Programming Languages — Ruby

IPA Ruby Standardization WG Draft

December 1, 2009

Contents

Page

Information technology — Programming Languages — Ruby

1 Scope

This document specifies the syntax and semantics of the computer programming language Ruby by specifying requirements for a conforming processor and for a conforming program.

This document does not specify:

- the size or complexity of a program text that exceeds the capacity of any specific data processing system or the capacity of a particular 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;
- the method for reporting syntactic and runtime errors.

2 Normative references

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

ISO/IEC 646:1991 *Information technology – ISO 7-bit coded character set for information interchange.*

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

3 Conformance

A conforming Ruby program shall:

- use only those features of the language specified in this document;
- not rely on implementation dependent features;

A conforming Ruby processor shall:

- accept any conforming programs and behave as specified in this document;
- reject any program which does not conform to the syntax described in this document;
- report any unhandled exceptions raised during execution of the conforming program;

A conforming Ruby processor may use an internal model for the Ruby language other than the one specified in this document, if it does not change the meaning of a conforming program.

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
sequence of statements which is passed to a method invocation

4.2
class
object which defines the behavior of a set of other objects called its instances

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

4.3
class variable
variable whose value is shared by all the instances of a class

4.4
constant
variable which is defined in a class or a module and is accessible outside the class or module

NOTE The value of a constant is regularly expected to remain constant during the execution of a program, but Ruby does not force it. In some implementations, an assignment to a constant which already exists causes a warning, but this document does not specifity it.

4.5
eigenclass
special class which defines a behavior for only a single object

4.6
exception
object which represents an unexpected event

4.7
global variable
variable which is accessible everywhere in a Ruby program

4.8
implementation defined
possibly differing between implementations, but defined for every implementation

4.9
implementation dependent
possibly differing between implementations, and not necessarily defined for any particular implementation

4.10
instance method
method which can be invoked on all the instances of a class

4.11
instance variable
variable which belongs to a single object

4.12
local variable
variable which is accessible only in a certain scope introduced by a program construct such as a method definition, a block, a class definition, a module definition, an eigenclass definition, or the toplevel of a program

4.13
method
procedure which, when invoked on an object, performs a set of computations on the object

4.14
method visibility
attribute of a method which determines the conditions on which a method invocation is allowed

4.15
module
object which provides features to be included into a class or another module

4.16
object
computational entity which has a state and a behavior

4.17
singleton method
instance method of the eigenclass of an object

4.18
variable
computational entity which stores a reference to an object

5 Notational conventions

5.1 Syntax

The syntax of the language is presented as a series of productions. Each production consists of the name of the nonterminal symbol being defined followed by "::", followed by one or more alternatives separated by "|".

Terminal symbols are shown in `typewriter face`, and represent sequences of characters as they appear in a program text. Non-terminal symbols are shown in *italic face*.

Each alternative in a production consists of a sequence of terminal and/or nonterminal symbols separated by whitespace.

If the same nonterminal symbol occurs on the right side of a production more than once, each occurrence is subscripted with a number to distinguish it from the other occurrences of the same name.

An optional symbol is denoted by postfixing the symbol with "?".

A sequence of zero or more repetitions of a symbol is denoted by postfixing the symbol with "*".

A sequence of one or more repetitions of a symbol is denoted by postfixing the symbol with "+".

Parentheses are used to treat a sequence of symbols as a single symbol.

A symbol followed by the phrase **but not** and another symbol represents all sequences of characters represented by the first symbol except for sequences of characters represented by the second symbol.

EXAMPLE 1 The following example means that *non-escaped-character* is any member of *source-character* except *escape-character*:

---

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

Text enclosed by "[" and "]" is used to describe a sequence of characters or a location in a program text.

EXAMPLE 2 The following example means that *source-character* is any character specified in ISO/IEC 646:

---

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

In particular, the notation "[lookahead $\notin$ *set*]" indicates that the token immediately following the notation shall not begin with a sequence of characters represented by one of the members of *set*. The *set* is represented as a list of one or more terminal symbols separated by commas, and the list is enclosed by "{" and "}".

EXAMPLE 3 The following example means that the *argument* following the *method-modifier* shall not begin with "`{`":

---

command ::

method-identifier [lookahead ∉ { `{` }] *argument*

In this document, use of the words ***of*** and ***in***, when expressing a relationship between nonterminal symbols, has the following meanings:

> ***X* of *Y*:** refers to the *X* occurring directly in a production defining *Y*.
>
> ***X* in *Y*:** refers to any *X* occurring in a sequence which is derived directly or indirectly from *Y*.

5.2 Conceptual name

A ***conceptual name*** is a common name given to a set of semantically related nonterminal symbols in the grammar in order to refer to this set in a semantic description. A conceptual name is defined by a ***conceptual name definition***. A conceptual name definition consists of the conceptual name to be defined followed by "::=", followed by one or more nonterminal symbols or conceptual names, separated by "|".

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

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

---

6 Objects

6.1 General description

Ruby is a pure object-oriented language. It is pure in the sense that every value manipulated in a Ruby program is an object including primitive values such as integers.

An object is a computational entity which has a state and a behavior.

A variable is a computational entity which stores a reference to an object. A variable has a name. A variable is said to be ***bound*** to an object if the variable stores a reference 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.

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 the state of that object and is encapsulated in that object.

A method is a procedure which, when invoked, performs a set of computations. 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 to which a method is bound 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.

There are three constructs which define the behavior of objects: classes, eigenclasses, and modules. A class defines methods shared by objects of the same class. An eigenclass is a special class which defines methods for only a single object. A module defines, and provides methods to be included into a class or another module. All these three constructs are represented as objects, which are dynamically created and modified at run-time.

A class creates objects, and the created objects are called ***direct instance***s of the class. A class defines a set of methods which 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 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.

Every object, including classes, can be associated with at most one special class to the object. This special class is called the eigenclass of the object. The eigenclass 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 on only that object. If the object is a class, the singleton methods of the class are similar to so-called class methods because they can be invoked on only that class and its subclasses. An eigenclass is created, and associated with an object by an eigenclass definition (see §13.4.2) or a singleton method definition (see §13.4.3).

A class has a single class or `nil` as its ***direct superclass***. If a class A has a class B as its direct superclass, A is called a ***direct subclass*** of B. Classes form a tree-like hierarchy defined by the direct superclass-subclass relation. There is only one class which has *nil* as its direct superclass. It is the root of the tree. All the ancestors of a class in the tree are called ***superclasse***s of the class. All the descendants of a class in the tree are called ***subclasse***s 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.

Ruby does not support multiple inheritance; that is, a class can have only one direct superclass. However, Ruby supports module inclusion, which is a mechanism to append features into a class from multiple sources.

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 class variables and instance methods. Instance methods and class variables defined in a module can be used by other classes, modules and eigenclasses by including the module into them. While a class can have only one direct superclass, a class or a module 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).

Objects are created at some time during program execution. The lifetime of an object begins when the object is created and ends when all references to it are no longer possible.

6.2 Boolean values

An object is classified into either a ***true value*** or a ***false value***.

Only `false` and `nil` are false values. The pseudo variable `false` is the only instance of the class `FalseClass`, and is represented by the keyword `false`. The pseudo variable `nil` is the only instance of the class `NilClass`, and represented by the keyword `nil`.

Objects other than `false` and `nil` are classified into true values. The pseudo variable `true` is the only instance of the class `TrueClass`, and represented by the keyword `true`.

7 Execution context

7.1 Contextual attributes

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

An execution context is not a part of the language. It is defined in this document only for the description of the semantics of a program. A conforming processor shall evaluate a program as if it acted upon 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 logical stack. The names of attributes are enclosed in double square brackets "⟦" and "⟧". Attributes of an execution context are changed when a program construct is evaluated.

The following are the attributes of an execution context:

⟦self⟧ **:** A logical stack of objects, the top of which is the object to which the pseudo variable `self` is bound (see §11.4.3.7.3). The object at the top of the stack is called the ***current self***.

⟦class-module-list⟧ **:** A logical stack of lists of classes or modules. 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 logical 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 logical 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 §9.1.1), 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 logical stack of names by which methods are invoked.

⟦defined-method-name⟧ **:** A logical stack of names with which the invoked methods are defined.

⟦block⟧**:** A logical stack of blocks passed to method invocations. An element of the stack may be ***block-not-given***, which indicates that no block is passed to a method invocation.

⟦global-variable-bindings⟧**:** A set of bindings of global variables.

The term ***unset*** is used to describe the state of an attribute which is set to nothing.

7.2 The initial state

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

a) Create an empty set of variable bindings, and set ⟦global-variable-bindings⟧ to the set of variable bindings.

b) Create built-in classes and modules as described in §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 the class `Object` and push the list onto ⟦class-module-list⟧.

f) Push the private visibility onto ⟦default-visibility⟧.

g) Push block-not-given onto ⟦block⟧.

8 Lexical structure

When several prefixes of the input under parsing process have matching productions, the production that matches the longest prefix is selected.

8.1 Source text

Syntax

---

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

A program is represented as a sequence of characters. A conforming processor shall accept any conforming program which consists of characters in ISO/IEC 646, encoded with the octet values as specified in ISO/IEC 646. The support for any other character sets and encodings is implementation dependent.

Terminal symbols are sequences of those characters in ISO/IEC 646. Control characters in ISO/IEC 646 are represented by hexadecimal notation.

EXAMPLE “0x0a” represents a line feed character.

8.2 Line terminators

Syntax

---

line-terminator ::
    0x0d? 0x0a

A *line-terminator* is ignored when it is used to separate *tokens*. For this reason, except in §8.4 and §8.5, *line-terminators* are omitted from productions. However, in some cases, the presence or absence of a *line-terminator* changes the meaning of a program.

A location of program text where a *line-terminator* shall occur is indicated by the notation "[*line-terminator* here]". A location of program text where a *line-terminator* shall not occur is indicated by the notation "[no *line-terminator* here]"; however, a conforming processor may ignore the notation where the ignorance does not introduce ambiguity.

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

---

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

The source

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

is therefore separated to two statements `x = 1 + 2` and `puts x` by a line-terminator.

The source

```
x =
  1 + 2
```

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

```
x
  = 1 + 2
```

is not a valid Ruby program because a *line-terminator* shall not occur before = in a *single-variable-assignment-expression*, and `= 1 + 2` is not a valid *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 §11.3.1.1.1):

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

8.3 Whitespace

Syntax

```
whitespace ::
        0x09 | 0x0b | 0x0c | 0x0d | 0x20 | \ 0x0d? 0x0a
```

whitespace is ignored when it is used to separate *tokens*. For this reason, except in §8.4 and §8.5, *whitespace* is omitted from productions. However, in some cases, the presence or absence of *whitespace* changes the meaning of a program.

A location of program text where *whitespace* shall occur is indicated by the notation "[*whitespace* here]". A location of program text where *whitespace* shall not occur is indicated by the notation "[no *whitespace* here]". A *line-terminator* shall not occur in the location where *whitespace* shall not occur. Therefore, this notation also indicates that a *line-terminator* shall not occur.

8.4 Comments

Syntax

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

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

comment-content ::
        line-content

line-content ::
        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*

line ::
  comment-content *line-terminator*

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

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

Any characters that are considered as *line-terminator*s are not allowed within a *line-content*.

A *comment* is either a *single-line-comment* or a *multi-line-comment*. A *comment* is considered to be *whitespace*.

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* on a *multi-line-comment-end-line*, if any, is considered to be part of the comment.

8.5 Tokens

Syntax

---

token ::
  reserved-word
 | *identifier*
 | *punctuator*
 | *operator*
 | *literal*

8.5.1 Reserved words

Syntax

---

reserved-word ::
  `__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`

Reserved words are case-sensitive.

8.5.2 Identifiers

Syntax

---

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

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`

An *identifier* is a sequence of *identifier-character*s optionally prefixed by one of "`$`", "`@@`", or "`@`", and optionally postfixed by one of "`?`", "`!`", or "`=`".

A *global-variable-identifier* begins with "`$`". A *class-variable-identifier* starts with "`@@`". An *instance-variable-identifier* begins with "`@`". A *constant-identifier* begins with an *uppercase-character*.

A *local-variable-identifier* begins with a *lowercase-character* or "`_`". A *method-identifier* is a *constant-identifier* or a *local-variable-identifier* optionally followed by one of "`?`", "`!`", or "`=`".

8.5.3 Punctuators

Syntax

---

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

8.5.4 Operators

Syntax

---

operator ::
    operator-method-name

| *assignment-operator*

operator-method-name ::
`^` | `&` | `|` | `<=>` | `==` | `===` | `!~` | `=~` | `>` | `>=` | `<` | `<=` | `<<` | `>>` | `+`
| `-` | `*` | `/` | `%` | `**` | `~` | `+@` | `-@` | `[]` | `[]=` | `` ` ``

assignment-operator ::
assignment-operator-name `=`

assignment-operator-name ::
`+` | `-` | `*` | `**` | `/` | `^` | `%` | `<<` | `>>` | `&` | `&&` | `||` | `|`

8.5.5 Literals

---

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

8.5.5.1 Numeric literals

Syntax

---

numeric-literal ::
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 ::
digit-decimal-integer-literal
| *prefixed-decimal-integer-literal*

digit-decimal-integer-literal ::
    `0`
  | *decimal-digit-without-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 ::
    decimal-float-literal
  | *exponent-float-literal*

decimal-float-literal ::
    digit-decimal-integer-literal `.` *digit-decimal-part*

exponent-float-literal ::
    base-part *exponent-part*

base-part ::
    decimal-float-literal
  | *digit-decimal-integer-literal*

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

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

decimal-digit-without-zero ::
    `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

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

binary-digit ::
    `0` | `1`

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

Semantics

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

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 alternatives on the right-hand side.

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 alternatives on the right-hand side.

A *signed-number* which begins with “`+`” evaluates to an instance represented by 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 instance represented by the *unsigned-number*.

The value of an *integer-literal*, a *decimal-integer-literal*, a *float-literal*, or a *base-part* is the value of one of the alternatives on the right-hand side.

The value of a *digit-decimal-integer-literal* is either 0 or the value of a sequence of characters, which consist of a *decimal-digit-without-zero* followed by 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 value of a *decimal-float-literal* is the value of the *digit-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 an *exponent-float-literal* is the value of the *base-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*.

There is no limitation on the maximum magnitude for the value of an *integer-literal*. The precision of the value of a *float-literal* is implementation defined; however, if the underlying platform of a conforming processor supports IEC 60559:1989, the representation of an instance of the class `Float` should be the 64-bit double format as specified in §3.2.2 of IEC 60559:1989. The value of a float-literal is rounded to fit in the representation of an instance of the class Float in an implementation defined way.

8.5.5.2 String literals

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`.

8.5.5.2.1 Single quoted strings

Syntax

---

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

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

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

single-escape-character-sequence ::
    \ *single-escaped-character*

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

single-escaped-character ::
    ' | ****

non-escaped-single-quoted-string-character ::
    source-character **but not** *single-escaped-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 program text literally, except for *single-escape-character-sequence*s. The sequence "\\\\" represents "\\". The sequence "\\'" represents "'".

8.5.5.2.2 Double quoted strings

Syntax

---

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

double-quoted-string-character ::
    source-character **but not** (**"** | ****)
    | *double-escape-sequence*
    | *interpolated-character-sequence*

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

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

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

line-terminator-escape-sequence ::
    **** *line-terminator*

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

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

octal-escape-sequence ::
`\` *octal-digit* (*octal-digit* *octal-digit*?)?

hex-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* `}`

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 program text.

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

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 *hex-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 a *non-escaped-double-quoted-string-character*.

A *line-terminator-escape-sequence* is used to break the content of a string into separate lines in 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-*

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

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 *interpolated-character-sequences* within a *string-literal* are evaluated in the order in which they occur in program text.

The value of a *string-literal* which contains *interpolated-character-sequences* 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 §11.4.3.3). Let V be the resulting value.

b) If it is of the form `#` *class-variable-identifier*, evaluate the *class-variable-identifier* (see §11.4.3.4). Let V be the resulting value.

c) If it is of the form `#` *instance-variable-identifier*, evaluate the *instance-variable-identifier* (see §11.4.3.5). 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`, V is the value of *interpolated-character-sequence*.

f) Otherwise, invoke the method `to_s` on V with an empty list of arguments. Let S be the resulting value.

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

h) Otherwise, the value of *interpolated-character-sequence* is an instance of the class `String`, the content of which is implementation defined.

8.5.5.2.3 Quoted non-expanded literal strings

Syntax

---

quoted-non-expanded-literal-string ::
  `%q` *literal-beginning-delimiter* *non-expanded-literal-string** *literal-ending-delimiter*

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

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

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*

The *literal-beginning-delimiter* of a *non-expanded-delimited-string* shall be the same character as the *literal-beginning-delimiter* of the *quoted-non-expanded-literal-string*.

A *literal-ending-delimiter* shall be the same character as the corresponding *literal-beginning-delimiter*, except when the *literal-beginning-delimiter* is one of the characters on the left in Table 2. In that case, the *literal-ending-delimiter* is the corresponding character on the right in Table 2.

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

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

The production *non-expanded-delimited-string* applies only when the *literal-beginning-delimiter* is one of the characters of *matching-literal-beginning-delimiter*.

Semantics

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

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*.

8.5.5.2.4 Quoted expanded literal strings

Syntax

---

quoted-expanded-literal-string ::
  `%` `Q`? *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
  | *double-escape-sequence*
  | *interpolated-character-sequence*

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

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

alpha-numeric-character-or-separator ::
  whitespace
  | *line-terminator*
  | *uppercase-character*
  | *lowercase-character*
  | *decimal-digit*

literal-ending-delimiter ::
  [depending on the *literal-beginning-delimiter*]

matching-literal-beginning-delimiter ::
  `(` | `{` | `<` | `[`

The *literal-beginning-delimiter* of an *expanded-delimited-string* shall be the same character as the *literal-beginning-delimiter* of the *quoted-expanded-literal-string*.

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in §8.5.5.2.3.

The production *expanded-delimited-string* applies only when the *literal-beginning-delimiter* is one of the characters of *matching-literal-beginning-delimiter*.

Semantics

A *expanded-literal-string* represents the content of a 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 program text. A *double-escape-sequence* and an *interpolated-character-sequence* represent characters as described in §8.5.5.2.2.

8.5.5.2.5 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 ::
    line **but not** *heredoc-end-line*

Semantics

A *here-document* is represented by several lines of program text, and evaluates to a direct instance of the class `String` or the value of the invocation of the method ` .

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 object to which *here-document* evaluates is either a direct instance S of the class `String` whose content is represented by the *heredoc-body* or the value of the invocation of the method ` with S as the only argument.

The form of the *heredoc-delimiter-specifier* determines both the form of the *heredoc-end-line* and the way in which the *here-document* is processed, as described below.

Syntax

---

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 **but not** '

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

double-quoted-delimiter-identifier ::
    source-character **but not** "

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

command-quoted-delimiter-identifier ::
    source-character **but not** `

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*

Semantics

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*.

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

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

   - If *heredoc-delimiter* is of the form *single-quoted-delimiter*, the *heredoc-body* is treated as a sequence of *source-character*s as it occurs in program text literally.

- If *heredoc-delimiter* is in any of the forms *non-quoted-delimiter*, *double-quoted-delimiter*, or *command-quoted-delimiter*, the *heredoc-body* is treated as a sequence of *double-quoted-string-character*s as described in §8.5.5.2.2.

Let S be that instance of the class `String`.

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 whose only element is S. Let V be the resulting value of the method invocation.

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

8.5.5.2.6 External command execution

Syntax

---

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

backquoted-external-command-execution ::
    ` *double-quoted-string-character\** `

quoted-external-command-execution ::
    %x *literal-beginning-delimiter expanded-literal-string* literal-ending-delimiter*

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in §8.5.5.2.

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 of the class `String` S by replacing the two "`" with """ and evaluating the resulting *double-quoted-string* as described in §8.5.5.2.2.

b) If the *external-command-execution* is of the form *quoted-external-command-execution*, construct a direct instance of the class `String` S by replacing "%x" with "%Q" and evaluating the resulting *quoted-expanded-literal-string* as described in §8.5.5.2.4.

c) Invoke the method "`" on the current self with a list of arguments whose only element is S.

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

8.5.5.3 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-item-character
| *non-expanded-array-escape-sequence*

non-escaped-array-item-character ::
non-escaped-array-character
| *matching-literal-delimiter*

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

matching-literal-delimiter ::
`(` | `{` | `<` | `[` | `)` | `}` | `>` | `]`

non-expanded-array-escape-sequence ::
non-expanded-literal-escape-sequence **but not** *escaped-quoted-array-item-separator*
| *escaped-quoted-array-item-separator*

escaped-quoted-array-item-separator ::
`\` *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
  | *expanded-array-escape-sequence*
  | *interpolated-character-sequence*

expanded-array-escape-sequence ::
    double-escape-sequence **but not** *escaped-quoted-array-item-separator*
  | *escaped-quoted-array-item-separator*

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in §8.5.5.2.

When the *literal-beginning-delimiter* is one of the *matching-literal-beginning-delimiter*, the *quoted-non-expanded-array-constructor* and the *quoted-expanded-array-constructor* is determined as follows.

Let N be 0. For each character C which appears after "`%w`" or "`%W`", take the following steps.

a) If C is a *literal-beginning-delimiter* which is not prefixed by a "\", increment N by 1.

b) If C is a *literal-ending-delimiter* which is not prefixed by a "\", decrement N by 1.

c) If N is 0 and C is the *literal-ending-delimiter*, terminate these steps.

The *literal-ending-delimiter* in Step c is the *literal-ending-delimiter* of the *quoted-non-expanded-array-constructor* or the *quoted-expanded-array-constructor*.

Semantics

An *array-literal* evaluates to a direct instance of the class `Array`.

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

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

b) If *non-expanded-array-item-list* occurs, for each *non-expanded-array-item* of the *non-expanded-*

array-item-list, take the following steps:

1) Create a direct instance of the class `String` S, 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 as described in §8.5.5.2.3, except in the case of an *escaped-quoted-array-item-separator*. An *escaped-quoted-array-item-separator* represents a *quoted-array-item-separator*.

2) Append S to A.

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

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

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

b) If *expanded-array-item-list* occurs, process each *expanded-array-item* of the *expanded-array-item-list* as follows:

   1) Create a direct instance of the class `String` S, 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 as described in §8.5.5.2.2, except in the case of an *escaped-quoted-array-item-separator*. An *escaped-quoted-array-item-separator* represents a *quoted-array-item-separator*. An *interpolated-character-sequence* represents a sequence of characters as described in §8.5.5.2.2.

   2) Append S to A.

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

8.5.5.4 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** (`/` | `\`)
    | `\ \`

| *line-terminator-escape-sequence*
| *interpolated-character-sequence*

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

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

The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in §8.5.5.2.3.

If a *regular-expression-literal* of the form `/` *regular-expression-body* `/` *regular-expression-option** is the first argument (see §11.2.1), the first character of the *regular-expression-body* shall not be *whitespace*.

Semantics

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

The pattern of an instance of the class `Regexp` resulting from a *regular-expression-literal* is the string which *regular-expression-character*s or *expanded-literal-string*s represent. If the string cannot be derived from the *pattern* (see §15.2.15.3), the evaluation of the program shall be terminated and a syntax error shall be reported.

A *regular-expression-character* other than the sequence `\ \`, a *line-terminator-escape-sequence*, or *interpolated-character-sequence* represents themselves. A *expanded-literal-string* other than a *line-terminator-escape-sequence* or *interpolated-character-sequence* represents themselves.

The sequence `\ \` of *regular-expression-character* represents a single 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 §8.5.5.2.2, and represents a string which is the content of the resulting an instance of the class `String`.

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

8.5.5.5 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 ::
    method-identifier
  | *operator-method-name*
  | *reserved-word*
  | *instance-variable-identifier*
  | *global-variable-identifier*
  | *class-variable-identifier*

---

*single-quoted-string*s, *double-quoted-string*s, and *non-expanded-literal-string*s shall not contain any sequences which represent the character 0x00.

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 §8.5.5.2.3.

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 §8.5.5.2.1), *double-quoted-string*(see §8.5.5.2.2), or *non-expanded-literal-string*(see §8.5.5.2.3).

9 Scope of variables

A ***scope*** is a region of a program text with which a set of bindings of variables is associated.

9.1 Local variables

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

9.1.1 Scopes of local variables

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)
- *eigenclass-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 §11.2.2)

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.

When a *local-variable-identifier* which is a reference to a local variable occurs (see §9.1.2), 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:

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

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

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

- Otherwise, a binding is considered not resolved.

9.1.2 References to local variables

An occurrence of a *local-variable-identifier* can be a reference to a local variable or a method invocation. In order to determine whether the occurrence of a *local-variable-identifier* is a reference to a local variable or a method invocation, before the evaluation of a local variable scope, the scope is scanned sequentially for *local-variable-identifier*s.

For each occurrence of a *local-variable-identifier* I, take the following steps:

a) If I occurs in one of the forms below, I is a reference to a local variable.

- *mandatory-parameter*
- *optional-parameter-name*
- *array-parameter-name*
- *block-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*

b) If I occurs in one of the forms below:

- *variable* of *singleton*
- *variable* of *primary-expression*

and the following condition holds, I is a reference to a local variable.

- Let P be the point where I occurs and let S be the innermost local variable scope which encloses P and which does not correspond to a *block*. Let R be the region of a program between the beginning of S and P.

  The same identifier as I occurs as a reference to a local variable in R.

c) Otherwise, I is a method invocation.

9.2 Global variables

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

10 Program structure

10.1 Program

Syntax

---

program ::
    compound-statement

Semantics

A *program* is evaluated as follows:

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

b) Evaluate the *compound-statement*.

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

d) Restore the execution context by removing the element from the top of ⟦local-variable-bindings⟧ , even when an exception is raised and not handled during Step b.

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* does not occur, the value of the *compound-statement* is `nil`.

b) If the *statement-list* occurs, evaluate each *statement* of the *statement-list* in the order it appears in the program text.

c) If one of the *statement*s of the *statement-list* is terminated by a *jump-expression*, terminate the evaluation of the *statement-list* immediately. None of the following *statement*s of the *statement-list* is evaluated. In this case, the value of the *compound-statement* is undefined.

d) If none of the *statement*s of the *statement-list* is terminated by a *jump-expression*, the value of the *compound-statement* is the value of the last *statement* of the *statement-list*.

11 Expressions

Syntax

expression ::
    keyword-logical-expression

Semantics

See §11.1 for *keyword-logical-expression*.

11.1 Logical expressions

Syntax

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

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

logical-NOT-expression ::=
    logical-NOT-with-method-invocation-without-parentheses
    | *logical-NOT-with-unary-expression*

logical-NOT-with-method-invocation-without-parentheses ::
    `!` *method-invocation-without-parentheses*

logical-NOT-with-unary-expression ::
    `!` *unary-expression*

keyword-AND-expression ::
    expression `and` *keyword-NOT-expression*

keyword-OR-expression ::
    expression `or` *keyword-NOT-expression*

logical-OR-expression ::
    logical-AND-expression
    | *logical-OR-expression* `||` *logical-AND-expression*

logical-AND-expression ::
equality-expression
| *logical-AND-expression* `&&` *equality-expression*

Semantics

A *logical-NOT-expression* or a *keyword-NOT-expression* of the form `not` *keyword-NOT-expression* is evaluated as follows:

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

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

c) If X is a true value, the value of the *keyword-NOT-expression* or the *logical-NOT-expression* is `false`.

d) Otherwise, the value of the *keyword-NOT-expression* or the *logical-NOT-expression* is `true`.

Instead of the above process, a conforming processor may evaluate a *logical-NOT-expression* as follows:

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

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

In this case, the processor shall:

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

A *logical-AND-expression* of the form *logical-AND-expression* `&&` *equality-expression* or a *keyword-AND-expression* is evaluated as follows:

a) Evaluate the *expression* or the *logical-AND-expression*. Let X be the resulting value.

b) If X is a true value, evaluate the *keyword-NOT-expression* or *equality-expression*. Let Y be the resulting value. The value of the *keyword-AND-expression* or the *logical-AND-expression* is Y.

c) Otherwise, the value of the *keyword-AND-expression* or the *logical-AND-expression* is X.

A *keyword-OR-expression* or a *logical-OR-expression* of the form *logical-OR-expression* `||` *logical-AND-expression* is evaluated as follows:

a) Evaluate the *expression* or the *logical-OR-expression*. Let X be the resulting value.

b) If X is a false value, evaluate the *keyword-NOT-expression* or the *logical-AND-expression*. Let Y be the resulting value. The value of the *keyword-OR-expression* or *logical-OR-expression* is Y.

c) Otherwise, the value of the *keyword-OR-expression* or *logical-OR-expression* is X.

