Ruby master - Feature #14912

Introduce pattern matching syntax

07/14/2018 11:24 PM - ktsj (Kazuki Tsujimoto)

Status:	Assigned	
Priority:	Normal	
Assignee:	ktsj (Kazuki Tsujimoto)	
Target version:	2.7	

Description

I propose new pattern matching syntax.

Pattern syntax

Here's a summary of pattern syntax.

```
# case version
case expr
in pat [if|unless cond]
  ...
in pat [if|unless cond]
  ...
else
  ...
end

pat: var                                                  # Variable pattern. It matches any valu
e, and binds the variable name to that value.
   | literal                                              # Value pattern. The pattern matches an
 object such that pattern === object.
   | Constant                                             # Ditto.
   | var_                                                 # Ditto. It is equivalent to pin operat
or in Elixir.
   | (pat, ..., *var, pat, ..., id:, id: pat, ..., **var)  # Deconstructing pattern. See below for
 more details.
   | pat(pat, ...)                                        # Ditto. Syntactic sugar of (pat, pat,
...).
   | pat, ...                                             # Ditto. You can omit the parenthesis (
top-level only).
   | pat | pat | ...                                      # Alternative pattern. The pattern matc
hes if any of pats match.
   | pat => var                                           # As pattern. Bind the variable to the
value if pat match.

# one-liner version
$(pat, ...) = expr                                        # Deconstructing pattern.
```

The patterns are run in sequence until the first one that matches.
If no pattern matches and no else clause, NoMatchingPatternError exception is raised.

Deconstructing pattern

This is similar to Extractor in Scala.

The patten matches if:

- An object have #deconstruct method
- Return value of #deconstruct method must be Array or Hash, and it matches sub patterns of this

```
class Array
  alias deconstruct itself
end
```

```
case [1, 2, 3, d: 4, e: 5, f: 6]
in a, *b, c, d:, e: Integer | Float => i, **f
  p a #=> 1
  p b #=> [2]
  p c #=> 3
  p d #=> 4
  p i #=> 5
  p f #=> {f: 6}
  e   #=> NameError
end
```

This pattern can be used as one-liner version like destructuring assignment.

```
class Hash
  alias deconstruct itself
end

$(x:, y: (_, z)) = {x: 0, y: [1, 2]}
p x #=> 0
p z #=> 2
```

Sample code

```
class Struct
  def deconstruct; [self] + values; end
end

A = Struct.new(:a, :b)
case A[0, 1]
in (A, 1, 1)
  :not_match
in A(x, 1) # Syntactic sugar of above
  p x #=> 0
end

require 'json'

$(x:, y: (_, z)) = JSON.parse('{"x": 0, "y": [1, 2]}', symbolize_names: true)
p x #=> 0
p z #=> 2
```

Implementation

- https://github.com/k-tsj/ruby/tree/pm2.7-prototype
  - Test code: https://github.com/k-tsj/ruby/blob/pm2.7-prototype/test_syntax.rb

Design policy

- Keep compatibility
  - Don't define new reserved words
  - 0 conflict in parse.y. It passes test/test-all
- Be Ruby-ish
  - Powerful Array, Hash support
  - Encourage duck typing style
  - etc
- Optimize syntax for major use case
  - You can see several real use cases of pattern matching at following links :)
    - https://github.com/k-tsj/power_assert/blob/8e9e0399a032936e3e3f3c1f06e0d038565f8044/lib/power_assert.rb#L106
    - https://github.com/k-tsj/pattern-match/network/dependents

Related issues:

Related to Ruby master - Feature #14709: Proper pattern matching	**Closed**
Related to Ruby master - Feature #15865: `<expr> in <pattern>` expression	**Closed**

Related to Ruby master - Feature #15918: Pattern matching for Set	**Open**
Related to Ruby master - Feature #15881: Optimize deconstruct in pattern matc...	**Open**
Related to Ruby master - Feature #15824: respond_to pattern for pattern match	**Open**
Has duplicate Ruby master - Feature #15814: Capturing variable in case-when b...	**Closed**

