Ruby master - Feature #15973

Let Kernel#lambda always return a lambda

07/02/2019 01:20 PM - alanwu (Alan Wu)

Status:	Closed
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	3.0

Description

When Kernel#lambda receives a Proc that is not a lambda,
it returns it without modification. I propose to change Kernel#lambda
so it always returns a lambda.

Calling a method called lambda and having it do nothing in effect is
not very intuitive.

https://github.com/ruby/ruby/pull/2262

Judging from marcandre's investigation here: https://bugs.ruby-lang.org/issues/15620#note-1,
changing the behavior should not cause much breakage, if any.

This also happens to fix [Bug #15620]

Related issues:

Related to Ruby master - Feature #12957: A more OO way to create lambda Procs	**Feedback**	
Related to Ruby master - Feature #7314: Convert Proc to Lambda doesn't work i...	**Assigned**	
Related to Ruby master - Feature #9777: Feature Proposal: Proc#to_lambda	**Feedback**	
Related to Ruby master - Feature #8693: lambda invoked by yield acts as a pro...	**Closed**	**07/26/2013**
Related to Ruby master - Bug #16004: Kernel#lambda captured with Kernel#metho...	**Closed**	
Related to Ruby master - Bug #15620: Block argument usage affects lambda sema...	**Closed**	
Related to Ruby master - Feature #16499: define_method(non_lambda) should not...	**Rejected**	

Associated revisions

Revision 2188d6d1 - 06/11/2020 02:30 PM - jeremyevans (Jeremy Evans)

Warn when passing a non-literal block to Kernel#lambda

Implements [Feature #15973]

Revision 1efc3d6d - 06/13/2020 01:57 PM - nobu (Nobuyoshi Nakada)

Suppress warnings [Feature #15973]

History

#1 - 07/02/2019 01:31 PM - matz (Yukihiro Matsumoto)

I agree. Even though we have to investigate how big the consequence of the change first.

Matz.

#2 - 07/02/2019 02:14 PM - alanwu (Alan Wu)

- Subject changed from Make Kernel#lambda always return lambda to Make it so Kernel#lambda always return a lambda

#3 - 07/02/2019 08:49 PM - Eregon (Benoit Daloze)

I'm not sure changing this is good, because it can be very surprising for the code to change semantics dynamically.
Also, should proc(&lambda) make a non-lambda Proc then? It would be inconsistent if not.

As I said in https://bugs.ruby-lang.org/issues/15620#note-2, the current rule is AFAIK only change the semantics if a block is directly given, because the programmer means to use those semantics then.
If not, don't change the semantics as it would break the user (the Proc's code)'s intention.
The only exception to this is define_method with a pre-existing Proc, which is very rare, and understandably needed because methods should check arguments strictly. (maybe we don't need that exception)

I believe #15620 should be fixed on its own, it's a bug of the optimization.

Here is an example (a bit contrived, I'm tired):

```
def foo(arg)
  early_check = Proc.new {
    # everything here assumes it will always be proc/non-lambda semantics
    return :early_return if arg > 3
  }

  # Some way for some external code to access `early_check`
  early_check = yield early_check

  early_check.call

  :method_return_value
end

p foo(4) { |pr| lambda(&pr) }
```

With current semantics, it returns :early_return.
With the proposed change, it returns :method_return_value.
That's very surprising, isn't it? The code clearly means it wants a non-local return to exit the method,
and yet somehow it was transformed into a local lambda return!

I think such a surprising transformation of user code should happen as little as possible, so I'm against this proposal.

Maybe we should simply forbid calling proc/lambda without a literal block (i.e., with an explicit Proc like lambda(&pr)), since it doesn't do anything useful ?
That would make more sense to me, and be in line with ko1 (Koichi Sasada)'s recent changes to disallow things like Proc.new, etc without a block.

#4 - 07/03/2019 01:59 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #12957: A more OO way to create lambda Procs added

#5 - 07/03/2019 01:59 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #7314: Convert Proc to Lambda doesn't work in MRI added

#6 - 07/03/2019 02:03 AM - shyouhei (Shyouhei Urabe)

No. I'm against it. We have discussed this before multiple times. See the issues I linked just now.

