prev - next - index

Ruby syntax

Lexical structure

The character set used in the Ruby source files for the current implementation is based on ASCII. The case of characters in source files is significant. All syntactic constructs except identifiers and certain literals may be separated by an arbitrary number of whitespace characters and comments. The whitespace characters are space, tab, vertical tab, backspace, carriage return, and form feed. Newlines works as whitespace only when expressions obviously continues to the next line.

Identifiers

Examples:

```
foobar
ruby_is_simple
```

Ruby identifiers are consist of alphabets, decimal digits, and the underscore character, and begin with a alphabets(including underscore). There are no restrictions on the lengths of Ruby identifiers.

Comment

Examples:

```
# this is a comment line
```

Ruby comments start with "#" outside of a string or character literal (`?#`) and all following text until the end of the line.

Embedded Documentation

Example:

```
=begin
the everything between a line beginning with `=begin' and
that with `=end' will be skipped by the interpreter.
=end
```

If the Ruby interpreter encounters a line beginning with `=begin`, it skips that line and all remaining lines through and including a line that begins with `=end`.

Reserved words

The reserved words are:

```
BEGIN    class    ensure   nil      self     when
END      def      false    not      super    while
alias    defined  for      or       then     yield
and      do       if       redo     true
begin    else     in       rescue   undef
break    elsif    module   retry    unless
case     end      next     return   until
```

Program

Example:

```
print "hello world!\n"
```

Ruby programs are sequence of expressions. Each expression are delimited by semicolons(;) or newlines. Backslashes at the end of line does not terminate expression.

Expression

Examples:

```
true
(1+2)*3
foo()
if test then ok else ng end
```

Ruby expressions can be grouped by parentheses.

String literals

Examples:

```
"this is a string expression\n"
"concat#{foobar}"
'concat#{foobar}'
%q!I said, "You said, 'She said it.'"!
%!I said, "You said, 'She said it.'"!
%Q('This is it.'\n)
```

String expressions begin and end with double or single quote marks. Double-quoted string expressions are subject to backslash escape and expression substitution. Single-quoted strings are not (except for `\'` and `\\`).

The string expressions begin with `%` are the special form to avoid putting too many backslashes into quoted strings. The `%q/STRING/` expression is the generalized single quote. The `%Q/STRING/` (or `%/STRING/`) expression is the generalized double quote. Any non-alphanumeric delimiter can be used in place of `/`, including newline. If the delimiter is an opening bracket or parenthesis, the final delimiter will be the corresponding closing bracket or parenthesis. (Embedded occurrences of the closing bracket need to be backslashed as usual.)

Backslash notation

`\t`
: tab(0x09)

`\n`
: newline(0x0a)

`\r`
: carriage return(0x0d)

`\f`
: form feed(0x0c)

`\b`
: backspace(0x08)

`\a`
: bell(0x07)

`\e`
: escape(0x1b)

`\s`
: whitespace(0x20)

`\nnn`
: character in octal value nnn

`\xnn`
: character in hexadecimal value nn

`\cx`
: control x

`\C-x`
: control x

`\M-x`
: meta x (c | 0x80)

`\M-\C-x`
: meta control x

`\x`
: character x itself

The string literal expression yields new string object each time it evaluated.

Command output

Examples:

```
`date`
%x{ date }
```

Strings delimited by backquotes are performed by a subshell after escape sequences interpretation and expression substitution. The standard output from the commands are taken as the value. Commands performed each time they evaluated.

The `%x/STRING/` is the another form of the command output expression.

Regular expression

Examples:

```
/^Ruby the OOPL/
/Ruby/i
/my name is #{myname}/o
%r|^/usr/local/.*|
```

Strings delimited by slashes are regular expressions. The characters right after latter slash denotes the option to the regular expression. Option `i` means that regular expression is case insensitive. Option `i` means that regular expression does expression substitution only once at the first time it evaluated. Option `x` means extended regular expression, which means whitespaces and commens are allowd in

the expression. Option `p` denotes POSIX mode, in which newlines are treated as normal character (matches with dots).

