i) Replace the value of the field with *obj*,

ii) Return *obj*.

3) Otherwise, let S be an instance of the class `Symbol` with name N and raise a direct instance of the class `NameError` which has S as its name attribute.

b) If *name* is an instance of the class `Integer`, let i be the value of *name*. Let n be the number of the fields of the receiver.

1) If i is negative, let new i be $n + i$.

2) If i is still negative or i is larger than or equal to n, raise a direct instance of the class `IndexError`.

3) Otherwise,

i) Replace the value of the field whose index is i with *obj*

ii) Return *obj*.

c) Otherwise, the behavior of the method is unspecified.

15.2.18.4.4 Struct#each

`each(&block)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) For each field of the receiver, in the indexing order, call *block* with the value of the field as the only argument.

c) Return the receiver.

15.2.18.4.5 Struct#each_pair

`each_pair(&block)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) For each field of the receiver, in the indexing order, take the following steps:

   1) Let N and V be the name and the value of the field respectively. Let S be an instance of the class `Symbol` with name N.

   2) Call *block* with the list of arguments which contains S and V in this order.

c) Return the receiver.

15.2.18.4.6 Struct#members

`members`

Visibility: public

Behavior: Same as the method `members` described in 15.2.18.3.1.

15.2.18.4.7 Struct#select

`select(&block)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Create an empty direct instance of the class `Array`. Let A be the instance.

c) For each field of the receiver, in the indexing order, take the following steps:

1) Let V be the value of the field.

2) Call *block* with V as the only argument. Let R be the resulting value of the call.

3) If R is a trueish object, append V to A.

d) Return A.

15.2.18.4.8 Struct#initialize

`initialize(*args)`

Visibility: private

Behavior: Let N_a be the length of *args*, and let N_f be the number of the fields of the receiver.

a) If N_a is larger than N_f, raise a direct instance of the class `ArgumentError`.

b) Otherwise, for each field f of the receiver, let i be the index of f, and set the value of f to the ith element of *args*, or to **nil** when i is equal to or larger than N_a.

c) Return an implementation-defined value.

15.2.18.4.9 Struct#initialize_copy

`initialize_copy(original)`

Visibility: private

Behavior:

a) If the receiver and *original* are the same object, return an implementation-defined value.

b) If *original* is not an instance of the class of the receiver, raise a direct instance of the class `TypeError`.

c) If the number of the fields of the receiver and the number of the fields of *original* are different, raise a direct instance of the class `TypeError`.

d) For each field f of *original*, let i be the index of f, and set the value of the ith field of the receiver to the value of f.

e) Return an implementation-defined value.

15.2.19 Time

15.2.19.1 General description

Instances of the class `Time` represent dates and times.

An instance of the class `Time` holds the following attributes.

Microseconds: Microseconds since January 1, 1970 00:00 UTC. Microseconds is an integer whose range is implementation-defined. The value of microseconds attributes is rounded to fit in the representation of microseconds in an implementation-defined way. If an out of range value is given as microseconds when creating an instance of the class `Time`, a direct instance of either of the class `ArgumentError` or the class `RangeError` shall be raised. Which class is chosen is implementation-defined.

Time zone: The time zone.

15.2.19.2 Direct superclass

The class `Object`

15.2.19.3 Time computation

Mathematical functions introduced in this subclause are used throughout the descriptions in 15.2.19. These functions are assumed to compute exact mathematical results using mathematical real numbers.

Leap seconds are ignored in this document. However, a conforming processor may support leap seconds in an implementation-defined way.

15.2.19.3.1 Day

The number of microseconds of a day is computed as follows:

$$MicroSecPerDay = 24 \times 60 \times 60 \times 10^6$$

The number of days since January 1, 1970 00:00 UTC which corresponds to microseconds t is computed as follows:

$$Day(t) = floor\left(\frac{t}{MicroSecPerDay}\right)$$

$$floor(t) = \text{The integer } x \text{ such that } x \leq t < x + 1$$

The weekday which corresponds to microseconds t is computed as follows:

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

15.2.19.3.2 Year

A year has 365 days, except for leap years, which have 366 days. Leap years are those which are either:

- divisible by 4 and not divisible by 100, or
- divisible by 400.

The number of days from January 1, 1970 00:00 UTC to the beginning of a year y is computed as follows:

$$DayFromYear(y) = 365 \times (y - 1970) + floor\left(\frac{y - 1969}{4}\right) - floor\left(\frac{y - 1901}{100}\right) + floor\left(\frac{y - 1601}{400}\right)$$

Microseconds elapsed since January 1, 1970 00:00 UTC until the beginning of y is computed as follows:

$$MicroSecFromYear(y) = DayFromYear(y) \times MicroSecPerDay$$

The year number y which corresponds to microseconds t measured from January 1, 1970 00:00 UTC is computed as follows.

$$YearFromTime(t) = y \text{ such that,} \quad MicroSecFromYear(y - 1) < t \leq MicroSecFromYear(y)$$

The number of days from the beginning of the year for the given microseconds t is computed as follows.

$$DayWithinYear(t) = Day(t) - DayFromYear(YearFromTime(t))$$

15.2.19.3.3 Month

Months have usual number of days. Leap years have the extra day in February. Each month is identified by the number in the range 1 to 12, in the order from January to December.

The month number which corresponds to microseconds t measured from January 1, 1970 00:00 UTC is computed as follows.

$$MonthFromTime(t) = \begin{cases} 1 & \text{if } 0 \le DayWithinYear(t) < 31 \\ 2 & \text{if } 31 \le DayWithinYear(t) < 59 + LeapYear(t) \\ 3 & \text{if } 59 + LeapYear(t) \le DayWithinYear(t) < 90 + LeapYear(t) \\ 4 & \text{if } 90 + LeapYear(t) \le DayWithinYear(t) < 120 + LeapYear(t) \\ 5 & \text{if } 120 + LeapYear(t) \le DayWithinYear(t) < 151 + LeapYear(t) \\ 6 & \text{if } 151 + LeapYear(t) \le DayWithinYear(t) < 181 + LeapYear(t) \\ 7 & \text{if } 181 + LeapYear(t) \le DayWithinYear(t) < 212 + LeapYear(t) \\ 8 & \text{if } 212 + LeapYear(t) \le DayWithinYear(t) < 243 + LeapYear(t) \\ 9 & \text{if } 243 + LeapYear(t) \le DayWithinYear(t) < 273 + LeapYear(t) \\ 10 & \text{if } 273 + LeapYear(t) \le DayWithinYear(t) < 304 + LeapYear(t) \\ 11 & \text{if } 304 + LeapYear(t) \le DayWithinYear(t) < 334 + LeapYear(t) \\ 12 & \text{if } 334 + LeapYear(t) \le DayWithinYear(t) < 365 + LeapYear(t) \end{cases}$$

$$LeapYear(t) = \begin{cases} 1 & \text{if } YearFromTime(t) \text{ is a leap year} \\ 0 & \text{otherwise} \end{cases}$$

15.2.19.3.4 Days of month

The day of the month which corresponds to microseconds t measured from January 1, 1970 00:00 UTC is computed as follows.

$$DayWithinMonth(t) = \begin{cases} DayWithinYear(t) + 1 & \text{if } MonthFromTime(t) = 1 \\ DayWithinYear(t) - 30 & \text{if } MonthFromTime(t) = 2 \\ DayWithinYear(t) - 58 - LeapYear(t) & \text{if } MonthFromTime(t) = 3 \\ DayWithinYear(t) - 89 - LeapYear(t) & \text{if } MonthFromTime(t) = 4 \\ DayWithinYear(t) - 119 - LeapYear(t) & \text{if } MonthFromTime(t) = 5 \\ DayWithinYear(t) - 150 - LeapYear(t) & \text{if } MonthFromTime(t) = 6 \\ DayWithinYear(t) - 180 - LeapYear(t) & \text{if } MonthFromTime(t) = 7 \\ DayWithinYear(t) - 211 - LeapYear(t) & \text{if } MonthFromTime(t) = 8 \\ DayWithinYear(t) - 242 - LeapYear(t) & \text{if } MonthFromTime(t) = 9 \\ DayWithinYear(t) - 272 - LeapYear(t) & \text{if } MonthFromTime(t) = 10 \\ DayWithinYear(t) - 303 - LeapYear(t) & \text{if } MonthFromTime(t) = 11 \\ DayWithinYear(t) - 333 - LeapYear(t) & \text{if } MonthFromTime(t) = 12 \end{cases}$$

15.2.19.3.5 Hours, Minutes, and Seconds

The numbers of microseconds in an hour, a minute, and a second are as follows:

$$
\begin{aligned}
MicroSecPerHour &= 60 \times 60 \times 10^6 \\
MicroSecPerMinute &= 60 \times 10^6 \\
MicroSecPerSecond &= 10^6
\end{aligned}
$$

The hour, the minute, and the second which correspond to microseconds t measured from January 1, 1970 00:00 UTC are computed as follows.

$$
\begin{aligned}
HourFromTime(t) &= floor\left(\frac{t}{MicroSecPerHour}\right) \quad \text{modulo} \quad 24 \\
MinuteFromTime(t) &= floor\left(\frac{t}{MicroSecPerMinute}\right) \quad \text{modulo} \quad 60 \\
SecondFromTime(t) &= floor\left(\frac{t}{MicroSecPerSecond}\right) \quad \text{modulo} \quad 60
\end{aligned}
$$

15.2.19.4 Time zone and Local time

The current time zone is determined from time zone information provided by the underlying system. If the system does not provide information on the current local time zone, the time zone attribute of an instance of the class `Time` is implementation-defined.