Associated revisions

Revision 9738f96f - 04/17/2019 06:48 AM - ktsj (Kazuki Tsujimoto)

Introduce pattern matching [EXPERIMENTAL]

[ruby-core:87945] [Feature #14912]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@67586 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 67586 - 04/17/2019 06:48 AM - ktsj (Kazuki Tsujimoto)

Introduce pattern matching [EXPERIMENTAL]

[ruby-core:87945] [Feature #14912]

Revision d4da74ea - 11/08/2019 02:37 AM - ktsj (Kazuki Tsujimoto)

Define Struct#deconstruct_keys

History

#1 - 07/15/2018 04:36 PM - shevegen (Robert A. Heiler)

I have one question:

- Is the above exclusive for "in" in case, or can it be combined with "when"?

E. g.:

```
case A[0, 1]
when 3
puts 'bla'
in (A, 1, 1)
# etc
```

?

#2 - 07/15/2018 11:03 PM - ktsj (Kazuki Tsujimoto)

> Is the above exclusive for "in" in case, or can it be combined with "when"?

The former is right, but I don't have any strong opinion about it.

The reason why I select the former is that "in" is superset of "when".

#3 - 07/18/2018 07:14 AM - mrkn (Kenta Murata)

- Related to Feature #14913: Extend case to match several values at once added

#4 - 07/18/2018 07:14 AM - akr (Akira Tanaka)

I expect deconstrocut methods will be defined for core classes if this proposal is accepted.

But I feel the deconstruct method of Struct in the sample code is tricky
because it duplicates values.
(s.deconstruct[0][0] and s.deconstruct[1] has same value)

```
class Struct
  def deconstruct; [self] + values; end
end
```

I doubt that the deconstruct method is suitable for
standard definition.

I guess "& pattern", pat & pat & ..., may solve this problem.
("pat1 & pat2 & ..." matches if all patterns (pat1, pat2, ...) matches.)

#5 - 07/18/2018 07:31 AM - shyouhei (Shyouhei Urabe)

- Related to deleted (Feature #14913: Extend case to match several values at once)

#6 - 07/18/2018 10:07 AM - shyouhei (Shyouhei Urabe)

We had some in-detail discussuion about the possibility of this issue in todays developer meeting. Though it seemed a rough cut that needs more brush-ups, the proposal as a whole got positive reactions. So please continue developing.

Some details the attendees did not like:

- Deconstruction seems fragile; For instance the following case statement matches, which is very counter-intuitive.

```
def foo(obj)
  case obj
  in a, 1 => b, c then
    return a, b, c
  else
    abort
  end
end

A = Struct.new(:x, :y)
p foo(A[1, 2]) # => [A, 1, 2]
```

- There is | operator that is good. But why don't you have counterpart & operator?
- Pinning operator is necessary. However the proposed syntax do not introduce an operator rather it introduces naming convention into local variable naming. This is no good. We need a real operator for that purpose.
- One-liner mode seems less needed at the moment. Is it necessary for the first version? We can add this later if a real-world use-case is found that such shorthand is convenient, rather than cryptic.
- Some attendees do not like that arrays cannot be pattern matched as such.

```
case [1, 2, [3, 4]]
in [a, b, [3, d]] # <- unable to do this
  ...
end
```

- Should #deconstruct be called over and over again to the same case target? Shouldn't that be cached?

But again, these points are about details. The proposal as a whole seemed roughly okay.

#7 - 07/21/2018 12:34 AM - ktsj (Kazuki Tsujimoto)

Thanks for the feedback.

> But I feel the deconstruct method of Struct in the sample code is tricky
> because it duplicates values.
>
> - Deconstruction seems fragile; For instance the following case statement matches, which is very counter-intuitive.

