Allow syntax like `obj.method(arg)=value`

I propose here to allow a syntax like:

```
obj.method(argv) = value
```

It would be translated to the following:

```
obj.__send__(:method=, arg, value)
```

The lack of this syntax kind of limits the ability to design DSLs in Ruby in my opinion. I don't think this would bring any conflicts with existing parser rules.

My proposal would be to put the value at the last argument, akin to how []= works. So, for example this code would work:

```ruby
module Indexable
  def dig(*path, last, value)
    if path.empty?
      self[last] = value
    else
      first = path.shift
      self[first].dig(*path, last) = value
    end
  end
end
```

```ruby
Hash.include Indexable
Array.include Indexable
```

The kwargs may be supported similarly to how they work on []=, ie. becoming a penultimate Hash argument. While maybe not perfect, it is consistent with how []= works and I imagine most usecases won't require kwargs.

**History**

**#1 - 02/27/2022 08:03 AM - sawa (Tsuyoshi Sawada)**

Notice that, while [] in []= cannot be omitted, () in method invocation can be omitted. Are you expecting () in the proposed feature to be omittable or not? Either way, I think it would introduce complexity. I think that is a crucial difference between []= and the proposed feature; the latter is not as simple as the former.

**#2 - 02/27/2022 08:17 AM - baweaver (Brandon Weaver)**

I believe in this particular case it would make more sense to have a dual method to dig, rather than adding additional complexity to the syntactic sugar around =. There were proposals in the past for Enumerable#bury that targeted this behavior:

```ruby
module Enumerable
  def bury(*paths, &value_fn)
    return unless block_given?
    *lead, target = paths
    if lead.empty? # Single item in path
      self[target] = yield(self[target])
    else
      above_value = self.dig(*lead)
      above_value[target] = yield(above_value[target])
    end
  end
end
```

### Description

I propose here to allow a syntax like:

```ruby
obj.method(arg) = value
```

It would be translated to the following:

```ruby
obj.__send__(:method=, arg, value)
```

The lack of this syntax kind of limits the ability to design DSLs in Ruby in my opinion. I don't think this would bring any conflicts with existing parser rules.

My proposal would be to put the value at the last argument, akin to how []= works. So, for example this code would work:

```ruby
module Indexable
  def dig(*(path, last), value)
    if path.empty?
      self[last] = value
    else
      first = path.shift
      self[first].dig(*(path, last)) = value
    end
  end
end
```

```ruby
Hash.include Indexable
Array.include Indexable
```

The kwargs may be supported similarly to how they work on []=, ie. becoming a penultimate Hash argument. While maybe not perfect, it is consistent with how []= works and I imagine most usecases won't require kwargs.

### History

**#1 - 02/27/2022 08:03 AM - sawa (Tsuyoshi Sawada)**

Notice that, while [] in []= cannot be omitted, () in method invocation can be omitted. Are you expecting () in the proposed feature to be omittable or not? Either way, I think it would introduce complexity. I think that is a crucial difference between []= and the proposed feature; the latter is not as simple as the former.

**#2 - 02/27/2022 08:17 AM - baweaver (Brandon Weaver)**

I believe in this particular case it would make more sense to have a dual method to dig, rather than adding additional complexity to the syntactic sugar around =. There were proposals in the past for Enumerable#bury that targeted this behavior:

```ruby
module Enumerable
  def bury(*paths, &value_fn)
    return unless block_given?
    *lead, target = paths
    if lead.empty? # Single item in path
      self[target] = yield(self[target])
    else
      above_value = self.dig(*lead)
      above_value[target] = yield(above_value[target])
    end
  end
end
```
The difficulty for such functions is how to differentiate between the varadic path and the value setter. For me I believe block functions strike a good medium here, and may be a viable solution, though Matz has previously rejected the idea of adding a bury function. Perhaps it may make sense to bring it up again if this is compelling, as it does bear some similarity to merge behavior.

#3 - 02/27/2022 02:31 PM - hmdne (hmdne -)
sawa (Tsuyoshi Sawada) - My proposal would be to allow omitting parentheses only if there are no arguments provided, ie. how it is currently.

```
sself.xyz = 6  # correct currently
self.xyz() = 6  # correct under the proposal
self.xyz(a) = 6  # correct under the proposal
self.xyz a = 6  # parser conflict, it is already a correct code meaning something else
```

I am not very familiar with MRI code unfortunately, so I can't estimate if this will introduce a lot of complexity or not. Certainly `value` will need to accept a lot more types of expression. I develop an alternative Ruby implementation (Opal), though not its parser, and there I have a clear path for implementation of this feature.

A similar, though a little different, feature exists in C, where `*get_mem(123) = 123` is a correct code.

Another argument for this feature is that it's easy to pass an arbitrary number of arguments to any operator, like `+`, `*`, `[]`, `[]=` something?, `something!`. From what I know, something= is the only kind of operator that needs `__send__` to pass other number of arguments than 1 (`(self.x=(1,2,3)` is not a correct code, perhaps it could be an alternative to accept it, instead of this proposal, but I assume it will be a lot harder).

I happen to sometimes want to add an optional argument to setter and it ends up with a code refactor that makes code a lot more complicated, needing methods like set_x, while I can easily add arguments to a getter.

baweaver (Brandon Weaver)

The difficulty for such functions is how to differentiate between the varadic path and the value setter

While block adds a lot of flexibility for certain cases, like bury, this proposal more clearly separates what's a variadic path and what's a value (following the semantics of `[]` and `[=]` operators). Perhaps my pseudo-code with dig= isn't the greatest idea, but it demonstrates the concept.

A more real life example happened in my code. I wanted to create an API like the following:

```
entity.attribute(3)  # => value of attribute 3
entity.attribute("attrtype 3 by name")  # => value of attribute 3
t = entity.attribute(AttrType.new(3))  # => value of attribute 3
entity.attribute(3) = 10
entity.attribute("attrtype 3 by name") = 10
t = entity.attribute(AttrType.new(3)) = 10
```

Of course I ended up with extending Hash to accurately resolve the hash keys. Instead of the API described above, I created this:

```
entity.attributes[3]  # => value of attribute 3
t = entity.attributes["attrtype 3 by name"]  # => value of attribute 3
t = entity.attributes[AttrType.new(3)]  # => value of attribute 3
entity.attributes[3] = 10
entity.attributes("attrtype 3 by name") = 10
t = entity.attributes[AttrType.new(3)] = 10
```

While I achieved the same goal by it, the resulting API implementation added a lot of complexity.

#4 - 02/27/2022 05:04 PM - Eregon (Benoit Daloze)

IMHO way too complicated and `[=]` seems good enough for this use case.

Also it would be very confusing as the similar and existing syntax def obj.method(arg) = value means something completely different.