11.2 Method invocation expressions

Syntax

---

primary-method-invocation ::
    super-with-optional-argument
    | *indexing-method-invocation*
    | *method-only-identifier*
    | *method-identifier* ([no *whitespace* here] *argument-with-parentheses*)? *block*?
    | *primary-expression* [no *line-terminator* here]
        `.` *method-name* ([no *whitespace* here] *argument-with-parentheses*)? *block*?
    | *primary-expression* [no *line-terminator* here]
        `::` *method-name* [no *whitespace* here] *argument-with-parentheses* *block*?
    | *primary-expression* [no *line-terminator* here] `::` *method-name-without-constant*
  block?

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

optional-whitespace ::
    [*whitespace* here]

method-name-without-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*
    | *return-with-argument*
    | *break-with-argument*
    | *next-with-argument*

command ::
    super-with-argument
    | *yield-with-argument*
    | *method-identifier* *argument*
    | *primary-expression* [no *line-terminator* here] (`.` | `::`) *method-name* *argument*

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

chained-method-invocation ::
  (`.` | `::`) *method-name*
  | (`.` | `::`) *method-name* [no *whitespace* here]
    [lookahead ∉ { `{` }] *argument-with-parentheses*

command-with-do-block ::
  super-with-argument-and-do-block
  | *method-identifier* *argument* *do-block*
  | *primary-expression* [no *line-terminator* here] (`.` | `::`) *method-name* *argument* *do-block*

The *primary-expression* of a *primary-method-invocation*, *command*, and *indexing-method-invocation* shall not be a *jump-expression*.

If the *argument-with-parentheses* of a *primary-method-invocation* occurs, and the *block-argument* of the *argument* of the *argument-with-parentheses* occurs, the *block* of the *primary-method-invocation* shall not occur.

If the *argument* of a *command-with-do-block* occurs, and the *block-argument* of the *argument-in-parentheses* of the *argument* (see §11.2.1) occurs, the *do-block* of the *command-with-do-block* shall not occur.

The *optional-whitespace* of an *indexing-method-invocation* shall not occur if its *primary-expression* is any of the following construct:

- A *primary-method-invocation* of the form *method-only-identifier*, *method-identifier*, *primary-expression* `.` *method-name*, or *primary-expression* `::` *method-name-without-constant*
- A *method-invocation-without-parentheses* of the form *chained-command-with-do-block* which satisfies all of the following conditions:
  a) Let M be the *chained-command-with-do-block*. One or more *chained-method-invocation* of M occurs.
  b) Let I be the last *chained-method-invocation* of M, in the order they appear in program text. I is of the form (`.`|`::`) *method-name*.

Semantics

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

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

b) • 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.

- If the *method-identifier* of the *primary-method-invocation* occurs:
  1) Let O be the current self and let M be the *method-identifier*.
  2) If the *argument-with-parentheses* occurs, construct a list of arguments and a block from the *argument-with-parentheses* as described in §11.2.1. Let L be the resulting list. Let B be the resulting block, if any.

     If the *argument-with-parentheses* does not occur, create an empty list of arguments L.
  3) If the *block* occurs, let B be the *block*.
- If the `.` of the *primary-method-invocation* occurs:
  1) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name*.
  2) If the *argument-with-parentheses* occurs, construct a list of arguments and a block from the *argument-with-parentheses* as described in §11.2.1. Let L be the resulting list. Let B be the resulting block, if any.

     If the *argument-with-parentheses* does not occur, create an empty list of arguments L.
  3) If the *block* occurs, let B be the *block*.
- If the `::` and *method-name* of the *primary-method-invocation* occur:
  1) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name*.
  2) Construct a list of arguments and a block from the *argument-with-parentheses* as described in §11.2.1. Let L be the resulting list. Let B be the resulting block, if any.
  3) If the *block* occurs, let B be the *block*.
- If the `::` and *method-name-without-constant* of the *primary-method-invocation* occur:
  1) Evaluate the *primary-expression* and let O be the resulting value. Let M be the *method-name-without-constant*.
  2) Create an empty list of arguments L.
  3) If the *block* occurs, let B be the *block*.

c) 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 resulting value is the value of the *primary-method-invocation*.

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

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

b) If the *indexing-argument-list* occurs, construct a list of arguments from the *indexing-argument-list* as described in §11.2.1. Let L be the resulting list.

c) If the *indexing-argument-list* does not occur, Create an empty list of arguments L.

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

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

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

- If the *method-invocation-without-parentheses* is a *return-with-argument*, *break-with-argument* or *next-with-argument*, evaluate it (see §11.4.1.3).

- If the *chained-command-with-do-block* of the *method-invocation-without-parentheses* occurs:

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

  b) If the *method-name* and the *argument* of the *method-invocation-without-parentheses* occur:

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

    2) Construct a list of arguments from the *argument* as described in §11.2.1 and let L be the resulting list. If the *block-argument* of the *argument-in-parentheses* of the *argument* occurs, let B be the *block* to which the *block-argument* corresponds.

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

    4) Replace V with the resulting value.

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

A *command* is evaluated as follows:

a) If the *command* is a *super-with-argument* or a *yield-with-argument*, evaluate it.

b) Otherwise:

  1) If the *method-identifier* of the *command* occurs:

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

    ii) Construct a list of arguments from the *argument* as described in §11.2.1 and let L be the resulting list.

    If the *block-argument* of the *argument-in-parentheses* of the *argument* occurs, let B be the *block* to which the *block-argument* corresponds.

2) If the *primary-expression*, *method-name*, and the *argument* of the *command* occurs:

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* as described in §11.2.1 and let L be the resulting list.

If the *block-argument* of the *argument-in-parentheses* of the *argument* occurs, let B be the *block* to which the *block-argument* corresponds.

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

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

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

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

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

2) If the *argument-with-parentheses* occurs, construct a list of arguments and a block from the *argument-with-parentheses* as described in §11.2.1 and let L be the resulting list. Let B be the resulting block, if any.

If the *argument-with-parentheses* does not occur, create an empty list of arguments L.

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

4) Replace V with the resulting value.

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

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

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

a) If the *method-identifier* of the *command* occurs, let O be the current self and let M be the *method-name*.

If the *method-identifier* of the *command* does not occur, evaluate the *primary-expression*, let O be the resulting value and let M be the *method-name*.

b) Construct a list of arguments from the *arguments* of the *command-with-do-block* and let L be the resulting list.

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

11.2.1 Method arguments

Syntax

---

indexing-argument-list ::
    command
  | *operator-expression-list* `,`?
  | *operator-expression-list* `,` *splatting-argument*
  | *association-list* `,`?
  | *splatting-argument*

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

operator-expression-list ::
    operator-expression (`,` *operator-expression*)*

argument-with-parentheses ::
    `(` `)`
  | `(` *argument-in-parentheses* `)`
  | `(` *operator-expression-list* `,` *chained-command-with-do-block* `)`
  | `(` *chained-command-with-do-block* `)`

argument ::
    [no *line-terminator* here] [lookahead $\notin$ { `{` }] *optional-whitespace*?
      argument-in-parentheses

argument-in-parentheses ::
    command
  | (*operator-expression-list* | *association-list*)
      (`,` *splatting-argument*)? (`,` *block-argument*)?
  | *operator-expression-list* `,` *association-list*
      (`,` *splatting-argument*)? (`,` *block-argument*)?
  | *splatting-argument* (`,` *block-argument*)?
  | *block-argument*

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

The *operator-expression* of a *splatting-argument*, *operator-expression-list*, and *block-argument* shall not be a *jump-expression*.

If the *operator-expression-list* of an *argument-in-parentheses* occurs, the first *operator-expression* of the *operator-expression-list* is called the ***first argument***.

If a *splatting-argument* is the first argument, *whitespace*s shall not occur between its `*` and *operator-expression*. If a *block-argument* is the first argument, *whitespace*s shall not occur between its `&` and *operator-expression*.

If the first argument of an *argument* is other than the following constructs, the *optional-whitespace* shall occur.

- A *variable-reference* of the form *global-variable-identifier*, *class-variable-identifier* or *instance-variable-identifier* (see §11.4.3).
- A *single-quoted-string* or *double-quoted-string* (see §8.5.5.2).
- A *symbol-literal*, or a *dynamic-symbol* of the form `:` [no *whitespace* here] *single-quoted-string* or `:` [no *whitespace* here] *double-quoted-string* (see §8.5.5.5).
- An *external-command-execution* of the form *backquoted-external-command-execution* (see §8.5.5.2.6).
- A *scoped-constant-reference* whose *primary-expression* occurs and the *primary-expression* is any of these constructs.
- A *primary-method-invocation* whose *primary-expression* occurs and the *primary-expression* is any of these constructs.

Semantics

The list of arguments used for method invocation is constructed from *indexing-argument-list*, *splatting-argument*, *argument-with-parentheses*, or *argument*.

An *indexing-argument-list* is processed as follows:

a) Create an empty list of arguments L.

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

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

A *splatting-argument* is processed as follows:

a) Create an empty list of arguments L.

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

c) If V is not an instance of the class `Array`, the behavior is implementation dependent.

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

An *argument-with-parentheses* is processed as follows:

a) Create an empty list of arguments L.

b) If the *argument-in-parentheses* occurs, construct a list of arguments from it and concatenate the resulting list to L. If *block-argument* of *argument-in-parentheses* occurs, the *block* to which the *block-argument* corresponds is the block which is passed to the method invocation with L.

c) If the *operator-expression-list* occurs, for each *operator-expression* of the *operator-expression-list*, in the order they appears in the program text, take the following steps:

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

2) Append V to L.

d) If the *chained-command-with-do-block* occurs, evaluate it. Append the resulting value to L.

An *argument* is processed as follows:

a) Evaluate the *argument-in-parentheses*.

b) Let L be the resulting list.

An *argument-in-parentheses* is processed as follows:

a) Create an empty list of arguments L.

b) If the *command* occurs, evaluate it. Append the resulting value to L.

c) If the *operator-expression-list* occurs, for each *operator-expression* of the *operator-expression-list*, in the order they appears in the program text, take the following steps:

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

2) Append V to L.

d) If the *association-list* occurs, evaluate it. Append the resulting value to L.

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

f) If the *block-argument* occurs, construct a *block* which is passed to a method invocation as described below.

A block which is passed to a method invocation is constructed from the *block-argument* as follows:

a) Evaluate the *operator-expression*. Let P be the resulting value.

b) If P is not an instance of the class `Proc`, the behavior is implementation dependent.

c) Otherwise, the resulting block is the block which P represents.

11.2.2 Blocks

Syntax

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

brace-block ::
  `{` *block-formal-argument*? *block-body* `}`

do-block ::
  `do` *block-formal-argument*? *block-body* `end`

block-formal-argument ::
  `| |`
  | `||`
  | `|` *block-formal-argument-list* `|`

block-formal-argument-list ::
  left-hand-side
  | *multiple-left-hand-side*

block-body ::
  compound-statement

Whether the *left-hand-side* (see §11.3.1.3) in the *block-formal-argument-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*

Whether the *grouped-left-hand-side* in the *block-formal-argument-list* is allowed to be the following form is implementation defined.

- `(` (*multiple-left-hand-side-item* `,`)+ `)`;

Semantics

A *block* is a sequence of statements or expressions passed to a method invocation.

A *block* can be called either by a *yield-expression* (see §11.2.4) or by invoking the method `call` on an instance of the class `Proc` which is created by an invocation of the method `Proc.new` to which the block is passed (see §15.2.17.3.3).

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 under 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 affected execution context.

Both the *do-block* and the *brace-block* of the *block* are 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-formal-argument-list* in the *do-block* or the *brace-block* occurs:

- If the *block-formal-argument-list* is of the form *left-hand-side* or *grouped-left-hand-side*:
  - — If the length of L is 0, let X be `nil`.
  - — If the length of L is 1, let X be the only element of L.
  - — If the length of L is larger than 1, the result of this step is implementation dependent.
  - — If the *block-formal-argument-list* is of the form *left-hand-side*, evaluate a *single-variable-assignment-expression* E (see §11.3.1.1.1), where the *variable* of E is the *left-hand-side* and the value of the *operator-expression* of E is X.
  - — If the *block-formal-argument-list* is of the form *grouped-left-hand-side*, evaluate a *many-to-many-assignment-expression* E (see §11.3.1.3), 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.
- If the *block-formal-argument-list* is of the form *multiple-left-hand-side* and the *multiple-left-hand-side* is not a *grouped-left-hand-side*:
  1) 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 implementation dependent.
     ii) Create a list of arguments Y which contains the elements of L, preserving their order.

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

3) Evaluate the *many-to-many-assignment-statement* E as described in §11.3.1.3, where the *multiple-left-hand-side* of E is the *block-formal-argument-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*:

- is terminated by a *break-expression*:

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

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

    2) If the *jump-argument* of the *break-expression* occurs, 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.

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

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

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

- is terminated by a *next-expression*:

  — If the *jump-argument* of the *next-expression* occurs, let V be the value of the *jump-argument*.

  — Otherwise, let V be `nil`.

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

- 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, even when an exception is raised and not handled in Step d or e.

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

11.2.3 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 *whitespace* here] *argument-with-parentheses*)? *block*?

super-with-argument ::
  `super` *argument*

super-with-argument-and-do-block ::
  `super` *argument* *do-block*

The *block-argument* of the *argument-in-parentheses* of the *argument* (see §11.2.1) of a *super-with-argument-and-do-block* shall not occur.

Semantics

A *super-expression* is evaluated as follows:

a) If the current self is pushed by an *eigenclass-definition* (see §13.4.2), or an invocation of one of the following methods, the behavior is implementation dependent:

- the method `class_eval` of the class `Module` (see §15.2.2.3.15)
- the method `module_eval` of the class `Module` (see §15.2.2.3.35)
- the method `instance_eval` of the class `Kernel` (see §15.3.1.2.18)

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

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

  1) 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.

  2) If the *mandatory-parameter-list* occurs 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.

3) If the *optional-parameter-list* occurs 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.

4) If the *array-parameter* occurs 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.

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

— If the *argument-with-parentheses* occurs, construct a list of arguments and a block as described in §11.2.1. Let A be the resulting list. Let B be the resulting block, if any.

— If the *block* occurs, Let B be the *block*.

- If the *super-expression* is a *super-with-argument*, construct the list of arguments from the *argument* as described in §11.2.1. Let A be the resulting list. If *block-argument* of the *argument-in-parentheses* of *argument* occurs, let B be the *block* constructed from the *block-argument*.

- If the *super-expression* is a *super-with-argument-and-do-block*, construct a list of arguments from the *argument* as described in §11.2.1. 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) 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.

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

4) Otherwise, add a direct instance of the class `Symbol` with name N to the head of A, and invoke the method `method_missing` on the current self with A as arguments and B as the block. Then, terminate the evaluation of the *super-expression*. The value of the *super-expression* is the resulting value of the method invocation.

d) Take Step 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.

11.2.4 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 *whitespace* here] `(` *argument-in-parentheses* `)`

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

yield-with-argument ::
  `yield` *argument*

The *block-argument* of the *argument-in-parentheses* (see §11.2.1) of a *yield-with-parentheses-and-argument* shall not occur.

The *block-argument* of the *argument-in-parentheses* of the *argument* (see §11.2.1) of a *yield-with-argument* shall not occur.

Semantics

A *yield-expression* calls the block at the top of ⟦block⟧ .

A *yield-with-optional-argument* 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) If the *yield-with-optional-argument* is of the form *yield-with-parentheses-and-argument*, create a list of arguments from the *argument* as described in §11.2.1. Let L be the list.

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

d) Call B with L as described in §11.2.2.

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

A *yield-with-argument* 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) Create a list of arguments from the *argument* as described in §11.2.1. Let L be the list.

c) Call B with L as described in §11.2.2.

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

11.3 Operator expressions

Syntax

---

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

11.3.1 Assignments

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

Assignments create or update variable bindings, or invoke a method whose name ends with `=`.

Evaluation of each construct is described below.

11.3.1.1 Single assignments

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*

11.3.1.1.1 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*

scoped-constant-assignment ::=
scoped-constant-assignment-expression

| *scoped-constant-assignment-statement*

scoped-constant-assignment-expression ::
primary-expression [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 *whitespace* here] :: *constant-identifier*
[no *line-terminator* here] = *method-invocation-without-parentheses*
| :: *constant-identifier* [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) • If the *variable* is a *constant-identifier*:

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

   2) 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.

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

   • If the *variable* is a *global-variable-identifier*:

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

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

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

   • If the *variable* is a *class-variable-identifier*:

   1) Let C be the first class or module in the list at the top of ⟦class-module-list⟧ which is not an eigenclass.

      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*.

2) If 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, let *B* be one of those bindings. Which binding is selected is implementation defined.

Replace the value of *B* with *V*.

3) 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*.

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

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

  2) 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*.

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

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

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

  2) Search for a binding of a local variable with name *N* as described in §9.1.1.

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

  4) 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*.

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

a) If the *primary-expression* occurs, 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) Create a variable binding with name *N* and value *V* in the set of bindings of constants of *M*.

  4) 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`.

11.3.1.1.2 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] `[` *indexing-argument-list*? `]`
      [no *line-terminator* here] `=` *operator-expression*

single-indexing-assignment-statement ::
    primary-expression [no *line-terminator* 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 §11.2.1. 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.

11.3.1.1.3 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.

11.3.1.2 Abbreviated assignments

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*

11.3.1.2.1 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 §11.4.3). 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) Evaluate the *operator-expression* of the form L OP R, where the value of L is V and the value of R is W. Let X be the resulting value.

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 §11.3.1.1.1) where its *variable* is I and the value of the *operator-expression* is X.

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

11.3.1.2.2 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] `[` *indexing-argument-list*? `]`
      [no *line-terminator* here] *assignment-operator* *operator-expression*

abbreviated-indexing-assignment-statement ::
primary-expression [no *line-terminator* 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 §11.2.1. 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.

11.3.1.2.3 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* 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 *single-method-assignment* 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 X as the argument.

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

11.3.1.3 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] `[` *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* `,`)+ *multiple-left-hand-side-item* ?
| (*multiple-left-hand-side-item* `,`)+ *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 (`,` *splatting-right-hand-side*)?
| *splatting-right-hand-side*

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

Any of the *operator-expression*s in a *multiple-assignment-statement* or *splatting-right-hand-side* shall not be a *jump-expression*.

Semantics

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

a) Construct a list of values from the *multiple-right-hand-side* (see below). Let L be the resulting list.

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

c) 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`.

d) Evaluate a *single-variable-assignment-expression* (see §11.3.1.1.1) where its *variable* is the *left-hand-side* and the value of its *operator-expression* is A.

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

A list of values is constructed from a *multiple-right-hand-side* as follows:

a) If the *operator-expression-list* occurs, 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.

b) If the *operator-expression-list* does not occur, create an empty list of values *L1*.

c) If the *splatting-right-hand-side* occurs, construct a list of values from its *splatting-argument* as described in §11.2.1 and let *L2* be the resulting list.

d) If the *splatting-right-hand-side* does not occur, create an empty list of values *L2*.

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

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

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

b) If V is an instance of the class `Array`, let A be an implementation defined value.

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

d) If the *left-hand-side* of the *packing-left-hand-side* occurs, evaluate a *single-variable-assignment-expression* (see §11.3.1.1.1) where its *variable* is the *left-hand-side* and the value of the *operator-expression* is A.

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

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

a) If the *multiple-right-hand-side* occurs, construct a list of values from it (see above) and let R be the resulting list.

b) If the *multiple-right-hand-side* does not occur:

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

   2) If V is not an instance of the class `Array`, the behavior is implementation dependent.

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

c) Create an empty list of variables L.

   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.

   If the *packing-left-hand-side* of the *multiple-left-hand-side* occurs, append it to L.

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

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

- Let i be the index of L_i within L. Let N_R be the number of elements of R.
- If L_i is a *left-hand-side*:
  1) If i is larger than N_R, let V be `nil`.
  2) Otherwise, let V be the ith element of R.
  3) Evaluate the *single-variable-assignment* of the form L_i `=` V.
- If L_i is a *packing-left-hand-side* and its *left-hand-side* occurs:
  1) If i is larger than N_R, create an empty direct instance of the class `Array`. Let A be the instance.
  2) 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 store in R. let A be the instance.
  3) Evaluate a *single-variable-assignment-expression* (see §11.3.1.1.1) where its *variable* is the *left-hand-side* and the value of the *operator-expression* is A.
- If L_i is a *grouped-left-hand-side*:
  1) If i is larger than N_R, let V be `nil`.
  2) Otherwise, let V be the ith element of R.
  3) 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.

11.3.1.4 Assignments with `rescue` modifiers

Syntax

---

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

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 let V be the resulting value.

b) Evaluate a *single-variable-assignment-expression* (see §11.3.1.1.1) 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.

11.3.2 Unary operators

Syntax

---

unary-minus-expression ::
  power-expression$_1$
  | `-` *power-expression*$_2$

unary-expression ::
  primary-expression
  | *logical-NOT-with-unary-expression*
  | `~` *unary-expression*$_1$
  | `+` *unary-expression*$_2$

If a *unary-minus-expression* of the form `-` *power-expression*$_2$ is the first argument (see §11.2.1), *whitespaces* shall not occur between its `-` and *power-expression*$_2$.

If a *unary-expression* of the form `+` *unary-expression*$_2$ is the first argument (see §11.2.1), *whitespaces* shall not occur between its `+` and *unary-expression*$_2$.

Semantics

See §11.1 for *logical-NOT-with-unary-expression*.

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

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

b) 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.

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

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

b) 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) If the *unary-expression*$_2$ is a *numeric-literal* (see §8.5.5.1), instead of the above process, a conforming processor may evaluate the *unary-expression* to the value of the *numeric-literal*.

An *unary-expression* of the form `-` *power-expression*$_2$ is evaluated as follows:

a) Evaluate the *power-expression*$_2$. Let X be the resulting value.

b) 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.

11.3.2.1 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) If the *defined?-expression* is a *defined?-with-parentheses*, let E be the *expression*.

If the *defined?-expression* is a *defined?-without-parentheses*, let E be the *operator-expression*.

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 §11.4.3.1. 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 true value.

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

• If E is a *global-variable-identifier*:

— 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 true value.

— Otherwise, the value of the *defined?-expression* is `nil`.

• 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 true value.

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

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

  — 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 true value.

  — Otherwise, the value of the *defined?-expression* is `nil`.

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

  1) If the *local-variable-identifier* is a reference (see §9.1.2), the value of the *defined?-expression* is an implementation defined value, which shall be a true value.

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

     — 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 true value.

     — Otherwise, the value of the *defined?-expression* is `nil`.

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

11.3.3 Binary operators

Syntax

---

equality-expression ::
  relational-expression
  | *relational-expression* `<=>` *relational-expression*
  | *relational-expression* `==` *relational-expression*
  | *relational-expression* `===` *relational-expression*
  | *relational-expression* `!=` *relational-expression*
  | *relational-expression* `=~` *relational-expression*
  | *relational-expression* `!~` *relational-expression*

relational-expression ::
  bitwise-OR-expression
  | *relational-expression* `>` *bitwise-OR-expression*
  | *relational-expression* `>=` *bitwise-OR-expression*
  | *relational-expression* `<` *bitwise-OR-expression*
  | *relational-expression* `<=` *bitwise-OR-expression*

