Ruby - Bug #20063

Inconsistent behavior with required vs optional parameters

12/13/2023 05:59 PM - jemmai (Jemma Issroff)

Status:	Rejected		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	3.0: UNKNOWN, 3.1: UNKNOWN, 3.2: UNKNOWN

Description

Using repeated anonymous parameters, gives different behavior with the parameters in the same order based on whether they are required or optional.

```
def example_with_optional(_a = 1, _a = 2)
  _a
end

def example_with_required(_a, _a)
  _a
end

p example_with_optional # 2
p example_with_required(1, 2) # 1
```

It is unexpected that these two methods would return differently given the parameters are in the same order, based on whether they are optional or required.

History

#1 - 12/13/2023 06:21 PM - zverok (Victor Shepelev)

It seems that the difference is not in whether they are required or optional but rather in "the second default evaluates even if the first one is already evaluated."

```
p example_with_optional(1, 2) #=> 1
```

So when they are passed explicitly, the first assignment "wins."

But when there are defaults, it seems that "what statements should be evaluated" is decided before the call (so when going to the second _a, it doesn't "see" that the variable doesn't need a default anymore, and just evaluates the second statement like an assignment):

```
def example_with_optional(_a = begin; puts "a1"; p binding.local_variables; 1 end, _a = begin; puts "a2"; 2 end)
  _a
end
# Prints:
# a1 -- first block is called
# [:_a] -- we already have a local var
# a2 -- the second block is invoked nevertheless
# 2 -- the result
```

I don't have a good mental model of what happens with explicitly passed arguments, though :)
"Intuitively" it seems that it should be evaluated as two subsequent assignments to the same name, so the 2 should "win".

#2 - 12/13/2023 08:36 PM - rubyFeedback (robert heiler)

That last example is one very complicated def - it is probably
the most complicated one I have seen so far.

In regards to jemmai's example: what would you expect to be the
correct behaviour? I have not seen methods defined with the
same arguments (such as "(_a = 1, _a = 2)"). Do such examples
come up in real code? Is that behaviour documented or specified?

I guess the example can be generalized to using the same name
in the method definition, but then I tried it myself, thinking

this is equivalent to this:

```
def foobar_optional(x = 1, x = 2)
  x
end
```

Then I realised it is not:

```
duplicated argument name (SyntaxError)
```

So now I wonder about the use case and behaviour of _ variables.

Why isn't this a SyntaxError?

If this is perfectly fine, then indeed, I am a bit confused as
to why the behaviour is then different. Intuitively I would then
agree with jemmai; at the least this is what my brain would
associate with, if the default argument for that is = 1 for
both variables (even though they seem to be the same variable).

#3 - 12/20/2023 07:59 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

This situation arises due to the absence of a duplicate check, since the variable name is prefixed with _. I believe such cases should be treated as undefined behavior.

Matz.