Ruby has borrowed concepts/idioms from allot of languages.

I am proposing borrowing a feature from Forth to provide for compile time interpretation of Constants. This should make executed code faster|efficient, while maintaining source code brevity|clarity.

Below is actual code used in a large rubygem I have.

To develop this method, I had to do allot of test runs to determine the range values. Once found, these values don't change, but I just kept the computed forms of the values, in case I want to upgrade them. In Forth I can interpret those expressions that result in constants, which will be compiled as single values for run time.

See wikeipedia article on Forth below starting at Mixing states of compiling and interpreting.

https://en.wikipedia.org/wiki/Forth_(programming_language)

Forth was designed for, and is still used most frequently, in hardware controllers, and with microprocessors. IMHO this feature would also make MRuby more code efficient and faster for this domain too, and IOT devices.

Below is an example of real code that would benefit from this. While this example would result in numerical constant, string constants could also be interpreted.

def select_pg(endnum, startnum)
    start_num = end_num
    end_num = endnum; start_num = startnum
    range = end_num - start_num
    pg = 5
    if start_num <= Integer.sqrt(end_num) # for one array of primes upto N
        pg = 7 if end_num > 50 * 10**4
        pg = 11 if end_num > 305 * 10**5
    else # for split array cases
        pg = 7 if ((10**6 ... 10**7).include?(range) && start_num < 10**8) ||
                  ((10**7 ... 10**8).include?(range) && start_num < 46 * 10**8) ||
                  ((10**8 ... 10**9).include?(range) && start_num < 16 * 10**10) ||
                  (range >= 10**9 && start_num < 26 * 10**12)
        pg = 11 if ((10**8 ... 10**9).include?(range) && start_num < 55 * 10**7) ||
                  (range >= 10**9 && start_num < 45 * 10**9)
    end
    primes = [2, 3, 5, 7, 11, 13].select { |p| p <= pg }
    {primes, primes.reduce(:*)} # [excluded primes, modpg] for PG
end

Allowing for compile time interpretation, the code could be rewritten as below.

def select_pg(endnum, startnum)
    start_num = end_num
    end_num = endnum; start_num = startnum
    range = end_num - start_num
    pg = 5
    if start_num <= Integer.sqrt(end_num) # for one array of primes upto N
        pg = 7 if end_num > 50 * 10**4
        pg = 11 if end_num > 305 * 10**5
    else # for split array cases
        pg = 7 if (((10**6 ... [10**7]).include?(range) && start_num < [10**8])) ||
                  (((10**7 ... [10**8]).include?(range) && start_num < [46 * 10**8]) ||
                  (((10**8 ... [10**9]).include?(range) && start_num < [16 * 10**10]))) ||
(range >= [10**9] & & start_num < [26 * 10**12])
pg = 11 if (((10**8) ... (10**9)).include?(range) & & start_num < [55 * 10**7]) ||
(range >= [10**9] & & start_num < [45 * 10**9])
end
primes = [2, 3, 5, 7, 11, 13].select { |p| p <= pg }
{primes, primes.reduce(:*)} # [excluded primes, modpg] for PG
end

This maintains the original form, so if I need/want to change the range limits again
I can just change the calculation inline, without having to remember where those values came from.

As 3.0 has introduced many new features and idioms, this could be introduced with no breaking change too.
Old code would work as before, while new code could take advantage of this feature.

Thanks in advance of giving this proposal serious consideration.

Related issues:
Related to Ruby master - Feature #8804: ONCE syntax

History
#1 - 12/26/2020 09:40 PM - chrisseaton (Chris Seaton)
Why do we need a new syntax to do this? Couldn't the compiler already work out 10**6 at compile-time, and invalidate the constant if Integer#** is
redefined?

Or is the idea that [...] will contain expressions that potentially have side effects?

#2 - 12/27/2020 02:36 AM - jzakiya (Jabari Zakiya)
My intent is to ensure the operations needed to create the constant values are performed at compile time (not runtime) and the results are compiled to
use at runtime.

It doesn't seem like in cases using multiple operations, as in the examples given, this is the case.

In Forth, the ""] is a word that starts an interpolation of what's up to "][]", and inlines the resultant constant.

In Ruby you'd use some other syntax, maybe like [[...]], to distinguish from cases like arry = [], et al.

This is a very powerful feature that can make source code (especially for numerical algorithms) much simpler to write and optimize because the code
can show exactly what the operations are that can be interpreted into constants.

If there are existing ways to already do this I was not aware of them, and would like to see examples of them.

I hope this answers your questions.