bitwise-OR-expression ::
  bitwise-AND-expression

```
    | bitwise-OR-expression | bitwise-AND-expression
    | bitwise-OR-expression ^ bitwise-AND-expression

bitwise-AND-expression ::
      bitwise-shift-expression
    | bitwise-AND-expression whitespace-before-operator? & bitwise-shift-expression

bitwise-shift-expression ::
      additive-expression
    | bitwise-shift-expression whitespace-before-operator? << additive-expression
    | bitwise-shift-expression >> additive-expression

additive-expression ::
      multiplicative-expression
    | additive-expression whitespace-before-operator? + multiplicative-expression
    | additive-expression whitespace-before-operator? - multiplicative-expression

multiplicative-expression ::
      unary-minus-expression
    | multiplicative-expression whitespace-before-operator? * unary-minus-expression
    | multiplicative-expression whitespace-before-operator? / unary-minus-expression
    | multiplicative-expression whitespace-before-operator? % unary-minus-expression

power-expression ::
      unary-expression
    | - ( numeric-literal ) ** power-expression
    | unary-expression ** power-expression

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

If a *whitespace-before-operator* occurs, *whitespace*s shall not occur between the operator after the *whitespace-before-operator* (i.e. `&`, `<<`, `+`, `-`, `*`, `/`, or `%`) and the nonterminal after that operator.

Semantics

An *operator-expression* of the form x `!=` y is evaluated as follows:

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

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

c) Invoke the method `==` on X with a list of arguments which contains only one value Y. If the resulting value is a true value, the value of the *operator-expression* is `false`. Otherwise, the value of the *operator-expression* is `true`.

An *operator-expression* of the form x `!~` y is evaluated as follows:

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

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

c) Invoke the method `=~` on X with a list of arguments which contains only one value Y. If the resulting value is a true value, the value of the *operator-expression* is `false`. Otherwise, the value of the *operator-expression* is `true`.

A conforming processor may include the operators `!=` and `!~` in *binary-operator*. In this case, *operator-expression*s of the form `x != y` or `x !~ y` are evaluated as described below.

An *operator-expression* of the form x *binary-operator* y is evaluated as follows:

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

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

c) Invoke the method whose name is the *binary-operator* on X with a list of arguments which contains only one value Y. The value of the *operator-expression* is the resulting value of the invocation.

11.4 Primary expressions

Syntax

---

primary-expression ::
    class-definition
  | *eigenclass-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*
  | *rescue-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*.

See §13.4.2 for *eigenclass-definition*.

See §13.1.2 for *module-definition*.

See §13.3.1 for *method-definition*.

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

See §11.2.4 for *yield-with-optional-argument*.

See §11.3.2.1 for *defined?-with-parentheses*.

See §11.2 for *primary-method-invocation*.

11.4.1 Control structures

11.4.1.1 Conditional expressions

11.4.1.1.1 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*

The *expression* of an *if-expression* or *elsif-clause* shall not be a *jump-expression*.

Semantics

The *if-expression* is evaluated as follows:

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

b) If V is a true value, 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 false value, 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 false value, 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 false value, 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 true value. Let T be this *elsif-clause*.

   2) If T exists, evaluate the *compound-expression* of its *then-clause*. The value of the *if-expression* is the resulting value. Other *elsif-clause*s 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`.

11.4.1.1.2 The `unless` expression

Syntax

---

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

The *expression* of an *unless-expression* shall not be a *jump-expression*.

Semantics

The *unless-expression* is evaluated as follows:

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

b) If V is a false value, 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 true value, and if there is no *else-clause*, then the value of the *unless-expression* is `nil`.

d) If V is a true value, 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.

11.4.1.1.3 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 (`,` *splatting-argument*)?
    | *splatting-argument*

The *expression* of a *case-expression-with-expression* shall not be a *jump-expression*.

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 this step is defined as follows:

- 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 true value.

- If the *case-expression* is a *case-expression-without-expression*, O is matching if and only if O is a true value.

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

1) If the *operator-expression-list* of the *when-argument* occurs:

   - For each of its *operator-expressions*, evaluate it and test if the resulting value is matching.

   - If a matching value is found, other *operator-expressions*, if any, are not evaluated.

2) If no matching value is found, and the *splatting-argument* occurs:

- Construct a list of values from it as described in §11.2.1. For each element of the resulting list, in the indexing order, test if it is matching.

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

3) 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.

c) 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.

d) 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.

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

11.4.1.1.4 Conditional operator

Syntax

---

conditional-operator-expression ::
  range-constructor
  | *range-constructor* `?` *operator-expression*$_1$ `:` *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 true value, 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.

11.4.1.2 Iteration expressions

Syntax

---

iteration-expression ::=
  while-expression
  | *until-expression*
  | *for-expression*
  | *while-modifier-statement*
  | *until-modifier-statement*

Each *iteration-expression* has a ***body***. The body of a *while-expression* or an *until-expression* is its *compound-statement*. The body of a *while-modifier-statement* or an *until-modifier-statement* is its *statement*.

See §12.4 for *while-modifier-statement*.

See §12.5 for *until-modifier-statement*.

11.4.1.2.1 The `while` expression

Syntax

while-expression ::
    `while` *expression* *do-clause* `end`

do-clause ::
    separator *compound-statement*
    | `do` *compound-statement*

The *expression* of a *while-expression* shall not be a *jump-expression*.

Semantics

A *while-expression* is evaluated as follows:

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

b) If V is a false value, terminate the evaluation of the *while-expression*. The value of the *while-expression* is `nil`.

c) Otherwise, evaluate the *compound-statement* of the *do-clause*. If this evaluation:

    1) is terminated by a *break-expression*, terminate the evaluation of the *while-expression*.

       If the *jump-argument* of the *break-expression* occurs, the value of the *while-expression* is the value of the *jump-argument*. Otherwise, the value of the *while-expression* is `nil`.

    2) is terminated by a *next-expression*, continue processing from Step a.

    3) is terminated by a *redo-expression*, continue processing from Step c.

    Otherwise, unless this evaluation is terminated by a *return-expression*, continue processing from Step a.

11.4.1.2.2 The `until` expression

Syntax

until-expression ::
    `until` *expression* *do-clause* `end`

The *expression* of an *until-expression* shall not be a *jump-expression*.

Semantics

An *until-expression* is evaluated as follows:

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

b) If V is a true value, terminate the evaluation of the *until-expression*. The value of the *until-expression* is `nil`.

c) Otherwise, evaluate the *compound-statement* of the *do-clause*. If this evaluation:

    1) is terminated by a *break-expression*, terminate the evaluation of the *until-expression*.

       If the *jump-argument* of the *break-expression* occurs, the value of the *until-expression* is the value of the *jump-argument*. Otherwise, the value of the *until-expression* is `nil`.

    2) is terminated by a *next-expression*, continue processing from Step a.

    3) is terminated by a *redo-expression*, continue processing from Step c.

    Otherwise, unless this evaluation is terminated by a *return-expression*, continue processing from Step a.

11.4.1.2.3 The `for` expression

Syntax

for-expression ::
    `for` *for-variable* `in` *expression* *do-clause* `end`

for-variable ::
    left-hand-side
    | *multiple-left-hand-side*

The *expression* of a *for-expression* shall not be a *jump-expression*.

Semantics

A *for-expression* is evaluated as follows:

a) Evaluate the *expression*. 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-formal-argument-list* ` | ` *block-body* `end`, where the value of the *primary-expression* is O, the *block-formal-argument-list* is the *for-variable*, the *block-body* is the *compound-statement* of the *do-clause*.

Evaluate E, but skip Step c of §11.2.2.

c) The value of the *for-expression* is the resulting value of the invocation.

11.4.1.3 Jump expressions

Syntax

---

jump-expression ::=
    return-expression
  | *break-expression*
  | *next-expression*
  | *redo-expression*
  | *retry-expression*

Semantics

Jump expressions 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*.

In this document, the ***current block*** or the ***current iteration-expression*** refers to either of the following:

- If the current method invocation exists, the current *block* or the current *iteration-expression* is the *block* or the *iteration-expression* whose evaluation started most recently among the *block*s or *iteration-expression*s which are being evaluated on during the evaluation of the current method invocation.

- Otherwise, the current *block* or the current *iteration-expression* is the *block* or the *iteration-expression* whose evaluation started most recently among the *block*s or *iteration-expression*s which are under evaluation.

11.4.1.3.1 The `return` expression

Syntax

---

return-expression ::=
    return-without-argument
  | *return-with-argument*

return-without-argument ::
    `return`

return-with-argument ::
    `return` *jump-argument*

jump-argument ::
    argument

The *block-argument* of the *argument-in-parentheses* of the *argument* (see §11.2.1) of a *jump-argument* shall not occur.

Semantics

A *return-expression* is evaluated as follows:

a) Let M be the *method-body* which corresponds to the current method invocation. If such an invocation does not exist, or has already terminated:

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

   2) If the *jump-argument* of the *return-expression* occurs, 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.

b) Evaluate the *jump-argument*, if any, as described below.

c) 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 §11.2.2).

d) Terminate the evaluation of M (see §13.3.3).

A *jump-argument* is evaluated as follows:

a) If the *jump-argument* is a *splatting-argument*:

   1) Construct a list of values from the *splatting-argument* as described in §11.2.1 and let L be the resulting list.

   2) If the length of L is 0 or 1, the value of the *jump-argument* is 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. The value of the *jump-argument* is the instance of the class `Array`.

b) Otherwise:

   1) Construct a list of values from the *argument* as described in §11.2.1 and let L be the resulting list.

2) If the length of L is 1, the value of the *jump-argument* is the only element of L.

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. The value of the *jump-argument* is the instance of the class `Array`.

11.4.1.3.2 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 §11.4.1.3.1.

b) Let E be the current *block* or the current *iteration-expression*. If such a *block* or an *iteration-expression* does not exist:

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

   2) If the *jump-argument* of the *break-expression* occurs, 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.

c) If E is a *block*, terminate the evaluation of the *block-body* of E (see §11.2.2).

d) If E is an *iteration-expression*, terminate the evaluation of the body of E (see §11.4.1.2).

11.4.1.3.3 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 §11.4.1.3.1.

b) Let E be the current *block* or the current *iteration-expression*. If such a *block* or an *iteration-expression* does not exist:

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

    2) If the *jump-argument* of the *next-expression* occurs, 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.

c) If E is a *block*, terminate the evaluation of the *block-body* of E (see §11.2.2).

d) If E is an *iteration-expression*, terminate the evaluation of the body of E (see §11.4.1.2).

11.4.1.3.4 The `redo` expression

Syntax

---

redo-expression ::
    `redo`

Semantics

A *redo-expression* is evaluated as follows:

a) Let E be the current *block* or the current *iteration-expression*. If such a *block* or an *iteration-expression* does not exist,

    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`.

b) If E is a *block*, terminate the evaluation of the *block-body* of E (see §11.2.2).

c) If E is an *iteration-expression*, terminate the evaluation of the body of E (see §11.4.1.2).

11.4.1.3.5 The `retry` expression

Syntax

---

retry-expression ::
    `retry`

Semantics

A *retry-expression* is evaluated as follows:

a) If the current method invocation exists, let M be the *method-body* which corresponds to the current method invocation. Otherwise, let M be the *program*.

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 implementation dependent.

c) Terminate the evaluation of the *compound-statement*$_2$ of E (see §11.4.1.4.1).

11.4.1.4 Exceptions

11.4.1.4.1 The `rescue` expression

Syntax

---

rescue-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*

The *operator-expression* of an *exception-class-list* shall not be a *jump-expression*.

Semantics

The value of a *rescue-expression* is the 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* occurs, evaluate the *else-clause* as described in §11.4.1.1.1.

   2) If the *ensure-clause* occurs, evaluate its *compound-statement*. The value of this evaluation is the value of the *ensure-clause*.

   If both the *else-clause* and the *ensure-clause* occur, the value of the *body-statement* is the value of the *ensure-clause*. If only one of these clauses occurs, the value of the *body-statement* is the value of the occurring clause.

   If neither the *else-clause* nor the *ensure-clause* occurs, 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* does not occur 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* occurs:

      - 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.

      - 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 §11.3.1.3).

      Let L be the list created by evaluating the *exception-class-list* as above. Compare each element D of L with E as follows:

      - If D is neither the class `Exception` nor a subclass of the class `Exception`, raise a direct instance of the class `TypeError`.

- If E is an instance of D, the *rescue-clause* handles E. In this case, any remaining *rescue-clause*s 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 occurs, 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* occurs, evaluate it. The value of the *body-statement* is the value of the *ensure-clause*.

4) If the *ensure-clause* does not occur, the value of the *body-statement* is V.

e) If no *rescue-clause* occurs or if a *rescue-clause* which can handle E is not found, evaluate the *ensure-clause*. In this case, the value of the *body-statement* is undefined.

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*.

11.4.2 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.

11.4.3 Variable references

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 *whitespace* here] `::` *constant-identifier*
  | `::` *constant-identifier*

11.4.3.1 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:

   - If C is a class:

     1) Let S be the direct superclass of C.

     2) If S is `nil`, 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.

     3) If S is not `nil`, search for a binding of a constant with name N in S.

     4) 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.

     5) If the binding is not found, let S be the direct superclass of S. Continue processing from Step e-2.

- If C is a module:

    1) Search for a binding of a constant with name N in the class `Object`.

    2) 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.

    3) 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.

11.4.3.2 Scoped constants

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

a) If the *primary-expression* occurs, 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, search for a binding of a constant with name N from Step e of §11.4.3.1.

11.4.3.3 Global variables

A *global-variable-identifier* is evaluated as follows:

- Let N be the *global-variable-identifier*.
- If a binding with name N exists in $[\![$global-variable-bindings$]\!]$, the value of *global-variable-identifier* is the value of the binding.
- Otherwise, the value of *global-variable-identifier* is `nil`.

11.4.3.4 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 an eigenclass.

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 dependent.

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 property.

f) The value of the *class-variable-identifier* is V.

11.4.3.5 Instance variables

An *instance-variable-identifier* is evaluated as follows:

- Let N be the *instance-variable-identifier*.
- 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.
- Otherwise, the value of the *instance-variable-identifier* is `nil`.

11.4.3.6 Local variables

This subclause describes a *local-variable-identifier* which is a reference to a local variable (see §9.1.2).

A *local-variable-identifier* 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 §9.1.1.

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`.

11.4.3.7 Pseudo variables

Syntax

---

pseudo-variable ::
    nil

| *true*
| *false*
| *self*

11.4.3.7.1 `nil`

Syntax

---

nil ::
    `nil`

Semantics

The pseudo variable `nil` evaluates to the only instance of the class `NilClass` (see §15.2.4).

11.4.3.7.2 `true` and `false`

Syntax

---

true ::
    `true`

false ::
    `false`

Semantics

The pseudo variable `true` is evaluates to the only instance of the class `TrueClass` (see §15.2.5).

The pseudo variable `false` is evaluates to the only instance of the class `FalseClass` (see §15.2.6).

11.4.3.7.3 `self`

Syntax

---

self ::
    `self`

Semantics

The pseudo variable `self` is evaluates to the value of the current self.

11.4.4 Object constructors

11.4.4.1 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 §11.2.1. Let L be the resulting list.

b) Otherwise, create an empty list of values L.

c) Create a direct instance of the class `Array` 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.

11.4.4.2 Hash constructor

Syntax

---

hash-constructor ::
    `{` (*association-list* `,`?)? `}`

association-list ::
    association (`,` *association*)*

association ::
    association-key `=>` *association-value*

association-key ::
    operator-expression

association-value ::
    operator-expression

The *operator-expression* of an *association-key* or *association-value* shall not be a *jump-expression*.

Semantics

Both *hash-constructor*s or *association-list*s evaluate to a direct instance of the class `Hash` (see §15.2.13).

A *hash-constructor* is evaluated as follows:

a) If there is an *association-list*, evaluate the *association-list*. The value of the *hash-constructor* is the resulting value.

b) Otherwise, create an empty direct instance of the class `Hash`. The value of the *hash-constructor* is the resulting instance.

An *association-list* is evaluated as follows:

a) Create a direct instance of the class `Hash` H.

b) For each *association* A_i, in the order it appears in the program text, take the following steps:

   1) Evaluate the *operator-expression* of the *association-key* of A_i. Let K_i be the resulting value.

   2) Evaluate the *operator-expression* of the *association-value*. Let V_i be the resulting value.

   3) Store a pair of K_i and V_i in H, as if by invoking the method `[]=` on H with K_i and V_i as the arguments.

c) The value of the *association-list* is H.

11.4.4.3 Range constructor

Syntax

---

range-constructor ::
    logical-OR-expression
   | *logical-OR-expression*$_1$ *range-operator* *logical-OR-expression*$_2$

range-operator ::
    `..`
   | `...`

Semantics

A *range-constructor* of the form *logical-OR-expression*$_1$ *range-operator* *logical-OR-expression*$_2$ is evaluated as follows:

a) Evaluate the *logical-OR-expression*$_1$. Let A be the resulting value.

b) Evaluate the *logical-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` with L as the list of arguments. The value of the *range-constructor* is the resulting value.

11.4.5 Literals

See §8.5.5.1 for integer literals.

See §8.5.5.1 for float literals.

See §8.5.5.2 for string literals.

See §8.5.5.5 for symbol literals.

See §8.5.5.4 for regular expression literals.

12 Statements

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*

See §13.3.6 for *alias-statement*.

See §13.3.7 for *undef-statement*.

See §11.3.1 for *assignment-statement*.

Semantics

See §11.3.1 for *assignment-statement*.

12.1 The expression statement

Syntax

expression-statement ::
    expression

Semantics

An *expression-statement* is evaluated as follows:

a) Evaluate the *expression*.

b) The resulting value is the value of the *expression-statement*.

12.2 The `if` modifier statement

Syntax

if-modifier-statement ::
    statement [no *line-terminator* here] `if` *expression*

The *expression* of an *if-modifier-statement* shall not be a *jump-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 resulting value is the value of the *if-modifier-statement*.

12.3 The `unless` modifier statement

Syntax

unless-modifier-statement ::
    statement [no *line-terminator* here] `unless` *expression*

The *expression* of an *unless-modifier-statement* shall not be a *jump-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 resulting value is the value of the *unless-modifier-statement*.

12.4 The `while` modifier statement

Syntax

---

while-modifier-statement ::
    statement [no *line-terminator* here] `while` *expression*

The *expression* of a *while-modifier-statement* shall not be a *jump-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) Evaluate the *while-expression* of the form `while` E `do` S `end`.

b) The resulting value is the value of the *while-modifier-statement*.

12.5 The `until` modifier statement

Syntax

---

until-modifier-statement ::
    statement [no *line-terminator* here] `until` *expression*

The *expression* of an *until-modifier-statement* shall not be a *jump-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) Evaluate the *until-expression* of the form `until` E `do` S `end`.

b) The resulting value is the value of the *until-modifier-statement*.

12.6 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*

statements-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 an instance of the class `StandardError` is raised and not handled in Step a, evaluate *fallback-statement-of-rescue-modifier-statement*. The resulting value is the value of the *rescue-modifier-statement*.

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: An ordered 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:

- Assignments (see §11.3.1)

- Module definitions (see §13.1.2)

- Class definitions (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 §11.3.1).

Instance methods: A set of method bindings. A method binding is created by a method definition (see §13.3.1) or a singleton method definition (see §13.4.3). 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 in which a binding with name *module-name* is to be created or modified as follows:

   - If the *module-path* is of the form *top-module-path*, let C be the class `Object`.

   - If the *module-path* is of the form *module-name*, let C be the current class or module.

   - 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 of the class `Module` and let M be that 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:

- 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⟧ .
- Push M onto ⟦self⟧ .
- Push the public visibility onto ⟦default-visibility⟧ .
- Push an empty set of bindings onto ⟦local-variable-bindings⟧ .

d) Evaluate the *module-body*. The value of the *module-definition* is the value of the *module-body*.

e) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧ , ⟦self⟧ , ⟦default-visibility⟧ , and ⟦local-variable-bindings⟧ , even when an exception is raised and not handled during Step d.

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 §11.4.3.4, §13.3.3, and §11.4.3.1).

Modules and classes can include other modules by invoking the method `include` (see §15.2.2.3.27) or the method `extend` (see §15.3.1.2.13).

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, each class has a single direct superclass.

13.2.2 Class definition

Syntax

---

class-definition ::
  `class` *class-path* [no *line-terminator* here] *superclass* *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 ::
  separator
  | `<` *expression* *separator*

class-body ::
  body-statement

Semantics

A *class-definition* is evaluated as follows:

a) Determine the class or module in which the binding with name *class-name* is to be created or modified as follows:

- If the *class-path* is of the form *top-class-path*, let M be the class `Object`.
- If the *class-path* is of the form *class-name*, let M be the current class or module.
- 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* occurs, 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* occurs, 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 *superclass* is an eigenclass or the class `Class`, the behavior is implementation dependent. Otherwise, let the direct superclass of C be the value of the *superclass*.

ii) If the *superclass* of the *class-definition* does not occur, let the direct superclass of C be the class `Object`.

iii) Create an eigenclass, and associate it with C. The eigenclass shall have the eigenclass of the direct superclass of C as one of its superclasses.

iv) Create a variables binding with name N and value C in the set of bindings of constants of M.

c) Modify the execution context as follows:

- 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⟧.
- Push C onto ⟦self⟧.
- Push the public visibility onto ⟦default-visibility⟧.
- Push an empty set of bindings onto ⟦local-variable-bindings⟧.

d) Evaluate the *class-body*. The value of the *class-definition* is the value of the *class-body*.

e) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧, ⟦self⟧, ⟦default-visibility⟧, and ⟦local-variable-bindings⟧, even when an exception is raised and not handled during Step d.

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 referenced (see §11.4.3.1 and §11.4.3.4).
- 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` on the class (see §15.2.3.2.3).

13.3 Methods

13.3.1 Method definition

Syntax

---

method-definition ::
    `def` *method-name* [no *line-terminator* here] *method-parameter-part*
        method-body `end`

method-name ::
    method-identifier
    | *operator-method-name*
    | *reserved-word*

method-body ::
    body-statement

The following constructs shall not occur 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*, where the form of any of the *left-hand-side*s which occurs in it is any of the following:
  - — *constant-identifier*
  - — *primary-expression* [no *line-terminator* here] (**.** | **::**) (*local-variable-identifier* | *constant-identifier*)
  - — **::** *constant-identifier*.

However, those constructs may occur within an *eigenclass-definition* in the *method-parameter-part* or the *method-body*.

Semantics

A method is defined by a *method-definition* and has the *method-parameter-part* and the *method-body* of the *method-definition*. 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 *method-definition* is evaluated as follows:

a) Let N be the *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 an eigenclass, 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? , *block-parameter*?
    | *optional-parameter-list* , *array-parameter*? , *block-parameter*?

| *array-parameter* , *block-parameter* ?
| *block-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

block-parameter ::
& *block-parameter-name*

block-parameter-name ::
local-variable-identifier

All the *local-variable-identifier*s of *mandatory-parameter*s, *optional-parameter-name*s, *array-parameter-name*, and *block-parameter-name* of a *parameter-list* shall be pairwise 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 arguments and optional arguments, 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 block parameter: This parameter is represented by *block-parameter-name*. The parameter is bound to the block passed to the method invocation.

13.3.3 Method invocation

The way in which a list of arguments is created are described in §11.2.

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 eigenclass of R if R has an eigenclass. 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 implementation dependent. 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` on R with A as arguments and B as the block. Let O be the resulting value, and go to Step j.

f) If the method is not invoked internally by a Ruby processor, 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:

- Push the class module list of V onto ⟦class-module-list⟧.
- Push R onto ⟦self⟧.

- Push M onto ⟦invoked-method-name⟧.
- Push the public visibility to ⟦default-visibility⟧.
- Push the defined name of V onto ⟦defined-method-name⟧.
- Push B onto ⟦block⟧.
- 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 argument A_i in A, in the same order in A, take the following steps:

i) Let P_i be the *mandatory-parameter* or the *optional-parameter* whose position in the L corresponds to the position of A_i in A.

- If such P_i exists, let n be the *mandatory-parameter* if P_i is a mandatory parameter, or *optional-parameter-name* if P_i is an optional parameter. Create a variable binding with name n and value A_i in S_b.
- If such P_i does no exist, i.e. if N_A is larger than the sum of N_{Pm} and N_{Po}, and P_a exists:

I) Create a direct instance of the class `Array` X 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.

ii) If N_A is smaller than the sum of N_{Pm} and N_{Po}:

I) For each optional argument P_{Oi} to which no argument corresponds, evaluate the *default-parameter-expression* of P_{Oi}, and let V 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 V in S_b.

iii) 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 of the class `Array` V.

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

III) Create a variable binding with name n and value V in S_b.

iv) If the *block-parameter* of L occurs, let D be the top of ⟦block⟧ .

I) If D is block-not-given, let V be `nil`.

II) Otherwise, invoke the method `new` on the class `Proc` with an empty list of arguments and D as the block. Let V be the resulting value of the method invocation.

III) Let n be the *block-parameter-name* of *block-parameter*.

IV) Create a variable binding with name n and value V in S_b.

i) Evaluate the *method-body* of V.

- If the evaluation of the *method-body* is terminated by a *return-expression*:
  - — If the *jump-argument* of the *return-expression* occurs, let O be the value of the *jump-argument*.
  - — Otherwise, let O be `nil`.
- 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-visibility⟧ , ⟦defined-method-name⟧ , ⟦block⟧ , and ⟦local-variable-bindings⟧ , even when an exception is raised and not handled in Step i.

k) The value of the method invocation is O.

The method invocation or the super expression (see Step d of §11.2.3) 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 list of included modules 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 this binding.

    2) Otherwise:

        - If the direct superclass of C is `nil`, the binding is considered not resolved.

        - Otherwise, replace C with the direct superclass of C, and continue processing from Step a.

c) B is the resolved method binding.

13.3.5 Method visibility

Methods are categorized into one of public, private, or protected methods according to the conditions on which the method invocation is allowed. The attribute of a method which determines these conditions is called the ***visibility*** of the method.

13.3.5.1 Public methods

A public method is a method whose visibility is the public visibility.

A public method can be invoked on an object anywhere within a program.

13.3.5.2 Private methods

A private method is a method whose visibility is the private visibility.

A private method can not be invoked with explicit receiver, i.e. method invocations of the forms where *primary-expression* or *chained-method-invocation* occurs at the position which corresponds to the method receiver are not allowed, except for method invocations of the following forms where the *primary-expression* is `self`.

- *single-method-assignment*
- *abbreviated-method-assignment*
- *single-indexing-assignment*
- *abbreviated-indexing-assignment*

13.3.5.3 Protected methods

A protected method is a method whose visibility is 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 an eigenclass, whether the method can be invoked or not may be determined in a implementation defined way.

13.3.5.4 Visibility change

The visibility of methods can be changed with built-in methods `public` (§15.2.2.3.38), `private` (§15.2.2.3.36), and `protected` (§15.2.2.3.37), which are defined in the class `Module`.

13.3.6 The `alias` statement

Syntax

---

alias-statement ::
  `alias` *new-name* *aliased-name*

new-name ::
  method-name
  | *symbol*

aliased-name ::
  method-name
  | *symbol*

Semantics

An *alias-statement* is evaluated as follows:

a) Evaluate the *new-name* as follows:

- If the *new-name* is of the form *method-name*, let N be the *method-name*.
- 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:

- If *aliased-name* is of the form *method-name*, let A be the *method-name*.
- 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 property.

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 *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 ::
    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 *method-name*, let N be the *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 property.

b) The value of *undef-statement* is `nil`.

13.4 Eigenclass

13.4.1 General description

An eigenclass is a class which is associated with a single object. An object, unless it is an instances of the class `Class`, becomes associated with an eigenclass through a *singleton-method-definition* or an *eigenclass-definition*. An instance of the class `Class` becomes associated with an eigenclass when it is created.

The direct superclass of an eigenclass is implementation defined. However, an eigenclass shall be a subclass of the class of the object with which it is associated.

NOTE 1 For example, the eigenclass 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 eigenclass of a class whose direct superclass is not `nil` shall satisfy the following condition:

- Let E_c be the eigenclass of a class C, and let S be the direct superclass of C, and let E_s be the eigenclass 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 eigenclass of the class `File` has the eigenclass of the class `IO` as its superclass. Thereby, the class `File` inherits the singleton method `open` of the class `IO`.

The eigenclass of an object is unique in the sense that no two objects become associated with the same eigenclass.

Although eigenclasses are instances of the class `Class`, they cannot create an instance of themselves. When the method `new` is invoked on an eigenclass, a direct instance of the class `TypeError` shall be raised (see Step a of §15.2.3.2.3).

Whether an eigenclass can be a superclass of other classes is implementation dependent (see Step b-2-i of §13.2.2 and Step c of §15.2.3.2.1).

Whether an eigenclass can have class variables or not is implementation defined.

13.4.2 Eigenclass definition

Syntax

---

eigenclass-definition ::
  `class <<` *expression* *separator* *eigenclass-body* `end`

eigenclass-body ::
  body-statement

Semantics

An *eigenclass-definition* is evaluated as follows:

a) Evaluate the *expression*. Let O be the resulting value. A conforming processor may specify the set of classes such that if O is an instance of one of the classes in the set, a direct instance of the class `TypeError` is raised.

b) If O is one of `nil`, `true`, or `false`, let E be the class of O and go to Step e.

c) If O is not associated with an eigenclass, create a new eigenclass. Let E be the newly created eigenclass.

d) If O is associated with an eigenclass, let E be that eigenclass.

e) Modify the execution context as follows:

- 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⟧ .
- Push E onto ⟦self⟧ .
- Push the public visibility onto ⟦default-visibility⟧ .
- Push an empty set of bindings onto ⟦local-variable-bindings⟧ .

f) Evaluate the *eigenclass-body*. The value of the *eigenclass-definition* is the value of the *eigenclass-body*.

g) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧ , ⟦self⟧ , ⟦default-visibility⟧ , and ⟦local-variable-bindings⟧ , even when an exception is raised and not handled during Step f.

13.4.3 Singleton method definition

Syntax

---

singleton-method-definition ::
  `def` *singleton* (`.` | `::`) *method-name* [no *line-terminator* here]
    method-parameter-part *method-body* `end`

singleton ::
  variable
  | *pseudo-variable*
  | `(` *expression* `)`

Semantics

A *singleton-method-definition* is evaluated as follows:

a) Evaluate the *singleton*. Let S be the resulting value.

b) If S is one of `nil`, `true`, or `false`, let E be the class of O and go to Step e.

c) If S is not associated with an eigenclass, create a new eigenclass. Let E be the newly created eigenclass.

d) If S is associated with an eigenclass, Let E be that eigenclass.

e) Let N be the *method-name*.

f) 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 the public visibility.

g) 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.

h) Otherwise, create a method binding with name N and value U in the set of bindings of instance methods of E.

i) The value of the *singleton-method-definition* is implementation defined.

14 Exceptions

If an instance of the class `Exception` is raised, the current evaluation process stops, and the evaluation process is transfered to a program construct that can handle this exception.

14.1 Cause of exceptions

An exception is raised when:

- the method `raise` (see §15.3.1.1.13) is invoked.
- a certain exceptional condition occurs as described in various parts of this document.

Only instances of the class `Exception` shall be raised.

14.2 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 §11.3.1.4), 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 §12.6), 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 §11.4.1.4.1), except when the exception is raised in one of its *rescue-clause*s, *else-clause*s, or *ensure-clause*s. In this case, *body-statement* does not handle the exception. If an *ensure-clause* of a *body-statement* occurs, it is evaluated even if the handler does not handle the exception (see §11.4.1.4.1).

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 §11.4.1.4.1, §11.3.1.4, and §12.6).

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 , and take Step j of the current method invocation. 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*. Continue the search from Step a, under the assumption that the exception is raised at the location where the block is called.

   3) Otherwise, terminate the evaluation of the *program*.

15 Built-in classes and modules

Built-in classes and modules are specified in this clause. Those classes and modules shall be defined in the class `Object` as constants.

15.1 General description

Each built-in class or module is specified by describing its attribute values, as described in §13.1 and §13.2.

When a clause specifying a built-in class or module contains a subclause titled "Included modules", the built-in class or module shall include the modules listed in that subclause in the order of that listing.

Each subclause in the subclause titled "Singleton methods" with the title of the form *C.m* specifies the singleton method *m* of the class *C*.

Each subclause in the subclause titled "Instance methods" with the 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 §13.3.1).

EXAMPLE 1 The following example defines the parameter specification of a method `sample`.

