Ruby master - Feature #15799

pipeline operator

04/27/2019 08:31 AM - nobu (Nobuyoshi Nakada)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

Implemented the pipeline operator |>, a topic of "ruby committers vs the world" in RubyKaigi 2019.
Also a casual idea of rightward assignment.

```
1.. |> take 10 |> map {|e| e*2} |> (x)
p x #=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

https://github.com/nobu/ruby/tree/feature/pipeline

Related issues:

Related to Ruby master - Feature #15921: R-assign (rightward-assignment) oper...	**Closed**
Related to Ruby master - Feature #14392: Pipe operator	**Open**

Associated revisions

Revision f169043d - 06/13/2019 09:44 AM - nobu (Nobuyoshi Nakada)

Add pipeline operator [Feature #15799]

Revision 2ed68d0f - 08/29/2019 06:27 AM - nobu (Nobuyoshi Nakada)

Revert "Add pipeline operator [Feature #15799]"

This reverts commits:

- d365fd5a024254d7c105a62a015a7ea29ccf3e5d
- d780c3662484d6072b3a6945b840049de72c2096
- aa7211836b769231a2a8ef6b6ec2fd0ec882ef29
- 043f010c28e82ea38978bf8ed885416f133b5b75
- bb4dd7c6af05c7821d572e2592ea3d0cc748d81f
- 043f010c28e82ea38978bf8ed885416f133b5b75
- f169043d81524b5b529f2c1e9c35437ba5bc3a7a

http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/94645

History

#1 - 04/27/2019 10:28 AM - duerst (Martin Dürst)

Thanks for creating an issue.

The |> symbol looks reasonable to me, but I'd like to see more examples where this notation is preferable to

