Ruby trunk - Misc #15723

Reconsider numbered parameters

03/22/2019 01:11 PM - sos4nt (Stefan Schüßler)

Status:	Feedback	
Priority:	Normal	
Assignee:		

Description

I just learned that *numbered parameters* have been merged into Ruby 2.7.0dev.

For readers not familiar with this feature: it allows you to reference block arguments solely by their *index*, e.g.

```
[1, 2, 3].each { |i| puts i }

# can become

[1, 2, 3].each { puts @1 }
```

I have an issue with this new feature: I think **it encourages sloppy programming** and results in **hard to read code**.

The original proposal was to include a special variable (or keyword) with a **readable name**, something like:

```
[1, 2, 3].each { puts it }

# or

[1, 2, 3].each { puts this }
```

Granted, that looks quite lovely and it actually speaks to me – I can *understand* the code. And it fits Ruby: (quoting the website)

> [Ruby] has an elegant syntax that is natural to read and easy to write.

But the proposed it / this has limited application. It's only useful when dealing with a single argument. You can't have multiple it-s or this-es. That's why @1, @2, @3 etc. were chosen instead.

However, limiting the usefulness to a single argument isn't bad at at. In fact, a single argument seem to be the limit of what makes sense:

```
h = Hash.new { |hash, key| hash[key] = "Go Fish: #{key}" }

# vs

h = Hash.new { @1[@2] = "Go Fish: #{@2}" }
```

Who wants to read the latter? That looks like an archaic bash program (no offense). We already discourage Perl style $-references: (from The Ruby Style Guide)

> Don't use the cryptic Perl-legacy variables denoting last regexp group matches ($1, $2, etc). Use Regexp.last_match(n) instead.

I don't see how our code can benefit from adding @1 and @2.

Naming a parameter isn't useless – it gives context. With more than one parameter, naming is crucial. And yes, naming is hard. But avoiding proper naming by using indices is the wrong way.

So please reconsider numbered parameters.

Use a readable named variable (or keyword) to refer to the first argument or ditch the feature entirely.

Related issues:

Related to Ruby trunk - Feature #4475: default variable name for parameter	**Closed**

History

#1 - 03/22/2019 01:49 PM - mame (Yusuke Endoh)

- Related to Feature #4475: default variable name for parameter added

#2 - 03/22/2019 01:54 PM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

Yes, I admit { @1[@2] = "Go Fish: #{@2}" } is cryptic. But {@1 * @2} is not. So use numbered parameters with care (just like other features in Ruby). The **possibility** to make code cryptic itself should not be the reason to withdraw a feature.

The problem is introducing it or this could break existing code. That **is** the problem. Numbered parameters are a compromise to simplify block expression without breaking compatibility.

FYI, I am not fully satisfied with the beauty of the code with numbered parameters, so I call it a compromise.

Matz.

#3 - 03/22/2019 02:12 PM - shevegen (Robert A. Heiler)

I personally like the change. I think it is good for quick debugging.

Stefan picked only one example, though:

```
Hash.new { @1[@2] = "Go Fish: #{@2}" }
```

But who is to say that ruby users have to (only) write code like the above? Also, why would this have to be assumed that this "must" be in production code? I don't quite understand Stefan here.

Consider the following code for example:

```
big_array_data_structure_with_lots_of_entries.each {|name_of_person, age_of_person, hash_where_this_person_may
_live, hash_storing_the_favourite_cats_of_this_person|
  pp @3

  if @4.has_key? 'tom'
    pp 'yup he has a cat named tom'
  end

}
```

None of the above would really make it into production code, but I do happen to have code that is vaguely similar to the above. Actually I put all the lines with "pp" and the rest on the very left hand side, so that when I finish writing the code, I can easily remove all the pp statements. The alternative to the above would be to have to use the longer parameters names, and the code does not magically become "better" just because I use the longer name. And I would remove the 1 (1 1) @2 etc... lateron anyway, so I am not sure why Stefan thinks that everyone will end up including this in their code? Is there some mysterious cat sitting on the shoulder of people forcing them to let that code remain? I can not remove the pp @3 at a later time anymore?

I am not stating that this is very pretty, yes, I agree here, it won't win beauty contests. But we can say the same about $1 $2 $3 etc... but also other features/syntax changes, so I don't think this is a very good argument against the feature in general.

> So please reconsider numbered parameters.

Please don't. :D (I am actually serious though, I like that change.)

In general I think you simply have to find a style that is comfortable for you in ruby, and then use what you prefer here, and reject what you dislike. I do that in many other ways too, e. g. not needing @@foo variables, but others like/use them. But anyway, matz said it is up for feedback, so I guess you all should use that opportunity and comment, if you would like to, EITHER way, and ideally also state WHY you like (or not like) the change.

PS: Before I forget, though, Stefan referred to the 8 years old suggestion. This is understandable, but I would like to remind everyone that it is not solely about that suggestion alone - there have been other comments made over the years. For example, I first thought that the suggestion would be to have mandatory names given to the parameters, like:

```
object.each {|a, b, c|
```

And only then to refer to it (e. g. via 1 (1 1) @2 etc..., but we can also omit the names. I have no strong preference either way, since I think both variants have advantages; but I think personally I'll prefer the variant where I give names, but STILL would like to be able to easily access that with a number.

My main argument is not about beauty, though - it is of "practical use". When I work with a data structure, it can be much easier for me to remember the position, rather than the name I gave. Perhaps that may be an argument that others also have, in the way they write ruby code. As matz once said (and it is true in general), people are different. And ruby has always been multi-paradigm (and "more than one way"), even if OOP is in my opinion ruby's core approach.

#4 - 03/22/2019 02:18 PM - mame (Yusuke Endoh)

sos4nt (Stefan Schüßler), I agree with you. Naming a parameter is good, I think.

By the way, The Ruby Style Guide that you mentioned says:

```
# bad
names.map { |name| name.upcase }

# good
names.map(&:upcase)
```

I dislike this style too, because it omits a name. Worse, the &: style is incomplete: we cannot pass an argument ({|name| name.upcase(:ascii) }), and we cannot use it for function call ({|name| foobar(name) }). There are some proposals that try to solve the incompleteness, e.g., #12115, #15301, #15302, #15483, but all of them look to me truly cryptic, in their appearance and/or semantics.

I understand that names.map { @1.upcase } is still a bit cryptic, but it is definitely much easier than all of the proposals above. And, more importantly, we can naturally extend it to { @1.upcase(:ascii) } and { foobar(@1) }.

This is one of the reasons why numbered parameters were introduced.

#5 - 03/22/2019 03:38 PM - alanwu (Alan Wu)

I don't like this feature because it adds a special case to the meaning of @.
Before this, I can look at @ and understand that it's an instance variable, now there is extra mental overhead.

Every new feature added to the language has an educational cost. I imagine lots of people will spend time searching about what @1 is, after seeing it for the first time and being confused. The feature isn't even released and we have confusion already #15708. For new users, this is yet another thing they have to know about the core language.

I would much prefer a smaller ruleset with less exceptions than saving a few keystrokes. The core language is already pretty hard to master because of the many special cases added over the years.

#6 - 03/22/2019 04:06 PM - sawa (Tsuyoshi Sawada)

alanwu (Alan Wu) wrote:

> The feature isn't even release[d] and we have confusion already #15708.

That is because it has not been documented yet. After the comments from the developers, now I understand it. There is no problem understanding it. Although I am not sure if I like it.

#7 - 03/22/2019 07:22 PM - sos4nt (Stefan Schüßler)

matz (Yukihiro Matsumoto) wrote:

> I am not fully satisfied with the beauty of the code with numbered parameters, so I call it a compromise.

shevegen (Robert A. Heiler) wrote:

> I am not stating that this is very pretty, yes, I agree here, it won't win beauty contests.

Sorry, but it sounds like the new feature is an ugly compromise ...

I learned Ruby – and stuck with it – because of its beauty and its simplicity, two of Ruby's core values. I wouldn't mind having a new, elegant way to express myself. But the new syntax isn't elegant. Referring to an invisible argument via its index, prefixed by a sigil from a completely different context (instance variables) is hacky

shevegen (Robert A. Heiler), you say that I don't have to use the new feature, but that's like saying I could keep using :a => 1 instead of a: 1. I don't write code just for myself in isolation. I have to deal with other people's code and projects as well. Not adopting or ignoring new features is unrealistic.

You also mention that it's "good for quick debugging". But do we really need a language change for easier debugging? I don't think so.

Besides, I don't think this is a negligibly syntax quirk no one is going to use. People on Stack Overflow already tend to prefer terser code (the so-called "one-liner") to a more readable method with 3 lines. This change will have an impact.

Maybe someone could provide a real-world example where the new syntax really shines? I can't think of one.

#8 - 03/24/2019 04:19 PM - pascalbetz (Pascal Betz)

matz (Yukihiro Matsumoto) wrote:

> FYI, I am not fully satisfied with the beauty of the code with numbered parameters, so I call it a compromise.

A compromise does not sound like something that should be added to the language.

{@1 * @2} could easily be written as {|a, b| a * b} some chars more, yes. But IMHO more readable. I don't think that the language should be extended for something that does not have any benefit besides saving some typing.

#9 - 03/25/2019 01:33 AM - shyouhei (Shyouhei Urabe)

pascalbetz (Pascal Betz) wrote:

> {@1 * @2} could easily be written as {|a, b| a * b} some chars more, yes. But IMHO more readable. I don't think that the language should be extended for something that does not have any benefit besides saving some typing.

While I'm not totally against your opinion, I would like you to understand that this feature was not made out of thin air; it was feature requested. People think this is a neat feature. If you are against the feature itself not against the syntax, please show us it harms more than it benefits.

#10 - 03/25/2019 04:47 AM - sawa (Tsuyoshi Sawada)

I think there has been strong desire over the years for a shorthand feature like this, and it is unrealistic to ask for the feature to be withdrawn completely.

However, it is true that there is controversy regarding the use of the sigil @. Couldn't it be something else? Those of you who are against should provide an alternative syntax, especially an alternative sigil. I remember someone proposing the backslash \. Will that have problems? Are there other alternatives?

#11 - 03/25/2019 05:53 AM - maedi (Maedi Prichard)

I think the usefulness of this feature is when there's only one parameter. When there's multiple parameters it becomes more confusing to read @1, @2, @3 and less confusing to just name the variables.

I think limiting the "magic variable" to the first param and giving it a nice name/symbol is a good limit. It clearly defines why we have this magic variable in the first place; to make it easier for situations when we didn't have many parameters anyway and would like to match a simple situation with simple syntax.

Why not a single numberless @? Though I agree it's not the best symbol as it screams instance variable. We should search for a single keyword/symbol. I kind of like:

```
posts.each { ^.author = 'Santa Clause' }
```

It's almost as if the caret is pointing up to refer to the current item that it is within.

For another code example I saw in #4475 it has a nice visual flow to it:

```
Account.all.each { my_account << ^.money }
```

We're referencing the money property of the current item then pushing it into the array.

#12 - 03/25/2019 06:22 AM - jeremyevans0 (Jeremy Evans)

As the person who suggested the @1 syntax, I guess I should give my opinion.

First, any approach that did not use a sigil (e.g. this or it) would probably break backwards compatibility, and that is not considered acceptable. So

the only acceptable approach would be something that uses a sigil. In order to support more than 1 argument, the sigil would need to be numbered. As this is being added to reduce verbosity, you probably want the sigil followed by the number representing the argument.

There are a limited number of sigils that are possible options:

@: Chosen syntax for block argument access, and @ already associated with scoped (instance) variable access
$: Also associated with variable access, but unscoped (global), and $1 and such already used for regular expression captures ($#1 and such would be possible, but longer)
\: Associated with escaping in strings, not variable access
%: Associated with string/array creation (e.g. %[a], %w[a b]) and modulus/formatting, not variable access
:: Associated with symbol creation, ternary operator, and method creation operator (.:), not variable access

I think most other sigils would not be backwards compatible as the syntax would be already valid (~) or would probably cause parsing issues if introduced (/). Any sigil that could be a unary or binary operator could break backwards compatibility if used.

As I said when proposing the syntax, I'm not convinced an implicit block argument syntax is a good idea. However, if we are going to have an implicit block argument syntax, I think @1, @2, etc. is probably the best choice.

#13 - 03/25/2019 07:47 AM - sawa (Tsuyoshi Sawada)

Here are some alternative sigils.

Provided that we obsolete the character literal notation ?a # => "a", which is rarely used, then we may use the ? sigil like:

```
[1, 2, 3].each { puts ?1}
```

Provided that we obsolete the command evaluation literal `echo "foo"`, which I think I heard that Matz was thinking to do so in the future, and that we can wait till then, then, whenever that is done, we can use the ` sigil like:

```
[1, 2, 3].each { puts `1}
```

#14 - 03/25/2019 12:12 PM - pascalbetz (Pascal Betz)

shyouhei (Shyouhei Urabe) wrote:

> pascalbetz (Pascal Betz) wrote:
>
> > {@1 * @2} could easily be written as {|a, b| a * b} some chars more, yes. But IMHO more readable. I don't think that the language should be extended for something that does not have any benefit besides saving some typing.
>
> While I'm not totally against your opinion, I would like you to understand that this feature was not made out of thin air; it was feature requested. People think this is a neat feature. If you are against the feature itself not against the syntax, please show us it harms more than it benefits.

