Ruby master - Feature #8237

Logical method chaining via inferred receiver

04/08/2013 02:10 PM - wardrop (Tom Wardrop)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

=begin
This is a feature suggestion that was raised while discussing issue #8191. The feature suggestion is to introduce some form of logical method chaining to address this reasonably common pattern:

```
user && user.profile && user.profile.website && user.profile.website.thumbnail
```

It would be reasonably trivial to shorten this to:

```
user && .profile && .website && .thumbnail
```

The implementation I propose would be for Ruby to allow an inferred receiver; the dot prefix would be the syntax for this. The inferred receiver would resolve to the result of the last expression in the current scope. For illustrative purposes, the following would work under this proposal:

```
"some string"
puts .upcase #=> SOME STRING
```

Another example:

```
puts .upcase if obj.success_message || obj.error_message

# Instead of...

message = (obj.success_message || obj.error_message)
puts message.upcase if message
```

This can also potentially provide an alternative option in syntactically awkward scenario's, such as dealing with the return value of an if statement or a catch block, avoiding the need for temporary variable assignment:

```
catch :halt do
  # Do something
end

if .nil?
   log.info "Request was halted"
   response.body = "Sorry, but your request could not be completed"
end
```

The logical chaining scenario is the main use case however. I just wanted to demonstrate how the proposed implementation could also be used in other creative ways.

=end

Related issues:

Related to Ruby master - Feature #8246: Hash#traverse	**Closed**	**04/11/2013**
Related to Ruby master - Feature #8191: Short-hand syntax for duck-typing	**Closed**	
Related to Ruby master - Feature #11537: Introduce "Safe navigation operator"	**Closed**	

History

#1 - 04/08/2013 09:39 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I really believe it would be better to isolate each wanted feature in a separate ticket instead of mixing several ideas in the same ticket. Also, I guess you should state clearly that a&&.b&&.c should mean (tmp1 = a; tmp1 && (tmp2 = tmp1.b); tmp2 && tmp2.c), meaning that each method should be evaluated just once in the chain.

Anyway, I'm -1 for this proposal (with the implementation above) because it doesn't make any sense to me to call any method on the "false" object in most real-world codes out there. I'm favorable to adding a shortcut syntax for chaining methods, but I believe it should check for "nil?" only. I don't have an opinion yet if it should abort the entire statement if a subexpression is nil, like CoffeeScript does, or simply replace the subexpression with "nil" and let the expression go on with the evaluation, like Groovy does.

#2 - 04/08/2013 09:51 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

By the way, I'm also -1 for using an inferred receiver as the last result as it leads to less readable code and disallows other future language improvements requests that otherwise could be impossible to accept due to backward compatibility issues with regards to this inferred receiver feature.

#3 - 04/08/2013 09:53 PM - phluid61 (Matthew Kerwin)

On Apr 8, 2013 3:10 PM, "wardrop (Tom Wardrop)" tom@tomwardrop.com wrote:

> This is a feature suggestion that was raised while discussing issue
> #8191. The feature suggestion is to introduce some form of logical method
> chaining to address this reasonably common pattern:
>
> `user && user.profile && user.profile.website &&`
>
> user.profile.website.thumbnail
>
> It would be reasonably trivial to shorten this to:
>
> `user && .profile && .website && .thumbnail`
>
> The implementation I propose would be for Ruby to allow an inferred
> receiver; the dot prefix would be the syntax for this. The inferred
> receiver would resolve to the result of the last expression in the current
> scope.

What would be the result of:

```
"abc"
foo = lambda { .upcase }
"def"
foo.call
```

Is it "ABC" ?

#4 - 04/08/2013 10:23 PM - phluid61 (Matthew Kerwin)

On Apr 8, 2013 10:39 PM, "rosenfeld (Rodrigo Rosenfeld Rosas)" <
rr.rosas@gmail.com> wrote:

> I really believe it would be better to isolate each wanted feature in a
> separate ticket instead of mixing several ideas in the same ticket. Also, I
> guess you should state clearly that a&&.b&&.c should mean (tmp1 = a; tmp1
> && (tmp2 = tmp1.b); tmp2 && tmp2.c), meaning that each method should be
> evaluated just once in the chain.
>
> Anyway, I'm -1 for this proposal (with the implementation above) because
> it doesn't make any sense to me to call any method on the "false" object in
> most real-world codes out there. I'm favorable to adding a shortcut syntax
> for chaining methods, but I believe it should check for "nil?" only. I
> don't have an opinion yet if it should abort the entire statement if a
> subexpression is nil, like CoffeeScript does, or simply replace the
> subexpression with "nil" and let the expression go on with the evaluation,
> like Groovy does.