#3 - 12/27/2020 02:39 AM - chrisseaton (Chris Seaton)
But why do you need to specially mark expressions as constant? 10**6 is already obviously constant (modulo the redefinition of #** which we could
deal with separately.) Why not evaluate everything that is obviously constant at compile-time, whether it's marked with a special syntax or not?

Finding things which are constant seems like it should be the compiler's job, not the programmer's.

#4 - 12/27/2020 03:08 AM - jzakiya (Jabari Zakiya)
But is that the case?

Does 10**6 get's used at runtime as 1_000_000?

What about something like Math.sqrt(Math.log(2*PI))? This is a constant that should be determined at compile time.

Providing semantics to allow users to write these expressions in code and to explicitly tell the compiler to do this provides them certainty and more
control of the runtime. This is preferable than creating magic numbers that are the result of such calculations but people reading the code don't know
where they come from.

My goal is to eliminate any of the math operations having to be done at runtime, and just use the pre-determinable results.

#5 - 12/27/2020 03:17 AM - chrisseaton (Chris Seaton)

What about something like Math.sqrt(Math.log(2*PI))?
Isn't that obviously a constant as well? Or at least Math.sqrt(Math.log(2*Math::PI)) is. (Again, apart from any method redefinition that can be handled separately.)

Some Ruby compilers can already automatically turn this into a constant at compile-time if it appears in your program source code, like TruffleRuby:

```
> Math.sqrt(Math.log(2*Math::PI))
Yes! Truffle can constant fold this to 1.3556832470785147
```

provides them certainty and more control of the runtime

I think that's more of a valid argument.

But really I think you're talking about an optimisation which should happen automatically and transparently and the user shouldn't be responsible for doing it.

#6 - 12/27/2020 03:41 AM - jzakiya (Jabari Zakiya)
I think you're missing my point.

I agree 100% the compiler(s) should do it, but they don't, not across all CRuby versions, and other VMs.

I am proposing standard semantics to provide for users they can ensure this happens going forward.

#7 - 12/27/2020 01:44 PM - jzakiya (Jabari Zakiya)
Also to reiterate, this "feature" also applies to string constants.

So 
```
["Hello World".reverse * 3]
```

would be interpolated at compile time with its results used at runtime.

I would imagine no current Ruby VM does this.

This has been a part of Forth since its creation (by Chuck Moore) in 1970.

No compiler (writer) could ever catch all the possible ways expressions could be reduced to constant values.

This is a powerful feature for users.

As Ruby has been introducing other features to experiment with, now would be the time to introduce this.

Again, this would also be a great feature for MRuby for the domain of usages it targets.

#8 - 12/27/2020 04:08 PM - Eregon (Benoit Daloze)
- Status changed from Open to Rejected

There is no "compile time" for Ruby, and there is no way to execute arbitrary Ruby code at any other time than runtime.

Simply use constants if you want to ensure things are computed once, or rely on the JIT if it's simple enough:

```
MILLION = 10**6
```

#9 - 12/27/2020 10:28 PM - marcandre (Marc-Andre Lafortune)
- Assignee set to matz (Yukihiro Matsumoto)
- Status changed from Rejected to Open

```
tldr; I feel that the possibility to have "inline shareable constant expressions" could improve expressiveness and allow for easier optimizations.
```

```
There is no "compile time" for Ruby, and there is no way to execute arbitrary Ruby code at any other time than runtime
```

This is true, but there is "parse time" and there is also "first time" like /...#{run_once}.../

I believe the request is for something similar to the "o" mode, but for general expressions.

I'll note that the code example above creates Range objects every time it runs too, while these would gain in being created only once. The example given would need about 20 constants, some of them being difficult to name.

Moreover, constants would ideally use private_constant, making the resulting code very verbose.

Finally, while the above constants are mostly Integer and Range (frozen), there is one Array (can be frozen implicitly with #shareable_constant_value)
and there could be user types (e.g. Set) that can only implicitly frozen with experimental mode currently.

If one does not freeze the constant, then the code will run, until someone tries it from a non-main Ractor.

In short, I find the use-case compelling, and the current solution less than ideal:

```
MILLIONS = 10**6 .. 10**7
TENS_OF_MILLIONS = 10**7 .. 10**8
# ... (what is good name for 16 * 10**10??)
private_constant :MILLIONS, :TENS_OF_MILLIONS #, ...
```

```
def select_pg(endnum, startnum)
    # ...
end
```

I've seen multiple cases of code with inline [...] that were constant and would have benefited from extraction to a named constant, at the cost of verbosity. I've also requested a dedicated syntax for such sets of symbols, or strings in #16994, for that purpose.