The local time for an instance of the class `Time` is computed from its microseconds t and time zone z as follows.

$$
\begin{aligned}
LocalTime &= t + ZoneOffset(z) \\
ZoneOffset(z) &= \text{UTC offset of } z \text{ measured in microseconds}
\end{aligned}
$$

15.2.19.5 Daylight saving time

On system where it is possible to determine the daylight saving time for each time zone, a conforming processor should adjust the microseconds attributes of an instance of the class `Time` if that microseconds falls within the daylight saving time of the time zone attributes of the instance. An algorithm used for the adjustment is implementation-defined.

15.2.19.6 Singleton methods

15.2.19.6.1 Time.at

`Time.at(*args)`

Visibility: public

Behavior:

a) If the length of *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.

b) If the length of *args* is 1, let A be the only argument.

  1) If A is an instance of the class `Time`, return a new instance of the class `Time` which represents the same time and has the same time zone as A.

  2) If A is an instance of the class `Integer` or an instance of the class `Float`:

    i) Let N be the value of A.

    ii) Create a direct instance of the class `Time` which represents the time at $N \times 10^6$ microseconds since January 1, 1970 00:00 UTC, with the current local time zone.

    iii) Return the resulting instance.

  3) Otherwise, the behavior is unspecified.

c) If the length of *args* is 2, let S and M be the first and second element of *args*.

  1) If S is an instance of the class `Integer`, let N_S be the value of S.

  2) Otherwise, the behavior is unspecified.

  3) If M is an instance of the class `Integer` or an instance of the class `Float`, let N_M be the value of M.

  4) Otherwise, the behavior is unspecified.

  5) Create a direct instance of the class `Time` which represents the time at $N_S \times 10^6 + N_M$ microseconds since January 1, 1970 00:00 UTC, with the current local time zone.

  6) Return the resulting instance.

15.2.19.6.2 Time.gm

`Time.gm(` *year*, *month*`=1`, *day*`=1`, *hour*`=0`, *min*`=0`, *sec*`=0`, *usec*`=0` `)`

Visibility: public

Behavior:

a) Compute an integer value for *year*, *day*, *hour*, *min*, *sec*, and *usec* as described below. Let Y, D, H, Min, S, and U be integers thus converted.

An integer I is determined from the given object O as follows:

1) If O is an instance of the class `Integer`, let I be the value of O.

2) If O is an instance of the class `Float`, let I be the integral part of the value of O.

3) If O is an instance of the class `String`:

i) If the content of O is a sequence of *decimal-digit*s, let I be the value of those sequence of digits computed using base 10.

ii) Otherwise, the behavior is unspecified.

4) Otherwise, the behavior is unspecified.

b) Compute an integer value from *month* as follows:

1) If *month* is an instance of the class `Integer`, let Mon be the value of *month*.

2) If *month* is an instance of the class `String`:

i) If the content of *month* is the same as one of the names of the months in the lower row on Table 5, ignoring the differences in case, let Mon be the integer which corresponds to *month* in the upper row on the same table.

ii) If the first character of *month* is *decimal-digit*, compute an integer value from *month* as in Step a). Let Mon be the resulting integer.

iii) Otherwise, raise a direct instance of the class `ArgumentError`.

3) Otherwise, the behavior is unspecified.

c) If Y is an integer such that $0 \leq Y \leq 138$, the behavior is implementation-defined.

d) If each integer computed above is outside the range as listed below, raise a direct instance of the class `ArgumentError`.

- $1 \leq Mon \leq 12$
- $1 \leq D \leq 31$
- $0 \leq H \leq 23$
- $0 \leq Min \leq 59$
- $0 \leq S \leq 60$

Whether any conditions are placed on Y is implementation-defined.

e) Let t be an integer which satisfies all of the following equations.

- $YearFromTime(t) = Y$
- $MonthFromTime(t) = Mon$
- $DayWithinMonth(t) = 1$

f) Compute microseconds T as follows.

$$T = t + D \times MicroSecPerDay + H \times MicroSecPerHour + Min \times MicroSecPerMinute + S \times 10^6 + U$$

g) Create a direct instance of the class `Time` which represents the time at T since January 1, 1970 00:00 UTC, with the UTC time zone.

h) Return the resulting instance.

Table 5 – The names of months and corresponding integer

1	2	3	4	5	6	7	8	9	10	11	12
Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec

15.2.19.6.3 Time.local

`Time.local(` *year*, *month*`=1,` *day*`=1,` *hour*`=0,` *min*`=0,` *sec*`=0,` *usec*`=0 )`

Visibility: public

Behavior: Same as the method `Time.gm` (see 15.2.19.6.2), except that the method returns a direct instance of the class `Time` which has the current local time zone as its time zone.

15.2.19.6.4 Time.mktime

`Time.mktime(` *year*, *month*`=1,` *day*`=1,` *hour*`=0,` *min*`=0,` *sec*`=0,` *usec*`=0 )`

Visibility: public

Behavior: Same as the method `Time.local` (see 15.2.19.6.3).

15.2.19.6.5 Time.now

```
Time.now
```

Visibility: public

Behavior: This method returns a direct instance of the class `Time` which represents the current time with the current local time zone.

The behavior of this method is the same as the method `new` (see 15.2.3.3.3).

15.2.19.6.6 Time.utc

`Time.utc(` *year*, *month*`=1`, *day*`=1`, *hour*`=0`, *min*`=0`, *sec*`=0`, *usec*`=0` `)`

Visibility: public

Behavior: Same as the method `Time.gm` (see 15.2.19.6.2).

15.2.19.7 Instance methods

15.2.19.7.1 Time#+

`+(` *offset* `)`

Visibility: public

Behavior:

a) If *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is unspecified.

b) Let V be the value of *offset*.

c) Let o be the result of computing $V \times 10^6$.

d) Let t and z be the microseconds and time zone attribute of the receiver.

e) Create a direct instance of the class `Time` which represents the time at $(t+o)$ microseconds since January 1, 1970 00:00 UTC, with z as its time zone.

f) Return the resulting instance.

15.2.19.7.2 Time#−

`-(` *offset* `)`

Visibility: public

Behavior:

a) If *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is unspecified.

b) Let V be the value of *offset*.

c) Let o be the result of computing $V \times 10^6$.

d) Let t and z be the microseconds and time zone attribute of the receiver.

e) Create a direct instance of the class `Time` which represents the time at $t-o$ microseconds since January 1, 1970 00:00 UTC, with z as its time zone.

f) Return the resulting instance.

15.2.19.7.3 Time#<=>

`<=>( other )`

Visibility: public

Behavior:

a) If *other* is not an instance of the class `Time`, return **nil**.

b) Otherwise, let T_r and T_o be microseconds attributes of the receiver and *other*, respectively.

  1) If $T_r > T_o$, return an instance of the class `Integer` whose value is 1.

  2) If $T_r = T_o$, return an instance of the class `Integer` whose value is 0.

  3) If $T_r < T_o$, return an instance of the class `Integer` whose value is -1.

15.2.19.7.4 Time#asctime

`asctime`

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Let W be the name of the day of the week in the second row on Table 6 which corresponds to $WeekDay(t)$ in the upper row on the same table.

c) Let *Mon* be the name of the month in the second row on Table 5 which corresponds to $MonthFromTime(t)$ in the upper row on the same table.

d) Let D, H, M, S, and Y be as follows:

$$
\begin{aligned}
D &= DayWithinMonth(t)\\
H &= HourFromTime(t)\\
M &= MinuteFromTime(t)\\
S &= SecondFromTime(t)\\
Y &= YearFromTime(t)
\end{aligned}
$$

e) Create a direct instance of the class `String`, the content of which is the following sequence of characters:

$W\ Mon\ D\ H{:}M{:}S\ Y$

D is formatted as two digits with a leading space character (0x20) as necessary. H, M, and S are formatted as two digits with a leading zero as necessary.

EXAMPLE `Time.local(2001, 10, 1, 13, 20, 5).asctime` returns `"Mon Oct  1 13:20:05 2001"`.

f) Return the resulting instance.

Table 6 – The names of the days of the week corresponding to integers

0	1	2	3	4	5	6
Sun	Mon	Tue	Wed	Thu	Fry	Sat

15.2.19.7.5 Time#ctime

```
ctime
```

Visibility: public

Behavior: Same as the method `asctime` (see 15.2.19.7.4).

15.2.19.7.6 Time#day

```
day
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $DayWithinMonth(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.7 Time#dst?

```
dst?
```

Visibility: public

Behavior: Let T and Z be the microseconds and time zone attribute of the receiver.

a) If T falls within the daylight saving time of Z, return **true**.

b) Otherwise, return **false**.

15.2.19.7.8 Time#getgm

```
getgm
```

Visibility: public

Behavior: Same as the method `getutc` (see 15.2.19.7.10).

15.2.19.7.9 Time#getlocal

```
getlocal
```

Visibility: public

Behavior: The method returns a new instance of the class `Time` which has the same microseconds as the receiver, but has current local time zone as its time zone.

15.2.19.7.10 Time#getutc

```
getutc
```

Visibility: public

Behavior: The method returns a new instance of the class `Time` which has the same microseconds as the receiver, but has UTC as its time zone.

15.2.19.7.11 Time#gmt?

