

# Ruby master - Feature #16986

## Anonymous Struct literal

06/26/2020 06:58 AM - ko1 (Koichi Sasada)

<b>Status:</b>	Open
<b>Priority:</b>	Normal
<b>Assignee:</b>	matz (Yukihiro Matsumoto)
<b>Target version:</b>	

### Description

## Abstract

How about introducing anonymous Struct literal such as `#{a: 1, b: 2}`?  
It is almost the same as `Struct.new(:a, :b).new(1, 2)`.

## Proposal

### Background

In many cases, people use hash objects to represent a set of values such as `person = {name: "ko1", country: 'Japan'}` and access its values through `person[:name]` and so on. It is not easy to write (three characters `[:!]`), and it easily introduces misspelling (`person[:nama]` doesn't raise an error).

If we make a Struct object by doing `Person = Struct.new(:name, :age)` and `person = Person.new('ko1', 'Japan')`, we can access its values through `person.name` naturally. However, it costs coding. And in some cases, we don't want to name the class (such as `Person`).

Using `OpenStruct` (`person = OpenStruct.new(name: "ko1", country: "Japan")`), we can access it through `person.name`, but we can extend the fields unintentionally, and the performance is not good.

Of course, we can define a class `Person` with `attr_readers`. But it takes several lines.

To summarize the needs:

- Easy to write
  - Doesn't require declaring the class
  - Accessible through `person.name` format
- Limited fields
- Better performance

### Idea

Introduce new literal syntax for an anonymous Struct such as: `#{ a: 1, b: 2 }`.  
Similar to Hash syntax (with labels), but with `$` prefix to distinguish.

Anonymous structs which have the same member in the same order share their class.

```
s1 = #{a: 1, b: 2, c: 3}
s2 = #{a: 1, b: 2, c: 3}
assert s1 == s2
```

```
s3 = #{a: 1, c: 3, b: 2}
s4 = #{d: 4}
```

```
assert_equal false, s1 == s3
assert_equal false, s1 == s4
```

### Note

Unlike Hash literal syntax, this proposal only allows `label: expr` notation. No `#{**h}` syntax.  
This is because if we allow to splat a Hash, it can be a vulnerability by splatting outer-input Hash.

Thanks to this spec, we can specify anonymous Struct classes at compile time. We don't need to find or create Struct classes at runtime.

## Implementation

<https://github.com/ruby/ruby/pull/3259>

## Discussion

### Notation

Matz said he thought about `{a: 1, b: 2}` syntax.

### Performance

Surprisingly, Hash is fast and Struct is slow.

```
Benchmark.driver do |r|
  r.prelude <<~PRELUDE
  st = Struct.new(:a, :b).new(1, 2)
  hs = {a: 1, b: 2}
  class C
    attr_reader :a, :b
    def initialize() = (@a = 1; @b = 2)
  end
  ob = C.new
  PRELUDE
  r.report "ob.a"
  r.report "hs[:a]"
  r.report "st.a"
end
__END__
Warming up -----
           ob.a   38.100M i/s -    38.142M times in 1.001101s (26.25ns/i, 76clocks/i)
           hs[:a] 37.845M i/s -    38.037M times in 1.005051s (26.42ns/i, 76clocks/i)
           st.a   33.348M i/s -    33.612M times in 1.007904s (29.99ns/i, 87clocks/i)
Calculating -----
           ob.a   87.917M i/s -   114.300M times in 1.300085s (11.37ns/i, 33clocks/i)
           hs[:a] 85.504M i/s -   113.536M times in 1.327850s (11.70ns/i, 33clocks/i)
           st.a   61.337M i/s -   100.045M times in 1.631064s (16.30ns/i, 47clocks/i)
Comparison:
           ob.a:  87917391.4 i/s
           hs[:a]: 85503703.6 i/s - 1.03x slower
           st.a:  61337463.3 i/s - 1.43x slower
```

I believe we can speed up Struct similarly to ivar accesses, so we can improve the performance.

BTW, OpenStruct (os.a) is slow.

```
Comparison:
           hs[:a]: 92835317.7 i/s
           ob.a:  85865849.5 i/s - 1.08x slower
           st.a:  53480417.5 i/s - 1.74x slower
           os.a: 12541267.7 i/s - 7.40x slower
```

For memory consumption, Struct is more lightweight because we don't need to keep the key names.