It is trade-off with duck typing.

Consider following case.

```
class MyA
  def deconstruct
    dummy = A[nil, nil]
    return dummy, my_x, my_y
  end
end

obj = MyA.new

case obj
in A(x, y)
  ...
end
```

We can match the pattern even if obj is not an instance of A class.

> I guess "& pattern", pat & pat & ..., may solve this problem.

("pat1 & pat2 & ..." matches if all patterns (pat1, pat2, ...) matches.)

- There is | operator that is good. But why don't you have counterpart & operator?

If & operator is also introduced, I think a user wants to use parenthesis to control precedence of patterns.
It conflicts with syntax of my proposal.

- Pinning operator is necessary. However the proposed syntax do not introduce an operator rather it introduces naming convention into local variable naming. This is no good. We need a real operator for that purpose.

Agreed.

- One-liner mode seems less needed at the moment. Is it necessary for the first version? We can add this later if a real-world use-case is found that such shorthand is convenient, rather than cryptic.

Agreed.

- Some attendees do not like that arrays cannot be pattern matched as such.

```
case [1, 2, [3, 4]]
in [a, b, [3, d]] # <- unable to do this
  ...
end
```

I can understand motivation of this request.

- Should #deconstruct be called over and over again to the same case target? Shouldn't that be cached?

I think it should be cached.

#8 - 07/23/2018 12:24 AM - shyouhei (Shyouhei Urabe)

ktsj (Kazuki Tsujimoto) wrote:

- Deconstruction seems fragile; For instance the following case statement matches, which is very counter-intuitive.

It is trade-off with duck typing.

Do you really think they are worth trading off?

What is, then, the purpose of pattern matching at the first place?

To me it is very troublesome when case obj in a, b, c then ... end matches something non-Array. That should ruin the whole benefit of pattern matching. Pattens should never match against something you don't want to match.

#9 - 07/23/2018 01:54 AM - akr (Akira Tanaka)

ktsj (Kazuki Tsujimoto) wrote:

Thanks for the feedback.

But I feel the deconstruct method of Struct in the sample code is tricky
because it duplicates values.

- Deconstruction seems fragile; For instance the following case statement matches, which is very counter-intuitive.

It is trade-off with duck typing.

Consider following case.

```
class MyA
  def deconstruct
    dummy = A[nil, nil]
    return dummy, my_x, my_y
  end
end
```

```
obj = MyA.new

case obj
in A(x, y)
  ...
end
```

We can match the pattern even if obj is not an instance of A class.

Hm. I didn't explan my intent well.

My intent is deconstructing pattern is not well correspondence to
pattern matching of functional languages.

In functional languages, a data type is defined with
constructors and their arguments.

For example list of int can be defined in OCaml as follows.

```
type intlist =
  | Inil
  | Icons of int * intlist
```

There are two constructors:

- constructor for empty list, Inil. It has no arguments.
- constructor for non-empty list, Icons. It has two arguments: first element and remaining list.

We can use pattern matching on the value of a data type.

For example, the length of list of int can be defined as follows.

```
let rec len il =
  match il with
  | Inil -> 0
  | Icons (_, il1) -> 1 + len il1
```

pattern matching distinguish the constructor of a value and
extract arguments of their constructors.

I think Ruby's pattern matching should support this style.

In Ruby, a data type of functional language can be implemented as
multiple classes: one class for one constructor.

```
class Inil
  def initialize()
  end
end
class Icons
  def initialize(e, il)
    @e = e
    @il = il
  end
end
```

(More realistic example, such as AST, may be more interesting.)

So, pattern matching need to distinguish class (correspond to constructor)
AND extract arguments for constructor.

In your proposal, it needs that deconstruct method must be implemented like
your Struct#deconstruct.

This means that, if deconstruct method is not defined like Struct#deconstruct,
Ruby's pattern matching is not usable as pattern matching of functional
languages.