```
(1..).take(10).map {|x| x*2}
```

As for the assignment, just using parentheses looks confusing to me. It's clear we can't use >= or |=, but |>= or some other combination would be much clearer than just parentheses.

#2 - 04/27/2019 01:23 PM - Eregon (Benoit Daloze)

My early thinking about this syntax is it's a very narrow use-case.
Is it anything more than . and no need for parentheses in some rare cases?
There is also the RHS assignment which feels very unnatural to me.

I think there is nothing wrong with having parentheses for Range, I think they actually help readability.
Martin's desugared version which works today is actually shorter and I believe most would agree it's also clearer.

To be fair, I don't particularly like Haskell code and find it very cryptic, which this is getting closer to.

Also, if we actually introduce a pipeline operator, I think it's much more useful to have Elixir semantics of passing the result as the first argument of the RHS, than just a different syntax for ..

#3 - 04/27/2019 02:19 PM - shevegen (Robert A. Heiler)

> I think there is nothing wrong with having parentheses for Range, I think they actually help readability.

My comment is not meant in regards to the example given for Range here, but more in general. I am not sure if having mandatory parentheses is a (visual/syntactic) improvement. Python folks often claim this, while forgetting that you can easily use () in ruby. I myself like to avoid (), but I never avoid them when I do "def foobar(a, b, c)". This is often just to satisfy the expectation that I have when looking at ruby code; my brain works easier/faster for method definitions having (). I don't have any particularly strong preference for () elsewhere, though.

There is one slight problem in regards to syntax diversification in general - it can make the code that ruby users write difficult for others to read.

You mentioned Haskell lateron, and I remember an older discussion on #gobolinux with a hacker from denmark (oejet) who was using Haskell. Back then he stated that writing haskell is quite easy for him, but reading haskell is quite difficult.

While ruby is, in my opinion, much easier than haskell, I think adding syntax flexibility, while this in itself is not necessarily bad, and it may add new feature to ruby, it makes it hard(er) to work with code written by other people. Of course it may be great for individual freedom, but for larger projects I think this may be somewhat ... hmm, more difficult. I myself would not be able to adjust to projects that would use features that I do not use in my own code base. (Although I also tend to avoid looking at ruby code written by other people these days, because there is so much terrible code out there ... I am sure others may say this about my own code bases too ;)).

> Martin's desugared version which works today is actually shorter and I believe most would agree it's also clearer.

Yes, I think that example was clearer - I have this often with some other changes too. Oldschool ruby beats many newer additions (syntax-wise; I don't think people have much against features/functionality per se, but I myself think that syntax is hugely important too. Otherwise we could use languages with a terrible syntax.)

Note that I do not have a huge preference in this regard though. I slightly dislike |> but not so much because of the syntax, more that to me it feels as if it came from elixir and people who are in favour of the syntax seem to be elixir-users who cross language ideas. It is not even against ruby's "philosophy" per se, since matz always tried to make it convenient for people to use ruby (e. g. collect versus map, people use whichever variant they prefer), but I still don't feel it is necessarily a great idea to adopt syntax that works very well in other languages. My opinion is that this often just does not work well in e. g. ruby.

People seem to base their reasoning largely on their own personal preferences primarily; I guess one can say the same about people disliking any feature/change, too. :)

> To be fair, I don't particularly like Haskell code and find it very cryptic, which this is getting closer to.

Oddly enough, I actually felt that Haskell has a fairly clean syntax. What I dislike about Haskell is the complexity. I don't want to think when I code (yes, I am not the thinker; I am more a random tinkerer; I probe/change at code until it works and does what I want it to do, hopefully).

> Also, if we actually introduce a pipeline operator, I think it's much more useful to have Elixir semantics of passing the result as the first argument of the RHS, than just a different syntax for ..

Hah! I actually thought that above code was elixir-inspired. :) Now your comment is more similar to Haskell (I don't quite know haskell very well; nor elixir either really).

But otherwise I agree - I think oldschool ruby is clearer.

Still, having said that, what matz once wrote somewhat applies to this here too - you don't have to use it if you dislike it. I do this with a lot of ruby, e. g. I only use

a subset of ruby, and I quite happily ignore the rest. :P

I guess perhaps there should be more examples given. Right now only nobu gave an example here and I think it would be more fair if others e. g. in particular during rubykaigi but also on the bug tracker, show more examples for the pipeline operator/idea (I don't know what was discussing during ruby kaigi of course; I guess some other folks may also not know that either).

#4 - 04/28/2019 01:26 AM - nobu (Nobuyoshi Nakada)

duerst (Martin Dürst) wrote:

> As for the assignment, just using parentheses looks confusing to me. It's clear we can't use >= or |=, but |>= or some other combination would be much clearer than just parentheses.

OK, I separated the right-assign to https://github.com/nobu/ruby/tree/feature/rassgn-pipeline, and another operator https://github.com/nobu/ruby/tree/feature/rassgn-funnel.

#5 - 05/01/2019 01:53 AM - duerst (Martin Dürst)

Eregon (Benoit Daloze) wrote:

> To be fair, I don't particularly like Haskell code and find it very cryptic, which this is getting closer to.

Haskell code can be very clear (or very cryptic). In my personal experience, I found the $ operator in Haskell one of the most difficult to get used to. It's essentially "replace it by a '(', and add a ')' at the very end of the line.

Compared to this, the proposed |> for Ruby seems rather easy in the examples that have been brought up until now. The use case seems to be to avoid parentheses around method arguments rather than parentheses that span all the way to the end of the line. That's more localized and therefore easier to grok.

But we haven't seen many different usage examples, and the ones we have seen haven't been very convincing, at least not to me.

#6 - 05/19/2019 10:05 PM - jonathanhefner (Jonathan Hefner)

Eregon (Benoit Daloze) wrote:

> Also, if we actually introduce a pipeline operator, I think it's much more useful to have Elixir semantics of passing the result as the first argument of the RHS, than just a different syntax for ..

I agree that the pipeline operator should introduce new semantics, instead of being an alias of ..

While Elixir uses |> to insert a first argument to the RHS, F# and Elm use |> to append a last argument to the RHS (due to different conventions regarding function parameter order).

One use-case for such "last argument" behavior in Ruby is file IO:

```
File.read("file.txt").gsub(/foo/, "bar") |> File.write "file.txt"
```

Of course it gets tricky when option arguments are involved. But perhaps kwargs could be handled specially, such that the LHS of |> is inserted before them. For example, the following would both work as expected:

```
File.read("other.txt") |> File.write "file.txt", mode: "a"
```

```
File.read("other.txt") |> File.write "file.txt", **options
```

#7 - 05/25/2019 01:28 AM - jeremyevans0 (Jeremy Evans)

I think a pipeline operator can be helpful in a functional language, based on my experience with $ in Haskell. However, I don't think it is a good idea to implement a replacement for the . operator just to avoid parentheses. If it has different semantics, maybe it could be useful, but it would depend on what those semantics are. Because Ruby uses methods instead of functions, I'm not sure what semantics we would want from a pipeline operator.

#8 - 05/28/2019 04:54 AM - konsolebox (K B)

nobu (Nobuyoshi Nakada) wrote:

> duerst (Martin Dürst) wrote:
>
> > As for the assignment, just using parentheses looks confusing to me. It's clear we can't use >= or |=, but |>= or some other combination would be much clearer than just parentheses.

> OK, I separated the right-assign to https://github.com/nobu/ruby/tree/feature/rassgn-pipeline,
> and another operator https://github.com/nobu/ruby/tree/feature/rassgn-funnel.

|>= looks heavy. Please consider |: instead.

#9 - 05/28/2019 07:37 AM - konsolebox (K B)

konsolebox (K B) wrote:

> nobu (Nobuyoshi Nakada) wrote:
>
> > duerst (Martin Dürst) wrote:
> >
> > > As for the assignment, just using parentheses looks confusing to me. It's clear we can't use >= or |=, but |>= or some other combination would be much clearer than just parentheses.
> >
> > OK, I separated the right-assign to https://github.com/nobu/ruby/tree/feature/rassgn-pipeline,
> > and another operator https://github.com/nobu/ruby/tree/feature/rassgn-funnel.
>
> |>= looks heavy. Please consider |: instead.

Or =: which is the reverse of Pascal's assignment operator. Personally I would want it to have it as an alias to |: than replace |: because =: would look good if it's placed last, but |: would look better halfway.

```
1.. |> take 10 |: ten |> map{ |x| x * 2 } =: doubled
```

My opinion on this new set of operators is that they're good for writing drafts of code quickly because you can easily place an assignment to a variable at the end of a statement, or insert it somewhere, but old-school is better for formal writes since it's more readable. In old-school assignments, you can easily find where the assignment happens and know when it happens.

#10 - 05/28/2019 09:33 AM - phluid61 (Matthew Kerwin)

a|:b means a | :b and a=:b means a = :b

#11 - 05/28/2019 11:41 AM - zverok (Victor Shepelev)

```
1.. |> take 10 |> map {|x| x*2} |> (x)
```

I believe that the ONLY sane reason for the new operator is ending the long chain with "...and now, put it into variable". The rest is total mistery, however you look at it, e.g. why not

```
(1..).take(10).map {|x| x*2}
```

...that's perfectly readable and usual Ruby, why rip the Elixir?

> As a side note, I became less wanting for the feature "...and now put it into variable" since introduction of then, because you always can
>
> ```
> (1..).take(10).map {|x| x*2}
>   .then { |res|
>      #...work with the result
>   }
> ```

But **if** "...and now put it into a variable" is necessary, why not reuse =>? It is invalid syntax currently (without braces), and would be consistent with rescue Foo => x and the same use in a new pattern-matching.

So the result would be

```
(1..).take(10).map {|x| x*2} => x
```

#12 - 05/28/2019 01:45 PM - konsolebox (K B)

phluid61 (Matthew Kerwin) wrote:

> a|:b means a | :b and a=:b means a = :b

Yes I implied that a space is necessary for |: or =: to be distinguishable. It's better to have that requirement than have the three-character operator. But I don't mind other better alternatives.

To be fair I'm not really a fan of incorporating anything that's coming from the functional world, and I prefer writing explicit code (e.g. prefers

parentheses over white space), but I don't mind having these features because I think it would help me write prototype code faster. But not with a three-character operator. Even |> is already awkward to type. I can write 86 WPM but I care about efficiency.

#13 - 06/12/2019 06:36 AM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

> But **if** "...and now put it into a variable" is necessary, why not reuse =>? It is invalid syntax currently (without braces), and would be consistent with rescue Foo => x and the same use in a new pattern-matching.

OK, done.

```
$ ./ruby -v -e '(1..).lazy.map {|x| x*2} => x' -e 'p x.first(10)'
ruby 2.7.0dev (2019-06-12T06:32:32Z feature/rassgn-assoc c928f06b79) [x86_64-darwin18]
last_commit=Rightward-assign by ASSOC
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

https://github.com/nobu/ruby/tree/feature/rassgn-assoc

#14 - 06/12/2019 01:35 PM - zverok (Victor Shepelev)

nobu (Nobuyoshi Nakada) Awesome. I don't know what other's would think, but from my perspective, the feature now becomes consistent with the language and have (however small) chances to be appreciated by the public.

#15 - 06/13/2019 08:28 AM - nobu (Nobuyoshi Nakada)

- Description updated

#16 - 06/13/2019 09:48 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset git|f169043d81524b5b529f2c1e9c35437ba5bc3a7a.

Add pipeline operator [Feature #15799]

#17 - 06/13/2019 10:00 AM - Eregon (Benoit Daloze)

Why was this accepted? I cannot see anyone really supporting this idea in this issue's comments.
As everyone said, it hurts readability, looks like Haskell, and there seems to be really no need for a different syntax for methods calls.