```
gmt?
```

Visibility: public

Behavior: Same as the method `utc?` (see 15.2.19.7.28).

15.2.19.7.12 Time#gmt_offset

```
gmt_offset
```

Visibility: public

Behavior: Same as the method `utc_offset` (see 15.2.19.7.29).

15.2.19.7.13 Time#gmtime

```
gmtime
```

Visibility: public

Behavior: Same as the method `utc` (see 15.2.19.7.27).

15.2.19.7.14 Time#gmtoff

```
gmtoff
```

Visibility: public

Behavior: Same as the method `utc_offset` (see 15.2.19.7.29).

15.2.19.7.15 Time#hour

```
hour
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $HourFromTime(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.16 Time#initialize

```
initialize
```

Visibility: private

Behavior:

a) Set the microseconds attribute of the receiver to microseconds elapsed since January 1, 1970 00:00 UTC.

b) Set the time zone attribute of the receiver to the current local time zone.

c) Return an implementation-defined value.

15.2.19.7.17 Time#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If *original* is not an instance of the class `Time`, raise a direct instance of the class `TypeError`.

b) Set the microseconds attribute of the receiver to the microseconds attribute of *original*.

c) Set the time zone attribute of the receiver to the time zone attribute of *original*.

d) Return an implementation-defined value.

15.2.19.7.18 Time#localtime

`localtime`

Visibility: public

Behavior:

a) Change the time zone attribute of the receiver to the current local time zone.

b) Return the receiver.

15.2.19.7.19 Time#mday

`mday`

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $DayWithinMonth(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.20 Time#min

```
min
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $MinuteFromTime(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.21 Time#mon

```
mon
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $MonthFromTime(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.22 Time#month

```
month
```

Visibility: public

Behavior: Same as the method `mon` (see 15.2.19.7.21).

15.2.19.7.23 Time#sec

```
sec
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $SecondFromTime(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.24 Time#to_f

`to_f`

Visibility: public

Behavior: Let t the microseconds attribute of the receiver.

a) Compute $t/10^6$.

b) Return a direct instance of the class `Float` whose value is the result of Step a).

15.2.19.7.25 Time#to_i

`to_i`

Visibility: public

Behavior: Let t the microseconds attribute of the receiver.

a) Compute $floor(t/10^6)$.

b) Return an instance of the class `Integer` whose value is the result of Step a).

15.2.19.7.26 Time#usec

`usec`

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute t modulo 10^6.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.27 Time#utc

```
utc
```

Visibility: public

Behavior:

a) Change the time zone attribute of the receiver to UTC.

b) Return the receiver.

15.2.19.7.28 Time#utc?

```
utc?
```

Visibility: public

Behavior: Let Z be the time zone attribute of the receiver.

a) If Z is UTC, return **true**.

b) Otherwise, return **false**.

15.2.19.7.29 Time#utc_offset

```
utc_offset
```

Visibility: public

Behavior: Let Z be the time zone attribute of the receiver.

a) Compute $floor(ZoneOffset(Z)/10^6)$.

b) Return an instance of the class `Integer` whose value is the result of Step a).

15.2.19.7.30 Time#wday

```
wday
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $WeekDay(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b)

15.2.19.7.31 Time#yday

```
yday
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $DayWithinYear(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.32 Time#year

```
year
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see 15.2.19.4). Let t be the result.

b) Compute $YearFromTime(t)$.

c) Return an instance of the class `Integer` whose value is the result of Step b).

15.2.19.7.33 Time#zone

```
zone
```

Visibility: public

Behavior: Let Z be the time zone attribute of the receiver.

a) Create a direct instance of the class `String`, the content of which represents the name of Z. The exact content of the instance is implementation-defined.

b) Return the resulting instance.

15.2.20 IO

15.2.20.1 General description

An instance of the class `IO` represents a stream, which is a source and/or a sink of data.

An instance of the class `IO` has the following attributes:

readability flag: A boolean value which indicates whether the stream can handle input operations.

An instance of the class `IO` is said to be ***readable*** if and only if this flag is true.

Reading from a stream which is not readable raises a direct instance of the class `IOError`.

writability flag: A boolean value which indicates whether the stream can handle output operations.

An instance of the class `IO` is said to be ***writable*** if and only if this flag is true.

Writing to a stream which is not writable raises a direct instance of the class `IOError`.

openness flag: A boolean value which indicates whether the stream is open.

An instance of the class `IO` is said to be ***open*** if and only if this flag is true. An instance of the class `IO` is said to be ***closed*** if and only if this flag is false.

A closed stream is neither readable nor writable.

buffering flag: A boolean value which indicates whether the data to be written to the stream is buffered.

When this flag is true, the output to the receiver may be delayed until the instance methods `flush` or `close` is invoked.

An instance of the class `SystemCallError` may be raised when the underlying system reported an error during the execution of methods of the class `IO`.

The behavior of the method `initialize` of the class `IO` is unspecified, i.e. whether a direct instance of the class `IO` other than the constnats STDIN, STDOUT and STDERR of the class `Object` (see 15.2.1) can be created is unspecified.

NOTE Note that an instance of the class `File`, which is a subclass of the class `IO`, can be created by the method `new` because the behavior of the method `initialize` is specified in 15.2.21.4.1.

In the following description of the methods of the class `IO`, a ***byte*** means an integer from 0 to 255.

15.2.20.2 Direct superclass

The class `Object`

15.2.20.3 Included modules

The following module is included in the class `IO`.

- `Enumerable`

15.2.20.4 Singleton methods

15.2.20.4.1 IO.open

`IO.open(` `*`*args,* `&`*block* `)`

Visibility: public

Behavior:

a) Invoke the method `new` on the receiver with all the elements of *args* as the arguments. Let I be the resulting value.

b) If *block* is not given, return I.

c) Otherwise, call *block* with I as the argument. Let V be the resulting value.

d) Invoke the method `close` (see 15.2.20.5.1) on I with no arguments, even when an exception is raised and not handled in Step c).

e) Return V.

EXAMPLE If *block* is given, the method `close` is invoked automatically.

```
File.open("data.txt"){|f|
  puts f.read
}
```

If *block* is not given, the method `close` should be invoked explicitly.

```
f = File.open("data.txt")
puts f.read
f.close
```

NOTE The behavior of invoking the method `new` on the class `IO` is unspecified. Therefore, the behavior of invoking the method `open` on the class `IO` is also unspecified; however, the method `open` can be invoked on the class `File`, which is a subclass of the class `IO`.

15.2.20.5 Instance methods

15.2.20.5.1 IO#close

`close`

Visibility: public

Behavior:

a) If the receiver is closed, raise a direct instance of the class `IOError`.

b) If the buffering flag of the receiver is true, and the receiver is buffering any output, immediately write all the buffered data to the stream which the receiver represents.

c) Set the openness flag of the receiver to false.

d) Return an implementation-defined value.

15.2.20.5.2 IO#closed?

```
closed?
```

Visibility: public

Behavior:

a) If the receiver is closed, return **true**.

b) Otherwise, return **false**.

15.2.20.5.3 IO#each

each(&*block*)

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) If the receiver is not readable, raise a direct instance of the class `IOError`.

c) If the receiver has reached its end, return the receiver.

d) Otherwise, read characters from the receiver until 0x0a is read or the receiver reaches its end.

e) Create a direct instance of the class `String` whose content is the sequence of characters read in Step d). Call *block* with this instance as an argument.

f) Continue processing from Step c).

15.2.20.5.4 IO#each_byte

each_byte(&*block*)

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) If the receiver is not readable, raise a direct instance of the class `IOError`.

c) If the receiver has reached its end, return the receiver.

d) Otherwise, read a single byte from the receiver. Call *block* with an argument, an instance of the class `Integer` whose value is the byte.

e) Continue processing from Step c).

15.2.20.5.5 IO#each_line

`each_line(&block)`

Visibility: public

Behavior: Same as the method `each` (see 15.2.20.5.3).

15.2.20.5.6 IO#eof?

`eof?`

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end, return **true**. Otherwise, return **false**.

15.2.20.5.7 IO#flush

`flush`

Visibility: public

Behavior:

a) If the receiver is not writable, raise a direct instance of the class `IOError`.

b) If the buffering flag of the receiver is true, and the receiver is buffering any output, immediately write all the buffered data to the stream which the receiver represents.

c) Return the receiver.

15.2.20.5.8 IO#getc

```
getc
```

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end, return **nil**.

c) Otherwise, read a character from the receiver. Return an instance of the class `Object` which represents the character (see 15.2.10.1).

15.2.20.5.9 IO#gets

```
gets
```

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end, return **nil**.

c) Otherwise, read characters from the receiver until 0x0a is read or the receiver reaches its end.

d) Create a direct instance of the class `String` whose content is the sequence of characters read in Step c) and return this instance.

15.2.20.5.10 IO#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior: The behavior of the method is unspecified.

15.2.20.5.11 IO#print

`print(` **args* `)`

Visibility: public

Behavior:

a) For each element of *args* in the indexing order:

 1) Let V be the element. If the element is **nil**, a conforming processor may create a direct instance of the class `String` whose content is "nil" and let V be the instance.

 2) Invoke the method `write` on the receiver with V as the argument.

b) Return an implementation-defined value.