For example, your Array#deconstruct and Hash#deconstruct is not like
Struct#deconstruct.
So, I guess your pattern matching is difficult to use data structures
mixing Array and Hash.
I.e. "distinguish Array and Hash, and extract elements" seems difficult.

I expect Ruby's pattern matching support the programming style of
pattern matching of functional languages.
So, I'm suspicious with the deconstructing pattern of your proposal.

Note that I don't stick to "and" pattern.

#10 - 07/24/2018 04:49 AM - egi (Satoshi Egi)

Let me propose you to import the functions for non-linear pattern matching with backtracking that I have implemented as a Ruby gem in the following repository.

https://github.com/egison/egison-ruby/

This pattern-matching system allows programmers to replace the nested for loops and conditional branches into simple pattern-match expressions (Please see README of the above GitHub repository).

It achieved that by fulfilling all the following features.

- Efficiency of the backtracking algorithm for non-linear patterns
- Extensibility of pattern matching
- Polymorphisim in patterns

There are no other programming languages that support all the above features (especially the first and second features) though many works exist for pattern matching (as listed up in the following link: https://ghc.haskell.org/trac/ghc/wiki/ViewPatterns).
Therefore, if Ruby has this pattern-matching facility, it will be a great advantage even over advanced programming languages with academic background such as Haskell.

#11 - 07/24/2018 04:59 AM - shyouhei (Shyouhei Urabe)

egi (Satoshi Egi) wrote:

> Let me propose you to import the functions for non-linear pattern matching with backtracking that I have implemented as a Ruby gem in the following repository.

Open a new issue for that, please. Don't hijack this thread.

#12 - 07/28/2018 01:14 AM - ktsj (Kazuki Tsujimoto)

shyouhei-san:
I changed my mind. We should be able to avoid such "fragile" case.
Though duck typing is important, it should be designed by another approach.

akr-san:

> I think Ruby's pattern matching should support this style.

I agree, but it isn't enough.
I expect that Ruby's pattern matching also lets us write following code.

```
module URI
  def deconstruct
    {scheme: scheme, host: host, ...}
  end
end

case URI('http://example.com')
in scheme: 'http', host:
  ...
end

class Cell
  def deconstruct
    @cdr ? [@car] + @cdr.deconstruct : [@car]
  end
  ...
end

list = Cell[1, Cell[2, Cell[3, nil]]]
case list
in 1, 2, 3
  ...
end
```

So, how about an idea which divides deconstructing pattern into typed and non-typed one?

```
pat:: pat, ...        # (Non-typed) deconstructing pattern
      val(pat, ...)  # Typed deconstructing pattern. It matches an object such that `obj.kind_of?(val)` and `p
at, ...` matches `obj.deconstruct`
      [pat, ...]     # Syntactic sugar of `Array(pat, ...)`. (if needed)
      {id: pat, ...} # Syntactic sugar of `Hash(id: pat, ...)`. (if needed)

class Struct
  alias deconstruct values
end

A = Struct.new(:a, :b)

def m(obj)
  case obj
  in A(a, b)
    :first
  in a, b
    :second
  end
end

m(A[1, 2]) #=> :first
m([1, 2])  #=> :second
m([A[nil, nil], 1, 2]) #=> NoMatchingPatternError
```

#13 - 08/01/2018 12:18 AM - baweaver (Brandon Weaver)

There was a previous discussion on this which had many good details and discussion:

https://bugs.ruby-lang.org/issues/14709

zverok (Victor Shepelev) and I had done some work and writing on this which may yield some new ideas.

#14 - 08/01/2018 02:32 PM - ktsj (Kazuki Tsujimoto)

- Related to Feature #14709: Proper pattern matching added

#15 - 08/12/2018 05:14 AM - bozhidar (Bozhidar Batsov)

Btw, won't it better to introduce a new expression named match than to extend case? Seems to me this will make the use of patter matching clearer, and I assume it's also going to simplify the implementation.