```
sample( arg1, arg2, opt=expr, *ary, &blk )
```

For a singleton method, the 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 Step i of §13.3.3), 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 the *block* with X as the argument" indicates that the block corresponding to the block parameter *block* shall be called as described in §11.2.2 with X as the argument to the block call. The phrase "return X" indicates that the evaluation of the *method-body* shall be terminated at that point, and X shall be 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 of N.

c) Otherwise, the behavior of the method is implementation dependent.

The class module list of an instance method of a built-in class or module shall be a list which consists of two elements: the first is the built-in class or module; the second is the class `Object`. The class module list of a singleton method of a built-in class or module shall be a list which consists of two elements: the first is the eigenclass of the built-in class or module; the second is the class `Object`.

A conforming processor may provide additional attributes and/or values: a specific initial value for a predefined attribute whose initial value is not specified in this document, constants, singleton methods, instance methods, and additional inclusion of modules into built-in classes/modules.

15.2 Built-in classes

15.2.1 Object

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 referenced through constants of the class `Object`.

15.2.1.1 Direct superclass

The value `nil`.

A conforming processor may define a sequence of superclasses of the class `Object`. However, the direct superclass of the class at the top of the class hierarchy shall always be `nil`.

15.2.1.2 Included modules

The following module is included in the class `Object`.

- `Kernel`

15.2.1.3 Constants

The following constants are defined in the class `Object`.

STDIN: An implementation defined instance of the class `IO`, which shall be readable.

STDOUT: An implementation defined instance of the class `IO`, which shall be writable.

STDERR: An implementation defined instance of the class `IO`, which shall be writable.

Besides, every built-in class or module, including the class `Object` itself, shall be defined in the class `Object` as a constant, whose name is the name of the class or module, and whose value is the class or module.

15.2.1.4 Instance methods

15.2.1.4.1 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.

15.2.2 Module

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 classes and B is a superclass of A.

15.2.2.1 Direct superclass

The class `Object`

15.2.2.2 Singleton methods

15.2.2.2.1 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 direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which of the these classes of instance 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`, append S to A.

d) Return A.

15.2.2.2.2 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.

15.2.2.3 Instance methods

15.2.2.3.1 Module#<

`<(` *other* `)`

Visibility: public

Behavior: Let A be the *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 is the same object, return `false`.

c) If $R \sqsubset A$, return `true`.

d) If $A \sqsubset R$, return `false`.

e) Otherwise, return `nil`.

15.2.2.3.2 Module#<=

`<=(` *other* `)`

Visibility: public

Behavior:

a) If the *other* and the receiver are the same object, return `true`.

b) Otherwise, the behavior is the same as the method `<` (see §15.2.2.3.1).

15.2.2.3.3 Module#<=>

`<=>(` *other* `)`

Visibility: public

Behavior: Let A be the *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 is 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`.

15.2.2.3.4 Module#==

`==(` *other* `)`

Visibility: public

Behavior: Same as the method `==` of the module `Kernel` (see §15.3.1.2.1).

15.2.2.3.5 Module#===

`===(` *object* `)`

Visibility: public

Behavior: The method behaves as if the method `kind_of?` were invoked on the *object* with the receiver as the only argument (see §15.3.1.2.26).

15.2.2.3.6 Module#>

`>(` *other* `)`

Visibility: public

Behavior: Let A be the *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 is the same object, return `false`.

c) If $R \sqsubset A$, return `false`.

d) If $A \sqsubset R$, return `true`.

e) Otherwise, return `nil`.

15.2.2.3.7 Module#>=

`>=(` *other* `)`

Visibility: public

Behavior:

a) If the *other* and the receiver are the same object, return `true`.

b) Otherwise, the behavior is the same as the method `>` (see §15.2.2.3.6).

15.2.2.3.8 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 the *new_name*. Let A be the name designated by the *aliased_name*.

b) Take Step 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.

15.2.2.3.9 Module#ancestors

`ancestors`

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array` A.

b) Let C be the receiver of the method.

c) If C is not an eigenclass, append C to A.

d) Append each element of the included module list of C, in the order in the list, to A.

e) If C is a class, replace C with the direct superclass of the current C.

f) If C is not `nil`, repeat from Step c.

g) Return A.

15.2.2.3.10 Module#append_features

`append_features(` *module* `)`

Visibility: private

Behavior: Let *L1* and *L2* be the included modules list of the receiver and the *module* respectively.

a) If the *module* and the receiver is the same object, the behavior is implementation dependent.

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 the *module* are the same object, the behavior is implementation dependent.

   2) If M is not in *L2*, append M to the end of *L2*.

d) Append the receiver to *L2*.

e) Return the receiver.

15.2.2.3.11 Module#attr

`attr(` *symbol, writable*`=false` `)`

Visibility: private

Behavior: Let C be the method receiver.

a) If the *symbol* is not an instance of the class `Symbol`, the behavior is implementation dependent.

b) Let N be the name of the *symbol*.

c) If N is not of the form *local-variable-identifier* or *constant-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

d) 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
```

e) If the *writable* is `true`, 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
```

f) Return `nil`.

15.2.2.3.12 Module#attr_accessor

`attr_accessor(` **symbol_list* `)`

Visibility: private

Behavior:

a) For each element S of the *symbol_list*, invoke the method `attr` with S as the first argument and `true` as the second argument (see §15.2.2.3.11).

b) Return `nil`.

15.2.2.3.13 Module#attr_reader

`attr_reader(` **symbol_list* `)`

Visibility: private

Behavior:

a) For each element S of the *symbol_list*, invoke the method `attr` with S as the first argument and `false` as the second argument (see §15.2.2.3.11).

b) Return `nil`.

15.2.2.3.14 Module#attr_writer

`attr_writer(` `*`*symbol_list* `)`

Visibility: private

Behavior:

a) For each element S of the *symbol_list*, invoke the method `attr` with S as the first argument and `true` as the second argument, but skip Step d (see §15.2.2.3.11).

b) Return `nil`.

15.2.2.3.15 Module#class_eval

`class_eval(` *string* `=` *nil,* `&`*block* `)`

Visibility: public

Behavior:

a) Let M be the receiver.

b) If the *block* is given:

1) If the *string* is given, raise a direct instance of the class `ArgumentError`.

2) Call the *block* with implementation defined arguments as described in §11.2.2, and let V be the resulting value. A conforming processor shall modify the execution context just before Step d of §11.2.2 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-visibility⟧.

In Step d and e of §11.2.2, a conforming processor may 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 the *block* is not given:

1) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

2) Modify the execution context 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-visibility⟧ .

3) Parse the content of the *string* as a *program* (see §10.1). If it fails, raise a direct instance of the class `SyntaxError`.

4) Evaluate the *program*. Let *V* be the resulting value of the evaluation.

5) Restore the execution context by removing the elements from the tops of ⟦class-module-list⟧ , ⟦self⟧ , and ⟦default-visibility⟧ , even when an exception is raised and not handled in Step c-3 or c-4.

6) Return *V*.

In Step c-4, the *string* is evaluated under the new local variable scope in which references to *local-variable-identifier*s are resolved in the same way as in scopes created by *block*s (see §9.1.1).

15.2.2.3.16 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 the *symbol*.

b) If *N* is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

c) Search for a binding of the class variable with name *N* by taking Step b through d of §11.4.3.4, assuming that *C* and *N* in §11.4.3.4 to be *C* and *N* in the above steps.

d) If a binding is found, return `true`.

e) Otherwise, return `false`.

15.2.2.3.17 Module#class_variable_get

`class_variable_get(` *symbol* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

c) Search for a binding of the class variable with name N by taking Step b through d of §11.4.3.4, assuming that C and N in §11.4.3.4 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 the *symbol* as its name property.

15.2.2.3.18 Module#class_variable_set

`class_variable_set(` *symbol, obj* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

c) Search for a binding of the class variable with name N by taking Step b through d of §11.4.3.4, assuming that C and N in §11.4.3.4 to be C and N in the above steps.

d) If a binding is found, replace the value of the binding with the *obj*.

e) Otherwise, create a variable binding with name N and value *obj* in the set of bindings of class variables of C.

15.2.2.3.19 Module#class_variables

`class_variables`

Visibility: public

Behavior: The methods returns a 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.

15.2.2.3.20 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 the *symbol*.

c) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

d) If a binding with name N exists in the set of bindings of constants of C, return `true`.

e) Otherwise, return `false`.

15.2.2.3.21 Module#const_get

`const_get(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

c) Search for a binding of a constant with name N from Step e of §11.4.3.1, assuming that C in §11.4.3.1 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 Step e-2 of §11.4.3.1).

15.2.2.3.22 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 §11.4.3.1).

When the method is invoked, take the following steps:

a) Take Step a through c of §15.2.2.3.20.

b) Raise a direct instance of the class `NameError` which has the *symbol* as its name property.

15.2.2.3.23 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 the *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

c) If a binding with name N exists in the set of bindings of constants of C, replace the value of the binding with the *obj*.

d) Otherwise, create a variable binding with N and value *obj* in the set of bindings of constants of C.

e) Return the *obj*.

15.2.2.3.24 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.

15.2.2.3.25 Module#extend_object

`extend_object(` *object* `)`

Visibility: private

Behavior: Let S be the eigenclass of the *object*. The method behaves as if by invoking the method `append_features` on the receiver with S as the only argument.

15.2.2.3.26 Module#extended

`extended(` *object* `)`

Visibility: private

Behavior: The method returns `nil`.

15.2.2.3.27 Module#include

`include( *`*module_list* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) For each element A of the *module_list*, in the reverse order in the *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` on A with C as the only argument.

   4) Invoke the method `included` on A with C as the only argument.

b) Return C.

15.2.2.3.28 Module#include?

`include?(` *module* `)`

Visibility: public

Behavior: Let C be the receiver of the method.

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

b) If the *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 the *module* is an element of the included module list of one of the superclasses of C, then return `true`.

d) Otherwise, return `false`.

15.2.2.3.29 Module#included

`included(` *module* `)`

Visibility: private

Behavior: The method returns `nil`.

15.2.2.3.30 Module#included_modules

`included_modules`

Visibility: public

Behavior: Let C be the receiver of the method.

a) Create an empty direct instance of the class `Array` A.

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`, replace C with the direct superclass of the current C

d) If C is not `nil`, repeat from Step b.

e) Return A.

15.2.2.3.31 Module#initialize

`initialize(` &*block* `)`

Visibility: private

Behavior:

a) If the *block* is given, call the *block* as if invoking the method `class_eval` of the class `Module` on the receiver with no arguments and the *block* as the block.

b) Return an implementation defined value.

15.2.2.3.32 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 the *original* as the argument.

b) If the receiver is associated with an eigenclass, let E_o be the eigenclass, and take the following steps:

1) Create an eigenclass whose direct superclass is the direct superclass of E_o. Let E_n be the eigenclass.

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`, set the direct superclass of the receiver to the direct superclass of the *original*.

d) Append each element of the included module list of the *original*, in the same order, to the receiver.

e) For each binding B_{v3} of the constants of the *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 the *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 the *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.

15.2.2.3.33 Module#instance_methods

`instance_methods(` *include_super* `=true )`

Visibility: public

Behavior: Let C be the receiver of the method.

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

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 direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which of the these classes of instance 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 the *include_super* is a true value:

  1) For each module M in included module list of C, take Step b, assuming that C in that step to be M.

  2) Replace C with the direct superclass of C.

  3) If C is not *nil*, repeat from Step b.

d) Return A.

15.2.2.3.34 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 the *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`.

15.2.2.3.35 Module#module_eval

`module_eval(` *string* `=` *nil,* `&`*block* `)`

Visibility: public

Behavior: Same as the method `class_eval` (see §15.2.2.3.15)

15.2.2.3.36 Module#private

`private(` `*`*symbol-list* `)`

Visibility: private

Behavior: Same as the method `public` (see §15.2.2.3.38), except that the method changes current visibility or visibilities of methods corresponding to each element of the *symbol_list* to private.

15.2.2.3.37 Module#protected

`protected(` `*`*symbol-list* `)`

Visibility: private

Behavior: Same as the method `public` (see §15.2.2.3.38), except that the method changes current visibility or visibilities of methods corresponding to each element of the *symbol_list* to protected.

15.2.2.3.38 Module#public

`public(` `*`*symbol_list* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) If the length of *symbol_list* is 0, change the current visibility to public and return C

b) Otherwise, for each element S of the *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 property.

5) If C is the class or module in which the binding is found, change the visibility of V to the public visibility.

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 the public visibility.

c) Return C.

15.2.2.3.39 Module#remove_class_variable

`remove_class_variable(` *symbol* `)`

Visibility: private

Behavior: Let C be the receiver of the method.

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *class-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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 the *symbol* as its name property.

15.2.2.3.40 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 the *symbol*.

b) If N is not of the form *constant-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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 the *symbol* as its name property.

15.2.2.3.41 Module#remove_method

```
remove_method( *symbol_list )
```

Visibility: private

Behavior: Let C be the receiver of the method.

a) For each element S of the *symbol_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, remove the binding from the set.

   3) Otherwise, raise a direct instance of the class `NameError` which has S as its name property. In this case, the remaining elements of the *symbol_list* are not processed.

b) Return C.

15.2.2.3.42 Module#undef_method

```
undef_method( *symbol_list )
```

Visibility: private

Behavior: Let C be the receiver of the method.

a) For each element S of the *symbol_list*, take following steps:

   1) Let N be the name designated by S.

   2) Take Step a-3 and a-4 of §13.3.7, assuming that C in §13.3.7 to be C and N in the above steps.

b) Return C.

15.2.3 Class

All classes are instances of the class `Class`. Therefore, behaviors defined in the class `Class` are shared by all classes.

A conforming processor shall undefine the instance methods `append_features` and `extend_object` of the class `Class`, as if by invoking the method `undef_method` on the class `Class` with instances of the class `Symbol` whoses names are "append_features" and "extend_object" as the arguments (see §15.2.2.3.42).

15.2.3.1 Direct superclass

The class `Module`

15.2.3.2 Instance methods

15.2.3.2.1 Class#initialize

`initialize(` *superclass*`=`*Object*, `&`*block* `)`

Visibility: private

Behavior:

a) If the receiver has its direct superclass, raise a direct instance of the class `TypeError`.

b) If the *superclass* is not an instance of the class `Class`, raise a direct instance of the class `TypeError`.

c) If the *superclass* is an eigenclass or the class `Class`, the behavior is implementation dependent.

d) Set the direct superclass of the receiver to the *superclass*.

e) Create an eigenclass, and associate it with the receiver. The eigenclass shall have the eigenclass of the *superclass* as one of its superclasses.

f) If the *block* is given, call the *block* as if invoking the method `class_eval` of the class `Module` on the receiver with no arguments and the *block* as the block.

g) Return an implementation defined value.

15.2.3.2.2 Class#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the direct superclass of the reciver has already been set, raise a direct instance of the class `TypeError`.

b) If the receiver is an eigenclass, raise a direct instance of the class `TypeError`.

c) Invoke the instance method `initialize_copy` defined in the class `Module` on the receiver with the *original* as the argument.

d) Return an implementation defined value.

15.2.3.2.3 Class#new

`new(` `*`*args,* `&`*block* `)`

Visibility: public

Behavior:

a) If the receiver is an eigenclass, 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 the *args* as arguments and the *block* as the block.

d) Return O.

15.2.3.2.4 Class#superclass

`superclass`

Visibility: public

Behavior: Let C be the receiver of the methods.

a) If C is an eigenclass, return an implementation defined value.

b) Otherwise, return the direct superclass of C.

15.2.4 NilClass

The class `NilClass` has only one instance, which is represented by the pseudo variable `nil`.

Instances of the class `NilClass` shall not be created by the method `new` of the class `NilClass`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `NilClass`, as if by invoking the method `undef_method` on the eigenclass of the class `NilClass` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

15.2.4.1 Direct superclass

The class `Object`

15.2.4.2 Instance methods

15.2.4.2.1 NilClass#&

`&(` *other* `)`

Visibility: public

Behavior: The method returns `false`.

15.2.4.2.2 NilClass#^

`^(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is a false value, return `false`.

b) Otherwise, return `true`.

15.2.4.2.3 NilClass#nil?

`nil?`

Visibility: public

Behavior: The method returns `true`.

15.2.4.2.4 NilClass#|

`|( other )`

Visibility: public

Behavior:

a) If the *other* is a false value, return `false`.

b) Otherwise, return `true`.

15.2.5 TrueClass

The class `TrueClass` has only one instance, which is represented by the pseudo variable `true`. `true` represents a logical true value.

Instances of the class `TrueClass` shall not be created by the method `new` of the class `TrueClass`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `TrueClass`, as if by invoking the method `undef_method` on the eigenclass of the class `TrueClass` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

15.2.5.1 Direct superclass

The class `Object`

15.2.5.2 Instance methods

15.2.5.2.1 TrueClass#&

`&( other )`

Visibility: public

Behavior:

a) If the *other* is a false value, return `false`.

b) Otherwise, return `true`.

15.2.5.2.2 TrueClass#^

`^( other )`

Visibility: public

Behavior:

a) If the *other* is a false value, return `true`.

b) Otherwise, return `false`.

15.2.5.2.3 TrueClass#to_s

```
to_s
```

Visibility: public

Behavior: The method returns an instance of the class `String`, the content of which is "true".

15.2.5.2.4 TrueClass#|

```
|( other )
```

Visibility: public

Behavior: The method returns `true`.

15.2.6 FalseClass

The class `FalseClass` has only one instance, which is represented by the pseudo variable `false`. `false` represents a logical false value.

Instances of the class `FalseClass` shall not be created by the method `new` of the class `FalseClass`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `FalseClass`, as if by invoking the method `undef_method` on the eigenclass of the class `FalseClass` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

15.2.6.1 Direct superclass

The class `Object`

15.2.6.2 Instance methods

15.2.6.2.1 FalseClass#&

```
&( other )
```

Visibility: public

Behavior: The method returns `false`.

15.2.6.2.2 FalseClass#^

```
^( other )
```

Visibility: public

Behavior:

a) If the *other* is a false value, return `false`.

b) Otherwise, return `true`.

15.2.6.2.3 FalseClass#to_s

`to_s`

Visibility: public

Behavior: The method returns an instance of the class `String`, the content of which is "false".

15.2.6.2.4 FalseClass#|

`|(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is a false value, return `false`.

b) Otherwise, return `true`.

15.2.7 Numeric

Instances of the class `Numeric` represent numbers. The class `Numeric` is a 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 which N represent.

15.2.7.1 Direct superclass

The class `Object`

15.2.7.2 Included modules

The following module is included in the class `Numeric`.

- `Comparable`

15.2.7.3 Instance methods

15.2.7.3.1 Numeric#+@

```
+@
```

Visibility: public

Behavior: The method returns the receiver.

15.2.7.3.2 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`.

15.2.7.3.3 Numeric#abs

```
abs
```

Visibility: public

Behavior:

a) Invoke the method `<` on the receiver with an instance of the class `Integer` whose value is 0.

b) If this invocation results in a true value, invoke the method `-@` on the receiver and return the resulting value.

   Otherwise, return the receiver.

15.2.7.3.4 Numeric#coerce

`coerce(` *other* `)`

Visibility: public

Behavior:

a) If the class of the receiver and the class of the *other* are the same class, let X and Y be the *other* and the receiver, respectively.

b) Otherwise, let X and Y be instances of the class `Float` which are converted from the *other* and the receiver, respectively. the *other* and the receiver are converted as follows:

1) Let O be the *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 false value, 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 implementation dependent.

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

Instances of the class `Integer` represent integers. The ranges of these integers are unbounded.

Instances of the class `Integer` shall not be created by the method `new` of the class `Integer`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `Integer`, as if by invoking the method `undef_method` on the eigenclass of the class `Integer` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

A conforming processor may define subclasses of the class `Integer` which differ only in the ranges of the representing integer values. In this case, a conforming processor:

- shall define methods `+`, `-`, `*`, `/`, and `%` in all of these classes.

- shall not create a direct instance of the class `Integer`, but shall create a direct instance of one of these subclasses, instead of the class `Integer`.

If a conforming processor does not define any subclass of the class `Integer`, it shall define methods `+`, `-`, `*`, `/` and `%` in the class `Integer`.

15.2.8.1 Direct superclass

The class `Numeric`

15.2.8.2 Instance methods

15.2.8.2.1 Integer#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other*.

b) If the *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 the *other*.

c) Otherwise, invoke the method `coerce` on the *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`.

15.2.8.2.2 Integer#−

`-(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other* from the value of the receiver.

b) If the *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 the *other* from R.

c) Otherwise, invoke the method `coerce` on the *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`.

15.2.8.2.3 Integer#*

`*(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other*.

b) If the *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 the *other*.

c) Otherwise, invoke the method `coerce` on the *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`.

15.2.8.2.4 Integer#/

`/(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Integer`:

   1) If the value of the *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 the *other*. Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to n.

b) Otherwise, invoke the method `coerce` on the *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`.

15.2.8.2.5 Integer#%

`%(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Integer`:

   1) If the value of the *other* is 0, raise a direct instance of the class `ZeroDivisionError`.

   2) Otherwise, let x and y be the values of the receiver and the *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) If $m \times y < 0$, return an instance of the class `Integer` whose value is $m + y$.

iv) Otherwise, return an instance of the class `Integer` whose value is m.

b) Otherwise, invoke the method `coerce` on the *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`.

15.2.8.2.6 Integer#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Integer`:

1) If the value of the receiver is larger than the value of the *other*, return an instance of the class `Integer` whose value is 1.

2) If the values of the receiver and the *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 the *other*, return an instance of the class `Integer` whose value is -1.

b) Otherwise, invoke the method `coerce` on the *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 implementation dependent.

iii) Otherwise, return the value of this invocation.

2) Otherwise, return `nil`.

15.2.8.2.7 Integer#==

`==( other )`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Integer`:

1) If the values of the receiver and the *other* are the same integer, return `true`.

2) Otherwise, return `false`.

b) Otherwise, invoke the method `==` on the *other* with the receiver as the argument. Return the resulting value of this invocation.

15.2.8.2.8 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.

15.2.8.2.9 Integer#&

`&( other )`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, the behavior is implementation dependent.

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 the *other*.

15.2.8.2.10 Integer#|

`|( other )`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, the behavior is implementation dependent.

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 the *other*.

15.2.8.2.11 Integer#ˆ

`^`(*other*)

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, the behavior is implementation dependent.

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 the *other*.

15.2.8.2.12 Integer#<<

`<<`(*other*)

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) Otherwise, let x and y be the values of the receiver and the *other*.

c) Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to $x \times 2^y$.

15.2.8.2.13 Integer#>>

`>>`(*other*)

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) Otherwise, let x and y be the values of the receiver and the *other*.

c) Return an instance of the class `Integer` whose value is the largest integer smaller than or equal to $x \times 2^{-y}$.

15.2.8.2.14 Integer#ceil

```
ceil
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.2.15 Integer#downto

`downto(` *num,* `&`*block* `)`

Visibility: public

Behavior:

a) If the *num* is not an instance of the class `Integer`, or the *block* is not given, the behavior is implementation dependent.

b) Let i be the value of the receiver.

c) If i is smaller than the value of the *num*, return the receiver.

d) Call the *block* with an instance of the class `Integer` whose value is i.

e) Decrement i by 1 and continue processing from Step c.

15.2.8.2.16 Integer#eql?

`eql?(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Integer`, return `false`.

b) Otherwise, invoke the method `==` on the *other* with the receiver as the argument.

c) If this invocation results in a true value, return `true`. Otherwise, return `false`.

15.2.8.2.17 Integer#floor

```
floor
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.2.18 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 the H_1 and H_2 shall be the same integer, if and only if the values of I_1 and I_2 are the same integer.

15.2.8.2.19 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.

15.2.8.2.20 Integer#round

```
round
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.2.21 Integer#succ

```
succ
```

Visibility: public

Behavior: Same as the method `next` (see §15.2.8.2.19).

15.2.8.2.22 Integer#times

```
times( &block )
```

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Let i be 0.

c) If i is larger than or equal to the value of the receiver, return the receiver.

d) Call the *block* with an instance of the class `Integer` whose value is i.

e) Increment i by 1 and continue processing from Step c.

15.2.8.2.23 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.

15.2.8.2.24 Integer#to_i

```
to_i
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.2.25 Integer#truncate

```
truncate
```

Visibility: public

Behavior: The method returns the receiver.

15.2.8.2.26 Integer#upto

`upto(` *num*, `&`*block* `)`

Visibility: public

Behavior:

a) If the *num* is not an instance of the class `Integer`, or the *block* is not given, the behavior is implementation dependent.

b) Let i be the value of the receiver.

c) If i is larger than the value of the *num*, return the receiver.

d) Call the *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

Instances of the class `Float` represent floating point numbers. A conforming processor should use the native binary floating point representation of the underlying platform.

When an arithmetic operation involving floating point numbers results in a value which cannot be represented exactly as an instance of the class `Float`, how the result is rounded to fit in the representation of an instance of the class `Float` is implementation defined.

If the underlying platform of a conforming processor supports IEC 60559:1989:

- The representation of an instance of the class `Float` should be the 64-bit double format as specified in §3.2.2 of IEC 60559:1989.
- 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 implementation dependent.

Instances of the class `Float` shall not be created by the method `new` of the class `Float`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `Float`, as if by invoking the method `undef_method` on the eigenclass of the class `Float` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

15.2.9.1 Direct superclass

The class `Numeric`

15.2.9.2 Instance methods

15.2.9.2.1 Float#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other*.

b) If the *other* is an instance of the class `Integer`, let R be the value of the *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 the *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`.

15.2.9.2.2 Float#−

`-(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other* from the value of the receiver.

b) If the *other* is an instance of the class `Integer`, let R be the value of the *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 the *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`.

15.2.9.2.3 Float#*

`*(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other*.

b) If the *other* is an instance of the class `Integer`, let R be the value of the *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 the *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`.

15.2.9.2.4 Float#/

`/(` *other* `)`

Visibility: public

Behavior:

a) If the *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 the *other*.

b) If the *other* is an instance of the class `Integer`, let R be the value of the *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 the *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`.

15.2.9.2.5 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 the *other* is an instance of the class `Integer` or the class `Float`:

   Let x be the value of the receiver.

   1) If the *other* is an instance of the class `Float`, let y be the value of the *other*. If the *other* is an instance of the class `Integer`, let y be the value of the *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 the *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`.

15.2.9.2.6 Float#<=>

`<=>`(*other*)

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Integer` or the class `Float`:

   1) Let a be the value of the receiver. If the *other* is an instance of the class `Float`, let b be the value of the *other*. Otherwise, let b be the value of the *other* as a floating point number.

   2) If a conforming processor supports IEC 60559:1989, 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 the *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 implementation dependent.

      iii) Otherwise, return the value of this invocation.

   2) Otherwise, return `nil`.

15.2.9.2.7 Float#==

`==( other )`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Float`:

1) If a conforming processor supports IEC 60559:1989, and if the value of the receiver is NaN, then return `false`.

2) If the values of the receiver and the *other* are the same number, return `true`.

3) Otherwise, return `false`.

b) If the *other* is an instance of the class `Integer`:

1) If the values of the receiver and the *other* are the mathematically the same, return `true`.

2) Otherwise, return `false`.

c) Otherwise, invoke the method `==` on the *other* with the receiver as the argument and return the resulting value of this invocation.

15.2.9.2.8 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.

15.2.9.2.9 Float#finite?

`finite?`

Visibility: public

Behavior:

a) If the value of the receiver is a finite number, return `true`.

b) Otherwise, return `false`.

15.2.9.2.10 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.

15.2.9.2.11 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`.

15.2.9.2.12 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.

15.2.9.2.13 Float#to_f

```
to_f
```

Visibility: public

Behavior: The method returns the receiver.

15.2.9.2.14 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.

15.2.9.2.15 Float#truncate

```
truncate
```

Visibility: public

Behavior: Same as the method `to_i` (see §15.2.9.2.14).

15.2.10 String

Instances of the class `String` represent sequences of characters.

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.

The notation "the nth character of a string" means the character whose index is n counted up from 0.

15.2.10.1 Direct superclass

The class `Object`

15.2.10.2 Included modules

The following modules are included in the class `String`.

- `Comparable`

15.2.10.3 Instance methods

15.2.10.3.1 String#*

`*(` *num* `)`

Visibility: public

Behavior:

a) If the *num* is not an instance of the class `Integer`, the behavior is implementation dependent.

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 of the class `String` S the content of which is C repeated n times.

f) Return S.

15.2.10.3.2 String#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `String`, the behavior is implementation dependent.

b) Let S and O be the contents of the receiver and the *other* respectively.

c) Return a newly created instance of the class `String` the content of which is the concatenation of S and O.

15.2.10.3.3 String#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `String`, the behavior is implementation dependent.

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) If *S1* is empty, return an instance of the class `Integer` whose value is -1.

e) 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, replace *S1* and *S2* with *S1* and *S2* excluding their first characters, respectively. Continue processing from Step c.

15.2.10.3.4 String#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `String`, the behavior is implementation dependent.

b) If the *other* is an instance of the class `String`:

   1) If the content of the receiver and the *other* is the same, return `true`.

   2) Otherwise, return `false`.

15.2.10.3.5 String#=~

`=~(` *regexp* `)`

Visibility: public

Behavior:

a) If the *regexp* is not an instance of the class `Regexp`, the behavior is implementation dependent.

b) Otherwise, behave as if the method `match` is invoked on the *regexp* with the receiver as the argument (see §15.2.15.6.7).

15.2.10.3.6 String#[]

`[]( *args )`

Visibility: public

Behavior:

a) If the length of the *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.

b) Let P be the first element of the *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 the *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 the *args* is 2:

      i) If the last element of the *args* is an instance of the class `Integer`, let l be the value of the instance. Otherwise, the behavior is implementation dependent.

      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 the *args* is 1, let i be 0.

   2) If the length of the *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 implementation dependent.

3) Match the pattern of P against the content of the receiver. (see §15.2.15.3 and Step 15.2.15.4). 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 of M, return `nil`.

6) If i is smaller than 0, increment i by the length of the match result 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 of M. Create a direct instance of the class `String` whose content is the first element of m and return the instance.

e) If P is an instance of the class `String`:

1) If the length of the *args* is 2, the behavior is implementation dependent.

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 P and return the instance.

3) Otherwise, return `nil`.

f) Otherwise, the behavior is implementation dependent.

15.2.10.3.7 String#capitalize