### Naming

If we name an anonymous class, literals with the same members share the name.

```
s1 = ${a:1}
s2 = ${a:2}
p [s1, s2] #=> [#<struct a=1>, #<struct a=2>]
```

```
A = s1.class
p [s1, s2] #=> [#<struct A a=1>, #<struct A a=2>]
```

Maybe that is not a good behavior.

## History

### #1 - 06/26/2020 07:49 AM - ko1 (Koichi Sasada)

- Description updated

### #2 - 06/26/2020 09:36 AM - shevegen (Robert A. Heiler)

First, I like the idea, so +1 for the idea. It also reminds me of a more "prototypic-based approach" in general with structs, so if the syntax could be made simpler to type, that seems to be a possible improvement. And it fits into other "shortcut" variants, such as `%w()` `%l()` and so forth. I make use of `%w()` and so forth a LOT - it really helps when you have to create longer arrays; we can avoid all the middle `","` parts.

What I dislike a bit about the suggestion here is the proposed syntax. There are two aspects as to why that is the case for me:

(1) While I do sometimes use `$` variables (in particular the regex variants `$1` `$2` are so convenient to use), sometimes I use `$stdin` as well, but I don't quite like global variables in general. In particular remembering `$_` `$_?` `$__` (if these even exist) is a bit tedious. So most of the time when we can avoid `$`, I think this is better than having to use `$`. But this is just one part.

(2) The other, perhaps slightly more problematic thing, is that we would introduce a new variant of how people have to understand `$` variables.

Specifically this variant:

```
#{a: 1, b: 2}
```

May look like regular variable substitution such as:

```
cat = 'Tom'
puts "#{cat} is hunting Jerry."
```

Or, if people use global variables:

```
$cat = 'Tom'
puts "#{$cat} is hunting Jerry."
puts "#$cat is hunting Jerry." # ok perhaps not quite as much, since we can omit {} in that case.
```

Possibly the `{}` may have ruby user associate this with a regular substitution.

The other aspect is that this would be the first global variable use that combines `{}` upon "definition" time. We used to have something like this:

```
$abc = 'def'
```

That feels a bit different to:

```
abc = #{a: 1, b: 2}
```

Hmm.

Matz said he thought about `{a: 1, b: 2 }` syntax.

Do you mean without `'$'` there? If so then I think the syntax by matz is better. Though people could assume it is a block. :)

However had, having said that, I do not have a good alternative suggestion; and even then I think the feature may quite useful.

Perhaps:

```
struct a: 1, b: 2
```

Would be somewhat short too.

Implying:

```
struct {a: 1, b: 2}
```

Though, admittedly the struct variant is still longer to type than the \$ suggestion here. While I don't quite like the \$, it is arguably short to type. And I suppose one requirement for this proposal is that it should be shorter than:

```
Struct.new(:a, :b).new(1, 2)
```

Even if made shorter, like:

```
Struct.new(a: 1, b: 2)
```

Admittedly the \$ is shortest:

```
#{a: 1, b: 2}
```

matz' variant:

```
foobar = {|a: 1, b: 2 |}  
foobar = {|a: 1, b: 2|}  
foobar = ${|a: 1, b: 2 |}
```

(I was not sure which variant was the one; I assume matz' variant is actually without the \$, so perhaps the second variant.)

If we name the anonymous class, the same member literals share the name.

```
s1 = ${a:1}  
s2 = ${a:2}
```

Hmmmmm. The matz' variant would then be like this?

```
s1 = {|a: 1|}
```

To be honest, even if that is longer, I think it is better than the variant with \$. I am not sure if I am the only one disliking the \$ there but I think it would be better to not have to use \$, even if the second variant is longer. But as said, I think the idea itself is fine.

### #3 - 06/26/2020 10:09 AM - osyo (manga osyo)

hi, I like this idea :)

I was wondering if \${} can do the following things compared to Hash literals:

```
# Can a value that is not a Symbol be used as a key (Symbol only?)
```

```
#{ "a" => 1 }  
#{ 1 => 1 }
```

```
# Can variables be used as keys
```

```
key = :a  
#{ key => 1 }
```

```
# Can Hash be expanded with `**`
```

```
hash = { a: 1, b: 2 }  
#{ c: 3, **hash }
```

Thank you.

### #4 - 06/26/2020 11:05 AM - zverok (Victor Shepelev)

WDYT about half-solution (without syntax change)?

E.g. for me, the problem with Struct.new(:a, :b).new(1, 2) is not that it is "too long to write" but just that it looks "hacky" (like, "you are using Struct against its expectations/best practices"), and non-atomic.