I know that this feature was not made out of thin air. But out of a 8 year old request that does not show a real benefit (other than "it's a bit shorter"). The harm it does is, IMHO, that

- it uses the @ syntax and applies it to a variable that has a different scope and meaning
- it proposes a syntax where variables without clear name are a good thing

So I'm actually against the feature as well as the syntax. But my concerns are much more for the syntax.

I asked around with my ruby-friends and none of them is missing this feature or thinks that it should be added. So my concerns are also not out of thin air (but is also not representative for the whole ruby community).

Thanks Jeremy Evans for explaining how the syntax came to be. Aren't there any other symbols we can use? paragraph, backtick, caret, (double-)dagger, ...?

I just want to add again: nobody seems to be entirely happy. I think this is not good enough to add a feature to the language. After all we will have to live with it for decades to come□□

#15 - 03/25/2019 12:37 PM - matz (Yukihiro Matsumoto)

sawa (Tsuyoshi Sawada) Obsoleting and recycling ? or ` sigils sounded like a nice idea at first, but incompatibility caused by the obsoletion remains years (or even decades). It seems nearly impossible.

Matz.

#16 - 03/25/2019 04:48 PM - bozhidar (Bozhidar Batsov)

> While I'm not totally against your opinion, I would like you to understand that this feature was not made out of thin air; it was feature requested. People think this is a neat feature. If you are against the feature itself not against the syntax, please show us it harms more than it benefits.

Every language change obviously comes at a cost - it affects every tool that does some AST-based analysis, it adds cognitive overload, it opens up potential for misuse of the features, etc, so language changes are definitely not for free. I realize that the feature was requested by someone, but I don't see any strong argument for adding it, I see that even Matz seems to be on the fence about it, and clearly some people (like me) are frustrated by the exact syntax. I don't think that's how language design is supposed to happen (there should be more research up front) and that's definitely not a recipe for (universal) programmer happiness. The Unix people had it right - often "less is more". I'd rather have fewer, but very polished features, as opposed to many somewhat useful, but half-baked ones.

> I just want to add again: nobody seems to be entirely happy. I think this is not good enough to add a feature to the language. After all we will have to live with it for decades to come

Spot on. Let's either rollback this, while there is still time, or come up with a syntax that more people would endorse.

Btw, isn't it possible to just add $it and make it a hash if there's more than one param to a block? E.g. $it[1] * $it[2]. (or something along those lines)

#17 - 03/25/2019 05:43 PM - maedi (Maedi Prichard)

Why not a single dot? (.) It mirrors the UNIX concept of "current directory":

```
[1, 2, 3].each { puts . }
```

With a method/property:

```
posts.each { ..author = 'Santa Clause' }
```

Keeps it simple. Makes it easier to understand as people already use this on the command line every day. The concept of "current directory" is eerily similar to the concept of "current item".

#18 - 03/25/2019 06:19 PM - jeremyevans0 (Jeremy Evans)

pascalbetz (Pascal Betz) wrote:

> Aren't there any other symbols we can use? paragraph, backtick, caret, (double-)dagger, ...?

We should not add any sigils that are non-ASCII. That removes paragraph, dagger, double-dagger.

^ (caret) would not be backwards compatible as it is already a binary operator:

```
proc{a.foo ^1} # already means `a.foo.^(1)`
```

backtick is also not backwards compatible:

```
foo = Object.new
def foo.`(x) x end
proc{foo. `1}
```

bozhidar (Bozhidar Batsov) wrote:

> Btw, isn't it possible to just add $it and make it a hash if there's more than one param to a block? E.g. $it[1] * $it[2]. (or something along those lines)

$it can already be defined, so this is not backwards compatible.

In terms of using a hash/array instead of separate arguments, we should try to avoid introducing shortcuts that make code slower by forcing object allocations.

maedi (Maedi Prichard) wrote:

> Why not a single dot? (.) It mirrors the UNIX concept of "current directory":
>
> ```
> [1, 2, 3].each { puts . }
> ```
>
> With a method/property:
>
> ```
> posts.each { ..author = 'Santa Clause' }
> ```

Thing about parsing the following code:

```
posts.each { foo.to_i ..num }
```

Currently, this is a range, and if . was used as a reference to the current object, it would become ambigious as then it could mean foo.to_i(..num).

As stated earlier, we need to have a syntax that can handle multiple block arguments, and .1 causes problems with ranges (e.g. is foo...1 the same as (foo)..(.1) or (foo)...(1)?) and would not be backwards compatible.

#19 - 03/25/2019 06:59 PM - maedi (Maedi Prichard)

jeremyevans0, is it possible to have method/property calls that when without an object, reference the "current item" of the block they are in?

```
posts.each { .author = 'Santa Clause' }
```

This stops the range situation (..). Then still have the single . for non method/property situations.

I'm still of the "single magic variable" camp, so if it's going the way of supporting multiple params I'll leave it here.

#20 - 03/25/2019 07:18 PM - jeremyevans0 (Jeremy Evans)

maedi (Maedi Prichard) wrote:

> jeremyevans0, is it possible to have method/property calls that when without an object, reference the "current item" of the block they are in?
>
> ```
> posts.each { .author = 'Santa Clause' }
> ```

Think about handling:

```
posts.each { puts .author }
```

Currently, that means call the author method on the object returned by calling the puts method with 0 arguments. If we added the syntax you are proposing, it would be ambiguous as it could mean calling the puts method with a single argument, which is the object returned by calling the author method on the block argument.

> This stops the range situation (..). Then still have the single . for non method/property situations.

Unfortunately, it does not address the range situation completely:

```
posts.each { 1...author }
```

This could then mean (1)..(.author) or (1)...(author)

Or even:

```
posts.each { 1... }
```

This could then mean (1)..(.) or (1)... (infinite range).

Let's say you want to call a method on the block argument and pass the result as a block to another method:

```
posts.each { foo &.a }
```

Unfortunately, this already is used for the lonely operator, so this would break backwards compatibility.

#21 - 03/25/2019 07:38 PM - maedi (Maedi Prichard)

Great explanation, thanks :)

If it will be multiple params @1, @2, etc. Can we also have @ aliasing @1? I think it's going to be a common use case to just have the one param.

#22 - 03/26/2019 07:57 AM - bozhidar (Bozhidar Batsov)

> $it can already be defined, so this is not backwards compatible.

It can, but that's extremely unlikely, so I think the impact to backwards compatibility would be negligible. I've almost never seen anyone define globals themselves. I don't think it's wise to eliminate some possibilities simply because they *might* cause minimal problems.

Btw, using %1, %2, %3 is fine, right? How do people feel about that one in general. I'm kind of used to it, because that's how Clojure does it, but I'm curious if it feels very weird in general.

> If it will be multiple params 1 (1 1), @2, etc. Can we also have @ aliasing 1 (1 1)? I think it's going to be a common use case to just have the one param.

I think that would look extra weird, but in general it's a good idea (especially if the special var had some meaningful name).

#23 - 03/26/2019 08:16 AM - decuplet (Nikita Shilnikov)

bozhidar (Bozhidar Batsov) wrote:

> Btw, using %1, %2, %3 is fine, right? How do people feel about that one in general. I'm kind of used to it, because that's how Clojure does it, but I'm curious if it feels very weird in general.

I guess it's not. Bits like n%2 would be a problem most likely.

> If it will be multiple params 1 (1 1), @2, etc. Can we also have @ aliasing 1 (1 1)? I think it's going to be a common use case to just have the one param.

> I think that would look extra weird, but in general it's a good idea (especially if the special var had some meaningful name).

And by induction @@ is an alias for @2 and so on. Sorry, couldn't keep it :)

#24 - 03/26/2019 08:52 AM - duerst (Martin Dürst)

Answering to 4 different posts in one go, sorry.

bozhidar (Bozhidar Batsov) wrote:

> Btw, using %1, %2, %3 is fine, right? How do people feel about that one in general. I'm kind of used to it, because that's how Clojure does it, but I'm curious if it feels very weird in general.

I definitely prefer @1 over %1 for Ruby, because for me it 'rhymes' with @instance_var and @@class_var, as jeremyevans0 has explained.

pascalbetz (Pascal Betz) wrote:

> I know that this feature was not made out of thin air. But out of a 8 year old request that does not show a real benefit (other than "it's a bit shorter").

Well, the original request in definitely 8 years old. But there was quite some discussion recently. And if I remember correctly, there were also other proposals in the same direction. I think this comes from the fact that even more programmers than before get familiar with functional programming.

bozhidar (Bozhidar Batsov) wrote:

> Every language change obviously comes at a cost - it affects every tool that does some AST-based analysis, it adds cognitive overload, it opens up potential for misuse of the features, etc, so language changes are definitely not for free.

Yes indeed. But let's for a moment look at cognitive overload. For a much more familiar example, let's look at
a simple assignment expression: a -= b, which we all understand is a shortcut for a = a - b. This is very old and well-known from C.

There are actually companies that forbid this because (probably for their average programmer) it's too complicated. But most programmers use it, even though one could say it adds 'cognitive overload'. For people coming from a = a - b, it indeed adds cognitive overload because when they see a -= b, they first translate it to a = a - b. But after a while, that cognitive overload goes away, because a -= b is read simply as "substract b from a", whereas now the cognitive overhead is with a = a - b, which is read "assign to a the difference between a and b", which is longer and where the fact that a appears twice has a special significance.

Now after that little example from the past, let's look again at something like { |n| n-1 }. We see that n appears twice, but we couldn't care less about the actual choice of variable name (except a faint scent that n may be an integer, or a number, or so). Once we get used to it, { @1-1 } may look much more direct. In both cases, it's a block (anonymous function) that subtracts 1. In Haskell, a fundamentally functional language, this special case is even shorter, just (-1), and is called an operator section. (1-) is the reverse, { 1-@1 } or { |n| 1-n } in Ruby. (Other cases need explicit argument names in Haskell.)

There is an additional cognitive issue involved: There are studies that say that whatever the language, programmers are able to grasp about the same amount of code (e.g. number of lines) at a time. That means that whenever code can be truly shortened (not just squeezed by eliminating spaces and newlines,...) there is a gain
in the actual functionality that can be grasped. That's where the productivity gains e.g. from C to Ruby,
and from plain Ruby to e.g. Rails,..., mostly come from.

maedi (Maedi Prichard) wrote:

> I think the usefulness of this feature is when there's only one parameter. When there's multiple parameters it becomes more confusing to read @1, @2, @3 and less confusing to just name the variables.

I agree that most of the usefulness of this feature is with @1. But something like
array1.zip(array2).map { @1 - @2 } (or better yet, array1.zip_with(array2) { @1 - @2 }, once zip_with makes it into Ruby, see issue #4539) can also

be quite useful.

But I can imagine having Rubocop rules that limit the use of @1 and friends to some contexts such as one-liners. There are many Ruby features that are very helpful when used in moderation, but are counterproductive when overused. This is no exception. But we will have to learn where the boundary between usefulness and overuse is.

#25 - 03/26/2019 12:20 PM - maedi (Maedi Prichard)

I believe a bare @ should still be implemented as it reconciles the original need of the ticket of “hard to read code” and “a readable named variable to refer to the first argument“, as well as matz’s “not fully satisfied with the beauty of the code”. It lets people write both the simple version and/or the verbose but more flexible version.

Maybe because it's already in a release, but the more I stare at the @ symbol the more I like it. I think it makes sense: @ with letters are instance variables, @@ with letters are class variables and @ with or without numbers are instances. The @ symbol itself is an "instance accessor" (@ = instance and @variable = instance variable). I'm just trying to build some semantics around this and see if it sticks.

#26 - 03/26/2019 02:20 PM - bozhidar (Bozhidar Batsov)

> I definitely prefer 1 (1 1) over %1 for Ruby, because for me it 'rhymes' with @instance_var and @@class_var, as jeremyevans0 has explained.

That's why I dislike it so much. Block vars would ideally not look like class/instance vars. Apart from my preferences this will also impact some editors, which might have had custom font locking for something starting with @ and now they have to update it to differentiate the two categories.

I get that there are no great options right now, but that's also what bothers me so much about this - it was clear that the feature is somewhat problematic, clearly there's also backlash against it, and we are still moving forward with it on some really vague merits. If that's what we call "optimizing for happiness" these days Ruby has lost its way... It's obviously up to Matz to decide how he wants to do things, but I think that controversial features might be rollbacked until they are researched better and ideally solved better. With the strong focus on backwards compatibility every time something gets accepted into the language we're basically stuck with it forever, so I think it makes sense to think long and hard about any language change.
I also feel there's a communication problem - as people outside the core team generally have no way to know what's being worked on it's hard for them to provide any feedback on shortlisted issues until they see some announcement that a few feature was added.

#27 - 03/26/2019 03:24 PM - jeremyevans0 (Jeremy Evans)

bozhidar (Bozhidar Batsov) wrote:

> > I definitely prefer 1 (1 1) over %1 for Ruby, because for me it 'rhymes' with @instance_var and @@class_var, as jeremyevans0 has explained.
>
> That's why I dislike it so much. Block vars would ideally not look like class/instance vars. Apart from my preferences this will also impact some editors, which might have had custom font locking for something starting with @ and now they have to update it to differentiate the two categories.

Is there a particular reason why block argument access shouldn't look like variable access (@), but instead should look like string/array creation or modulus (%)? Other than looking like Clojure? As I haven't done much programming in Clojure, using %1 for block argument access and such looks quite weird to me.