#7 - 07/08/2019 12:20 AM - alanwu (Alan Wu)

The 2.4 spec is a bit problematic since it makes it impossible to forward a
block to Kernel#lambda with a block pass. In the rest of the language
forwarding a block has no effect on semantics compared to passing one
literally. The idea of literal blocks being different from non-literal ones
exists only in the case of Kernel#lambda.

A concept at odds with the rest of the language specific to one single method
is sure to surprise. Special one-off concepts make the language harder to learn
and add complications to implementaitons.

As others have pointed out, it's already possible to transform a proc into a
lambda-proc via define_method. My proposal isn't adding anything new with
regards to messing with the perscribed usage of code within blocks.

Transforming a proc into a lambda-proc is certainly a sharp tool. For me it
falls in the same category as instance_variable_set and const_set. However,
I think the situation in which it ends up being surprinsg is very rare.
return within a Proc.new do ... comes up rarely as is.

Is "someone might misuse this in specific sutations" a good reason to keep an
ad-hoc concept in the language? For me, no.

#8 - 07/08/2019 12:35 AM - alanwu (Alan Wu)

I would also like to note that if we revert back to the behavior in 2.4, code that relied on lambda without block will not have an easy upgrade path.

```
def make_a_lambda
  lambda
end
```

is not equivalent to

```
def make_a_lambda(&block)
  lambda(&block)
end
```

under 2.4 spec.

#9 - 07/08/2019 01:49 PM - shyouhei (Shyouhei Urabe)

"It's already broken, why not break it more" is not what I can follow.

Can I ask you why you need this feature? If this not more than a matter of consistency, I would like to second Eregon (Benoit Daloze)'s proposal: lambdas without literal blocks to be prohibited at all.

#10 - 07/09/2019 02:32 PM - alanwu (Alan Wu)

> Can I ask you why you need this feature?

I want to be able to forward a block to Kernel#lambda.

Kernel#lambda is in a weird spot. Even though it's a method, making it behave like one has the unfortunate side-effect of allowing proc transformation.
On the other hand, restricting it to just literal blocks is fairly magical and makes it just another way to do stabby lambda.
It seems like there is no perfect solution here.
I don't feel too strongly about my proposal. Banning block-pass to Kernel#lambda sounds good to me too if others are not comfortable with making proc-to-lambda transformation more accessible.

#11 - 07/09/2019 02:38 PM - Eregon (Benoit Daloze)

alanwu (Alan Wu) wrote:

> Kernel#lambda is in a weird spot. Even though it's a method, making it behave like one has the unfortunate side-effect of allowing proc transformation.

Yes, the semantics of Kernel#lambda have always been a bit weird because no method should be allowed to convert a Proc to a lambda and inversely.
And visually, lambda { ... } receives a block, which is a non-lambda Proc.

My point of view is lambda is a relic of the past, and we should use -> { ... } instead, which has clear semantics and obviously you can't pass a pre-existing block to it.
Yes, I think we should raise lambda(&existing_block) as that can't behave intuitively (it seems either useless or confusing).

#12 - 07/11/2019 05:20 AM - akr (Akira Tanaka)

I'm against it.

Changing lambda <-> proc can violate the intent of programmers who write a block.

It seems good to raise an exception at lambda(&b) where b is (non-lambda) proc.

#13 - 07/11/2019 08:18 AM - matz (Yukihiro Matsumoto)

In my opinion, lambda should return lambda object as a principle. We need to keep the discussion.
Maybe we can generate a delegating lambda object in this case.

Matz.

#14 - 07/11/2019 10:00 AM - knu (Akinori MUSHA)

I don't think the lambda flag should be altered after the creation of a proc, because it's all up to the writer of a block how return/break etc. in it should work.

What about just deprecate lambda in the long run in favor of the lambda literal ->() {}?