So, may be this would be enough for most cases:

```
Struct.anonymous(a: 1, b: 2)
# method name is debatable.
# or, IDK, maybe just
Struct(a: 1, b: 2)
```

Also, I'd say that maybe the value produced this way should be immutable? (as in [#16122](#))

Otherwise, one might just use `OpenStruct` (if the value has the same amount of mutability as `hash`), or normal `Type = Struct.new(:a, :b)` (if the structure of value is fixed, but content is mutable — it means structure of type has some fixed semantic and probably should have a name).

#### #5 - 06/26/2020 11:08 AM - zverok (Victor Shepelev)

Another (unrelated, but conflicting) matter: I am not sure we have/had a discussion for this, but I remember Bozhidar Batsov in his "Ruby 4: To Infinity and Beyond" proposed `${}` as a literal for *set* (which for me seems more important and potentially more widespread).

#### #6 - 06/26/2020 11:14 AM - byroot (Jean Boussier)

Matz said he thought about `{a: 1, b: 2}` syntax.

Might be just me but `{}` on first sight makes me think: this is a block.

What about a `%` syntax? `%s` and `%S` already exists, but `%O` could make sense: `%O{foo: 1, bar: 42}`. Or yeah just a simple `Struct()` or `Struct[]` with no additional syntax as suggested just above.

#### #7 - 06/26/2020 11:15 AM - byroot (Jean Boussier)

Or yeah just a simple `Struct()` or `Struct[]` with no additional syntax as suggested just above.

Actually I just realized that it can't really work as it would allow for `**kwargs`.

#### #8 - 06/26/2020 04:22 PM - zverok (Victor Shepelev)

Thinking a bit more about it, I see more of a conceptual problem.

Introducing new literal in a language, we kinda introduce a new concept of the core collection. But what's that collection is, how'd you explain it to a novice?

It is:

- "a Struct", but it can't be created with `Struct.new` (which, already confusingly enough, creates something that not `is_a?(Struct)`)
- which has keys/values like `Hash`,
- ...but allows access by `.<key>...` oh, but also `[:key]` and `["key"]`. And `[0]` :)
- ...and does not allow adding/removing keys
- ...but is not fully immutable, as it allows assigning new values to keys
- ...also it, for example, unpacks `(*${a: 1})` in just values? (at least structs currently behave so)

So, what core concept it represents?..

#### #9 - 06/27/2020 12:37 PM - ttnimichi (Tsukuru Tanimichi)

`struct = (a: 1, b: 2)` would be a great syntax, because I think anonymous struct is something like Tuple of Python.

But, there are two problems:

1. `()` returns `nil`, not an empty struct.
2. The meaning of `p(a: 1)` and `p (a: 1)` are different. The former prints `{:a=>1}`, but the latter prints `#<struct a=1>`

#### #10 - 06/27/2020 02:28 PM - bkuhlmann (Brooke Kuhlmann)

I like the idea of being able to quickly create an anonymous `Struct` but am concerned about the use of `$` for the syntax since that hinders readability and causes confusion due to `$` generally denoting a *global variable*. Why not allow structs to have similar behavior to existing objects in Ruby like hashes, arrays, etc in order to remain *consistent*? Example:

```
# Kernel.Hash
Hash a: 1, b: 2 # => {a: 1, b: 2}
# Hash.[]
Hash[a: 1, b: 2] # => {a: 1, b: 2}

# Kernel.Array
Array 1 # => [1]
# Array.[]
Array[1] # => [1]
```

With the suggestion above, we could implement the same for Structs too. Example:

```
# Kernel.Struct
Struct a: 1, b: 2 # => #<struct a=1, b=2>
# Struct.[]
Struct[a: 1, b: 2] # => #<struct a=1, b=2>
```

#11 - 06/27/2020 05:55 PM - ko1 (Koichi Sasada)

## Q&A

### Splat like Hash literal and method arguments

<https://bugs.ruby-lang.org/issues/16986#note-3>

I was wondering if `#{}` can do the following things compared to Hash literals:

Not allowed because of vulnerability concerns (please read a ticket for more details).

