Ruby master - Feature #6868

Make `do` in block syntax optional when the block is the last argument of a method and is not an optional argument

08/15/2012 01:26 AM - alexeymuranov (Alexey Muranov)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**Description**

```
=begin
I propose to make the use of "(({do}))" in block syntax optional if the block is the last argument of a method and if it is a required argument, at least if the block does not take parameters. (I think this would be in line with how for example the last hash argument is treated.) I also think that this syntactic change may allow in future versions of Ruby to make (({def})), (({class})), (({module})) methods instead of keywords.

Something like:

3.time
puts "Hi!"
end

instead of

3.time do
puts "Hi!"
end

I know this is not a good example, because for (({#times})) the block argument is optional.
==end
```

**History**

#1 - 08/15/2012 02:06 AM - alexeymuranov (Alexey Muranov)

I have just realized that parameters are parsed before looking up the method definition, so maybe this is not possible...

I was just thinking if it is possible to unify somehow all Ruby objects and classes. (No matter how hard Class pretends to be a simple object, it is clear that it is not only not a simple object, but not even a simple class, it cannot be subclassed for example.)

#2 - 08/15/2012 05:24 AM - drbrain (Eric Hodel)

- Status changed from Open to Feedback
- Priority changed from 3 to Normal

```
=begin
What would this code do:

def method n
n = n.times
p n
n
end

Currently it prints (({#})) and returns an Enumerator. Would it instead raise SyntaxError for an unexpected end?
==end
```

#3 - 08/15/2012 06:02 AM - alexeymuranov (Alexey Muranov)

You mean, what should method(3) output? This was not considered in my proposal, i think it should be the same. In fact, i understand that there are issues with making def a method, i was only proposing to make blocks look more like class and method definitions in certain circumstances. I do not remember if there are Ruby core methods that require a block, all that come to my mind have an optional block parameter. So here is a made-up example of the use of the proposed syntax:
def do_three_times(&block)
  3.times(&block)
end

do_three_times
puts "Hi"
end

The interpreter would need to look up the method definition for `do_three_times` before parsing the parameters (I think this is different from the current behavior), see that there is a required block parameter at the end, and so to expect either do or just new line to start the block after `do_three_times`.

#4 - 08/15/2012 08:43 AM - pedz (Perry Smith)
The idea seems extremely bad to me.

#5 - 08/15/2012 10:11 AM - drbrain (Eric Hodel)
=begin
It seems impossible to implement without fundamentally changing the feel of writing ruby.

Due to enumerators, currently your example results in a SyntaxError:

$ pbpaste | ruby20
.-:7: syntax error, unexpected keyword_end, expecting $end

How do we disambiguate between existing uses of `((&block)))` and your proposal?

Currently a method with `((&block)))` accepts an optional block argument that is captured to the local variable `((block))`. Since a named block argument is always optional it can be used by methods that return Enumerators (such as creating an enumerator from an Enumerable internal to the object). Changing this to mean "a block argument is required" breaks such usage.

After solving the problem of specifying a method’s signature to enable your special parsing, how do we look up this information at runtime if the method is dynamically defined or method_missing is involved?

How do we parse this example?

class C
def method
do_three_times
  puts "Hi"
end
end

def initialize
  # defines the do_three_times method in a manner that enables "do"-less syntax
  require 'c/do_three_times'
end
end

C.new.do_three_times

The definition of `((do_three_times)))` is ambiguous when the call is encountered, so the number of `((end)))s` to consume is impossible to determine. Ruby can’t look in `((c/do_three_times.rb)))` to determine how to parse `((do_three_times)))` since it is dynamic.

If the parser assumes `((do_three_times)))` is a do-less block then `((C.new)))` will require `((c/do_three_times.rb)))`. However, if that file defines `((do_three_times)))` to be a method that returns an Enumerator (not a do-less block) then the file was required at the wrong spot (as the second `((end)))` after `((puts)))` closed C and `#initialize` was not part of class C so the require should not have happened.

I'm unsure how you are going to resolve ambiguity due to the dynamic nature of ruby absent a `((do)))` or introduction of some other discriminator (such as python's significant whitespace).
=end

#6 - 08/15/2012 04:33 PM - alexeymuranov (Alexey Muranov)
Thanks for the explanation and for the last example with method call inside def, i did not think well enough. I see that keywords can be parsed because they are known to be keywords. I think this can be closed, sorry.

#7 - 08/16/2012 02:35 AM - drbrain (Eric Hodel)
- Status changed from Feedback to Rejected

Do not be sorry!
Adding syntax to a language is very difficult and touches all of the language. It is great that you tried!