```
capitalize
```

Visibility: public

Behavior: The method returns a newly created 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.

15.2.10.3.8 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.

15.2.10.3.9 String#chomp

`chomp(` *rs*`=`*"\n"*`)`

Visibility: public

Behavior:

a) If the *rs* is `nil`, return a newly created instance of the class `String` whose content is the same as the receiver.

b) If the receiver is empty, return a newly created empty instance of the class `String`.

c) If *rs* is not an instance of the class `String`, the behavior is implementation dependent.

d) Otherwise, return a newly created instance of the class `String` whose content is the same as the receiver, except the following characters:

1) If the *rs* consists of only one character 0x0a, the *line-terminator* on the end, if any, is excluded.

2) If the *rs* is empty, the 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 the characters at the end of the receiver is excluded.

15.2.10.3.10 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 the *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.

15.2.10.3.11 String#chop

`chop`

Visibility: public

Behavior:

a) If the receiver is empty, return a newly created empty instance of the class `String`.

b) Otherwise, return a newly created instance of the class `String` whose content is the receiver without the last character. If the last character is 0x0a, and the character just before the 0x0a is 0x0d, the 0x0d is also dropped.

15.2.10.3.12 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.

15.2.10.3.13 String#downcase

`downcase`

Visibility: public

Behavior: The method returns a newly created 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.

15.2.10.3.14 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.

15.2.10.3.15 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 the *block* is not given, the behavior is implementation dependent.

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 of the class `String` S whose content is the sequence of the characters from c to d.

   3) Call the *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 the sequence of the characters from c to the last character of s. Call the *block* with this instance as the argument.

d) Return the receiver.

15.2.10.3.16 String#empty?

`empty?`

Visibility: public

Behavior:

a) If the receiver is empty, return `true`.

b) Otherwise, return `false`.

15.2.10.3.17 String#eql?

`eql?(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `String`:

   1) If the contents of the receiver and the *other* are the same, return `true`.

   2) Otherwise, return `false`.

b) If the *other* is not an instance of the class `String`, return `false`.

15.2.10.3.18 String#gsub

`gsub(` **args*, *&block* `)`

Visibility: public

Behavior:

a) If the length of the *args* is 0 or larger than 2, or the length of the *args* is 1 and the *block* is not given, raise a direct instance of the class `ArgumentError`.

b) Let P be the first element of the *args*. If P is not an instance of the class `Regexp`, or the length of the *args* is 2 and the last element of the *args* is not an instance of the class `String`, the behavior is implementation dependent.

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) Match the pattern of P against S at the offset n (see §15.2.15.3 and Step 15.2.15.4). 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 the *args* is 1:

      i) Call the *block* with a 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 implementation dependent.

2) Let *pre* be the pre-match 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 offset of the last character of *pre*.

3) If the length of the *args* is 1, append the content of V to L. If the length of the *args* is 2, append to L the content of the last element of the *args*.

4) Let *post* be the post-match of *M*. Let i be the offset 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 string whose content is the ith character of *S*.

II) Increment n by 1.

ii) Otherwise, replace n with 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.

15.2.10.3.19 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.

15.2.10.3.20 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 and only if S_1 and S_2 has the same content, the values of H_1 and H_2 shall be the same integer.

15.2.10.3.21 String#include?

`include?(` *obj* `)`

Visibility: public

Behavior:

a) If the *obj* is an instance of the class `Integer`:

If the receiver includes the character whose character code is the *obj*, return `true`. Otherwise, return `false`.

b) If the *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 *obj*, return `true`. Otherwise, return `false`.

c) Otherwise, the behavior is implementation dependent.

15.2.10.3.22 String#initialize

`initialize(` *str*`=""` `)`

Visibility: private

Behavior:

a) If the *str* is not an instance of the class `String`, the behavior is implementation dependent.

b) Otherwise, initialize the content of the receiver to the same sequence of characters as the content of the *str*.

c) Return an implementation defined value.

15.2.10.3.23 String#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class `String`, the behavior is implementation dependent.

b) Change the content of the receiver to the content of the *original*.

c) Return an implementation defined value.

15.2.10.3.24 String#intern

`intern`

Visibility: public

Behavior:

a) If the length of the receiver is 0, or the receiver contains 0x00, raise a direct instance of the class `ArgumentError`.

b) Otherwise, return a direct instance of the class `Symbol` whose name is the content of the receiver.

15.2.10.3.25 String#length

`length`

Visibility: public

Behavior: The method returns the number of characters of the content of the receiver.

15.2.10.3.26 String#match

`match(` *regexp* `)`

Visibility: public

Behavior:

a) If the *regexp* is an instance of the class `Regexp`, let R be the *regexp*.

b) If the *regexp* is an instance of the class `String`, create a direct instance of the class `Regexp` as if the method `new` is invoked on the class `Regexp` with the *regexp* as the argument. Let R be the instance of the class `Regexp`.

c) Otherwise, the behavior is implementation dependent.

d) Invoke the method `match` on R with the receiver as the argument.

e) Return the resulting value of the invocation.

15.2.10.3.27 String#replace

`replace(` *other* `)`

Visibility: public

Behavior: Same as the method `initialize_copy` (see §15.2.10.3.23).

15.2.10.3.28 String#reverse

`reverse`

Visibility: public

Behavior: The method returns a newly created instance of the class `String` which contains all the characters of the content of the receiver in the reverse order.

15.2.10.3.29 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.

15.2.10.3.30 String#scan

`scan(` *reg*, `&`*block* `)`

Visibility: public

Behavior:

a) If the *reg* is not an instance of the class `Regexp`, the behavior is implementation dependent.

b) If the *block* is not given, create an empty direct instance of the class `Array` A.

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) Match the pattern of the *reg* against S at the offset n (see §15.2.15.3 and Step 15.2.15.4). Let M be the result of the matching process.

f) If M is not `nil`:

1) Let L be the match result of M.

2) If the length of L is 1, create a direct instance of the class `String` V whose content is the matched substring of M.

3) If the length of L is larger than 1:

i) Create an empty direct instance of the class `Array` V.

ii) Except for the first element, for each element e of L, in the same order in the list, append to V a direct instance of the class `String` whose content is the first element of e.

4) If the *block* is given, call the *block* with V as the argument. Otherwise, append V to A.

5) Let *post* be the post-match of M. Let i be the offset of the first character of *post* within S.

i) If i and n are the same, i.e. if the *reg* matched the empty string, increment n by 1.

ii) Otherwise, replace n with i.

6) If $n < l$, continue processing from Step e.

g) If the *block* is given, return the receiver. Otherwise, return A.

15.2.10.3.31 String#size

```
size
```

Visibility: public

Behavior: Same as the method `length` (see §15.2.10.3.25).

15.2.10.3.32 String#slice

```
slice(*args)
```

Visibility: public

Behavior: Same as the method `[]` (see §15.2.10.3.6).

15.2.10.3.33 String#split

```
split(sep)
```

Visibility: public

Behavior:

a) If the *sep* is not an instance of the class `Regexp`, the behavior is implementation dependent.

b) Create an empty direct instance of the class `Array` *A*.

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) Match the pattern of the *sep* against *S* at the offset *sp* (see §15.2.15.3 and Step 15.2.15.4). Let *M* be the result of the matching process.

f) If *M* is `nil`, append to *A* a 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 direct instance of the class `String` whose content is the *bp*th character of *S*.

      ii) Otherwise, increment *sp* by 1. If $sp < l$, replace *was-empty* with true and continue processing from Step e.

   2) Otherwise, replace *was-empty* with false. Let *pre* be the pre-match of *M*. Append to *A* an 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 offset of the last character of *pre*.

   3) Let *L* be the match result 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 first element of e.

ii) If c is not `nil`, append to A a 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 offset 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 the this condition does not hold.

i) return A.

15.2.10.3.34 String#sub

`sub(` `*`*args*, `&`*block* `)`

Visibility: public

Behavior:

a) If the length of the *args* is 1 and the *block* is given, or the length of the *args* is 2:

1) If the first element of the *args* is not an instance of the class `Regexp`, the behavior is implementation dependent.

2) Match the pattern of the first element of the *args* against the content of the receiver (see §15.2.15.3 and Step 15.2.15.4). 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 the *args* is 1, call the *block* with a 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 implementation dependent.

ii) If the length of the *args* is 2, let S be the last element of the *args*. If S is not an instance of the class `String`, the behavior is implementation dependent.

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`.

15.2.10.3.35 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.

15.2.10.3.36 String#upcase

`upcase`

Visibility: public

Behavior: The method returns a newly created 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.

15.2.10.3.37 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.

15.2.10.3.38 String#to_i

`to_i( base=10 )`

Visibility: public

Behavior:

a) If the *base* is not an instance of the class `Integer` whose value is 2, 8, 10, nor 16, the behavior is implementation dependent. Otherwise, let b be the value of the *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*.

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 implementation dependent.

i) Let n be the value of 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.

15.2.10.3.39 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 the sequence of the characters which is a *float-literal*, return a direct instance of the class `Float` whose value is the value of the *float-literal* (see §8.5.5.1).

c) If the receiver starts with the sequence of the characters which is a *digit-decimal-integer-literal*, return a direct instance of the class `Float` whose value is the value of the *digit-decimal-integer-literal* as a floating point number (see §8.5.5.1).

d) Otherwise, return a direct instance of the class `Float` whose value is implementation defined.

15.2.10.3.40 String#to_s

`to_s`

Visibility: public

Behavior: The method returns the receiver.

15.2.10.3.41 String#to_sym

`to_sym`

Visibility: public

Behavior: Same as the method `intern` (see §15.2.10.3.24).

15.2.11 Symbol

Instances of the class `Symbol` represent names (see §8.5.5.5). No two instances of the class `Symbol` have the same name.

Instances of the class `Symbol` shall not be created by the method `new` of the class `Symbol`. Therefore, a conforming processor shall undefine the singleton method `new` of the class `Symbol`, as if by invoking the method `undef_method` on the eigenclass of the class `Symbol` with a direct instance of the class `Symbol` whose name is "new" as the argument (see §15.2.2.3.42).

15.2.11.1 Direct superclass

The class `Object`

15.2.11.2 Instance methods

15.2.11.2.1 Symbol#===

`===(` *other* `)`

Visibility: public

Behavior: Same as the method `==` of the module `Kernel` (see §15.3.1.2.1).

15.2.11.2.2 Symbol#id2name

```
id2name
```

Visibility: public

Behavior: The method returns an instance of the class `String`, the content of which represents the name of the receiver.

15.2.11.2.3 Symbol#to_s

```
to_s
```

Visibility: public

Behavior: Same as the method `id2name` (see §15.2.11.2.2).

15.2.11.2.4 Symbol#to_sym

```
to_sym
```

Visibility: public

Behavior: The method returns the receiver.

15.2.12 Array

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` has their indexes counted up from 0.

Given an array A, operations ***append***, ***prepend***, ***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.

15.2.12.1 Direct superclass

The class `Object`

15.2.12.2 Included modules

The following module is included in the class `Array`.

- `Enumerable`

15.2.12.3 Singleton methods

15.2.12.3.1 Array.[]

`Array.[](` **items* `)`

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` which contains the elements of the *items*, preserving their order.

15.2.12.4 Instance methods

15.2.12.4.1 Array#*

`*(` *num* `)`

Visibility: public

Behavior:

a) If the *num* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) If the value of the *num* is smaller than 0, raise a direct instance of the class `ArgumentError`.

c) If the value of the *num* is 0, return an empty direct instance of the class `Array`.

d) Otherwise, create a direct instance of the class `Array` A and repeat the following for the num times:

Append all the elements of the receiver to A, preserving their order.

e) Return A.

15.2.12.4.2 Array#+

`+(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is an instance of the class `Array`, let A be the *other*. Otherwise, the behavior is implementation dependent.

b) Create an empty direct instance of the class `Array` R.

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.

15.2.12.4.3 Array#<<

`<<(` *obj* `)`

Visibility: public

Behavior: The method appends the *obj* to the receiver and return the receiver.

15.2.12.4.4 Array#[]

`[](` **args* `)`

Visibility: public

Behavior:

a) Let n be the length of the receiver.

b) If the length of the *args* is 0, raise a direct instance of the class `ArgumentError`.

c) If the length of the *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 implementation dependent.

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 the *args* is 2:

1) If the elements of the *args* are instances of the class `Integer`, let b and l be the values of the first and the last element of the *args*, respectively. Otherwise, the behavior is implementation dependent.

2) If $b < 0$, increment b by n. If b is still smaller than 0, return `nil`.

3) If $b = n$, create an empty direct instance of the class `Array` and return this instance.

4) If $b > n$ or $l < 0$, return `nil`.

5) If $l > n - b$, let new l be $n - b$.

6) Create an empty direct instance of the class `Array` A. Append the l elements of the receiver to A, from the bth index, preserving their order. Return A.

e) If the length of the *args* is larger than 2, raise a direct instance of the class `ArgumentError`.

15.2.12.4.5 Array#[]=

`[]=(` `*`*args* `)`

Visibility: public

Behavior:

a) Let n be the length of the receiver.

b) If the length of the *args* is smaller than 2, raise a direct instance of the class `ArgumentError`.

c) If the length of the *args* is 2:

1) If the first element of the *args* is an instance of the class `Integer`, let k be the value of the element and let V be the last element of the *args*. Otherwise, the behavior is implementation dependent.

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 the *args* is 3, the behavior is implementation dependent.

e) If the length of the *args* is larger than 3, raise a direct instance of the class `ArgumentError`.

15.2.12.4.6 Array#clear

`clear`

Visibility: public

Behavior: The method removes all the elements from the receiver and return the receiver.

15.2.12.4.7 Array#collect!

`collect!(`&*block*`)`

Visibility: public

Behavior:

a) If the *block* is given:

1) For each element of the receiver in the indexing order, call the *block* with the element as the only argument and replace the element with the resulting value.

2) Return the receiver.

b) If the *block* is not given, the behavior is implementation dependent.

15.2.12.4.8 Array#concat

`concat(` *other*`)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Array`, the behavior is implementation dependent.

b) Otherwise, append all the elements of the *other* to the receiver, preserving their order.

c) Return the receiver.

15.2.12.4.9 Array#each

`each(&block)`

Visibility: public

Behavior:

a) If the *block* is given:

1) For each element of the receiver in the indexing order, call the *block* with the element as the only argument.

2) Return the receiver.

b) If the *block* is not given, the behavior is implementation dependent.

15.2.12.4.10 Array#each_index

`each_index(&block)`

Visibility: public

Behavior:

a) If the *block* is given:

1) For each element of the receiver in the indexing order, call the *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 the *block* is not given, the behavior is implementation dependent.

15.2.12.4.11 Array#empty?

`empty?`

Visibility: public

Behavior:

a) If the receiver is empty, return `true`.

b) Otherwise, return `false`.

15.2.12.4.12 Array#first

`first(` *`*args`* `)`

Visibility: public

Behavior:

a) If the length of the *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 the *args* is 1:

1) If the only argument is not an instance of the class `Integer`, the behavior is implementation dependent. 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`.

15.2.12.4.13 Array#initialize

`initialize(` *size*`=`*0*, *obj*`=nil`, `&`*block* `)`

Visibility: private

Behavior:

a) If the *size* is not an instance of the class `Integer`, the behavior is implementation dependent. Otherwise, let n be the value of the *size*.

b) If n is smaller than 0, raise a direct instance of the class `ArgumentError`.

c) If n is 0, return an implementation defined value.

d) If n is larger than 0:

1) If the *block* is given:

i) Let k be 0.

ii) Call the 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 d-1-ii.

2) Otherwise, append the *obj* to the receiver n times.

3) Return an implementation defined value.

15.2.12.4.14 Array#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class `Array`, the behavior is implementation dependent.

b) Remove all the elements from the receiver.

c) Append all the elements of the *original* to the receiver, preserving their order.

d) Return an implementation defined value.

15.2.12.4.15 Array#join

`join(` *sep* `=nil )`

Visibility: public

Behavior:

a) If the *sep* is neither `nil` nor an instance of the class `String`, the behavior is implementation dependent.

b) Let S be an empty direct instance of the class `String`.

c) For each element X of the receiver, in the indexing order:

1) If the *sep* is not `nil`, and X is not the first element of the receiver, append the content of the *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 the content of the instance of the class `String` returned as if by the invocation of the method `join` on X with the *sep* as the argument.

4) Otherwise, the behavior is implementation dependent.

d) Return S.

15.2.12.4.16 Array#last

`last(`*`*args`*`)`

Visibility: public

Behavior:

a) If the length of the *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 the *args* is 1:

1) If the only argument is not an instance of the class `Integer`, the behavior is implementation dependent. 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`.

15.2.12.4.17 Array#length

`length`

Visibility: public

Behavior: The method returns the number of elements of the receiver.

15.2.12.4.18 Array#map!

`map!( &block )`

Visibility: public

Behavior: Same as the method `collect!` (see §15.2.12.4.7).

15.2.12.4.19 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.

15.2.12.4.20 Array#push

`push( *items )`

Visibility: public

Behavior:

a) For each element of the *items*, in the indexing order, append it to the receiver.

b) Return the receiver.

15.2.12.4.21 Array#replace

`replace( other )`

Visibility: public

Behavior: Same as the method `initialize_copy` (see §15.2.12.4.14).

15.2.12.4.22 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.

15.2.12.4.23 Array#reverse!

`reverse!`

Visibility: public

Behavior: The method reverses the order of the elements of the receiver and return the receiver.

15.2.12.4.24 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.

15.2.12.4.25 Array#size

`size`

Visibility: public

Behavior: Same as the method `length` (see §15.2.12.4.17).

15.2.12.4.26 Array#slice

`slice(` **args* `)`

Visibility: public

Behavior: Same as the method `[]` (see §15.2.12.4.4).

15.2.12.4.27 Array#unshift

`unshift( *items )`

Visibility: public

Behavior:

a) For each element of the *items*, in the reverse indexing order, prepend it to the receiver.

b) Return the receiver.

15.2.13 Hash

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`.

A hash cannot contain more than one key/value pair for each key.

An instance of the class `Hash` has the following property:

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 hash.

- A default proc, which is called to generate the return value of the method `[]` when the specified key is not found in the 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 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 true value.

- 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, an implementation shall 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 and K_2 are considered to be the same is implementation dependent.

Note that $K_1 \equiv K_2$ is not equivalent to $K_2 \equiv K_1$.

15.2.13.1 Direct superclass

The class `Object`

15.2.13.2 Included modules

The following module is included in the class `Hash`.

- `Enumerable`

15.2.13.3 Instance methods

15.2.13.3.1 Hash#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Hash`, the behavior is implementation dependent.

b) If all of the following conditions hold, return `true`:

- The receiver and the *other* have the same number of key/value pairs.
- For each key/value pair P in the receiver, the *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 results in a true value.

c) Otherwise, return `false`.

15.2.13.3.2 Hash#[]

`[](` *key* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where the *key* $\equiv$ the key of P, return the value of P.

b) Otherwise, invoke the method `default` on the receiver with the *key* as the argument and return the resulting value.

15.2.13.3.3 Hash#[]=

`[]=(` *key, value* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where the *key* $\equiv$ the key of P, replace the value of P with the *value*.

b) Otherwise:

   1) If the *key* is a direct instance of the class `String`, create a copy of the *key*, i.e. create a direct instance of the class `String` K whose content is the same as the *key*.

   2) If the *key* is not an instance of the class `String`, let K be the *key*.

   3) If the *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 the *value* into the receiver.

c) Return the *value*.

15.2.13.3.4 Hash#clear

`clear`

Visibility: public

Behavior:

a) Remove all the key/value pairs from the receiver.

b) Return the receiver.

15.2.13.3.5 Hash#default

```
default( *args )
```

Visibility: public

Behavior:

a) If the length of the *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 the *args* is 0, return `nil`.

  2) If the length of the *args* is 1, invoke the method `call` on the default proc of the receiver with two arguments, the receiver and the only element of the *args*. Return the resulting value of this invocation.

d) Otherwise, return `nil`.

15.2.13.3.6 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 the *value*.

c) Return the *value*.

15.2.13.3.7 Hash#default_proc

```
default_proc
```

Visibility: public

Behavior:

a) If the receiver has the default proc, return the default proc.

b) Otherwise, return `nil`.

15.2.13.3.8 Hash#delete

`delete(` *key*, `&`*block* `)`

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where the *key* $\equiv$ the key of P, remove P from the receiver and return the value of P.

b) Otherwise:

1) If the *block* is given, call the *block* with the *key* as the argument. Return the resulting value of this call.

2) Otherwise, return `nil`.

15.2.13.3.9 Hash#each

`each(` `&`*block* `)`

Visibility: public

Behavior:

a) If the *block* is given, call the *block* for each key/value pair of the receiver with an instance of the class `Array` as the argument, which contains two elements, the key and the value of that pair. The order of key/value pairs are implementation defined. Return the receiver.

b) If the *block* is not given, the behavior is implementation dependent.

15.2.13.3.10 Hash#each_key

`each_key(` `&`*block* `)`

Visibility: public

Behavior:

a) If the *block* is given, for each key/value pair of the receiver, in an implementation defined order, call the *block* with the key of the pair as the argument. Return the receiver.

b) If the *block* is not given, the behavior is implementation dependent.

15.2.13.3.11 Hash#each_value

```
each_value( &block )
```

Visibility: public

Behavior:

a) If the *block* is given, call the *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 the *block* is not given, the behavior is implementation dependent.

15.2.13.3.12 Hash#empty?

```
empty?
```

Visibility: public

Behavior:

a) If the receiver is empty, return `true`.

b) Otherwise, return `false`.

15.2.13.3.13 Hash#has_key?

```
has_key?( key )
```

Visibility: public

Behavior:

a) If the receiver has a key/value pair P where the *key* $\equiv$ the key of P, return `true`.

b) Otherwise, return `false`.

15.2.13.3.14 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 the *value* as the argument result in a true value.

b) Otherwise, return `false`.

15.2.13.3.15 Hash#include?

`include?(` *key* `)`

Visibility: public

Behavior: Same as the method `has_key?` (see §15.2.13.3.13).

15.2.13.3.16 Hash#initialize

`initialize(` `*`*args,* `&`*block* `)`

Visibility: private

Behavior:

a) If the *block* is given, and the length of the *args* is not 0, raise a direct instance of the class `ArgumentError`.

b) If the *block* is given and the length of the *args* is 0, set the default proc of the receiver to a direct instance of the class `Proc` which represents the *block*.

c) If the *block* is not given:

1) If the length of the *args* is 0, let D be `nil`.

2) If the length of the *args* is 1, let D be the only argument.

3) If the length of the *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.

15.2.13.3.17 Hash#initialize_copy

`initialize_copy( original )`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class `Hash`, the behavior is implementation dependent.

b) Remove all the key/value pairs from the receiver.

c) For each key/value pair P of the *original*, in an implementation defined order, store P in the receiver.

d) Remove the default value and the default proc from the receiver.

e) If the *orignal* has a default value, set the default value of the receiver to that value.

f) If the *orignal* has a default proc, set the default proc of the receiver to that proc.

g) Return an implementation defined value.

15.2.13.3.18 Hash#key?

`key?( key )`

Visibility: public

Behavior: Same as the method `has_key?` (see §15.2.13.3.13).

15.2.13.3.19 Hash#keys

`keys`

Visibility: public

Behavior: The method returns a newly created instance of the class `Array` whose content is the keys of the receiver. The order of the keys stored in somewhere are implementation defined.

15.2.13.3.20 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.

15.2.13.3.21 Hash#member?

`member?(` *key* `)`

Visibility: public

Behavior: Same as the method `has_key?` (see §15.2.13.3.13).

15.2.13.3.22 Hash#merge

`merge(` *other,* `&`*block* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Hash`, the behavior is implementation dependent.

b) Otherwise, create a direct instance of the class `Hash` H which has the same key/value pairs as the receiver.

c) For each key/value pair P of the *other*, in an implementation defined order:

   1) If the *block* is given:

      i) If H has the key/value pair Q where the key of $P \equiv$ the key of Q, call the *block* with three arguments, the key of P, the value of Q, and the value of P. Store the key/value pair whose key is the key of P and whose value is the resulting value of this call.

      ii) Otherwise, store P in H.

   2) If the *block* is not given, store P in H.

d) Return H.

15.2.13.3.23 Hash#replace

`replace(` *other* `)`

Visibility: public

Behavior: Same as the method `initialize_copy` (see §15.2.13.3.17).

15.2.13.3.24 Hash#shift

`shift`

Visibility: public

Behavior:

a) If the receiver has no key/value pairs:

   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.

15.2.13.3.25 Hash#size

`size`

Visibility: public

Behavior: Same as the method `length` (see §15.2.13.3.20).

15.2.13.3.26 Hash#store

`store(` *key, value* `)`

Visibility: public

Behavior: Same as the method `[]=` (see §15.2.13.3.3).

15.2.13.3.27 Hash#value?

`value?(` *value* `)`

Visibility: public

Behavior: Same as the method `has_value?` (see §15.2.13.3.14).

15.2.13.3.28 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 are implementation defined.

15.2.14 Range

Instances of the class `Range` represent ranges between two values, the start and end point.

An instance of the class `Range` has the following properties:

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 §15.3.1.2.8) or the method `dup` (see §15.3.1.2.9) of the class `Kernel` is invoked on an instance of the class `Range`, those properties shall be copied from the receiver to the resulting value.

15.2.14.1 Direct superclass

The class `Object`

15.2.14.2 Included modules

The following module is included in the class `Range`.

- `Enumerable`

15.2.14.3 Instance methods

15.2.14.3.1 Range#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If all of the following conditions hold, return `true`:

- the *other* is an instance of the class `Range`.

- Let S be the start point of the *other*. Invocation of the method `==` on the start point of the receiver with S as the argument results in a true value.

- Let E be the end point of the *other*. Invocation of the method `==` on the end point of the receiver with E as the argument results in a true value.

- The exclusive flag of the receiver and the one of the *other* are the same boolean value.

b) Otherwise, return `false`.

15.2.14.3.2 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 implementation dependent.

b) Invoke the method `<=>` on the start point of the receiver with the *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 implementation dependent.

   2) If the value of S is larger than 0, return `false`.

c) Invoke the method `<=>` on the *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 implementation dependent.

   - 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`.

15.2.14.3.3 Range#begin

`begin`

Visibility: public

Behavior: The method returns the start point of the receiver.

15.2.14.3.4 Range#each

`each( &block )`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

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 false value, 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 implementation dependent.

   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 the *block* with V as the argument, then, return the receiver.

e) Call the *block* with V as the argument.

f) Invoke the method `succ` on V with no argument, and let new V be the resulting value.

   Continue processing from Step d.

15.2.14.3.5 Range#end

`end`

Visibility: public

Behavior: The method returns the end point of the receiver.

15.2.14.3.6 Range#exclude_end?

```
exclude_end?
```

Visibility: public

Behavior: The method returns the exclusive flag of the receiver.

15.2.14.3.7 Range#first

```
first
```

Visibility: public

Behavior: Same as the method `begin` (see §15.2.14.3.3).

15.2.14.3.8 Range#include?

`include?(` *obj* `)`

Visibility: public

Behavior: Same as the method `===` (see §15.2.14.3.2).

15.2.14.3.9 Range#initialize

`initialize(` *left, right, exclusive* `=false )`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the *left* with the *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 implementation dependent.

b) If the *exclusive* is a true value, let F be `true`. Otherwise, let F be `false`.

c) Set the start point, end point, and exclusive flag of the receiver to the *left*, the *right*, and F, respectively.

d) Return an implementation defined value.

15.2.14.3.10 Range#last

```
last
```

Visibility: public

Behavior: Same as the method `end` (see §15.2.14.3.5).

15.2.14.3.11 Range#member?