In any case, as decuplet pointed out, I was wrong and % is not possible as a sigil due to backwards compatibility, as it is used for the modulus operator:

```
proc{foo %1}
```

This currently means call the foo method, and call the % method on the result with the argument 1. If %1 where added, it would become ambigious.

> It's obviously up to Matz to decide how he wants to do things, but I think that controversial features might be rollbacked until they are researched better and ideally solved better.

I disagree. To only accept uncontroversial features is the death knell for progress. Design by committee is the last thing Ruby needs.

Now, if introducing syntax causes regressions, sure, back it out until the regressions are fixed. However, that is not the case here.

> With the strong focus on backwards compatibility every time something gets accepted into the language we're basically stuck with it forever, so I think it makes sense to think long and hard about any language change.

I agree. However, to imply that matz has not already done this before accepting the feature is a tad insulting (I think). matz took over a month from when the @1 syntax was proposed before accepting it.

> I also feel there's a communication problem - as people outside the core team generally have no way to know what's being worked on it's hard

> for them to provide any feedback on shortlisted issues until they see some announcement that a few feature was added.

The proposal was posted as a note to an existing issue. Provided you are notified about all notes posted to issues (and if you are interested in ruby-core development, you probably should be), you should have had over a month to provide feedback for matz to consider before he accepted the feature.

#28 - 03/26/2019 05:12 PM - noniq (Stefan Daschek)

sawa (Tsuyoshi Sawada) wrote:

> I remember someone proposing the backslash \. Will that have problems? Are there other alternatives?

What about \1, \2, etc.?

```
numbers.zip(other_numbers).map{ \1 * \2 }
```

Having something that looks similar to regexp backreferences would even make sense on a conceptual level, imho.

#29 - 03/26/2019 08:35 PM - shevegen (Robert A. Heiler)

Syntax-wise I think 1 (1 1) @2 is better than %1 %2 (see what jeremy wrote, even the backwards-compatibility issue aside) or \1 \2. But this probably comes down to aesthetics primarily. We only have a limited subset available in regards to syntax.

Edit: Oops, redmine swallowed the syntax, it was:

```
@1 @2 # is better, in my opinion, than:
%1 %2
# or
\1 \2
```

% in particular reminds me a lot of:

```
%w( big long array here )
```

I have a slightly smaller gripe with \1 or \2; I just don't think they are very elegant here. So from these three possibilities, I'd prefer **@ 1 @ 2** and so forth. (Redmine swallowed it again ...so I spaced it out a bit this time.)

#30 - 03/26/2019 10:15 PM - joanbm (Joan Blackmoore)

Gray Kemmey summed it up perfectly .

I guess there is really hard to find some adequate counter-argument, defending idea behind #4475 proposal and concluded implementation.

I would not like sound too harsh, but additions to the core language in recent years are questionable, at least. Object#yield_itself and its #then alias, "safe" navigation operator &. vs object model consistency, excessive syntax for Proc instance invocation .(), and numbered block parameters now have come to mind.

Changes in libraries to stay relevant and current are easily defensible and intelligible, but touching language itself should be handled with greatest care and make changes/additions when there is really *really* good reason. Some hasty decisions and mistakes may simply become irreversible and can only distract senior developers in the meantime …

#31 - 03/27/2019 12:32 AM - phluid61 (Matthew Kerwin)

joanbm (Joan Blackmoore) wrote:

> I would not like sound too harsh, but additions to the core language in recent years are questionable, at least. Object#yield_itself and its #then alias, "safe" navigation operator &. vs object model consistency, excessive syntax for Proc instance invocation .(), and numbered block parameters now have come to mind.

You forgot to complain about sym: value hash syntax, -> lambdas, foo(&:bar) proc-blocks, etc. etc.

Personally I find @1 the least intrusive bit of syntactic magic added to Ruby in the past, like, ten years.

People really do care about it, though.

#32 - 03/27/2019 09:18 AM - ahvetm (Erik Madsen)

I signed up just to comment on this thread, because I also feel that this change is rather weird, and I would like to propose an alternative, that I don't

think has been discussed.

I understand the need for the feature, and it's a welcome addition, but the syntax feels iffy to me. Two concrete issues:

- The argument numbering dilutes the convention of counting from 0, e.g. when referencing an element in an array.
- The '@n' syntax complicates the idea of what '@' identifies in Ruby, making the language potentially harder to learn.

A better way in my mind would be to provide access to an array representation of the arguments (borrowing an example from above) like so:

```
numbers.zip(other_numbers).map { them[0] * them[1] }
```

I chose 'them' here as a reference to 'it', but that's just to illustrate the concept, I haven't thought about good names for the array.

Such a feature would also be useful to have in methods in general, for instance it would be nice to have a short-hand for just passing all input from the current method to the next, including keyword args.

```
def cows(country:, size:)
  puts "talking about cows in #{country}"
  abstract_cows(**them, colour: 'blue')
end

def abstract_cows(country:, size:, colour:); end
```

I know this is beyond the current discussion, but I just wanted to show how a shorthand for accessing all args in general could fit in nicely with the whole language.

#33 - 03/27/2019 09:51 AM - bozhidar (Bozhidar Batsov)

> I disagree. To only accept uncontroversial features is the death knell for progress. Design by committee is the last thing Ruby needs.

I'm not a fan of design by committee as well. I was referring to the fact he seemed unhappy with the proposal, even though he accepted it. I guess a lot can be lost when you're just reading about something, but it definitely sounded weird to me that he opted to go forward with something he has reservations about. And because there's more backlash than usual for that particular change it stands to reason that there's room for improvement. :-)

> I agree. However, to imply that matz has not already done this before accepting the feature is a tad insulting (I think). matz took over a month from when the 1 (1 1) syntax was proposed before accepting it.

Fair point. I obviously don't know how exactly the process went. I just know what I can read in the issue tracker.

> The proposal was posted as a note to an existing issue. Provided you are notified about all notes posted to issues (and if you are interested in ruby-core development, you probably should be), you should have had over a month to provide feedback for matz to consider before he accepted the feature.

Another fair point. Sadly I haven't had as much time as I wanted to monitor ruby-core development. It would have been nice if there was some low-volume mailing list about language changes that are pre-approved or whatever. Maybe some idea for the future?

Going back to that particular feature - I still think that using some special var is the best course of action, as I remain convinced that something like $it is unlikely to cause problems in real world apps. Even the idea for $# is more appealing to me, as that allow us to use just $# for a single block param. I think the single param usage is probably the biggest selling point of that change and it's definitely weird that now we have to use a number for this. A shorthand to access all params as array would be handy as well in some instances.

#34 - 03/27/2019 10:04 AM - duerst (Martin Dürst)

ahvetm (Erik Madsen) wrote:

> I signed up just to comment on this thread, because I also feel that this change is rather weird, and I would like to propose an alternative, that I don't think has been discussed.
>
> I understand the need for the feature, and it's a welcome addition, but the syntax feels iffy to me. Two concrete issues:
>
> - The argument numbering dilutes the convention of counting from 0, e.g. when referencing an element in an array.

In general I'd agree, but there are quite a few cases (e.g. C argv) where number 0 is special, and arguments start from 1, so for this case, I don't think it's a problem.

> - The '@n' syntax complicates the idea of what '@' identifies in Ruby, making the language potentially harder to learn.

Well, it's already @ and @@, so it's not getting much more difficult. I'd actually prefer this to some completely different character; at least that way, it

helps to understand it's some kind of variable.

> A better way in my mind would be to provide access to an array representation of the arguments (borrowing an example from above) like so:
>
> ```
> numbers.zip(other_numbers).map { them[0] * them[1] }
> ```

I'm sorry, but to me, it would smell too much like Perl or C varargs.

#35 - 03/27/2019 10:26 AM - ahvetm (Erik Madsen)

duerst (Martin Dürst) wrote:

> - The '@n' syntax complicates the idea of what '@' identifies in Ruby, making the language potentially harder to learn.
>
> Well, it's already @ and @@, so it's not getting much more difficult. I'd actually prefer this to some completely different character; at least that way, it helps to understand it's some kind of variable.

Yes, but in my mind it carries misleading hints about the scope of the variable.

> > A better way in my mind would be to provide access to an array representation of the arguments (borrowing an example from above) like so:
> >
> > ```
> > numbers.zip(other_numbers).map { them[0] * them[1] }
> > ```
> >
> > I'm sorry, but to me, it would smell too much like Perl or C varargs.

I'm afraid I'm not sufficiently familiar with those languages to understand the specific negative implications in this context

#36 - 03/27/2019 09:27 PM - Eregon (Benoit Daloze)

duerst (Martin Dürst) wrote:

> > ```
> > numbers.zip(other_numbers).map { them[0] * them[1] }
> > ```
> >
> > I'm sorry, but to me, it would smell too much like Perl or C varargs.

Actually, I would think @1, @2 start to look much like Perl, i.e. cryptic sigils, that someone not already familiar with them would wonder why would people want to use something so unclear, instead of plain old good local/block variables.

They are somewhat similar to $1, $2 which I think are also fairly cryptic and not really needed (but we need to keep them for compatibility).
Those additionally make the language implementation significantly more complex ($~ is a thread-local frame-local totally magic variable, and the semantics to this day are still unclear, #12689).

> array1.zip(array2).map { @1 - @2 } (or better yet, array1.zip_with(array2) { @1 - @2 })

Both of these already feel hard to read for me (and I'm implementing Ruby, so I would think I'm familiar with the Ruby syntax).
I have to read the entire block body (which could be long), just to find out how many block arguments it takes, and even then it's only a guess because maybe some block arguments are just not used.
array1.zip(array2).map { |a,b| a - b } or array1.zip(array2).map { |(a,b)| a - b } is so much clearer, isn't it?

What if we had a each_slice which would find how many elements to yield based on the block arity, like ary.each_slice { |a,b,c| ... }? (FWIW, there already are methods checking the block arity)
Then this new syntax would be pretty confusing: ary.each_slice { do_one_thing_with(@2).and_sth_else(@3, @1) }.

I started pretty neutral about this issue, but ends up quite against it given how it seems to impact readability negatively.

Maybe this is a good topic to discuss at the RubyKaigi meeting? Or to make a poll during one of the talks?

#37 - 03/27/2019 09:40 PM - Eregon (Benoit Daloze)

matz (Yukihiro Matsumoto) wrote:

> The **possibility** to make code cryptic itself should not be the reason to withdraw a feature.

Agreed. However, so far I think almost every example I saw with more than one @-parameter feels cryptic.
Maybe we should just have @ and nothing else, like:

```
ary.map { @ * 3 }
```

That's nice, I would say even nicer than ary.map { @1 * 3 }.

Why should variables be 1-indexed and not 0-indexed?
Why should we have indexes for variables like the bytecode/machine has when we are humans and the Ruby language emphasizes the human, not the machine?
These questions do not need to be with just @.

I think we should prevent things like h.each_pair { foo[@1] = @2 }, which e.g., makes it harder to know which one is the key and which one is the value.

#38 - 03/28/2019 12:14 AM - duerst (Martin Dürst)

Eregon (Benoit Daloze) wrote:

> duerst (Martin Dürst) wrote:
>
> > array1.zip(array2).map { @1 - @2 } (or better yet, array1.zip_with(array2) { @1 - @2 })
>
> Both of these already feel hard to read for me (and I'm implementing Ruby, so I would think I'm familiar with the Ruby syntax).
> I have to read the entire block body (which could be long), just to find out how many block arguments it takes, and even then it's only a guess because maybe some block arguments are just not used.
> array1.zip(array2).map { |a,b| a - b } or array1.zip(array2).map { |(a,b)| a - b } is so much clearer, isn't it?

What's easy or hard to read is definitely to quite some extent subjective, but just a few more points:

- You have seen the later many many times over already, but not the former, because it's new syntax. It takes some time to get used to new syntax. Of course, that's an argument against introducing new syntax in general, but we should also look at how things might feel once we get used to the new syntax.
- When I see zip(_with), I know there will be two arguments. When I see @1-@2 (after getting used to), I know it's subtraction.
- Every one of us, and we as a collective of Ruby programmers, will have to find out where the boundaries of using these variables are. Everybody agrees that using them in long blocks is a bad idea, and using too many of them is also a bad idea. Most people agree that for a single variable in a very short block, it can be useful. But we have very little practice, at least in Ruby, about where (approximately) the boundaries are. Maybe people used to other programming languages that have such a feature (e.g. Closure) can give some hints?

> What if we had a each_slice which would find how many elements to yield based on the block arity, like ary.each_slice { |a,b,c| ... }? (FWIW, there already are methods checking the block arity)
> Then this new syntax would be pretty confusing: ary.each_slice { do_one_thing_with(@2).and_sth_else(@3, @1) }.

I agree for this example. each_slice doesn't give any hint as to arity (and that's most probably why it doesn't work that way in Ruby anyway), the block is longer, the number of parameters is higher, and they are not in order, all of which affect the overall readability balance.

> I started pretty neutral about this issue, but ends up quite against it given how it seems to impact readability negatively.
>
> Maybe this is a good topic to discuss at the RubyKaigi meeting? Or to make a poll during one of the talks?