#15 - 07/11/2019 11:57 AM - marcandre (Marc-Andre Lafortune)

knu (Akinori MUSHA) wrote:

> I don't think the lambda flag should be altered after the creation of a proc, because it's all up to the writer of a block how return/break etc. in it should work.

akr wrote similar opinion.

This same opinion would call for deprecation of define_method(&block), which I believe would be a mistake.

I am assuming it is clear that lambda(&block) would never mutate block and return a new object that would be a lambda.

#16 - 07/11/2019 04:27 PM - shyouhei (Shyouhei Urabe)

Suppose we change the spec so that lambda(&nonlambda) generates a lambda.
Suppose we have the following nonlambda proc:

```
1: foldr = proc do |x, *xs|
2:   foo(x, foldr.(xs)) if x ||! xs.empty?
3: end
```

A straight-forward foldr implementation, with foo defined elsewhere.
Now let's "convert" it into a lambda:

```
4: foldr = lambda(&foldr)
```

This magically breaks the program. The meaning of |x, *xs| changed.
The problem I see is that the broken program would generate a backtrace like this:

```
Traceback (most recent call last):
        5444: from tmp.rb:2:in `block in <main>'
        5443: from tmp.rb:2:in `block in <main>'
        5442: from tmp.rb:2:in `block in <main>'
        5441: from tmp.rb:2:in `block in <main>'
        5440: from tmp.rb:2:in `block in <main>'
        5439: from tmp.rb:2:in `block in <main>'
        5438: from tmp.rb:2:in `block in <main>'
         ... 5433 levels...
           4: from tmp.rb:2:in `block in <main>'
           3: from tmp.rb:2:in `block in <main>'
           2: from tmp.rb:2:in `block in <main>'
           1: from tmp.rb:2:in `block in <main>'
tmp.rb:2:in `block in <main>': stack level too deep (SystemStackError)
```

There is no single bit of information that the prolem is caused by that lambda conversion.
An end user has no other way than to blame tmp.rb:2 not tmp.rb:4, because the backtrace says so.

This is really bad. Think of for instance the block was from another library. The library authors suddenly get angry bug reports due to some random other guy turned their proc into lambda. This is nonsense.

Non-lambda to lambda conversion disturbs the boundary point of responsibility. Should not be allowed I believe.

#17 - 07/12/2019 04:11 AM - alanwu (Alan Wu)

The idea to generate a delegating lambda seems to side-step a lot of the issues posted here.
For reference here are some quotes from the log in #15930 (courtesy of ko1):

```
b = proc {|x| x }
lambda(&b) #=> lambda {|x| b[x] }

b = proc {|x, y, k:1| x }
lambda(&b) #=> lambda {|x, y, k:1| b[x, y, k:k] }
```

This preserves the return semantics in the original proc. I would imagine it would also add an entry to the call stack.
(side note, I think the default parameter evaluation has to be left to the original proc too since a return could be in there)

What do you folks think about this?

#18 - 07/12/2019 05:20 AM - akr (Akira Tanaka)

- Related to Feature #9777: Feature Proposal: Proc#to_lambda added

#19 - 07/12/2019 05:22 AM - akr (Akira Tanaka)

- Related to Feature #8693: lambda invoked by yield acts as a proc with respect to return added

#20 - 07/12/2019 05:31 AM - shyouhei (Shyouhei Urabe)

Yes, I can compromise with delegation scheme, as long as it leaves its own frame in the call stack. An optional method for a proc to prevent such delegation is desirable but can be added later.

#21 - 07/12/2019 05:43 AM - akr (Akira Tanaka)

Eregon (Benoit Daloze) wrote:

> With current semantics, it returns :early_return.
> With the proposed change, it returns :method_return_value.
> That's very surprising, isn't it?

I agree.

The lambda-ness of Proc object affects control flow: the behavior of "return" and "break".

If lambda(&b) changes the lambda-ness of b, two control flow can exist.
I think no programmer want to consider two control flow when implementing one block.

#22 - 07/12/2019 09:30 AM - Eregon (Benoit Daloze)

alanwu (Alan Wu) wrote:

> The idea to generate a delegating lambda seems to side-step a lot of the issues posted here.