I think you've missed the point of this request. There is only one wanted
feature, and it has nothing to do with "checking for nil", except that it
can be used to simplify a common pattern already employed for that purpose.

Note that this one also theoretically allows such constructs as:

```
foo[bar] || .inspect
baz if .empty?
```

or

```
"41" << .to_i(16).chr  #=> "41A"
```

Strange examples, but possible nonetheless.

#5 - 04/08/2013 10:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 08-04-2013 10:07, Matthew Kerwin escreveu:

> On Apr 8, 2013 10:39 PM, "rosenfeld (Rodrigo Rosenfeld Rosas)"
> > wrote:
>
> > I really believe it would be better to isolate each wanted feature
> > in a separate ticket instead of mixing several ideas in the same
> > ticket. Also, I guess you should state clearly that a&&.b&&.c should
> > mean (tmp1 = a; tmp1 && (tmp2 = tmp1.b); tmp2 && tmp2.c), meaning that
> > each method should be evaluated just once in the chain.
> >
> > Anyway, I'm -1 for this proposal (with the implementation above)
> > because it doesn't make any sense to me to call any method on the
> > "false" object in most real-world codes out there. I'm favorable to
> > adding a shortcut syntax for chaining methods, but I believe it should
> > check for "nil?" only. I don't have an opinion yet if it should abort
> > the entire statement if a subexpression is nil, like CoffeeScript
> > does, or simply replace the subexpression with "nil" and let the
> > expression go on with the evaluation, like Groovy does.
>
> I think you've missed the point of this request. There is only one
> wanted feature, and it has nothing to do with "checking for nil",
> except that it can be used to simplify a common pattern already
> employed for that purpose.
>
> Note that this one also theoretically allows such constructs as:
>
> ```
> foo[bar] || .inspect
> baz if .empty?
> ```

I didn't miss the point, I'm simply -1 for it. I see the case for
chain-able calls and I do use it sometimes when coding in Groovy and
CoffeeScript. But I can't find this proposal will lead to more readable
code and I'm unsure what problem exactly it is trying to solve.

> or
>
> ```
> "41" << .to_i(16).chr  #=> "41A"
> ```
>
> Strange examples, but possible nonetheless.

Exactly. If you could present us more concrete examples, maybe you could
change my mind.

#6 - 04/08/2013 11:45 PM - trans (Thomas Sawyer)

There are at least three other ways to approach this:

```
  user.try.profile.try.website.try.thumbnail
```

```
  user.trying{ |u| u.profile.website.thumbnail }
```

```
  user.trying.profile.website.thumbnail.resolve
```

It occurs to me the first could be fairly concise if we think of #s as possessive:

```
  user.s.profile.s.website.s.thumbnail
```

Albeit it looks a bit odd.

#7 - 04/08/2013 11:53 PM - trans (Thomas Sawyer)

There are at least three other ways to approach this:

```
   user.try.profile.try.website.try.thumbnail
```

```
   user.trying{ |u| u.profile.website.thumbnail }
```

```
  user.trying.profile.website.thumbnail.resolve
```

It occurs to me the first could be fairly concise if we think of #s
as possessive:

```
  user.s.profile.s.website.s.thumbnail
```

Albeit it looks a bit odd.

#8 - 04/09/2013 12:59 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 08-04-2013 11:50, Trans escreveu:

> There are at least three other ways to approach this:
>
> ```
>   user.try.profile.try.website.try.thumbnail
> ```
>
> ```
>   user.trying{ |u| u.profile.website.thumbnail }
> ```
>
> ```
>   user.trying.profile.website.thumbnail.resolve
> ```
>
> It occurs to me the first could be fairly concise if we think of #s
> as possessive:
>
> ```
>   user.s.profile.s.website.s.thumbnail
> ```
>
> Albeit it looks a bit odd.

