Ruby master - Feature #15236
add support for hash shorthand
10/19/2018 10:33 AM - ignatiusreza (Ignatius Reza Lesmana)

Status: Rejected
Priority: Normal
Assignee:
Target version:

Description
PR in github: https://github.com/ruby/ruby/pull/1990

inspired by javascript support for object literal shorthand notation { a }, which will be expanded into { a: a }..

to avoid ambiguity, this shorthand is only supported when hash is defined with { } notation.. in other situation where the brackets is optional, e.g. function call, we still need to write it in full (m(a : a) instead of m(a), or m(a, b, c: c) instead of m(a, b, c)..

Related issues:
Related to Ruby master - Feature #15286: Proposal: Add Kernel.#expand(*args) Rejected
Is duplicate of Ruby master - Feature #11105: ES6-like hash literals Rejected

History
#1 - 10/19/2018 02:04 PM - nobu (Nobuyoshi Nakada)
- Is duplicate of Feature #11105: ES6-like hash literals added

#2 - 10/19/2018 02:13 PM - shevegen (Robert A. Heiler)
Hmm. It's hard for me to say whether I am in favour of this suggestion or whether I am not.

I think this link may help a bit in regards to JavaScript, even though JavaScript is not Ruby; neither is the syntax:  

Old JavaScript variant:

```javascript
var o = {};
var o = {a: 'foo', b: 42, c: {}};
```

versus New JavaScript variant:

```javascript
var a = 'foo', b = 42, c = {};
var o = {a, b, c};
```

I understand the second part being more convenient and more concise. I am not really sure whether it makes sense for ruby to adopt this, though.

The part where "omission means something more", is sometimes confusing.

I myself got used to be able to omit {} in a method definition, such as your example:

```ruby
m(a, b, c: c)
```

which I think would be this:

```ruby
m(a, b, { c: c })
```

I also use the somewhat new Hash syntax in ruby a lot, like:

```ruby
foo: :bar
```

versus the old variant (but still the "real" variant)

```ruby
:foo => :bar
```
I am not entirely sure about the new omission-meaning-infinity in ranges (1..) either, or \{ a \} meaning \{ a: a \} like in the proposal here, where a is a variable that must exist already, if \{ a \} is to work. This also reminds me a bit about the shortcut suggestion for initializing instance variables within the method-argument, rather than the body of the method at hand (usually "def initialize").

I don't really have a definite pro or con view but I think it should be thought through for some time either way. While experienced ruby developers have it easy learning new syntax parts, newcomers may have to gradually learn more and more syntax parts, which may not be ideal for them, even if the new syntax may be shorter. Or where the syntax allows us to do more with {}, rather than with Hash.new - that should also be considered to evaluate all trade-offs and advantages/disadvantages.

If you would like to, you could add your suggestion to any upcoming developer meeting where you could get some opinions from the ruby core team and of course matz (which would be at https://bugs.ruby-lang.org/issues/15229 for the next one in November 2018; or perhaps for a later one in January 2019 since I assume most energy of the team may go into the upcoming x-mas release of ruby :) ).

#3 - 10/19/2018 02:13 PM - shevegen (Robert A. Heiler)
Ah, Nobu found it and was faster. :)

So it was indeed a duplicate.

#4 - 10/19/2018 02:16 PM - shevegen (Robert A. Heiler)
Matz wrote in the other thread the following:

"I am not positive about this syntax mostly because it appears to be set syntax, or old style hash in 1.8. Once ES6 syntax become more popular, there will be chance for this change in the future.

Matz."

So I guess it could be discussed at another developer meeting in the future.

#5 - 10/19/2018 04:23 PM - Ksec (E C)
shevegen (Robert A. Heiler) wrote:

Matz wrote in the other thread the following:

"I am not positive about this syntax mostly because it appears to be set syntax, or old style hash in 1.8. Once ES6 syntax become more popular, there will be chance for this change in the future.

Matz."

So I guess it could be discussed at another developer meeting in the future.