So then, would it be the same semantics as this?

```
b = proc {|x, y, k:1| x }
l = lambda { |*args, **kwargs, &block| b.call(*args, **kwargs, &block) }
```

I think the lambda cannot have the same (in the code) arguments as the proc, otherwise https://bugs.ruby-lang.org/issues/15973#note-16 would fail too.

alanwu (Alan Wu) Can you present use case(s) for turning procs into lambdas?
So far I only see consistency but the delegating lambda is not that consistent with #lambda with a literal block.
Without a good use-case, I think raising on #lambda without a literal block would be the safest and least surprising.

#23 - 07/15/2019 09:19 AM - Eregon (Benoit Daloze)

- Related to Bug #16004: Kernel#lambda captured with Kernel#method doesn't create lambdas added

#24 - 07/17/2019 09:20 PM - alanwu (Alan Wu)

> Can you present use case(s) for turning procs into lambdas?

One use case is reading the parameters of blocks as if they were written for a method definition: https://bugs.ruby-lang.org/issues/9777#note-11
Another one is getting lambda style parameter checking for the incoming block. msgpack-ruby does this through the CAPI:
https://github.com/msgpack/msgpack-ruby/blob/b48f9580bfd33be6a86652f10a41bd691a7d0c91/ext/msgpack/unpacker_class.c#L367

I asked for opinions about the delegation scheme because it seems like a good way to go if we must return a lambda. At the time I didn't make up my mind as to whether it would be a good idea to return one.
I now believe that banning everything but the literal block form lambda is overall better. lambda being a method exposes it to many different ways of calling it, some of which don't have obvious semantics.
We can pick a semantic for lambda(&thing), but there will always be some percentage of users that find the behavior surprising. We can minimizes surprises, but we can't eliminate it, as long
as lambda is a method.

To illustrate, here are two more ways to call into Kernel#lambda that are hard to define good semantics for: super (zsuper) and method(:lambda).call {}. Right now zsuper creates a lambda while
method(:lambda).call {} doesn't. I'm not sure what these should do and different people probably have different ideas.

I think lambda was supposed to be a psudo-keyword but implementing it as a method is unfortunately exposing it to a whole array of usages that were not considered.
If we can promote it to a psudo-keyword, that would be best. Failing that, banning lambda(&thing) at least removes one source of surprise with clear messaging.

#25 - 07/26/2019 05:53 PM - Dan0042 (Daniel DeLorme)

I think the delegating lambda idea doesn't really work. I mean, it doesn't *do* anything. You just get a lambda that behaves like a proc. It just changes the return value of Proc#lambda? which I don't think has any benefit by itself. In fact it might be counterproductive to hide the fact that this Proc object really behaves like a proc and not a lambda.

It seems the main use case for proc->lambda conversion is to validate the Proc defines the correct parameters? But I believe it's quite ok to let the writer of the Proc worry about that. And if you *really* want a lambda you should just enforce it.

```
def foo(lambda)
  lambda.lambda? or raise ArgumentError
  lambda.call(1,2,3)
end
foo -> (x) do
  42
end
#=> ArgumentError (wrong number of arguments (given 3, expected 1))
```

So I haven't yet seen a single good use case for this conversion, and I can't think of one despite trying. You'd need a situation where you want a Proc to have either proc or lambda behavior based on a condition. That's....

On the other hand there's a good case to make for just letting the programmer do what s/he wants. If you want to convert a proc into a lambda presumably you have a reason for doing so, and understand the consequences. The only real danger is if a proc was converted to a lambda *inadvertently* . But I can't come up with a realistic situation where this could occur. All the "surprising" examples I've seen are contrived and in fact not surprising at all. If a method expects a proc and you give it a lambda, that's no different than giving it an Integer or any other object that fails expectations. I honestly can't think of a situation where you'd want lambda(&myproc) to just pass through the proc unchanged; if you don't want to convert you'd just use myproc directly, right?