Discussion is definitely a good idea. Polling may be worse than design by committee (the IETF occasionally uses humming as an alternative). The result can change very much depending on who gave what talk before the question(s), what question(s) are asked, how the question(s) are asked, who asks the question(s), and who is in the audience. But of course if Matz wants to ask his audience, that's his choice.

#39 - 03/28/2019 12:25 PM - mame (Yusuke Endoh)

The reason why this feature was introduced, is that people want a shorthand for ary.map {|x| x.to_i(16) } and ary.map {|x| x.to_i.chr }.

Vote:

1. Ad-hocly extend ary.map(&:to_i): ary.map(&:to_i.(16)) and ary.map(&(:to_i >> :chr)) (#12115, #15483, etc.)
2. Introduce '@1': ary.map { @1.to_i(16) } and ary.map { @1.to_i.chr }. It can naturally support ary.map { foo(@1) }, too.
3. Nothing. People must write |x| explicitly.

I vote for 3. But if I had to choose 1 or 2, I would go for 2.

#40 - 03/29/2019 08:21 AM - vo.x (Vit Ondruch)

mame (Yusuke Endoh) wrote:

> Vote:

I vote for 3 (and hashrockets ;))

#41 - 03/29/2019 09:33 AM - shevegen (Robert A. Heiler)

> I vote for 3 (and hashrockets ;))

I'd add a fourth variant - that is to use both explicit names and
1 (1 1) @2 @3. :)

But to keep my comment somewhat short - since hashrockets was mentioned, even with a twinkling smiley, I initially did not like hashrockets, and I think => is also visually nicer, I slightly changed my mind over time and am fine with the foo: :bar variant. I use it a lot, too.

THe primry reason as to why I like foo: :bar is indeed that you can write less code. Which is quite nice if you really have a lengthy Hash.

For similar reasons I like:

```
%w( foo bar bla )
```

I find it much easier to write than:

```
['foo','bar','bla']
```

Even if the latter variant may be less confusing to a newcomer. But the first variant is so much nicer to type - I can omit all these various ','. Similar for 1 (1 1) @2 at the least for debugging. I don't think I would use 1 (1 1) @2 in production code per se, but am I the only one who sometimes forgets what is stored in a hash or a similar object that uses .each?

There has been another suggestion where we would be allowed to omit ',' in ... Hash definitions, I think, like:

```
x = {
  'foo' => 'bar'
  'bla' => 'ble'
}
```

Or something like this. I forgot whether this can be done or not and matz' opinion on it, but if we ignore this, then I think it's not bad if we were able to omit ',' in code constructs such as that.

Similar reasoning can be applied to 1 (1 1) @2 too, by the way. It's not as if people are suddenly forced to have to use it. But it would be an additional option, similar to the addition of the safe navigation operator. Before the addition of the safe navigation, it was not easily possible, in a succinct way, to query/check for operations that may fail.

I used to bring this example before:

https://github.com/ruby/ruby/commit/91fc0a91037c08d56a43e852318982c9035e1c99

Old code:

```
f.close if f && !f.closed?
```

New code:

```
f&.close
```

The first variant, in my opinion, is simpler to understand (to me at the least); but the second variant is significantly shorter, I think we all have to concede this. And the second aspect to not forget is, syntax preferences aside, is that it added something that was not doable/possible in the same way before.

I completely understand that syntax matters and that syntax is very, very important, but it should not be the sole reason for evaluation, in particular if other changes have had a similar pattern before - be it hashrockets or any other changes. There will be always some folks who will dislike any change; I do so too. :) But it really should not be the sole focus as such, since we can say the same about ANY change in

regards to syntax. Of course there should be reasons for changes, but folks who dislike syntax may often dislike any reason given in favour of a syntax change too, so ... ;)

#42 - 03/30/2019 11:57 AM - bozhidar (Bozhidar Batsov)

For me a big problem is that the syntax additions obviously affect what's being valued/promoted and so on. In the example you gave about the safe navigation operator, the real problem is that f could be nil in the first place. When you add a special syntax for such cases you're effectively promoting the creation of nil-happy APIs. The syntax here by itself is fine, but the underlying sentiment is scary for me.

With this new numbered params syntax many people will feel encouraged to optimize for code golf than for readability (shiny and new things usually have huge uptake in Ruby-land, with the notable exception of refinements). I don't know what "optimizing for happiness" means for everyone, but for me it always meant optimizing for readability and maintainability. I'm certain that golfers would celebrate the new syntax and maybe that's ok - they have to have reasons to be happy as well. :-)

> Nothing. People must write |x| explicitly.

That would be my vote as well. :-)

#43 - 03/30/2019 05:05 PM - dunrix (Damon Unrix)

Hi.

I'm aware this issue is just a tip of an iceberg and Matz has the final say,however I'm worried the Ruby project is slipping from its original path. I mean, how easily it gives green-light to half-assed proposals, how there is considered only one side of the coin.

Is there still valued simplicity, readability and consistency of the language ? Approach the problem in a generic way, without creating superfluous yet-another exceptions to the basic rules ? Are those systemic virtues just relics of the past ?

Getting back to the topic, I'd give a few reasons why the accepted idea of implicit block arguments is intrinsically bad:

- cost of introduced loss in readability and obviousness outweighs benefits of no need for explicit state of arguments
- creates another gap at unifying blocks/procs/lambdas with methods
- creates inconsistency in system of identifiers - symbol @ referencing instance/class variables loses its internal logic
- IDEs, code editors and analytics, would need add another condition - special case for @ followed by digits only
- is limited in use, only very simple cases when one or two arguments are passed and only for a subset of obvious unambiguous calls. Imagine indirect calls/yields to a stored Proc and resulting need for detective work to figure out what is actually passed, when explicit and proper named arguments would help greatly there.

I'd expect development of Ruby language will bring more clean versions with stabilized API, make MRI faster while holding to sane memory footprint. Instead one have to contend with increased complexity without any apparent benefits. Just sharing similar sentiment with my co-workers..

#44 - 03/31/2019 09:35 AM - zverok (Victor Shepelev)

(my small branch in this burning fire)

For me the biggest problem with the new feature is not the particular syntax, but the simple fact that the feature (as far as I can understand) is "orphan", just kinda "nice (or not-that-nice, depends on personal preference) thing to have, why not".

I'd argue that intuitive inclining to "don't repeat variable name in simple blocks" is a good and Rubyish thing, but the solution with "hack" (just throwing some syntax on top) leads us nowhere. @1/@2 doesn't provide new concepts that can be reused in different contexts; it doesn't give a push to rethink how to structure code; it doesn't like any other feature and absolutely unrelated to them (or rather confusingly similar to instance variables -- without trying to conceptualize it, like, for example, saying "OK, let's say Proc instance has internal instance variables named @<num>, and ...").

Counter-example of "good feature" in 2.7 (even if some dislike the syntax) is &obj.:method thing. It targets the same audience ("I want my blocks DRYer"), and probably even will lose the competition if both will be released in 2.7 (because map { File.read(@1) } is kinda "easier to explain" than map(&File.:read)). But "method reference" feature leads to:

- more functional and well-thought code -- designing library and application methods that are easier to use in such chains, without "5 optional arguments";
- movement towards first-class method objects, that are acceptable and usual to people to see stored in variables and collections, passed to other methods and so on;
- keep people thinking on good design for real powerful currying (not the one we have now, which is too verbose and obscure to use), to, maybe, make their peace with something like

```
HTTP.get(url).body.then(&JSON.:parse.(symbolyze_names: true))
```

So, it is about overall shifting of idiomatics towards "functional" way, which could be argued about (whether it is good or bad thing), but at least it is **conscious shift of idiomatics**.

Numbered parameters is "just semantic hack", it doesn't change anything conceptually, just challenges the parsers and makes syntax incompatible with previous versions in a hard way (which, again, is not necessary a bad thing, but definitely a bad thing for just convenience feature).

#45 - 04/02/2019 01:29 AM - phluid61 (Matthew Kerwin)

zverok (Victor Shepelev) wrote:

```
  HTTP.get(url).body.then(&JSON.:parse.(symbolyze_names: true))
```

I hate this. Suddenly you have & and .: and .(), and where everything outside the parens has an obvious receiver-message relationship, nothing inside does.

At least this:

```
HTTP.get(url).body.then { JSON.parse(@1, symbolize_names: true) }
```

removes a bunch of conceptual magic, and replaces it with a bit of syntactic magic.

> Numbered parameters is "just semantic hack", it doesn't change anything conceptually ...

I see that as a positive.

(Side note: the syntax highlighter doesn't like @1 yet...)

#46 - 04/02/2019 07:57 AM - ted (Ted Johansson)

What strikes me is that this feature attempts to solve a problem that is very limited in scope:

- Blocks where the arguments can't be given a meaningful name arguably become more noisy than they may need to.

with a very big hammer:

- Every single block now has access to unnamed variables, which also increase the footprint of the Ruby syntax.

The magnitude of the change doesn't seem to yield proportional value.

Off the top of my head, I can imagine a solution that instead gives the flexibility to yield with injected, block-scope variables. This could:

1. Be implemented as a new method (e.g. yield_with), for backwards compatibility.
2. Be overridden by using the normal block parameter syntax, for backwards compatibility.
3. Work like regular local variables, letting them be overridden, for backwards compatibility.

Example:

```
# Definition

def foo
  yield_with a: 1, b: 2
end

# Usage

foo { a + b }
#=> 3

foo { |c, d| c + d }
#=> 3

foo do
  a = 3
  a + b
end
#=> 5
```

This would allow people to introduce magical, semantic variables at their own discretion.

It doesn't require any new syntax, and the standard library could choose to adopt this partially or fully, at any rate, perhaps even reinforcing existing idioms:

```
[1, 2, 3].map { itself * 2 }
#=> [2, 4, 6]
```

Now. I just came up with this as I was reading through the thread, so I haven't had time to contemplate any of the potential pitfalls, but the point was to highlight that there might be solutions that are more proportional to the problem being solved.

#47 - 04/02/2019 10:22 AM - shevegen (Robert A. Heiler)

I was about to comment on what ted wrote, but it would have become too long,

so here is a significantly shorter variant to just focus on a few points:

- I think "itself" is very similar to "it". Kotlin makes use of "itself", so it is understandable that some folks may also like to see "itself" in ruby.

In my opinion, though, translating from another language into ruby is not always trivial, e. g. see elixir's pipe operator. It is, however had, somewhat interesting to note that languages inspire other languages (or people who use either language).

- To the name "itself" versus "it" alone. Personally I would prefer "it", if we were to have only these two choices available. The primary reason is that it is shorter to type "it" than "itself". For similar reasons "then" is better than "yield_self" (and I am not even using or needing yield_self either; this is just a comment).

- If the argument is that the "scope is too limited", well - not every change has to be of epic proportions either. Omitting require 'pp' is an example of a small but (to me) very nice change. It is also a bit curious to see that a scope would be too limited, but to then see another suggestion made. ;)

I think it is perfectly fine to reason in favour of no change too, by the way. Mame wrote so before. You still have to include the situation that this also means that you may not get new functionality - see prior changes, such as the safe navigation operator or the addition of .: or, in the case here, being able to access block parameters differently ON TOP OF the already existing variant.

It's always a trade-off and you can argue either way, a simpler language with less syntax (yes), but also fewer features. See also @@foo variables - while I myself do not use @@ anymore, others use them and have no problem with them, or have code bases that include them. And without @@ well - you would not have that feature (which can be good or bad depending on your use case too, but it is also a lot to the personal opinion, or the "style" that you use when writing ruby code).

- I am also not entirely sure why some folks reason in favour of things being mutually exclusive here. Let's ignore for the moment the possibility of no change at all (and, would that make people happier? Probably not those who would like any given change suggested in, say, the last three years. But let's just ignore the no-change situation for the moment).

Why would it be completely impossible to, for example, add BOTH 1 (1 1) @2 AND "it" or "itself" or anything else?

I am not saying that this should be done, either. I am just not entirely sure why some reason that it must be mutually exclusive. Why would it not be possible to offer both? Purely from a theoretical point of view to consider alone.

- Syntax is important, but it is also about features and functionality, which should not be forgotten. And not everything can be done without change to syntax alone.

Take the Io language:

```
https://iolanguage.org/about.html
https://iolanguage.org/tutorial.html
```

Syntax-wise it is simpler than ruby, possibly. But it also is restricted in what it can offer, due to that constraint. And less flexible. It's not that it is necessarily not elegant, but ruby's syntax is in my opinion much better. Smalltalk has a, in my opinion, somewhat similar problem syntax wise.

- yield_with is different as to what Stefan wrote about here; at the least he did not mention that we would have to use yield_with. It's also not quite the same or covers other potential use cases, in my opinion, such as accessing nested Arrays/Hashes. Though we could assume that, for example, "it" could be treated like MatchData (for procs/blocks, such as ProcData or BlockData, just to illustrate names), and access via [1] [2] method calls. But even then it is not the same. In my opinion, long names can be somewhat clumsy, so it should ideally be short. But this all comes down to personal opinion really.

#48 - 04/02/2019 11:00 AM - sawa (Tsuyoshi Sawada)

shevegen (Robert A. Heiler) wrote:

> Why would it be completely impossible to, for example, add BOTH 1 (1 1) @2 AND "it" or "itself" or anything else?

The itself keyword is already used under different scope (wider than a block). And it has already been excluded since that would conflict with a testing

framework.

> [W]e could assume that, for example, "it" could
> be treated like MatchData (for procs/blocks, such as ProcData or BlockData,
> just to illustrate names), and access via [1] [2] method calls.