#16 - 08/13/2018 12:07 PM - zverok (Victor Shepelev)

> Btw, won't it better to introduce a new expression named match than to extend case?

I have exactly the opposite question: do we really need in, why not reuse when?.. For all reasonable explanations, case+when IS Ruby's "pattern-matching" (however limited it seems), and I believe introducing new keywords with "similar yet more powerful" behavior will lead to a deep confusion.

#17 - 08/27/2018 06:34 PM - dsisnero (Dominic Sisneros)

"The patterns are run in sequence until the first one that matches."

This means O(n) time for matching. If we are adding this with changes to compiler, we should try to compile it to hash lookup O(1) time

Does this allow nesting of patterns Do patterns compose or nest?

def simplify(n)
case n
in IntNode(n)
then n
in NegNode(Negnode n) then
simplify(NegNode.new(simplify (n))
in AddNode(IntNode(0), right) then
simplify(right)
in MulNode(IntNode(1) right) then
simplify(right)

etc

Java is going to get pattern matching also and has some good ideas

https://www.youtube.com/watch?v=n3_8YcYKScw&list=PLX8CzqL3ArzXJ2EGftrmz4SzS6NRr6p2n&index=1&t=907s

and

http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html

Their proposal includes nesting and also is O(1) time.

Not sure how much translates though.

#18 - 10/08/2018 04:03 PM - jwmittag (Jörg W Mittag)

I don't have anything specific to say about this particular proposal, I just want to point out that a lot of people have been thinking about how Pattern Matching relates to Object-Oriented Data Abstraction and Dynamic Languages recently. This proposal already mentions Scala and its Extractors, which guarantee that Pattern Matching preserves Abstraction / Encapsulation.

Another language that is semantically even closer to Ruby (highly dynamic, purely OO, Smalltalk heritage) is Newspeak. The Technical Report Pattern Matching for an Object-Oriented and Dynamically Typed Programming Language, which is based on Felix Geller's PhD Thesis, gives a good overview.

Also, influenced by the approach to Pattern Matching in Scala and Newspeak is Grace's approach, described in Patterns as Objects in Grace.

#19 - 04/17/2019 06:48 AM - ktsj (Kazuki Tsujimoto)

- Status changed from Open to Closed

Applied in changeset trunk|r67586.

Introduce pattern matching [EXPERIMENTAL]

[ruby-core:87945] [Feature #14912]

#20 - 04/19/2019 05:28 AM - ktsj (Kazuki Tsujimoto)

- Target version set to 2.7

- Assignee set to ktsj (Kazuki Tsujimoto)

- Status changed from Closed to Assigned

The specification is still under discussion, so I reopend the issue.

Current specifation summary:

```
case obj
in pat [if|unless cond]
  ...
in pat [if|unless cond]
  ...
else
  ...
end
```

```
pat: var                                                # Variable pattern. It matches any value, and binds
 the variable name to that value.
   | literal                                            # Value pattern. The pattern matches an object such
 that pattern === object.
   | Constant                                           # Ditto.
   | ^var                                               # Ditto. It is equivalent to pin operator in Elixir
.
   | pat | pat | ...                                    # Alternative pattern. The pattern matches if any o
f pats match.
   | pat => var                                         # As pattern. Bind the variable to the value if pat
 match.
   | Constant(pat, ..., *var, pat, ...)                # Array pattern. See below for more details.
   | Constant[pat, ..., *var, pat, ...]                # Ditto.
   | [pat, ..., *var, pat, ...]                        # Ditto (Same as `BasicObject(pat, ...)` ). You can
 omit brackets (top-level only).
   | Constant(id:, id: pat, "id": pat, ..., **var)     # Hash pattern. See below for more details.
   | Constant[id:, id: pat, "id": pat, ..., **var]     # Ditto.
```

```
   | {id:, id: pat, "id": pat, ..., **var}                  # Ditto (Same as `BasicObject(id:, ...)` ). You can
 omit braces (top-level only).


An array pattern matches if:
* Constant === object returns true
* The object has a #deconstruct method that returns Array
* The result of applying the nested pattern to object.deconstruct is true

A hash pattern matches if:
* Constant === object returns true
* The object has a #deconstruct_keys method that returns Hash.
* The result of applying the nested pattern to object.deconstruct_keys(keys) is true
```