I'd probably use _? instead of "s" for the andand approach:

https://github.com/raganwald/andand

class NullClass
def method_missing(name, *args, &block); nil; end
end

module NullPattern
def _?; nil? ? NullClass.new : self; end
end

BasicObject.send :include, NullPattern

user.?.profile.?.website._?.thumbnail

But anyway, this is kind of how it currently works using Groovy's
pattern... What I don't understand is how you're thinking about
implementing the CoffeeScript pattern:

user.trying.profile.website.thumbnail.resolve

By the way, I'd write it as:

user._?.profile.website.thumbnail.resolve

You can't inherit NullClass from NilClass. Also, you can't force an
instance to behave like a falsey value in Ruby. Avdi made a great job
explaining the reasons:

http://devblog.avdi.org/2011/05/30/null-objects-and-falsiness/

So, how the "trying" approach would be implemented?

Cheers,
Rodrigo.

#9 - 04/09/2013 05:57 AM - trans (Thomas Sawyer)

=begin
As long as you always "resolve" it should be okay. This is untested, but it's going to be something along the lines of:

```
class AndAnd < BasicObject
  def initialize(receiver)
    @receiver = receiver
    @calls = []
```

```
  end
  def method_missing(s, *a, &b)
    @calls << [s, a, b]
    self
  end
  def resolve
    @calls.inject(@receiver) do |r, (s, a, b)|
     x = r.send(s, *a, &b)
     break nil if x.nil?
     x
  end
end

def trying
  AndAnd.new(self)
end
```

Btw, _? isn't half bad. I wonder if it could just support ?:

```
user.?.profile.?.website.?.thumbnail
```

Or via "trying":

```
user.?.profile.website.thumbnail.!
```

=end

#10 - 04/09/2013 11:04 AM - wardrop (Tom Wardrop)

=begin
phluid61 (Matthew Kerwin) In your example...

```
"abc"
 foo = lambda { .upcase }
 "def"
 foo.call
```

I would imagine that to either produce an error along the lines of "could not infer receiver" or "inferred receivers must be preceded by a valid expression", or we believe it's logical to default the inferred receiver or nil, it'll produce a NoMethodError.

Perhaps the inferred receiver should behave as if it was assigned to a variable, e.g.

```
 last_expression = "abc"
 last_expression = foo = lambda { last_expression = last_expression.upcase }
 last_expression = "def"
 last_expression = foo.call
```

I'd prefer it to be scoped though, so anywhere a local variable can be redefined, last_expression should be essentially reset. Something like this...

```
last_expression = "abc"
last_expression = foo = lambda do
    last_expression = nil
    last_expression = last_expression.upcase
end
last_expression = "def"
last_expression = foo.call
```

I think I'd prefer it to behave like that, but instead of defaulting to ((|nil|)), have it default to some special internal value that indicates that the last expression hasn't be set, and to raise an appropriate error if it's used.

I'm not sure how difficult this would be to implement from a performance perspective though. I imagine it would be expensive to assign the result of every expression to a variable, so hopefully it could be optimised in a way that it's only stored when an inferred receiver is used in the next expression.

@rosenfield I believe what you're after is different to this request. This is about an extension to a particular pattern involving logical AND, as well as proposing an implementation that has other practical applications. The behaviour you want would result in a completely different proposal that solves a similar but logically different problem. Feel free to raise such a proposal as a feature request and link to it though; maybe you could propose the introduction of a new logical operator. I'd just like to keep that discussion somewhat independent of this one. The two proposal's can then be more easily compared if they overlap.

I'd also like to add that some of the other suggestions make the desired behaviour unclear. The #try method and question-mark prefix suggest they have something to do with calling a potentially non-existant method, like discussed in issue #8191. I think it makes more sense to introduce something that can be used with pre-existing language constructs such as logical AND.
=end

#11 - 04/09/2013 02:29 PM - henry.maddocks (Henry Maddocks)

I don't support this because inferred or implicit variables are a 'Perlism' that we should avoid. It makes programs difficult to understand and it is easy to add subtle bugs that are hard to track down.

The sample usages are poor code and your proposal doesn't improve understanding.

```
user && user.profile && user.profile.website && user.profile.website.thumbnail
```

What is the intent of this line? Chaining methods this way should be discouraged, not encouraged.

```
"some string".upcase #=> SOME STRING
```

I saved you 6 characters.

