# Ruby trunk - Feature #9076

## New one-argument block syntax: &.

11/04/2013 11:04 PM - asterite (Ary Borenszweig)

| | | |
|---|---|---|
| **Status:** | Feedback | |
| **Priority:** | Normal | |
| **Assignee:** | | |
| **Target version:** | Next Major | |

**Description**

Hello,

I'd like to introduce a new syntax for blocks that have one argument.

Currently you can do this:

```
[1, 2, 3].map &:to_s
```

With the proposed syntax this will be written as:

```
[1, 2, 3].map &.to_s
```

Instead of ":" we use a ".".

The idea is that this new syntax is just syntax sugar that is expanded by the parser to this:

```
[1, 2, 3].map { |arg| arg.to_s }
```

This new syntax allows passing arguments:

```
[1, 2, 3, 4].map &.to_s(2) #=> ["1", "10", "11", "100"]
```

It also allows chaining calls:

```
[1, 10, 100].map &.to_s.length #=> [1, 2, 3]
```

You can also use another block:

```
[[1, -2], [-3, -4]].map &.map &.abs #=> [[1, 2], [3, 4]]
```

Pros:

- Doesn't conflict with any existing syntax, because that now gives a syntax error, so it is available.
- Allows passing arguments and chaining calls
- It's *fast*: it's just syntax sugar. The "&:to_s" is slower because the to_proc method is invoked, you have a cache of procs, etc.
- It looks ok (in my opinion) and allows very nice functional code (like the last example).

Cons:

- Only supports one (implicit) argument. But this is the same limitation of "&:to_s". If you want more than one argument, use the traditional block syntax.
- It's a new syntax, so users need to learn it. But to defend this point, users right now need to understand the &:to_s syntax, which is hard to explain (this calls the "to_proc" method of Symbol, which creates a block... vs. "it's just syntax sugar for")

What do you think?

We are using this syntax in a new language we are doing, Crystal, which has a syntax very similar to Ruby, and so far we think it's nice, simple and powerful. You can read more about it here: http://crystal-lang.org/2013/09/15/to-proc.html

**Related issues:**

| | |
|---|---|
| Related to Ruby trunk - Feature #4146: Improvement of Symbol and Proc | **Rejected** |

**History**

#### #1 - 11/04/2013 11:42 PM - Hanmac (Hans Mackowiak)

my first idea is this:

```
class Symbol
  def call(*args)
    proc {|obj| obj.public_send(self,*args) }
  end
end

[1,2,3,4].map &:to_s.(2) # => ["1", "10", "11", "100"]
```

it is not chainable yet but its a nice beginning (and already valid ruby code)

#### #2 - 11/05/2013 12:55 AM - shevegen (Robert A. Heiler)

I dislike the amount of special meanings that would get associated with & and I already think that & is used too much.

Every new syntax increases the complexity of the language. Rather than extend on &, I'd rather remove the meaning of & in regards to call to_proc anyway - how great it would be if we could remove & altogether from ruby!

I will also tell you what happens once that syntax is in place:

People will come to IRC and ask what the difference is between &:to_s and &.to_s, just as they currently do all the time when they ask "what is the difference between a symbol and a string?"

#### #3 - 11/05/2013 12:57 AM - shevegen (Robert A. Heiler)

Note that if &: would be removed at the same time and &. added I would not mind it that much. You could use & as a reference to a global object, just similar to how $1 or $2 is automagically set for regexes. But to keep two different syntaxes for semi-similar issues is very bad.

#### #4 - 11/05/2013 05:29 AM - fuadksd (Fuad Saud)

Introducing both syntaxes indeed is a bit confusing. I don't think

```
[1, 10, 100].map &.to_s.length
```

is much better than

```
[1, 10, 100].map { |i| i.to_s.length }
```

I think the latter is clearer. I'm fond of having some sugar for positional parameter access (like % in clojure or & in elixir), though.

--

Fuad Saud
Sent with Sparrow (http://www.sparrowmailapp.com/?sig)

#### #5 - 11/05/2013 05:59 AM - nobu (Nobuyoshi Nakada)

*- Status changed from Open to Feedback*

*- Priority changed from Normal to 3*

*- Target version changed from 2.1.0 to Next Major*

You seem confusing that &: is an operator, but it's not.
They are & + : of the beginning of a symbol literal.
To make &.to_s valid, .to_s needs to be valid solely.

#### #6 - 11/05/2013 06:17 AM - asterite (Ary Borenszweig)

nobu (Nobuyoshi Nakada) wrote:

> You seem confusing that &: is an operator, but it's not
> They are & + : of the beginning of a symbol literal.