But that comment was made three years ago. Time to do another review?

#6 - 10/20/2018 02:23 AM - matz (Yukihiro Matsumoto)
As a conservative old timer who does not use JavaScript at all, I still feel negative. It seems to work best with destructuring (left-hand side of assignments) which is nearly impossible in current Ruby syntax.

But at the same time, I admit many of Ruby users use Rails and JavaScript, so I am open to hearing your opinion.

Matz.

#7 - 10/22/2018 02:07 AM - ignatiusreza (Ignatius Reza Lesmana)
Hi guys,

Thanks for the discussions! Sorry I didn't noticed that it was proposed (multiple times) before. I tried to search, but couldn't find a hit.

The ES6 syntax that this gets inspired from is strongly becoming the standard now, partly thanks to it being enabled by default in https://www.npmjs.com/package/eslint-config-airbnb-base
I found a strong desire for this syntax especially when working on API server alongside JavaScript heavy front end, where one would need to work a lot with building hashes to be transformed into JSON string... hence, the primary use case where I'm interested in is in building hashes as return value of method call, e.g.

```ruby
def respond_with(resource, options)
  meta = extract_meta(resource, options)
  etc = extract_etc(resource, options)
  { resource, meta, etc }
end
```

having

```ruby
{ resource, meta, etc }
```

is much more concise and cleaner compared to

```ruby
{ resource: resource, meta: meta, etc: etc }
```

within this context, `{ }` is already non-optional, and the new syntax increase readability and save a lot of typing..

To address the concern in https://bugs.ruby-lang.org/issues/11105... I think, I agree that this shorthand syntax should only be allowed for `a`, but not for `@a`, `@@a`, or `$a` to avoid ambiguity in what key should be generated for everything else other than `a`...

It seems to work best with destructuring (left-hand side of assignments)

I agree that this syntax `{ a, b, c }` and destructuring `{ a, b, c } = rhs does go hand in hand... though, the former can be supported just by updating parser, while the later need update in parser and some run time check, since there's no guarantee during compile time as to what rhs will return... I'll be interested in exploring on ways to make destructuring works if this request gets accepted..

#8 - 10/22/2018 08:54 AM - janfri (Jan Friedrich)

matz (Yukihiro Matsumoto) wrote:

As a conservative old timer who does not use JavaScript at all, I still feel negative. It seems to work best with destructuring (left-hand side of assignments) which is nearly impossible in current Ruby syntax.

But at the same time, I admit many of Ruby users use Rails and JavaScript, so I am open to hearing your opinion.

Matz.

I also don't use JavaScript. I think the use of destructuring for this is much more Rubyish than the ES6 syntax.

#9 - 10/30/2018 12:47 AM - ignatiusreza (Ignatius Reza Lesmana)

janfri (Jan Friedrich) wrote:

I think the use of destructuring for this is much more Rubyish than the ES6 syntax.

Agreed that destructuring hash is very much ruby, since destructuring array is already supported... but, I think supporting that does not mean that we can't support this one too... in fact, I think supporting both would be best, since I would expect that if I can do one, I should also be able to do the other..

#10 - 11/02/2018 08:29 PM - blakewest (Blake West)

Hi all,

I've really wanted a feature like this for a long time. I find myself often using named arguments, which I love for the clarity. But then the callers often have variables of the same name as the parameter name, making it more verbose. eg. foo(bar: bar, baz: baz). If one concern here is that `{a, b, c}` looks like set syntax, then I'd be curious to get opinions on a different syntax I had thought about for a while. The general idea is to extend the `%w` idea for hashes. For example, we could do `%h{foo bar baz}` which would expand into `{foo: foo, bar: bar, baz: baz}`. In both the `%w` and `%h` cases, we're allowing for a more concise version of an often used pattern.

I think the downside is using a `%h` syntax feels "separated" from the typical hash syntax. Using a straight `{a,b,c}` is "cleaner" as there are fewer characters, and may be more intuitive to some.