That would require additional method call on whatever the object would be. The point of this feature is that it be directly accessible as a variable.

#49 - 04/02/2019 11:39 AM - DarkWiiPlayer (Dennis Fischer)

I fail to see how the downsides outweigh the problems. The @-syntax looks ugly, misleading and adds even more complexity to the language. The \1, \2, etc. syntax makes much more sense if you deal with regular expressions a lot, but otherwise just looks weird. Using some other random symbol doesn't make it any better at the end of the day.

My opinion: the best compromise would be to add a keyword like that; it instantly reminds of this and has a very similar meaning to it as well. it would be my second pick, for the same reason. For multiple argumets I really think people should just name them; or an alternative syntax could be used. *that seems like a good way to collect all the arguments into an array, but it's ambiguous with the * operator applied to the first argument as an array.

One other thing that could work is copy Luas vararg syntax and make it implicit: When no || is given, ... becomes the first argumet. This would of course look very ugly when calling methods.

As a last idea, any otherwise used character could be chosen, with the rule that, in case of ambiguity, the older meaning always gets chosen, like with the example of using ., so to call and print the foo method, one would have to write puts (.).foo, otherwise Ruby would interpret it as a range.

Ultimately, I really think this feature should just be left out alltogether.

And, just for fun, I will end this on the worst possible idea: use the characters ¹, ² and ³ (Superscript 1, 2 and 3). They are 1. not ascii, 2. incredibly hard to type and 3. small and hard to miss. A true compromise of all ideas, combining the worst of all and the best of all.

#50 - 04/03/2019 06:48 AM - burlesona (Andrew Burleson)

Thanks everyone for the discussion. I think the feature looks nice, but the syntax does concern me, as I think it looks too much like instance variables.

If this is going to be in the language I think $1 and $2 etc. make a lot more sense, they already have a similar use for regex captures, but I understand it may be complicated to disambiguate for that reason.

Everyone has already expressed this more elegantly than me, so rather than put a whole bunch more words I'll just say that I think Ruby is at it's best when it is elegant -- which is most of the time! -- and stumbles when it's ambiguous or inconsistent. This feature, while admittedly a cool idea, seems to me to fall too far on the ambiguous and inconsistent side to be worthwhile, especially when foo.map{|n| n**2 } etc. is already so concise and easy to read.

Thanks again!

#51 - 04/03/2019 10:45 AM - andrewzah (Andrew Zah)

vo.x (Vit Ondruch) wrote:

> mame (Yusuke Endoh) wrote:
>
> > Vote:
>
> I vote for 3 (and hashrockets ;))

I also vote for #3. (and hashrockets as the default syntax).

I'm sorry, but @1 syntax is grotesquely ugly for a language like Ruby. (If this feature absolutely is necessary, it should be \, not @. \1, \2 syntax is already familiar for some people, and @ is already a special character in Ruby.)

But I strongly oppose this feature (with this syntax). Ruby doesn't need random magic symbols instead of the current named variable syntax. It doesn't matter whether or not people will avoid abusing this syntax. (People *will* abuse this syntax for sure, making ruby codebases harder to read, guaranteed). Why even add cryptic sigils in the first place, just to appease some requests?

A significant change to Ruby syntax should not be added merely as a "compromise". If this functionality is so often requested, then a proper, elegant solution should be implemented. Not an ugly compromise.

#52 - 04/03/2019 11:05 AM - andrewzah (Andrew Zah)

> But who is to say that ruby users have to (only) write code like the above? Also, why would this
> have to be assumed that this "must" be in production code? I don't quite understand Stefan here.

There is nothing more permanent than temporary code. What I'm concerned about is open-source libraries and quickly-made scripts using this syntax and never cleaning it up. This also begs the question of "Why add a syntax change if we're already discussing **NOT** using it as good practice?"..

I truly do not understand this change to Ruby.

#53 - 04/03/2019 12:35 PM - xor (Mariusz Ewerest Laszi)

> However, limiting the usefulness to a single argument isn't bad at at. In fact, a single argument seem to be the limit of what makes sense:
>
> ```
> h = Hash.new { |hash, key| hash[key] = "Go Fish: #{key}" }
>
> # vs
>
> h = Hash.new { @1[@2] = "Go Fish: #{@2}" }
> ```

Very ugly and not readable. Plese dont go this way!

#54 - 04/04/2019 05:23 AM - dvogel (Drew Vogel)

matz (Yukihiro Matsumoto) wrote:

> Yes, I admit { @1[@2] = "Go Fish: #{@2}" } is cryptic. But {@1 * @2} is not. So use numbered parameters with care (just like other features in Ruby). The **possibility** to make code cryptic itself should not be the reason to withdraw a feature.

A small syntactic tweak could avoid overloading @ while also making it possible for the meaning to be inferred:

```
h = Hash.new { *0[*1] = "Go Fish: #{*1}" }
```

Using the asterisk symbol and switching the indexes to base-0 makes this a shorthand for:

```
h = Hash.new { |*args| args[0][args[1]] = "Go Fish: #{args[1]}" }
```

The asterisk is already overloaded for multiplication and the splat operator. This means everyone already has to consider the context when mentally parsing it. Implied aliases for positional args are at least conceptually near to the splat operator. Much more so than the @ symbol.

#55 - 04/04/2019 06:09 AM - psychoslave (mathieu lovato stumpf guntz)

What about introducing a keyword that refer to an array rather than a direct access to each element? For example naming this hypothetical array accessor by:

```
h = Hash.new { by[0][by[1]] = "Go Fish: #{by[1]}" }

# Or to my mind even more readable :
h = Hash.new { by.first[by.last] = "Go Fish: #{by.last}" }
# Assume that `.last:` returns the second element, something I'm not sure about here, but you get the point an
yway, don't you?
```

#56 - 04/04/2019 06:34 AM - jeremyevans0 (Jeremy Evans)

dvogel (Drew Vogel) wrote:

> matz (Yukihiro Matsumoto) wrote:
>
> > Yes, I admit { @1[@2] = "Go Fish: #{@2}" } is cryptic. But {@1 * @2} is not. So use numbered parameters with care (just like other features in Ruby). The **possibility** to make code cryptic itself should not be the reason to withdraw a feature.
>
> A small syntactic tweak could avoid overloading @ while also making it possible for the meaning to be inferred:
>
> ```
> h = Hash.new { *0[*1] = "Go Fish: #{*1}" }
> ```
>
> Using the asterisk symbol and switching the indexes to base-0 makes this a shorthand for:
>
> ```
> h = Hash.new { |*args| args[0][args[1]] = "Go Fish: #{args[1]}" }
> ```
>
> The asterisk is already overloaded for multiplication and the splat operator. This means everyone already has to consider the context when mentally parsing it. Implied aliases for positional args are at least conceptually near to the splat operator. Much more so than the @ symbol.

This would not be backwards compatible:

```
class Integer
  def to_a
    [:a]
  end
end

h = {:a=>1, :b=>2}
proc{h[*0]}.call(:b)
# => 1
```

psychoslave (mathieu lovato stumpf guntz) wrote:

> What about introducing a keyword that refer to an array rather than a direct access to each element? For example naming this hypothetical array accessor by:
>
> ```
> h = Hash.new { by[0][by[1]] = "Go Fish: #{by[1]}" }
>
> # Or to my mind even more readable :
> h = Hash.new { by.first[by.last] = "Go Fish: #{by.last}" }
> # Assume that `.last:` returns the second element, something I'm not sure about here, but you get the poin
> t anyway, don't you?
> ```

Some issues with this:

- Not backwards compatible as by could already be a local variable or method.
- You cannot calculate arity with a syntax that uses a single variable for all arguments.
- Requires 5 characters minimum to access an implicit variable.
- We should avoid adding syntax that requires allocating an array or any other object, as that is bad for performance.
- Any approach that used a single variable that was not a true object would be problematic. Consider:

```
x = nil
lambda{x = by}.call(0)
x[0]
```

#57 - 04/04/2019 09:35 AM - psychoslave (mathieu lovato stumpf guntz)

Thank you Jeremy for your detailed feedback.

> Not backwards compatible as by could already be a local variable or method.

One could just assume precedence of locally defined identifiers over keywords, but that is a whole other point. Personally I would be fine with a facility that would not make keywords locked reserved words, but won't argue for that here, and don't expect it to be a retained solution in this scope.

One easy way to do that in a backward compatible way though is to disable it by default. If the goal is to provide a syntax convenient for quick and dirty read–eval–print loops while discourage its use in code bases, that would also seem a saner approach.

> You cannot calculate arity with a syntax that uses a single variable for all arguments.

If you mean that an array would not be the right structure to use, what about using a hash indexed with integer? (goodbye by.first of course)

> Requires 5 characters minimum to access an implicit variable.

Well, sure, but as far as I'm concerned readability largely outweighs the benefit of a few keystrokes

> We should avoid adding syntax that requires allocating an array or any other object, as that is bad for performance.

I agree that performance matter, but syntax definition is not implementation enforcement.

On this regard, I personally feel that Ruby is not pushing its "everything is an object" far enough: if.class is not expected to work. Of course there are (understandable) conceptual simplicity and performance issues underlying this choice. But here too, this would not require the implementation to treat such a construct as it indeed treats user defined classes, just to provide an homogeneous behavior to the language user.

> Any approach that used a single variable that was not a true object would be problematic.

Well that achieves to convince me the suggestion is not that great, thank you.

I'm not convinced with the current proposal either, that said. If there is one thought that I would think worth considering in my text above, it's to turn the feature off by default and let user enable it at their own discretion.

#58 - 04/04/2019 10:07 AM - phluid61 (Matthew Kerwin)

psychoslave (mathieu lovato stumpf guntz) wrote:

> If there is one thought that I would think worth considering in my text above, it's to turn the feature off by default and let user enable it at their own discretion.

That's like a worse version of "don't include it in your code by default and let user include it in their code at their own discretion."

The argument against the feature is people who don't want to use it, who inherit code (one way or another) from people who did use it and are therefore forced to use it. If you don't see it, it doesn't hurt you.

#59 - 04/05/2019 01:01 AM - dvogel (Drew Vogel)

jeremyevans0 (Jeremy Evans) wrote:

> dvogel (Drew Vogel) wrote:
>
> > The asterisk is already overloaded for multiplication and the splat operator. This means everyone already has to consider the context when mentally parsing it. Implied aliases for positional args are at least conceptually near to the splat operator. Much more so than the @ symbol.
>
> This would not be backwards compatible:
>
> ```
> class Integer
>   def to_a
>     [:a]
>   end
> end
>
> h = {:a=>1, :b=>2}
> proc{h[*0]}.call(:b)
> # => 1
> ```

I make this suggestion knowing that it would have to be delayed until ruby 3. I consider the improvement in terms of simplicity and approachability to be worth the cost of waiting for ruby 3. I think about keyword arguments as another useful feature that was delayed long past the point it was obvious the feature should be added in order to make sure it was consistent and sustainable. As useful as keyword arguments are, I suspect most of us would have balked at awkward syntax in order to get it sooner. Could you imagine if this were proposed for a theoretical 1.10?

```
def f(@*kwargs)
  g(@*kwargs)
end
```

#60 - 04/05/2019 01:53 AM - jeremyevans0 (Jeremy Evans)

dvogel (Drew Vogel) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > dvogel (Drew Vogel) wrote:
> >
> > > The asterisk is already overloaded for multiplication and the splat operator. This means everyone already has to consider the context when mentally parsing it. Implied aliases for positional args are at least conceptually near to the splat operator. Much more so than the @ symbol.
> >
> > This would not be backwards compatible:
> >
> > ```
> > class Integer
> >   def to_a
> >     [:a]
> >   end
> > end
> >
> > h = {:a=>1, :b=>2}
> > proc{h[*0]}.call(:b)
> > # => 1
> > ```
>
> I make this suggestion knowing that it would have to be delayed until ruby 3. I consider the improvement in terms of simplicity and approachability to be worth the cost of waiting for ruby 3.

Even if you don't consider someone defining Integer#to_a to use * as a unary operator, it would create issues when * is used as a binary operator:

```
proc{foo *0}.call(1)
```

Does this mean call foo, and call * on the result with the argument 0, or call foo with the first argument passed to the block?

> I think about keyword arguments as another useful feature that was delayed long past the point it was obvious the feature should be added in order to make sure it was consistent and sustainable.

That's probably not the best counterexample. Keyword arguments were added too early, before the problematic issues with optional positional arguments and positional splats were understood, which we are trying to address in #14183.

> As useful as keyword arguments are, I suspect most of us would have balked at awkward syntax in order to get it sooner.

People complained about the "awkward" syntax for the lonely operator &., the call shortcut .(), stabby lambdas ->{}, and probably many other additions to Ruby's syntax. In my opinion, the @1 syntax probably only looks awkward because it is unfamiliar. That doesn't mean that it is good syntax, or that the feature itself is worthwhile, of course. But of the other proposals considered, I think only \1 wouldn't present backwards compatibility issues, and that is associated with string escaping, newline continuation, and regexp backreferences, not object access.

#61 - 04/05/2019 08:41 AM - psychoslave (mathieu lovato stumpf guntz)

Hello Matthew! You wrote:

> The argument against the feature is people who don't want to use it, who inherit code (one way or another) from people who did use it and are therefore forced to use it. If you don't see it, it doesn't hurt you.

