Ruby master - Bug #17619

if false foo=42; end creates a foo local variable set to nil

02/10/2021 09:36 AM - pkmuldoon (Phil Muldoon)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
<tr>
<td>Backport:</td>
<td>2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: UNKNOWN</td>
</tr>
</tbody>
</table>

Description

Take this following code

```ruby
[1] pry(main)> defined?(foo)
nil

[2] pry(main)> if false
[2] pry(main)*   foo = 42
[2] pry(main)* end

[3] pry(main)> defined?(foo)
"local-variable"
```

The inner scope inherits the parent scope (ok) but also modifies the parent scope even if the child scope is never entered. A lesser effect of this:

```ruby
[1] pry(main)> defined?(bar)
nil

[2] pry(main)> if false
[2] pry(main)* end

[3] pry(main)> defined?(bar)
"local-variable"

[5] pry(main)> bar
99
```

That somewhat lesser affecting because I can just about accept a variable invading the parent scope, and existing after, as a hoisting event. But surely that should not be the case in the negative program-flow case?

The side effects of this are defined?(foo) can't be trusted anymore.

Apologies if this bug has been filed. I did search for it, but couldn't find anything quite matching it. Thanks!

History

#1 - 02/10/2021 09:37 AM - pkmuldoon (Phil Muldoon)
This affects Ruby 2.7.1 and also Ruby 3.0.0. I've not had time to test earlier versions.

#2 - 02/10/2021 09:39 AM - zverok (Victor Shepelev)
To the best of my understanding, if doesn't create its own scope with its own local variables. So this if means for the interpreter that "local variable bar is present in current scope" (just not assigned).

#3 - 02/10/2021 09:39 AM - pkmuldoon (Phil Muldoon)
The second code example is incorrect, it should read

```ruby
1] pry(main)> defined?(bar)
nil
```
The second code example is incorrect, it should read:

```
if true
  bar = 99
end
```

zverok (Victor Shepelev) wrote in #note-2:

To the best of my understanding, if doesn't create its own scope with its own local variables. So this if means for the interpreter that "local variable bar is present in current scope" (just not assigned).

Okay, but why would a variable be created in a program-flow primitive ie (if etc) if that condition is such that that code is never executed?

I think this is specified and intended behaviour.

https://github.com/ruby/spec/blob/7f0a098a57667ccf70f25aa82cb459dab9a3d960/language/variables_spec.rb#L750-L758

but why would a variable be created in a program-flow primitive ie (if etc) if that condition is such that that code is never executed?

Local variables are 'created' (we could say 'declared') during parse-time in Ruby. That's why they become defined as soon as they are found lexically. We could also call this 'hoisting'.

While the hoisting issue is meh to mostly okay, I guess, this block:

```
if false
  foo = 32
end
```

defined?(foo)  # "local-variable"

is arguably an interpreter side effect that can cause problematic issues to developers. Especially if they use defined?(foo) as a conditional return value. If a condition in a control flow primitive equates to false, arguably, in the case above, that code is being executed, if only partially (the creation of a local variable foo), and causing side effects. If as a developer, I read that code above, I would expect that condition not to be executed, and foo remain undefined. Thanks for looking at this. Your attention to it valued!

In Ruby, local variables are defined by assignment expressions, and it is never in "undefined" state but is initialized as nil.
It's intentional. Eg what if you set the same variable in both branches? Then it would be clearer that you are expecting the variable to be visible outside the scope of the conditional.

Contrived code example:

```ruby
if eligible?
  bonus = 100
else
  bonus = 0
end
score + bonus
```

In that case, I would expect code to flow down one block, which will define bonus, or the other, which will define bonus.

I had forgotten about this WAT. To be honest, even reading the spec, I don't know that I would have expected non-executed code to have defined the variable.

Having said that, if you need to check later that a section of code was executed, checking against a variable being defined is surely setting a landmine for later programmers.

Let's check some history...

```ruby
if condition
  foo = 42
end
...
if defined?(foo)
  ...

v2:

if condition
  foo = 42
end
...

if condition2
  foo = 6 * 9
else
  foo = 0
end
...
if defined?(foo)
  ...

Just don't, unless you're implementing some sort of inspect or something...
```