In the end it seems the only benefit of proc<->lambda conversion is for the sake of consistency of the proc and lambda methods? I guess that makes some sense, because this situation is definitely weird and surprising:

```
lambda{ }                                       #=> #<Proc:0x000055e295d1d648@(irb):1 (lambda)>
class X;def lambda;super;end;end; X.new.lambda{ } #=> #<Proc:0x000055e295d0b0d8@(irb):2 (lambda)>
lambda(&proc{ })                                #=> #<Proc:0x000055e295d08d88@(irb):3>
method(:lambda).call{ }                         #=> #<Proc:0x000055e295d01da8@(irb):4>
def foo(&b);lambda(&b);end; foo{ }              #=> #<Proc:0x000055e295cf4810@(irb):6 (lambda)>
```

#26 - 07/30/2019 08:08 AM - ko1 (Koichi Sasada)

- Assignee set to matz (Yukihiro Matsumoto)

- Status changed from Open to Assigned

does anyone can write a discussion summary? :p

#27 - 07/30/2019 01:31 PM - matz (Yukihiro Matsumoto)

I agree with akr (Akira Tanaka) here, as long as lambda with a block argument warns.
We need to keep compatibility. But I think we should warn this inconsistent behavior.

Matz.

#28 - 07/30/2019 04:32 PM - Dan0042 (Daniel DeLorme)

akr (Akira Tanaka) wrote:

> The lambda-ness of Proc object affects control flow: the behavior of "return" and "break".
>
> If lambda(&b) changes the lambda-ness of b, two control flow can exist.
> I think no programmer want to consider two control flow when implementing one block.

I think everyone can agree with that. The issue I guess is **should it be allowed to define a lambda using block syntax**, with the two main viewpoints being

A. lambda() and define_method() already allow this, so it's a well established pattern. So why not allow my_anonymous_function_dsl{ } ? In this case the one writing the block knows it's supposed to have lambda semantics. In 2.5 it became allowed in certain circumstances but that was apparently unintended (#15620)? A search through major gems showed no incompatibilities. Numerous tickets through the years indicate current behavior is surprising to many.

B. lambda() and define_method() should never have allowed this; proc/lambda semantics should be defined **lexically**. Changing it dynamically is surprising. We can't break compatibility but we should not dig ourselves any deeper. Use my_anonymous_function_dsl&->{ } instead. It's a change in behavior so there may be incompatibility issues.

Does this seem like an accurate summary?

#29 - 08/05/2019 04:04 PM - Dan0042 (Daniel DeLorme)

This may be irrelevant but I would like to point out that proc->lambda conversion was supported in ruby 1.8

```
b = Proc.new{ return 42 }
b.call          #=> LocalJumpError: unexpected return
lambda(&b).call #=> 42
```

I'm not sure if the change was intentional or not, but this feature request is really about restoring lost behavior, not introducing new one. :-)

#30 - 08/05/2019 05:08 PM - sawa (Tsuyoshi Sawada)

- Description updated

- Subject changed from Make it so Kernel#lambda always return a lambda to Let Kernel#lambda always return a lambda

#31 - 09/28/2019 03:46 AM - alanwu (Alan Wu)

I have updated my fix for [Bug #15620] to also issue a warning in cases it returns the argument it receives without modification.

Since both 2.5.x and 2.6.x has [Bug #15620], there may be code in the wild that depend on the unintended lambda conversion behavior. It might make sense to put the 2.4.x behavior back for 2.7.0-preview2 to find out whether it's an issue.

https://github.com/ruby/ruby/pull/2289

#32 - 12/20/2019 12:48 PM - Eregon (Benoit Daloze)

- Related to Bug #15620: Block argument usage affects lambda semantic added

#33 - 12/21/2019 09:09 PM - Eregon (Benoit Daloze)

- Target version set to 36

matz (Yukihiro Matsumoto) Is it OK to warn for this in the next Ruby version?

I think it's going to highlight wrong usages (good) and I would expect there are very few usages of lambda(&existing_proc).

