From [https://bugs.ruby-lang.org/issues/15799#change-78465](https://bugs.ruby-lang.org/issues/15799#change-78465), proposal of the rightward-assignment operator by =>.

```
$ ./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]
```


## Related issues:
- Related to Ruby master - Feature #15799: pipeline operator
  - Closed
- Related to Ruby master - Misc #16802: Prefer use of RHS assignment in documentation
  - Closed

## Associated revisions
**Revision 1b2d351b - 04/10/2020 06:03 AM - nobu (Nobuyoshi Nakada)**
Rightward-assign by ASSOC

[Feature #15921]

**Revision 878e21c6 - 04/10/2020 08:35 AM - nobu (Nobuyoshi Nakada)**
R-assign is still experimental [Feature #15921] [ci skip]

## History

**#1 - 06/14/2019 03:08 AM - nobu (Nobuyoshi Nakada)**
- Related to Feature #15799: pipeline operator added

**#2 - 06/14/2019 10:49 AM - Hanmac (Hans Mackowiak)**
where does the rightward assign works and where it is blocked? y => x might be treated as Hash Parameter
like m y => x is this m(y) => x or still m{|y => x|}

**#3 - 06/14/2019 12:27 PM - nobu (Nobuyoshi Nakada)**
This has lower precedence, so the latter.

**#4 - 06/16/2019 12:55 AM - ioquatix (Samuel Williams)**
There are two areas where I think this is a great addition:

```ruby
x = if foo
  bar
else
  baz
end

if foo
  bar
else
  baz
end => x
```

I prefer the latter, because it avoids messing with the indentation/readability of the if expression.

Additionally, sometimes I find using irb I have made very large expression. In terminal, going to start of line isn't always obvious/easy. So, I wish to
save expression, usually I just press enter and then write \( x = _ \) to save last result. But when I go back in history to execute statement again, I must make same "hack". So, I wish I can just write:

very long query to get list of users => users

That way I don't need to think so hard or go back to start of statement. It might also be nice in middle of expressions, e.g.

\[
\text{Users.where(active: true) => active_users.where(type: "admin") => admin_users}
\]

I don't know if such usage is possible or anticipated, I just wanted to show some ideas - for long expressions sometimes I want to check the middle of the expression.

#5 - 06/16/2019 12:55 AM - ioquatix (Samuel Williams)

If it's not clear, previous statement is evaluated like:

\[
\text{Users.where(active: true) => active_users}.where(type: "admin") => admin_users
\]

#6 - 06/17/2019 12:56 PM - nobu (Nobuyoshi Nakada)

ioquatix (Samuel Williams) wrote:

If it's not clear, previous statement is evaluated like:

\[
\text{Users.where(active: true) => active_users}.where(type: "admin") => admin_users
\]

It can't be higher precedence than \( . \), or it will conflict with other syntaxes too much. Rather it should be interpreted like as:

\[
\text{admin_users} = \text{active_users}.where(type: "admin") = \text{Users.where(active: true)}
\]

Though it is a syntax error at the parenthesis after where currently.

#7 - 07/29/2019 08:26 AM - ko1 (Koichi Sasada)

- Assignee set to matz (Yukihiro Matsumoto)

#8 - 07/29/2019 08:31 AM - nobu (Nobuyoshi Nakada)

nobu (Nobuyoshi Nakada) wrote:

ioquatix (Samuel Williams) wrote:

If it's not clear, previous statement is evaluated like:

\[
\text{Users.where(active: true) => active_users}.where(type: "admin") => admin_users
\]

It can't be higher precedence than \( . \), or it will conflict with other syntaxes too much. You may be able to use \( | > \) here.

\[
\text{Users.where(active: true) => active_users | > where(type: "admin") => admin_users}
\]

#9 - 02/25/2020 10:34 AM - sawa (Tsuyoshi Sawada)

I think \( => \) is okay, but in case we want to use a keyword (ordinary word) for this feature, I think as would be good. as in SQL is similar to rightward assignment.

\[
\text{(1..).lazy.map |{|x| x*2} as x p x.first(10)}
\]

#10 - 04/10/2020 05:17 AM - matz (Yukihiro Matsumoto)

Accepted. I choose \( => \). Some confusing cases should be warned (by the compiler or a cop) e.g.

\[
m{{{a=>b}}}
m (a=>b)
\]

Matz.