I know. I don't think I'm confusing them.

> To make &.to_s valid, .to_s needs to be valid solely.

I don't see why ".to_s" needs to be valid solely.

In my proposal the new keyword is &.. The parser needs to be modified so that when you have a block argument, if a . comes after the & (something

which is currently illegal) then it tries to parse a method call after the dot, and then it transforms the block argument to a regular block.

#### #7 - 11/05/2013 06:57 AM - alexeymuranov (Alexey Muranov)

In my opinion, this is a bad idea: there would be a dot ., an ampersand &, and an ampersand-dot &., unrelated to either of the two.

What is wrong with [1, 2, 3, 4].map{|x| x.to_s(2)}?

#### #8 - 11/05/2013 11:59 AM - asterite (Ary Borenszweig)

alexeymuranov (Alexey Muranov) wrote:

> In my opinion, this is a bad idea: there would be a dot ., an ampersand &, and an ampersand-dot &., unrelated to either of the two.
>
> What is wrong with [1, 2, 3, 4].map{|x| x.to_s(2)}?

The "x" variable is just noise. What you want to do in that line of code is "map every element with a call to to_s(2)". With an "x" what you read is "let x be each element of this array, then invoke to_s(2) on it...". The first sentence is shorter and clearer on the mind.

This is why people tend to do &:to_s instead of {|x| x.to_s}. The later just adds noise.

However, what does a Symbol has to do there? Why doesn't it work with a String, like &"to_s"? Well, because it's just a hack. It's an abuse of the language. And that abuse not only comes with a hard explanation (it calls the to_proc method on Symbol to turn it into a Proc) but also with a performance penalty.

By using &.to_s you get 1. no performance penalty, 2. a chainable syntax, 3. an easier way to explain it (it's just syntax sugar for a single argument block, where you invoke a method on it).

But maybe its too late to add it now to the language... nobody seems to like it or see its advantages.

#### #9 - 11/05/2013 06:01 PM - Hanmac (Hans Mackowiak)

i got a working sample with .()

```
class Symbol
  class SymbolHelper < BasicObject
    def initialize(obj,methId,*args,&blk)
      @obj    = obj
      @methId = methId
      @args   = args
      @blk    = blk
    end
    def method_missing(methId,*args,&blk)
      return SymbolHelper.new(self,methId,*args,&blk)
    end

    def to_proc
      ::Kernel::proc {|obj| (@obj == nil ? obj : @obj.to_proc.(obj)).public_send(@methId,*@args,&@blk) }
    end
  end

  def call(*args,&blk)
    return SymbolHelper.new(nil,self,*args,&blk)
  end
end

[1,2,3,4,5].map(&:to_s.(2)) #=> ["1", "10", "11", "100", "101"]

[1,2,3,4,5].map(&:to_s.(2).size) # => [1, 2, 2, 3, 3]

[[1,2,3],[1,3],[],[2]].map(&:any?.(&:even?)) #=> [true, false, false, true]
```

i hope that suits you guys

Edit: added blk

#### #10 - 11/05/2013 08:08 PM - asterite (Ary Borenszweig)

Hanmac: thanks for the code to make it work. Ruby is very powerful.

However, I'm sure that code is very slow. At least slower than writing a block. That's why I'm not interested in any code that can make it work, because it will always be slower than syntax sugar.

#### #11 - 11/06/2013 06:39 AM - alexeymuranov (Alexey Muranov)

Ary,

as far as i understand, the ampersand is used with symbols and not with strings because method names and identifiers are symbols and not strings. What follows after the colon is not the symbol's "content," but the symbol's "label" or "name." Replacing a symbol's name with any other unique name should not change the "meaning" of the program.

Arbitrarily named variables are always "noise," they are price for intuitive and notational simplicity. IMO, the only alternatives to using arbitrarily named variables are the following.

1. Use predefined implicitly bound variables, or "placeholders," similar to the meaning of & in you proposal. To go further, why not to introduce &1 or a1 for the first argument, &2 or a2 for the second, etc.? Then you would also be able to do, with your proposal:

```
[1, 2, 3, 4].reduce &1 + &2
```

I do not think this would be a great idea. But i agree that one-argument functions are in some sense special.

Your proposal seems to break existing rules for block syntax: curly braces or do..end, which always designate a block, are absent in your case.

Using "implicitly bound variable &" would not generalize well to more complicated situations and would be confusing: how to be with

```
[1, 2, 3, 4].map 2 * &
[1, 2, 3, 4].map &.foo(&)
[1, 2, 3, 4].map &.foo([5, 6].map &.bar)
```

?