The example in the commit seems very unconvincing to me:

```
# today
x = (12 ** 2).to_s(11)
# the new syntax
x = 12 |> pow(2) |> to_s(11)
```

It's even much longer.

#18 - 06/13/2019 11:43 AM - shevegen (Robert A. Heiler)

I assume matz approved it. I don't think it is good to want to reason against a change since the decision making is not a community-decision in general.

As for the feature itself - I do not like the syntax, but as is the case with other changes, if I don't have to use it I don't really mind. ;)
Otherwise I agree with Benoit's comment in regards to syntax; I don't necessarily agree on the other implied statement.

I actually can see some reason for the pipeline, aside from elixir - it may look a bit more natural purely from when one can want to avoid using the "." (dot). That does not mean that I like the syntax but I can understand it a bit. I sort of infer indirectly, though.

There is, however had, one thing that absolutely should happen - please consider adding some documentation and comments to explain what this feature is doing or why it was added. I understand that ruby kaigi was a driving factor here, but keep in mind that not everyone was at ruby kaigi; lots of folks may not know what was discussed at kaigi, so it would be really helpful if it could be commented what it does or what the discussion was about - like developer meetings where short summaries are provided too.

No worries if this takes a while, but please don't forget it - right now I, for example, don't really know what the feature is really doing. I assume it is partially inspired by elixir but I may be wrong too - right now we just don't quite know. Or at the least I guess folks outside japan perhaps.

I don't mind the decision making process in ruby at all, different to benoit perhaps :P - but I think documentation, comments etc... is an area where ruby should get better in general. The more, and better, documentation/explanation is given to ruby users, the better it will be for them to work with ruby in general, in particular when it comes to changes. There will probably be blog posts about this eventually, but the folks who write the blog posts will probably also appreciate as much information as possible.

Thanks.

#19 - 06/13/2019 01:23 PM - dgutov (Dmitry Gutov)

"I don't have to use it" doesn't work in the real world where you have projects with multiple contributors, dependencies and multi-year history.

This new "feature" does not add *anything* except a new way to write a method call, and will confuse anybody familiar with pipeline operators in other (functional) languages.

Please revert. And FFS, could you slow down on adding new syntax without due discussion?

Also see the thoughtful comments on the commit by others: https://github.com/ruby/ruby/commit/f169043d#commitcomment-33926163

#20 - 06/13/2019 02:09 PM - ttilberg (Tim Tilberg)

> a topic of "ruby committers vs the world" in RubyKaigi 2019.

Link to video of discussion Nobu referenced: https://youtu.be/5eAXAUTtNYU?t=1944

I turned on Auto-Translate to English which marginally helped add a small bit of context, but was generally poorly translated.

#21 - 06/14/2019 12:12 AM - shioyama (Chris Salzberg)

Responses to this change:

- Twitter: https://twitter.com/a_matsuda/status/1139110957450375168
- Reddit: https://www.reddit.com/r/ruby/comments/c059d2/pipeline_operator_in_ruby/
- Github (mentioned above): https://github.com/ruby/ruby/commit/f169043d#commitcomment-33926163

I watched the video in Japanese, and here's my rough summary.

The operator has lower precedence than ., so you can do this:

```
a .. b |> each do
end
```

With ., because of its higher precedence, you'd have to do use braces:

```
(a..b).each do
end
```

That's actually the main reason this change was proposed and accepted (AFAICT). No other substantial arguments are given. A few people mention they really want the "less than" pipeline (<| I guess) more than the proposed "greater than" pipeline (|>).

Matz does very briefly address what most people have said they dislike about this feature, by saying that actually, Elixir's pipeline is not a "real" pipeline either. I'm not really sure what he means by that, but Elixir's pipeline operator is one of the most attractive features of the language, so if you're going to borrow the syntax, you'd better follow the meaning. And this doesn't.

Beyond that, there's a bunch of unicode jokes about arrows and pointing hands, etc., but no real substantive debate or discussion. Interestingly, Matz actually mentions that he doesn't think this feature should be released in 2.7 (but now it seems like it will).

What's funny though is that in the video, this example is given as Elixir usage:

```
x |> func1 |> func2
func2(func1(x))
```

Then this corresponding example is given in Ruby with methods:

```
x |> method1(1) |> method2(2)
```

but when they start translating that to what it should mean in live code, it starts coming out as:

```
method()...
```

then Matz says "Naranai naranai, chigau gengo ni natteiru" ("no no, that's a different language") and it's "fixed" to:

```
x.method1(1).method2(2)
```

So obviously, this is something that is going to trip up almost everyone, including Ruby committers!

