Ruby - Feature #3163

SyntaxError when using variable which is also a method in current scope with a Symbol argument

04/18/2010 03:26 AM - Eregon (Benoit Daloze)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	2.6	

Description

=begin
Hi,
Here is a simple example:

irb for ruby-1.9.2-r27362

```
print = 0
print :symbol
SyntaxError: (irb):2: syntax error, unexpected ':', expecting $end
print :symbol
^
from .../bin/irb:17:in `'
```

Why does this generate a SyntaxError ?
It doesn't if you use parentheses of courses: print(:symbol)

But it shouldn't fail as long as these methods are not keywords.
This mean that *any* method which in current scope is also a variable,
and accept a symbol as argument (else you would not call it with a symbol), cause this issue:

```
def render(*args)
puts "in render"
end
render = 1
render :a
SyntaxError: (irb):5: syntax error, unexpected ':', expecting $end
render :a
^
```

and this cause it too:

```
render :a => "b"
SyntaxError:...
```

(and "method param: value" does not works either)

I ran a few times in this bug, while using some "p :done"(and having a local var p) to trace the program execution quickly.

Sorry if this has already been reported, but I didn't see.
I think the use of "method :symbol" or "method" is high, so this can be really frustrating to need to use parentheses.
=end

History

#1 - 04/18/2010 09:06 AM - coatl (caleb clausen)

=begin
I *think* this behavior is correct. At any rate, the behavior is the same all the way back to 1.8.6. However, this is a confusing part of parsing ruby, so I may be remembering wrong.

If I remember right, it goes something like this: ambiguous characters (such as '%' and ':') which could be operators or the start of literals are always treated as operators if preceded by a variable, no matter what whitespace precedes or follows them. Constants do behave the way you are expecting.
=end

#2 - 04/18/2010 09:23 AM - Eregon (Benoit Daloze)

=begin
On 18 April 2010 02:06, caleb clausen redmine@ruby-lang.org wrote:

> Issue #3163 has been updated by caleb clausen.
>
> I *think* this behavior is correct. At any rate, the behavior is the same all the way back to 1.8.6. However, this is a confusing part of parsing ruby, so I may be remembering wrong.
>
> Yes, this behavior is not new. And it's confusing, in fact.
>
> If I remember right, it goes something like this: ambiguous characters (such as '%' and ':') which could be operators or the start of literals are always treated as operators if preceded by a variable, no matter what whitespace precedes or follows them. Constants do behave the way you are expecting.

I can understand easily operators can be confusing for the parser, but I didn't know ':' is an operator.

And the easiest way to resolve the ambiguity variable/methods wouldn't be to look for '=' or '.' following a variable ?

I might be highly wrong, my apologies for the noise if I am,

Regards,
B.D.

On 18 April 2010 02:06, caleb clausen <redmine@ruby-lang.org> wrote:

> Issue #3163 has been updated by caleb clausen.
>
> I _think_ this behavior is correct. At any rate, the behavior is the same all the way back to 1.8.6. However, this is a confusing part of parsing ruby, so I may be remembering wrong.

Yes, this behavior is not new. And it's confusing, in fact.

> If I remember right, it goes something like this: ambiguous characters (such as '%' and ':') which could be operators or the start of literals are always treated as operators if preceded by a variable, no matter what whitespace precedes or follows them. Constants do behave the way you are expecting.

I can understand easily operators can be confusing for the parser, but I didn't know ':' is an operator.

And the easiest way to resolve the ambiguity variable/methods wouldn't be to look for '=' or '.' following a variable ?

I might be highly wrong, my apologies for the noise if I am,

Regards,
B.D.

=end

#3 - 04/18/2010 09:54 AM - murphy (Kornelius Kalnbach)

=begin
On 18.04.10 02:23, Benoit Daloze wrote:

> I can understand easily operators can be confusing for the parser, but I
> didn't know ':' is an operator.
> as in a ? b : c.

=end

#4 - 04/18/2010 12:08 PM - murphy (Kornelius Kalnbach)

=begin
On 18.04.10 04:34, Caleb Clausen wrote:

> In my judgment, this would be too much additional complication in an area of the parser/lexer that's already extremely squirelly.
> Ironically, it could still make the language easier to use for humans.
> Because our eyes are not yacc.