I don't think that's the main point here, at least to my mind it's . Readability, consistency and lake of surprise are the main concern here. The @ prefix coming in semantic conflict with the already used for instance variable is the most worrying point, to my mind.

This morning I woke up with this idea in mind: maybe we could use a letter suffix.

Sticking to English custom and habit regarding ordinal number notation, the proposal could take the form of this arguments could become : 1st, 2nd, 3rd, 4th, 5th, and so on. This as the obvious con of irregularity on the three firsts that are most likely being the most used. Also a bit longer than the arobase prefix equivalent. The pro however is that it should make sense to anyone with a basic knowledge of English.

An other proposal would be to use a single letter. For example using a as suffix, you end up with 1a, 2a, 3a and so on. Fully regular, as short to type as the arobase prefixed version, and I guess conflict-free with other use of the letter, as well as backward compatible. Bonus, -a is actually used to form ordinal number in Esperanto! Ok, I don't expect this last one to come as a strong point, but it's funny to note that. One con with this option would be that one could at first be confused that 1a actually stands for 0x1a (26). But once you have the notation in mind it's not more problematic than 1e3 (1000, engineering notation). At worst you can always peak any other letter out of the [a-f] range.

Cheers.

#62 - 04/05/2019 09:36 AM - sawa (Tsuyoshi Sawada)

psychoslave (mathieu lovato stumpf guntz) wrote:

> Sticking to English custom and habit regarding ordinal number notation, the proposal could take the form of this arguments could become : 1st, 2nd, 3rd, 4th, 5th, and so on.

I hope you are not trying to repeat the joke in #15741 posted four days before.

#63 - 04/05/2019 10:18 AM - psychoslave (mathieu lovato stumpf guntz)

Hello Tsuyoshi Sawada:

> I hope you are not trying to repeat the joke in #15741 posted four days before.

I wasn't aware of this thread (thank you for the hint), and didn't try to make a joke with my own totally independent proposal.

#64 - 04/05/2019 11:05 AM - duerst (Martin Dürst)

psychoslave (mathieu lovato stumpf guntz) wrote:

> An other proposal would be to use a single letter. For example using a as suffix, you end up with 1a, 2a, 3a and so on. Fully regular, as short to type as the arobase prefixed version, and I guess conflict-free with other use of the letter, as well as backward compatible.
>
> One con with this option would be that one could at first be confused that 1a actually stands for 0x1a (26).

As @1,... could be confused with @instance variables and @@class variables.

> But once you have the notation in mind it's not more problematic than 1e3 (1000, engineering notation).

I and others have argued above that once you have the @1 notation in mind, it's also not a problem.

#65 - 04/05/2019 12:58 PM - psychoslave (mathieu lovato stumpf guntz)

Thank you Martin for your feedback.

I agree there is a degree of similarity between the thought disambiguation. Actually to some extent it even makes me consider using @ prefix as a more coherent option.

In the same time, this *feels* like it's not something that would improve Ruby. So here there is something very subjective. But subjectivity matter too. Not that my personal opinion matter particularly, of course. This would feel far less uncomfortable with something like 1.given, just like 1.times is great. Although that last definitely doesn't go toward winning the shortest token contest (you can of course go with shortest words like 1.it).

#66 - 04/06/2019 12:38 AM - marcandre (Marc-Andre Lafortune)

I would like a notation to replace {|x| ...}, and *only* to replace this.

I strongly disagree with current implementation:
1) that allows multiple unnamed parameters
2) where {@1} actually doesn't replace {|x| ...} but {|x, | ...}

- Rationale *

1) allowing multiple unnamed parameters is a mistake

- it usually wouldn't make code clear
- it's not that useful (see stats at the end)
- the conciseness gain of allowing to write a number in @2, ... is negated by having to write "1" for the simple case @1 (see stats at the end)

2) { @1 ... } (or whichever shorthand we use) should replace {|x| ...}, not {|x, | ...}

- the |x| form is used 128x times more than the |x, | (see stats at the end). That's basically always what we want to say.
- the |x, | can lead to errors that can be hard to detect.

A typical example is anything that handles heterogenous objects:

```
  def process(*args)
    args.each { _process_single(@1) }
  end
```

This will appear to work fine, until one argument is an array and this will call _process_single on the first element of the array instead of the array itself.

- Stats *

```
493693 (78 %): {}
115049 (18 %): {|x|}
17685 (2.8 %): {|x, y|}
2330 (0.37 %): {|x, y, z|}
1053 (0.17 %): {|*rest|}
901  (0.14 %): {|x,|}
890  (0.14 %): {|x, (...)}
558  (0.09 %): {|w, x, y, z|}
      (0.14%): others
```

These are for 633k block signatures of the 400+ top gems.

Note that stats vary depending on the project style, e.g. mongoid has 96% of its blocks without arguments, compared to 71% for Rails, but the totals on 400+ projects should give a pretty good image.

Gain for {|x, y| x + y} => {@1 + @2} is at most 5 characters
The extra "1" for {|x| x} => {@1} costs (at least) 1 character compared to {@}. Assuming we use the shorthand all the time, this cost is incurred 6.5 times more often than the "gain" for the @1 + @2 so is a net loss compared to simply @!

- Conclusion *

Let's first agree that a shorthand notation would only simplify {|x| foo(x) } and no other block signature.

Then we can discuss if it should be @1, @, it, or my personal favorite (that I havn't seen discussed): _.

#67 - 04/06/2019 01:52 AM - jeremyevans0 (Jeremy Evans)

Marc,

Thank you for your analysis, I think you have raised a number of good points.

marcandre (Marc-Andre Lafortune) wrote:

> Let's first agree that a shorthand notation would only simplify {|x| foo(x) } and no other block signature.

I agree that a shorter notation for {|x|} instead of {|x,|} would be beneficial. That does not imply that that a shorthand for {|x,y|} is not useful (e.g. {@1 + @2}), though I think it is definitely less useful.

> Then we can discuss if it should be @1, @, it, or my personal favorite (that I havn't seen discussed): _.

My original proposal was @ for a single block argument, and @1 and @2 for multiple block arguments as a possible extension. It may make sense to have {@} means {|x| x} and {@1 + @2} means {|x,y| x + y}. We would want to disallow mixing @ and @1/@2 in that case, and maybe we should consider disallowing @1 without @2 or some higher number, since the actual desire for {|x,|} behavior is probably limited.

As for it and _, both are not backwards compatible as they are valid local variables:

```
it = 1
_ = 2
lambda{it}.call
# => 1
lambda{_}.call
# => 2
```

#68 - 04/06/2019 03:36 AM - marcandre (Marc-Andre Lafortune)

jeremyevans0 (Jeremy Evans) wrote:

> Marc,
>
> I agree that a shorter notation for {|x|} instead of {|x,|} would be beneficial. That does not imply that that a shorthand for {|x,y|} is not useful (e.g. {@1 + @2}), though I think it is definitely less useful.

I showed it is *much less* useful. The cases |x, y|, |x, y, z| and |w, x, y, z| are making only 3.26 % of the block signatures (14.8% of signatures with arguments).

That is not much *and* most of them would not benefit from the shorthand notation.

Here are stats on |x, y| signatures:

```
3378 (22 %): [:key, :value]    (or synonyms like :k, :v)
545 (3 %): [:some_word, :_]
497 (3 %): [:hash, :key]   (or synonyms like :h, :k)
416 (2.6 %): [:x, :y]    (or any sequence like :a, :b)
369 (2.3 %): [:some_word, :i]
279 (1.8 %): [:x, :_]    (or any single letter and :_)
187 (1.2 %): [:name, :value]
180 (1.1 %): [:x, :i]     (or any single letter and :i)
1..174 (63 %): [:record, :time], [:args, :options], [:data, :type] and 4414 more
```

I would argue that the only cases where using the shorthand might be as readable as before are [:x, :y], [:x, :_]. If we are very generous, we can include [:some_word, :_] and [:x, :i]. These still make up 8.5% of the two argument blocks.

Let's be generous again and apply that same 8.5% to the case with 3 and 4 parameters, we get that 0.27% of all blocks might benefit from @1 and @2... That seems completely negligible.

> As for it and _, both are not backwards compatible as they are valid local variables

Yes, I understand that. The fact remains that _ is meant for unused variables, so cases where not only is the _ variable actually read from but passed to a closure should be none or close to that. I might try to look these up in my code database.

#69 - 04/06/2019 07:00 AM - sawa (Tsuyoshi Sawada)

Marc, regarding |x, |, I see plenty of cases like map(&:first) (or map(&:last)). Probably taking care of these cases is more important than the explicit |x, | cases. If you are going to count, you better count these in. And if directly using an implicit block variable @1 is more efficient than calling first (which probably is the case), then replacing map(&:first) with map{@1} would make sense.

At the same time, this would also mean that, if we are going to have the @1 notation, then there would be a desire to count the arguments from the end as well using a negative number n in order to replace map(&:last) with something like map{@n}. (Note that n = -1 would not work here since that

conflicts with the 1-origin numbering.)

If we are to adopt named implicit block variables instead, then we might simply name these something like first and last, and perhaps whole for the x in |x|.

#70 - 04/07/2019 06:08 PM - Eregon (Benoit Daloze)

I completely agree with marcandre (Marc-Andre Lafortune) here.

The cases using @2 and higher are extremely rare, all feel pretty hard to read to me and force the simple case of a single argument to be more complicated and look more weird (@1 instead of just @, _, it, ...).
These are probably part of the reasons why some languages only provide a shortcut for a single argument, and not multiple:

- it's not worth the readability loss.
- @2 can only be used in very few cases without being clearly worse than naming arguments.

sawa (Tsuyoshi Sawada) The |x,| behavior for @1 can only be considered a bug. It prevents array_of_arrays.each { p @1 } to work correctly.
nobu (Nobuyoshi Nakada) Could you fix this? I don't think it's intended.

#71 - 04/07/2019 06:25 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> But of the other proposals considered, I think only \1 wouldn't present backwards compatibility issues, and that is associated with string escaping, newline continuation, and regexp backreferences, not object access.

Block arguments are not object accesses. They are local variables. That's part of why I do not like @ much for this purpose (but it looks already more reasonable to me than @1).
Block arguments do not access a receiver, which @ indicates. Since @foo means read the instance variable foo of self, then @ would logically just be self.

If we think about pronunciation, @1 would be at(1) such as in array@1, which has nothing to with reading a block argument.

If we cannot find a nice syntax for this feature, I think it's a sign maybe we shouldn't have that feature, or wait to find a nice syntax for it.

#72 - 04/07/2019 06:43 PM - Eregon (Benoit Daloze)

It's kind of interesting that in this issue, I can count only 4 people which seem happy with the current state, and all the 25 others think the syntax should be changed or be removed.

Maybe it's time to rethink this feature and acknowledge the current compromise is not good enough.
Then, we could start the discussion without the current syntax forced on us, and think what would be a better syntax for this feature.
I wouldn't want this feature to get released in Ruby 2.7 just because time passed and people could not agree on an alternative syntax.

#73 - 04/08/2019 11:58 AM - sawa (Tsuyoshi Sawada)

Eregon (Benoit Daloze) wrote:

> sawa (Tsuyoshi Sawada) The |x,| behavior for @1 can only be considered a bug. It prevents array_of_arrays.each { p @1 } to work correctly.
> nobu (Nobuyoshi Nakada) Could you fix this? I don't think it's intended.

Eregon (Benoit Daloze) I think you better check the relevant threads before dumping comments in this and other threads.

https://bugs.ruby-lang.org/issues/15708#note-1

#74 - 04/08/2019 06:15 PM - Eregon (Benoit Daloze)

sawa (Tsuyoshi Sawada) Thank you for think link, I missed that issue.
I stand by my opinion though, and expressed it there too.

#75 - 04/08/2019 07:17 PM - pascalbetz (Pascal Betz)

Eregon (Benoit Daloze) wrote:

> sawa (Tsuyoshi Sawada) Thank you for think link, I missed that issue.
> I stand by my opinion though, and expressed it there too.

Even more confusion ahead.

I don't see that this feature adds any benefit besides saving some chars in an ideal case.
On the downside it will lead to much confusion, especially with people new to the language (but not only).

#76 - 04/08/2019 10:08 PM - marcandre (Marc-Andre Lafortune)

sawa (Tsuyoshi Sawada) wrote:

> Marc, regarding |x, |, I see plenty of cases like map(&:first) (or map(&:last)). Probably taking care of these cases is more important than the explicit |x, | cases. If you are going to count, you better count these in. And if directly using an implicit block variable @1 is more efficient than calling first (which probably is the case), then replacing map(&:first) with map{@1} would make sense.

Did you try to verify any your assertions before commenting?

1) &:first and { @1 } are not always equivalent, where { @1 } could fail to show an error in the code or data. The "gain" is very disputable.

2) I got 174 occurrences of &:first, which adds to a negligible 0.03% contribution and is 5x less frequent that |x, |

3) There is no noticeable speed difference between map(&:first) and map{ @1 }

> At the same time, this would also mean that, if we are going to have the @1 notation, then there would be a desire to count the arguments from the end as well using a negative number n in order to replace map(&:last) with something like map{@n}. (Note that n = -1 would not work here since that conflicts with the 1-origin numbering.)