#10 - 12/27/2020 10:29 PM - marcandre (Marc-Andre Lafortune)
- Assignee deleted (matz (Yukihiro Matsumoto))

(Unassigning, as there is no acceptable proposal yet)

#11 - 12/28/2020 02:22 AM - Dan0042 (Daniel DeLorme)
Some time ago I thought of basically the same idea, but in the end decided that constants were the appropriate way to handle this, so I didn't make it a proposal.

But the cases presented by Marc-Andre are remarkably compelling. It's the first time I hear about the Regexp o modifier, but deduplicated strings have a similar function to this proposal, and Array literals would benefit from that kind of optimization too, not to mention Sets as proposed in #16994.

So this proposal can "unify" various proposals and parts of the language into a single reusable concept, and I find that's usually the sign there's something very worthwhile here.

Let's say that we use $(expr) to denote these "global single-eval expressions", and of course they have to be made ractor-shareable as Marc-Andre pointed out. So instead of introducing multiple special-case syntaxes for basically the same purpose, we could have something like:

```
#{1+2}        #equivalent /#{1+2}/o
"foo"         #equivalent to "foo" or "foo".freeze
{%w[x y z]}.include?(v)
$(Set["x","y","z"]).include?(v)
```

Yes, I'm warming up to the idea.

#12 - 12/28/2020 04:57 AM - nobu (Nobuyoshi Nakada)
Regarding "once" syntax, I've had an idea to use BEGIN.

```
BEGIN /#{1+2}/        #equivalent /#{1+2}/o
BEGIN Set["x","y","z"]).include?(v)
BEGIN {{10**6 ... 10**7}}.include?(range)
```

I haven't considered freezing the result, so no equivalent to "foo".

#13 - 12/28/2020 12:32 PM - Eregon (Benoit Daloze)
marcandre (Marc-Andre Lafortune) wrote in #note-9:

Moreover, constants would ideally use private_constant, making the resulting code very verbose.

Doesn't private_constant (no arguments) work? (if not, sounds worth adding)

If we'd want to do this, then deep freezing seems essential.
Execute once and cache is an anti-pattern if it returns something mutable.

If the once-expression can capture state, it's also a problem. Here is a trivial example:

```
def m(i)
    $("foo_#{i}"
end
m(1) # => "foo"
The original example is not compelling to me, because it's something a JIT can easily fold. And escape analysis can avoid allocating those Ranges.

So basically we'd manually encode in the source code something a JIT can already do. Such code will anyway always run quite a bit faster with a JIT able to see through these operations, with or without once-expression.

Overall, it feels un-Ruby to me. It's a manual low-level optimization hack, just to avoid the need to name a constant.

#14 - 12/28/2020 04:48 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-13:

Doesn't private_constant (no arguments) work? (if not, sounds worth adding)

No it doesn't work; "warning: private_constant with no argument is just ignored"

Being able to set the default visibility independently for methods and constants would feel a bit weird to me.

If the once-expression can capture state, it's also a problem. Here is a trivial example:

I don't think it's really a problem. Here's a counter-example:

```
def log(msg)
  t = Time.now
  t -= $t
  puts "#{t} #{msg}"
end
```

log "a" #=> 0.0  a
log "b" #=> 0.000565776  b

Now, I'm not saying this is a particularly good example, but I don't see a reason to artificially impose a restriction.

So basically we'd manually encode in the source code something a JIT can already do.

Agree but... can a JIT optimize Set["x","y","z"].include?(v) so that the Set is only allocated once?

#15 - 12/28/2020 04:52 PM - chrisseaton (Chris Seaton)

Agree but... can a JIT optimize Set["x","y","z"].include?(v) so that the Set is only allocated once?

Absolutely it could - TruffleRuby will today already optimise ['x', 'y', 'z'].include?('y') to be true and it doesn't allocate anything. It doesn't optimise your exact example - I didn't look into why but if it's important to people I'm sure it could be done.

#16 - 12/28/2020 06:30 PM - jzakiya (Jabari Zakiya)

Your example:

```
def m(i)
  $({"foo#{i}"})
end
```

would not work because it doesn't evaluate at parse-time to a constant value. It requires a runtime parameter i, thus it can't evaluate to a constant value/object it can be substituted with. This example should throw an error.

Overall, it feels un-Ruby to me. It's a manual low-level optimization hack, just to avoid the need to name a constant.

Matz keeps saying Ruby has to advance to keep relevant. Ruby has made allot of additions to the language recently to do that, some still experimental, some not fully/efficiently implemented yet.

IMHO, this feature falls in line with recent additions, e.g. endless methods, infinite ranges: (1..).each, and Ractors. They were all proposed, and eventually included, to make Ruby easier/better for users. This proposed feature/capability exists, in some form, in other languages besides Forth, where it originated.

I appreciate others see the potential utility of this feature, and once an implementation has been created, we can then empirically evaluate its impact (on all levels), and see and measure them directly. I'm sure this will allow code to create faster runtimes, which is a major goal for Ruby 3 over its ancestors.

Now users can develop code faster/easier and maintain showing the details of the code developments without performance hits, and can tell the
parser which expressions to evaluate at parse-time, and substitute in place to use at runtime. This is much easier/efficient than doing a postcode
analysis to see which expressions could be extracted into constant runtime values/objects (which is rarely done), while maintaining source code
brevity and conciseness.

#17 - 01/04/2021 02:50 AM - Dan0042 (Daniel DeLorme)
Similar/related to #8804

#18 - 01/04/2021 12:35 PM - Eregon (Benoit Daloze)
- Related to Feature #8804: ONCE syntax added

#19 - 01/04/2021 12:37 PM - Eregon (Benoit Daloze)
jzakiya (Jabari Zakiya) wrote in #note-16:

would not work because it doesn't evaluate at parse-time to a constant value.

It's not possible to evaluate any Ruby code except literals at parse time, as I already said in https://bugs.ruby-lang.org/issues/17474#note-8.
C++ has a concept of compile-time expressions/constants (constexpr), but it doesn't make any sense for Ruby, since most expressions in Ruby need
the runtime to be evaluated.

I think this feature would just confuse people.
Constants are a thousand times clearer, and simpler.

#20 - 01/04/2021 01:18 PM - chrisseaton (Chris Seaton)
Another point to consider - does this have an impact on our ability to use optimisations such as lazy parsing and do you know how you would
precisely specify the semantics of when the expression is executed in relation to other events?

#21 - 01/04/2021 04:47 PM - Dan0042 (Daniel DeLorme)
Eregon (Benoit Daloze) wrote in #note-19:

It's not possible to evaluate any Ruby code except literals at parse time, as I already said in https://bugs.ruby-lang.org/issues/17474#note-8.
It's completely possible for ruby to just eval that special expression separately from the rest of the file. It would be somewhat equivalent to
TOPLEVEL_BINDING.eval. It could eval either while parsing, or after the current file has parsed and evaled. And of course, while jzakiya intends for a
"parse time" expression, that doesn't exclude the possibility of "first time" which has slightly different but still very close semantics.

Constants are a thousand times clearer, and simpler.

Except when they're not... it depends on the code, on the situation.
chrisseaton (Chris Seaton) wrote in #note-20:

Another point to consider - does this have an impact on our ability to use optimisations such as lazy parsing and do you know how you would
precisely specify the semantics of when the expression is executed in relation to other events?

Going with "first time" semantics this would not be an issue, but going with "parse time" semantics I think it would make sense to say that precisely
"when" the expression is executed is undefined. So in the case of lazy parsing it would be parsed and evaled when the containing code is parsed, whenever that is.

#22 - 01/07/2021 01:36 PM - Eregon (Benoit Daloze)
Dan0042 (Daniel DeLorme) wrote in #note-21:

I think it would make sense to say that precisely "when" the expression is executed is undefined. So in the case of lazy parsing it would be
parsed and evaled when the containing code is parsed, whenever that is.

"undefined" is for me a synonym of broken, incompatible, useless semantics.
The semantics need to be clearly defined, otherwise there will be incompatibilities.

Literally "parse time" doesn't make sense, because e.g., one can parse once to bytecode, and reuse the bytecode without parsing for future
executions.
So I guess what is meant is "load/require time", similar to when constant would be defined, before evaluating any code in the file.

There is no good way to decide if e.g. 10 ** 6 is pure and safe to evaluate during load time, because ** could be redefined and potentially check the
caller.

09/25/2021
It also seems very unnatural that the expression couldn't use anything in its evaluation context, not even constants, for example this wouldn't work if done at load time:

```ruby
module A
  N = 10
  def self.foo
    $(N ** 6)
  end
end
```

So that's why evaluating on first execution is somewhat sensible, but at parse/load time I believe it's not.

#23 - 01/07/2021 04:40 PM - shevegen (Robert A. Heiler)

New year, new comment! :D

jzakiya wrote:

Ruby has borrowed concepts/idioms from allot of languages.

This is true; ruby has always been multi-paradigm and "more than one way", even though I think its core strength has been "flexible" OOP. Or at least that is the style variant that is closer to the way I think; sort of like Alan Kay's "objects everywhere". I'd even like to think of Erlang/Elixir as a bit of "distributed objects", but the syntax is weird ... even elixir's syntax is a bit weird. Ruby's syntax is, for the most part, IMO the best among all programming languages - not all of the syntax perhaps, but the subset one can use.

However had there are also many idioms ruby has NOT integrated.

So I don't think you can generalize from the addition of some features, while not ignoring how other features/changes were rejected in the past or were not a great fit; matz sometimes said this before, such as the perl $ variables that are very short (I can never remember them offhand; I always have to look at a cheat sheet to remember what they do).

mame once explained this in another thread in regards to feature suggestions and discussions, e. g. when matz does not like a particular feature. :-)

(It's also explained in the very old interview matz gave about ruby's philosophy and orthogonality of features/ideas, so this is a trade-off that exists AT THE SAME TIME in regards to "more than one way". The "more than one way" never meant to be a 'have so many ways nobody understands anything now". Ruby is still designed. I think an example for this is the pattern matching change. For me personally it is too complicated to use, but I understand the use case and "batch-variable assignment" that you can do via pattern matching.)

My own personal style in ruby is much closer to a "prototyping", flexible/dynamic OOP variant, without much restrictions to it (so I don't really use private, excluding a few libraries).

Other ruby users use a different style, such as zverok with a more functional-centric style approach - which is quite creative too. For me personally, while it is an interesting style, it's a style I can not see myself wanting to use; for similar reasons I don't use haskell. It's simply way above my brain powers; more importantly I can't be bothered to want to have to think about code that much, I just want to write something and move on. Monad discussions are fun to have, though. :-)

jzakiya wrote:

Thanks is advance of giving this proposal serious consideration.

Ultimately you only have to convince matz. The ruby core team tends to recommend to focus on the use case(s) - evaluating the use cases, or potential use cases, helps a LOT. But matz also commented on other proposals before, where people making a suggestion often don't evaluate trade-offs or potential drawbacks from a suggestion. You can see these problems with the type discussion; some love types and want them everywhere. Others don't love types and don't want them everywhere.

In regards to the proposal here, I personally don't really see why that is necessary, but your mileage may vary.

One problem I see in general is that ruby branches out a lot and includes many disparate "functionality", and syntax changes, some of which may all end up with an orthogonal design (I don't mind it when I can avoid it, so I just focus on the "subset" of ruby that I use). I am not saying this should be the primary point to consider, but it sort of creates a problem when you add more and more features that are not quite "the ruby way", whatever that way is.

jzakiya wrote:

As 3.0 has introduced many new features and idioms, this could be introduced with no breaking change too.

I don't think this can be a criterium as to why any new functionality should HAVE to be added, merely because OTHER functionality/feature was added before. My recommendation would be on what the proposal adds, irrespective of any other feature that was added.

Forth is an interesting language but ruby is not forth. There is no "compilation" as such in MRI, although it would be interesting for this as a new, or VM-like language on top of ruby (e. g. some meta-ruby language that people could target; then you could add a "layer" that the type-liking folks focus on, without this modifying MRI itself, such as the .must() suggestion or similar changes). IMO the main focus of MRI is still sort of like an "improved
perl" and these use cases. :-) 

Or perhaps python; I feel these are all part of the same "family" of languages.

I am not matz, so I don't make the decisions, but I think matz much prefers to keep the "ruby philosophy" as it was (so, within that "family" of languages, in regards to MRI itself), while adapting ruby to current usage patterns in the present day - I guess ractors are sort of a way to move forward there, e. g. more easily taking advantage of multiple cores and trying to avoid the GIL block.

Otherwise, I think Benoit really gave a great statement in regards to the suggestion here, that I personally agree with most in this regard:

I think this feature would just confuse people. Constants are a thousand times clearer, and simpler.

I think so too. Then again I also prefer simplicity at all times. And strict definitions. :-)

(I do like ruby being dynamic and flexible too, though. But for my own code quality, and in my own projects, I want to focus on simplicity, and being strict helps a LOT, IMO. For example, specifications. I always found it simplifies so many other things when specifications are strict, and correct at all times.)

So my personal opinion: I don't think this would be a good change. But as said, ultimately you only have to convince matz; it may be interesting to see what matz has to say about this feature or similar proposals in this regard. See nobu's code example (although I have to admit, I also don't like the suggested syntax. I do, however had, use END sometimes, which is a great feature. I avoid "begin" and "after" or whatever the names were ... BEGIN/END or something, though, as I always feel as if I lose control over my own code whenever I use it. I noticed this in a gem, where this was triggered from someone else using it, and I did not know how to avoid or prevent it. And I dislike when I have no control over ruby code, so I also started to avoid that. But I digress.)