The warnings were implemented in https://github.com/ruby/ruby/pull/2289 by alanwu (Alan Wu) but ko1 (Koichi Sasada) said we should postpone them as it's too late for 2.7:
https://github.com/ruby/ruby/pull/2289#issuecomment-567820054

#34 - 12/24/2019 05:35 PM - ko1 (Koichi Sasada)

This is one idea: how about to prohibit lambda(&...) method call? (block literal is always prohibited)

3.0: deprecation warning and show 3.1 will raise exception for lambda(&...)
3.1: raise exception for lambda(&...)

same as proc.

#35 - 12/24/2019 05:37 PM - ko1 (Koichi Sasada)

more aggressive proposal is obsolete lambda call, and use -> (maybe it is too aggressive).

#36 - 12/25/2019 11:04 AM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote:

> This is one idea: how about to prohibit lambda(&...) method call? (block literal is always prohibited)

I think you mean non-literal block (&) becomes prohibited.

> 3.0: deprecation warning and show 3.1 will raise exception for lambda(&...)
> 3.1: raise exception for lambda(&...)
>
> same as proc.

And whether it's lambda(&...) or lambda { } is detected by the lambda method itself (like for the warning)?

That sounds good to me.

#37 - 12/25/2019 11:13 AM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote:

> more aggressive proposal is obsolete lambda call, and use -> (maybe it is too aggressive).

Sounds even better to me, but I guess that might be disruptive for everyone to adopt -> {} style.
It would be interesting to see what matz thinks about this.

-> exists since at least 2.0, so it shouldn't be a compatibility issue to use it.

From the point of view of a Ruby VM, not having to handle #lambda magically changing the semantics of the passed block would be great, because it's really awkward and hard to check if the block is literal or not (considering lambda can be alias-ed).

It would also simplify the semantics for the user: method_call { ... } is always a non-lambda Proc, and -> {} is the only way for a lambda.

#38 - 12/25/2019 11:15 AM - Eregon (Benoit Daloze)

On a related note I think we should also prohibit define_method(&non_lambda) because that confusingly treats the same block body differently (e.g., the same return in the code means something different).

#39 - 12/25/2019 11:20 AM - zverok (Victor Shepelev)

> On a related note I think we should also prohibit define_method(&non_lambda) because that confusingly treats the same block body differently (e.g., the same return in the code means something different).

Seem it will have an awful impact on any DSLs?..

Like

```
skip_if { |user| user.admin? }

# ...implemented as
def skip_if(&condition)
  define_method(:skip?, &condition)
end
```

#40 - 12/25/2019 11:51 AM - Eregon (Benoit Daloze)

zverok (Victor Shepelev) wrote:

> Seem it will have an awful impact on any DSLs?..

Which is IMHO a perfect example of how dangerous that is.
What if it's skip_if { |user| return 42 }?
Do you expect that to return from the surrounding method or to return from the block?
For any non-lambda Proc it should mean return from the surrounding method, define_method breaks that (by converting a proc to a lambda implictly).

```
# Assuming your definition of skip_if

skip_if { |user| return 42 } # should return from the surrounding method but does not
skip?(1) # but it does not, it just returns from the block!

proc { |user| return 43 }.call # returns from the surrounding method
raise "unreachable" # never reached, as expected
```

There is an easy workaround:

```
def skip_if(&condition)
  define_method(:skip?, -> *args { condition.call(*args) })
end
```

With that skip?(1) returns from the surrounding method, as it should like every other literal block.

#41 - 12/25/2019 07:34 PM - zverok (Victor Shepelev)

Ugh, let me be a bit philosophical here (how many times I promised myself to not engage in discussions about Ruby's "spirit"? I've lost count.)

I understand the point you are speaking from (language implementer's background, who constantly fights with "dangerously illogical" parts), but for me, one of the key points of Ruby's attractiveness is how far it goes to reduce boilerplate in a logical and humane way. There is a non-neglectible gap between "human" consistency and "computer" (formal) consistency.

I am writing in Ruby for >15 years now (and teaching it for >5 years, and blah-blah-blah). I've implemented and used metric shit-ton of DSLs. So, for me and my intuition is experience, it is just this way:

```
skip_if { |user| return 42 }
```

...is something that will behave "unexpectedly"? Yes.
Have I met a lot of cases when it was really written and shot somebody in the head? No.
What if somebody really happens to meet with this case and is surprised? I believe they'll spend some time to wrap they head about it. That's a price we pay to have "complex language for writing simple code".