### Syntax

- (1) `#{a:1, b:2}` # original
- (2) `{a:1, b:2}` # matz's idea ... conflict with block parameters.
- (3) `struct a: 1, b: 2 # #1` ... introducing new struct keyword can introduce incompatibility.
- (4) `%o{a:1, b:2} # #6`
- (5) `(a:1, b:2) # #9`
- (6) Methods
  - (6-1) `Struct.anonymous(a:1, b:2) # #4`
  - (6-2) `Struct(a:1, b:2) # #4, #10`
  - (6-3) `Struct[a:1, b:2] # #10`

Some support comments on `#{...}`:

- I can recognize it is not global variables.
- `$` is seems as an initial letter of Struct ... S !!! (50% joking)
- If we can introduce `#{ ... }`, we can also consider about `#{...}` (Set?) and `$(...)`. I agree it can introduce further chaotic.
- We can replace `$` with `@`, but no reason to choose `@` ... ah, not a support comment.

I thought similar idea on "(4) `%o{a:1, b:2} # #6`", and my idea was `%t (S**struct)`.

However, there is no `%` notation which allows Ruby's expression in it.

In other words, existing `%` notation defines different language in `%` (`%w`, `%i` for example).

This is why I gave up this idea. But I don't against it (new language can accept Ruby's expression).

For "(6) Methods", my first proposal was `Struct(a;1, b:2)`. [https://twitter.com/\\_ko1/status/1276055259241046016?s=20](https://twitter.com/_ko1/status/1276055259241046016?s=20)

However, there are several advantages by introducing new syntax which are described in a ticket.

And real reason I make a demonstrate moving code <https://github.com/ruby/ruby/pull/3259> is I want to modify `parse.y` to escape from Ractor's debugging.

The biggest advantage of choosing a method approach is simplicity. No new syntax and only a few learning cost.

It is easy to introduce same method for older versions (~2.7).

(For performance of object creation, we can introduce specialization for `Struct()` method to prepare an anonymous class at compile time)

"(5) `(a:1, b:2)`" seems interesting, but I agree there are issues which are pointed in the comment 9.

#12 - 06/27/2020 06:08 PM - ko1 (Koichi Sasada)

Other syntax ideas, by others:

- Other prefixes
  - `::{a: 1}`
  - `\{a: 1}`
- `<>` to indicate
  - `{<> a:1}` for anonymous Struct.
  - `{<A> a:1}` for named Struct, the name is A.
- Similar with `%`
  - `{% a:1}` for anonymous Struct (it can conflict with `%` notation).
  - `{%A a:1}` for named Struct, the name is A.

#13 - 06/27/2020 06:31 PM - retro (Josef Šimánek)

First of all, this is super cool idea!

I do have habit to use hash since it seems to be elegant (as described in original proposal background section) and I end up having problems later (since I need to use fetch everywhere to get at least some kind of consistency and to avoid typos for example).

I think there's no need for new syntax. "Struct.new" and "Kernel.Struct()" should be enough (if possible to extend Struct.new and keep the same performance).

Regarding syntax used, it would be great to support also "nested" structs, which I'm not sure if possible for all current ideas. For example:

```
config = Struct(assets: Struct(reload: true))
config.assets.reload #=> true
```

For simple structs I think %t or %o notation would be handy as well.

As mentioned at [#10](#), by introducing Struct(), extending Struct.new and allowing %o or %t it would just follow already common patterns used for Array and Hash.

#### #14 - 06/28/2020 05:52 AM - sawa (Tsuyoshi Sawada)

- Description updated

#### #15 - 07/02/2020 08:44 AM - Hanmac (Hans Mackowiak)

i think this is more of a confusing feature

IF `#{a: 1, b: 2}` is like `Struct.new(:a, :b).new(1, 2)` then my gut is telling me that

```
s1 = #{a: 1, b: 2, c: 3}
s2 = #{a: 1, b: 2, c: 3}
assert s1 == s2
```

should not be the same because their class is different

#### #16 - 07/02/2020 11:34 AM - sobrinho (Gabriel Sobrinho)

+1 for the idea, I'm using `my_hash.fetch(:my_key)` to ensure consistency in a lot of places.

Concerns:

1. Struct is slow (should not be that hard to improve, though)
2. `#{}` will be hard to read, search and explain about
3. It would be useful to have nested support like `Struct(a: Struct(b: 1))` as mentioned before, using `#{a: #{b: 1}}` is a bit confusing to read

I'm also with the `Kernel.Struct` method, it seems the best proposal so far.

#### #17 - 07/02/2020 04:06 PM - calebhearth (Caleb Hearth)

I'm also +1 for this. The utility of such a feature is obvious and would improve a lot of Ruby code both in readability and in being more bug-free as it helps with the `Hash#[]` problem of missing/misspelled keys.