And maybe our eyes are on the right track here. I don't think this should be valid:

```
print = 0
foo ? print :bar
```

because the : obviously belongs to :bar. I doubt there's an editor on this planet which realizes that bar is not a symbol.

Ruby should instead throw a syntax error, unexpected $end, expecting ':'. In other words, an unevenly-spaced colon immediately followed by an identifier (as in " :bar") should never be interpreted as a ternary operator colon.

An improved even-spaces rule for the ternary operator might really help.

[murphy]

=end

#5 - 04/18/2010 12:57 PM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

- % Done changed from 0 to 100

=begin
This issue was solved with changeset r27388.
Benoit, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

=end

#6 - 04/18/2010 02:02 PM - coatl (caleb clausen)

=begin
OK. But now what about similar ambiguous cases, such as:

```
p=0; p %[foo]
p=0; p /foo/x
p=0; p &foo
p=0; p *foo
p=0; p ?f : g
p=0; p <<foo
123
foo
```

In each of the above lines, the 2nd p is treated as a variable, as is traditional. Shouldn't these cases also be changed to allow the 2nd p to be a method call? Or is there to be a special case just for colon here?
=end

#7 - 04/18/2010 02:26 PM - murphy (Kornelius Kalnbach)

=begin
On 18.04.10 07:07, Caleb Clausen wrote:

> The small improvement in readability didn't seem worth the trouble to me. But to my surprise, nobu just went ahead and implemented it. I'm still wondering about the general case.
> I'm surprised, too. I thought at most, a change would be made in Ruby 2.0. My argument was pro-discussion, not pro-change.

> > An improved even-spaces rule for the ternary operator might really help.
> Can you be specific? I vaguely recall there being some unique special
> cases around ternaries, but the details elude me.
> What I meant was a behavior like this one:

ruby -wce 'p /4'
warning: ambiguous first argument; put parentheses or even spaces

As far as I understood, the idea behind this is that an ambiguous symbol is an operator if it's used in the form "a:b" or "a : b", and a prefix / opening delimiter if it is used in the form "a :b" or "a(:b)". I have no specific idea how to minimize surprise while retaining backwards compatibility. But the more code example we have, the better we can decide. Your examples are great:

```
p=0; p %[foo]
p=0; p /foo/x
p=0; p &foo
p=0; p *foo
p=0; p ?f : g
p=0; p <<foo
123
foo
```

In addition:

```
p=0; p +foo
p=0; p -foo
```

If we really change syntax, all of these should be interpreted as p(...).

But it might not be a good idea to change it suddenly in 1.9.2.

[murphy]

=end

#8 - 04/18/2010 02:45 PM - murphy (Kornelius Kalnbach)

=begin
On 17.04.10 20:26, Benoit Daloze wrote:

> I ran a few times in this bug, while using some "p :done"(and having
> a local var p) to trace the program execution quickly.
> Translated into a Ruby style guide haiku:

Never use p
As a local variable;
It breaks.

[murphy]

=end

#9 - 04/18/2010 11:10 PM - Eregon (Benoit Daloze)

=begin

> as in a ? b : c.
> Oups, I always forget that one. But to my defense it's only an operator if
> preceded by ?

On 18 April 2010 07:44, Kornelius Kalnbach murphy@rubychan.de wrote:

> On 17.04.10 20:26, Benoit Daloze wrote:
>
> > I ran a few times in this bug, while using some "p :done"(and having
> > a local var p) to trace the program execution quickly.
> > Translated into a Ruby style guide haiku:

> Never use p
> As a local variable;
> It breaks.
>
> [murphy]

Sure, but that's why I showed that any variable which is also a method in a scope can cause this problem.
That's right, I should really not use p, it's just because that's what happened to me most of the time.

```
class Foo
def bar(*args)
:bar
end

def main
bar = 2
# ...
bar my: "hash"
end
end
Foo.new.main
```

I myself agree it's often not a good idea to name a variable with the name of a method you will use there without () and with an argument like a literal Symbol/Hash. And the names of the methods usually are not appropriate for the name of a variable.

But, it's not because this shouldn't appear too much it shouldn't be fixed.

Also in this case, the error is very disturbing:

> p = 0; p %Q{a}
> NoMethodError: undefined method `Q' for main:Object

Let's take your examples:

```
p = 0
p %[foo]
p /foo/x
p &foo
p *foo
p ?f : g
p <<foo
```

They are all raising "NameError: undefined local variable or method ‘foo’ for main:Object" except the one with the ternary condition.