Now, when I see this:

```
def skip_if(&condition)
  define_method(:skip?, &condition)
end
```

...which "should" be "simply workarounded" to this:

```
def skip_if(&condition)
  define_method(:skip?, -> *args { condition.call(*args) })
end
```

...I see it as "your language implementation is really busy doing its important things. So it will not help you to write clean and obvious code. I (language implementation) see what you want to do (define methods implementation from a block in the variable), and I know how to fix it (block should be converted to lambda), but you (puny human) should do it with your own soft hands".

It can and will harm the will to write simple helping DSLs (which, one may argue, "is a good thing" anyways?..)

Why, then, bother with "all this proc vs lambda nonsense" (as some may, and do, argue)? All in all, "you don't need this complexity at all", right? Because

```
sources.zip(targets).map { |src, tgt| tgt.write(src) }
```

...can be "easily workarounded" as this:

```
sources.zip(targets).map(->(arguments) { arguments[1].write(arguments[0]) } }
```

...and everything suddenly becames more consistent and homogenous...

We can go this way really far, honestly :)

#42 - 12/26/2019 07:53 AM - ko1 (Koichi Sasada)

Ok, make a new ticket about define_method. How about lambda?

Koichi

> 2019/12/26 4:34 zverok.offline@gmail.com :
>
> Issue #15973 has been updated by zverok (Victor Shepelev).
>
> Ugh, let me be a bit philosophical here (how many times I promised myself to not engage in discussions about Ruby's "spirit"? I've lost count.)
>
> I understand the point you are speaking from (language implementer's background, who constantly fights with "dangerously illogical" parts), but for me, one of the key points of Ruby's attractiveness is how far it goes to reduce boilerplate in a logical and humane way. There is a non-neglectible gap between "human" consistency and "computer" (formal) consistency.
>
> I am writing in Ruby for >15 years now (and teaching it for >5 years, and blah-blah-blah). I've implemented and used metric shit-ton of DSLs. So, for me and my intuition is experience, it is just this way:
>
> ```
> skip_if { |user| return 42 }
> ```
>
> ...is something that will behave "unexpectedly"? Yes.
> Have I met a lot of cases when it was really written and shot somebody in the head? No.
> What if somebody really happens to meet with this case and is surprised? I believe they'll spend some time to wrap they head about it. That's a price we pay to have "complex language for writing simple code".
>
> Now, when I see this:
>
> ```
> def skip_if(&condition)
>  define_method(:skip?, &condition)
> end
> ```
>
> ...which "should" be "simply workarounded" to this:
>
> ```
> def skip_if(&condition)
>  define_method(:skip?, -> *args { condition.call(*args) })
> end
> ```
>
> ...I see it as "your language implementation is really busy doing its important things. So it will not help you to write clean and obvious code. I

(language implementation) see what you want to do (define methods implementation from a block in the variable), and I know how to fix it (block should be converted to lambda), but you (puny human) should do it with your own soft hands".

It can and will harm the will to write simple helping DSLs (which, one may argue, "is a good thing" anyways?..)

Why, then, bother with "all this proc vs lambda nonsense" (as some may, and do, argue)? All in all, "you don't need this complexity at all", right? Because

```
sources.zip(targets).map { |src, tgt| tgt.write(src) }
```

...can be "easily workarounded" as this:

```
sources.zip(targets).map(->(arguments) { arguments[1].write(arguments[0]) } }
```

...and everything suddenly becames more consistent and homogenous...

We can go this way really far, honestly :)

Feature #15973: Let Kernel#lambda always return a lambda
https://bugs.ruby-lang.org/issues/15973#change-83402

- Author: alanwu (Alan Wu)
- Status: Assigned
- Priority: Normal
- Assignee: matz (Yukihiro Matsumoto)

* Target version: 2.8

When Kernel#lambda receives a Proc that is not a lambda,
it returns it without modification. I propose to change Kernel#lambda
so it always returns a lambda.