I don't know if you are serious or are trolling. This thread is about showing that @1, @2 are not really useful and potential source of confusion / errors. I feel that proposing instead to extend this notation for accessing arguments in reverse order is not appropriate.

> If we are to adopt named implicit block variables instead, then we might simply name these something like first and last, and perhaps whole for the x in |x|.

Again, it's not clear to me if you are seriously proposing to add a source of incompatibility for something that I showed to be of little use.

#77 - 04/09/2019 07:39 AM - sawa (Tsuyoshi Sawada)

Marc, okay, I agree with you. But if not in the form of &:first or &:last, I still frequently see the methods first and last used in blocks, and I see that those can be accessed as _.first, _.last according to your proposal, and that would be convenient enough. Though, I don't understand why you are suspecting that I am trolling. That is not good.

#78 - 04/11/2019 09:33 AM - lloeki (Loic Nageleisen)

I very much like the feature itself, esp. for one liners you can come up with on a pry prompt. What I very much dislike though is the use of @ which is a glorious hack if there is any since currently @ tokenizably resolves to 'instance variable' in many a brain. In fact I'd very much be happy to use %1 as a syntax since that would match the %1 placeholders in strings right away: % easily translates to "positional". "%1" in strings, \1 in regex for backrefs, $1 for regex groups... Yes, { %1 } in blocks somehow works as a metapattern, but reusing @1 vs @myvar is such a mismatch that it brings a terrible cognitive dissonance (just see e.g how in [1, 2, 3].each { @1 + @l } both @ have different scopes!).

_ is often used to ignore an argument (like e.g {}.each { |_, v| ... }) or to ignore a return value (like a, _ = "".split(',', 1)), so using it as JS's arguments keyword feels off. Yet simultaneously, a block that has no explicit arguments would obviously not have _ in its argument list, so this somehow works, but still conflicts with the second case I think. Continuing with the % notation, maybe %_ would be semantically nice, echoing both to the % "positional" meaning and the $_ variable.

#79 - 04/12/2019 07:11 PM - headius (Charles Nutter)

What about using _1, _2? These are value variable names that could be reserved for argument offsets. This would mimic some golfed block code like {|_| ...} (though I know there's some debate about whether @1 should map to |x| or |x,|).

There are pros:

- It still looks like a local variable.
- Things with leading underscores are often hidden or internal; in this case, these names would be hidden references to the passed-in arguments.
- It will parse on older Ruby impls.

And some cons:

- Though it parses on older impls, it will parse as a method call.

This is probably fine, though, since it will error using NameError as being a missing local *or* method.

- It may conflict in some cases with Ruby code using _### variable names.

However this new meaning is only valid within a block, and only within a block that has no arguments, plus since these are treated as normal local variables, doing {_1 = 'foo'} will not conflict in any way (user does not expect to receive vars, so arg list is irrelevant; _1 is a valid local var name, so it will still work correctly).

#80 - 04/12/2019 07:12 PM - headius (Charles Nutter)

Oh, another simplication... if user code in 2.7+ attempts to read _1 before writing it in a block, that can be an indication that it's expected to be an

argument offset. If it's assigned first it's just a normal local variable.

#81 - 04/12/2019 08:08 PM - Eregon (Benoit Daloze)

Another idea for syntax, already mentioned in https://bugs.ruby-lang.org/issues/4475#note-12 : &1, &2, etc.

& associates much better with blocks than @: map(&:name), foo(&block), etc.
In fact, I think many previous suggestions for more concise block syntax used &.

Yes, it would break a program that define Integer#to_proc (https://bugs.ruby-lang.org/issues/4475#note-13).
But I would bet those are extremely rare, and breaking those programs sounds worth it for a meaningful syntax
(I think many people agree @ has little to nothing to do with blocks).

#82 - 04/12/2019 08:10 PM - headius (Charles Nutter)

> Another idea for syntax, already mentioned in https://bugs.ruby-lang.org/issues/4475#note-12 : &1, &2, etc.

Conflicts with bitwise AND, doesn't it?

```
foo &1
```

Is it foo() & 1 or foo(&1)?

#83 - 04/12/2019 08:29 PM - Eregon (Benoit Daloze)

headius (Charles Nutter) wrote:

> Conflicts with bitwise AND, doesn't it?

No, binary operators already care about spacing to clarify which one it is, and this syntax "conflict" already exists anyway:

```
irb(main):001:0>  class Integer; def to_proc; -> x { x + self }; end; end
=> :to_proc
irb(main):002:0> 2.then(&3)
=> 5
irb(main):003:0> 2.then &3
=> 5
irb(main):004:0> 2.then & 3
Traceback (most recent call last):
        4: from /home/eregon/.rubies/ruby-trunk/bin/irb:23:in `<main>'
        3: from /home/eregon/.rubies/ruby-trunk/bin/irb:23:in `load'
        2: from /home/eregon/prefix/ruby-trunk/lib/ruby/gems/2.7.0/gems/irb-1.0.0/exe/irb:11:in `<top (require
d)>'
        1: from (irb):4
NoMethodError (undefined method `&' for #<Enumerator: 2:then>)
```

We live fine with it so far, and I think it's fairly intuitive.

#84 - 04/12/2019 09:11 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> headius (Charles Nutter) wrote:
>
> > Conflicts with bitwise AND, doesn't it?
>
> No, binary operators already care about spacing to clarify which one it is, and this syntax "conflict" already exists anyway:
>
> ```
> irb(main):001:0>  class Integer; def to_proc; -> x { x + self }; end; end
> => :to_proc
> irb(main):002:0> 2.then(&3)
> => 5
> irb(main):003:0> 2.then &3
> => 5
> irb(main):004:0> 2.then & 3
> Traceback (most recent call last):
>         4: from /home/eregon/.rubies/ruby-trunk/bin/irb:23:in `<main>'
>         3: from /home/eregon/.rubies/ruby-trunk/bin/irb:23:in `load'
>         2: from /home/eregon/prefix/ruby-trunk/lib/ruby/gems/2.7.0/gems/irb-1.0.0/exe/irb:11:in `<top (req
> uired)>'
> ```

```
        1: from (irb):4
NoMethodError (undefined method `&' for #<Enumerator: 2:then>)
```

We live fine with it so far, and I think it's fairly intuitive.

The behavior changes if foo is a local variable:

```
foo = 1
proc{foo &1}.call
# => 1
proc{foo & 1}.call
# => 1
```

This proposal has the following issue, where the same token (&1) could mean different things:

```
proc{&1.(&1)}
```

Is this:

```
proc{|x| x.call(x)}
```

or:

```
proc{|x| x.call(&(1.to_proc))}
```

@1 has the advantage that such a syntax is not currently valid at any point, and therefore it does not cause backwards compatibility issues. I think using &1 would cause problems, because any time you passed it as the last argument to a method, it would ambiguous as to whether you mean Integer#to_proc or the implicit block argument.

#85 - 04/12/2019 10:38 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> The behavior changes if foo is a local variable:
>
> ```
> foo = 1
> proc{foo &1}.call
> # => 1
> proc{foo & 1}.call
> # => 1
> ```

Good catch, I forgot about that case.
Indeed, that's confusing.
This case can be worked around with parentheses if there is a local variable of the same name: foo(&1).
(This issue already exists if there is a variable of the same name and the method takes no argument; it then needs explicit parentheses or receiver: foo=1; foo())

I think a possible way to solve this is to make &1, &2, ... special in the lexer (e.g., a /(\s+|delimiter)&[1-9]/ rule), so they follow the spacing rules, ignoring whether LHS is a variable or something else.
Another way to see this is tweaking the rule for distinguishing unary and binary operators, so that for &1 there is no special behavior if LHS is a variable.

> This proposal has the following issue, where the same token (&1) could mean different things:
> I think using &1 would cause problems, because any time you passed it as the last argument to a method, it would ambiguous as to whether you mean Integer#to_proc or the implicit block argument.

I think that can be solved easily with "unary & + integer literal" becoming a single AST node and expression, so they always mean "block parameter" and never "Integer#to_proc".
(We'd anyway change the semantics of when Integer#to_proc is called, I think basically nobody uses Integer#to_proc, and making it clear is better)

#86 - 04/13/2019 04:44 AM - dunrix (Damon Unrix)

Don't you think it may be just the high time stop throwing yet-another of *half-baked hacks* and cease work on implementation & release of the #4475 feature, until other and more mature solution is found ?

It seems the whole concept of implicit block arguments need to be rethought carefully, if sustainability and relevance of the language is still taken seriously. I'd suggest start from anew under a new feature request, while requisite rules of good language design are taken into a consideration.

Existing solution for example breaks rule of the general approach, as it only works with (anonymous)blocks and procs, not with lambdas and methods which are strict in arguments. It only adds to fragmentation of the language.
Existing solution also breaks rule of the semantic consistency with introduction of contextually-dependent exception, when existence of implicit block variables is dependent upon absence of explicitly stated arguments.

Implicit arguments should work universally, ie. also for lambdas and methods. It follows that they have to be independent on presence of explicitly stated arguments. In addition, it requires transparency for all kind of arguments - both positional and keyword incl. "greedy" varargs, with or without default values.
Should not clash with existing syntax or scope/identifier naming or even break backwards compatibility.

I'd suggest start some deeper discussion in some other place, then fill a new feature request after main objections will be already addressed.

#87 - 04/13/2019 08:08 AM - jeremyevans0 (Jeremy Evans)

dunrix (Damon Unrix) wrote:

> Existing solution for example breaks rule of the general approach, as it only works with (anonymous)blocks and procs, not with lambdas and methods which are strict in arguments. It only adds to fragmentation of the language.

I'm not sure what led you to this statement, as the feature does work with lambdas and methods:

```
lambda{@1+@2}.call(1, 2)
# => 3

lambda{@1+@2}.call(1)
# ArgumentError (wrong number of arguments (given 1, expected 2))

define_method(:a){@1+@2}
a(1, 2)
# => 3

a(1)
# ArgumentError (wrong number of arguments (given 1, expected 2))
```

> Implicit arguments should work universally, ie. also for lambdas and methods.

They do.

> It follows that they have to be independent on presence of explicitly stated arguments.

I do not think that follows, and having both explicit arguments and implicit arguments is more likely to lead to confusion.

> In addition, it requires transparency for all kind of arguments - both positional and keyword incl. "greedy" varargs, with or without default values.

I disagree. This syntax can make simple blocks even simpler, and simple blocks are much more common than complex blocks. Implicit block arguments do not have to support all complex cases to be useful.

> Should not clash with existing syntax or scope/identifier naming or even break backwards compatibility.

The syntax was explicitly chosen to be invalid in earlier versions of Ruby, to avoid backwards compatibility issues. "Clashes with existing syntax" is either false if taken literally (as the syntax is invalid in previous versions of Ruby), or subjective if not taken literally (such as "it doesn't feel right because I am used to @ being only used for instance variables").

#88 - 04/13/2019 10:09 AM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> I'm not sure what led you to this statement, as the feature does work with lambdas and methods:

Well, only with define_method methods, not def methods, which are the huge majority:

```
def a
  @1 + @2
end
```

gives

```
tt.rb:2: numbered parameter outside block
  @1 + @2
tt.rb:2: syntax error, unexpected unary+, expecting `end'
  @1 + @2
tt.rb:2: numbered parameter outside block
  @1 + @2
```

So it doesn't work with literal def method definitions at all.
Method definitions are rarely single-line, so I don't think this syntax would be useful for them anyway.

Since stabby lambdas have a different syntax for arguments, the gain of characters is so tiny that it doesn't seem worth it:

```
-> { @1 + 2 } # 1 char gain, for less readability
-> x { x + 2 }

-> { @1 ** @1 } # nothing gained
-> x { x ** x }

-> { @1.zip(@1, @1) } # 1 char longer!
-> x { x.zip(x, x) }

-> { @2 / @1 }
-> x,y { y / x }
```

I also feel it loses the nice "lambda" feeling of a mathematical formula, and just becomes some cryptic syntax which is unclear about how many arguments it takes,
while a lambda should be very clear/strict about how many arguments it takes, since it's the whole points of a lambda vs a block/Proc in the first place.

#89 - 04/13/2019 10:14 AM - dunrix (Damon Unrix)

jeremyevans0 (Jeremy Evans) wrote:

> I'm not sure what led you to this statement, as the feature does work with lambdas and methods:
> …

Hmm, not obvious from description in the accepted feature request. By recent 2.7.0-dev implementation I can confirm numbered params are also present in those constructs, however that makes things even more inconsistent and confusing.
It now makes lambdas arguments exclusively derived from occurrences of numbered params in their bodies, while it may be ok for procs with their non-strict arguments.

```
# lambda with(out?) how many arguments ?
lambda { "some multi-line string with #{@1} "\
         "argument and eventually, "\
         "some #{@22} another one."}.call(1)
# ArgumentError (wrong number of arguments (given 1, expected 22))

# not possible combine with safeguard explicit arguments
lambda {|x,y| "some multi-line string with #{@1} "\
              "some #{@22} another one."}
# SyntaxError ((irb):57: ordinary parameter is defined)
# ...ome multi line string with #{@1}" }
```

> define_method(:a){@1+@2}

Only this special way of method definition, when body is replaced with passed block/proc.
Not unified with normal method definition:

```
def a();@1+@2;end
# SyntaxError ((irb):63: numbered parameter outside block)
```

> I disagree. This syntax can make simple blocks even simpler, and simple blocks are much more common than complex blocks. Implicit block arguments do not have to support all complex cases to be useful.

