Proposal: new "ends" keyword

I'm submitting this in the same spirit that "endless methods" was, to promote and produce more concise and easier to write/read code.

Proposal

This is a proposal to introduce a new keyword ends (or endall) as a terminal point to resolve the end of nested "loops|conditionals".

Why

It's a common code occurrence to have multiple levels of loops and/or conditionals, which require separate end keywords to designate their termination points. The end statements themselves are merely for syntactic purposes.

It would be a benefit to programmers, and code readers, to be able to produce|read more concise code, by reducing the "code noise" of these nested multiple end keywords with a shorter|cleaner syntax.

Thus, I propose creating the keyword ends as a shorter|cleaner syntax to replace having to write multiple end keywords.

Example

Below is an example of real code which performs nested loops. With "standard" format it looks like this.

```ruby
def render(scene, image, screenWidth, screenHeight)
  screenHeight.times do |y|
    screenWidth.times do |x|
      color = self.traceRay(....)
      r, g, b = Color.toDrawingColor(color)
      image.set(x, y, StumpyCore::RGBA.from_rgb(r, g, b))
    end
  end
end
```

However, from the point of view of the parser, these are all legal|equivalent.

```ruby
def render(scene, image, screenWidth, screenHeight)
  screenHeight.times do |y|
    screenWidth.times do |x|
      color = self.traceRay(....)
      r, g, b = Color.toDrawingColor(color)
      image.set(x, y, StumpyCore::RGBA.from_rgb(r, g, b))
    end
  end
end
```

This proposal would allow this type of code to be written as:

```ruby
def render(scene, image, screenWidth, screenHeight)
  screenHeight.times do |y|
    screenWidth.times do |x|
      color = self.traceRay(....)
      r, g, b = Color.toDrawingColor(color)
      image.set(x, y, StumpyCore::RGBA.from_rgb(r, g, b))
    end
  end
ends
```
**Pros**

1) code conciseness  
2) better readability  
3) no whitespace dependencies  
4) no conflict with legacy code  
5) attractive to people coming from Python|Nim, et al

**Cons**

No technical implementation restrictions I can think of.  
Maybe alternative name (endall)?

Thanks for consideration.

**Related issues:**

Is duplicate of Ruby master - Feature #5054: Compress a sequence of ends  
Rejected  
Is duplicate of Ruby master - Feature #12241: super end  
Rejected

**History**

#1 - 04/08/2021 04:55 PM - chrisseaton (Chris Seaton)

no conflict with legacy code

How do you differentiate between a call to a method called ends in legacy code, and this new keyword? Do you have some kind of unlimited lookahead during parsing to see if it's needed to make a successful parse? That seems like it'd be very expensive.

#2 - 04/08/2021 04:56 PM - mame (Yusuke Endoh)

- Is duplicate of Feature #5054: Compress a sequence of ends added

#3 - 04/08/2021 05:03 PM - jeremyevans0 (Jeremy Evans)

I don't think you could get reasonable and useful semantics for ends. From your example, ends applies not just the end of the blocks, but also the end of the method. To be consistent, it would have to apply to all containing scopes. Let's look at an example:

```ruby
class A
  def b
    c do
      d do
        e
      end
    end

  def c
    end
end
```

This would be a SyntaxError, since ends would end class A, and the final end would be unexpected.

You could have ends apply only to method definitions and not module/class definitions. You then have to consider this case:

```ruby
A.class_eval do
  define_method :b do
    c do
      d do
        e
      end
    end

  def c
    end
end
```

Surely the only reasonable semantics would have the ends also close the class_eval block, since Ruby couldn’t determine syntactically it is reopening a class. This different behavior in different contexts would be a huge footgun.

Another consideration is this approach of using ends encourages the user to not care about the return value of the methods. In general, that's a bad thing to encourage. Methods that are called for side-effects should probably return nil or self to avoid returning an arbitrary value that callers of the method may accidentally depend on.
Additionally, supporting ends would make it more difficult to move code around (e.g. copy/paste), since the effect of ends could change if the context differs.

#4 - 04/08/2021 08:09 PM - jzakiya (Jabari Zakiya)

The process to implement this proposal are actually much simpler than you make it out to be.

This would be perfectly syntactically legal code.

```ruby
A.class_eval do
  define_method :b do
    c do
      d do
        e
      end
    end
  end

def c
  end
end
```

It would be expanded to

```ruby
A.class_eval do
  define_method :b do
    c do
      d do
        e
      end
  end

def c
  end
end
```

So in your code example, starting at the beginning (outer most layer) the parser starts counting how many things (module|class|method names, loops, conditionals, etc) are currently open (haven't been resolved as terminated). At some point, it counts the last thing that needs to be resolved. When it encounters the first end it tries to resolve it with the last (highest count) thing that's still open. It then continues backing up the tree count, until all the unresolved things count is zero.

So now the source code AST is fully resolved, and w|should look just like normal. This is what is then feed to the compiler. But there has to be a way to know how many things need to be terminated and how many are still open.