The `%r/STRING/` is the another form of the regular expression.

`^`
: beginning of a line or string

`$`
: end of a line or string

`.`
: any character except newline

`\w`
: word character[0-9A-Za-z_]

`\W`
: non-word character

`\s`
: whitespace character[\t\n\r\f]

`\S`
: non-whitespace character

`\d`
: digit, same as[0-9]

`\D`
: non-digit

`\A`
: beginning of a string

`\Z`
: end of a string, or before newline at the end

`\z`
: end of a string

`\b`
: word boundary(outside[]only)

`\B`
: non-word boundary

`\b`
: backspace(0x08)(inside[]only)

`[ ]`
: any single character of set

`*`
: 0 or more previous regular expression

`*?`
: 0 or more previous regular expression(non greedy)

`+`
: 1 or more previous regular expression

`+?`
: 1 or more previous regular expression(non greedy)

`{m,n}`
: at least m but most n previous regular expression

`{m,n}?`
: at least m but most n previous regular expression(non greedy)

`?`
: 0 or 1 previous regular expression

`|`
: alternation

`( )`
: grouping regular expressions

`(?# )`
: comment

`(?: )`
: grouping without backreferences

`(?= )`
: zero-width positive look-ahead assertion

`(?! )`
: zero-width negative look-ahead assertion

`(?ix-ix)`
: turns on (or off) `i' and `x' options within regular expression. These modifiers are localized inside an enclosing group (if any).

`(?ix-ix: )`
: turns on (or off) `i' and `x' options within this non-capturing group.

Backslash notation and expression substitution available in regular expressions.

Expression substitution in strings

Examples:

```
"my name is #{$ruby}"
```

In double-quoted strings, regular expressions, and command output expressions, the form like "#{expression}" extended to the evaluated result of that expression. If the expressions are the variables which names begin with the character either `$',`@', expressions are not needed to be surrounded by braces. The character `#' is interpreted literally if it it not followed by characters `{',`$',`@'.

line-oriented string literals (Here document)

There's a line-oriente form of the string literals that is usually called as `here document'. Following a << you can specify a string or an identifier to terminate the string literal, and all lines following the current line up to the terminator are the value of the string. If the terminator is quoted, the type of quotes determines the type of the line-oriented string literal. Notice there must be no space between << and the terminator.

If the - placed before the delimiter, then all leading whitespcae characters (tabs or spaces) are stripped from input lines and the line containing delimiter. This allows here-documents within scripts to be indented in a natural fashion.

```
  print <<EOF
The price is #{$Price}.
EOF

  print <<"EOF";                          # same as above
The price is #{$Price}.
EOF

  print <<`EOC`                 # execute commands
echo hi there
echo lo there
EOC

  print <<"foo", <<"bar"        # you can stack them
I said foo.
foo
I said bar.
bar

  myfunc(<<"THIS", 23, <<'THAT')
Here's a line
or two.
THIS
and here's another.
```

```
THAT

  if need_define_foo
    eval <<-EOS                # delimiters can be indented
      def foo
        print "foo\n"
      end
    EOS
  end
```

Numeric literals

`123`
: integer

`-123`
: integer(signed)

`1_234`
: integer(underscore within decimal numbers ignored)

`123.45`
: floating point number

`1.2e-3`
: floating point number

`0xffff`
: hexadecimal integer

`0b01011`
: binary integer

`0377`
: octal integer

`?a`
: ASCII code for character `a'(97)

`?\C-a`
: Control-a(1)

`?\M-a`
: Meta-a(225)

`?\M-\C-a`
: Meta-Control-a(129)

`:symbol`
: Integer corresponding identifiers, variable names, and operators.

In ?-representation all backslash notations are available.

Variables and constants

The variable in Ruby programs can be distinguished by the first character of its name. They are either global variables, instance variables, local variables, and class constants. There are no restriction for variable name length (except heap size).

Global variables

Examples:

```
$foobar
$/
```

The variable which name begins with the character `$', has global scope, and can be accessed from any location of the program. Global variables are available as long as the program lives. Non-initialized global variables has value `nil`.

