Ruby - Feature #21015

Add in a `-g` flag, like `-s` but with a few more quality of life features

01/08/2025 01:50 AM - sampersand2 (Sam Westerman)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Add in a -g flag, like -s but with a few more QOL features

TL;DR

(PR.)

Ruby's -s flag has quite a few shortfalls that make it difficult to use when writing scripts . A new -g flag will fix it:

```
# Support `-abc` as a shorthand for `-a -b -c`
ruby -ge'p [$a, $b, $c]' -- -abc #=> [1, 1, 1]
# ^ (see "How short-form options (`-abc`) are handled" for why it's `1` and not `true`)

# Support `-vvv` as a shorthand for `-v=3`
ruby -ge'p $v' -- -vvv #=> 3

# Support things other than strings:
ruby -ge'p [$n, $f]' -- -n90 -f=false #=> [90, false]

# long-forms don't have a leading `_`
ruby -ge'p $foo_bar' -- --foo-bar #=> true
```

Background: What is -s

Ruby's -s flag is an extremely helpful feature when writing short little scripts to set config options; it automatically processes ARGV and removes leading "flags" for you, assigning them to global variables.

```
#!/bin/ruby -s

# echo, ruby-style! If `-n` is given, no newline is printed
print ARGV.join(' '), ($n ? "" : "\n")
```

While -s is significantly less powerful than OptionParse, or even via just parsing options yourselves, it's **incredibly** useful when writing simple little scripts where the option parsing code would be longer than the script itself.

Problem 1 (The big one): No support for chained short-form flags

The biggest problem with -s is that it doesn't let you concatenate short-form flags together: conventionally, ./myprogram -x -y -z should be the same as ./myprogram -xyz. However, if ./myprogram were a ruby program using -s, you'd end up with the global variable $xyz.

Short scripts that use -s to parse options thus need to add the following code to handle all different permutations of -x, -y, and -z:

```
# Handle all permutations of `-xyz`
$xyz || $xzy || $yxz || $yzx || $zxy || $zyx and $x=$y=$z=true

# Handle only two options given
$xy || $yx and $x=$y=true
$xz || $zx and $x=$z=true
$yz || $zy and $y=$z=true
```

Four flags becomes even worse:

```
$wxyz || $wxzy || $wyxz || $wyzx || \
    $wzxy || $wzyx || $xwyz || $xwzy || \
    $xywz || $xyzw || $xzwy || $xzyw || \
    $ywxz || $ywzx || $yxwz || $yxzw || \
    $yzwx || $yzxw || $zwxy || $zwyx || \
    $zxwy || $zxyw || $zywx || $zyxw and $w=$x=$y=$z=true

$wxy || $wyx || $xwy || $xyw || $ywx || $yxw and $w=$x=$y=true
$wxz || $wzx || $xwz || $xzw || $zwx || $zxw and $w=$x=$z=true
$wyz || $wzy || $ywz || $yzw || $zwy || $zyw and $w=$y=$z=true
$xyz || $xzy || $yxz || $yzx || $zxy || $zyx and $x=$y=$z=true

$wx || $xw and $w=$x=true
$wy || $yw and $w=$y=true
$wz || $zw and $w=$z=true
$xy || $yx and $x=$y=true
$xz || $zx and $x=$z=true
$yz || $zy and $y=$z=true
```

This is a huge problem for simple little scripts, as they're forced into an uncomfortable choice:

1. Use OptionParser, which is very much overkill for tiny scripts
2. Break from unix standards and require passing short-forms individually (i.e. ./program.rb -x -y -z)
3. Do the cumbersome permutation checks as shown above
4. Give up on flags all together and use environment variables.

None of these options are great, *especially* because Ruby's all about programmer happiness and none of these options spark joy.