1. Use combinators like in combinatory logic (http://plato.stanford.edu/entries/logic-combinatory/). It should be possible to introduce various operations of composition on procs and methods (see #6284 for example) to avoid using variables at all, but it will be verbose and not syntactic sugar. This is clearly not what you are proposing.

2. Instead of writing text, draw graphs and represent argument bindings by arrows.

It looks to me like you are suggesting to use a random syntactic sugar for a random special case. Apart from using & instead of an arbitrarily named variable, the main difference you introduce seems to be the absence of curly braces. This looks to me more like an inconsistency than like an advantage.

#### #12 - 11/06/2013 09:53 PM - asterite (Ary Borenszweig)

Alexey,

You are right about every point you make. It's indeed a random syntactic sugar for a special case. It only happens that that special case happens very often. Otherwise Symbol#to_proc, &:to_s, wouldn't exist.

Note that in doing array.map &:to_s the do ... end and curly braces are also missing. However, the & signals a block, just as when you do foo &block. This is no different than foo &.something where, again, the & signals a block.

However, you are right that a more powerful solution allowing you to refer to any of the block's arguments, not just the first one, would be much nicer. But I think this is harder because when you have nested blocks then how you refer to arguments in the parent block? A syntax like &&1 comes to my mind... Mmm... Just kidding :-)

#### #13 - 11/06/2013 10:44 PM - alexeymuranov (Alexey Muranov)

> Note that in doing array.map &:to_s the do ... end and curly braces are also missing. However, the & signals a block, just as when you do foo &block. This is no different than foo &.something where, again, the & signals a block.

Ary, in array.map &:to_s curly braces are missing because there is no literal block definition, the block is the result of the & operator applied to a symbol.

The main problem IMO with your proposed syntactic sugar for the common special case is that it adds a completely new syntactic rule to Ruby, and also breaks one or more of existing ones. Normally in Ruby

```
<method_name1> <identifier1>.<method_name2>.<method_name3>
```

means: "call the method named by <method_name2> on the object named by <identifier1> or value returned by the method <identifier1>, then call the method named <method_name3> on the result, then yield the result as the argument to a call of the method named by <method_name1>"

(I am not a specialist, i am not sure i am using all the terms correctly.)

It seems to me that what you are looking for is probably a shorter notation for a one-argument lambda or block. I personally doubt that there is much space in Ruby syntax to introduce it.

Skipping the curly braces does not look to me like a benefit and may make the syntax ambiguous or not flexible.

The performance penalty of the & operator probably can be worked around by compiling the code.

Avoiding the hassle of naming the bound variable IMHO is not ... worth the hassle. :) Well, if i really wanted that myself, i might have proposed something like

```
[1, 2, 3, 4].map { ..to_s(2) }
[1, 2, 3, 4].map { ..foo([5, 6].map { ..bar }) }  # the argument is shadowed
```

o_O

*Edited 2013-11-09*

#### #14 - 11/06/2013 11:22 PM - asterite (Ary Borenszweig)

alexeymuranov (Alexey Muranov) wrote:

> Note that in doing array.map &:to_s the do ... end and curly braces are also missing. However, the & signals a block, just as when you do foo &block. This is no different than foo &.something where, again, the & signals a block.

Ary, in array.map &:to_s curly braces are missing because there is no literal block definition, the block is the result of the & operator applied to a symbol.

I know.

Did you know that you can't do &:to_s wherever you want?

```
irb(main):001:0> &:to_s
SyntaxError: (irb):1: syntax error, unexpected tAMPER
&:to_s
 ^
irb(main):002:0> a = &:to_s
SyntaxError: (irb):2: syntax error, unexpected tAMPER
a = &:to_s
     ^
```

That means, Ruby only recognizes &<expression> as the last argument to a call. That also means that the & operator can only mean a block, somehow. Similarly, &. will mean a block with the semantics I already explained.

> The main problem IMO with your proposed syntactic sugar for the common special case is that it adds a completely new syntactic rule to Ruby, and also breaks one or more of existing ones.

No, it doesn't break anything because right now that &.to_s gives syntax error, which means that syntax is available for defining new meanings.

Normally in Ruby

> `<method_name1> <identifier1>.<method_name2>.<method_name3>`
>
> means: "call the method named by on the object named by or value returned by the method , then call the method named on the result, then yield the result as the argument to a call of the method named by "
>
> (I am not a specialist, i am not sure i am using all the terms correctly.)
>
> It seems to me that what you are looking for is probably a shorter notation for a one-argument lambda. I personally doubt that there is much space in Ruby syntax to introduce it.