But similarly, I think that actually the upside of this syntax is it's more "separated". As Matz points out, the `{a,b,c}` syntax could look like set syntax, and as shevegen points out, it could be confusing for beginners. Neither of these problems would exist using `%h`, as it would pretty clearly be a "special" syntax, used by people who know what it's doing. Also, using `%h[]` would enable not having to use commas, which could make it even more compact.

I'll leave with a few more examples of what I'd be suggesting.

```ruby
def foo(param1:, param2:)
  param + param2
```
#11 - 11/02/2018 08:47 PM - osyo (manga osyo)

hi, blakewest.
I proposaled it.
https://bugs.ruby-lang.org/issues/14973
I need to consider an implementation that does not use #eval.

#12 - 11/02/2018 08:55 PM - osyo (manga osyo)

I would like to use Hash shorthand in the following cases.

describe User do
  let(:id) { ... }
  let(:name) { ... }
  let(:age) { ... }
  let(:registered_at) { ... }
  # I want to hash shorthand!
  let(:user) { User.new(id: id, age: age, name: name, registered_at: registered_at) } 
  context刳name is nil сохрани
  let(:name) { nil } 
  it { ... }
  end
  context觚age is nil сохрани
  let(:age) { nil } 
  it { ... }
  end
end

#13 - 11/06/2018 01:21 AM - ignatiusreza (Ignatius Reza Lesmana)

blakewest (Blake West) wrote:

Hi all,
I've really wanted a feature like this for a long time. I find myself often using named arguments, which I love for the clarity. But then the callers
often have variables of the same name as the parameter name, making it more verbose. eg. foo(bar: bar, baz: baz). If one concern here is that
{a, b, c} looks like set syntax, then I'd be curious to get opinions on a different syntax I had thought about for a while. The general idea is to
extend the %w idea for hashes. For example, we could do %h{foo bar baz} which would expand into {foo: foo, bar: bar, baz: baz}. In both the
"%w" and "%h" cases, we're allowing for a more concise version of an often used pattern.
I think the downside is using a %h syntax feels "separated" from the typical hash syntax. Using a straight {a,b,c} is "cleaner" as there are fewer
characters, and may be more intuitive to some.
But similarly, I think that actually the upside of this syntax is it's more "separated". As Matz points out, the {a,b,c} syntax could look like set
syntax, and as shevegen points out, it could be confusing for beginners. Neither of these problems would exist using %h, as it would pretty
clearly be a "special" syntax, used by people who know what it's doing. Also, using %h[] would enable not having to use commas, which could
make it even more compact

I'll leave with a few more examples of what I'd be suggesting.

def foo(param1, param2)
  param1 + param2
end

param1 = 7
param2 = 42
foo(%h{param1 param2})
def respond_with(resource, options)
  meta = extract_meta(resource, options)
  etc = extract_etc(resource, options)

  %h{ resource meta etc }
end

# destructuring could obviously be left for later.
%h{data meta etc} = {data: [1,2,3], meta: {mobile: true}, etc: "more info"}

Thoughts?

osyo (manga osyo) wrote:

  hi, blakewest.
  I proposaled it.
  https://bugs.ruby-lang.org/issues/14973
  I need to consider an implementation that does not use #eval.

I like this :+1:

  [ a, b, c ] looks a bit cleaner to me, probably because of my familiarity with es6 syntax, so it require less context switching.. and it is more versatile, since it allow us to mix and match with other way hash key can be defined, e.g. [ a, b: 2, :c => 3 ].. but if it is considered confusing to some, %h[a b c] also works for me..

#14 - 11/22/2018 10:31 PM - matz (Yukihiro Matsumoto)
- Related to Feature #15286: Proposal: Add Kernel.#expand(*args) added

#15 - 04/29/2019 05:56 PM - tleish (Tony Fenleish)
+1

I also vote for this option for both hashes and named params. After using Es6 and going back to Ruby, the ruby way felt clunky to me.