I would assume for Ruby, the parsing stage is separate from compilation. You have to go through some process to turn raw source code, through whatever number of stages of processing, to format it into runtime code.

So in my mind, all the parser has to do is account for all open things, and start resolving them from the most inner layer things back out to the outer layers, until every thing is resolved.

This should be a much simpler/easier process to do than what Python does, because it has whitespace dependencies, whereas Ruby doesn't care. All Ruby would have to do is go down, and back up, the parsing tree.

So in the big picture, you don't have to care much about what the thing is, you just have to keep track of how many there are, because all you're going to first do is put the source code in equivalent standard format with expanded out end statements, which can then be processed as usual.

#5 - 04/08/2021 09:43 PM - marcandre (Marc-Andre Lafortune)

Please no. Error prone, not particularly useful, can not be nested, potentially incompatible, ...

My recommendation is to use a text editor that add the end for you.

#6 - 04/08/2021 11:36 PM - duerst (Martin Dürst)

Similar proposals have been made in the past, see e.g. #5054, #12241 (make sure to check the date on the second one).

jzakiya (Jabari Zakiya) wrote in #note-4:
So in your code example, starting at the beginning (outer most layer) the parser starts counting how many things (module|class|method names, loops, conditionals, etc) are currently open (haven't been resolved as terminated). At some point, it counts the last thing that needs to be resolved. When it encounters the first end it tries to resolve it with the last (highest count) thing that's still open. It then continues backing up the tree count, until all the unresolved things count is zero.

So in the big picture, you don't have to care much about what the thing is, you just have to keep track of how many there are, because all you're going to first do is put the source code in equivalent standard format with expanded out end statements, which can then be processed as usual.

So is do a thing, or not? In the above example, you treat three of the dos as things, but not the forth one. Why? You say to count things down to zero, but why did you only count down to one? If counting down to zero, it means you are back at the top level. That means that ends cannot be used inside a class or module (or inside anything else, for that matter, except maybe for {} blocks, if these are not a thing).

---

#7 - 04/08/2021 11:37 PM - duerst (Martin Dürst)
- Is duplicate of Feature #12241: super end added

#8 - 04/09/2021 12:30 AM - hsbt (Hiroshi SHIBATA)
- Tags set to joke

#9 - 04/09/2021 02:43 AM - mame (Yusuke Endoh)
- Tags deleted (joke)

I agree that this proposal is very unlikely to be successful, but I guess the proposer is serious, so I'm removing "joke" tag.

#10 - 04/09/2021 03:52 AM - shevegen (Robert A. Heiler)
When I read the proposal I had to think about the "ennnmnd" proposal. :) That one was linked in above:
https://bugs.ruby-lang.org/issues/5054

It's hard to say how serious people are, but I think "ennnmmnd" was a joke, while this here is probably not a joke.

The reason I think this suggestion is not a joke is because I can actually understand SOME of the rationale behind it. Perhaps mame had a similar idea when he suggested "endless method definition" - after all you can now omit some syntax, and since mame is a known golfer, I am sure there is some golf-synergy too :) (although the suggestion was cleverly made on a first april without being an april joke, so a hidden double joke that was no joke!).

I also wanted a way to be able to omit "end", so from that point of view I can understand jzakiya's basic idea behind the proposal.

The reason why I think being able to omit "end" may be useful (sometimes) is actually somewhat similar as to what jzakiya wrote; he wrote this specifically:

"The end statements themselves are merely for syntactic purposes."

And, indeed, although I would not use the same description as he did, I can understand what he means. Perhaps I can explain it with another example.

To me, personally, the "end" is not giving me a lot of new information .

Take a typical module and class definition:

```ruby
module Foo
class Bar
def hello_world
  end
end
end
```

I think most people may use a style like the above. So, if you look at it then perhaps the first "end" is somewhat useful ... closes the method. But the second end is not so much useful to you probably, as a writer of ruby code. You essentially only "close" class Bar, although if you'd reach the end of the file, well ... that file is closed.
already. So you kind of have to do an additional syntax cue :P

And the very last "end" is actually quite annoying, even more so than the second one. Because we don't really, in that example at least, use the toplevel module "namespace" much at all. We just use it to make it easier to integrate code written by other people too, for the most part - like define our "namespace", such as "module Foobar", where the other classes and files reside.

When you write a lot of ruby code, this can indeed be tedious. And in some ways it distracts a bit as well, especially in larger projects where there is a LOT of nesting, LOTS of classes and lots of files.

My old idea was, however had, a bit different. Rather than this being a default variant (to omit "end"), I thought whether it may be beneficial to define this on a per .rb basis instead. So, per file.

A bit like frozen strings, say in the toplevel comment, where we say "ruby, I will use mandatory indent now for this file, and omit all end as a consequence. You figure out where the ends are or should be, based on that mandatory indent information".

Like a bit a "lazy-mode" AST-like handling of the code, where we as writers of ruby code could simple omit "end".