I disagree. As already demonstrated, it makes code less obvious while saved typing is minuscule - only *single* character for one argument, *none* for two or more. Where numbered arguments can be scattered anywhere in block with jumbled order. Not mentioning less flexibility, like compound object deconstruction or splat-consuming.

> The syntax was explicitly chosen to be invalid in earlier versions of Ruby, to avoid backwards compatibility issues. "Clashes with existing syntax" is either false if taken literally (as the syntax is invalid in previous versions of Ruby), or subjective if not taken literally (such as "it doesn't feel right because I am used to @ being only used for instance variables").

Huh, I've been speaking about brand new, completely reworked proposal ..

#90 - 04/16/2019 12:20 AM - nobu (Nobuyoshi Nakada)

- Related to Bug #15708: Implicit numbered argument decomposes an array added

#91 - 04/17/2019 12:13 PM - jeremyevans0 (Jeremy Evans)

As I expressed in the developer meeting today, after a lot of thought, I believe if we want to support implicit block arguments, we should limit support to a single argument, and use @ to represent the argument.

As Marc showed, blocks that accept a single argument are much more popular than blocks that accept more than one argument. For blocks that accept multiple arguments, referencing the arguments by position rather than by name will make the code less understandable, not more.

Most of the objections with @ in regards to syntax are because the references look like instance variables, and most Rubyists do not know that @1 is not a valid instance variable. Using a bare @ should avoid or at least mitigate that problem. I don't know if there is a sigil other than @ that will work. \ worked for the case where you are using a position number, but it will not work without that, because then it can be interpreted as a line continuation. % and similar sigils that are binary operators cannot be used because they will be interpreted as a binary operator:

```
proc do
  foo %
  bar
end
# parsed as proc{foo().%(bar())}
```

#92 - 04/17/2019 12:34 PM - waheedi (Waheed Barghouthi)

jeremyevans0 (Jeremy Evans) wrote:

> As I expressed in the developer meeting today, after a lot of thought, I believe if we want to support implicit block arguments, we should limit support to a single argument, and use @ to represent the argument.
>
> As Marc showed, blocks that accept a single argument are much more popular than blocks that accept more than one argument. For blocks that accept multiple arguments, referencing the arguments by position rather than by name will make the code less understandable, not more.
>
> Most of the objections with @ in regards to syntax are because the references look like instance variables, and most Rubyists do not know that @1 is not a valid instance variable. Using a bare @ should avoid or at least mitigate that problem. I don't know if there is a sigil other than @ that will work. \ worked for the case where you are using a position number, but it will not work without that, because then it can be interpreted as a line continuation. % and similar sigils that are binary operators cannot be used because they will be interpreted as a binary operator:
>
> ```
> proc do
>   foo %
>   bar
> end
> # parsed as proc{foo().%(bar())}
> ```

That's great news. It makes more sense for a single argument. @ on its own should be enough. But why it can't be like this? @[0] I think its more clear, just saying.

#93 - 04/17/2019 12:59 PM - jeremyevans0 (Jeremy Evans)

waheedi (Waheed Barghouthi) wrote:

> But why it can't be like this? @[0] I think its more clear, just saying.

If you mean that you would like the intended behavior to be:

```
proc{@[0]}.call(1,2)
# => 1
```

You would only want this if you wanted to support multiple implicit block arguments, which is almost always going to result in code that is more difficult to understand, as referencing arguments by position instead of by name is problematic. Only supporting a single implicit block argument makes the simple case simpler and in some cases clearer.

Even assuming you wanted to support multiple implicit block arguments, I responded earlier with some issues with using a single variable for that:

- You cannot calculate arity with a syntax that uses a single variable for all arguments.
- Requires 4 characters minimum to access an implicit variable.
- We should avoid adding syntax that requires allocating an array or any other object, as that is bad for performance.
- Any approach that used a single variable that was not a true object would be problematic.

Note that my proposal would support @[0], but it would mean call the [] method on the first argument to the block with the value 0 (i.e. @[0] means @.[](0)). Example:

```
proc{@[0]}.call({0=>1})
# => 1
```

#94 - 04/17/2019 02:09 PM - waheedi (Waheed Barghouthi)

jeremyevans0 (Jeremy Evans) wrote:

> waheedi (Waheed Barghouthi) wrote:
>
> > But why it can't be like this? @[0] I think its more clear, just saying.
>
> If you mean that you would like the intended behavior to be:
>
> ```
> proc{@[0]}.call(1,2)
> # => 1
> ```
>
> You would only want this if you wanted to support multiple implicit block arguments, which is almost always going to result in code that is more difficult to understand, as referencing arguments by position instead of by name is problematic. Only supporting a single implicit block argument makes the simple case simpler and in some cases clearer.
>
> Even assuming you wanted to support multiple implicit block arguments, I responded earlier with some issues with using a single variable for that:
>
> - You cannot calculate arity with a syntax that uses a single variable for all arguments.
> - Requires 4 characters minimum to access an implicit variable.
> - We should avoid adding syntax that requires allocating an array or any other object, as that is bad for performance.
> - Any approach that used a single variable that was not a true object would be problematic.
>
> Note that my proposal would support @[0], but it would mean call the [] method on the first argument to the block with the value 0 (i.e. @[0] means @.[](0)). Example:
>
> ```
> proc{@[0]}.call({0=>1})
> # => 1
> ```

Crystal clear. Thanks for the explanation once more.

#95 - 04/19/2019 10:28 AM - maedi (Maedi Prichard)

jeremyevans says:

> we should limit support to a single argument, and use @ to represent the argument.
>
> blocks that accept a single argument are much more popular than blocks that accept more than one argument. For blocks that accept multiple arguments, referencing the arguments by position rather than by name will make the code less understandable, not more.
>
> the references look like instance variables... Using a bare @ should avoid or at least mitigate that problem

Well said. Completely agree!

#96 - 04/22/2019 11:09 PM - chocolateboy (Chocolate Boy)

TL; DR: a keyword (e.g. it) could be used without breaking backwards-compatibility via a **pragma** e.g.:

```
# implicit_parameter: true

http.get(url).then { JSON.parse(it) }
```

I don't mind @1, @2 etc., given the constraints, and wouldn't mind Clojure's %1, %2 etc. either if that were possible, but I have always preferred Groovy's (and now Kotlin's) it. Although it's not as flexible, it feels like "if you need more arguments, name them" is a reasonable tradeoff/affordance.

it has been proposed and ruled out on the grounds that it could break or conflict with existing code. This then leads to the search for other (syntactical) solutions, since backwards compatibility is required in this case, which inevitably leads to aesthetic concerns and compromises.

However, it *is* possible to introduce it *without breaking any existing code* by using a pragma i.e. by signalling to the parser/compiler that a new syntax/semantic is in effect in the current scope. Perl has safely introduced features in this way for years, and JavaScript has the same facility, although, for historical reasons, it doesn't use it much.

Of course, Ruby already supports pragmas by overloading comments e.g.:

```
# frozen_string_literal: true
```

That could be used here as well e.g.:

```
# implicit_parameter: true

http.get(url).then { JSON.parse(it) }
```

- but I personally find magic comments hacky/ugly and would prefer pragmas to be available in a cleaner, more explicit, and possibly (down the line)

more extensible way. One option could be to overload using e.g.:

```
using 'it'

http.get(url).then { JSON.parse(it) }
```

By restricting this usage to the top-level and requiring a string literal (or a fixed set of bareword feature names), this could be detected at parse/compile time in the same way as a magic comment. Calling Module#using with a string currently raises a runtime error, so it wouldn't conflict with working code, though it might (in theory) require a few tests that are expected to fail to be updated (i.e. an impact similar to changing the wording of an internal error message).

Alternatively, a new syntax could be used e.g.:

```
@[ImplicitParameter]

http.get(url).then { JSON.parse(it) }
```

Either way, pragmas are already supported by Ruby and IMO this is a natural use case for them which delivers the best of both worlds: a clean and familiar syntax without breaking existing code.

#97 - 04/24/2019 02:57 AM - sawa (Tsuyoshi Sawada)

chocolateboy (Chocolate Boy) wrote:

> [P]ragmas by overloading comments ... could be used here as well e.g.:
>
> ```
> # implicit_parameter: true
>
> http.get(url).then { JSON.parse(it) }
> ```

This looks to me like splitting Ruby into two dialects and letting the programmer declare when using the non-default one. I think it would be a start of a nightmare in which Ruby has many dialects.

> One option could be to overload using e.g.:
>
> ```
> using 'it'
>
> http.get(url).then { JSON.parse(it) }
> ```
>
> ... Calling Module#using with a string currently raises a runtime error, so it wouldn't conflict with working code

using is already used for refinements, and here were are dealing with block arguments, which are unrelated. It would be confusing to use the same keyword with different meanings.

> Alternatively, a new syntax could be used e.g.:
>
> ```
> @[ImplicitParameter]
>
> http.get(url).then { JSON.parse(it) }
> ```

Pragma is ugly and we want to get rid of it. I think that adding even another form of pragma would make the situation even worse.

#98 - 04/24/2019 05:49 AM - jeremyevans0 (Jeremy Evans)

- File implicit-param.diff added

jeremyevans0 (Jeremy Evans) wrote:

> As I expressed in the developer meeting today, after a lot of thought, I believe if we want to support implicit block arguments, we should limit support to a single argument, and use @ to represent the argument.

Attached is a patch that implements my proposal:

```
proc{@}.call(1)
# => 1

proc{@}.call([1, 2])
# => [1, 2]

proc{@}.call(1, 2)
# => 1
```

```
proc{@.abs}.call(-1)
# => 1

proc{Math.log(@)}.call(Math::E)
# => 1.0

@
# SyntaxError: implicit parameter outside block
```

This patch is also available as a branch on GitHub: https://github.com/jeremyevans/ruby/tree/implicit-param

#99 - 04/25/2019 03:48 PM - headius (Charles Nutter)

I'm +1 for single variable using @.

#100 - 04/26/2019 07:54 PM - chocolateboy (Chocolate Boy)

sawa (Tsuyoshi Sawada) wrote:

> This looks to me like splitting Ruby into two dialects and letting the programmer declare when using the non-default one. I think it would be a start of a nightmare in which Ruby has many dialects.

it doesn't require new syntax, so I don't see how it can be considered a dialect any more than "use strict" in JavaScript (which similarly changes the semantics of an existing keyword (this), amongst other things). Even if it did, it's had no such effect in Perl in the 13 years since it was introduced.

> using is already used for refinements, and here were are dealing with block arguments, which are unrelated. It would be confusing to use the same keyword with different meanings.

The behavior/feature is different but the mechanism is the same i.e. it instructs the parser/compiler/interpreter to make a scoped change in semantics that can't otherwise be implemented in userspace.

> Pragma is ugly and we want to get rid of it. I think that adding even another form of pragma would make the situation even worse.

I agree that magic comments are hacky and ugly and would be happy to see them phased out. But I don't see an issue with implementing compiler directives in a non-hacky way. While a keyword like use (Perl, JavaScript, Groovy) or using might be cleaner or more familiar, a dedicated syntax is customary and arguably clearer (as well as safer/more compatible), and the suggested syntax (which is used by Crystal) is consistent with annotations and directives in other languages such as C#, Java, JavaScript, Kotlin, and Rust.

As things currently stand, the only alternatives to opting in (via a pragma) to Groovy/Kotlin's simple/standard it are a) introducing it globally, which could break some existing code or b) adding more controversial/compromised syntax which no-one is entirely happy with.

#101 - 04/26/2019 09:12 PM - Eregon (Benoit Daloze)

I think the new proposition by jeremyevans0 (Jeremy Evans) is much better than the current state, and sounds good to me.
@ might not be ideal, but at least it's less confusing than @ and doesn't have complicated compatibility problems.

From my poll on Twitter (721 votes), https://twitter.com/eregontp/status/1116775815461711872, here are the results:
Should numbered parameters be:
18% Kept, I like them
21% Changed, @ is for @ivars
5% Simplified, just 1-arg @
56% Removed, hurt readability

I think the "Kept" would be mostly fine with this change, the "Changed" would see more clearly the distinction between @ and @ivars, the Simplified would obviously agree, and I'd guess some of the Removed would think this simplification helps readability significantly.
So in summary, I think a majority of Rubyists can agree on the new proposal (@), while a majority of voters disagreed on the current status (@1, @2, etc).

#102 - 04/27/2019 03:13 AM - nobu (Nobuyoshi Nakada)

chocolateboy (Chocolate Boy) wrote:

> > using is already used for refinements, and here were are dealing with block arguments, which are unrelated. It would be confusing to use the same keyword with different meanings.
>
> The behavior/feature is different but the mechanism is the same i.e. it instructs the parser/compiler/interpreter to make a scoped change in semantics that can't otherwise be implemented in userspace.

using is a method but not a syntax, so it cannot instruct the parser/compiler.

Changing it into a syntax breaks the compatibility of course.

#103 - 04/30/2019 10:34 AM - sawa (Tsuyoshi Sawada)

If we are interested only in the entire single block parameter, then this https://bugs.ruby-lang.org/issues/10394 might be a possibility. We would then be able to refer to the parameter as self without modfying the syntax.

Files

implicit-param.diff	20 KB	04/24/2019	jeremyevans0 (Jeremy Evans)