There are multiple reasons why this would be advantageous.

Readability:

I like the named params that was added to Ruby. It allows flexibility in the order of the params, but often struggle in convincing myself to use them because of readability. Code examples often show the following:

  my_hash = {name: 'joe', age: 8}

  # or

  my_method(name: 'Joe', age: 8)

But in the real world, the values are usually stored in other variables and 95% of the time the code looks like this:

  my_hash = {name: name, age: age}

  # or

  my_method(name: name, age: age)

The above code does not ready well with it's redundant words. With the change suggested, it reads so much better:

  my_hash = {name, age}

  # or

  my_method(name, age)

If the variable is different than the param name, then it easily communicates that:

  my_hash = {name, age: calculate_age}

  # or

  my_method(name, age: calculate_age)
Refactoring

If a method is implemented without keyword arguments:

```ruby
def my_method(name, age); end
my_method(name, age)
```

And then later it is decided to refactor using Ruby keywords

```ruby
def my_method(name:, age:, address: nil); end
```

This of course is a breaking change. All locations which use this method now need to be updated to use the new convention.

```ruby
my_method(name: name, age: age)
```

However, for instances where the method was implemented using variables of the same as the parameters (which often they are), then the change is non-breaking.

```ruby
my_method(name, age) # no change required
```

#16 - 06/09/2019 03:11 AM - ignatiusreza (Ignatius Reza Lesmana)
I was reading the presentation slide for the experimental feature for syntax matching.. in which, there's support for "shorthand" matching with hash, in the form of { a: }.. or, following the examples presented above:

```ruby
def foo(param1:, param2:)
  param1 + param2
end
param1 = 7
param2 = 42
foo(param1:, param2:)
def respond_with(resource, options)
  meta = extract_meta(resource, options)
  etc = extract_etc(resource, options)
  { resource:, meta:, etc: }
end
# destructuring
{data, meta:, etc:} = {data: [1,2,3], meta: {mobile: true}, etc: "more info"}
```

I think, this answers the concern mentioned above regarding { a } being confused with a Set instead of Hash.. wydt?


#17 - 06/09/2019 04:22 AM - ignatiusreza (Ignatius Reza Lesmana)
PR for alternative syntax { a: }: [https://github.com/ruby/ruby/pull/2231](https://github.com/ruby/ruby/pull/2231)

#18 - 07/29/2019 08:01 AM - ko1 (Koichi Sasada)
- Status changed from Open to Rejected

Please reopen (re-create) new proposal which can persuade Matz.

#19 - 08/11/2019 03:12 PM - D1mon (Dim F)
Well, if the syntax is misleading or intersects with an existing design. Make another syntax to work, why cancel (reject) a very cool thing which will make the code smaller (write less) and it will be nice to read the code. ? Please do not reject the request, redo it and add it to the "ruby core". People have been asking for this opportunity for several years now, please do not refuse them. Thank.

#20 - 02/12/2020 01:14 AM - joallard (Jonathan Allard)
I support this proposal in its simplest form. I do not think changing method argument syntax is a good idea.

osyo (manga osyo) above makes a good point, if your variables are well named:

```ruby
User.new(id: id, age: age, name: name, registered_at: registered_at)
```

It is rewarding to have a shorter, clear syntax:
User.new({id, age, name, registered_at})

I do not see a reason to reject this. I would strongly encourage re-opening.

#21 - 02/13/2020 09:06 AM - nobu (Nobuyoshi Nakada)
jonald (Jonathan Allard) wrote in #note-20:

    User.new({id, age, name, registered_at})

Note that you will have to put ** between the parenthesis and brace now.

User.new(**{id, age, name, registered_at})

#22 - 09/02/2020 02:18 PM - palkan (Vladimir Dementyev)

For all interested in this feature, I’ve added support for it in the latest Ruby Next (Ruby transpiler) release (https://github.com/ruby-next/ruby-next/releases/tag/v0.10.0), so it should be easier to give it a try and share opinions.