Instance variables

Examples:

```
@foobar
```

The variable which name begins which the character `@', is an instance variable of `self`. Instance variables are belong to the certain object. Non-initialized instance variables has value `nil`.

Constants

Examples:

```
FOOBAR
```

The identifier which name begins with upper case letters ([A-Z]) is an constant. The constant definitions are done by assignment in the class definition body. Assignment to the constants must be done once. Changing the constant value or accessing to the non-initialized constants raises a `NameError` exception.

The constants can be accessed from:

- the class or module body in which the constant is defined, including the method body and the nested module/class definition body.
- the class which inherit the constant defining class.
- the class or module which includes the constant defining module.

Class definition defines the constant automatically, all class names are constants.

To access constants defined in certain class/module, operator `::` can be used.

To access constants defined in the Object class, operator `::` without the left hand side operand can be used.

Examples:

```
Foo::Bar
::Bar
```

No assignment using operator `::' is permitted.

Local variables

Examples:

```
foobar
```

The identifier which name begins with lower case character or underscore, is a local variable or a method invocation. The first assignment in the local scope (bodies of class, module, method definition) to such identifiers are declarations of the local variables. Non-declared identifiers are method invocation without arguments.

The local variables assigned first time in the blocks are only valid in that block. They are called `dynamic variables.' For example:

```
i0 = 1
```

```
loop {
  i1 = 2
  print defined?(i0), "\n"      # true
  print defined?(i1), "\n"      # true
  break
}
print defined?(i0), "\n"        # true
print defined?(i1), "\n"        # false
```

Pseudo variables

There are special variables called `pseudo variables'.

`self`
: the receiver of the current method

`nil`
: the sole instance of the Class NilClass(represents false)

`true`
: the sole instance of the Class TrueClass(typical true value)

`false`
: the sole instance of the Class FalseClass(represents false)

`__FILE__`
: the current source file name.

`__LINE__`
: the current line number in the source file.

The values of the pseudo variables cannot be changed. Assignment to these variables causes exceptions.

Array expression

Examples:

```
[1, 2, 3]
```

Syntax:

```
`[' expr,...`]'
```

Returns an array, which contains result of each expressions. Arrays are instances of the class Array.

`%w` expressions make creation of the arrays of strings easier. They are equivalent to the single quoted strings split by the whitespaces. For example:

```
%w(foo bar baz)
```

is equivalent to `["foo", "bar", "baz"]`. Note that parenthesis right after `%s` is the quote delimiter, not usual parenthesis.

Hash expression

Examples:

```
{1=>2, 2=>4, 3=>6}
```

Syntax:

```
{ expr => expr...}
```

Returns a new Hash object, which maps each key to corresponding value. Hashes are instances of the class Hash.

Method invocation

Examples:

```
foo.bar()
foo.bar
bar()
print "hello world\n"
print
```

Syntax:

```
[expr `.'] identifier [`(' expr...[`*' [expr]],[`&' ] expr`)']
[expr `::'] identifier [`(' expr...[`*' [expr]],[`&' expr] `)']
```

Method invocation expression invokes the method of the receiver (right hand side expression of the dot) specified by the identifier. If no receiver specified, `self` is used as a receiver.

Identifier names are normal identifiers and identifier suffixed by character `?` or `!`. As a convention, `identifier?` are used as predicate names, and `identifier!` are used for the more destructive (or more dangerous) methods than the method which have same name without `!`.

If the last argument expression preceded by `*`, the value of the expression expanded to arguments, that means

```
foo(*[1,2,3])
```

equals

```
foo(1,2,3)
```

If the last argument expression preceded by `&`, the value of the expression, which must be a `Proc` object, is set as the block for the calling method.

Some methods are *private*, and can be called from function form invocations (the forms that omits receiver).

`super`

Examples:

```
super
super(1,2,3)
```

Syntax:

```
super
super(expr,...)
```

the `super` invokes the method which the current method overrides. If no arguments given, arguments to the current method passed to the method.