I immediately understood the `#{}` meaning, and so I would have to disagree with those who suggest it might be harder to read. `#{}` for Struct was my first thought, and `{}` makes the hash-iness of it obvious so I think this is good syntax.

Perhaps as a compromise `#{}` could be shorthand for some longform method such as we see in `->()` for `lambda {}` or `.()` for `.call`.

The longform might simply be `Struct.new().new()`, but ko1 mentioned that it was "almost the same" - what differences are there?

#### #18 - 07/02/2020 05:02 PM - byroot (Jean Boussier)

it was "almost the same" - what differences are there?

The actual implementation is more like this:

```
$structs = {}
def Struct(**kwargs)
  klass = $structs[kwargs.keys.sort] ||= Struct.new(*kwargs.keys, keyword_init: true)
  klass.new(**kwargs).freeze
end
```

Two literal structs with the same set of fields, will share the same class, that's the subtlety.

I believe this also answers [Hanmac \(Hans Mackowiak\)](#)'s interrogation.

#### #19 - 07/02/2020 06:58 PM - ko1 (Koichi Sasada)

```
klass.new(**kwargs).freeze
```

This proposal doesn't include freeze.

I agree it is one option, but need a discussion.

#### #20 - 07/02/2020 08:56 PM - marcandre (Marc-Andre Lafortune)

Without expressing an opinion on the proposal (yet), I'd like to point an easy way to avoid having to use fetch all the time: the default\_proc.

```
my_hash = Hash.new { |h, k| raise "Invalid key: #{k}" }.merge(default: 42)
```

```
my_hash[:default1] # => Invalid key: default1
```

#### #21 - 07/02/2020 11:14 PM - dimasamodurov (Dima Samodurov)

Accessing object methods via dots is more convenient than via brackets. And I tend to think of {a: 1, b: 2} as of object with properties.

That is why I like \${} syntax: I see the same object in square brackets and can easily copy/paste. Using of \$ does not confuse, as global variables are used rarely.

I would rather think of concept differences between Hash and Struct classes which would have similar literals. Hashes are extendable, using hash.merge(hash) is great. This is kind of symbols vs strings. Both are great, but using symbols was a bit less understood. Collecting more common use cases and antipatterns would help adopting the feature.

#### #22 - 07/03/2020 10:52 AM - esquinas (Enrique Esquinas)

Hello, this is a great idea. I would just love to have this feature in Ruby, so I thought I may provide some helpful input:

First, I would like to read more ideas about the exact implementation, specially:

1. Will the literal produce a frozen struct by default or not?
2. Will other objects (hashes, for instance) be allowed to be included by reference or will they be copied by value? Recursively?

I think some hint about the implementation will make the case for the usefulness and, more important, the **purpose** of the new syntax/feature and help us all argue and decide.

I have not enough experience or knowledge to debate about the implementation problems of any of the options, so I apologize in case I propose a non-sensical approach in this regard.

My comment about the \${} syntax would be that it feels very arbitrary. Others have proposed even more arbitrary notations, but if I had to choose one, I would vote for a .{} notation, the dot is an obvious reminder of how you will access the struct literal later on. Again, sorry if this syntax is non-sensical and unfeasible, just trying to come up with a short syntax which is less arbitrary.

Another notation that I liked was the proposed %o{} or %t{}. They feel very familiar and even allow the capital letter variation. The meaning of %O{} or %T{} could depend on the first point I made, the purpose behind it.

A new option I didn't see yet is to extend the Hash class to have a to\_struct method, so we avoid the new syntax altogether: { a: 1, b: 2 }.to\_struct. I wrote a quick & dirty implementation to explore some ideas around this concept in this [Gist](https://gist.github.com/esquinas/6c47046b1557a7b372466032187b152f)

**To summarize**, if I had to rank my top 5 of proposed notations:

1. { a: 1, b: 2 }.to\_struct no need for a new syntax. Very "Ruby" IMO. The Hash#to\_struct method could take arguments for the class name, the frozen state, etc.
2. Struct a: 1, b: 2 or similar, very obvious and readable.
3. %o{ a: 1, b: 2 } or %t{ a: 1, b: 2 } again, very familiar and allow the capital-letter variation.
4. .{ a: 1, b: 2 } very short syntax alternative and a little less arbitrary than \${} or \{}.
5. ::{ a: 1, b: 2 } same as the previous one, my\_struct::a just works, so why not?

Thanks!