```
member?( obj )
```

Visibility: public

Behavior: Same as the method `===` (see §15.2.14.3.2).

15.2.15 Regexp

Instances of the class `Regexp` represent regular expressions, and have the following properties.

pattern: A *pattern* of the regular expression (see §15.2.15.3). The default value of this property is empty.

If the value of this property 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 implementation dependent.

ignorecase: A boolean value which denotes whether a match is performed in the case insensitive manner. The default value of this property is false.

multiline: A boolean value which denotes whether the pattern "`.`" matches a *line-terminator* (see §15.2.15.3). The default value of this property is false.

15.2.15.1 Direct superclass

The class `Object`

15.2.15.2 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 and multiline properties of an instance of the class `Regexp` (see §15.2.15.6.1).

15.2.15.3 Patterns

Syntax

---

pattern ::
  alternative$_1$
  | *pattern*$_1$ `|` *alternative*$_2$

alternative ::
  [empty]
  | *alternative*$_3$ *term*

term ::
  anchor
  | *atom*$_1$
  | *atom*$_2$ *quantifier*

anchor ::
  `^` | `$`

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-without-zero*

regexp-character-escape-sequence ::
    regexp-escape-sequence
  | *regexp-non-escaped-sequence*
  | *hex-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` | `b`

regexp-non-escaped-sequence ::
    `\` *regexp-non-escaped-character*

regexp-non-escaped-character ::
    source-character **but not** *regexp-escaped-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 *pattern* matches the following string:

a) If the *pattern* is an *alternative*$_1$, it matches the string which the *alternative*$_1$ matches.

b) If the *pattern* is a *pattern*$_1$ `|` *alternative*$_2$, it matches the string which either the *pattern*$_1$ or the *alternative*$_2$ matches.

An *alternative* matches the following string:

a) If the *alternative* is [empty], it matches an empty string.

b) If the *alternative* is an *alternative*$_3$ *term*, it matches the concatenation of two strings, the one which the *alternative*$_3$ matches and the other which the *term* matches, in this order.

A *term* matches the following string:

a) If the *term* is an *atom*$_1$, it matches the string which the *atom*$_1$ matches.

b) If the *term* is an *atom*$_2$ *quantifier*, it matches a string as follows:

 1) If the *quantifier* is `*`, it matches zero or more strings which the *atom*$_2$ matches.

 2) If the *quantifier* is `+`, it matches one or more strings which the *atom*$_2$ matches.

 3) If the *quantifier* is `?`, it matches at most one string which the *atom*$_2$ matches.

c) If the *term* is an *anchor*, it matches the position within the string S which the *pattern* is matched against, as follows:

 1) If the *anchor* is `^`, it matches the beginning of S or the position just after a *line-terminator* which is followed by at least one character.

 2) If the *anchor* is `$`, it matches the end of S or the position just before a *line-terminator*.

An *atom* matches the following string:

a) If the *atom* is a *pattern-character*, it matches a single character C represented by the *pattern-character*. If the *atom* occurs in the pattern of an instance of the class `Regexp` whose ignorecase property is true, it also matches a corresponding uppercase character of C, if C is a lowercase character, or a corresponding lowercase character of C, if C is an uppercase character.

b) If the *atom* is a *grouping*, it matches the string which the *grouping* matches.

c) If the *atom* is "`.`", it matches any character except for a *line-terminator*. If the *atom* occurs in the pattern of an instance of the class `Regexp` whose multiline property is true, it also matches a *line-terminator*.

d) If the *atom* is an *atom-escape-sequence*, it matches the string which the *atom-escape-sequence* matches.

A *grouping* matches the string which the *pattern* matches.

An *atom-escape-sequence* matches the following string:

a) If the *atom-escape-sequence* is a *decimal-escape-sequence*, it matches the string which the *decimal-escape-sequence* matches.

b) 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 which the *regexp-character-escape-sequence* represents.

A *decimal-escape-sequence* matches the following string:

a) Let i be an integer represented by *decimal-digit-without-zero*.

b) 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*.

c) If the *decimal-escape-sequence* occurs before G within the *pattern*, it does not match any string.

d) If G matches any string, the *decimal-escape-sequence* matches the same string.

e) Otherwise, the *decimal-escape-sequence* does not match any string.

A *regexp-character-escape-sequence* represents a character as follows:

- A *regexp-escape-sequence* represents a character as shown in Table 1 in §8.5.5.2.2.
- A *regexp-non-escaped-sequence* represents a *regexp-non-escaped–character*.
- A *hex-escape-sequence* represents a character as described in §8.5.5.2.2.
- A *regexp-octal-escape-sequence* is interpreted in the same way as an *octal-escape-sequence* (see §8.5.5.2.2).
- 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.

15.2.15.4 Matching process

A *pattern* P is considered to successfully match the given string S, if there exists a substring of S (including S itself) which P matches, satisfying the following conditions.

a) If P is of the form *pattern*$_1$ `|` *alternative*$_2$, and if both the *pattern*$_1$ and the *alternative*$_2$ matches some substrings, the matched substring is the one among the substrings matched by the *pattern*$_1$ which meets conditions b and c as follows.

b) If P matches more than one substrings, the substring which begins earliest in S is the matched substring. This condition takes precedence over the condition c.

c) If there are more than one substrings which satisfy the condition b, the longest one is the matched substring.

These conditions are applied to any substring matched by any sub-pattern of P as much as possible in the way that the resulting substring matched by P still satisfies the conditions a, b and c.

When a numerical offset is specified, P is matched against the part of S which begins at the offset. Note, however, that if the match succeeds, the string property of the resulting instance of the class `MatchData` is S, not the part of S which begins at the offset, as described below.

A matching process returns either an instance of the class `MatchData` (see §15.2.16) if the match succeeds or `nil` if the match failed.

An instance of the class `MatchData` is created as follows:

a) Let B be the substring of S which P matched.

b) Create a direct instance of the class `MatchData`, and let M be the instance.

c) Set the string of M to S.

d) Create a new empty list L.

e) Let O be an ordered pair whose first element is B and whose second element is the offset of the first character of B within S, counted from 0. Append O to L.

f) For each *grouping* G in P, in the order of the occurrence of its "(" within P, take the following steps:

   1) If G contributed to the match of P, let B be the substring which G matched. Let O be an ordered pair whose first element is B and whose second element is the offset of the first character of B within S, counted from 0. Append O to L.

   2) Otherwise, append to L an ordered pair whose elements are both `nil`.

g) Set the match result of M to L.

h) M is the instance of the class `MatchData` returned by the matching process.

A matching process creates or updates a local variable binding with name "~", which is specifically used by the method `Regexp.last_match` (see §15.2.15.5.3), as follows:

a) Let M be the value which the matching process returns.

b) If the binding for the name "~" can be resolved by the process described in §9.1.1 as if "~" were a *local-variable-identifier*, replace the value of the binding with M.

c) 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*.

A conforming processor may name the binding other than "~"; however, it shall not be of the form *local-variable-identifier*.

15.2.15.5 Singleton methods

15.2.15.5.1 Regexp.compile

```
Regexp.compile( *args )
```

Visibility: public

Behavior: Same as the method `new` (see §15.2.3.2.3).

15.2.15.5.2 Regexp.escape

`Regexp.escape(` *string* `)`

Visibility: public

Behavior:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Let S be the content of the *string*.

c) Return a new instance of the class `String` whose content is same as S, except that every occurrences of characters on the left of Table 3 are replaced with the corresponding sequences of characters on the right of the Table 3.

15.2.15.5.3 Regexp.last_match

`Regexp.last_match(*`*index*`)`

Visibility: public

Behavior:

a) Search for a binding of a local variable with name “~” as described in §9.1.1 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 the *index* is 0, return M.

d) If the length of the *index* is larger than 1, raise a direct instance of the class `ArgumentError`.

e) If the length of the *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 implementation dependent.

g) Let R be the result returned as if by invoking the method `[]` on M with A as the only argument (see §15.2.16.2.1).

h) Return R.

Table 3 – Regexp escaped characters

Characters replaced	Escaped sequence
0x0a	`\n`
0x09	`\t`
0x0d	`\r`
0x0c	`\f`
0x20	\0x20
`#`	`\#`
`$`	`\$`
`(`	`\(`
`)`	`\)`
`*`	`\*`
`+`	`\+`
`-`	`\-`
`.`	`\.`
`?`	`\?`
`[`	`\[`
`\`	`\\`
`]`	`\]`
^	\^
{	\{
|	\\|
}	\}

15.2.15.5.4 Regexp.quote

```
Regexp.quote
```

Visibility: public

Behavior: Same as the method `escape` (see §15.2.15.5.2).

15.2.15.6 Instance methods

15.2.15.6.1 Regexp#initialize

`initialize(` *source*, *flag* `)`

Visibility: private

Behavior:

a) If the *source* is an instance of the class `Regexp`, let S be the pattern of the *source*. If the *source* is an instance of the class `String`, let S be the content of the *source*. Otherwise, the behavior is implementation dependent.

b) If S cannot be derived from the *pattern* (§15.2.15.3), raise a direct instance of the class `RegexpError`.

c) Set the pattern of the receiver to S.

d) If the *flag* is an instance of the class `Integer`, let n be its value.

   1) If computing bitwise AND of the value of `Regexp::IGNORECASE` and n results in non-zero value, set the ignorecase property of the receiver to true.

   2) If computing bitwise AND of the value of `Regexp::MULTILINE` and n results in non-zero value, set the multiline property of the receiver to true.

e) If the *flag* is true value other than an instance of the class `Integer`, set the ignorecase property of the receiver to true.

f) Return an implementation defined value.

15.2.15.6.2 Regexp#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class of the receiver, raise a direct instance of the class `TypeError`.

b) Set the pattern of the receiver to the patten of the *original*.

c) Set the ignorecase property of the receiver to the ignorecase property of the *orginal*.

d) Set the multiline property of the receiver to the multiline property of the *orginal*.

e) Return an implementation defined value.

15.2.15.6.3 Regexp#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Regexp`, return `false`.

b) If the corresponding properties of the receiver and the *other* are the same, return `true`.

c) Otherwise, return `false`.

15.2.15.6.4 Regexp#===

`===(` *string* `)`

Visibility: public

Behavior:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Let S be the content of the *string*.

c) Match the pattern of the receiver against S (see §15.2.15.3 and Step 15.2.15.4). 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`.

15.2.15.6.5 Regexp#=~

`=~(` *string* `)`

Visibility: public

Behavior:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Let S be the content of the *string*.

c) Match the pattern of the receiver against S (see §15.2.15.3 and Step 15.2.15.4). 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 property of M, and let i be the second element of P.

f) Return an instance of the class `Integer` whose value is i.

15.2.15.6.6 Regexp#casefold?

```
casefold?
```

Visibility: public

Behavior: The method returns the value of the ignorecase property of the receiver.

15.2.15.6.7 Regexp#match

`match(` *string* `)`

Visibility: public

Behavior:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Let S be the content of the *string*.

c) Match the pattern of the receiver against S (see §15.2.15.3 and Step 15.2.15.4). Let M be the result of the matching process.

d) Return M.

15.2.15.6.8 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

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 properties called ***string*** and ***match result***, which are initialized as described in §15.2.15.4. Elements of the match result are indexed by integers starting from 0.

Given an instance of the class `MatchData` M, the ***matched substring***, ***pre-match*** and ***post-match*** of M are defined as follows:

Let S be the string of M. Let F be the first element of the match result of M. Let B and O be the first portion (the substring matched) and the second portion (offset of that 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.

15.2.16.1 Direct superclass

The class `Object`

15.2.16.2 Instance methods

15.2.16.2.1 MatchData#[]

`[]( *`*args*`)`

Visibility: public

Behavior: The method behaves as if the method `to_a` were invoked on the receiver (see §15.2.16.2.12), and then, the method `[]` were invoked on the resulting instance of the class `Array` with the same arguments passed to an invocation of this method (see §15.2.12.4.4).

15.2.16.2.2 MatchData#begin

`begin(` *index*`)`

Visibility: public

Behavior:

a) If the *index* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) Let L be the match result of the receiver, and let i be the value of the *index*.

c) If i is smaller than 0 or equal to, or larger than 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.

15.2.16.2.3 MatchData#captures

`captures`

Visibility: public

Behavior:

a) Let L be the match result of the receiver.

b) Create an empty direct instance of the class `Array` A.

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 first portion of e.

d) Return A.

15.2.16.2.4 MatchData#end

`end(` *index* `)`

Visibility: public

Behavior:

a) If the *index* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) Let L be the match result of the receiver, and let i be the value of the *index*.

c) If i is smaller than 0 or equal to, or larger than the number of elements of L, raise a direct instance of the class `IndexError`.

d) Let F and S be the first and the second portions of the ith element of L.

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.

15.2.16.2.5 MatchData#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class of the receiver, raise a direct instance of the class `TypeError`.

b) Set the string property of the receiver to the string property of the *original*.

c) Set the match result property of the receiver to the match result property of the *original*.

d) Return an implementation defined value.

15.2.16.2.6 MatchData#length

`length`

Visibility: public

Behavior: The method returns the number of elements of the match result of the receiver.

15.2.16.2.7 MatchData#offset

`offset(` *index* `)`

Visibility: public

Behavior:

a) If the *index* is not an instance of the class `Integer`, the behavior is implementation dependent.

b) Let L be the match result of the receiver, and let i be the value of the *index*.

c) If i is smaller than 0 or equal to, or larger than the number of elements of L, raise a direct instance of the class `IndexError`.

d) Let S and b be the first and the second portions of the ith element of L. 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.

15.2.16.2.8 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.

15.2.16.2.9 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.

15.2.16.2.10 MatchData#size

```
size
```

Visibility: public

Behavior: Same as the method `length` (see §15.2.16.2.6).

15.2.16.2.11 MatchData#string

```
string
```

Visibility: public

Behavior: The method returns an instance of the class `String` the content of which is the string of the receiver.

15.2.16.2.12 MatchData#to_a

```
to_a
```

Visibility: public

Behavior:

a) Let L be the match result of the receiver.

b) Create an empty direct instance of the class `Array` A.

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 first portion of e.

d) Return A.

15.2.16.2.13 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

Instances of the class `Proc` represent *block*s.

An instance of the class `Proc` has the following property.

block: The block represented by the instance.

15.2.17.1 Direct superclass

The class `Object`

15.2.17.2 Singleton methods

15.2.17.2.1 Proc.new

`Proc.new(`&*block*`)`

Visibility: public

Behavior:

a) If the *block* is given, let B be the *block*.

b) Otherwise:

   1) If the top of ⟦block⟧ is block-not-given, raise a direct instance of the class `ArgumentError`.

   2) Otherwise, let B be the top of ⟦block⟧.

c) Create a new instance of the class `Proc` which has B as its block.

d) Return the instance.

15.2.17.3 Instance methods

15.2.17.3.1 Proc#[]

`[](`**args*`)`

Visibility: public

Behavior: Same as the method `call` (see §15.2.17.3.3).

15.2.17.3.2 Proc#arity

`arity`

Visibility: public

Behavior: Let B be the block represented by the receiver.

a) If a *block-formal-argument* does not occur in B, return an instance of the class `Integer` whose value is implementation defined.

b) If a *block-formal-argument* occurs in B:

  1) If a *block-formal-argument-list* does not occur in the *block-formal-argument*, return an instance of the class `Integer` whose value is 0.

  2) If a *block-formal-argument-list* occurs in the *block-formal-argument*:

    i) If the *block-formal-argument-list* is of the form *left-hand-side*, return an instance of the class `Integer` whose value is 1.

    ii) If the *block-formal-argument-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 `-1`.

      III) Otherwise, let n be the number of *multiple-left-hand-side-item*s 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.

15.2.17.3.3 Proc#call

`call(`*`*args`*`)`

Visibility: public

Behavior: Let B be the block of the receiver. Let L be an empty list.

a) Append each element of the *args*, in the indexing order, to L.

b) Call B with L as the arguments (see §11.2.2). Let V be the result of the call.

c) Return V.

15.2.17.3.4 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 an eigenclass, let E_o be the eigenclass, and take the following steps:

1) Create an eigenclass whose direct superclass is the direct superclass of E_o. Let E_n be the eigenclass.

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 of O to the block of the receiver.

e) Return O.

15.2.17.3.5 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 of O to the block of the receiver.

c) Return O.

15.2.18 Struct

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 §15.2.18.2.1). An instance of a generated class has values for the set of fields. Those values can be referenced and updated with accessor methods for their fields.

15.2.18.1 Direct superclass

The class `Object`

15.2.18.2 Singleton methods

15.2.18.2.1 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 the *string* is not an instance of the class `String` or the class `Symbol`, the behavior is implementation dependent.

c) If the *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` error.

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 the *string* is an instance of the class `Symbol`, prepend the instance to the *symbol_list*.

e) Let i be 0.

f) For each element S of the *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 value of the field named N to the given argument and returns the argument.

4) Increment i by 1.

g) Return C.

Singleton methods of classes created by the Struct.new

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 the *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 of the class `Array` which contains instances of the class `String`, each of which represents a field name of the receiver. Let A be the instance of the class `Array`.

The elements in A are arranged in the indexing order of corresponding fields.

b) Return A.

15.2.18.3 Instance methods

15.2.18.3.1 Struct#==

==(*other*)

Visibility: public

Behavior:

a) If the *other* and the receiver is the same object, return `true`.

b) If the class of the *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 the *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 false value, return `false`.

d) Return `true`.

15.2.18.3.2 Struct#[]

[](*name*)

Visibility: public

Behavior:

a) If the *name* is an instance of the class `Symbol` or the class `String`:

1) Let N be the name designated by the *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 property.

b) If the *name* is an instance of the class `Integer`, let i be the value of the *name*. Let n be the number of the fields of the receiver.

1) If i is negative, replace i with $n + i$.

2) If i is still negative or i equal or larger than 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 implementation dependent.

15.2.18.3.3 Struct#[]=

`[]=(` *name*, *obj* `)`

Visibility: public

Behavior:

a) If the *name* is an instance of the class `Symbol` or an instance of the class `String`:

1) Let N be the name designated by the *name*.

2) If the receiver has the field named N,

i) Replace the value of the field with the *obj*,

ii) Return the *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 property.

b) If the *string* is an instance of the class `Integer`, let i be the value of the *name*. Let n be the number of the fields of the receiver.

1) If i is negative, replace i with $n + i$.

2) If i is still negative or i equal or larger than n, raise a direct instance of the class `IndexError`.

3) Otherwise,

i) Replace the value of the field whose index is i with the *obj*

ii) Return the *obj*.

c) Otherwise, the behavior of the method is implementation dependent.

15.2.18.3.4 Struct#each

`each( &block )`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) For each field of the receiver, in the indexing order, call the *block* with the value of the field as the only argument.

c) Return the receiver.

15.2.18.3.5 Struct#each_pair

`each_pair( &block )`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

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 the *block* with the list of arguments which contains S and V in this order.

c) Return the receiver.

15.2.18.3.6 Struct#members

```
members
```

Visibility: public

Behavior: Same as the method `members` described in §15.2.18.2.1.

15.2.18.3.7 Struct#select

`select(&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Create an empty instance 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 the *block* with V as the only argument. Let R be the resulting value of the call.

   3) If R is a true value, append V to A.

d) Return A.

15.2.18.3.8 Struct#initialize

`initialize(*`*args*`)`

Visibility: private

Behavior: Let N_a be the length of the *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 the *args*, or to `nil` when i is equal to or larger than N_a.

c) Return an implementation defined value.

15.2.18.3.9 Struct#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the receiver and the *original* are the same object, return an implementation defined value.

b) If the *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 the *original* are different, raise a direct instance of the class `TypeError`.

d) For each field f of the *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

Instances of the class `Time` represent dates and times.

An instance of the class `Time` holds the following data.

Microseconds: Microseconds since January 1, 1970 00:00 UTC. Microseconds is an integer whose range is implementation defined. 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.

Time zone: The time zone.

15.2.19.1 Direct superclass

The class `Object`

15.2.19.2 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.

15.2.19.2.1 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:

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

15.2.19.2.2 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.

$$DayWithinYear(t) = Day(t) - DayFromYear(YearFromTime(t))$$

15.2.19.2.3 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.

$$MonthFromTime(t) = \begin{cases} 1 & \text{if } 0 \leq DayWithinYear(t) < 31 \\ 2 & \text{if } 31 \leq DayWithinYear(t) < 59 \\ 3 & \text{if } 59 + LeapYear(t) \leq DayWithinYear(t) < 90 + LeapYear(t) \\ 4 & \text{if } 90 + LeapYear(t) \leq DayWithinYear(t) < 120 + LeapYear(t) \\ 5 & \text{if } 120 + LeapYear(t) \leq DayWithinYear(t) < 151 + LeapYear(t) \\ 6 & \text{if } 151 + LeapYear(t) \leq DayWithinYear(t) < 180 + LeapYear(t) \\ 7 & \text{if } 181 + LeapYear(t) \leq DayWithinYear(t) < 212 + LeapYear(t) \\ 8 & \text{if } 212 + LeapYear(t) \leq DayWithinYear(t) < 243 + LeapYear(t) \\ 9 & \text{if } 243 + LeapYear(t) \leq DayWithinYear(t) < 273 + LeapYear(t) \\ 10 & \text{if } 273 + LeapYear(t) \leq DayWithinYear(t) < 304 + LeapYear(t) \\ 11 & \text{if } 304 + LeapYear(t) \leq DayWithinYear(t) < 334 + LeapYear(t) \\ 12 & \text{if } 334 + LeapYear(t) \leq DayWithinYear(t) < 365 + LeapYear(t) \end{cases}$$

$$LeapYear(t) = \begin{cases} 1 & \text{if } YearFromTime(t) \text{ is a leap year} \\ 0 & \text{otherwise} \end{cases}$$

1 15.2.19.2.4 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.

$$DayWithinMonth(t) = \begin{cases} DayWithinYear(t) + 1 & \text{if } MonthFromTime(t) = 1 \\ DayWithinYear(t) - 30 & \text{if } MonthFromTime(t) = 2 \\ DayWithinYear(t) - 58 - LeapYear(t) & \text{if } MonthFromTime(t) = 3 \\ DayWithinYear(t) - 89 - LeapYear(t) & \text{if } MonthFromTime(t) = 4 \\ DayWithinYear(t) - 119 - LeapYear(t) & \text{if } MonthFromTime(t) = 5 \\ DayWithinYear(t) - 150 - LeapYear(t) & \text{if } MonthFromTime(t) = 6 \\ DayWithinYear(t) - 180 - LeapYear(t) & \text{if } MonthFromTime(t) = 7 \\ DayWithinYear(t) - 211 - LeapYear(t) & \text{if } MonthFromTime(t) = 8 \\ DayWithinYear(t) - 242 - LeapYear(t) & \text{if } MonthFromTime(t) = 9 \\ DayWithinYear(t) - 272 - LeapYear(t) & \text{if } MonthFromTime(t) = 10 \\ DayWithinYear(t) - 303 - LeapYear(t) & \text{if } MonthFromTime(t) = 11 \\ DayWithinYear(t) - 333 - LeapYear(t) & \text{if } MonthFromTime(t) = 12 \end{cases}$$

2 15.2.19.2.5 Hours, Minutes, and Seconds

The number of microseconds in an hour, a minute, 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.

$$HourFromTime(t) = floor\left(\frac{t}{MicroSecPerHour}\right) \quad \text{modulo} \quad 24$$
$$MinuteFromTime(t) = floor\left(\frac{t}{MicroSecPerMinute}\right) \quad \text{modulo} \quad 60$$
$$SecondFromTime(t) = floor\left(\frac{t}{MicroSecPerSecond}\right) \quad \text{modulo} \quad 60$$

15.2.19.3 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 time zone, the time zone 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.

$$LocalTime = t + ZoneOffset(z)$$
$$ZoneOffset(z) = \text{UTC offset of } z \text{ measured in microseconds}$$

15.2.19.4 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 of an instance of the class `Time` if that microseconds falls within the daylight saving time of the time zone of the instance. An algorithm used for the adjustment is implementation defined.

15.2.19.5 Singleton methods

15.2.19.5.1 Time.at

`Time.at(*`*args*`)`

Visibility: public

Behavior:

a) If the length of the *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.

b) If the length of the *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) If A is an instance of the class `Integer`, let N_S be the value of A. Let N_M be 0.

ii) If A is an instance of the class `Float`, let F be the value of A. Let N_S be the largest integer such that $N_S \leq F$. Let N_M be an integer which is the result of computing $(F - N_S) \times 10^6$, rounded off the first decimal place.

iii) 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.

iv) Return the resulting instance.

3) Otherwise, the behavior is implementation dependent.

c) If the length of the *args* is 2, let S and M be the first and second element of the *args*.

1) i) If S is an instance of the class `Integer`, let N_S be the value of A.

ii) If S is an instance of the class `Float`, let F be the value of A. If F is positive, let N_M be the largest integer such that $N \leq F$. Otherwise, let N_M be the smallest integer such that $N \geq F$.

iii) Otherwise, the behavior is implementation dependent.

2) Compute an integer which corresponds to M in the same way as S as described in Step c-1-i and c-1-ii. Let N_M be the integer.

3) 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.

4) Return the resulting instance.

15.2.19.5.2 Time.gm

`Time.gm(` *year*, *month*`=1`, *day*`=0`, *hour*`=0`, *min*`=0`, *sec*`=0`, *usec*`=0` `)`

Visibility: public

Behavior:

a) Compute an integer value for the *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 implementation dependent.

4) Otherwise, the behavior is implementation dependent.

b) Compute an integer value from the *month* as follows:

1) If the *month* is not an instance of the class `String`, the behavior is implementation dependent.

2) If the content of the *month* is the same as one of the names of the months in the upper row on Table 4, ignoring the differences in case, let *Mon* be the integer which corresponds to the *month* in the lower row on the same table.

3) If the first character of the *month* is *decimal-digit*, compute an integer value from the *month* as in Step a. Let *Mon* be the resulting integer.

4) Otherwise, raise a direct instance of the class `ArgumentError`.

c) If Y is an integer such that $0 \leq Y \leq 38$, replace Y with $2000 + Y$.

d) If Y is an integer such that $69 \leq Y \leq 138$, replace Y with $1900 + Y$.

e) 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.

f) Let t be an integer which satisfies all of the following equations.

- $YearFromTime(t) = Y$
- $MonthFromTime(t) = Mon$
- $DayWithinMonth(t) = 1$

g) Compute microseconds T as follows.

$$T = t + D \times MicroSecPerDay + H \times MicroSecPerHour + \\ Min \times MicroSecPerMinute + S \times 10^6 + U$$

h) 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.

i) Return the resulting instance.

Table 4 – 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

15.2.19.5.3 Time.local

`Time.local(` *year*, *month*`=1`, *day*`=0`, *hour*`=0`, *min*`=0`, *sec*`=0`, *usec*`=0)`

Visibility: public

Behavior: Same as the method `Time.gm` (see §15.2.19.5.2), except that the method returns a direct instance of the class `Time` which has the current local time zone as its time zone.

15.2.19.5.4 Time.mktime

`Time.mktime(` *year*, *month*`=1`, *day*`=0`, *hour*`=0`, *min*`=0`, *sec*`=0`, *usec*`=0)`

Visibility: public

Behavior: Same as the method `Time.local` (see §15.2.19.5.3).

15.2.19.5.5 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 time zone.

The behavior of this method is the same as the method `new` (see §15.2.3.2.3).

15.2.19.5.6 Time.utc

```
Time.utc( year, month=1, day=0, hour=0, min=0, sec=0, usec=0 )
```

Visibility: public

Behavior: Same as the method `Time.gm` (see §15.2.19.5.2).

15.2.19.6 Instance methods

15.2.19.6.1 Time#+

```
+( offset )
```

Visibility: public

Behavior:

a) If the *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is implementation dependent.

b) Let V be the value of the *offset*.

c) Let o be an integer which is the result of computing $V \times 10^6$, rounded of the first decimal place, if any.

d) Let t and z be the microseconds and time zone 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.

15.2.19.6.2 Time#−

```
-( offset )
```

Visibility: public

Behavior:

a) If the *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is implementation dependent.

b) Let V be the value of the *offset*.

c) Let o be an integer which is the result of computing $V \times 10^6$, rounded of the first decimal place, if any.

d) Let t and z be the microseconds and time zone 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.

15.2.19.6.3 Time#<=>

`<=>(` *other* `)`

Visibility: public

Behavior:

a) If the *other* is not an instance of the class `Time`, return `nil`.

b) Otherwise, let T_r and T_o be microseconds of the receiver and the *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.

15.2.19.6.4 Time#asctime

`asctime`

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). Let t be the result.

b) Let W be the name of the day of the week in the second row on Table 5 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 4 which corresponds to $MonthFromTime(t)$ in the upper row on the same table.

d) Let D, Min, S, and Y be as follows:

$$D = DayWithinMonth(t)$$
$$H = HourFromTime(t)$$
$$M = MinuteFromTime(t)$$
$$S = SecondFromTime(t)$$
$$Y = YearFromTime(t)$$

e) Create a direct instance of the class `String`, the content of which is the following sequence of characters:

$W\ Mon\ D\ H{:}M{:}S\ Y$<*line-terminator*>

f) Return the resulting instance.

Table 5 – The names of the days of the week corresponding integer

0	1	2	3	4	5	6
Sun	Mon	Tue	Wed	Thu	Fly	Sat

15.2.19.6.5 Time#ctime

```
ctime
```

Visibility: public

Behavior: Same as the method `asctime` (see §15.2.19.6.4).

15.2.19.6.6 Time#day

```
day
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.7 Time#dst?

```
dst?
```

Visibility: public

Behavior: Let T and Z be the microseconds and time zone of the receiver.

a) If T falls within the daylight saving time of Z, return `true`.

b) Otherwise, return `false`.