The bigger point IMHO though is that major controversial decisions are made based on this kind of very brief, mostly closed discussion. I don't think that's a good thing for our community.

#22 - 06/14/2019 02:01 AM - inopinatus (Joshua GOODALL)

Please consider adjusting the precedence of the pipeline operator to be above that of assignment. I was surprised by this outcome:

```
result = 3 |> pow(2) #=> 9
result #=> 3 (!?!?!)
```

#23 - 06/14/2019 02:51 AM - baweaver (Brandon Weaver)

I have written on my opinions here: https://dev.to/baweaver/ruby-2-7-the-pipeline-operator-1b2d

But in summary, I believe this feature could be substantially more expressive and powerful if it adjusted its lookup chain from directly calling on an object to searching the local scope for procs and methods.

Please consider:

```
def double(n) n * 2 end

increment = -> n { n + 1 }

5
|> double    # Method
|> increment # Proc
|> to_s(2)   # self.to_s
```

This would find the double method locally and use it, passing the value as an argument in a way that is expected in other pipeline implementations. It would then pass to a local proc (without & prefix) and also give its argument to it. Lastly, to_s would not be found so it would resolve to the Object itself and follow the regular call-chain through.

I believe strongly that this would be a huge win for expressive power in the Ruby language if such a feature were fully implemented, but as it is now it is only an alias for what already exists.

#24 - 06/14/2019 03:00 AM - matz (Yukihiro Matsumoto)

inopinatus (Joshua GOODALL) We are working on the right side assignment operator. Combine pipelines with it.

Matz.

#25 - 06/14/2019 03:08 AM - nobu (Nobuyoshi Nakada)

- Related to Feature #15921: R-assign (rightward-assignment) operator added

#26 - 06/14/2019 03:18 AM - jeremyevans0 (Jeremy Evans)

baweaver (Brandon Weaver) wrote:

> But in summary, I believe this feature could be substantially more expressive and powerful if it adjusted its lookup chain from directly calling on an object to searching the local scope for procs and methods.

The issue with that is it either leads to ambiguity or the requirement to remember the precedence:

```
def double(n) n * 2 end
double = -> n { n << 4 }
x = [2]
def x.double; n.map{|x| x * 2} end

x
|> double
# [2, 2] or [2, 4] or [4]?
```

One possible lookup order is:

1. local variable, and call call on the local variable with LHS as the only argument
2. method call on self with LHS as additional argument
3. method call on LHS with no additional arguments

However, what about method_missing? Logically, if you support a method call on self before a method call on the LHS, method_missing should be called on self if the method is not found. In that case, you would never get to 3. unless you did something like rescue NoMethodError internally, which is not desired.

> I believe strongly that this would be a huge win for expressive power in the Ruby language if such a feature were fully implemented, but as it is now it is only an alias for what already exists.

While I'm overall against the addition of the pipeline operator (with or without R-assign operator), having identical operators with different precedences has a history in Ruby, with &&/and and ||/or.

#27 - 06/14/2019 05:42 AM - shioyama (Chris Salzberg)

I think one big issue here is simply the choice of symbols for this thing. If it were any other symbols, I suspect a lot of people would not have reacted so emotionally.

To most Rubyists who know it, |> has a distinctive meaning. It's that bit of functional programming that they want, that they wish Ruby had, that Elixir has had from day one.

Whether you like it or not, the . alias here is entirely different from a "pipeline". Giving it the same name is bound to frustrate a lot of people (like myself) who have been hoping and waiting for the day when Ruby would support function composition in a more natural way.

Barring a revert, how about considering a different pair of symbols for this thing? That would seem to be a reasonable compromise.

#28 - 06/14/2019 06:15 AM - baweaver (Brandon Weaver)

That's a very fair point jeremyevans0 (Jeremy Evans), it would introduce two lookup chains even under the most ideal of circumstances. Perhaps if & were slightly extended, one could do this:

```
def double(n) n * 2 end

increment = -> n { n + 1 }

5
|> &double
|> &increment
|> to_s(2)
|> reverse
|> to_i
```

...wherein & would retain its meaning of coercing something to a proc, make the call chain more explicit, and kill the speed hit potential.

The downside is it would add a feature to & that it could convert a method to a proc which does not currently exist and would do all types of interesting things to the interpreter for parens. It had been mentioned in the past to have anonymous or self-oriented .: to get around this:

```
5
|> &.:double
|> &increment
|> to_s(2)
|> reverse
|> to_i
```

Though that presents issues of 2+ arity functions. It's certainly a hard issue to solve for, but if a solution can be found it would be extremely powerful for the language.