#### #23 - 07/03/2020 01:31 PM - Hanmac (Hans Mackowiak)

the problem i have with that is that each time it creates a (cached) struct class.

after taking so long to make frozen literals and making even such frozen objects GC able, something that creates new (class) objects which then can never be cleared sounds like a problem that would bite you in the butt sooner or later

other than that, Struct a: 1, b: 2 would probably be the most clean variant



#### #24 - 07/07/2020 03:07 PM - jrochkind (jonathan rochkind)

Why is more special syntax needed, when it can just be a method?

```
def Kernel.AStruct(**key_values)
  Struct.new(key_values.keys).new(key_values.values)
end
```

```
AStruct(a: 1, b: 2)
```

If that implementation isn't efficient enough, implement it in C with a high-performance memoizing implementation or something, why not.

But additional syntax makes the language larger, harder to implement, harder to learn. It's hard to google for punctuation. At its best Ruby is just objects and methods, if we can provide this functionality just fine with a plain old method, why the need for new syntax?

#### #25 - 07/10/2020 08:16 AM - zliang (Gimi Liang)

Maybe {a = 1, b = "hello world"}?

#### #26 - 08/04/2020 12:40 AM - ko1 (Koichi Sasada)

Matz said: "good to have, but current proposed syntax are not acceptable. If there is good syntax, I can consider to accept."

#### #27 - 08/12/2020 05:56 PM - okuramasafumi (Masafumi OKURA)

I found that

```
{{a: 1, b: 2}}
```

is a syntax error and could be a good candidate for this feature.

#### #28 - 08/12/2020 07:48 PM - osyo (manga osyo)

okuramasafumi (Masafumi OKURA) wrote in [#note-27](#):

I found that

```
{{a: 1, b: 2}}
```

is a syntax error and could be a good candidate for this feature.

hi.

hoge {{a: 1, b: 2}} is not syntax error. {{a: 1, b: 2}} is block argument.

```
def hoge(&block)
  block.call
end
```

```
# OK
hoge {{a: 1, b: 2}}
# => {:a=>1, :b=>2}
```

#### #29 - 08/13/2020 05:56 AM - nobu (Nobuyoshi Nakada)

okuramasafumi (Masafumi OKURA) wrote in [#note-27](#):

I found that

```
{{a: 1, b: 2}}
```

is a syntax error and could be a good candidate for this feature.

Read the error message.

```
$ ruby -e '{{a: 1, b: 2}}'
-e:1: syntax error, unexpected '}', expecting =>
{{a: 1, b: 2}}
  ^
```

That means {{ itself is not a syntax error and conflicts with a nested hash.

#### #30 - 08/14/2020 05:17 AM - okuramasafumi (Masafumi OKURA)

hoge `{{a: 1, b: 2}}` is not syntax error. `{{a: 1, b: 2}}` is block argument.

That means `{{` itself is not a syntax error and conflicts with a nested hash.

Thank you for pointing it out. Obviously it doesn't work.

What about `?{a: 1, b: 2}`?

Question mark (?) is not be able to be a method by itself.

Character literal follows only one character, but in this form at least two characters follow.

So I think this form will not interpreted other ways but a Struct literal.

### #31 - 08/17/2020 06:09 AM - nobu (Nobuyoshi Nakada)

okuramasafumi (Masafumi OKURA) wrote in [#note-30](#):

What about `?{a: 1, b: 2}`?

Question mark (?) is not be able to be a method by itself.

Character literal follows only one character, but in this form at least two characters follow.

Do you mean a: by "two characters"?

As `{` is a punctuation, the `a` cannot be the part of that character literal.

### #32 - 08/17/2020 07:16 AM - sawa (Tsuyoshi Sawada)

nobu (Nobuyoshi Nakada) wrote in [#note-31](#):

Do you mean a: by "two characters"?

As `{` is a punctuation, the `a` cannot be the part of that character literal.

I think he meant the characters `{` and `a` (or even `:`). I think his point is that having them in succession causes a syntax error, which means that it does not conflict with existing code.

```
?{ # => "{ "  
?a # => "a "  
?{a # >> Syntax error
```

### #33 - 08/18/2020 05:05 PM - esquinas (Enrique Esquinas)

Issue [#16122](#) gave me this idea:

I we already had `Struct::Value`, then I think it would make a lot of sense to allow the following syntax:

```
%v{ a: 1, b: 2, c: 3 } v
```

 short for `Struct.Value => immutable: true, enumerable: false, hash_accessors: false`

I think the above syntax is great because `%v{ a: 1, b: 2, c: 3 }` is memorable ("v" for value) and friendly to people coming from JavaScript/Nodejs: "Same as JS, just start with %v". This will cover the use-case of having a very quickly way to create an efficient, safe, well structured, value-like object to store data, which'd be amazing.