15.2.20.5.12 IO#putc

`putc(` *obj* `)`

Visibility: public

Behavior:

a) If *obj* is not an instance of the class `Integer` or an instance of the class `String`, the behavior is unspecified. If *obj* is an instance of the class `Integer` whose value is smaller than 0 or larger than 255, the behavior is unspecified.

b) If *obj* is an instance of the class `Integer`, create a direct instance S of the class `String` whose content is a single character, whose character code is the value of *obj*.

c) If *obj* is an instance of the class `String`, create a direct instance S of the class `String` whose content is the first character of *obj*.

d) Invoke the method `write` on the receiver with S as the argument.

e) Return *obj*.

15.2.20.5.13 IO#puts

`puts(*`*args* `)`

Visibility: public

Behavior:

a) If the length of *args* is 0, create a direct instance of the class `String` whose content is a single character 0x0a and invoke the method `write` on the receiver with this instance as an argument.

b) Otherwise, for each element E of *args* in the indexing order:

1) If E is an instance of the class `Array`, for each element X of E in the indexing order:

i) If X is the same object as E, i.e. if E contains itself, invoke the method `write` on the receiver with an instance of the class `String`, whose content is implementation-defined.

ii) Otherwise, invoke the method `write` on the receiver with X as the argument.

2) Otherwise:

i) If E is **nil**, a conforming processor may create a direct instance of the class `String` whose content is “nil” and let E be the instance.

ii) If E is not an instance of the class `String`, invoke the method `to_s` on the E. If the resulting value is an instance of the class `String`, let E be the resulting value. Otherwise, the behavior is unspecified.

iii) Invoke the method `write` on the receiver with E as the argument.

iv) If the last character of E is not 0x0a, create a direct instance of the class `String` whose content is a single character 0x0a and invoke the method `write` on the receiver with this instance as an argument.

c) Return an implementation-defined value.

15.2.20.5.14 IO#read

`read(` *length* `=nil )`

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end:

1) If *length* is **nil**, create an empty instance of the class `String` and return that instance.

2) If *length* is not **nil**, return **nil**.

c) Otherwise:

1) If *length* is **nil**, read characters from the receiver until the receiver reaches its end.

2) If *length* is an instance of the class `Integer`, let N be the value of *length*. Otherwise, the behavior is unspecified.

3) If N is smaller than 0, raise a direct instance of the class `ArgumentError`.

4) Read bytes from the receiver until N bytes are read or the receiver reaches its end.

d) Create a direct instance of the class `String` whose content is the sequence of characters read in Step c) and return this instance.

15.2.20.5.15 IO#readchar

```
readchar
```

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end, raise a direct instance of the class `EOFError`.

c) Otherwise, read a character from the receiver. Return an instance of the class `Object` which represents the character.

15.2.20.5.16 IO#readline

```
readline
```

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) If the receiver has reached its end, raise a direct instance of the class `EOFError`.

c) Otherwise, read characters from the receiver until 0x0a is read or the receiver reaches its end.

d) Create a direct instance of the class `String` whose content is the sequence of characters read in Step c) and return this instance.

15.2.20.5.17 IO#readlines

```
readlines
```

Visibility: public

Behavior:

a) If the receiver is not readable, raise a direct instance of the class `IOError`.

b) Create an empty direct instance A of the class `Array`.

c) If the receiver has reached to its end, return A.

d) Otherwise, read characters from the receiver until 0x0a is read or the receiver reaches its end.

e) Create a direct instance of the class `String` whose content is the sequence of characters read in Step d) and append this instance to A.

f) Continue processing from Step c).

15.2.20.5.18 IO#sync

```
sync
```

Visibility: public

Behavior:

a) If the receiver is closed, raise a direct instance of the class `IOError`.

b) If the buffering flag of the receiver is true, return **false**. Otherwise, return **true**.

15.2.20.5.19 IO#sync=

`sync=(` *bool* `)`

Visibility: public

Behavior:

a) If the receiver is closed, raise a direct instance of the class `IOError`.

b) If *bool* is a trueish object, set the buffering flag of the receiver to false. If *bool* is a falseish object, set the buffering flag of the receiver to true.

c) Return *bool*.

15.2.20.5.20 IO#write

`write(` *str* `)`

Visibility: public

Behavior:

a) If *str* is an instance of the class `String`, let S be *str*.

b) Otherwise, invoke the method `to_s` on *str*, and let S be the resulting value. If S is not an instance of the class `String`, the behavior is unspecified.

c) If S is empty, return an instance of the class `Integer` whose value is 0.

d) If the receiver is not writable, raise a direct instance of the class `IOError`.

e) Write all the characters in S to the stream which the receiver represents, preserving their order.

f) Return an instance of the class `Integer`, whose value is implementation-defined.

15.2.21 File

15.2.21.1 General description

Instances of the class `File` represent opened files.

A conforming processor may raise an instance of the class `SystemCallError` during the execution of the methods of the class `File` if the underlying system reports an error.

An instance of the class `File` has the following attribute:

path: The sequence of characters which represents the location of the file. The correct syntax of paths is implementation-defined.

15.2.21.2 Direct superclass

The class `IO`

15.2.21.3 Singleton methods

15.2.21.3.1 File.exist?

`File.exist?(` *path* `)`

Visibility: public

Behavior:

a) If the file specified by *path* exists, return **true**.

b) Otherwise, return **false**.

15.2.21.4 Instance methods

15.2.21.4.1 File#initialize

```
initialize( path, mode="r" )
```

Visibility: private

Behavior:

a) If *path* is not an instance of the class `String`, the behavior is unspecified.

b) If *mode* is not an instance of the class `String` whose content is a single character "r" or "w", the behavior is unspecified.

c) Open the file specified by *path* in an implementation-defined way, and associate it with the receiver.

d) Set the path of the receiver to the content of *path*.

e) Set the openness flag and the buffering flag of the receiver to true.

f) Set the readability flag and the writability flag of the receiver as follows:

   1) If *mode* is an instance of the class `String` whose content is a single character "r", set the readability flag of the receiver to true and set the writability flag of the receiver to false.

   2) If *mode* is an instance of the class `String` whose content is a single character "w", set the readability flag of the receiver to false and set the writability flag of the receiver to true.

g) Return an implementation-defined value.

15.2.21.4.2 File#path

```
path
```

Visibility: public

Behavior: The method creates a direct instance of the class `String` whose content is the path of the receiver, and returns this instance.

15.2.22 Exception

15.2.22.1 General description

Instances of the class `Exception` represent exceptions. The class `Exception` is a superclass of all the other exception classes.

Instances of the class `Exception` have the following attribute.

message: An object returned by the method `to_s` (see 15.2.22.5.3).

When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel` is invoked on an instance of the class `Exception`, the message attribute shall be copied from the receiver to the resulting value.

15.2.22.2 Direct superclass

The class `Object`

15.2.22.3 Built-in exception classes

This document defines several built-in subclasses of the class `Exception`. Figure 1 shows the list of these subclasses and their class hierarchy. Instances of these built-in subclasses are raised in various erroneous conditions as described in this document. The class hierarchy shown in Figure 1 is used to handle an exception (see Clause 14).

Figure 1 – The exception class hierarchy

```
Exception
  ├── StandardError
  │     ├── ArgumentError
  │     ├── LocalJumpError
  │     ├── RangeError
  │     ├── RegexpError
  │     ├── RuntimeError
  │     ├── TypeError
  │     ├── ZeroDivisionError
  │     ├── NameError
  │     │     └── NoMethodError
  │     ├── IndexError
  │     ├── IOError
  │     │     └── EOFError
  │     └── SystemCallError
  └── ScriptError
        ├── SyntaxError
        └── LoadError
```

15.2.22.4 Singleton methods

15.2.22.4.1 Exception.exception

`Exception.exception(` $*args$, $\&block$ `)`

Visibility: public

Behavior: Same as the method `new` (see 15.2.3.3.3).

15.2.22.5 Instance methods

15.2.22.5.1 Exception#exception

`exception(` **string* `)`

Visibility: public

Behavior:

a) If the length of *string* is 0, return the receiver.

b) If the length of *string* is 1:

   1) If the only argument is the same object as the receiver, return the receiver.

   2) Otherwise let M be the argument.

      i) Create a direct instance of the class of the receiver. Let E be the instance.

      ii) Set the message attribute of E to M.

      iii) Return E.

c) If the length of *string* is larger than 1, raise a direct instance of the class `ArgumentError`.

15.2.22.5.2 Exception#message

`message`

Visibility: public

Behavior:

a) Invoke the method `to_s` on the receiver with no arguments.

b) Return the resulting value of the invocation.

15.2.22.5.3 Exception#to_s

`to_s`

Visibility: public

Behavior:

a) Let M be the message attribute of the receiver.

b) If M is **nil**, return an implementation-defined value.

c) If M is not an instance of the class `String`, the behavior is unspecified.

d) Otherwise, return M.

15.2.22.5.4 Exception#initialize

`initialize(` *message* `=nil )`

Visibility: private

Behavior:

a) Set the message attribute of the receiver to *message*.

b) Return an implementation-defined value.

15.2.23 StandardError

15.2.23.1 General description

Instances of the class `StandardError` represent standard errors, which can be handled in a *rescue-clause* without a *exception-class-list* (see 11.5.2.5).