#29 - 06/14/2019 07:45 AM - matz (Yukihiro Matsumoto)

Unlike JavaScript and Python (Lisp-1 like languages), Ruby is a Lisp-2 like language, in which methods and variable have separated namespaces. In Lisp-1 like languages, f1 = function; f1() calls function (single namespace).

In a Lisp-2 like language, ordinary (Elixir like) pipeline operator does not work, because it's harder to retrieve a method object in the language. Besides that, the receiver of Ruby methods can be considered as the first argument. So

```
a |> method1() |> method2() # or a.method1().method2()
```

can be considered as

```
method2(method1(a))
```

in other languages. So calling it a pipeline operator is not that out scoped.

Maybe we should call it a **chaining operator** and replace different combination of characters (>>> for example?) to avoid confusion.

Matz.

#30 - 06/14/2019 08:27 AM - baweaver (Brandon Weaver)

I wonder if it would be possible to get method objects somehow. .: was capable of doing this off of objects, but doing it in the main namespace is not easy.

If pipeline operators in an Elixir style were considered, it may be able to aliased to then and be sugar for that. & could already be used for Proc type objects, could .: be a standalone prefix for local methods?

```
def double(n) n * 2 end

increment = -> n { n + 1 }

5
|> &.:double  # then / .: prefix
|> &increment # then / to_proc
|> to_s(2)    # . method call
|> reverse    # . method call
|> to_i       # . method call
```

This may address some concerns, but I'm still not happy with my implementation here yet. I will think on this and consider some alternatives.

I believe the crux here to be method -> proc coercion. If that can be done cleanly or inferred in a pipeline that would make it much easier.

If |> was treated as an operator that behaves differently when it receives a proc that may help, but still not sure.

#31 - 06/14/2019 10:30 AM - d-snp (Tinco Andringa)

From my perspective, the |> operator exists because in functional programming languages you have these increasingly nested function calls which have to end with a stack of parentheses that are hard to read. In Haskell I very frequently use the $ operator for a similar effect. In Ruby this problem is not significant at all, it almost never happens that we have nested parentheses, and in the case we do (like calling methods inside method parameters) the pipeline operator wouldn't be a solution anyway.

I think it's not good to introduce a new operator just for a rare use case that isn't idiomatic Ruby.

If my employees wrote

```
a |> method1 b |> method2 c
```

I would correct them that the code could be simplified to

```
a.method1(b).method2(c)
```

Not the other way around.

It's not even very common for Ruby methods to return self.

#32 - 06/14/2019 11:03 AM - rogeriochaves (Rogerio Chaves)

May I give yet another suggestion? What about :., it would keep the visual effect

```
1 + 1
:. to_s 2
:. reverse
:. to_i
```

#33 - 06/14/2019 01:15 PM - cichol (Renxiang Cai)

Hi,

I want to introduce a way to pipeline method calls in Ruby.

I imagined that the Ruby-styled pipelined calls should be like:

```
1.pipe do
  call 1 + _ # => after this line, _ becomes 2
  call _ * 3 # => after this line, _ becomes 6
end # => _ is returned as 6
```

With a seemed redundant call method, this pipe method can be implemented in current version of Ruby.

I think it would be so good if we can remove the need of call and allow a line result capturer defined as a hook.

The example now becomes:

```
1.pipe do
  1 + _
  _ * 3
end
```

Within the block following pipe, results of every line are captured and passed to a hook for assignments of _.

IMHO, this implementation of pipeline is preferable over the pipeline operator suggested above, for these reasons:

1. This method utilize the placeholder _ to pass the argument into any position of parameters instead of the last one.
   If you are using pipeline operator like those in functional languages, you will need to carefully deal with the order of parameters, which adds mental overhead.
   And most existing Ruby methods are not implemented with currying in mind. With use of placeholder we can simply re-use previous methods without additional costs.

2. Explicitly calling by a method pipe and a block.
   This allows less aggressive modification to the language over adding operators. It is more Ruby-way and clearer.

3. The abstraction of line result capturer can be useful for other use cases.
   For example, if we want to inspect every steps of a pipe, we can imagine a special pipe that prints out every step:

   ```
   1.inspected_pipe do
   1 + _ # => puts 2
   _ * 3 # => puts 6
   SomeService.new.process _ # => puts whatever the return value is
   end
   ```

   It helps debugging. We can apply tiny modification to make a block loggable.