I think in all these cases, it should be considered as an operator only if there is a space after, because you use a space before.
If there is no space, then it should be an operator.
If there is more spaces at left than right of the 'operator' it should be a method.

```
p % [a] # operator
p %[a] # method
p%[a] # operator
```

This idea is valid only if the right part is a literal expression:
p % a , p %a , p%a are in all cases operators.

Would it be possible to implement a rule like that:
"if more spaces at left than right and right is a literal expression, consider left as a method" (instead of always as an operator)

Are you thinking the same way to consider theses expressions ?

This change is only an improvement to my opinion, so I don't see when it can cause problems.

And the most important, thank you, Nobu, for resolving the main issue of this bug :)

>as in a ? b : c.
Oups, I always forget that one. But to my defense it's only an operator if preceded by ?

On 18 April 2010 07:44, Kornelius Kalnbach <murphy@rubychan.de> wrote:

> On 17.04.10 20:26, Benoit Daloze wrote:
> > I ran a few times in this bug, while using some "p :done"(and having
> > a local var p) to trace the program execution quickly.
>
> Translated into a Ruby style guide haiku:
>
> Never use p
> As a local variable;
> It breaks.
>
> [murphy]

Sure, but that's why I showed that any variable which is also a method in a scope can cause this problem.
That's right, I should really not use p, it's just because that's what happened to me most of the time.

```
class Foo
  def bar(*args)
    :bar
  end

  def main
    bar = 2
    # ...
    bar my: "hash"
  end
end
Foo.new.main
```

I myself agree it's often not a good idea to name a variable with the name of a method you will use there without () and with an argument like a literal Symbol/Hash. And the names of the methods usually are not appropriate for the name of a variable.

But, it's not because this shouldn't appear too much it shouldn't be fixed.

Also in this case, the error is very disturbing:
> p = 0; p %Q{a}
NoMethodError: undefined method `Q' for main:Object

Let's take your examples:

```
p = 0
p %[foo]
p /foo/x
p &foo
p *foo
p ?f : g
p <<foo
```

They are all raising "NameError: undefined local variable or method ‘foo’ for main:Object" except the one with the ternary condition.

I think in all these cases, it should be considered as an operator only if there is a space after, because you use a space before.
If there is no space, then it should be an operator.
If there is more spaces at left than right of the 'operator' it should be a method.

```
p % [a] # operator
p %[a] # method
p%[a] # operator
```

This idea is valid only if the right part is a literal expression:
p % a , p %a , p%a are in all cases operators.

Would it be possible to implement a rule like that:

"if more spaces at left than right and right is a literal expression, consider left as a method" (instead of always as an operator)

Are you thinking the same way to consider theses expressions ?

This change is only an improvement to my opinion, so I don't see when it can cause problems.

And the most important, thank you, Nobu, for resolving the main issue of this bug :)

=end

#10 - 04/19/2010 02:19 AM - murphy (Kornelius Kalnbach)

=begin
On 18.04.10 16:10, Benoit Daloze wrote:

> Sure, but that's why I showed that any variable which is also a method
> in a scope can cause this problem.
> I agree that it's a problematic part of Ruby's syntax. I think it comes
> partly from method-call parentheses being optional. It's a trade-off.

> If there is more spaces at left than right of the 'operator' it should
> be a method.
> p % [a] # operator
> p %[a] # method
> p%[a] # operator
> +1. I think this rule should ony distinguish "no space" and "one ore
> more spaces". Otherwise, we'd have to start counting spaces. Fun for the
> next obfuscation contest.

> This idea is valid only if the right part is a literal expression:
> p % a , p %a , p%a are in all cases operators.
> I'm not sure whether the lexer can look ahead this far.

> Would it be possible to implement a rule like that:
> "if more spaces at left than right and right is a literal expression,
> consider left as a method" (instead of always as an operator)
> As far as I understood nobus patch, it does exactly that.

> Are you thinking the same way to consider theses expressions ?
> I think everything should be evaluated "intuitively", whatever this
> means ;) The rules you outlined seem much more intuitive to me.

> This change is only an improvement to my opinion, so I don't see when it
> can cause problems.
> Incompatibility is a problem. I wouldn't start to write code that's only
> valid in Ruby 1.9.2, because 1.8.7 is so much more popular.