A bit like how python works, with the mandatory indent or mandatory whitespace rather ... (although python also uses a ":" for method definitions and I never fully understood why it needed BOTH indent-information, and the " ": too ... when I then also have to pass "self" explicit in python, I am quite annoying, since ruby has the better way to handle the syntax here really, in my opinion)

Unfortunately, while it may be great to be able to omit "end", this most likely creates a few new problems, trivial ones too, like ...

If you copy/paste ruby code, then you have to sort of figure that out, when other people can not just copy/paste it as-is, since the "end" would be missing. Any example on github where different syntax styles are allowed, may then either work or not work - even more so when you want to integrate code into an existing code base, and when there is lots of that code.

And, I was also never certain whether my idea in regards to being able to omit "end" would be any good really (because I myself really don’t know ... I only have that idea when I write a LOT of "end". When I use only a few "end"s, I don’t quite care about indent as much anymore :P )

Personally I think it may not really be worthwhile to implement the suggestion by jzakiya. It may also create problems for newcomers, since they then may have to understand when to use "end", and when not to use it. As much as "end" may not be fun, it is at the least simple.

Note that "endall" is, from a syntax point of view, uglier than "end". The joke suggestion "ennnnnnd" has a similar problem. And nobody counts the "n", that was clearly a joke. It takes me longer to count the "n" than the "end"s ... :P

I don't think "endall" is a joke ... but syntax-wise I think it's not good either.

So my personal opinion is rather against that proposal, even though I think the rationale is not completely without merit - "end" does not add as much useful information IMO.

One last comment:

I actually tend to use this style a lot:

```ruby
def foo(array)
  array.each {|entry|
    
  end
```

So, I actually deliberately use the [] there, as a visual cue. I guess nobody else does this really, since most prefer do/end in general, which I can understand. But I liked it as visual cue simply.

#11 - 04/09/2021 06:25 AM - duerst (Martin Dürst)

Just to be clear, I'd not be against a good way to shorten a series of ends into something simpler. But the current proposal is not at all clear on the exact semantics: What gets closed with an ends, and what doesn't get closed? In that sense, the ennnnd proposal was clearer, an ennd would stand
for two ends, and so on.

#12 - 04/09/2021 07:12 AM - nobu (Nobuyoshi Nakada)
- Description updated

I don't think a new keyword is acceptable, because of backward compatibility, ambiguity, and so on. If it is an issue about looking, it feels an editors' role to me. For example, like as hide-ifdef mode of Emacs, hide-end mode would be possible.

#13 - 04/10/2021 03:43 PM - jzakiya (Jabari Zakiya)
The examples I provided show the intent of what its use is for, which is to provide one termination point for a string of consecutive end statements, and nothing more.

Python|Nim show they can do this, using whitespace|indentation. This makes code much more concise, easier to read|write, and easier to understand. Those string of ends are merely for the benefit of the parser, and not humans.

Please focus on the intent and purpose, and not semantics.

Also, there are no backwards incompatibility issues because there are no issues with parsing old code, just as there are no backwards incompatibilities with endless methods. If a programmer doesn't write code to use it, there is no issue going forward, or backward. Obviously, if one wants code to run on pre 3.0 systems, one doesn't use endless methods, but old code will run on 3.0. This feature would create the same options for programmers to assess using, or not.

#14 - 04/10/2021 03:55 PM - chrisseaton (Chris Seaton)

Please focus on the intent and purpose, and not semantics.

But we have to decide what the semantics will be in order to specify and implement it! The point of this issue tracker is to debate semantics of proposals.

All Ruby would have to do is go down, and back up, the parsing tree.

Currently Ruby is able to parse and compile in broadly two single passes. What you're proposing is pretty radically different to that and could have serious implications for implementation, performance, and compatibility.

Maybe you could write a PR introducing a specification and implementing the feature to see how you envision it being done?

#15 - 04/11/2021 01:40 AM - duerst (Martin Dürst)
jzakiya (Jabari Zakiya) wrote in #note-13:

The examples I provided show the intent of what its use is for, which is to provide one termination point for a string of consecutive end statements, and nothing more.

We understand your "intent". And I think it would be great if we could reduce repeated ends in Ruby programs. But programming languages are not (or not yet) about somehow guessing a programmer's intent, they have to have exact definitions.

Python|Nim show they can do this, using whitespace|indentation. This makes code much more concise, easier to read|write, and easier to understand. Those string of ends are merely for the benefit of the parser, and not humans.

There are other languages that use indentation for program structure, such as Haskell. Are you saying that you ends proposal would include using indentation levels to decide how many end keywords a single ends would stand in for? If that's the case, please explain the specifics.

Also, there are no backwards incompatibility issues because there are no issues with parsing old code, just as there are no backwards incompatibilities with endless methods. If a programmer doesn't write code to use it, there is no issue going forward, or backward. Obviously, if one wants code to run on pre 3.0 systems, one doesn't use endless methods, but old code will run on 3.0. This feature would create the same options for programmers to assess using, or not.

There is a compatibility issue. Chris mentioned it in https://bugs.ruby-lang.org/issues/17786#note-1.