These two are buggy

```
(obj.success_message || obj.error_message).tap do |message|
  puts message.upcase if message
end
```

And if they're both nil?

```
catch :halt do
  throw :halt, "For reasons"
end

if .nil?
  log.info "Request was halted"
  response.body = "Sorry, but your request could not be completed"
end
```

Were we halted or not? The chance of someone innocently inserting an expression between the catch and the if and not noticing are very high.

As the Avdi Grimm article link above concluded, writing better code is the solution. In Ruby, if your code is ugly then that means you're doing it wrong.

#12 - 04/09/2013 05:58 PM - wardrop (Tom Wardrop)

=begin
Don't pick apart the trivial examples too much Henry. There's often situations where the return value may be nil or false, but which when truthy, you want to do some kind of operation on; the long method chain is just one example. As an even simpler example, let's say you expect a string to be returned from a method call, but there's a circumstance in which that call may result in nil. You can't say that this pattern is uncommon, and you can't say that it's bad code...

```
options[:title].upcase! if options[:title]
```

This suggestion is about making the double-call unnecessary, and more succinct. If fetching ((|:title|)) is an expensive operation or has side effects, you don't want to call it twice. In this case, the above example turns into either of these:

```
title.upcase! if title = options[:title]
```

```
options[:title].tap { |v| v.upcase! if v }
```

With this proposal, it becomes whichever of these you prefer...

```
.upcase! if options[:title]
```

```
options[:title] && .upcase!
```

That's the basic premise. There are many other similar scenario's, some of which have already been given. I don't think the potential for a feature to be abused or used incorrectly is a good enough reason to reject it. Ruby has the most potential for abuse out of any language I've seen or used. Ruby allows you to redefine even the most critical classes and methods at runtime - does that mean we should remove that ability? If the only valid use cases for a feature serve as examples of bad coding, then that's different, but I don't believe that's the case for this feature.

=end

#13 - 04/10/2013 05:31 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Tom, I can't point you any references from the top of my head but I'm almost sure about reading Matz himself saying that he likes to see feature requests born from real code needs. He prefers reading real-world code before deciding if the feature is worth or not. And he doesn't seem to be the only one to think this way as I've read this statement from many more people, including me.

That's why we're suggesting you to try to find good code examples when you suggest any features. It certainly requires more thinking about the subject but it also helps you to get your features accepted when they're based on real-world requirements. When I submit new feature requests I always spend some time looking for real code in my applications where I would find that feature useful and try to use them as examples. Of course, this is just an advice and you may follow it or not, but you'll often hear complaints about your poor examples if you don't take this time.

#14 - 04/12/2013 05:01 AM - headius (Charles Nutter)

As a feature that affects all Ruby implementations, this should probably move to CommonRuby: https://bugs.ruby-lang.org/projects/common-ruby

#15 - 04/15/2013 12:02 PM - henry.maddocks (Henry Maddocks)

wardrop (Tom Wardrop) wrote:

> =begin
> Don't pick apart the trivial examples too much Henry..
>
> =end

If you want your request to be taken seriously your example usage should show that you have thoroughly considered how this feature will be used, how it will effect existing code, and the edge cases that might crop up. It's our job to point out issues we see with what you have presented.

So, yes, I will pick apart your trivial examples until you demonstrate that you have thought the feature through.

#16 - 04/18/2013 01:29 AM - headius (Charles Nutter)

I believe this has been suggested and rejected before. It's similar to Groovy's ?. operator, which only proceeds with calls if the LHS is non-null. That particular syntax is incompatible with Ruby since methods can end in ? but a different syntax could be considered.

In the end I believe it still leads to bad code. You're ignoring the fact that you have nulls/nils in your system, which just leads to their propagation even further.

What I could see is a syntax that behaves sort of like instance_eval but without a closure or instance_eval's other quirks. So this code:

object.{foo && bar && baz}

would treat the fcalls and vcalls (calls without a receiver) as calls against the original object, and would be compiled/parsed like:

object.foo && object.bar && object.baz

Note that the example above is basically just like an instance_eval block call without the instance_eval. It also has a sort of "glob" feel to it.

#17 - 04/18/2013 01:42 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Charles, I understand that you don't like this code style. But I'm pretty sure you consider bad several other code styles used on Ruby by people out there, right? This is what I like about Ruby. It doesn't try to provide you a single obvious way of doing something. It just give you the tools and let you decide what pattern to use.