For other variations, as some have said on Issue [#16122](#) , it may be good enough to use keyword arguments for `Struct.new()`.

However, if we wanted to add a short syntax for good old regular Structs, I propose yet another option, apart from `%o{...}` :

```
%a{ a: 1, b: 2, c: 3 } a
```

 for **A**nonymous Struct?

Sorry for the brainstorming undertone and the slight diversion.

### #34 - 08/19/2020 07:05 PM - ko1 (Koichi Sasada)

how about `%struct{a: 1, b: 2}` (and `%value{...}` if needed)?

(znz-san's idea and it seems nice)

### #35 - 08/20/2020 02:48 PM - esquinas (Enrique Esquinas)

ko1 (Koichi Sasada) wrote in [#note-34](#):

how about `%struct{a: 1, b: 2}` (and `%value{...}` if needed)?

(znz-san's idea and it seems nice)

I like it! IMO, it's one of the most "natural to read" options and is undoubtedly easier to write than `Struct.new(:a, :b, :c).new(1, 2, 3)`.

My only tiny concern is, shouldn't be `%Struct{a: 1, b: 2}` also valid?

Cannot wait to know what others think about this.

---

PS: I love the idea this would allow more syntax in the future which could be short, explicit and readable, for instance:

```
# Alternative to `%q{Hello, World}`?
%string{Hello, World} #=> "Hello, World"

# Alternative to `%w[one two three]`?
%array[one two three] #=> ["one", "two", "three"]

# Extra nice if Issue #16122/#18 were implemented. An even shorter alternative to the "Value" helper `Struct.V
alue(a: 1, b: 2, c: 3)`!
quick_data = %value{ a: 1, b: 2, c: 3 } #=> #<value a=1, b=2, c=3>
quick_data.a #=> 1

# Or going even crazier:
obj = %Object{ a: 1, b: 2, c: 3 } #=> #<Object:0x0000xxxxxxxxxxxx>
obj.a #=> 1

%BasicObject{ inspect: "This is why we can't have nice things..." }
#=> This is why we can't have nice things...
```

### #36 - 08/21/2020 08:51 AM - mame (Yusuke Endoh)

At first, I misunderstood this feature. I thought that this would be a useful substitute of symbol-key Hash (like JSON data) which we can read the values by `s.name` instead of `s[:name]`.

However, it is not. This use is potentially *dangerous*.

```
# Dangerous usage

json = JSON.parse(external_input) # { :name => "John", :age => 20 }

data = Struct(**json) # or something

data.name #=> "John"
data.nme #=> NoMethodError: useful to detect typo or wrong data?
```

Because method-name symbols are never GC'ed, so converting arbitrary external input to anonymous Struct is vulnerable against Symbol DoS.

To make it foolproof, adding a new literal instead of method like Struct() is very important. And for the same reason, note that we will never have Hash#to\_anonymous\_struct or something.

Also, we should not use this feature when keys are optional. It creates a struct class for all unique set of keys. So `#{ name: "John" }` and `#{ name: "John", age: 20 }` will create two different class objects. This may cause combinational explosion. You need to always write `#{ name: "John", age: nil, (and other keys if any...) }`.

Now, I'm unsure if and when this feature is actually useful.

### #37 - 08/24/2020 03:13 AM - Dan0042 (Daniel DeLorme)

I thought that this would be a useful substitute of symbol-key Hash (like JSON data) which we can read the values by `s.name` instead of `s[:name]`.

This is the same thing stated in the Feature proposal, and I would also like this, and it seems a lot of other people here too. So even if the solution is not exactly an Anonymous Struct literal let's find a way to be able to write this kind of nice-looking `s.name` code.

Because method-name symbols are never GC'ed, so converting arbitrary external input to anonymous Struct is vulnerable against Symbol DoS.

It should be possible to work around this. Let say we create an anonymous struct with Struct(\*\*json), instead of creating an actual Struct.new it could be something like AnonStruct.new where only already-existing method-name symbols are created as methods, and the other keys are handled via method\_missing.

It creates a struct class for all unique set of keys.

Assuming that the struct class can be garbage-collected when there are no longer any instances, I think this should not be a big problem?