For more details, please see https://speakerdeck.com/k_tsj/pattern-matching-new-feature-in-ruby-2-dot-7.

#21 - 05/01/2019 08:21 AM - duerst (Martin Dürst)

- Has duplicate Feature #15814: Capturing variable in case-when branches added

#22 - 06/08/2019 11:56 AM - pitr.ch (Petr Chalupa)

Hi, I am really looking forward to this feature. Looks great!

However I'd like to make few suggestions which I believe should be part of the first pattern matching experimental release. I'll include use-cases and try to explain why it would be better to do so.

(1) Pattern matching as first-class citizen

Everything in Ruby is dynamically accessible (methods, classes, blocks, etc.) so it would be pity if patterns would be an exception from that. There should be an object which will represent the pattern and which can be lifted from the pattern literal.

It may seem that just wrapping the pattern in a lambda as follows is enough to get an object which represents the pattern.

```
-> value do
  case value
  in (1..10) => i
    do_something_with i
  end
end
```

In some cases it is sufficient however lets explore some interesting use cases which cannot be implemented without the first-class pattern-matching.

First use-case to consider is searching for a value in a data structure. Let's assume we have a data-structure (e.g. some in memory database) and we want to provide an API to search for an element with a pattern matching e.g. #search. The structure stores log messages as follows ["severity", "message"]. Then something as follows would be desirable.

```
def drain_erros(data)
  # pops all messages matching at least one pattern
  # and evalueates the appropriate branch with the destructured log message
  # for each matched message
    data.pop_all case
    in ["fatal", message]
      deal_with_fatal message
    in ["error", message]
      deal_with_error message
    end
end
```

There are few things to consider. Compared to the already working implementation there is no message given to the case since that will be later provided in the pop_all method. Therefore the case in here has to evaluate to an object which encapsulates the pattern matching allowing to match candidates from the data-structure later in the pop_all implementation. Another important feature is that the object has to allow to match a candidate without immediately evaluating the appropriate branch. It has to give the pop_all method a chance to remove the element from the data-structure first before the arbitrary user code from the branch is evaluated. That is especially important if the data-structure is thread-safe and does locking, then it cannot hold the lock while it runs arbitrary user code. Firstly it limits the concurrency since no other operation can be executed on the data-structure and secondly it can lead to deadlocks since the common recommendation is to never call a user code while still holding an internal lock.

Probably the simplest implementation which would allow the use-case work is to make case in without a message given behave as a syntax sugar for following.

```
case
in [/A/, b]
  b.succ
end
# turns into
```

```
-> value do
  case value
  in [/A/, b]
    -> { b.succ }
  end
end
```

Then the implementation of pop_all could then look as follows.

```
def pop_all(pattern)
  each do |candidate|
    # assuming each locks the structure to read the candidate
    # but releases the lock while executing the block which could
    # be arbitrary user code

    branch_continuation = pattern.call(candidate)
    if branch_continuation
      # candidate matched
      delete candidate # takes a lock internally to delete the element
      branck_continuation.call
    end
  end
end
```

In this example it never leaks the inner lock.