#11 - 04/10/2020 07:11 AM - nobu (Nobuyoshi Nakada)
Rightward-assign by ASSOC

[Feature #15921]

#12 - 04/10/2020 07:27 AM - jeremyevans0 (Jeremy Evans)
Cases where => is used outside hashes, arrays, and method call arguments currently are syntax errors. This changes things so that they are not syntax errors, but mean something quite different. I foresee this as a source of future bugs and confusion, especially to new Ruby programmers. Here are some other examples that look similar but act very different:

- `a=>b` vs `[a=>b]
- `{(a => b) => c} vs {p(a => b) => c}
- `->{a=>b}.call vs {a=>b}

Are we going to mark this as experimental (similar to the pipeline operator was when it was introduced last year), or are we sure this syntax will be supported in Ruby 3?

#13 - 04/10/2020 11:35 AM - Eregon (Benoit Daloze)
expr in var already allows rightward assignment:

```ruby
$ ruby -e '(1..).lazy.map { |x| x*2} in x; p x.first(10)
```

Why adding another operator for the same thing?

#14 - 04/16/2020 01:59 PM - Dan0042 (Daniel DeLorme)
Until now I thought => made perfect sense, given that it's already used in rescue, but Jeremy's counterpoint examples are very convincing. There's a high potential for confusion and bugs. Even matz says confusing cases should be warned.

I think => feels natural only because it's preceded by the rescue keyword. That makes it easy to tell apart from other => syntax. This is similar to how pattern matching has syntax very similar to hash literals but you can tell them apart because the pattern is preceded by the in keyword.

So I'd like to tentatively propose |= for rightward assignment. Full proposal at #16794

#15 - 04/20/2020 03:37 PM - Dan0042 (Daniel DeLorme)
I'd like to hear some clarifications on the expected behavior of rightward assignment.

Assignment at both ends? x = expr => y
Multiple assignment? expr => x => y
Splatted assignment? *expr => x,y
Auto-splatted assignment? expr => x,y
With method call chained on next line? expr => x
.foo

#16 - 04/20/2020 05:18 PM - zverok (Victor Shepelev)

In current head, it is so:

```ruby
x1 = 5 + 3 => y1
p [x1, y1] # [8, 8]
5 + 3 => x2 => y2
p [x2, y2] # [8, 8]
*{1, 2, 3} => x3, y3
^ syntax error, unexpected =>, expecting '::' or '. or :: or ']'
[1, 2, 3] => x4, y4
p [x4, y4] # => [1, 2]
5 + 3 => x5
  .then{method(:puts)}
^ syntax error, unexpected '\n', expecting '::' or '. or :: or ']'
```

That mostly makes sense to me (except for case 3, which seems to be "intuitively possible").

03/17/2022 3/6
Ah right, trying the master branch is the fastest way to get answers, duh. But this brings me to this little surprise:

```ruby
5 + 3 => x #=> 8
5 + 3 => x.to_s #=> undefined method `to_s=' for 8:Integer
5 + 3 => x
.to_s #=> undefined method `to_s=' for 8:Integer
```

So these last 2 expressions are equivalent to `x.to_s = 5 + 3` which tries to call the `to_s=` attribute writer. That makes sense but at the same time it really caught me by surprise! It turns out you can also do this with rescue, i.e. `rescue => obj.attr`. You learn something new everything day.

I don't see much examples here, where normal assignment wouldn't work just fine and be as or more readable (`fib(10) => x` in the NEWS seems no better than `x = fib(10)`).

For the if case I would usually just assign in both branches, which generalizes nicely if there is more than one variable to assign from the if.

It seems I'm not alone wonder the usefulness of this change:

https://twitter.com/devoncestes/status/1256222228431228933
https://twitter.com/eregontp/status/1256544073554563072

For me, the main use case is at the end of method chains. Instead of e.g.

```ruby
Word = Struct.new(:text, :count)
words = $stdin.read
    .scan(/[-\w']*+)/
    .group_by{&:downcase}
    .collect { |key, value| Word.new(key, value.count) }
    .sort_by { |w| [-w.text.length, w.text] }
```

we can now write

```ruby
$stdin.read
    .scan(/[-\w']*+)/
    .group_by{&:downcase}
    .collect { |key, value| Word.new(key, value.count) }
    .sort_by { |w| [-w.text.length, w.text] } => words
```

where the order of the code pieces aligns with the order of what's happening, and there's no need to go back with your eyes.

This is an example of a problem given to students, they have to count words in a text and output them in a specific order. It's from a C programming class where I show them after the submission deadline that it's much easier and shorter in Ruby.