15.2.23.2 Direct superclass

The class `Exception`

15.2.24 ArgumentError

15.2.24.1 General description

Instances of the class `ArgumentError` represent argument errors.

15.2.24.2 Direct superclass

The class `StandardError`

15.2.25 LocalJumpError

Instances of the class `LocalJumpError` represent errors which occur while evaluating *block*s and *jump-expression*s.

15.2.25.1 Direct superclass

The class `StandardError`

15.2.25.2 Instance methods

15.2.25.2.1 LocalJumpError#exit_value

```
exit_value
```

Visibility: public

Behavior: The method returns the value of the instance variable `@exit_value` of the receiver.

15.2.25.2.2 LocalJumpError#reason

```
reason
```

Visibility: public

Behavior: The method returns the value of the instance variable `@reason` of the receiver.

15.2.26 RangeError

15.2.26.1 General description

Instances of the class `RangeError` represent range errors.

15.2.26.2 Direct superclass

The class `StandardError`

15.2.27 RegexpError

15.2.27.1 General description

Instances of the class `ArgumentError` represent regular expression errors.

15.2.27.2 Direct superclass

The class `StandardError`

15.2.28 RuntimeError

15.2.28.1 General description

Instances of the class `RuntimeError` represent runtime errors, which are raised by the method `raise` of the class `Kernel` by default (see 15.3.1.2.12).

15.2.28.2 Direct superclass

The class `StandardError`

15.2.29 TypeError

15.2.29.1 General description

Instances of the class `TypeError` represent type errors.

15.2.29.2 Direct superclass

The class `StandardError`

15.2.30 ZeroDivisionError

15.2.30.1 General description

Instances of the class `ZeroDivisionError` represent zero division errors.

15.2.30.2 Direct superclass

The class `StandardError`

15.2.31 NameError

Instances of the class `NameError` represent errors which occur while resolving names to values.

Instances of the class `NameError` have the following attribute.

name: The name a reference to which causes this exception to be raised.

When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel` is invoked on an instance of the class `NameError`, the name attribute shall be copied from the receiver to the resulting value.

15.2.31.1 Direct superclass

The class `StandardError`

15.2.31.2 Instance methods

15.2.31.2.1 NameError#name

`name`

Visibility: public

Behavior: The method returns the name attribute of the receiver.

15.2.31.2.2 NameError#initialize

`initialize(` *message* `=nil,` *name* `=nil )`

Visibility: public

Behavior:

a) Set the name attribute of the receiver to the *name*.

b) Invoke the method `initialize` defined in the class `Exception`, which is a superclass of the class `NameError`, as if a *super-with-argument* were evaluated with a list of arguments which contains only *message* as the value of the *argument-without-parentheses* of the *super-with-argument*.

c) Return an implementation-defined value.

15.2.32 NoMethodError

Instances of the class `NoMethodError` represent errors which occur while invoking methods which do not exist or which cannot be invoked.

Instances of the class `NoMethodError` have attributes called ***name*** (see 15.2.31) and ***arguments***. The values of these attributes are set in the method `initialize` (see 15.2.32.2.2).

When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel` is invoked on an instance of the class `NoMethodError`, those attributes shall be copied from the receiver to the resulting value.

15.2.32.1 Direct superclass

The class `NameError`

15.2.32.2 Instance methods

15.2.32.2.1 NoMethodError#args

`args`

Visibility: public

Behavior: The method returns the value of the arguments attribute of the receiver.

15.2.32.2.2 NoMethodError#initialize

`initialize(` *message* `=nil`, *name* `=nil`, *args* `=nil )`

Visibility: private

Behavior:

a) Set the arguments attribute of the receiver to the *args*.

b) Perform all the steps of the method `initialize` described in 15.2.31.2.2.

c) Return an implementation-defined value.

15.2.33 IndexError

15.2.33.1 General description

Instances of the class `IndexError` represent index errors.

15.2.33.2 Direct superclass

The class `StandardError`

15.2.34 IOError

15.2.34.1 General description

Instances of the class `IOError` represent input/output errors.

15.2.34.2 Direct superclass

The class `StandardError`

15.2.35 EOFError

15.2.35.1 General description

Instances of the class `EOFError` represent errors which occur when a stream has reached its end.

15.2.35.2 Direct superclass

The class `IOError`

15.2.36 SystemCallError

15.2.36.1 General description

Instances of the class `SystemCallError` represent errors which occur while invoking the methods of the class `IO`.

15.2.36.2 Direct superclass

The class `StandardError`

15.2.37 ScriptError

15.2.37.1 General description

Instances of the class `ScriptError` represent programming errors such as syntax errors and loading errors.

15.2.37.2 Direct superclass

The class `Exception`

15.2.38 SyntaxError

15.2.38.1 General description

Instances of the class `SyntaxError` represent syntax errors.

15.2.38.2 Direct superclass

The class `ScriptError`

15.2.39 LoadError

15.2.39.1 General description

Instances of the class `LoadError` represent errors which occur while loading external programs (see 15.3.1.2.13).

15.2.39.2 Direct superclass

The class `ScriptError`

15.3 Built-in modules

15.3.1 Kernel

15.3.1.1 General description

The module `Kernel` is included in the class `Object`. Unless overridden, instance methods defined in the module `Kernel` can be invoked on any instance of the class `Object`.

15.3.1.2 Singleton methods

15.3.1.2.1 Kernel.`

`Kernel.`(` *string* `)`

Visibility: public

Behavior: The method ` is invoked in the form described in 8.7.6.3.7.

The method ` executes an external command corresponding to *string*. The external command executed by the method is implementation-defined.

When the method is invoked, take the following steps:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Execute the command which corresponds to the content of *string*. Let R be the output of the command.

c) Create a direct instance of the class `String` whose content is R, and return the instance.

15.3.1.2.2 Kernel.block_given?

```
Kernel.block_given?
```

Visibility: public

Behavior:

a) If the top of ⟦block⟧ is block-not-given, return **false**.

b) Otherwise, return **true**.

15.3.1.2.3 Kernel.eval

`Kernel.eval(` *string* `)`

Visibility: public

Behavior:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Parse the content of the *string* as a *program* (see 10.1). If it fails, raise a direct instance of the class `SyntaxError`.

c) Evaluate the *program* (see 10.1) within the execution context as it exists just before this method invoked. Let V be the resulting value of the evaluation.

d) Return V.

In Step c), the local variable scope which corresponds to the *program* is considered as a local variable scope which corresponds to a *block* in 9.2 d) 1).

EXAMPLE 1 The following program prints “123”.

```
x = 123
Kernel.eval("print x")
```

EXAMPLE 2 The following program raises an exception.

```
Kernel.eval("x = 123") # the scope of x is the program "x = 123"。
print x                # x is undefined here.
```

15.3.1.2.4 Kernel.global_variables

```
Kernel.global_variables
```

Visibility: public

Behavior: The method returns a new direct instance of the class `Array` which consists of names of all the global variables. These names are represented by direct instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-defined.

15.3.1.2.5 Kernel.iterator?

```
Kernel.iterator?
```

Visibility: public

Behavior: Same as the method `Kernel.block_given?` (see 15.3.1.2.2).

15.3.1.2.6 Kernel.lambda

Kernel.lambda(&*block*)

Visibility: public

Behavior: The method creates an instance of the class `Proc` as `Proc.new` does (see 15.2.17.3.1). However, the way in which *block* is evaluated differs from the one described in 11.3.3 except when *block* is called by a *yield-expression*.

The differences are as follows.

a) Before 11.3.3 d), the number of arguments is checked as follows:

1) Let A be the list of arguments passed to the block. Let N_a be the length of A.

2) If the *block-parameter-list* is of the form *left-hand-side*, and if N_a is not 1, the behavior is unspecified.

3) If the *block-parameter-list* is of the form *multiple-left-hand-side*:

i) If the *multiple-left-hand-side* is not of the form *grouped-left-hand-side* or *packing-left-hand-side*:

I) Let N_p be the number of *multiple-left-hand-side-item*s of the *multiple-left-hand-side*.

II) If $N_a < N_p$, raise a direct instance of the class `ArgumentError`.

III) If a *packing-left-hand-side* is omitted, and if $N_a > N_p$, raise a direct instance of the class `ArgumentError`.

ii) If the *multiple-left-hand-side* is of the form *grouped-left-hand-side*, and if N_a is not 1, the behavior is unspecified.

b) In 11.3.3 e), when the evaluation of the block associated with a `lambda` invocation is terminated by a *return-expression* or *break-expression*, the execution context is restored to E_o (i.e. the saved execution context), and the evaluation of the `lambda` invocation is terminated.

The value of the `lambda` invocation is determined as follows:

1) If the *jump-argument* of the *return-expression* or the *break-expression* is present, the value of the `lambda` invocation is the value of the *jump-argument*.

2) Otherwise, the value of the `lambda` invocation is **nil**.

15.3.1.2.7 Kernel.local_variables

`Kernel.local_variables`

Visibility: public

Behavior: The method returns a new direct instance of the class `Array` which contains all the names of local variable bindings which meet the following conditions.

- The name of the binding is of the form *local-variable-identifier*.
- The binding can be resolved in the scope of local variables which includes the point of invocations of this method by the process described in 9.2.

In the instance of the class `Array` returned by the method, names of the local variables are represented by instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-defined.

