Add a nil coalescing operator

It would be nice if Ruby had an operator that only considered nil as false, like the null coalescing operators or "Logical Defined-Or operator" (Perl) found in some other languages. I've seen things like `//` and `//=` `??` and `??=`, or `?:` used for this.

This would work like `||` and `||=` for short circuiting etc. except that only nil is considered a false condition.

While Ruby considers only "false" and "nil" as false, with everything else true ("", [], {}, etc.) I still find occasionally people trip up when using logical or, `||` and `||=` when the value may be false.

```ruby
a = 0     || 55
# = 0 Ruby already considers 0, ",", etc. as true (other languages do differ a lot here)
a = 0     ?? 55 # = 0 So no change here
a = nil   || 55 # = 55, nil is false so right side is evaluated.
a = nil   ?? 55 # = 55, again no change
a = false || 55 # = 55, however false is false for logical or
a = false ?? 55 # = false, but its still a non-nil value
```

For example when doing things like:

```ruby
def lazy
  @lazy ||= compute_this
end

def fetch(id, **opts)
  host = opts[:host] || default_host
  https = opts[:https] || true
  port = opts[:port] || (https ? 443 : 80)
  ...
```

Normally the intention is to use a default value or compute an action if no value is provided, which if the value may be false then requires special handling, or sometimes is missed and results in a bug.
In perl I find $x // $y useful vs $x || $y because sometimes you want to accept "" and 0 as values.

In ruby the only 'defined' falsey value is false, so I'm not sure how useful this feature is here.

If you're pulling options from a hash, for example, it's probably better to use a signal like h.fetch 'x', y to show that you accept falsey values from the hash, and/or x.nil? ? y : x to show that you explicitly only don't want nil

By the way, did you actually propose an actual syntax? The two "??

Not really personally set on any given syntax, just ?? and // are familiar to me from other programming. Although actually for ?? specifically, I guess the fact Ruby uses it in both methods and ternary causes a conflict rather than just one or the other (x.nil?? "was nil" : "not nil"). I wouldn't know if the parser can figure that out or not.

But more the concept that any specific syntax.

Hmm, maybe I didn't explain clearly. That is pretty much the pattern I come across repeatedly in Ruby code, and it fails for the false value because false is not "upgraded" when people do 'x || my_default'.

```
"a truthy value" || foo("something else")
# The operator also short circuits so the method `foo` will never even get called
false || foo("something else") # Left side is falsy, so evaluate the right side, but is often not the intent
nil || foo("something else") # Nil is also falsy, so evaluate the right side
```

"a truthy value" ?? foo("something else") # String is true and not nil, so nothing changes here
false ?? foo("something else") # This changes. Left side is not nil, so the right side is never evaluated
nil ?? foo("something else") # Like with `||`, left side is nil so evaluate the right side

```
# so this "does the right thing", as far as my maybe not great example goes
https = opts[:https] ?? true
```

While hash.fetch is nice, I still see || used a lot, in places that maybe wont convert so nice. Also it wont short circuit, in the event the default is not trivial (e.g. with say active record stuff, its easy to have something that goes to the DB without really thinking about it).

```
# Occasionally I see 3 or more chained together
hash[:foo] ||= fetch foo
# Does'nt assign the value like this does, and it wont short circuit `fetch foo`
@lazy ||= calc_lazy # Sometimes used with non-hashes
```

But yes your right, they can all be done other ways, and maybe the better answer is to discourage || in the first place, but I struggled finding ones as tidy to suggest instead.

```
hash[:foo] = fetch foo if !hash.has_key?(:foo)
@lazy = calc_lazy if @lazy.nil?
@lazy # needed because get nil if the if condition is false
if @lazy.nil?
  @lazy = calc_lazy
end
@lazy
```

phluid61 (Matthew Kerwin) wrote:

In perl I find $x // $y useful vs $x || $y because sometimes you want to accept "" and 0 as values.

But not false?

While hash.fetch is nice, I still see || used a lot, in places that maybe wont convert so nice. Also it wont short circuit, in the event the default is not trivial (e.g. with say active record stuff, its easy to have something that goes to the DB without really thinking about it).

```
# Occasionally I see 3 or more chained together
hash[:foo] ||= fetch foo
# Does'nt assign the value like this does, and it wont short circuit `fetch foo`
@lazy ||= calc_lazy # Sometimes used with non-hashes
```

But yes your right, they can all be done other ways, and maybe the better answer is to discourage || in the first place, but I struggled finding ones as tidy to suggest instead.

```
hash[:foo] = fetch foo if !hash.has_key?(:foo)
@lazy = calc_lazy if @lazy.nil?
@lazy # needed because get nil if the if condition is false
if @lazy.nil?
  @lazy = calc_lazy
end
@lazy
```

phluid61 (Matthew Kerwin) wrote:

In perl I find $x // $y useful vs $x || $y because sometimes you want to accept "" and 0 as values.
In perl I find $x // $y useful vs $x || $y because sometimes you want to accept "" and 0 as values.

But not false?

Not in perl ;)

While hash.fetch is nice, I still see || used a lot, in places that maybe wont convert so nice. Also it wont short circuit, in the event the default is not trivial (e.g. with say active record stuff, its easy to have something that goes to the DB without really thinking about it).

Yes, short-circuit is handy. It's why I was a proponent of &.. Maybe it's okay to add // even if it's only used sometimes.

#5 - 08/18/2017 12:16 AM - nobu (Nobuyoshi Nakada)
williamn (William Newbery) wrote:

shevegen (Robert A. Heiler) wrote:

By the way, did you actually propose an actual syntax? The two "??