Calling a method called lambda and having it do nothing in effect is
not very intuitive.

https://github.com/ruby/ruby/pull/2262

Judging from marcandre's investigation here: https://bugs.ruby-lang.org/issues/15620#note-1,
changing the behavior should not cause much breakage, if any.

This also happens to fix [Bug #15620]

--
https://bugs.ruby-lang.org/

#43 - 12/26/2019 08:46 PM - Dan0042 (Daniel DeLorme)

one of the key points of Ruby's attractiveness is how far it goes to reduce boilerplate in a logical and humane way. There is a non-neglectible gap between "human" consistency and "computer" (formal) consistency.

+1, very much. I'm glad there's still someone who cares about usefulness over "consistency"

Converting a proc to a lambda is very useful for DSLs. And generally speaking it's empowering to programmers. Maybe a bit dangerous, but I can take care of myself. I really don't need or want to be "protected" from powerful yet "unexpected" behaviors. This ticket was about making lambda always return a lambda; at which point did it become about preventing this from happening?

I understand there's many who are emotionally invested into making ruby stricter and easier to implement/optimize, but at least there should be good alternatives for what the proc->lambda conversion allows.

For DSLs:

```
#a block is specified, and this DSL is documented to have lambda semantics, so we need to convert the proc to lambda
register_lambda(:xyz) do |x,y,z|
  return x+y+z
end

#or a pretty syntax to pass a lambda as a block
register_lambda(:xyz)->(x,y,z) do
```

```
  return x+y+z
end

#because this is ugly, honestly
register_lambda(:xyz, &->(x,y,z) do
  return x+y+z
end)
```

Or what about:

```
block = proc{ |x,y,z=1| }
block.parameters #=> [[:opt, :x], [:opt, :y], [:opt, :z]]
                 # yes I know that all params are optional in a proc,
                 # but I wanted to know which have a default value and which don't
lambda(&block).parameters #=> [[:req, :x], [:req, :y], [:opt, :z]]
                          # this tells me what I wanted to know in a simple and easy way
```

Beyond the argument that this is dangerous (yes it is), there's also the fact that, used wisely, this is sometimes **useful**.

> it's really awkward and hard to check if the block is literal or not (considering lambda can be alias-ed).

Then wouldn't it be really simple if all blocks were converted to lambdas regardless of being literal or not? KISS.

#44 - 01/10/2020 10:21 PM - Eregon (Benoit Daloze)

Yes, sorry I should not have mentioned define_method here, even though it's related it's not the main topic.
I'll make a new issue for it (#16499).

#45 - 01/10/2020 10:36 PM - Eregon (Benoit Daloze)

- Related to Feature #16499: define_method(non_lambda) should not change the semantics of the given Proc added

#46 - 01/16/2020 08:12 AM - ko1 (Koichi Sasada)

Quote from today's devmeeting.

Discussion:

- Proposing to deprecate lambda(&block) (lambda call with Proc object as block)
- Benefits of obsoleting lambda?
  - Simplify for new Ruby developers. (no duplicated syntax.)
  - Suspend for short term (for years)
- Don't want more incompatibilities just after Ruby 2.7
- Migration pass to deprecate lambda(&block).
  1. Print warning at 3.0. (No compatibility layer.)
     - “The lambda call is meaningless. Delete it.”
     - “Delete meaningless lambda”
  2. Raises error at 3.1 (or 3.2, ...)

Conclusion:

- lambda(&b) will be prohibited.
  - Print warning at 3.0. (No compatibility layer.)
  - Raises error at 3.1 (or 3.2, ...)

#47 - 06/11/2020 02:31 PM - jeremyevans (Jeremy Evans)

- Status changed from Assigned to Closed

Applied in changeset git|2188d6d160d3ba82432c87277310a4d417e136d5.

Warn when passing a non-literal block to Kernel#lambda

Implements [Feature #15973]

#48 - 09/29/2020 03:37 AM - hsbt (Hiroshi SHIBATA)

- Target version changed from 36 to 3.0