15.3.1.2.8 Kernel.loop

`Kernel.loop(` &*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Otherwise, repeat calling *block*.

15.3.1.2.9 Kernel.p

`Kernel.p(` **args* `)`

Visibility: public

Behavior:

a) For each element E of *args*, in the indexing order, take the following steps:

   1) Invoke the method `inspect` (see 15.3.1.3.17) on E with no arguments and let X be the resulting value of this invocation.

   2) If X is not an instance of the class `String`, the behavior is unspecified.

   3) Invoke the method `write`(see 15.2.20.5.20) on `Object::STDOUT` with X as the argument.

   4) Invoke the method `write` on `Object::STDOUT` with an argument, which is a new direct instance of the class `String` whose content is a single character 0x0a.

b) Return an implementation-defined value.

15.3.1.2.10 Kernel.print

`Kernel.print(` *`*args`* `)`

Visibility: public

Behavior: Invoke the method `print` of the class `IO` (see 15.2.20.5.11) on `Object::STDOUT` with the same arguments, and return the resulting value.

15.3.1.2.11 Kernel.puts

`Kernel.puts(` *`*args`* `)`

Visibility: public

Behavior: Invoke the method `puts` of the class `IO` (see 15.2.20.5.13) on `Object::STDOUT` with the same arguments, and return the resulting value.

15.3.1.2.12 Kernel.raise

`Kernel.raise(` *`*args`* `)`

Visibility: public

Behavior:

a) If the length of *args* is larger than 2, the behavior is unspecified.

b) If the length of *args* is 0:

1) If the location of the method invocation is within an *operator-expression*$_2$ of an *assignment-with-rescue-modifier*, a *fallback-statement-of-rescue-modifier-statement*, or a *rescue-clause*, let E be the current exception (see 14.3).

2) Otherwise, invoke the method `new` on the class `RuntimeError` with no argument. Let E be the resulting value.

c) If the length of *args* is 1, let A be the only argument.

1) If A is an instance of the class `String`, invoke the method `new` on the class `RuntimeError` with A as the only argument. Let E be the resulting instance.

2) Otherwise, invoke the method `exception` on A. Let E be the resulting value.

3) If E is not an instance of the class `Exception`, raise a direct instance of the class `TypeError`.

d) If the length of *args* is 2, let F and S be the first and the second argument, respectively.

1) Invoke the method `exception` on F with S as the only argument. Let E be the resulting value.

2) If E is not an instance of the class `Exception`, raise a direct instance of the class `TypeError`.

e) Raise E.

15.3.1.2.13 Kernel.require

`Kernel.require(` *string* `)`

Visibility: public

Behavior: The method `require` evaluates the external program P corresponding to *string*. The way in which P is determined from *string* is implementation-defined.

When the method is invoked, take the following steps:

a) If *string* is not an instance of the class `String`, the behavior is unspecified.

b) Search for the external program P corresponding to *string*.

c) If the program does not exist, raise a direct instance of the class `LoadError`.

d) If P is not of the form *program* (see 10.1), raise a direct instance of the class `SyntaxError`.

e) Change the state of the execution context temporarily for the evaluation of P as follows:

1) ⟦self⟧ contains only one object which is the object at the bottom of ⟦self⟧ in the current execution context.

2) ⟦class-module-list⟧ contains only one list whose only element is the class `Object`.

3) ⟦default-method-visibility⟧ contains only one visibility, which is the private visibility.

4) All the other attributes of the execution context are empty stacks.

f) Evaluate P within the execution context set up in Step e).

g) Restore the state of the execution context as it is just before Step e), even when an exception is raised and not handled during the evaluation of P.

NOTE The evaluation of P may affect the restored execution context if it changes the attributes of objects in the original execution context.

h) Unless an exception is raised and not handled in Step f), return **true**.

15.3.1.3 Instance methods

15.3.1.3.1 Kernel#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If the receiver and *other* are the same object, return **true**.

b) Otherwise, return **false**.

If the class `Object` is not the root of the class inheritance tree, the method `==` shall be defined in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

15.3.1.3.2 Kernel#===

`===(` *other* `)`

Visibility: public

Behavior:

a) If the receiver and *other* are the same object, return **true**.

b) Otherwise, invoke the method `==` on the receiver with *other* as the only argument. Let V be the resulting value.

c) If V is a trueish object, return **true**. Otherwise, return **false**.

15.3.1.3.3 Kernel#__id__

```
__id__
```

Visibility: public

Behavior: Same as the method `object_id` (see 15.3.1.3.33).

15.3.1.3.4 Kernel#__send__

`__send__(` *symbol*, `*`*args*, `&`*block* `)`

Visibility: public

Behavior: Same as the method `send` (see 15.3.1.3.44).

If the class `Object` is not the root of the class inheritance tree, the method `__send__` shall be defined in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

15.3.1.3.5 Kernel#`

`` `( `` *string* `)`

Visibility: private

Behavior: Same as the method ``Kernel.` `` (see 15.3.1.2.1).

15.3.1.3.6 Kernel#block_given?

```
block_given?
```

Visibility: private

Behavior: Same as the method `Kernel.block_given?` (see 15.3.1.2.2).

15.3.1.3.7 Kernel#class

```
class
```

Visibility: public

Behavior: The method returns the class of the receiver.

15.3.1.3.8 Kernel#clone

`clone`

Visibility: public

Behavior:

a) If the receiver is an instance of one of the following classes: `NilClass`, `TrueClass`, `FalseClass`, `Integer`, `Float`, or `Symbol`, then raise a direct instance of the class `TypeError`.

b) Create a direct instance of the class of the receiver which has no bindings of instance variables. Let O be the newly created instance.

c) For each binding B of the instance variables of the receiver, create a variable binding with the same name and value as B in the set of bindings of instance variables of O.

d) If the receiver is associated with a singleton class, let E_o be the singleton class, and take the following steps:

   1) Create a singleton class whose direct superclass is the direct superclass of E_o. Let E_n be the singleton class.

   2) For each binding B_{v1} of the constants of E_o, create a variable binding with the same name and value as B_{v1} in the set of bindings of constants of E_n.

   3) For each binding B_{v2} of the class variables of E_o, create a variable binding with the same name and value as B_{v2} in the set of bindings of class variables of E_n.

   4) For each binding B_m of the instance methods of E_o, create a method binding with the same name and value as B_m in the set of bindings of instance methods of E_n.

   5) Associate O with E_n.

e) Invoke the method `initialize_copy` on O with the receiver as the argument.

f) Return O.

15.3.1.3.9 Kernel#dup

`dup`

Visibility: public

Behavior:

a) If the receiver is an instance of one of the following classes: `NilClass`, `TrueClass`, `FalseClass`, `Integer`, `Float`, or `Symbol`, then raise a direct instance of the class `TypeError`.

b) Create a direct instance of the class of the receiver which has no bindings of instance variables. Let O be the newly created instance.

c) For each binding B of the instance variables of the receiver, create a variable binding with the same name and value as B in the set of bindings of instance variables of O.

d) Invoke the method `initialize_copy` on O with the receiver as the argument.

e) Return O.

15.3.1.3.10 Kernel#eql?

`eql?(` *other* `)`

Visibility: public

Behavior: Same as the method `==` (see 15.3.1.3.1).

15.3.1.3.11 Kernel#equal?

`equal?(` *other* `)`

Visibility: public

Behavior: Same as the method `==` (see 15.3.1.3.1).

If the class `Object` is not the root of the class inheritance tree, the method `equal?` shall be defined in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

15.3.1.3.12 Kernel#eval

`eval(` *string* `)`

Visibility: private

Behavior: Same as the method `Kernel.eval` (see 15.3.1.2.3).

15.3.1.3.13 Kernel#extend

`extend(*`*module_list* `)`

Visibility: public

Behavior: Let R be the receiver of the method.

a) If the length of *module_list* is 0, raise a direct instance of the class `ArgumentError`.

b) For each element A of *module_list*, take the following steps:

1) If A is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

2) If A is an instance of the class `Class`, raise a direct instance of the class `TypeError`.

3) Invoke the method `extend_object` on A with R as the only argument.

4) Invoke the method `extended` on A with R as the only argument.

c) Return R.

15.3.1.3.14 Kernel#global_variables

`global_variables`

Visibility: private

Behavior: Same as the method `Kernel.global_variables` (see 15.3.1.2.4).

15.3.1.3.15 Kernel#hash

`hash`

Visibility: public

Behavior: The method returns an instance of the class `Integer`. When invoked on the same object, the method shall always return an instance of the class `Integer` whose value is the same.

When a conforming processor overrides the method `eql?` (see 15.3.1.3.10), it shall override the method `hash` in the same class or module in which the method `eql?` is overridden in such a way that, if an invocation of the method `eql?` on an object with an argument returns a trueish object, invocations of the method `hash` on the object and the argument return the instances of the class `Integer` with the same value.

15.3.1.3.16 Kernel#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior: The method `initialize_copy` is invoked when an object is created by the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9).

When the method is invoked, take the following steps:

a) If the classes of the receiver and the *original* are not the same class, raise a direct instance of the class `TypeError`.

b) Return an implementation-defined value.

15.3.1.3.17 Kernel#inspect

```
inspect
```

Visibility: public

Behavior: The method returns a new direct instance of the class `String`, the content of which represents the state of the receiver. The content of the resulting instance of the class `String` is implementation-defined.