4. BTW, to avoid parentheses for range, we can simply:

   ```
   # for this we need to change the `self` for every line but is still doable
   x = pipe do
   1..
   take 10
   map{|e| e*2}
   end
   ```

   to achieve the same functionality asked in the first post.

5. The reason we need pipeline is to make a step-by-step process clearer, and newlines are good seperators.
   This method encourages people to write a process in several lines (as several steps) instead of one liner, which will not risk hurting readability as the |> may do.

#34 - 06/14/2019 04:56 PM - shuber (Sean Huber)

cichol (Renxiang Cai) Agreed! Here is a working proof of concept for an "operator-less" pipe operator which feels more natural in Ruby: https://github.com/lendinghome/pipe_operator

```
# before
JSON.parse(Net::HTTP.get(URI.parse(url)))

# after
url.pipe { URI.parse; Net::HTTP.get; JSON.parse }

# with arguments and method chaining support
"https://api.github.com/repos/ruby/ruby".pipe do
  URI.parse
  Net::HTTP.get
  JSON.parse.fetch("stargazers_count")
  yield_self { |n| "Ruby has #{n} stars" }
  Kernel.puts
end
#=> Ruby has 15120 stars
```

#35 - 06/14/2019 07:30 PM - konsolebox (K B)

rogeriochaves (Rogerio Chaves) wrote:

May I give yet another suggestion? What about :., it would keep the visual effect

```
1 + 1
:. to_s 2
:. reverse
:. to_i
```

Or \ maybe. It's much easier to type.

1 + 1 \ to_s 2 \ reverse \ to_i

We can then have |> as an alias to then.

#36 - 06/14/2019 11:12 PM - dgutov (Dmitry Gutov)

Matz:

Ruby being a "Lisp-2" means that the pipeline operator couldn't be implemented like a "normal" operator (all of them being translated to methods on Object or etc), but it could be implemented on the parser level, like almost all the other languages do which have it. Clojure implements it as a macro, but I don't think it's an option here. Sure, you could say that it would make it un-Ruby-like, and I might agree. However, *that* is the pipeline operator a significant number of Ruby programmers might be happy to have.

I don't think, however, that there is much value in adding a language feature that looks like it, but behaves differently. We already have method chaining in the language (it's the most common feature of modern OO languages), why add this?

#37 - 06/16/2019 12:17 AM - mame (Yusuke Endoh)

I investigated history of pipeline operator. It is very long, so I wrote an article in my blog. In short: In my current opinion, the current spec is somewhat reasonable, never so strange. I'm yet unsure if or not the feature is good to have in Ruby, though.

#38 - 06/16/2019 12:49 AM - ioquatix (Samuel Williams)

At first, I wasn't so sure about how to use such an operator, and honestly, the ASCII symbol |> is a bit jarring, but using a font with ligatures you get a better idea of how it's supposed to look, and it does look really great.

I have always desired right-assignment operator, and now I see it's being worked on in separate issue so I'm really happy to see that.

I tried to think about some situations where the pipeline operator makes code more readable, and I think there are some areas where it is a great improvement.

```
# Can't use do...end because it binds block to `puts`
logger.debug Async::Clock.measure {
    # Slow things
}.round(2)

Async::Clock.measure do
    # Slow things
end.round(2) |> logger.debug
```

Can we combine it with numbered arguments? (with single argument)

```
Async::Clock.measure do
    # Slow things
end.round(2) |> "It took #@s" |> logger.debug
```

Can we branch pipelines (maybe bad idea)?

```
Async::Clock.measure do
    # Slow things
end.round(2) |>
    (@ > 1.0) ?
        "It was slower than expected #@s" |> logger.debug :
        "It was fast enough #@s" |> logger.info
```

(Might allow to make pipeline lazy evaluated/avoid evaluation if not needed).

Just throwing out some ideas.

#39 - 06/16/2019 08:11 AM - josh.cheek (Josh Cheek)

The operator doesn't bother me, though I can't think of any time I'd use it.

Several suggested alternatives seem to want to leave the syntax ambiguous, leaving it unclear whether the piped thing is the receiver or the arg and its unclear where the method comes from. This means the syntax would remain ambiguous until execution time. You could potentially add a method

in one location, which caused the syntax to change in another.