Problem 2: All values are Strings

Less important than the short-form issue is that all values provided to flags become Strings (e.g. ruby -se'p $foo' -- -foo=30 yields "30"). While this can be solved by $foo = $foo&.to_i somewhere early on, it's verbose:

```
#!ruby -s

# Very simple benchmarking program
$log_level = $log_level&.to_i
$amount = $amount&.to_i
$timeout = $timeout&.to_i
$precision = $precision&.to_i

$amount.times do |iter|
  start = Time.now

  $log_level > 1 and puts "[iteration: #{iter}] running: #{ARGV}"
  pipe = IO.popen ARGV

  unless select [pipe], nil, nil, $timeout
    warn "took too long!"
    next
  end

  printf "%0.#{$precision}f", Time.now - start
end
```

Moreover, there's no way to create a falsey flag other than just omitting it, which can make interacting with -s scripts somewhat irritating:

```
# Have to use `*[... ? ... : nil].compact`, otherwise
# we'd end up passing an empty string/nil as an arg
system("./myprogram.rb", *[enable_foo ? "--foo" : nil].compact)
```

A better solution would be to allow for --foo=false or --foo=true, and then parse the false/true.

Problem 3 (Minor): Long-form options have a leading _

Long-form options, such as --help, have a leading _ appended to them (to disambiguate them from -help). While it can be worked around (either alias $help $_help or just using $_help directly), it's irritating enough that I've been known to just force end-users to use -help. This diverges from the common "long-form options should have two dashes in front of them," further making ruby scripts with -s a bit awkward to use

Problem 4 (Minor): Repeated flags aren't supported

Sometimes it's useful to know how many times a flag was repeated, such as -vvv to enable "very very verbose mode": Using -s you must do

```
is_verbose = $v ? 1 : ($vv ? 2 : ($vvv ? 3 : ($vvvv ? 4 : ...)))
```

While I've yet to see a use-case for repeating a flag beyond three or four times, having to manually enumerate everything out is, again, a bit of a pain.

Solution: Add a new -g flag, which supports all this

I propose the addition of a new command-line flag, -g, which adds in a new form of argument parsing that solves all these issues. PR

Overview:

```
ruby -ge'p [$a, $b, $c]'    -- -abc                 #=> [1, 1, 1]
ruby -ge'p $d'              -- -d90                 #=> 90
ruby -ge'p [$d, $e]'        -- -d90e=foo            #=> [90, "foo"]
ruby -ge'p [$hi, $foo_bar]' -- --hi --foo-bar=false #=> [true, false]
ruby -ge'p $x'              -- -xxx                 #=> 3

# Putting it together now, lol.
ruby -g -e'p [$a, $b, $c, $d, $e, $world]' -- -abbcd90e=hello --world=false
# => [true, 2, true, 90, "hello", false]```
```

Specifics:

After all ruby arguments are parsed, just like -s, "switch parsing" is enabled. Just like -s, switches are read until -- or a non-switch (ie doesn't start with -) is encountered. The key differences from -s are *how* the switches are handled.

How short-form options (-abc) are handled:

The entire point of this feature, short-form options are handled in a more posix-like style: each character (ASCII-only, akin to -s) is parsed as a switch of just one character long. So -abc is handled the same exact way as -a -b -c. This allows for chaining short-form options together when calling ruby scripts in a convenient and natural way.

There's a few caveats: First, if an = is encountered, the last-most character gets that value. So -abc=foo is the same as -a -b -c=foo. (This is akin to most command-line options, eg ruby's ruby -aine..., the e gets the ....)

Secondly, because numbers aren't valid global variables ($1..$9 are used for regex groups), I think repurposing them to have an implicit = beforehand is beneficial. So you can do -n90 and it'd be the same as -n=90. It even supports signs, so -n+39 is -n=+39. (In fact, the implementation I have right now supports -a123b as -a=123 -b, which I'm a fan of.)

Lastly, to support -vvv implying "$v = 3", repeating a character sets that variable to the repetition count. A consequence of this is that for -x, -g will set $x to 1 (unlike -s which sets $x to true). This is convenient, as otherwise puts "log!" if $v > 1 wouldn't really work well. And, since they're both truthy, I think it's ok. However, I could be swayed to just drop this entirely.

How long-form options (--foo) are handled:

Unlike short-form options, long-form ones are handled much more closely to how -s handles them, with one key difference: The leading _ is dropped. (In -s, all - other than the very first is converted to a _, so --foo is $_foo, and ---bar is $__bar.) I feel like it's much more natural to have --long-form be equivalent to the global variable $long_form, especially when -x is equivalent to $x.