Second use case which somewhat expands the first one is to be able to implement receive method of the concurrent abstraction called Actors. (receive blocks until matching message is received.) Let's consider an actor which receives 2 Integers adds them together and then replies to an actor which asks for a result with [:sum, myself] message then it terminates.

```
Actor.act do
  # block until frist number is received
  first = receive case
                  in Numeric => value
                    value
                  end
  # block until second number is received, then add them
  sum = first + receive case
                        in Numeric => value
                          value
                        end
  # when a :sum command is received with the sender reference
  # send sum back
  receive case
          in [:sum, sender]
            sender.send sum
          end
end
```

It would be great if we could use pattern matching for messages as it is used in Erlang and in Elixir.
The receive method as the pop_all method needs to first find the first matching message in the mailbox without running the user code immediately, then it needs to take the matching message from the Actor's mailbox (while locking the mailbox temporarily) before it can be passed to the arbitrary user code in the case branch (without the lock held).

If case in without message is first class it could be useful to also have shortcut to define simple mono patterns.

```
case
in [:sum, sender]
  sender.send sum
end
# could be just
in [:sum, sender] { sender.send sum }

case
in ["fatal", _] -> message
 message
end
# could be just, default block being identity function
in ["fatal", _]
```

Then the Actor example could be written only as follows:

```
Actor.act do
  # block until frist number is received
  first = receive in Numeric
```

```
  # block until second number is received, then add them
  sum = first + receive in Numeric
  # when a :sum command is received with the sender reference
  # send sum back
  receive in [:sum, sender] { sender.send sum }
end
```

(2) Matching of non symbol key Hashes

This was already mentioned as one of the problems to be looked at in future in the RubyKaigi's talk. If => is taken for as pattern then it cannot be used to match hashes with non-Symbol keys. I would suggest to use just = instead, so var = pat. Supporting non-Symbol hashes is important for use cases like:

1. Matching data loaded from JSON where keys are strings

```
case { "name" => "Gustav", **other_data }
in "name" => (name = /^Gu.*/), **other
    name #=> "Gustav"
  other #=> other_data
end
```

1. Using pattern to match the key

```
# let's assume v1 of a protocol sends massege {foo: data}
# but v2 sends {FOO: data},
# where data stays the same in both versions,
# then it is desirable to have one not 2 branches
case message_as_hash
in (:foo | :FOO) => data
  process data
end
```

Could that work or is there a problem with parsing = in the pattern?

Note about in [:sum, sender] { sender.send sum }

in [:sum, sender] { sender.send sum } is quite similar to -> syntax for lambdas. However in this suggestion above it would be de-sugared to -> value { case value; in [:sum, sender]; -> { sender.send sum }} which is not intuitive. A solution to consider would be to not to de-sugar the branch into another inner lambda but allow to check if an object matches the pattern (basically asking if the partial function represented by the block with a pattern match accepts the object). Then the example of implementing pop_all would look as follows.

```
def pop_all(pattern)
  each do |candidate|
    # assuming each locks the structure to read the candidate
    # but releases the lock while executing the block which could
    # be arbitrary user code

    # does not execute the branches only returns true/false
    if pattern.matches?(candidate)
      # candidate matched
      delete candidate # takes a lock internally to delete the element
      pattern.call candidate
    end
  end
end
```

What are your thoughts?
Do you think this could become part of the first pattern matching release?

#23 - 06/09/2019 12:43 AM - mame (Yusuke Endoh)

pitr.ch (Petr Chalupa)
Please briefly summarize your proposal first. Use case is important, but explaining a proposal by use case is difficult to read (unless the use case is really simple).

I'm unsure but I guess your proposal:

1) Add a syntactic sugar: case in <pattern>; <expr>; ...; end → -> x { case x in <pattern>; <expr>; ...; end }
2) Allow hash rocket pattern: { <pattern> => <pattern> }
3) Add a syntactic sugar: in <pattern> { <expr> } → -> x { case x in <pattern>; <expr>; end }

The following is my opinion.