Not really personally set on any given syntax, just ?? and // are familiar to me from other programming. Although actually for ?? specifically, I guess the fact Ruby uses it in both methods and ternary causes a conflict rather than just one or the other (x.nil?? "was nil" : "not nil"). I wouldn't know if the parser can figure that out or not.

?? is a string literal, and // is a regexp literal.

```ruby
def fetch(id, **opts)
  host = opts[:host] || default_host
  https = opts[:https] || true
  port = opts[:port] || (https ? 443 : 80)
```

Why not keyword arguments?

#6 - 04/25/2020 08:39 PM - swrobel (Stefan Wrobel)
- Subject changed from Add a nill coalescing operator to Add a nil coalescing operator

#7 - 04/29/2020 08:05 PM - bsarrazin (Ben Sarrazin)

Kotlin has this feature, Swift has this feature, many other languages have this feature. Ruby needs this feature :D

"a truthy value" ?? foo("something else")
# The operator also short circuits so the method `foo` will never even get called
false ?? foo("something else") # Left side is falsy, so evaluate the right side, but is often not the intent
nil ?? foo("something else") # Nil is also falsy, so evaluate the right side

"a truthy value" ?? foo("something else") # String is true and not nil, so nothing changes here
false ?? foo("something else") # This changes. Left side is not nil, so the right side is never evaluated
nil ?? foo("something else") # Like with `||`, left side is nil so evaluate the right side

# so this "does the right thing", as far as my maybe not great example goes
https = opts[:https] ?? true

This is exactly why Ruby needs this feature. The lack of type safety, combined with the fact that many tools are taking "false" string as input is compelling argument to add this to the language.

#8 - 04/30/2020 02:50 AM - shyouhei (Shyouhei Urabe)
bsarrazin (Ben Sarrazin) wrote in #note-7:

Kotlin has this feature, Swift has this feature, many other languages have this feature.

No. Kotlin does not have this feature (distinguish false and null). It is a really bad idea for you to refer Kotlin to have something like that. Kotlin is a statically typed language, and its || operator does not take nullable values. No confusion over false versus null must happen. That is why they need ?: operator; they need something similar to || which also work for nullables.

So if you want a ruby operator because false and nil are confusing, that's a totally different story than Kotlin's.
PS. I'm not against the feature itself. I'm just telling that other languages have their own design that do not immediately apply here.

#9 - 04/30/2020 09:00 AM - sawa (Tsuyoshi Sawada)
Your proposal to distinguish nil from false is ad hoc, and is not a real solution for your use case, which is to add a value to a hash only when it does not yet have a corresponding key.

Following your way of doing it, you would still not be able to distinguish opts1 that explicitly has a key-value pair :foo => nil and opts2 that lacks such pair.

```
opts1 = {foo: nil}
opts2 = {}

opts1[:foo] # => nil
opts2[:foo] # => nil
```

Particularly, applying your code to update opts1 would overwrite the explicit nil value (?? stands for your proposed feature):

```
opts1[:foo] ??= "foo"
```

Ruby is aware of such use case, and has already prepared a real solution: the Hash#key? method. The correct way of doing it is:

```
opts1[:foo] = "foo" unless opts1.key?(:foo)
```

Another way of doing it is:

```
opts1 = {foo: "foo"}.merge(opts1)
```

#10 - 10/01/2020 08:41 AM - akim (Akim Demaille)
I second this proposal.

Sure, it is not needed, we can always write things in a different way. As a matter of fact, with such an argument, hardly any improvement should be accepted.

In practice when people use ||, they do mean ?? (whatever its spelling). It just so happens that most of the time, it does what you want, because you happen to not be dealing with Booleans. But the semantics you mean to express is not about "truthness", but about "nilness". And occasionally you get bitten because false does exist, and behaves differently.

Wrt syntax, ?? chains much more naturally than conditionals.

Many of the new languages make the difference, and support this explicitly.

#11 - 10/01/2020 08:47 AM - akim (Akim Demaille)
shyouhei (Shyouhei Urabe) wrote in #note-8:

bsarrazin (Ben Sarrazin) wrote in #note-7:

Kotlin has this feature, Swift has this feature, many other languages have this feature.

No. Kotlin does not have this feature (distinguish false and null).

I disagree. Kotlin does have the feature which is the topic of this page: a null coalescing operator (https://kotlinlang.org/docs/reference/null-safety.html#elvis-operator). And you are rightfully stating it:

That is why they need ??: operator

which exactly corresponds to the proposal about ??.

#12 - 03/15/2021 11:50 PM - lamont (Lamont Granquist)
This would cut down on a lot of bugs dealing with lazy initialization of values where false is valid and needs to be remembered:

```
# hammers on the expensive determination method if it returns false
def should_dosomething?
    @should_dosomething ||= expensively_determine_if_should_do_something
end

# correctly remembers either false or true
```
def should_dosomething?
    @should_dosomething ??= expensively_determine_if_should_do_something
end

This happens more than you might expect because people learn to use ||= in that situation and wind up accidentally writing the buggy code when they're faced with this problem of remembering an expensive thing that returns booleans.

The ??= operator should be introduced into the language and really the idiom should switch to consistently use it all the time to avoid this bug.

#13 - 03/16/2021 12:07 AM - lamont (Lamont Granquist)
Here's example code which works around the lack of a ??= operator in the wild:

https://github.com/chef/chef/blob/4d3b847dee1b917bb139862c623e9633d180fb31/lib/chef/fs/file_system/chef_server/data_bag_dir.rb#L43-L48

    def exists?
      if @exists.nil?
        @exists = parent.children.any? { |child| child.name == name }
      end
      @exists
    end

#14 - 03/16/2021 03:34 PM - p8 (Petrik de Heus)
This is also related to the Hash#fetch_set proposal

cache.fetch_set(key) { calculation }

https://bugs.ruby-lang.org/issues/17342