These are are some places I use such patterns in the Grails application I maintain:

def jsonData = [
...
lawFirm: transaction.lawFirm?.name, // instead of transaction.lawFirm ? transaction.lawFirm.name : null
...
]

if (grailsApplication.config.myappname?.logging?.enabled) ...

Honestly I don't see any problems with code like this and I'd love to be able to use something like this in ruby. Using "??" seems like a good trade-off to me, specially because I'd never use triple question marks (???) as it doesn't make sense to check if a boolean value is null. And if your method ending with a question mark isn't returning a boolean value, this is another issue that I'd fix in the method instead of using a triple question mark.

#18 - 04/18/2013 01:26 PM - Anonymous

Wow, inferred reciever, cool. But wait, in Ruby, there is already self! When you type often

fred.do_this && fred.do_that

why not teach fred

fred.do_it_all

How about if we finally allow Japanese into Ruby along with English, and introduce #□ for
temporary change in self (as headius hinted), #□(other) for replacing self temporarily by
[*self, other], and #□ for resseting inferred receiver back to self?

object.□; x = foo && bar && baz; y = quux + 42; □
result = user.□ && profile.□ && website.□ && thumbnail; □

We all need to learn typing Unicode efficiently nowadays, and kana would save space...

When at it, seeing Ruby message as a NL sentence, why not have also implicit verb?

fred.do_it_all
bill._   # bill do the same as fred

Or implicit arguments?

fred.do_A(foo: 1, bar: 2)
bill.do_B(_)   # bill use same args as fred

Or why reinvent NL grammar so slowly? Why just subject.verb(*args) { ... } ? Why not:

!!! Brainstorming ahead !!!

.   { "block" }

Let's be thorough. Let's ask Matz to quit squatting on and, or, if, then etc. so we have
more freedom in NL-like sentences such as:

fred.▯; hand me a new roll of paper; ▯

#19 - 04/19/2013 08:53 PM - duerst (Martin Dürst)

On 2013/04/18 1:42, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Issue #8237 has been updated by rosenfeld (Rodrigo Rosenfeld Rosas).
>
> Charles, I understand that you don't like this code style. But I'm pretty sure you consider bad several other code styles used on Ruby by people out there, right?

Yes. But my understanding is that in general, those things that have a
compact syntax have that compact syntax not just because there's some
occasionally useful (but maybe not generally recommended) coding style
that uses it, but because there are some uses/coding styles that are
very frequent and/or highly preferable where the syntax can be used.

So "I really want this to be a short syntax because occasionally I'm too
lazy to do the better thing" isn't good enough of an argument.

If you can find a reasonably frequent use case where the short syntax is
the preferred/better way to do things, then that's a good argument for
introducing a new syntax. Then once that syntax is around, of course
it's difficult to forbid people to use it, even in cases where it may
not be the "best" way to do things according to OO or some other theory.

> This is what I like about Ruby. It doesn't try to provide you a single obvious way of doing something.

Did anybody say that they want to forbid you to write
user&& user.profile&& user.profile.website&&
user.profile.website.thumbnail ?

No. It's just that this kind of style isn't important enough and/or good
enough for them to introduce new, shorter syntax for.

Let's take a different example. Ruby has global variables. We all know
that global variables, in general, are a bad idea. Ruby happens to have
global variables because Perl had them. If not for that, would Ruby have
global variables? I don't know.

But I'm quite sure that somebody coming and saying something along the
lines of "I want global variables, because it's a kind of coding style
(not necessarily exemplary, I know), and ruby allows different coding
styles, so why not mine." that wouldn't be a winning argument.

Regards,   Martin.

> ---
>
> Feature #8237: Logical method chaining via inferred receiver
> https://bugs.ruby-lang.org/issues/8237#change-38661
>
> Author: wardrop (Tom Wardrop)

Status: Open
Priority: Normal
Assignee:
Category:
Target version:

#20 - 04/19/2013 11:15 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Exactly, Martin. That was exactly my point. It is not because Charles don't like this particular code styling that it makes it unuseful for everyone out there. I use this pattern a lot in both Groovy and CoffeeScript and I don't find it to be a bad code styling. I even showed you a few samples of how they're being used in my own code.

