# Ruby master - Feature #16499

## define_method(non_lambda) should not change the semantics of the given Proc

01/10/2020 10:36 PM - Eregon (Benoit Daloze)

| | |
|---|---|
| **Status:** | Rejected |
| **Priority:** | Normal |
| **Assignee:** | |
| **Target version:** | |

**Description**

From https://bugs.ruby-lang.org/issues/15973?next_issue_id=15948&prev_issue_id=15975#note-38

But I think we should change define_method(&non_lambda) because that currently confusingly treats the same block body differently (e.g., the same return in the code means something different).

This is the only construct in Ruby that can change a non-lambda to a lambda, and it's very inconsistent.
It also forces implementations to have a way to convert a proc to a lambda, which is a non-trivial change.

We could maybe make define_method(name, non_lambda) just wrap the Proc in a lambda, automatically,
just like we can do manually with: define_method(name, -> *args { non_lambda.call(*args) }).
But it would also preserve arity, parameters, etc.
Then it wouldn't be any more verbose, but it would avoid the problem of treating the same return/break in the code differently.

My point is we shall never change the semantics of return/break somewhere in the code.
It should always mean exactly one thing.
define_method(name) { literal block } is fine with that rule, it always behave as a lambda.
But define_method(&non_lambda) is problematic as non_lambda can be passed to other methods or called directly.

I believe exactly 0 people want foo { return 42 } to change its meaning based on whether foo calls define_method or not.

OTOH, it seems people have repeatedly wanted to convert a proc to a lambda, but for other reasons.
We should look at those reasons and provide better alternatives.

I think sometimes people want to know how many arguments a non-lambda Proc takes.
For example, proc { |a,b=1| }.
proc.arity gives 1 here which might be helpful but also surprising as that Proc accepts any number of arguments.
They might also look at proc.parameters which gives [[:opt, :a], [:opt, :b]] which does not differentiate a and b even though only b has a proper default value.
lambda { |a,b=1| }.parameters returns the more useful [[:req, :a], [:opt, :b]].

Maybe we should return the same as for a lambda for non_lambda.parameters?
Proc#lambda? would still tell whether it's strict about arguments and whether it deconstructs them.

cc zverok (Victor Shepelev)

**Related issues:**

| | |
|---|---|
| Related to Ruby master - Feature #15973: Let Kernel#lambda always return a la... | **Closed** |
| Related to Ruby master - Feature #15357: Proc#parameters returns incomplete t... | **Open** |

### History

**#1 - 01/10/2020 10:36 PM - Eregon (Benoit Daloze)**

*- Related to Feature #15973: Let Kernel#lambda always return a lambda added*

**#2 - 01/11/2020 10:44 PM - marcandre (Marc-Andre Lafortune)**

> I believe exactly 0 people want foo { return 42 } to change its meaning based on whether foo calls define_method or not.

This is wrong, there is at least me 

I believe that many API use define_method for metaprogramming and allow return within their blocks.

One example is RSpec's let:

```
RSpec.describe Something do
  let(:foo) { return 42 }
end
```

It is 100% clear what is meant and there are gazillions let blocks in the wild. This is just one example.

This would be a compatibility nightmare, for a gain I can not see (here simply raising an error).

I am strongly against this.

#### #3 - 01/12/2020 02:20 PM - zverok (Victor Shepelev)

Eregon (Benoit Daloze) what is the exact proposal of this ticket? I am not sure neither from title nor from description :(

As a side note, in regards to the last part:

> They might also look at proc.parameters which gives [[:opt, :a], [:opt, :b]] which does not differentiate a and b even though only b has a proper default value.
> lambda { |a,b=1| }.parameters returns the more useful [[:req, :a], [:opt, :b]].
>
> Maybe we should return the same as for a lambda for non_lambda.parameters?

I believe curent behavior is pretty consistent, as it describes what it would realy accept. req means it will raise "Wrong number of arguments" if the argument is not provided, opt means it will accept argument's absence and will provide the default value. So, proc { |a, b=1| "real" signature (considering how it will process its args), is in fact proc { |a=nil, b=1, *|. **If** some complicated code accepts "any callable" and somehow validates "what args it requires", opt is more true for non-lambda's arg than req.

#### #4 - 01/12/2020 04:19 PM - Eregon (Benoit Daloze)

marcandre (Marc-Andre Lafortune) wrote:

> One example is RSpec's let:

I guess we'll have to disagree on that one, I think the code below should return from the surrounding method/file.

```
RSpec.describe Something do
  let(:foo) { return 42 }
end
```

> It is 100% clear what is meant and there are gazillions let blocks in the wild. This is just one example.

I would think very few let use return though, do you have a real world example?

> This would be a compatibility nightmare, for a gain I can not see (here simply raising an error).

If we do the approach where we just wrap the non-lambda Proc in a lambda automatically it would be compatible for that case.

#### #5 - 01/12/2020 04:24 PM - Eregon (Benoit Daloze)

zverok (Victor Shepelev) wrote:

> I believe curent behavior is pretty consistent, as it describes what it would realy accept.

Yes, in that regard it's inconsistent.
It might be impractical though, depending on whether you want something that reflects what the user writes (i.e., I'd argue always unexpected for the user to be called with 0 arguments) or how many arguments the method accepts.

But anyway Proc#arity is clearly inconsistent with parameters:

```
> proc { |a,b=1| }.arity
=> 1 # => should be -1, accepts any amount of arguments
> lambda { |a,b=1| }.arity
=> -2

In contrast to:
> proc { |*rest| }.arity
=> -1 # OK
> lambda { |*rest| }.arity
```

```
=> -1 # OK
```

And I'd argue Proc#arity is what should be used to know how many arguments are required and allowed, not Proc#parameters.

#### #6 - 01/12/2020 04:27 PM - Eregon (Benoit Daloze)

Eregon (Benoit Daloze) wrote:

> If we do the approach where we just wrap the non-lambda Proc in a lambda automatically it would be compatible for that case.

I'm tired, that's wrong, it would actually return from the file, just like any other non-lambda block.
That would be consistent, but yet it would be incompatible for those cases with return inside a block given to define_method later on.
I think those cases are very rare though.

#### #7 - 01/14/2020 01:19 PM - Eregon (Benoit Daloze)

*- Subject changed from define_method(non_lambda) should not the semantics of the given Proc to define_method(non_lambda) should not change the semantics of the given Proc*

#### #8 - 01/14/2020 01:19 PM - Eregon (Benoit Daloze)

*- Related to Feature #15357: Proc#parameters returns incomplete type information added*

#### #9 - 01/16/2020 08:18 AM - matz (Yukihiro Matsumoto)

*- Status changed from Open to Rejected*

There could be enormous code breakages by the proposed change. The compatibility is more important than slightly better consistency.

Matz.

#### #10 - 01/16/2020 09:57 AM - larskanis (Lars Kanis)

Unfortunately define_method is currently the only way to retrieve Proc#parameters without information loss. See #15357 and here for the workaround per default_method. Therefore fixing default_method would also require fixing Proc#parameters.