Assignment

Examples:

```
foo = bar
foo[0] = bar
foo.bar = baz
```

Syntax:

```
variable '=' expr
constant '=' expr
expr`['expr..`]' '=' expr
expr`.'identifier '=' expr
```

Assignment expression are used to assign objects to the variables or such. Assignments sometimes work as declarations for local variables or class constants. The left hand side of the assignment expressions can be either:

- variables

  ```
  variables `=' expression
  ```

  If the left hand side is a variables, then assignment is directly performed.

- array reference

  ```
  expr1`[' expr2...`]' `=' exprN
  ```

  This from is evaluated to the invocation of the method named `[]=`, with expr1 as the receiver, and values expr2 to exprN as arguments.

- attribute reference

  ```
  expr `.' identifier `=' expr
  ```

  This from is evaluated to the invocation of the method named `identifier=` with the right hand side expression as a argument.

self assignment

Examples:

```
foo += 12
```

Syntax:

```
expr op= expr     # left hand side must be assignable.
```

This form evaluated as `expr = expr op expr`. But right hand side expression evaluated once. op can be one of:

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

There may be no space between operators and `=`.

Multiple assignment

Examples:

```
foo, bar, baz = 1, 2, 3
foo, = list()
foo, *rest = list2()
```

Syntax:

```
expr `,' [expr `,'...] [`*' expr] = expr [, expr...][`*' [expr]]
`*' expr = expr [, expr...][`*' expr]
```

Multiple assignment form performs multiple assignment from expressions or an array. Each left hand side expression must be assignable. If single right hand side expression given, the value of the expression converted into an array, then each element in array assigned one by one to the left hand side expressions. If number of elements in the array is greater than left hand sides, they are just ignored. If left hand sides are longer than the array, `nil` will be added to the locations.

Multiple assignment acts like this:

```
foo, bar = [1, 2]       # foo = 1; bar = 2
foo, bar = 1, 2         # foo = 1; bar = 2
foo, bar = 1            # foo = 1; bar = nil

foo, bar, baz = 1, 2    # foo = 1; bar = 2; baz = nil
foo, bar = 1, 2, 3      # foo = 1; bar = 2
foo,*bar = 1, 2, 3      # foo = 1; bar = [2, 3]
```

The value of the multiple assignment expressions are the array used to assign.

Operator expressions

Examples:

```
1+2*3/4
```

As a syntax sugar, several methods and control structures has operator form. Ruby has operators show below:

```
high   ::
       []
       **
       -(unary)  +(unary)  !  ~
       *  /  %
       +  -
       <<  >>
       &
       |  ^
       >  >=  <  <=
       <=> ==  === !=  =~  !~
       &&
       ||
       .. ...
       =(+=, -=...)
       not
low    and or
```

Most of operators are just method invocation in special form. But some operators are not methods, but built in to the syntax:

```
=, .., ..., !, not, &&, and, ||, or, !=, !~
```

In addition, assignment operators(+= etc.) are not user-definable.

Control structure

Control structures in Ruby are expressions, and have some value. Ruby has the loop abstraction feature called iterators. Iterators are user-definable loop structure.

`if`

Examples:

```
if age >= 12 then
  print "adult fee\n"
else
  print "child fee\n"
end
gender = if foo.gender == "male" then "male" else "female" end
```

Syntax:

```
if expr [then]
  expr...
[elsif expr [then]
  expr...]...
[else
  expr...]
end
```

`if` expressions are used for conditional execution. The values `false` and `nil` are false, and everything else are true. Notice Ruby uses `elsif`, not `else if` nor `elif`.

If conditional part of `if` is the regular expression literal, then it evaluated like:

```
$_ =~ /re/
```

`if` modifier

Examples:

```
print "debug\n" if $debug
```

Syntax:

```
expr if expr
```

executes left hand side expression, if right hand side expression is true.