And the pattern isn't a && a.b && a.b.c, but (a.nil ? nil : (a.b.nil? ? nil : a.b.c)).

#21 - 04/22/2013 09:52 PM - Anonymous

> And the pattern isn't a && a.b && a.b.c, but (a.nil ? nil : (a.b.nil? ? nil : a.b.c)).

class NullObject
def method_missing sym, *a, &b; self end
def to_s; "null" end
end
class Integer # patch it with #b, #c
def b; self + 1 end
def c; self + 10 end
end

let's say a number does not qualify if not > 42...

classify = lambda { |arg| arg > 42 ? arg : NullObject.new }

and let's define a function that chains #b and #c to its argument

f = lambda { |arg| arg.b.c }

composition of classify and f:

l = lambda { |arg| f.(classify.(arg)) }

null object now allows bold messaging without fear of object "absence"

l.(42) #=> null
l.(43) #=> 54
l.(32) + 10 #=> null
l.(l.(2) ** 64 - 1)) #=> null
etc.

Thus, "try before do" question becomes "as patching NilClass is EVIL, shall we somehow support null object out of the box?"

#22 - 04/23/2013 02:26 AM - enebo (Thomas Enebo)

I don't know how many people care about Law of Demeter, but this operator would encourage violating it. Does Ruby want to actively help violate it?

#23 - 04/23/2013 06:34 AM - parndt (Philip Arndt)

enebo (Thomas Enebo) I care about the Law of Demeter too and seriously don't like that this encourages ignoring sound practice so freely.

I feel that this 'syntax' exposes Ruby developers and existing software to far more problems than it solves and the OP hasn't demonstrated a legitimate reason for inclusion yet as far as I'm concerned.

The best argument appears to be along the lines of "don't suppress my coding style!" :-)

#24 - 04/23/2013 07:40 AM - phluid61 (Matthew Kerwin)

I don't want to detract from the request/discussion too much, but I have a question about whether the Law of Demeter applies to structured data provided by a third party (e.g. a data API). The other day I wrote the following code in a real (PHP) application:

$status_data = $API->get_branch($branch['alias'], 'status');
if ($status_data && ($branch_list=$status_data['branches']) &&
($branch_data=$branch_list[0])) {
$status = $branch_data['status'];
} else {
$status = 'unknown';
}

(The API object is essentially performing a curl request and parsing a JSON object.) I know it could be structured better, with explicit exceptional-case handling and whatnot, but the end result is that the user wants to see the word "open" or "closed" or "unknown." My handler has access to the full schema describing the 'status_data' structure, including knowledge that it may or may not include certain elements, and the semantics of what those omissions mean. Does LoD apply? Should I have spent the time building a full object model to encapsulate the data structure so I could define meaningful "shallow" accessors instead of "reaching through" objects?

Note: my having written this code does not mean I'm for or against the proposal, this is just a discussion point. As you can see, it wasn't too much skin off my nose to write the if statement.

#25 - 04/23/2013 08:59 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I don't care about Law of Demeter, I only care about how good my code reads. If it reads nice even though it violates LoD, then it's fine. Most of the time my code respects LoD, but that just happens by accident, not because I target it.

#26 - 04/24/2013 09:10 AM - wardrop (Tom Wardrop)

=begin
Law of Demeter is blurry in Ruby. In Ruby, almost everything is a method call. Technically, (({5 + 5 + 5})) is a violation of the Law of Demeter if it's to be taken literally. The fact that Ruby also uses plain-old methods as accessors further demonstrates the issues with applying the Law of Demeter to Ruby. In the right context, Law of Demeter could certainly be a helpful rule of thumb, but it can't be applied as some kind of avoid-at-all-costs global rule. I believe Ruby is too broad and generic to be enforcing laws like that of Demeter when designing language features. Ruby can be used as an awk replacement, scripting language, or for fully stacked applications.