But it doesn't seem to be a problem yet. I checked the syntax of 20K
Ruby files in 300 gems before and after nobu's patch. The diff is
attached. Only obscure code (like Caleb's rubylexer examples ;) and ERB
templates (which are invalid anyway) seem to be hit.

[murphy]

Attachment: syntax-check-r27387vs27388.diff
=end

#11 - 04/19/2010 07:17 AM - Eregon (Benoit Daloze)

=begin
On 18 April 2010 23:46, Caleb Clausen vikkous@gmail.com wrote:

> On 4/18/10, Kornelius Kalnbach murphy@rubychan.de wrote:
>
> > On 18.04.10 16:10, Benoit Daloze wrote:

> > If there is more spaces at left than right of the 'operator' it should
> > be a method.
> > p % [a] # operator
> > p %[a] # method
> > p%[a] # operator
> > +1. I think this rule should ony distinguish "no space" and "one ore
> > more spaces". Otherwise, we'd have to start counting spaces. Fun for the
> > next obfuscation contest.
>
> I agree as well that it should be presence/absence of spaces, not the
> number.
> While we're at it, let's not forget about this case:

Me too, I just thought as one or zero space, so basically presence/absence

> p% [a] # operator
>
> The governing rule is what I call the before-but-not-after rule:
>
> Ambiguous operators are treated as (the beginning of) literals instead
> of operators if they follow what looks like a method call and there is
> whitespace before but not after them.
>
> What nobu has done is in keeping with that rule, just refining what
> 'looks like a method call' a little, in the case of variable/method
> collisions.
>
> > This idea is valid only if the right part is a literal expression:
> > p % a , p %a , p%a are in all cases operators.
> > I'm not sure whether the lexer can look ahead this far.
>
> Yeah, at first glance, I'd say that trying to determine what
> (nonwhitespace) token follows the operator is too much extra
> complication. My implementation just looks at the next character to
> see if it is whitespace. Judging by its behavior, that's what MRI does
> as well.
>
> > Would it be possible to implement a rule like that:
> > "if more spaces at left than right and right is a literal expression,
> > consider left as a method" (instead of always as an operator)
> > As far as I understood nobus patch, it does exactly that.
>
> For the most part, this rule is already implemented by ruby. Murphy
> and I have expressed our two reservations above. I prefer the
> before-but-not-after rule as I formulated above; do you have any
> quibbles with that, Benoit?

Not as far as I understand your rule, the main idea is the same.

> > This change is only an improvement to my opinion, so I don't see when it
> > can cause problems.
> > Incompatibility is a problem. I wouldn't start to write code that's only
> > valid in Ruby 1.9.2, because 1.8.7 is so much more popular.
>
> But it doesn't seem to be a problem yet. I checked the syntax of 20K
> Ruby files in 300 gems before and after nobu's patch. The diff is
> attached. Only obscure code (like Caleb's rubylexer examples ;) and ERB
> templates (which are invalid anyway) seem to be hit.
>
> Ha! glad to see my testcases made the language squeak again. Thanks
> for testing that, murphy. Looks like from that evidence there's little
> enough cause to worry about creating incompatibilities; I certainly am
> not concerned about any of the code I wrote.

> But it might not be a good idea to change it suddenly in 1.9.2.

Yeah, isn't there supposed to be a feature freeze right now?

Let me be happy my Ruby works better with that :)

Seriously, what is done and good should be kept, at anytime, if accepted.

Thanks for interesting discussion on this thread,

Regards,
B.D.

On 18 April 2010 23:46, Caleb Clausen <vikkous@gmail.com> wrote:

> On 4/18/10, Kornelius Kalnbach <murphy@rubychan.de> wrote:
> > On 18.04.10 16:10, Benoit Daloze wrote:
>
> >> If there is more spaces at left than right of the 'operator' it should
> >> be a method.
> >> p % [a] # operator
> >> p %[a] # method
> >> p%[a] # operator
> > +1. I think this rule should ony distinguish "no space" and "one ore
> > more spaces". Otherwise, we'd have to start counting spaces. Fun for the
> > next obfuscation contest.
>
> I agree as well that it should be presence/absence of spaces, not the number.
> While we're at it, let's not forget about this case:

Me too, I just thought as one or zero space, so basically presence/absence

> p% [a] # operator
>
> The governing rule is what I call the before-but-not-after rule:
>
> Ambiguous operators are treated as (the beginning of) literals instead
> of operators if they follow what looks like a method call and there is
> whitespace before but not after them.
>
> What nobu has done is in keeping with that rule, just refining what
> 'looks like a method call' a little, in the case of variable/method
> collisions.
>
> >> This idea is valid only if the right part is a literal expression:
> >> p % a , p %a , p%a are in all cases operators.
> > I'm not sure whether the lexer can look ahead this far.
>
> Yeah, at first glance, I'd say that trying to determine what
> (nonwhitespace) token follows the operator is too much extra

> complication. My implementation just looks at the next character to
> see if it is whitespace. Judging by its behavior, that's what MRI does
> as well.
>
> >> Would it be possible to implement a rule like that:
> >> "if more spaces at left than right and right is a literal expression,
> >> consider left as a method" (instead of always as an operator)
> > As far as I understood nobus patch, it does exactly that.
>
> For the most part, this rule is already implemented by ruby. Murphy
> and I have expressed our two reservations above. I prefer the
> before-but-not-after rule as I formulated above; do you have any
> quibbles with that, Benoit?

Not as far as I understand your rule, the main idea is the same.

> >> This change is only an improvement to my opinion, so I don't see when it
> >> can cause problems.
> > Incompatibility is a problem. I wouldn't start to write code that's only
> > valid in Ruby 1.9.2, because 1.8.7 is so much more popular.
> >
> > But it doesn't seem to be a problem yet. I checked the syntax of 20K
> > Ruby files in 300 gems before and after nobu's patch. The diff is
> > attached. Only obscure code (like Caleb's rubylexer examples ;) and ERB
> > templates (which are invalid anyway) seem to be hit.
>
> Ha! glad to see my testcases made the language squeak again. Thanks
> for testing that, murphy. Looks like from that evidence there's little
> enough cause to worry about creating incompatibilities; I certainly am
> not concerned about any of the code I wrote.
>
> > But it might not be a good idea to change it suddenly in 1.9.2.
>
> Yeah, isn't there supposed to be a feature freeze right now?

Let me be happy my Ruby works better with that :)

Seriously, what is done and good should be kept, at anytime, if accepted.

Thanks for interesting discussion on this thread,

Regards,
B.D.

=end

#12 - 04/19/2010 02:05 PM - nobu (Nobuyoshi Nakada)

Hi,

At Sun, 18 Apr 2010 14:26:00 +0900,
Kornelius Kalnbach wrote in [ruby-core:29594]:

> > The small improvement in readability didn't seem worth the trouble to
> > me. But to my surprise, nobu just went ahead and implemented it. I'm
> > still wondering about the general case.
> I'm surprised, too. I thought at most, a change would be made in Ruby
> 2.0. My argument was pro-discussion, not pro-change.

In general, ruby's binary operators need spaces in both side to
be balanced. So I think they are unintended exceptions, in

short mere bugs.

--
Nobu Nakada

#13 - 04/20/2010 02:59 AM - coatl (caleb clausen)

Nobu, can you please comment on those other examples of ambiguous operators I posted above? For example:

```
p=0; p %[foo]
```

Currently, the second p is a variable and %[foo] is an operator and an array, but to be consistent with the change you just made, the second p should be a method, and %[foo] should be a string. Should that and the other ambiguous operators be parsed as they currently are, or in the same way that 'p=0; p :foo' now works?

#14 - 04/21/2010 01:37 PM - nobu (Nobuyoshi Nakada)

- Status changed from Closed to Open

- Priority changed from 5 to 3

- Target version changed from 2.0.0 to 3.0

#15 - 06/26/2011 02:15 PM - akr (Akira Tanaka)

- Project changed from 8 to Ruby

- Assignee set to matz (Yukihiro Matsumoto)

#16 - 03/18/2012 06:46 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

#17 - 11/20/2012 09:20 PM - mame (Yusuke Endoh)

- Description updated

- Target version set to 2.6

#18 - 11/27/2017 07:13 AM - mame (Yusuke Endoh)

- Status changed from Assigned to Rejected

This ticket looks hopeless even if we leave it as open. Also, the current behavior is widely accepted by many people. Eregon, if you really want to change this behavior, could you please try to create a patch that has less side effect and then reopen this ticket? Thank you.