### #38 - 08/24/2020 01:43 PM - mame (Yusuke Endoh)

Dan0042 (Daniel DeLorme) wrote in [#note-37](#):

I thought that this would be a useful substitute of symbol-key Hash (like JSON data) which we can read the values by `s.name` instead of `s[:name]`.

This is the same thing stated in the Feature proposal, and I would also like this, and it seems a lot of other people here too.

I thought so. Thus, I wanted to let all of you know what this proposal is *not*.  
BTW, the proposal includes the following note that states the problem:

Note

Unlike Hash literal syntax, this proposal only allows `label: expr` notation. No `#{**h}` syntax.

This is because if we allow to splat a Hash, it can be a vulnerability by splatting outer-input Hash.

So, external input like JSON data is not the target of this proposal. Rather, this proposal is just a variant of Struct, which allows to omit the definition line: `Foo = Struct.new(...)`.

Because method-name symbols are never GC'ed, so converting arbitrary external input to anonymous Struct is vulnerable against Symbol DoS.

It should be possible to work around this. Let say we create an anonymous struct with `Struct(**json)`, instead of creating an actual `Struct.new` it could be something like `AnonStruct.new` where only already-existing method-name symbols are created as methods, and the other keys are handled via `method_missing`.

In my opinion, it is too hacky to include into the core.

It creates a struct class for all unique set of keys.

Assuming that the struct class can be garbage-collected when there are no longer any instances, I think this should not be a big problem?

Theoretically, yes. But it will require much work.

**#39 - 08/27/2020 12:13 AM - ko1 (Koichi Sasada)**

So, external input like JSON data is not the target of this proposal. Rather, this proposal is just a variant of Struct, which allows to omit the definition line: `Foo = Struct.new(...)`.

Yes. This proposal is strict one.

**#40 - 08/30/2020 10:46 AM - nobu (Nobuyoshi Nakada)**

ko1 (Koichi Sasada) wrote in [#note-34](#):

how about `%struct{a: 1, b: 2}` (and `%value{...}` if needed)?  
(znz-san's idea and it seems nice)

Is `%struct"a: 1, b: 2"` same?

**#41 - 09/02/2020 12:17 PM - duerst (Martin Dürst)**

name (Yusuke Endoh) wrote in [#note-38](#):

So, external input like JSON data is not the target of this proposal. Rather, this proposal is just a variant of Struct, which allows to omit the definition line: `Foo = Struct.new(...)`.

I'm still struggling to understand the actual uses of this proposal. I understand that many people like it, but can we see *actual real use cases*, i.e. code that not only gets shorter but also continues to be at least as easy to understand as before?

I'm trying to compare these "anonymous Structs" to anonymous functions. The success of anonymous functions would suggest that "anonymous structures" are also a good idea. But I'm not so sure about this.

For anonymous functions (blocks, lambdas), there's no need to discuss whether two of them that look identical are identical or not; it's rather rare to have the same thing twice anyway (something like `{[a,b] a+b}` might be an example that may turn up multiple times). Functions also don't have a

second level of instantiation with actual data. And there's not much of a question about the semantics of such functions (in the above example, one could wonder whether it's an addition or a concatenation, but duck typing mostly makes that unnecessary/impossible).

For "anonymous Structs", the question of when two of these are the same seems to have several different expectations with different implementation and runtime consequences, but no widely satisfying solution. But it doesn't seem rare to have an "anonymous Struct" with the same components, because the actual instance data can differ. But the more instances I have, the stronger the need for a name becomes. If I have something like `{ name: 'foo', number: 'abcde' }`, I'm starting to wonder what that is: A person in a company? A room with name and number? Anything else of a lot of different choices?

So as you might guess, at this point, I'm very much not convinced. Examples with `a: 1, b: 2` don't really count as actual use cases, and are not concrete enough to help understand the actual pros and cons (except for syntax, which should be secondary).

**#42 - 09/02/2020 05:21 PM - marcandre (Marc-Andre Lafortune)**

I'm also unconvinced of a good use case and why creating a `MyResult = Struct.new(:some_value, :name)` is really something that should be "saved".

**#43 - 09/03/2020 01:32 AM - duerst (Martin Dürst)**

One more point: I haven't seen much examples of similar features in other languages. The only suggestion I saw was that of a similarity to Python tuples. But tuples are much closer to Arrays than to Structs or hashes. The easiest description for them may be "fixed-length Arrays".

(While not being available in (m)any other language(s) isn't by itself an argument against a feature, it definitely strengthens the need for careful evaluation and explanation of a new feature, including actual practical use cases.)