15.3.1.3.18 Kernel#instance_eval

`instance_eval(` *string* `=` *nil,* `&`*block* `)`

Visibility: public

Behavior:

a) If the receiver is an instance of the class `Integer` or the class `Symbol`, or if the receiver is one of **nil**, **true**, or **false**, then the behavior is unspecified.

b) If the receiver is not associated with a singleton class, create a new singleton class. Let M be the newly created singleton class.

c) If the receiver is associated with a singleton class, let M be that singleton class.

d) Take steps b) through the last step of the method `class_eval` of the class `Module` (see 15.2.2.4.15).

If the class `Object` is not the root of the class inheritance tree, the method `instance_eval` shall be defined in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

15.3.1.3.19 Kernel#instance_of?

`instance_of?(` *module* `)`

Visibility: public

Behavior: Let C be the class of the receiver.

a) If *module* is not an instance of the class `Class` or the class `Module`, raise a direct instance of the class `TypeError`.

b) If *module* and C are the same object, return **true**.

c) Otherwise, return **false**.

15.3.1.3.20 Kernel#instance_variable_defined?

`instance_variable_defined?(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding of an instance variable with name N exists in the set of bindings of instance variables of the receiver, return **true**.

d) Otherwise, return **false**.

15.3.1.3.21 Kernel#instance_variable_get

`instance_variable_get(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding of an instance variable with name N exists in the set of bindings of instance variables of the receiver, return the value of the binding.

d) Otherwise, return **nil**.

15.3.1.3.22 Kernel#instance_variable_set

`instance_variable_set(` *symbol, obj* `)`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding of an instance variable with name N exists in the set of bindings of instance variables of the receiver, replace the value of the binding with *obj*.

d) Otherwise, create a variable binding with name N and value *obj* in the set of bindings of instance variables of the receiver.

e) Return *obj*.

15.3.1.3.23 Kernel#instance_variables

```
instance_variables
```

Visibility: public

Behavior: The method returns a new direct instance of the class `Array` which consists of names of all the instance variables of the receiver. These names are represented by direct instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-defined.

15.3.1.3.24 Kernel#is_a?

```
is_a?( module )
```

Visibility: public

Behavior:

a) If *module* is not an instance of the class `Class` or the class `Module`, raise a direct instance of the class `TypeError`.

b) Let C be the class of the receiver.

c) If *module* is an instance of the class `Class` and one of the following conditions holds, return **true**.

- The *module* and C are the same object.
- The *module* is a superclass of C.
- The *module* and the singleton class of the receiver are the same object.

d) If *module* is an instance of the class `Module` and is included in C or one of the superclasses of C, return **true**.

e) Otherwise, return **false**.

15.3.1.3.25 Kernel#iterator?

```
iterator?
```

Visibility: private

Behavior: Same as the method `Kernel.iterator?` (see 15.3.1.2.5).

15.3.1.3.26 Kernel#kind_of?

`kind_of?(` *module* `)`

Visibility: public

Behavior: Same as the method `is_a?` (see 15.3.1.3.24).

15.3.1.3.27 Kernel#lambda

`lambda(` `&`*block* `)`

Visibility: private

Behavior: Same as the method `Kernel.lambda` (see 15.3.1.2.6).

15.3.1.3.28 Kernel#local_variables

```
local_variables
```

Visibility: private

Behavior: Same as the method `Kernel.local_variables` (see 15.3.1.2.7).

15.3.1.3.29 Kernel#loop

`loop(` `&`*block* `)`

Visibility: private

Behavior: Same as the method `Kernel.loop` (see 15.3.1.2.8).

15.3.1.3.30 Kernel#method_missing

`method_missing(` *symbol*, `*`*args* `)`

Visibility: private

Behavior:

a) If *symbol* is not an instance of the class `Symbol`, the behavior is unspecified.

b) Otherwise, raise a direct instance of the class `NoMethodError` which has *symbol* as its name attribute and *args* as its arguments attribute. A direct instance of the class `NameError` which has *symbol* as its name attribute may be raised instead of `NoMethodError` if the method is invoked in 13.3.3 e) during evaluation of a *local-variable-identifier* as a method invocation.

If the class `Object` is not the root of the class inheritance tree, the method `method_missing` shall be defined in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

15.3.1.3.31 Kernel#methods

`methods(` *all*`=true )`

Visibility: public

Behavior: Let C be the class of the receiver.

a) If *all* is a trueish object, invoke the method `instance_methods` on C with no arguments (see 15.2.2.4.33), and return the resulting value.

b) If *all* is a falseish object, invoke the method `singleton_methods` on the receiver with **false** as the only argument (see 15.3.1.3.45), and return the resulting value.

15.3.1.3.32 Kernel#nil?

`nil?`

Visibility: public

Behavior:

a) If the receiver is **nil**, return **true**.

b) Otherwise, return **false**.

15.3.1.3.33 Kernel#object_id

`object_id`

Visibility: public

Behavior: The method returns an instance of the class `Integer` with the same value whenever it is invoked on the same object. When invoked on two distinct objects, the method returns an instance of the class `Integer` with different value for each invocation.

15.3.1.3.34 Kernel#p

`p(` *`*args`* `)`

Visibility: private

Behavior: Same as the method `Kernel.p` (see 15.3.1.2.9).

15.3.1.3.35 Kernel#print

`print(` *`*args`* `)`

Visibility: private

Behavior: Same as the method `Kernel.print` (see 15.3.1.2.10).

15.3.1.3.36 Kernel#private_methods

`private_methods(` *all*`=true )`

Visibility: public

Behavior:

a) Let MV be the private visibility.

b) Create an empty direct instance A of the class `Array`.

c) If the receiver is associated with a singleton class, let C be the singleton class.

d) Let I be the set of bindings of instance methods of C.

For each binding B of I, let N and V be the name and the value of B respectively, and take the following steps:

1) If V is undef, or the visibility of V is not MV, skip the next two steps.

2) Let S be either a new direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which is chosen as the value of S is implementation-defined.

3) Unless A contains the element of the same name (if S is an instance of the class `Symbol`) or the same content (if S is an instance of the class `String`) as S, append S to A.

e) For each module M in included module list of C, take step d), assuming that C in that step to be M.

f) Let new C be the class of the receiver, and take step d).

g) If *all* is a trueish object:

1) Take step e).

2) If C does not have a direct superclass, return A.

3) Let new C be the direct superclass of current C.

4) Take step d), and then, repeat from Step g) 1).

h) Return A.

15.3.1.3.37 Kernel#protected_methods

`protected_methods(` *all*`=true )`

Visibility: public

Behavior: Same as the method `private_methods` (see 15.3.1.3.36), except to let MV be the protected visibility in 15.3.1.3.36 a).

15.3.1.3.38 Kernel#public_methods

`public_methods(` *all*`=true )`

Visibility: public

Behavior: Same as the method `private_methods` (see 15.3.1.3.36), except to let MV be the public visibility in 15.3.1.3.36 a).

15.3.1.3.39 Kernel#puts

```
puts( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.puts` (see 15.3.1.2.11).

15.3.1.3.40 Kernel#raise

```
raise( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.raise` (see 15.3.1.2.12).

15.3.1.3.41 Kernel#remove_instance_variable

```
remove_instance_variable( symbol )
```

Visibility: private

Behavior:

a) Let *N* be the name designated by *symbol*.

b) If *N* is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

c) If a binding of an instance variable with name *N* exists in the set of bindings of instance variables of the receiver, let *V* be the value of the binding.

   1) Remove the binding from the set of bindings of instance variables of the receiver.

   2) Return *V*.