I'm not sure if having the assignment on the following line is possible (see below), but I think it should be. I'll submit a feature request if it is not yet possible.

```ruby
$stdin.read
    .scan(/[-\w']*+)/
    .group_by{&:downcase}
    .collect { |key, value| Word.new(key, value.count) }
    .sort_by { |w| [-w.text.length, w.text] }
=> words
```

I'm not sure if having the assignment on the following line is possible (see below), but I think it should be. I'll submit a feature request if it is not yet possible.

It was removed at https://github.com/ruby/ruby/commit/478135f480b4580d068d236f491b2a32048bc193.
Could matz and other committers clarify the motivation to introduce this? There is no pipeline operator currently so it seems of limited usage.

Changing syntax often divides the community (e.g., https://twitter.com/devoncestes/status/125622228431228933), should we be more careful when introducing new syntax? For instance, on syntax issues matz considers to merge, he could state his intention, tweet about the proposal, let it be for at least a month and then decide (merge or not). A blog post by ruby core would be another way to trigger feedback before merging. Often it feels like a decision "out of the blue" with no clear motivation. Discussion after merging feels suboptimal because people realize they have very little chance to change anything and so just love or hate it. Being in master doesn't help much because most people won't try it, yet they can share their opinion based on code snippets using the new syntax. Developers Meeting issues are not suited for extensive discussion, so I have copied these comments here. I agree quite a bit with the suggestion in the second part, but I'm very puzzled with the first part.

The "pipeline operator" is not the only syntactic construct where data flow goes from left to right. Indeed, Ruby's most basic construct, method invocation, leads to a data flow from left to right in the form of method chains. If an R-assign operator is suitable after some pipeline operator(s), it sure should be suitable after a method chain.

I do not have any particularly strong opinion either way, but I would like to point out that the example given by duerst made the most sense to me personally, from all the examples given above as to the potential usefulness. :) I would actually go as far and suggest to include that example, or a similar one, for the rightward-assignment in the official documentation (the content in this issue will otherwise be forgotten eventually, and when people may stumble upon that feature, they may ask "What is this feature used for?").

To be clear, I refer to this specific example by Martin there:

```ruby
$stdin.read
  .scan(/[-\w']+/)
  .group_by(&:downcase)
  .collect { |key, value| Word.new(key, value.count) }
  .sort_by { |w| [-w.text.length, w.text] } => words
```

where the order of the code pieces aligns with the order of what's happening, and **there's no need to go back with your eyes**.

In particular the last explanation made sense to me, and I write this in the sense that I will most likely not use that specific feature (I don't think I need it). I think duerst's explanation was really good - perhaps there are more use cases, but the explanation given there made by far the most sense to me personally.

I think it is necessary to clarify priorities first.

For example,
other operator method call > = right assignment > , || & & other syntax(if, while...)

```ruby
# method call takes precedence
func 42 >>> value    # value = func(42)

# method call takes precedence
func a, b >>> result  # value func(a, b)

# right assignment takes precedence
a || b >>> value      # a || (value = b)

# ?: operator takes precedence
cond ? a : b >>> result  # value = (cond ? a : b)

# method call and ?: operator takes precedence
func cond ? a : b >>> value  # value = func(cond ? a : b)

# right assignment takes precedence
for i in [1, 2, 3] >>> value; end  # for i in value = [1, 2, 3]; end

# right assignment takes precedence
42 >>> value if cond  # value = 42 if cond

Thank you :)
```

**#25 - 01/01/2021 03:03 PM - dgutov (Dmitry Gutov)**

This is not as bad as the "pipeline operator" (which didn't do what its name said), but still, why add this?

It's not like the new syntax makes anything possible that a simple assignment does not.

The "method chains" example is perhaps a counter-example, but having the assignment at the end of the line, after the last segment of the expression, is not so great for readability.

It also looks too much like an annotation ("here's what this expression returns") popularized by xmlfilter (e.g. http://til.justincampbell.me/annotate-ruby-code-in-vim-with-xmpfilter/), and not a real assignment.

**#26 - 01/01/2021 03:13 PM - marcandre (Marc-Andre Lafortune)**

dgutov (Dmitry Gutov) wrote in **#note-25**:

It's not like the new syntax makes anything possible that a simple assignment does not.

It does. It is now a pattern match, not just a rightward assignment:

```ruby
{x: 1, y: 2} => {x:}
# now x holds 1
```