15.2.19.6.8 Time#getgm

```
getgm
```

Visibility: public

Behavior: Same as the method `getutc` (see §15.2.19.6.10).

15.2.19.6.9 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.

15.2.19.6.10 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.

15.2.19.6.11 Time#gmt?

```
gmt?
```

Visibility: public

Behavior: Same as the method `utc?` (see §15.2.19.6.26).

15.2.19.6.12 Time#gmt_offset

```
gmt_offset
```

Visibility: public

Behavior: Same as the method `utc_offset` (see §15.2.19.6.27).

15.2.19.6.13 Time#gmtime

```
gmtime
```

Visibility: public

Behavior: Same as the method `utc` (see §15.2.19.6.25).

15.2.19.6.14 Time#gmtoff

```
gmtoff
```

Visibility: public

Behavior: Same as the method `utc_offset` (see §15.2.19.6.27).

15.2.19.6.15 Time#hour

```
hour
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.16 Time#localtime

```
localtime
```

Visibility: public

Behavior:

a) Change the time zone of the receiver to the current local time zone.

b) Return the receiver.

15.2.19.6.17 Time#mday

```
mday
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.18 Time#min

```
min
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.19 Time#mon

```
mon
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.20 Time#month

```
month
```

Visibility: public

Behavior: Same as the method `mon` (see §15.2.19.6.19).

15.2.19.6.21 Time#sec

```
sec
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.22 Time#to_f

```
to_f
```

Visibility: public

Behavior: Let t the microseconds of the receiver.

a) Compute $t/10^6$.

b) Create a direct instance of the class `Float` whose value is the result of Step a.

c) Return the resulting instance.

15.2.19.6.23 Time#to_i

```
to_i
```

Visibility: public

Behavior: Let t the microseconds 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.

15.2.19.6.24 Time#usec

```
usec
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). Let t be the result.

b) Compute t modulo 10^6.

c) Return the resulting instance.

15.2.19.6.25 Time#utc

```
utc
```

Visibility: public

Behavior:

a) Change the time zone of the receiver to UTC.

b) Return the receiver.

15.2.19.6.26 Time#utc?

```
utc?
```

Visibility: public

Behavior: Let Z be the time zone of the receiver.

a) If Z is UTC, return `true`.

b) Otherwise, return `false`.

15.2.19.6.27 Time#utc_offset

```
utc_offset
```

Visibility: public

Behavior: Let Z be the time zone 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.

15.2.19.6.28 Time#wday

```
wday
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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

15.2.19.6.29 Time#yday

```
yday
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.30 Time#year

```
year
```

Visibility: public

Behavior:

a) Compute the local time from the receiver (see §15.2.19.3). 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.

15.2.19.6.31 Time#zone

```
zone
```

Visibility: public

Behavior: Let Z be the time zone of the receiver.

a) Create a direct instance of the class `String`, the content of which represents Z. The exact content of the instance is implementation dependent.

b) Return the resulting instance.

15.2.19.6.32 Time#initialize

`initialize`

Visibility: private

Behavior:

a) Set the microseconds of the receiver to microseconds elapsed since January 1, 1970 00:00 UTC.

b) Set the time zone of the receiver to the current time zone.

c) Return an implementation defined value.

15.2.19.6.33 Time#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior:

a) If the *original* is not an instance of the class `Time`, raise a direct instance of the class `TypeError`.

b) Set the microseconds of the receiver to the microseconds of the *original*.

c) Set the time zone of the receiver to the time zone of the *original*.

d) Return an implementation defined value.

15.2.20 IO

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 properties:

readability flag: A boolean value which denotes 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 denotes 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 denotes 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. Reading from or writing to a stream which is not open raises an instance of the class `IOError`.

buffering flag: A boolean value which denotes whether the data to be written to the stream is buffered.

When this flag is true, a conforming processor may delay the output to the receiver until the instance methods `flush` or `close` is invoked.

A conforming processor may raise an instance of the class `SystemCallError` during the execution of instance methods of the class `IO`.

In the following description of the methods of the class `IO`, a ***byte*** means an integer from 0 to 255.

15.2.20.1 Direct superclass

The class `Object`

15.2.20.2 Included modules

The following module is included in the class `IO`.

- `Enumerable`

15.2.20.3 Singleton methods

15.2.20.3.1 IO.open

`IO.open(` `*`*args,* `&`*block* `)`

Visibility: public

Behavior:

a) Invoke the method `new` on the receiver with all the elements of the *args* as the arguments. Let *I* be the resulting value.

b) If the *block* is not given, return I.

c) Otherwise, call the *block* with I as the argument. Let V be the resulting value.

d) Invoke the method `close` on I with no arguments, even when an exception is raised and not handled in Step c.

e) Return V.

15.2.20.4 Instance methods

15.2.20.4.1 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 `nil`.

15.2.20.4.2 IO#closed?

`closed?`

Visibility: public

Behavior:

a) If the receiver is closed, return `true`.

b) Otherwise, return `false`.

15.2.20.4.3 IO#each

`each(&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

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) Call the *block* with an argument, a direct instance of the class `String` whose content is the sequence of characters read in Step d.

f) Continue processing from Step c.

15.2.20.4.4 IO#each_byte

---

each_byte(&*block*)

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

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 the *block* with an argument, an instance of the class `Integer` whose value is the byte.

Continue processing from Step c.

15.2.20.4.5 IO#each_line

---

each_line(&*block*)

Visibility: public

Behavior: Same as the method `each` (see §15.2.20.4.3).

15.2.20.4.6 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`.

15.2.20.4.7 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.

15.2.20.4.8 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.

15.2.20.4.9 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) Return a direct instance of the class `String` whose content is the sequence of characters read in Step c.

15.2.20.4.10 IO#initialize_copy

`initialize_copy(` *original* `)`

Visibility: private

Behavior: The behavior of the method is implementation dependent.

15.2.20.4.11 IO#print

`print( *`*args* `)`

Visibility: public

Behavior:

a) For each element of the *args* in the indexing order:

   1) If the element is `nil`, let V be an instance of the class `String` whose content is "nil".

      A conforming processor may let V be an empty instance of the class `String`.

   2) If the element is not `nil`, let V be the element.

   3) Invoke the method `write` on the receiver with V as the argument.

b) Return `nil`.

15.2.20.4.12 IO#putc

`putc(` *obj* `)`

Visibility: public

Behavior:

a) If the *obj* is not an instance of the class `Integer` or an instance of the class `String`, the behavior is implementation dependent. If the *obj* is an instance of the class `Integer` whose value is smaller than 0 or larger than 255, the behavior is implementation dependent.

b) If the *obj* is an instance of the class `Integer`, create a direct instance of the class `String` S whose content is a single character, whose character code is the integer represented by *obj*.

c) If the *obj* is an instance of the class `String`, create a direct instance of the class `String` S whose content is the first character of the *obj*.

d) Invoke the method `write` on the receiver with S as the argument.

e) Return the *obj*.

15.2.20.4.13 IO#puts

`puts(` *_args_ `)`

Visibility: public

Behavior:

a) If the length of the *args* is 0, Invoke the method `write` on the receiver with an argument, which is a direct instance of the class `String` whose content is a single character 0x0a.

b) Otherwise, for each element E of the *args* in the indexing order:

   1) If E is `nil`:

      i) let S be an instance of the class `String` whose content is "nil".

         A conforming processor may let S be an empty instance of the class `String`.

      ii) Invoke the method `write` on the receiver with S as the argument.

      iii) Invoke the method `write` on the receiver with an argument, which is an instance of the class `String` whose content is a single character 0x0a.

   2) 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.

   3) If E is an instance of the class `String`:

      i) Invoke the method `write` on the receiver with E as the argument.

      ii) If the last character of E is not 0x0a, invoke the method `write` on the receiver with an argument, which is an instance of the class `String` whose content is a single character 0x0a.

c) Return `nil`.

15.2.20.4.14 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 the *length* is `nil`, create an empty instance of the class `String` and return that instance.

2) If the *length* is not `nil`, return `nil`.

c) Otherwise:

1) If the *length* is `nil`, read characters from the receiver until the receiver reaches its end.

2) If the *length* is an instance of the class `Integer`, let N be the value of the *length*. Otherwise, the behavior is implementation dependent.

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) Return a direct instance of the class `String` whose content is the sequence of characters read in Step c.

15.2.20.4.15 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.

15.2.20.4.16 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) Return a direct instance of the class `String` whose content is the sequence of characters read in Step c.

15.2.20.4.17 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 of the class `Array` A.

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) Append a direct instance of the class `String` whose content is the sequence of characters read in Step d to A.

f) Continue processing from Step c.

15.2.20.4.18 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`.

15.2.20.4.19 IO#sync=

`sync=( bool )`

Visibility: public

Behavior:

a) If the receiver is closed, raise a direct instance of the class `IOError`.

b) If the *bool* is a true value, set the buffering flag of the receiver to false. If the *bool* is a false value, set the buffering flag of the receiver to true.

c) Return the *bool*.

15.2.20.4.20 IO#write

`write( str )`

Visibility: public

Behavior:

a) If the *str* is not an instance of the class `String`, the behavior is implementation dependent.

b) If the *str* is empty, return an instance of the class `Integer` whose value is 0.

c) If the receiver is not writable, raise a direct instance of the class `IOError`.

d) Write all the characters in the *str* to the stream which the receiver represents, preserving their order.

e) Return an instance of the class `Integer`, whose value is implementation defined.

15.2.21 File

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.

A ***path*** of a file is a sequence of characters which represents the location of the file. The correct syntax of paths is implementation defined.

An instance of the class `File` has the following property:

path: The path of the file.

15.2.21.1 Direct superclass

The class `IO`

15.2.21.2 Singleton methods

15.2.21.2.1 File.exist?

`File.exist?(` *path* `)`

Visibility: public

Behavior:

a) If the file specified by the *path* exists, return `true`.

b) Otherwise, return `false`.

15.2.21.3 Instance methods

15.2.21.3.1 File#initialize

`initialize(` *path, mode* `="r")`

Visibility: private

Behavior:

a) If the *path* is not an instance of the class `String`, the behavior is implementation dependent.

b) If the *mode* is not an instance of the class `String` whose content is a single character "r" or "w", the behavior is implementation dependent.

c) Open the file specified by the *path* in an implementation defined way, and associate it with the receiver.

d) Set the path of the receiver to the content of the *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 the *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 the *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.

15.2.21.3.2 File#path

```
path
```

Visibility: public

Behavior: The method returns a direct instance of the class `String` whose content is the path of the receiver.

15.2.22 Exception

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 property.

message: An object returned by the method `to_s` (see §15.2.22.4.3).

When the method `clone` (see §15.3.1.2.8) or the method `dup` (see §15.3.1.2.9) of the class `Kernel` is invoked on an instance of the class `Exception`, the message property shall be copied from the receiver to the resulting value.

15.2.22.1 Direct superclass

The class `Object`

15.2.22.2 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. A conforming processor shall raise instances of these built-in subclasses in various erroneous conditions as described in this document. The class hierarchy shown in Figure 1 is used to handle an exception (see §14).

15.2.22.3 Singleton methods

15.2.22.3.1 Exception.exception

`Exception.exception(` **args, &block* `)`

Visibility: public

Behavior: Same as the method `new` (see §15.2.3.2.3).

15.2.22.4 Instance methods

15.2.22.4.1 Exception#exception

Figure 1 – The exception class hierarchy

```
Exception
├── StandardError
│   ├── ArgumentError
│   ├── LocalJumpError
│   ├── RangeError
│   ├── RegexpError
│   ├── RuntimeError
│   ├── TypeError
│   ├── ZeroDivisionError
│   ├── NameError
│   │   └── NoMethodError
│   ├── IndexError
│   │   └── StopIteration
│   ├── IOError
│   │   └── EOFError
│   └── SystemCallError
└── ScriptError
    ├── SyntaxError
    └── LoadError
```

`exception(` **string* `)`

Visibility: public

Behavior:

a) If the length of the *string* is 0, return the receiver.

b) If the length of the *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 property of E to M.

    iii) Return E.

c) If the length of the *string* is larger than 1, raise a direct instance of the class `ArgumentError`.

15.2.22.4.2 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.

15.2.22.4.3 Exception#to_s

```
to_s
```

Visibility: public

Behavior:

a) Let M be the message property 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 implementation dependent.

d) Otherwise, return M.

15.2.22.4.4 Exception#initialize

initialize(*message* = nil)

Visibility: private

Behavior:

a) Set the message property of the receiver to the *message*.

b) Return an implementation defined value.

15.2.23 StandardError

Instances of the class `StandardError` represent standard errors, which can be handled in a *rescue-clause* without a *exception-class-list* (see §11.4.1.4.1).

15.2.23.1 Direct superclass

The class `Exception`

15.2.24 ArgumentError

Instances of the class `ArgumentError` represent argument errors.

15.2.24.1 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

Instances of the class `RangeError` represent range errors.

15.2.26.1 Direct superclass

The class `StandardError`

15.2.27 RegexpError

Instances of the class `RegexpError` represent regular expression errors.

15.2.27.1 Direct superclass

The class `StandardError`

15.2.28 RuntimeError

Instances of the class `RuntimeError` represent runtime errors, which are raised by the method `raise` of the class `Kernel` by default (see §15.3.1.1.13).

15.2.28.1 Direct superclass

The class `StandardError`

15.2.29 TypeError

Instances of the class `TypeError` represent type errors.

15.2.29.1 Direct superclass

The class `StandardError`

15.2.30 ZeroDivisionError

Instances of the class `ZeroDivisionError` represent zero division errors.

15.2.30.1 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 property.

name: The name a reference to which causes this exception to be raised.

When the method `clone` (see §15.3.1.2.8) or the method `dup` (see §15.3.1.2.9) of the class `Kernel` is invoked on an instance of the class `NameError`, the name property 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 property of the receiver.

15.2.31.2.2 NameError#initialize

`initialize(` *message* `=nil,` *name* `=nil )`

Visibility: public

Behavior:

a) Set the name property 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 *super-with-argument* were evaluated with the *message* as the *argument* 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 exists or which cannot be invoked.

Instances of the class `NoMethodError` have properties called ***name*** (see §15.2.31) and ***arguments***. The values of these properties are set in the method `initialize` (see §15.2.32.2.2).

When the method `clone` (see §15.3.1.2.8) or the method `dup` (see §15.3.1.2.9) of the class `Kernel` is invoked on an instance of the class `NoMethodError`, those properties 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 property of the receiver.

15.2.32.2.2 NoMethodError#initialize

`initialize(` *message* `=nil,` *name* `=nil,` *args* `=nil )`

Visibility: public

Behavior:

a) Set the arguments property 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

Instances of the class `IndexError` represent index errors.

15.2.33.1 Direct superclass

The class `StandardError`

15.2.34 StopIteration

Instances of the class `StopIteration` represent exceptions, which are raised to terminate the method `loop` of the class `Kernel` (see §15.3.1.1.8).

15.2.34.1 Direct superclass

The class `IndexError`

15.2.35 IOError

Instances of the class `IOError` represent input/output errors.

15.2.35.1 Direct superclass

The class `StandardError`

15.2.36 EOFError

Instances of the class `EOFError` represent errors which occur when a stream has reached its end.

15.2.36.1 Direct superclass

The class `IOError`

15.2.37 SystemCallError

Instances of the class `SystemCallError` represent errors which occur while invoking the instance methods of the class `IO`.

15.2.37.1 Direct superclass

The class `StandardError`

15.2.38 ScriptError

Instances of the class `ScriptError` represent programming errors.

15.2.38.1 Direct superclass

The class `Exception`

15.2.39 SyntaxError

Instances of the class `SyntaxError` represent syntax errors.

15.2.39.1 Direct superclass

The class `ScriptError`

15.2.40 LoadError

Instances of the class `LoadError` represent errors which occur while loading external programs (see §15.3.1.1.14).

15.2.40.1 Direct superclass

The class `ScriptError`

15.3 Built-in modules

15.3.1 Kernel

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`.

15.3.1.1 Singleton methods

15.3.1.1.1 Kernel.`

`Kernel.`(` *string* `)`

The method ` is invoked in the form described in §8.5.5.2.6.

Visibility: public

Behavior: The method ` executes an external command corresponding to the *string*. The external command executed by the method is implementation defined.

When the method is invoked, take the following steps:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Execute the command which corresponds to the content of the *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.

15.3.1.1.2 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`.

15.3.1.1.3 Kernel.eval

`Kernel.eval(` *string* `)`

Visibility: public

Behavior:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

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*. Let V be the resulting value of the evaluation.

d) Return V.

In Step c, the *string* is evaluated under the new local variable scope in which references to *local-variable-identifier*s are resolved in the same way as in scopes created by *block*s (see §9.1.1).

15.3.1.1.4 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 instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation defined.

15.3.1.1.5 Kernel.iterator?

```
Kernel.iterator?
```

Visibility: public

Behavior: Same as the method `Kernel.block_given?` (see §15.3.1.1.2).

15.3.1.1.6 Kernel.lambda

```
Kernel.lambda( &block )
```

Visibility: public

Behavior: The method creates an instance of the class `Proc` as `Proc.new` does (see §15.2.17.2.1). However, the way in which the *block* is evaluated differs from the one described in §11.2.2 except when the *block* is called by a *yield-expression*.

The differences are as follows.

a) Before Step d of §11.2.2, 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-formal-argument-list* is of the form *left-hand-side*, and if N_a is not 1, the behavior is implementation dependent.

   3) If the *block-formal-argument-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* does not occur, 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 implementation dependent.

b) In Step e of §11.2.2, 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* occurs, the value of the `lambda` invocation is the value of the *jump-argument*.

2) Otherwise, the value of the lambda invocation is `nil`.

15.3.1.1.7 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 §9.1.1.

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.

15.3.1.1.8 Kernel.loop

`Kernel.loop(` &*block* `)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Otherwise, repeat calling the *block*.

c) If a direct instance of the class `StopIteration` is raised and not handled in Step b, handle the exception and return `nil`.

15.3.1.1.9 Kernel.method_missing

`Kernel.method_missing(` *symbol,* `*`*args* `)`

Visibility: public

Behavior:

a) If the *symbol* is not an instance of the class `Symbol`, raise a direct instance of the class `ArgumentError`.

b) Otherwise, raise a direct instance of the class `NoMethodError` which has the *symbol* as its name property and the *args* as its arguments property. A conforming processor may raise a direct instance of the class `NameError` which has the *symbol* as its name property instead of `NoMethodError` if the method is invoked in Step e of §13.3.3 during an evaluation of a *local-variable-identifier* as a method invocation.

15.3.1.1.10 Kernel.p

`Kernel.p(` `*`*args* `)`

Visibility: public

Behavior:

a) For each element E of the *args*, in the indexing order, take the following steps:

   1) Invoke the method `inspect` 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 implementation dependent.

   3) Invoke the method `write` on `Object::STDOUT` with X as the argument.

   4) Invoke the method `write` on `Object::STDOUT` with an argument, which is a direct instance of the class `String` whose content is a single character 0x0a.

b) Return `nil`. A conforming processor may return the *args* instead of `nil`.

15.3.1.1.11 Kernel.print

`Kernel.print(` `*`*args* `)`

Visibility: public

Behavior: The method behaves as if the method `print` of the class `IO` (see §15.2.20.4.11) were invoked on `Object::STDOUT` with the same arguments.

15.3.1.1.12 Kernel.puts

`Kernel.puts(` `*`*args* `)`

Visibility: public

Behavior: The method behaves as if the method `puts` of the class `IO` (see §15.2.20.4.13) were invoked on `Object::STDOUT` with the same arguments.

15.3.1.1.13 Kernel.raise

`Kernel.raise(` *`*args`* `)`

Visibility: public

Behavior:

a) If the length of the *args* is larger than 2, the behavior is implementation dependent.

b) If the length of the *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 §14.2).

   2) Otherwise, invoke the method `new` on the class `RuntimeError` with no argument. Let E be the resulting value.

c) If the length of the *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 the *args* is 2, let F and S be the first and the second argument, respectively.

   1) Invoke `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.

15.3.1.1.14 Kernel.require

`Kernel.require(` *`string`* `)`

Visibility: public

Behavior: The method `require` evaluates the external program P corresponding to the *string*. The way in which P is determined from the *string* is implementation defined.

When the method is invoked, take the following steps:

a) If the *string* is not an instance of the class `String`, the behavior is implementation dependent.

b) Search for the external program P corresponding to the *string*.

c) If the program does not exist, raise a direct instance of the class `LoadError`.

d) If P cannot be derived from the *program* (§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-visibility⟧ contains only one visibility, which is the private visibility.

   4) All the other attributes of the execution context are empty logical stacks.

f) Evaluate P under 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 that the evaluation of P affects 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`.

15.3.1.2 Instance methods

15.3.1.2.1 Kernel#==

`==(` *other* `)`

Visibility: public

Behavior:

a) If the receiver and the *other* is the same object, return `true`.

b) Otherwise, return `false`.

15.3.1.2.2 Kernel#===

`===(` *other* `)`

Visibility: public

Behavior:

a) If the receiver and the *other* is the same object, return `true`.

b) Otherwise, invoke the method `==` on the receiver with the *other* as the only argument. Let V be the resulting value.

c) If V is a true value, return `true`. Otherwise, return `false`.

15.3.1.2.3 Kernel#__id__

```
__id__
```

Visibility: public

Behavior: Same as the method `object_id` (see §15.3.1.2.33).

15.3.1.2.4 Kernel#__send__

```
__send__( symbol, *args, &block )
```

Visibility: public

Behavior: Same as the method `send` (see §15.3.1.2.44).

15.3.1.2.5 Kernel#‘

```
‘( string )
```

The method ‘ is invoked in the form described in §8.5.5.2.6.

Visibility: private

Behavior: Same as the method Kernel.‘ (see §15.3.1.1.1).

15.3.1.2.6 Kernel#block_given?

```
block_given?
```

Visibility: private

Behavior: Same as the method `Kernel.block_given?` (see §15.3.1.1.2).

15.3.1.2.7 Kernel#class

```
class
```

Visibility: public

Behavior: The method returns the class of the receiver.

15.3.1.2.8 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 an eigenclass, let E_o be the eigenclass, and take the following steps:

   1) Create an eigenclass whose direct superclass is the direct superclass of E_o. Let E_n be the eigenclass.

   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.

15.3.1.2.9 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.

15.3.1.2.10 Kernel#eql?

`eql?(` *other* `)`

Visibility: public

Behavior: Same as the method `==` (see §15.3.1.2.1).

15.3.1.2.11 Kernel#equal?

`equal?(` *other* `)`

Visibility: public

Behavior: Same as the method `==` (see §15.3.1.2.1).

15.3.1.2.12 Kernel#eval

`eval(` *string* `)`

Visibility: private

Behavior: Same as the method `Kernel.eval` (see §15.3.1.1.3).

15.3.1.2.13 Kernel#extend

```
extend(*module_list)
```

Visibility: public

Behavior: Let R be the receiver of the method.

a) If the length of the *module_list* is 0, raise a direct instance of the class `ArgumentError`.

b) For each element A of the *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.

15.3.1.2.14 Kernel#global_variables

```
global_variables
```

Visibility: private

Behavior: Same as the method `Kernel.global_variables` (see §15.3.1.1.4).

15.3.1.2.15 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 values is same.

When a conforming processor overrides the method `eql?` (see §15.3.1.2.10), 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 true value, invocations of the method `hash` on the object and the argument return the instances of the class Integer with the same value.

15.3.1.2.16 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 §15.3.1.2.8) or the method `dup` (see §15.3.1.2.9).

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.

15.3.1.2.17 Kernel#inspect

```
inspect
```

Visibility: public

Behavior: The method returns an 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.

15.3.1.2.18 Kernel#instance_eval

```
instance_eval( string = nil, &block )
```

Visibility: public

Behavior:

a) If the receiver is an instance of one of the implementation defined set of classes as described in Step a of §13.4.2, or if the receiver is one of `nil`, `true`, or `false`, then the behavior is implementation dependent.

b) If the receiver is not associated with an eigenclass, create a new eigenclass. Let M be the newly created eigenclass.

c) If the receiver is associated with an eigenclass, let M be that eigenclass.

d) Take Step b through the last step of the method `class_eval` of the class `Module` (see §15.2.2.3.15).

15.3.1.2.19 Kernel#instance_of?

`instance_of?(` *module* `)`

Visibility: public

Behavior: Let C be the class of the receiver.

a) If the *module* is not an instance of the class `Class` or the class `Module`, raise a direct instance of the class `TypeError`.

b) If the *module* and C are the same object, return `true`.

c) Otherwise, return `false`.

15.3.1.2.20 Kernel#instance_variable_defined?

`instance_variable_defined?(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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`.

15.3.1.2.21 Kernel#instance_variable_get

`instance_variable_get(` *symbol* `)`

Visibility: public

Behavior:

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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`.

15.3.1.2.22 Kernel#instance_variable_set

`instance_variable_set(` *symbol, obj* `)`

Visibility: public

Behavior:

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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 the *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 the *obj*.

15.3.1.2.23 Kernel#instance_variables

`instance_variables`

Visibility: public

Behavior: The method returns a direct instance of the class `Array` which consists of names of all the instance variables of the receiver. These names are represented by instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is implementation defined.

15.3.1.2.24 Kernel#is_a?

`is_a?(` *module* `)`

Visibility: public

Behavior:

a) If the *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 the *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 eigenclass of the receiver are the same object.

d) If the *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`.

15.3.1.2.25 Kernel#iterator?

```
iterator?
```

Visibility: private

Behavior: Same as the method `Kernel.iterator?` (see §15.3.1.1.5).

15.3.1.2.26 Kernel#kind_of?

`kind_of?(` *module* `)`

Visibility: public

Behavior: Same as the method `is_a?` (see §15.3.1.2.24).

15.3.1.2.27 Kernel#lambda

`lambda(` &*block* `)`

Visibility: private

Behavior: Same as the method `Kernel.lambda` (see §15.3.1.1.6).

15.3.1.2.28 Kernel#local_variables

```
local_variables
```

Visibility: private

Behavior: Same as the method `Kernel.local_variables` (see §15.3.1.1.7).

15.3.1.2.29 Kernel#loop

```
loop( &block )
```

Visibility: private

Behavior: Same as the method `Kernel.loop` (see §15.3.1.1.8).

15.3.1.2.30 Kernel#method_missing

```
method_missing( symbol, *args )
```

Visibility: private

Behavior: Same as the method `Kernel.method_missing` (see §15.3.1.1.9).

15.3.1.2.31 Kernel#methods

```
methods( all=true )
```

Visibility: public

Behavior: Let C be the class of the receiver.

a) If the *all* is a true value, the method behaves as if the method `instance_methods` were invoked on C with no arguments (see §15.2.2.3.33).

b) If the *all* is a false value, the method behaves as if the method `singleton_methods` were invoked on the receiver with `false` as the only argument (see §15.3.1.2.45).

15.3.1.2.32 Kernel#nil?

```
nil?
```

Visibility: public

Behavior:

a) If the receiver is `nil`, return `true`.

b) Otherwise, return `false`.

15.3.1.2.33 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.

15.3.1.2.34 Kernel#p

```
p( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.p` (see §15.3.1.1.10).

15.3.1.2.35 Kernel#print

```
print( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.print` (see §15.3.1.1.11).

15.3.1.2.36 Kernel#private_methods

```
private_methods( all=true )
```

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array` A.

b) If the receiver is associated with an eigenclass, let C be the eigenclass.

c) 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 private, skip the next two steps.

2) Let S be either a direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which of the these classes of instance 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.

d) For each module M in included module list of C, take Step c, assuming that C in that step to be M.

e) Replace C with the class of the receiver, and take Step c.

f) If the *all* is a true value:

1) Take Step d.

2) Replace C with the direct superclass of current C.