I don't like (1). It would be more readable to write it explicitly:

```
data.pop_all do |entry|
  case entry
  in ["fatal", message]
    deal_with_fatal message
  in ["error", message]
    deal_with_error message
  end
end
```

We need to be careful about (2). If => pattern is allowed, we can write a variable as a key, but it brings ambiguity.

```
h = { "foo" => 1, "bar" => 1 }

case h
in { x => 1 }
  p x #=> "foo"? "bar"?
end
```

I think the current design (allowing only symbol keys) is reasonable.

(3) will introduce syntactic ambiguity. Consider the following example.

```
case x
in 1
  {}
in 2
  {}  # Is this the second pattern?  Or is this a lambda?
end
```

It looks like this case statement has two clauses, but in 2 {} (a lambda expression you propose) can be also parsed.

#24 - 06/14/2019 12:16 PM - pitr.ch (Petr Chalupa)

I've intentionally focused on use-cases because: I wanted to make clear why Ruby should to support this; and I've included a possible solution only to make it more clear, since I think Kazuki or somebody else who also spend significant time working on this is the best person to figure out how to support it.

Regarding yours comments.

(1) What I am proposing is different in one key aspect, it allows the pop_all method to execute its own code after the pattern is matched but **before** the branch with the user code is executed. Therefore the pattern of only wrapping case in in lambda is not sufficient to implement this.

To clarify I have described two approaches: First case in <pattern>; <expr>; ...; end => -> x { case x in <pattern>; -> { <expr> }; ...; end }. Second is without the inner lambda case in <pattern>; <expr>; ...; end => pm = -> x { case x in <pattern>; <expr>; ...; end } however an object in pm has to have a accepts?(value) method for the pop_all method to be able to execute just the pattern part without the bodies. I think the second is better.

(2) Thanks for pointing out the ambiguity problem. Id like to explore ideas how to deal with the ambiguity in { x => 1 }:

- Simply raising error when the hash cannot be matched exactly. Could often fail too surprisingly.
- x would be an array of values. What it should be if the value is only one though? It would produce mixed results unless always checked with in { x => 1 } if Array === x
- Evaluate the branch for each matched x. Combination explosion could be a problem for [{a=>1},{b=>2},...], would it run the branch for each value combination assigned to a and b variables?
- Force users to write { *x => 1 } when there is possibility of multiple results. Then x would be Array of all the matched keys (could be just one). This is however not that simple: { (*k = /^pre/) => *v } applied to {'pre-a'=>1, 'pre-b'=>2} could be k=%w[pre-a pre-b] and v=[a,2]; and then even more complicated { 1..10 => *([_, second]) } applied to {1=>[1,:a],2=>[2,:b]} would produce second=[:a,:b]. Feels like this should be possible to define precisely.

(3) Indeed the shortcut I've proposed would be problematic, could it rather be -> in <pattern> {} then?

#25 - 06/14/2019 11:40 PM - ktsj (Kazuki Tsujimoto)

- Related to Feature #15865: `<expr> in <pattern>` expression added

#26 - 06/14/2019 11:40 PM - ktsj (Kazuki Tsujimoto)

- Related to Feature #15918: Pattern matching for Set added

#27 - 06/14/2019 11:41 PM - ktsj (Kazuki Tsujimoto)

- Related to Feature #15881: Optimize deconstruct in pattern matching added

#28 - 06/14/2019 11:42 PM - ktsj (Kazuki Tsujimoto)

- Related to Feature #15824: respond_to pattern for pattern match added

#29 - 06/27/2019 12:20 AM - pvande (Pieter van de Bruggen)

As suggested by Yusuke on Twitter, I'm posting a link to my own personal "wishlist" around pattern matching. I'm happy to discuss any points that might benefit from clarification.

https://gist.github.com/pvande/822a1aba02e5347c39e8e0ac859d752b

#30 - 11/08/2019 05:28 AM - ktsj (Kazuki Tsujimoto)

pitr.ch (Petr Chalupa)

Matz rejcted your proposal.
Please check https://bugs.ruby-lang.org/issues/15930 for more details.