As I said, there *is* space in Ruby syntax for it, precisely because right now it's a syntax error.

> Skipping the curly braces does not look to me like a benefit and may make the syntax ambiguous or not flexible.

When you do map &:to_s you are skipping the curly braces.

> The performance penalty of the & operator probably can be worked around by compiling the code.

What do you mean?

I think this can be done by Ruby, yes: just make a.map &:to_s be the same as a.map { |x| x.to_s } by the parser... I think they are discussing similar things to do related to frozen strings.

#### #15 - 11/06/2013 11:32 PM - alexeymuranov (Alexey Muranov)

asterite (Ary Borenszweig) wrote:

> Did you know that you can't do &:to_s wherever you want?
>
> ```
> irb(main):001:0> &:to_s
> SyntaxError: (irb):1: syntax error, unexpected tAMPER
> &:to_s
>  ^
> ```

Yes, this is because blocks do not exist as objects in Ruby, they appear and are evaluated or captured in procs during method calls. So the result of & cannot be stored in a variable, it has to be run or converted to a proc.

> When you do map &:to_s you are skipping the curly braces.

I am not skipping curly braces, i am just not defining any block, it is obtained by the & operator.

> > The performance penalty of the & operator probably can be worked around by compiling the code.
>
> What do you mean?
>
> I think this can be done by Ruby, yes: just make a.map &:to_s be the same as a.map { |x| x.to_s } by the parser... I think they are discussing similar things to do related to frozen strings.

Yes, this is what i mean.

#### #16 - 03/02/2014 12:09 PM - sowieso (So Wieso)

I think this would be a really great idea.
Symbol#to_proc is technically a nice solution, but not nice from the esthetically viewpoint. Just have a look how many people are confused by this. &.a_method makes immediately clear that here a method call is happening. So & must be a (special) object. Context makes pretty clear which object that is, even if you do not know this syntax rule.
I agree that having two solutions is not nice, but only because we implemented a weak solution we should not restrict ourselves to it as there are mightier and more readable ones.

I'd like to remark, that getting a solution that solves this issue once and for all in official ruby would be much nicer than the current half-hearted .to_proc hack. There are many projects in the Internet that tried to solve this, thus demand is given. Let's unify them!
https://github.com/rapportive-oss/ampex
https://github.com/danielribeiro/RubyUnderscore
https://github.com/raganwald/homoiconic/blob/master/2012/05/anaphora.md
https://bugs.ruby-lang.org/issues/8987 (my request)

#### #17 - 03/02/2014 04:11 PM - phluid61 (Matthew Kerwin)

I share concerns that have been voiced earlier in the thread.

This code snippet: foo &.bar *looks* like you're either passing &.bar as the first positional parameter to foo, or casting .bar to a Proc and passing it as the block parameter. You might argue that that *is* what you're doing, but it's not; .bar isn't a thing that can be #to_proc'd, and & isn't an object you can send method calls. What we end up doing is confusing the syntax, adding a third option which looks like a hybrid of the others, but is something else again.

I think the ampex gem better captures the intent here by both using the & sigil/operator to clearly indicate that Proc->block magic is happening, and by providing an explicit object to receive the method calls. Of course it could never be promoted to core, because the name 'X' is far too valuable and I doubt anyone could come up with a better one, but personally I'm happy enough that the gem exists and can be used by those to whom it would be of benefit.

And if it's too slow for you, write out the full code, even if that means creating a throw-away variable in your block. We like variables, they show us what our code is doing. I doubt it's a goal of the language to remove them.

#### #18 - 03/02/2014 10:37 PM - sowieso (So Wieso)

Matthew Kerwin wrote:

> I share concerns that have been voiced earlier in the thread.
>
> This code snippet: foo &.bar *looks* like you're either passing &.bar as the first positional parameter to foo, or casting .bar to a Proc and passing it as the block parameter. You might argue that that *is* what you're doing, but it's not; .bar isn't a thing that can be #to_proc'd, and & isn't an object you can send method calls. What we end up doing is confusing the syntax, adding a third option which looks like a hybrid of the others, but is something else again.

You are totally right, this is yet another use for &. But if you take the new rule, it is not really confusing, just parse it like explained when you see & followed by a dot. And you still have the & warning you: here is something blockish going on.