I think Haskell's dollar sign would be more useful. It lowers precedence, which, would also allow dropping of parentheses. From mame (Yusuke Endoh)'s blog, it sounds like that's the purpose of this operator.

```
x = 1.. $ .take 10 $ .map {|e| e*2}
x # => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

```
p 10.times.map $ do |i|
  2*Math::PI*i/10
end
# >> [0.0, 0.6283185307179586, 1.2566370614359172, 1.8849555921538759
```

```
1 + 2 $ * 3  # => 9
```

The downside of using $ is that it may conflict with the Perl style hooked variables.

#40 - 06/16/2019 03:29 PM - shan (Shannon Skipper)

Seeing |> my assumption would be that you could use it in the functional style, so you could do:

```
42 |> Integer(exception: false) |> Math.sqrt
```

Instead of:

```
Math.sqrt(Integer(42, exception: false))
```

Since this seems like the opposite direction of |>, might <| be an acceptable alternative to >>>?

```
1.. <| take 10 <| map {|e| e*2} <| (x)
#=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
1.. |> take 10 |> map {|e| e*2} |> (x)
#=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
1.. >>> take 10 >>> map {|e| e*2} >>> (x)
#=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

#41 - 06/30/2019 09:52 AM - Eregon (Benoit Daloze)

inopinatus (Joshua GOODALL) wrote:

> ```
> result = 3 |> pow(2) #=> 9
> result #=> 3 (!?!?!)
> ```

This makes it clear to me: the pipeline operator as it is just seems a hack to avoid parentheses around Range literals.
The precedence seems not intuitive.
Unlike and, where if a = 3*2 and a == 6 is rather clear, here assignment before |> is never wanted.
So probably assignment before |> should be a SyntaxError.

This syntax saves Range literal parenthesis, but it doesn't even save characters, so I see not point for it as it is.

Others will say it's nice for multi-line method call chains, but . works just fine in that case too when using indentation:

```
1..
|> take 10
|> each { |x| p x }
```

```
(1..)
  .take(10)
  .each { |x| p x }
```

(the example is from mame (Yusuke Endoh)'s post)

Finally, as other have said, the most essential operator of OO languages, the method call operator, does not need to be reinvented, . works just fine.

#42 - 06/30/2019 10:19 AM - Eregon (Benoit Daloze)

matz (Yukihiro Matsumoto) wrote:

> Unlike JavaScript and Python (Lisp-1 like languages), Ruby is a Lisp-2 like language, in which methods and variable have separated namespaces. In Lisp-1 like languages, f1 = function; f1() calls function (single namespace).
>
> In a Lisp-2 like language, ordinary (Elixir like) pipeline operator does not work, because it's harder to retrieve a method object in the language.

Could you give an example of why it doesn't work and why Lisp-2/Lisp-1 is problematic?

I think the current distinction used for no-receiver no-arguments calls (i.e., vcall) like foo works:
if there is a local variable foo then use that, if not, call a method named foo.

We have the method operator now, so we could do this if we wanted pipeline expressions to be actual call-able objects (which makes it easier to understand and functional):

```
"42" |> Kernel.:Integer |> Math.:sqrt |> -> x { x * 2 }
```

Also, this is what many proposed and feel intuitive. Would it be problematic, why?

```
"42" |> Integer(exception: false) |> Math.sqrt
```

We could simply have different semantics for the expressions on the RHS of |>, so they are considered curried method calls, no?

#43 - 06/30/2019 10:28 AM - Eregon (Benoit Daloze)

- Related to Feature #14392: Pipe operator added

#44 - 06/30/2019 10:38 AM - Eregon (Benoit Daloze)

I remembered an old blog post from over 10 years ago by Dave Thomas, and finally found it:
https://pragdave.me/blog/2007/12/30/pipelines-using-fibers-in-ruby-19.html

Interestingly the syntax is a bit similar to functional languages:

```
pipeline = evens | multiples_of_three | multiples_of_seven
```

My interpretation of this is many Rubyists want a functional/Elixir-like pipe operator.
There is even still an open issue about that: #14392, and probably more issues and definitely more blog posts wanting that.
And yet the operator we introduced doesn't address that, is redundant with . and doesn't work for normal assignment.

#45 - 06/30/2019 11:53 AM - Eregon (Benoit Daloze)

I propose to put the experimental pipeline operator behind a flag, disabled by default, until the major issues with it reported here are solved: #15966.
I think this experimental feature needs more discussion to improve it, and it shouldn't be enforced to happen before the Ruby 2.7 release deadline.

#46 - 08/29/2019 05:39 AM - matz (Yukihiro Matsumoto)

After experiments, |> have caused more confusion and controversy far more than I expected. I still value the chaining operator, but drawbacks are bigger than the benefit. So I just give up the idea now. Maybe we would revisit the idea in the future (with different operator appearance).

During the discussion, we introduced the comment in the method chain allowed. It will not be removed.

Matz.