The primary purpose of this proposal is to avoid unnecessary temporary variable assignment and extra expressions. This was good enough justification for the introduction of #tap, so perhaps it's a good enough reason to be considering this proposal. Which this in mind, perhaps the logic for this proposal should be changed. Perhaps, instead of the return value of the last expression being the inferred receiver, it may be more useful for the inferred receiver to correspond to the last object to be operated on, i.e the last method receiver. To re-use the method chaining example (which I'll remind is only one potential use case):

```
user.nil? && !.profile.nil? && !.website.nil? && .thumbnail
```

This allows for potentially more use cases. For example, how often do you see patterns similar to the following?

```
.to_s if my_symbol.is_a?(Symbol)
```

If you could reference this inferred receiver without invoking a method call, it could be even more useful. This would probably require a change to the implementation, so that instead of the receiver being implied, a special variable instead holds the last object to be operated on, such as the underscore. Here's another common pattern:

```
return type if type =~ /^[a-z]/
# could be rewritten as...
return _ if type =~ /^[a-z]/
```

You may wonder what the benefit there is, but imagine something like this:

```
return _ if self.options[:email][:server] == /localhost|127\.0\.0\.1/
```

I hope this demonstrates that there's potential for something to be done here.
=end

#27 - 04/24/2013 12:10 PM - phluid61 (Matthew Kerwin)

wardrop (Tom Wardrop) wrote:

> =begin
> The primary purpose of this proposal is to avoid unnecessary temporary variable assignment and extra expressions. This was good enough justification for the introduction of #tap, so perhaps it's a good enough reason to be considering this proposal. Which this in mind, perhaps the logic for this proposal should be changed. Perhaps, instead of the return value of the last expression being the inferred receiver, it may be more useful for the inferred receiver to correspond to the last object to be operated on, i.e the last method receiver. To re-use the method chaining example (which I'll remind is only one potential use case):
>
> ```
> user.nil? && !.profile.nil? && !.website.nil? && .thumbnail
> ```
>
> This allows for potentially more use cases. For example, how often do you see patterns similar to the following?
>
> ```
> .to_s if my_symbol.is_a?(Symbol)
> ```

> If you could reference this inferred receiver without invoking a method call, it could be even more useful. This would probably require a change to the implementation, so that instead of the receiver being implied, a special variable instead holds the last object to be operated on, such as the underscore. Here's another common pattern:
>
> ```
> return type if type =~ /^[a-z]/
> # could be rewritten as...
> return _ if type =~ /^[a-z]/
> ```
>
> You may wonder what the benefit there is, but imagine something like this:
>
> ```
> return _ if self.options[:email][:server] == /localhost|127\.0\.0\.1/
> ```
>
> I hope this demonstrates that there's potential for something to be done here.
> =end

I don't like the "last object to receive a method" idea. For one you'd have to refine it to "the last object receive a method in the current scope", possibly with a "but not in the current expression" clause (depending on how a.b(_) is parsed); and clarify what is a method and what is not (e.g. & vs &&); and now there's so much mental baggage going along with it you lose a bunch of understandability to gain a little bit less typing.

And I particularly dislike those examples; there's too much obscure syntax magic going on, it's hard to get a feel for the line at a glance. With the if-modifier I'd really prefer the magic to come later *in lexical order*, e.g.:

return self.options[:email][:server] if _ =~ /localhost|127.0.0.1/

I have no idea how that would possibly be defined or implemented.

Since you brought up #tap, I'll note that that line could be represented almost as well (and without duplicated method calls) using:

self.options[:email][:server].tap{|s| return s if s =~ /localhost|127.0.0.1/ }

I find myself drifting more to the -1 side for this feature.

All that said, I quite like the idea of a magic variable that holds "the value of the last evaluated expression", in lexical order. Rather than _ I'll use $% in some examples:

Rodrigo will hate this one, but I don't care:
a && $%.b && $%.c && $%.d

foo = 1
bar = ->{ foo + 1 }
baz = ->{ $% + 1 }
foo = 99
bar[] # => 100
baz[] # => 2

However I'm not entirely convinced of its widespread usefulness.

#28 - 04/24/2013 05:53 PM - regularfry (Alex Young)

On 24/04/13 01:10, wardrop (Tom Wardrop) wrote:

> Issue #8237 has been updated by wardrop (Tom Wardrop).
>
> Law of Demeter is blurry in Ruby. In Ruby, almost everything is a
> method call. Technically, (({5 + 5 + 5})) is a violation of the Law
> of Demeter if it's to be taken literally.

No it isn't. The LoD talks about types, not individual receivers.
You've only got one type there. Getting more nitpicky, it's actually
impossible to tell whether that's a violation without context because we
don't know if Fixnum is already known to that scope, although core types
are generally given a free ride.