> I think the ampex gem better captures the intent here by both using the & sigil/operator to clearly indicate that Proc->block magic is happening, and by providing an explicit object to receive the method calls. Of course it could never be promoted to core, because the name 'X' is far too valuable and I doubt anyone could come up with a better one, but personally I'm happy enough that the gem exists and can be used by those to whom it would be of benefit.

I agree, X is a no-go. Wouldn't any symbol (in ascii) be possible? (map &@.to_s, or even map @.to_s)

> And if it's too slow for you, write out the full code, even if that means creating a throw-away variable in your block. We like variables, they show us what our code is doing. I doubt it's a goal of the language to remove them.

I disagree here. The usual one-letter-variables in real code do not show anything. This implementation would still force us to give them a name if we want to use them more than once, which is a compromise on a good level.

What if we do it like this?
[1,2,3,4].map{.to_s(2)}.reverse
=> ["100", "11", "10", "1"]
When there is no receiver for a method-call (can only be the first method-call in a block), send the message to yielded argument.

#### #19 - 03/03/2014 01:01 AM - phluid61 (Matthew Kerwin)

On 3 March 2014 08:37, sowieso@dukun.de wrote:

> Issue #9076 has been updated by So Wieso.
>
> Matthew Kerwin wrote:
>
> > I share concerns that have been voiced earlier in the thread.
> >
> > This code snippet: foo &.bar *looks* like you're either passing
> > &.bar as the first positional parameter to foo, or casting .bar to a
> > Proc and passing it as the block parameter. You might argue that that *is*
> > what you're doing, but it's not; .bar isn't a thing that can be
> > #to_proc'd, and & isn't an object you can send method calls. What we end
> > up doing is confusing the syntax, adding a third option which looks like a
> > hybrid of the others, but is something else again.
>
> You are totally right, this is yet another use for &. But if you take the
> new rule, it is not really confusing, just parse it like explained when you
> see & followed by a dot.

I think it's pretty confusing, and some of the conversation upthread seems to agree. I don't like having to carefully read and parse code to see if it's & or . or &. -- I'm not a computer, I don't read that way.

> I think the ampex gem better captures the intent here by both using the
> & sigil/operator to clearly indicate that Proc->block magic is happening,
> and by providing an explicit object to receive the method calls. Of course
> it could never be promoted to core, because the name 'X' is far too
> valuable and I doubt anyone could come up with a better one, but personally
> I'm happy enough that the gem exists and can be used by those to whom it
> would be of benefit.
>
> I agree, X is a no-go. Wouldn't any symbol (in ascii) be possible? (map
> &@.to_s, or even map @.to_s)

It may be possible, but I don't like Perl's sigil-heavy and magic-variable-creating voodoo, and I'd much prefer Ruby to keep it limited to what it already has. I think X is perfectly apt for the gem; there's a very strong convention that 'i' is the index in a loop and 'x' is the item, so having a globally defined X is sensible, it's just a bit too broad of a brush for the core.

And @ (and @@) are class scoping sigils, so they're not appropriate here.

> > And if it's too slow for you, write out the full code, even if that
> > means creating a throw-away variable in your block. We like variables, they
> > show us what our code is doing. I doubt it's a goal of the language to

> remove them.

> I disagree here. The usual one-letter-variables in real code do not show anything. This implementation would still force us to give them a name if we want to use them more than once, which is a compromise on a good level.

They show us several things: precisely where the variable comes from (again, I can't stand Perl's $_ just appearing -- or worse, changing -- in random places), precisely where and how it's used, and if it's called "x" it also strongly suggests that it's the item in an iteration that doesn't have an particular meaning outside the context of the iteration (unless we're doing something with coordinates, in which case context should be enough to inform the difference).

> What if we do it like this?
> [1,2,3,4].map{.to_s(2)}.reverse
> => ["100", "11", "10", "1"]
> When there is no receiver for a method-call (can only be the first method-call in a block), send the message to yielded argument.

I think this has been proposed before. You'd have to clearly enumerate edge cases (e.g.: def each() yield; yield 1, 2; end), and it still doesn't look great.

To summarise my opinion: I agree that there's a lot of value in being able to brain-dump the equivalent of foo.map{|x|x.to_s(2)} without having to break your train of thought, but I don't think &. is the right way.

--
Matthew Kerwin
http://matthew.kerwin.net.au/

#### #20 - 06/06/2014 05:24 AM - nobu (Nobuyoshi Nakada)

*- Related to Feature #4146: Improvement of Symbol and Proc added*

#### #21 - 06/06/2014 05:26 AM - nobu (Nobuyoshi Nakada)

*- Description updated*