d) Otherwise, raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

15.3.1.3.42 Kernel#require

```
require( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.require` (see 15.3.1.2.13).

15.3.1.3.43 Kernel#respond_to?

`respond_to?(` *symbol, include_private* `=false )`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) Search for a binding of an instance method named N starting from the receiver of the method as described in 13.3.4.

c) If a binding is found, let V be the value of the binding.

  1) If V is undef, return **false**.

  2) If the visibility of V is private:

    i) If *include_private* is a trueish object, return **true**.

    ii) Otherwise, return **false**.

  3) Otherwise, return **true**.

d) Otherwise, return **false**.

15.3.1.3.44 Kernel#send

`send(` *symbol,* `*`*args,* `&`*block* `)`

Visibility: public

Behavior:

a) Let N be the name designated by *symbol*.

b) Invoke the method named N on the receiver with *args* as arguments and *block* as the block, if any.

c) Return the resulting value of the invocation.

15.3.1.3.45 Kernel#singleton_methods

`singleton_methods(` *all* `=true )`

Visibility: public

Behavior: Let E be the singleton class of the receiver.

a) Create an empty direct instance A of the class `Array`.

b) Let I be the set of bindings of instance methods of E.

For each binding B of I, let N and V be the name and the value of B respectively, and take the following steps:

1) If V is undef, or the visibility of V is private, skip the next two steps.

2) Let S be either a new direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which is chosen as the value of S is implementation-defined.

3) Unless A contains the element of the same name (if S is an instance of the class `Symbol`) or the same content (if S is an instance of the class `String`), append S to A.

c) If *all* is a trueish object, for each module M in included module list of E, take step b), assuming that E in that step to be M.

d) Return A.

15.3.1.3.46 Kernel#to_s

`to_s`

Visibility: public

Behavior: The method returns a newly created direct instance of the class `String`, the content of which is the string representation of the receiver. The content of the resulting instance of the class `String` is implementation-defined.

15.3.2 Enumerable

15.3.2.1 General description

The module `Enumerable` provides methods which iterates over the elements of the object using the method `each`.

In the following description of the methods of the module `Enumerable`, an ***element*** of the receiver means one of the values which is yielded by the method `each`.

15.3.2.2 Instance methods

15.3.2.2.1 Enumerable#all?

`all?(&`*block*`)`

Visibility: public

Behavior:

a) Invoke the method `each` on the receiver.

b) For each element X which the method `each` yields:

  1) If *block* is given, call *block* with X as the argument.

   If this call results in a falseish object, return **false**.

  2) If *block* is not given, and X is a falseish object, return **false**.

c) Return **true**.

15.3.2.2.2 Enumerable#any?

`any?(&`*block*`)`

Visibility: public

Behavior:

a) Invoke the method `each` on the receiver.

b) For each element X which `each` yields:

  1) If *block* is given, call *block* with X as the argument.

   If this call results in a trueish object, return **true**.

  2) If *block* is not given, and X is a trueish object, return **true**.

c) Return **false**.

15.3.2.2.3 Enumerable#collect

`collect(&`*block*`)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Create an empty direct instance A of the class `Array`.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call *block* with X as the argument and append the resulting value to A.

e) Return A.

15.3.2.2.4 Enumerable#detect

`detect(` *ifnone* `=nil,` `&`*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, call *block* with X as the argument. If this call results in a trueish object, return X.

d) Return *ifnone*.

15.3.2.2.5 Enumerable#each_with_index

`each_with_index(` `&`*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Let i be 0.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields:

   1) Call *block* with X and i as the arguments.

   2) Increase i by 1.

e) Return the receiver.

15.3.2.2.6 Enumerable#entries

`entries`

Visibility: public

Behavior:

a) Create an empty direct instance A of the class `Array`.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, append X to A.

d) Return A.

15.3.2.2.7 Enumerable#find

`find(` *ifnone* `=nil,` `&`*block* `)`

Visibility: public

Behavior: Same as the method `detect` (see 15.3.2.2.4).

15.3.2.2.8 Enumerable#find_all

`find_all(` `&`*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Create an empty direct instance A of the class `Array`.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call *block* with X as the argument. If this call results in a trueish object, append X to A.

e) Return A.

15.3.2.2.9 Enumerable#grep

`grep(` *pattern*, `&`*block*`)`

Visibility: public

Behavior:

a) Create an empty direct instance A of the class `Array`.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, invoke the method `===` on *pattern* with X as the argument.

   If this invocation results in a trueish object:

   1) If *block* is given, call *block* with X as the argument and append the resulting value to A.

   2) Otherwise, append X to A.

d) Return A.

15.3.2.2.10 Enumerable#include?

`include?(` *obj*`)`

Visibility: public

Behavior:

a) Invoke the method `each` on the receiver.

b) For each element X which `each` yields, invoke the method `==` on X with *obj* as the argument. If this invocation results in a trueish object, return **true**.

c) Return **false**.

15.3.2.2.11 Enumerable#inject

`inject(*`*args*, `&`*block*`)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) If the length of *args* is 2, the behavior is unspecified. If the length of *args* is larger than 2, raise a direct instance of the class `ArgumentError`.

c) Invoke the method `each` on the receiver. If the method `each` does not yield any element, return **nil**.

d) For each element X which `each` yields:

1) If X is the first element, and the length of *args* is 0, let V be X.

2) If X is the first element, and the length of *args* is 1, call *block* with two arguments, which are the only element of *args* and X. Let V be the resulting value of this call.

3) If X is not the first element, call *block* with V and X as the arguments. Let new V be the resulting value of this call.

e) Return V.

15.3.2.2.12 Enumerable#map

`map( &`*block*`)`

Visibility: public

Behavior: Same as the method `collect` (see 15.3.2.2.3).

15.3.2.2.13 Enumerable#max

`max( &`*block*`)`

Visibility: public

Behavior:

a) Invoke the method `each` on the receiver.

b) If the method `each` does not yield any elements, return **nil**.

c) For each element X which the method `each` yields:

1) If X is the first element, let V be X.

2) Otherwise:

i) If *block* is given:

I) Call *block* with X and V as the arguments. Let D be the result of this call.

III) If the value of D is larger than 0, let new V be X.

ii) If *block* is not given:

I) Invoke the method `<=>` on X with V as the argument. Let D be the result of this invocation.

II) If D is not an instance of the class `Integer`, the behavior is unspecified.

III) If the value of D is larger than 0, let new V be X.

d) Return V.

15.3.2.2.14 Enumerable#min

`min(`&*block*`)`

Visibility: public

Behavior:

a) Invoke the method `each` on the receiver.

b) If the method `each` does not yield any elements, return **nil**.

c) For each element X which the method `each` yields:

1) If X is the first element, let V be X.

2) Otherwise:

i) If *block* is given:

I) Call *block* with X and V as the arguments. Let D be the result of this call.

II) If D is not an instance of the class `Integer`, the behavior is unspecified.

III) If the value of D is smaller than 0, let new V be X.

ii) If *block* is not given:

I) Invoke the method `<=>` on X with V as the argument. Let D be the result of this invocation.

II) If D is not an instance of the class `Integer`, the behavior is unspecified.

III) If the value of D is smaller than 0, let new V be X.

d) Return V.

15.3.2.2.15 Enumerable#member?

`member?(` *obj* `)`

Visibility: public

Behavior: Same as the method `include?` (see 15.3.2.2.10).

15.3.2.2.16 Enumerable#partition

`partition(` &*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Create two empty direct instances of the class `Array` T and F.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call *block* with X as the argument.

   If this call results in a trueish object, append X to T. If this call results in a falseish object, append X to F.

e) Return a newly created an instance of the class `Array`, which contains only T and F in this order.

15.3.2.2.17 Enumerable#reject

`reject(` &*block* `)`

Visibility: public

Behavior:

a) If *block* is not given, the behavior is unspecified.

b) Create an empty direct instance A of the class `Array`.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call *block* with X as the argument. If this call results in a falseish object, append X to A.

e) Return A.

15.3.2.2.18 Enumerable#select

`select(` `&`*block*`)`

Visibility: public

Behavior: Same as the method `find_all` (see 15.3.2.2.8).

15.3.2.2.19 Enumerable#sort

`sort(` `&`*block*`)`

Visibility: public

Behavior:

a) Create an empty direct instance A of the class `Array`.

b) Invoke the method `each` on the receiver.

c) Insert all the elements which the method `each` yields into A. For any two elements E_i and E_j of A, the following condition shall hold:

   1) Let i and j be the index of E_i and E_j, respectively.

   2) If *block* is given:

      i) Suppose *block* is called with E_i and E_j as the arguments.

      ii) If this invocation does not result in an instance of the class `Integer`, the behavior is unspecified.

      iii) If this invocation results in an instance of the class `Integer` whose value is larger than 0, j shall be larger than i.

      iv) If this invocation results in an instance of the class `Integer` whose value is smaller than 0, i shall be larger than j.

   3) If *block* is not given:

      i) Suppose the method `<=>` is invoked on E_i with E_j as the argument.

      ii) If this invocation does not result in an instance of the class `Integer`, the behavior is unspecified.

      iii) If this invocation results in an instance of the class `Integer` whose value is larger than 0, j shall be larger than i.

      iv) If this invocation results in an instance of the class `Integer` whose value is smaller than 0, i shall be larger than j.

d) Return A.

15.3.2.2.20 Enumerable#to_a

`to_a`

Visibility: public

Behavior: Same as the method `entries` (see 15.3.2.2.6).

15.3.3 Comparable

15.3.3.1 General description

The module `Comparable` provides methods which compare the receiver and an argument using the method `<=>`.

15.3.3.2 Instance methods

15.3.3.2.1 Comparable#<

`<(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *other* as the argument. Let I be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is unspecified.

c) If the value of I is smaller than 0, return **true**. Otherwise, return **false**.

15.3.3.2.2 Comparable#<=

`<=(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *other* as the argument. Let I be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is unspecified.

c) If the value of I is smaller than or equal to 0, return **true**. Otherwise, return **false**.

15.3.3.2.3 Comparable#==

`==(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *other* as the argument. Let I be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is unspecified.

c) If the value of I is 0, return **true**. Otherwise, return **false**.

15.3.3.2.4 Comparable#>

`>(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *other* as the argument. Let I be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is unspecified.

c) If the value of I is larger than 0, return **true**. Otherwise, return **false**.

15.3.3.2.5 Comparable#>=

`>=(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *other* as the argument. Let I be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is unspecified.

c) If the value of I is larger than or equal to 0, return **true**. Otherwise, return **false**.

15.3.3.2.6 Comparable#between?

`between?(` *left*, *right* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with *left* as the argument. Let I_1 be the resulting value of this invocation.

   1) If I_1 is not an instance of the class `Integer`, the behavior is unspecified.

   2) If the value of I_1 is smaller than 0, return **false**.

b) Invoke the method `<=>` on the receiver with *right* as the argument. Let I_2 be the resulting value of this invocation.

   1) If I_2 is not an instance of the class `Integer`, the behavior is unspecified.

   2) If the value of I_2 is larger than 0, return **false**. Otherwise, return **true**.