> The fact that Ruby also
> uses plain-old methods as accessor further demonstrates the issues
> with applying the Law of Demeter to Ruby. In the right context, Law
> of Demeter could certainly be a helpful rule of thumb, but it can't
> be applied a some kind of avoid-at-all-costs global rule. I believe
> Ruby is too broad and generic to be considering laws like that of
> Demeter when designing language features.

Yep. While LoD is useful, it's important to know a) when it shouldn't

apply, and b) what it actually is. It is *not* a polemic against method
chaining. I agree that it should not be a consideration in this case.

--
Alex

#29 - 04/24/2013 07:33 PM - phluid61 (Matthew Kerwin)

phluid61 (Matthew Kerwin) wrote:

> I find myself drifting more to the -1 side for this feature.
>
> All that said, I quite like the idea of a magic variable that holds "the value of the last evaluated expression", in lexical order. Rather than _ I'll use $% in some examples:
>
> ```
> # Rodrigo will hate this one, but I don't care:
> a && $%.b && $%.c && $%.d
>
> foo = 1
> bar = ->{ foo + 1 }
> baz = ->{ $% + 1 }
> foo = 99
> bar[] # => 100
> baz[] # => 2
> ```
>
> However I'm not entirely convinced of its widespread usefulness.

Apologies for replying to myself, but I've just convinced myself against this whole proposal. In the above code I originally had the baz lambda before the bar, but decided to swap them at the last minute. Only now has it occurred to me that this will break the code, because $% in baz will actually refer to the 'bar' lambda object. No matter what syntax is used, I'm convinced now that it will always be better to just create a local (explicitly named) variable.

(x=a) && (x=x.b) && (x=x.c) && x.d
!(x=a).nil? && !(x=x.b).nil? && !(x=x.c).nil? && x.d
type.tap{|x| return x if x =~ /$^{[a-z]/}$ }

These all work even if #a, #b, #c, #d, and #type have side-effects.

Sorry for the noise.

#30 - 04/24/2013 08:58 PM - Anonymous

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> So, how the "trying" approach would be implemented?

I overlooked the earlier null pattern discussion here. Apologies. I read Avdi's text months
ago, and I remember Avdi sayin, that null falseyness is "chasing the wind". The problem might
be with falsey nil. Currently, true is like "good", and false "bad": If a statement can't return
a "good thing" (and returns nil), "bad" is assumed. Falseyness of nil introduces inequality
between true and false, and forces nil to fulfill two disparate roles: That of "bad thing", and
that of null object.

So imho, I am against syntactic features, that would bring nil closer to the role of ersatz null
object. Users with null object needs should use null object explicitly, rather than ask to make
nil better fake null. That is, unless they are ready to break the taboo and ask Matz to make nil
"good", with the avalanche of changes and backwards incompatibility it would cause. (More apologies
for daring to discuss such a deep stuff here.)

I personally like null object idea, but fear to use it in practice. I fear they will slow things
down, bloat code, and whatnot. For example, with nil, I can say:

@instance_variable ||= :default

With null object, I have to say

@instance_variable.null? ? :default : @instance_variable

If these fears can be addressed, if Ruby creators can give users clear pragmatic encouragement
to use explicit null pattern where called for, much of the problem discussed here would go away.

#31 - 04/27/2013 12:27 PM - henry.maddocks (Henry Maddocks)

wardrop (Tom Wardrop) wrote:

> =begin
> The primary purpose of this proposal is to avoid unnecessary temporary variable assignment and extra expressions. This was good enough justification for the introduction of #tap, so perhaps it's a good enough reason to be considering this proposal.
> =end

Object#tap wasn't introduced to avoid unnecessary temporary variable assignments. #tap is an implementation of the K combinator which has lots of uses beyond avoiding unnecessary temporaries. Also it was just an addition of a new method NOT a change to the language.

#32 - 09/18/2015 09:22 AM - akr (Akira Tanaka)

- Related to Feature #8191: Short-hand syntax for duck-typing added

#33 - 09/18/2015 09:52 AM - akr (Akira Tanaka)

- Related to Feature #11537: Introduce "Safe navigation operator" added

#34 - 11/12/2015 01:16 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Open to Closed

this feature is supported at #11537