3) If C is not `nil`, take Step c, and then, repeat from Step f-1.

g) Return A.

15.3.1.2.37 Kernel#protected_methods

`protected_methods(` *all*`=true )`

Visibility: public

Behavior: Same as the method `private_methods` (see §15.3.1.2.36), except that the method returns a direct instance of the class `Array` which contains names of protected methods.

15.3.1.2.38 Kernel#public_methods

`public_methods`

Visibility: public

Behavior: Same as the method `private_methods` (see §15.3.1.2.36), except that the method returns a direct instance of the class `Array` which contains names of public methods.

15.3.1.2.39 Kernel#puts

`puts( *`*args*` )`

Visibility: private

Behavior: Same as the method `Kernel.puts` (see §15.3.1.1.12).

15.3.1.2.40 Kernel#raise

```
raise( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.raise` (see §15.3.1.1.13).

15.3.1.2.41 Kernel#remove_instance_variable

```
remove_instance_variable( symbol )
```

Visibility: private

Behavior:

a) Let N be the name designated by the *symbol*.

b) If N is not of the form *instance-variable-identifier*, raise a direct instance of the class `NameError` which has the *symbol* as its name property.

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 the *symbol* as its name property.

15.3.1.2.42 Kernel#require

```
require( *args )
```

Visibility: private

Behavior: Same as the method `Kernel.require` (see §15.3.1.1.14).

15.3.1.2.43 Kernel#respond_to?

```
respond_to?( symbol, include_private=false )
```

Visibility: public

Behavior:

a) Let N be the name designated by the *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 the *include_private* is a true value, return `true`.

ii) Otherwise, return `false`.

3) Otherwise, return `true`.

d) Otherwise, return `false`.

15.3.1.2.44 Kernel#send

`send(` *symbol,* `*`*args,* `&`*block* `)`

Visibility: public

Behavior:

a) Let N be the name designated by the *symbol*.

b) Invoke the method named N on the receiver with the *args* as arguments and the *block* as the block, if any.

c) Return the resulting value of the invocation.

15.3.1.2.45 Kernel#singleton_methods

`singleton_methods(` *all*`=true )`

Visibility: public

Behavior: Let E be the eigenclass of the receiver.

a) Create an empty direct instance of the class `Array` A.

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 direct instance of the class `String` whose content is N or a direct instance of the class `Symbol` whose name is N. Which of the these classes of instance 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 the *all* is a true value, 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.

15.3.1.2.46 Kernel#to_s

`to_s`

Visibility: public

Behavior: The method returns an 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

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`.

15.3.2.1 Instance methods

15.3.2.1.1 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 the *block* is given, call the *block* with X as the argument.

If this call results in a false value, return `false`.

2) If the *block* is not given, and X is a false value, return `false`.

c) Return `true`.

15.3.2.1.2 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 the *block* is given, call the *block* with X as the argument.

If this call results in a true value, return `true`.

2) If the *block* is not given, and X is a true value, return `true`.

c) Return `false`.

15.3.2.1.3 Enumerable#collect

`collect(`&*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Create an empty direct instance of the class `Array` A.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call the *block* with X as the argument and append the resulting value to A.

e) Return A.

15.3.2.1.4 Enumerable#detect

`detect(`*ifnone*`=`*nil,* `&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, call the *block* with X as the argument. If this call results in a true value, return X.

d) Return the *ifnone*.

15.3.2.1.5 Enumerable#each_with_index

`each_with_index(&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Let i be 0.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields:

   1) Call the *block* with X and i as the arguments.

   2) Increase i by 1.

e) Return the receiver.

15.3.2.1.6 Enumerable#entries

`entries`

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array` A.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, append X to A.

d) Return A.

15.3.2.1.7 Enumerable#find

`find(` *ifnone*`=`*nil,* `&`*block*`)`

Visibility: public

Behavior: Same as the method `detect` (see §15.3.2.1.4).

15.3.2.1.8 Enumerable#find_all

`find_all(&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Create an empty direct instance of the class `Array` A.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call the *block* with X as the argument. If this call results in a true value, append the element to A.

e) Return A.

15.3.2.1.9 Enumerable#grep

`grep(` *pattern,* `&`*block*`)`

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array` A.

b) Invoke the method `each` on the receiver.

c) For each element X which `each` yields, invoke the method `===` on the *pattern* with X as the argument.

If this invocation results in a true value:

1) If the *block* is given, call the *block* with X as the argument and append the resulting value to A.

2) Otherwise, append X to A.

d) Return A.

15.3.2.1.10 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 the *obj* as the argument. If this invocation results in a true value, return `true`.

c) Return `false`.

15.3.2.1.11 Enumerable#inject

`inject(*`*args,* `&`*block*`)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) If the length of the *args* is 2, the behavior is implementation dependent. If the length of the *args* is smaller than 0 or 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 the *args* is 0, let V be X.

2) If X is the first element, and the length of the *args* is 1, call the *block* with two arguments, which are the only element of the *args* and X. Let V be the resulting value of this call.

3) If X is not the first element, call the *block* with V and X as the arguments. Let new V be the resulting value of this call.

e) Return V.

15.3.2.1.12 Enumerable#map

`map(` &*block* `)`

Visibility: public

Behavior: Same as the method `collect` (see §15.3.2.1.3).

15.3.2.1.13 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, if the *block* is given:

i) Call the *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 implementation dependent.

iii) If the value of D is larger than 0, let new V be X.

If the *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 implementation dependent.

iii) If the value of D is larger than 0, let new V be X.

d) Return V.

15.3.2.1.14 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, if the *block* is given:

      i) Call the *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 implementation dependent.

      iii) If the value of D is smaller than 0, let new V be X.

   If the *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 implementation dependent.

      iii) If the value of D is smaller than 0, let new V be X.

d) Return V.

15.3.2.1.15 Enumerable#member?

`member?(` *obj* `)`

Visibility: public

Behavior: Same as the method `include?` (see §15.3.2.1.10).

15.3.2.1.16 Enumerable#partition

`partition(&block)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Create two empty 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 the *block* with X as the argument.

If this call results in a true value, append X to T. If this call results in a false value, append X to F.

e) Return a newly created instance the class `Array`, which contains only T and F in this order.

15.3.2.1.17 Enumerable#reject

`reject(&block)`

Visibility: public

Behavior:

a) If the *block* is not given, the behavior is implementation dependent.

b) Create an empty direct instance of the class `Array` A.

c) Invoke the method `each` on the receiver.

d) For each element X which `each` yields, call the *block* with X as the argument. If this call results in a false value, append the element to A.

e) Return A.

15.3.2.1.18 Enumerable#select

`select(&block)`

Visibility: public

Behavior: Same as the method `find_all` (see §15.3.2.1.8).

15.3.2.1.19 Enumerable#sort

`sort( &block )`

Visibility: public

Behavior:

a) Create an empty direct instance of the class `Array` A.

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, all of the following conditions shall hold:

1) Let i and j be the index of E_i and E_j, respectively.

2) If the *block* is given:

i) Suppose the *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 implementation dependent.

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 the *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 implementation dependent.

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.

15.3.2.1.20 Enumerable#to_a

`to_a`

Visibility: public

Behavior: Same as the method `entries` (see §15.3.2.1.6).

15.3.3 Comparable

The module `Comparable` provides methods which compare the receiver and an argument using the method `<=>`.

15.3.3.1 Instance methods

15.3.3.1.1 Comparable#<

`<(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *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 implementation dependent.

c) If the value of I is smaller than 0, return `true`. Otherwise, return `false`.

15.3.3.1.2 Comparable#<=

`<=(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *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 implementation dependent.

c) If the value of I is smaller than or equal to 0, return `true`. Otherwise, return `false`.

15.3.3.1.3 Comparable#==

`==(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *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 implementation dependent.

c) If the value of I is 0, return `true`. Otherwise, return `false`.

15.3.3.1.4 Comparable#>

`>(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *other* as the argument. Let N be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is implementation dependent.

c) If the value of I is larger than 0, return `true`. Otherwise, return `false`.

15.3.3.1.5 Comparable#>=

`>=(` *other* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *other* as the argument. Let N be the resulting value of this invocation.

b) If I is not an instance of the class `Integer`, the behavior is implementation dependent.

c) If the value of I is larger than or equal to 0, return `true`. Otherwise, return `false`.

15.3.3.1.6 Comparable#between?

`between?(` *left, right* `)`

Visibility: public

Behavior:

a) Invoke the method `<=>` on the receiver with the *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 implementation dependent.

   2) If the value of I_1 is smaller than 0, return `false`.

b) Invoke the method `<=>` on the receiver with the *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 implementation dependent.

   2) If the value of I_2 is larger than 0, return `false`. Otherwise, return `true`.

Annex A
(informative)
Grammar Summary

A.1 Lexical structure

A.1.1 Source text

see §8.1

---

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

A.1.2 Line terminators

see §8.2

---

line-terminator ::
0x0d? 0x0a

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

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

A.1.3 Whitespace

see §8.3

---

whitespace ::
0x09 | 0x0b | 0x0c | 0x0d | 0x20 | \ 0x0d? 0x0a

A.1.4 Comments

see §8.4

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

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

comment-content ::
  line-content

line-content ::
  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*

line ::
  comment-content *line-terminator*

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

A.1.5 Tokens

see §8.5

token ::
  reserved-word
  | *identifier*
  | *punctuator*
  | *operator*
  | *literal*

A.1.5.1 Reserved words

see §8.5.1

```
reserved-word ::
      __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
```

A.1.5.2 Identifiers

see §8.5.2

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

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`

A.1.5.3 Punctuators

see §8.5.3

---

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

A.1.5.4 Operators

see §8.5.4

---

operator ::
operator-method-name
| *assignment-operator*

operator-method-name ::
  ^ | & | | | <=> | == | === | !~ | =~ | > | >= | < | <= | << | >> | +
  | - | * | / | % | ** | ~ | +@ | -@ | [] | []= | `

assignment-operator ::
  assignment-operator-name =

assignment-operator-name ::
  + | - | * | ** | / | ^ | % | << | >> | & | && | || | |

A.1.5.5 Literals

see §8.5.5

---

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

A.1.5.5.1 Numeric literals

see §8.5.5.1

---

numeric-literal ::
  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 ::
  digit-decimal-integer-literal
  | *prefixed-decimal-integer-literal*

digit-decimal-integer-literal ::
`0`
| *decimal-digit-without-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 ::
decimal-float-literal
| *exponent-float-literal*

decimal-float-literal ::
digit-decimal-integer-literal `.` *digit-decimal-part*

exponent-float-literal ::
base-part *exponent-part*

base-part ::
decimal-float-literal
| *digit-decimal-integer-literal*

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

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

decimal-digit-without-zero ::
`1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

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

binary-digit ::
    `0` | `1`

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

A.1.5.5.2 String literals

see §8.5.5.2

---

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

A.1.5.5.2.1 Single quoted strings

see §8.5.5.2.1

---

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

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

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

single-escape-character-sequence ::
    `\` *single-escaped-character*

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

single-escaped-character ::
    `'` | `\`

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

A.1.5.5.2.2 Double quoted strings

see §8.5.5.2.2

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

double-quoted-string-character ::
    source-character **but not** (`"` | `\`)
    | *double-escape-sequence*
    | *interpolated-character-sequence*

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

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

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

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

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

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

octal-escape-sequence ::
    `\` *octal-digit* (*octal-digit* *octal-digit*?)?

hex-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* `}`

A.1.5.5.2.3 Quoted non-expanded literal strings

see §8.5.5.2.3

---

quoted-non-expanded-literal-string ::
    `%q` *literal-beginning-delimiter* *non-expanded-literal-string** *literal-ending-delimiter*

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

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

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*

A.1.5.5.2.4 Quoted expanded literal strings

see §8.5.5.2.4

---

quoted-expanded-literal-string ::
    `%` `Q`? *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
    | *double-escape-sequence*
    | *interpolated-character-sequence*

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

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

alpha-numeric-character-or-separator ::
    whitespace
    | *line-terminator*
    | *uppercase-character*
    | *lowercase-character*
    | *decimal-digit*

literal-ending-delimiter ::
    [depending on the *literal-beginning-delimiter*]

matching-literal-beginning-delimiter ::
    `(` | `{` | `<` | `[`

A.1.5.5.2.5 Here documents

see §8.5.5.2.5

---

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 ::
    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 **but not** `'`

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

double-quoted-delimiter-identifier ::
  source-character **but not** "

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

command-quoted-delimiter-identifier ::
  source-character **but not** `

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*

A.1.5.5.2.6 External command execution

see §8.5.5.2.6

---

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

backquoted-external-command-execution ::
  ` *double-quoted-string-character** `

quoted-external-command-execution ::
  `%x` *literal-beginning-delimiter* *expanded-literal-string** *literal-ending-delimiter*

A.1.5.5.3 Array literals

see §8.5.5.3

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

non-escaped-array-item-character ::
  non-escaped-array-character
  | *matching-literal-delimiter*

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

matching-literal-delimiter ::
  `(` | `{` | `<` | `[` | `)` | `}` | `>` | `]`

non-expanded-array-escape-sequence ::
  non-expanded-literal-escape-sequence **but not** *escaped-quoted-array-item-separator*
  | *escaped-quoted-array-item-separator*

escaped-quoted-array-item-separator ::
  `\` *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
  | *expanded-array-escape-sequence*
  | *interpolated-character-sequence*

expanded-array-escape-sequence ::
  double-escape-sequence **but not** *escaped-quoted-array-item-separator*
  | *escaped-quoted-array-item-separator*

A.1.5.5.4 Regular expression literals

see §8.5.5.4

---

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** (`/` | `\`)
  | `\` `\`
  | *line-terminator-escape-sequence*
  | *interpolated-character-sequence*

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

A.2 Program structure

A.2.1 Program

see §10.1

---

program ::
    compound-statement

A.3 Expressions

A.3.1 Logical expressions

see §11.1

---

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

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

logical-NOT-expression ::=
    logical-NOT-with-method-invocation-without-parentheses
  | *logical-NOT-with-unary-expression*

logical-NOT-with-method-invocation-without-parentheses ::
    `!` *method-invocation-without-parentheses*

logical-NOT-with-unary-expression ::
    `!` *unary-expression*

keyword-AND-expression ::
    expression `and` *keyword-NOT-expression*

keyword-OR-expression ::
    expression `or` *keyword-NOT-expression*

logical-OR-expression ::
    logical-AND-expression
  | *logical-OR-expression* `||` *logical-AND-expression*

logical-AND-expression ::
    equality-expression
    | *logical-AND-expression* && *equality-expression*

A.3.2 Method invocation expressions

see §11.2

---

primary-method-invocation ::
    super-with-optional-argument
    | *indexing-method-invocation*
    | *method-only-identifier*
    | *method-identifier* ([no *whitespace* here] *argument-with-parentheses*)? *block*?
    | *primary-expression* [no *line-terminator* here]
        . *method-name* ([no *whitespace* here] *argument-with-parentheses*)? *block*?
    | *primary-expression* [no *line-terminator* here]
        :: *method-name* [no *whitespace* here] *argument-with-parentheses* *block*?
    | *primary-expression* [no *line-terminator* here] :: *method-name-without-constant*
    block?

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

optional-whitespace ::
    [*whitespace* here]

method-name-without-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*
    | *return-with-argument*
    | *break-with-argument*
    | *next-with-argument*

command ::
    super-with-argument
    | *yield-with-argument*
    | *method-identifier* *argument*
    | *primary-expression* [no *line-terminator* here] (. | ::) *method-name* *argument*

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

chained-method-invocation ::
  (`.` | `::`) *method-name*
  | (`.` | `::`) *method-name* [no *whitespace* here]
    [lookahead ∉ { `{` }] *argument-with-parentheses*

command-with-do-block ::
  super-with-argument-and-do-block
  | *method-identifier* *argument* *do-block*
  | *primary-expression* [no *line-terminator* here] (`.` | `::`) *method-name* *argument* *do-block*

A.3.2.1 Method arguments

see §11.2.1

---

indexing-argument-list ::
  command
  | *operator-expression-list* `,`?
  | *operator-expression-list* `,` *splatting-argument*
  | *association-list* `,`?
  | *splatting-argument*

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

operator-expression-list ::
  operator-expression (`,` *operator-expression*)*

argument-with-parentheses ::
  `()`
  | `(` *argument-in-parentheses* `)`
  | `(` *operator-expression-list* `,` *chained-command-with-do-block* `)`
  | `(` *chained-command-with-do-block* `)`

argument ::
  [no *line-terminator* here] [lookahead ∉ { `{` }] *optional-whitespace*?
    argument-in-parentheses

argument-in-parentheses ::
  command
  | (*operator-expression-list* | *association-list*)
    (`,` *splatting-argument*)? (`,` *block-argument*)?
  | *operator-expression-list* `,` *association-list*
    (`,` *splatting-argument*)? (`,` *block-argument*)?
  | *splatting-argument* (`,` *block-argument*)?
  | *block-argument*

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

A.3.2.2 Blocks

see §11.2.2

---

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

brace-block ::
    `{` *block-formal-argument*? *block-body* `}`

do-block ::
    `do` *block-formal-argument*? *block-body* `end`

block-formal-argument ::
    `| |`
  | `||`
  | `|` *block-formal-argument-list* `|`

block-formal-argument-list ::
    left-hand-side
  | *multiple-left-hand-side*

block-body ::
    compound-statement

A.3.2.3 The `super` expression

see §11.2.3

---

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

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

super-with-argument ::
    `super` *argument*

super-with-argument-and-do-block ::
`super` *argument* *do-block*

A.3.2.4 The `yield` expression

see §11.2.4

---

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 *whitespace* here] `(` *argument-in-parentheses* `)`

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

yield-with-argument ::
`yield` *argument*

A.3.3 Operator expressions

see §11.3

---

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

A.3.3.1 Assignments

see §11.3.1

---

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*

A.3.3.1.1 Single assignments

see §11.3.1.1

---

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*

A.3.3.1.1.1 Single variable assignments

see §11.3.1.1.1

---

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*

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

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

A.3.3.1.1.2 Single indexing assignments

see §11.3.1.1.2

---

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

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

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

A.3.3.1.1.3 Single method assignments

see §11.3.1.1.3

---

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*

A.3.3.1.2 Abbreviated assignments

see §11.3.1.2

---

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*

A.3.3.1.2.1 Abbreviated variable assignments

see §11.3.1.2.1

---

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

A.3.3.1.2.2 Abbreviated indexing assignments

see §11.3.1.2.2

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

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

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

A.3.3.1.2.3 Abbreviated method assignments

see §11.3.1.2.3

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*

A.3.3.1.3 Multiple assignments

see §11.3.1.3

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] `[` *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* `,`)+ *multiple-left-hand-side-item*?
  | (*multiple-left-hand-side-item* `,`)+ *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 (`,` *splatting-right-hand-side*)?
  | *splatting-right-hand-side*

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

A.3.3.1.4 Assignments with `rescue` modifiers

see §11.3.1.4

---

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

A.3.3.2 Unary operators

see §11.3.2

---

unary-minus-expression ::
    power-expression$_1$
  | `-` *power-expression*$_2$

unary-expression ::
    primary-expression
  | *logical-NOT-with-unary-expression*
  | `~` *unary-expression*$_1$
  | `+` *unary-expression*$_2$

A.3.3.2.1 The `defined?` expression

see §11.3.2.1

---

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

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

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

A.3.3.3 Binary operators

see §11.3.3

---

equality-expression ::
    relational-expression
  | *relational-expression* `<=>` *relational-expression*
  | *relational-expression* `==` *relational-expression*
  | *relational-expression* `===` *relational-expression*
  | *relational-expression* `!=` *relational-expression*
  | *relational-expression* `=~` *relational-expression*
  | *relational-expression* `!~` *relational-expression*

relational-expression ::
  bitwise-OR-expression
  | *relational-expression* `>` *bitwise-OR-expression*
  | *relational-expression* `>=` *bitwise-OR-expression*
  | *relational-expression* `<` *bitwise-OR-expression*
  | *relational-expression* `<=` *bitwise-OR-expression*

bitwise-OR-expression ::
  bitwise-AND-expression
  | *bitwise-OR-expression* `|` *bitwise-AND-expression*
  | *bitwise-OR-expression* `^` *bitwise-AND-expression*

bitwise-AND-expression ::
  bitwise-shift-expression
  | *bitwise-AND-expression* *whitespace-before-operator*? `&` *bitwise-shift-expression*

bitwise-shift-expression ::
  additive-expression
  | *bitwise-shift-expression* *whitespace-before-operator*? `<<` *additive-expression*
  | *bitwise-shift-expression* `>>` *additive-expression*

additive-expression ::
  multiplicative-expression
  | *additive-expression* *whitespace-before-operator*? `+` *multiplicative-expression*
  | *additive-expression* *whitespace-before-operator*? `-` *multiplicative-expression*

multiplicative-expression ::
  unary-minus-expression
  | *multiplicative-expression* *whitespace-before-operator*? `*` *unary-minus-expression*
  | *multiplicative-expression* *whitespace-before-operator*? `/` *unary-minus-expression*
  | *multiplicative-expression* *whitespace-before-operator*? `%` *unary-minus-expression*

power-expression ::
  unary-expression
  | `-` (*numeric-literal*) `**` *power-expression*
  | *unary-expression* `**` *power-expression*

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

A.3.4 Primary expressions

see §11.4

---

primary-expression ::
  class-definition
  | *eigenclass-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*
  | *rescue-expression*
  | *grouping-expression*
  | *variable-reference*
  | *scoped-constant-reference*
  | *array-constructor*
  | *hash-constructor*
  | *literal*
  | *defined?-with-parentheses*
  | *primary-method-invocation*

A.3.4.0.0.1 The `if` expression

see §11.4.1.1.1

---

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*

A.3.4.0.0.2 The `unless` expression

see §11.4.1.1.2

---

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

A.3.4.0.0.3 The `case` expression

see §11.4.1.1.3

---

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 (`,` *splatting-argument*)?
 | *splatting-argument*

A.3.4.0.0.4 Conditional operator

see §11.4.1.1.4

---

conditional-operator-expression ::
  range-constructor
 | *range-constructor* `?` *operator-expression*$_1$ `:` *operator-expression*$_2$

A.3.4.0.1 Iteration expressions

see §11.4.1.2

---

iteration-expression ::=
  while-expression

| *until-expression*
| *for-expression*
| *while-modifier-statement*
| *until-modifier-statement*

A.3.4.0.1.1 The `while` expression

see §11.4.1.2.1

---

while-expression ::
`while` *expression* *do-clause* `end`

do-clause ::
separator *compound-statement*
| `do` *compound-statement*

A.3.4.0.1.2 The `until` expression

see §11.4.1.2.2

---

until-expression ::
`until` *expression* *do-clause* `end`

A.3.4.0.1.3 The `for` expression

see §11.4.1.2.3

---

for-expression ::
`for` *for-variable* `in` *expression* *do-clause* `end`

for-variable ::
left-hand-side
| *multiple-left-hand-side*

A.3.4.0.2 Jump expressions

see §11.4.1.3

---

jump-expression ::=
return-expression
| *break-expression*
| *next-expression*

| *redo-expression*
| *retry-expression*

A.3.4.0.2.1 The `return` expression

see §11.4.1.3.1

---

return-expression ::=
return-without-argument
| *return-with-argument*

return-without-argument ::
`return`

return-with-argument ::
`return` *jump-argument*

jump-argument ::
argument

A.3.4.0.2.2 The `break` expression

see §11.4.1.3.2

---

break-expression ::=
break-without-argument
| *break-with-argument*

break-without-argument ::
`break`

break-with-argument ::
`break` *jump-argument*

A.3.4.0.2.3 The `next` expression

see §11.4.1.3.3

---

next-expression ::=
next-without-argument
| *next-with-argument*

next-without-argument ::
    `next`

next-with-argument ::
    `next` *jump-argument*

A.3.4.0.2.4 The `redo` expression

see §11.4.1.3.4

---

redo-expression ::
    `redo`

A.3.4.0.2.5 The `retry` expression

see §11.4.1.3.5

---

retry-expression ::
    `retry`

A.3.4.0.2.6 The `rescue` expression

see §11.4.1.4.1

---

rescue-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*

A.3.4.1 Grouping expression

see §11.4.2

---

grouping-expression ::
    `(` *expression* `)`
    | `(` *compound-statement* `)`

A.3.4.2 Variable references

see §11.4.3

---

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 *whitespace* here] `::` *constant-identifier*
    | `::` *constant-identifier*

A.3.4.2.1 Pseudo variables

see §11.4.3.7

---

pseudo-variable ::
    nil
    | *true*
    | *false*
    | *self*

A.3.4.2.1.1 `nil`

see §11.4.3.7.1

nil ::

    `nil`

A.3.4.2.1.2 `true` and `false`

see §11.4.3.7.2

true ::

    `true`

false ::

    `false`

A.3.4.2.1.3 `self`

see §11.4.3.7.3

self ::

    `self`

A.3.4.2.2 Array constructor

see §11.4.4.1

array-constructor ::

    `[` *indexing-argument-list*? `]`

A.3.4.2.3 Hash constructor

see §11.4.4.2

hash-constructor ::

    `{` (*association-list* `,`?)? `}`

association-list ::

    association (`,` *association*)*

association ::

    association-key `=>` *association-value*

association-key ::
    operator-expression

association-value ::
    operator-expression

A.3.4.2.4 Range constructor

see §11.4.4.3

---

range-constructor ::
    logical-OR-expression
    | *logical-OR-expression*$_1$ *range-operator* *logical-OR-expression*$_2$

range-operator ::
    `..`
    | `...`

A.4 Statements

A.4.1 The expression statement

see §12.1

---

expression-statement ::
    expression

A.4.2 The `if` modifier statement

see §12.2

---

if-modifier-statement ::
    statement [no *line-terminator* here] `if` *expression*

A.4.3 The `unless` modifier statement

see §12.3

---

unless-modifier-statement ::
    statement [no *line-terminator* here] `unless` *expression*

A.4.4 The `while` modifier statement

see §12.4

---

while-modifier-statement ::
    statement [no *line-terminator* here] `while` *expression*

A.4.5 The `until` modifier statement

see §12.5

---

until-modifier-statement ::
    statement [no *line-terminator* here] `until` *expression*

A.5 Classes and modules

A.5.0.1 Module definition

see §13.1.2

---

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

A.5.0.2 Class definition

see §13.2.2

class-definition ::
`class` *class-path* [no *line-terminator* here] *superclass* *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 ::
separator
| `<` *expression* *separator*

class-body ::
body-statement

A.5.0.3 Method definition

see §13.3.1

method-definition ::
`def` *method-name* [no *line-terminator* here] *method-parameter-part*
method-body `end`

method-name ::
method-identifier
| *operator-method-name*
| *reserved-word*

method-body ::
body-statement

A.5.0.4 Method parameters

see §13.3.2

---

method-parameter-part ::
  `(` *parameter-list*? `)`
  | *parameter-list*? *separator*

parameter-list ::
  mandatory-parameter-list `,` *optional-parameter-list*? `,`
    array-parameter? `,` *block-parameter*?
  | *optional-parameter-list* `,` *array-parameter*? `,` *block-parameter*?
  | *array-parameter* `,` *block-parameter*?
  | *block-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

block-parameter ::
  `&` *block-parameter-name*

block-parameter-name ::
    local-variable-identifier

A.5.0.5 The `alias` statement

see §13.3.6

---

alias-statement ::
    `alias` *new-name* *aliased-name*

new-name ::
    method-name
    | *symbol*

aliased-name ::
    method-name
    | *symbol*

A.5.0.6 The `undef` statement

see §13.3.7

---

undef-statement ::
    `undef` *undef-list*

undef-list ::
    method-name-or-symbol (`,` *method-name-or-symbol*)*

method-name-or-symbol ::
    method-name
    | *symbol*

A.5.0.7 Eigenclass definition

see §13.4.2

---

eigenclass-definition ::
    `class << ` *expression* *separator* *eigenclass-body* `end`

eigenclass-body ::
    body-statement

A.5.0.7.1 Patterns

see §15.2.15.3

---

pattern ::
  $alternative_1$
  | $pattern_1$ `|` $alternative_2$

alternative ::
  [empty]
  | $alternative_3$ *term*

term ::
  anchor
  | $atom_1$
  | $atom_2$ *quantifier*

anchor ::
  `^` | `$`

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-without-zero*

regexp-character-escape-sequence ::
  regexp-escape-sequence
  | *regexp-non-escaped-sequence*
  | *hex-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` | `b`

regexp-non-escaped-sequence ::
  `\` *regexp-non-escaped-character*

regexp-non-escaped-character ::
  source-character **but not** *regexp-escaped-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** (`\` | `?`)