`unless`

Examples:

```
unless $baby
  feed_meat
else
  feed_milk
end
```

Syntax:

```
unless expr [then]
```

```
  expr...
[else
  expr...]
end
```

`unless` expressions are used for reverse conditional execution. It is equivalent to:

```
if !(cond)
  ...
else
  ...
end
```

`unless` modifier

Examples:

```
print "stop\n" unless valid($passwd)
```

Syntax:

```
expr unless expr
```

executes left hand side expression, if right hand side expression is false.

`case`

Examples:

```
case $age
when 0 .. 2
  "baby"
when 3 .. 6
  "little child"
when 7 .. 12
  "child"
when 12 .. 18
  # Note: 12 already matched by "child"
  "youth"
else
  "adult"
end
```

Syntax:

```
case expr
[when expr [, expr]...[then]
  expr..]..
[else
  expr..]
end
```

the `case` expressions are also for conditional execution. Comparisons are done by operator `===`. Thus:

```
case expr0
when expr1, expr2
  stmt1
when expr3, expr4
  stmt2
else
  stmt3
```

```
end
```

is basically same to below:

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
  stmt1
elsif expr3 === _tmp || expr4 === _tmp
  stmt2
else
  stmt3
end
```

Behavior of the `===` method varies for each Object. See docutmentation for each class.

`and`

Examples:

```
test && set
test and set
```

Syntax:

```
expr `&&' expr
expr `and' expr
```

Evaluates left hand side, then if the result is true, evaluates right hand side. `and` is lower precedence alias.

`or`

Examples:

```
demo || die
demo or die
```

Syntax:

```
expr `||' expr
expr or expr
```

Evaluates left hand side, then if the result is false, evaluates right hand side. `or` is lower precedence alias.

`not`

Examples:

```
! me
not me
i != you
```

Syntax:

```
`!' expr
not expr
```

Returns true if false, false if true.

```
expr `!=' expr
```

Syntax sugar for `!(expr == expr)`.

```
expr `!~' expr
```

Syntax sugar for `!(expr =~ expr)`.

Range expressions

Examples:

```
1 .. 20
/first/ ... /second/
```

Syntax:

```
expr `..' expr
expr `...' expr
```

If range expression appears in any other place than conditional expression, it returns range object from left hand side to right hand side.

If range expression appears in conditional expression, it gives false until left hand side returns true, it stays true until right hand side is true. `..` acts like `awk`, `...` acts like `sed`.

`while`

Examples:

```
while sunshine
  work()
end
```

Syntax:

```
while expr [do]
  ...
end
```

Executes body while condition expression returns true.

`while` modifier

Examples:

```
sleep while idle
```

Syntax:

```
expr while expr
```

Repeats evaluation of left hand side expression, while right hand side is true. If left hand side is `begin` expression, `while` evaluates that expression at lease once.

`until`

Examples:

```
until sunrise
  sleep
end
```

Syntax:

```
until expr [do]
  ...
end
```

Executes body until condition expression returns true.

`until` modifier

Examples:

```
work until tired
```

Syntax:

```
expr until expr
```

Repeats evaluation of left hand side expression, until right hand side is true. If left hand side is `begin` expression, `until` evaluates that expression at lease once.

Iterators

Examples:

```
[1,2,3].each do |i| print i*2, "\n" end
[1,2,3].each{|i| print i*2, "\n"}
```

Syntax:

```
method_call do [`|' expr...`|'] expr...end
method_call `{' [`|' expr...`|'] expr...`}'
```

The method may be invoked with the block (`do .. end` or `{..}`). The method may be evaluate back that block from inside of the invocation. The methods that calls back the blocks are sometimes called as iterators. The evaluation of the block from iterator is done by `yield`.

The difference between `do` and braces are:

- Braces has stronger precedence. For example:

```
foobar a, b do .. end   # foobar will be called with the block.
foobar a, b { .. }      # b will be called with the block.
```

- Braces introduce the nested local scopes, that is newly declared local variables in the braces are valid only in the blocks. For example:

```
foobar {
  i = 20                # local variable `i' declared in the block.
  ...
}
print defined? i        # `i' is not defined here.
foobar a, b { .. }      # it is not valid outside of the block
```

for

Examples:

```
for i in [1, 2, 3]
  print i*2, "\n"
end
```

Syntax:

```
for lhs... in expr [do]
  expr..
end
```

Executes body for each element in the result of expression. `for` is the syntax sugar for:

```
(expr).each `{' `|' lhs..`|' expr.. `}'
```

yield

Examples:

```
yield data
```

Syntax:

```
yield `(' [expr [`,' expr...]])
yield [expr [`,' expr...]]
```

Evaluates the block given to the current method with arguments, if no argument is given, `nil` is used as an argument. The argument assignment to the block prameter is done just like multiple assignment. If the block is not supplied for the current method, the exception is raised.

raise

Examples:

```
raise "you lose"  # raise RuntimeError
# both raises SyntaxError
raise SyntaxError, "invalid syntax"
raise SyntaxError.new("invalid syntax")
raise             # re-raise last exception
```

Syntax:

```
raise
raise message_or_exception
raise error_type, message
raise error_type, message, traceback
```

Raises a exception. In the first form, re-raises last exception. In second form, if the argument is the string, creates a new `RuntimeError` exception, and raises it. If the argument is the exception, `raise` raises it. In the third form, `raise` creates a new exception of type *error_type*, and raises it. In the last form, the third argument is the traceback information for the raising exception in the format given by variable *$@* or `caller` function.

The exception is assigned to the variable `$!`, and the position in the source file is assigned to the `$@`.

The word `raise' is not the reserved word in Ruby. raise is the method of the Kernel module. There is an alias named fail.

begin

Examples:

```
begin
  do_something
rescue
  recover
ensure
  must_to_do
end
```

Syntax:

```
begin
  expr..
[rescue [error_type,..]
  expr..]..
[else
  expr..]
[ensure
  expr..]
end
```

begin expression executes its body and returns the value of the last evaluated expression.

If an exception occurs in the begin body, the rescue clause with the matching exception type is executed (if any). The match is done by the kind_of?. The default value of the rescue clause argument is the StandardError, which is the superclass of most built-in exceptions. Non-local jumps like SystemExit or Interrupt are not subclass of the StandardError.

The begin statement has an optional else clause, which must follow all rescue clauses. It is executed if the begin body does not raise any exception.

For the rescue clauses, the error_type is evaluated just like the arguments to the method call, and the clause matches if the value of the variable *$!* is the instance of any one of the error_type of its subclass. If *error_type* is not class nor module, the rescue clause raises *TypeError* exception.

If ensure clause given, its clause body executed whenever beginbody exits.

retry

Examples:

```
retry
```

Syntax:

```
retry
```

If retry appears in rescue clause of begin expression, restart from the beginning of the 1begin body.

```
begin
  do_something # exception raised
rescue
  # handles error
```

```
  retry  # restart from beginning
end
```

If `retry` appears in the iterator, the block, or the body of the `for` expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

```
for i in 1..5
  retry if some_condition # restart from i == 1
end

# user defined "until loop"
def UNTIL(cond)
  yield
  retry if not cond
end
```

`retry` out of `rescue` clause or iterators raises exception.

`return`

Examples:

```
return
return 12
return 1,2,3
```

Syntax:

```
return [expr[`,' expr...]]
```

Exits from method with the return value. If more than two expressions are given, the array contains these values will be the return value. If no expression given, `nil` will be the return value.

`break`

Examples:

```
i=0
while i<3
  print i, "\n"
  break
end
```

Syntax:

```
break
```

Exits from the most internal loop. Notice `break` does not exit from `case` expression like C.

`next`

Examples:

```
next
```

Syntax:

```
next
```

Jumps to next iteration of the most internal loop.

redo

Examples:

```
redo
```

Syntax:

```
redo
```

Restarts this iteration of the most internal loop, without checking loop condition.

BEGIN

Examples:

```
BEGIN {
  ...
}
```

Syntax:

```
BEGIN '{'
  expr..
'}'
```

Registers the initialize routine. The block followed after `BEGIN` is evaluated before any other statement in that file (or string). If multiple `BEGIN` blocks are given, they are evaluated in the appearing order.

The `BEGIN` block introduce new local-variable scope. They don't share local variables with outer statements.

The `BEGIN` statement can only appear at the toplevel.

END

Examples:

```
END {
  ...
}
```

Syntax:

```
END '{' expr.. '}'
```

Registers finalize routine. The block followed after `END` is evaluated just before the interpreter termination. Unlike `BEGIN`, `END` blocks shares their local variables, just like blocks.

The `END` statement registers its block only once at the first execution. If you want to register finalize routines many times, use at_exit.

The `END` statement can only appear at the toplevel. Also you cannot cancel finalize routine registered by `END`.

Class definitions

Examples:

```
class Foo < Super
  def test
     :
  end
     :
end
```

Syntax:

```
class identifier [`<' superclass ]
  expr..
end
```

Defines the new class. The class names are identifiers begin with uppercase character.

Singleton-class definitions

Examples:

```
class << obj
  def test
     :
  end
     :
end
```

Syntax:

```
class `<<' expr
  expr..
end
```

Defines the class attribute for certain object. The definitions within this syntax only affect the specified object.

Module definitions

Examples:

```
module Foo
  def test
     :
  end
     :
end
```

Syntax:

```
module identifier
  expr..
end
```

Defines the new module The module names are identifiers begin with uppercase character.

Method definitions

Examples:

```
def fact(n)
  if n == 1 then
     1
  else
    n * fact(n-1)
  end
end
```

Syntax:

```
def method_name [`(' [arg ['=' default]]...[`,' `*' arg ]`)']
  expr..
end
```

Defines the new method. Method_name should be either identifier or re-definable operators (e.g. ==, +, -, etc.). Notice the method is not available before the definition. For example:

```
foo
def foo
  print "foo\n"
end
```

will raise an exception for undefined method invoking.

The argument with default expression is optional. The evaluation of the default expression is done at the method invocation time. If the last argument preceded by `*`, actual parameters which don't have corresponding formal arguments are assigned in this argument as an array.

If the last argument preceded by `&`, the block given to the method is converted into the `Proc` object, and assigned in this argument. In case both `*` and `&` are present in the argument list, `&` should come later.

The method definitions can not be nested.

The return value of the method is the value given to the `return`, or that of the last evaluated expression.

Some methods are marked as `private', and must be called in the function form.

When the method is defined outside of the class definition, the method is marked as private by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the `private' mark of the methods can be changed by `public` or `private` of the `Module`.

In addition, the methods named `initialize` are always defined as private methods.

Singleton-method definitions

Examples:

```
def foo.test
  print "this is foo\n"
end
```

Syntax:

```
def expr `.' identifier [`(' [arg [`=' default]]...[`,' `*' arg ]`)']
```

```
  expr..
end
```

The singleton-method is the method which belongs to certain object. The singleton-method definitions can be nested.

The singleton-methods of classes inherited to its subclasses. The singleton-methods of classes are acts like class methods in other object-oriented languages.

alias

Examples:

```
alias foo bar
alias $MATCH $&
```

Syntax:

```
alias method-name method-name
alias global-variable-name global-variable-name
```

Gives alias to methods or global variables. Aliases can not be defined within the method body.

The aliase of the method keep the current definition of the method, even when methods are overridden.

Making aliases for the numbered global variables (`$1`, `$2`,...) is prohibited. Overriding the builtin global variables may cause serious problems.

undef

Examples:

```
undef bar
```

Syntax:

```
undef method-name
```

Cancels the method definition. `Undef` can not appear in the method body. By using `undef` and `alias`, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

defined?

Examples:

```
defined? print
defined? File.print
defined?(foobar)
defined?($foobar)
defined?(@foobar)
defined?(Foobar)
```

Syntax:

```
defined? expr
```

Returns false if the expression is not defined. Returns the string that describes a kind of the expression.

---

prev - next - index

matz@netlab.co.jp