(Note: Unlike -x, --foo will set $foo to true, not 1.)

Parsing after =s

The -s flag allows supplying values to switches, such as -foo=abc will set $foo to "abc". However, this is a bit annoying when you

want to use integers (or booleans) as the values, as you have to manually check yourself.

So, when -g encounters a switch with a =, if the value is exactly true, false, or nil, it'll use that value (so --foo=false sets $foo to false, or -x=nil sets $x to nil). Otherwise, it'll attempt to parse the value as an integer, using standard ruby integer parsing rules (i.e. allows signs, underscores, and prefixes like 0x). This allows users to pass in --count=30 and then use ($count || 10) in their code, and not have to do any conversions.

While this implicit conversion usually'd not be great, I think it actually fits quite well: The whole point of -s (and -g) are for short scripts, which presumably aren't doing a large amount or robust error handling. If error handling's needed, then -g's not the right tool and OptionParser or something else should be used. (And, if you need a string always, you can just $count = $count&.to_s and get it anyways.)

Alternatives

(TODO)

1. Do nothing
2. Continue using -s
3. Create some function in the runtime, or on $*/ARGV, that does parsing

Prior art

Ruby's -s is based off of perl's -s, which functions (as far as I can tell) identically to Ruby's. I Don't know of any other language that does -s, much less -g.

Open Questions

How should supplying both -g and -s work?

This is the first flag that'd directly conflict with another command-line flag, so what should be done when both -g and -s are given (such as ruby -gs -e'p $x' -- -x=9). Here's the options as I see them:

1. Use to last supplied flag (in the example -s). This'd act like how incompatible flags work in other utilities
2. Always use -g, as it can be considered a sort-of a "super set" of -s.
3. Emit a warning, and then do either 1. or 2.
4. Emit an error, and then do either 1. or 2.

I'm personally partial to emitting a warning on -W2, and then using the last supplied flag, however I could be convinced to any of the options

What values after = should be parsed?

Currently, if the values are exactly true, false, nil it uses those literals; otherwise, it attempts to parse an int, and if that fails, uses a string.

Should additional types be supported, such as Floats, Symbols, or Pathname? I personally think no, as anything that complicated can be handled by $x and $x = Float $x or just OptionParser

Should -x do $x = true or $x = 1

I personally'd like -x to be true, just like -s. However, this conflicts with -xx yielding 2, as log if $x > 1 wouldn't work. Since 1 is truthy, I've defaulted -x to 1, but I could be convinced to remove it (and even remove the entire -xx thing if there was a good argument.)

Why introduce a new flag? Why name it -g?

I think adding in a new flag makes sense: -g is a modification of -s, so it naturally should be a new flag. (Attempting to shoehorn these options into -s would be a nightmare.)

I briefly considered using -ss as the flag name, to make it clear that it was a modification of -s, however that'd be the first two-character short flag and I didn't want to introduce that. (And, it'd be pretty ironic as a large portion of the impetus behind -g is to parse short flags, which -ss isn't :-P.)

As for -g specifically, I considered -o for "options" (nixed because most utilities use it for "output," and I didn't want the cognitive

overload), and -f for "flags" (nixed because some utilities use it to mean "file," and ruby might use it in the future.) I picked -g because it's short for "globals" or "getflags".

Conclusion

Ruby's all about programmer happiness, and making writing programs easier. Currently, -s provides a very rudimentary argument parser, and a slightly more sophisticated one is sorely needed.

History

#1 - 01/08/2025 01:50 AM - sampersand2 (Sam Westerman)

- Subject changed from Add in a `-g` flag, like `-s` but with a few more quality of life features to [DRAFt] Add in a `-g` flag, like `-s` but with a few more quality of life features

#2 - 01/08/2025 01:56 AM - sampersand2 (Sam Westerman)

- Description updated

#3 - 01/08/2025 02:22 AM - nobu (Nobuyoshi Nakada)

sampersand2 (Sam Westerman) wrote:

> # Prior art
>
> (TODO) Perl has -s, and it works like Ruby's -s. IDK of any other languages which even attempt this.

Inversion. Ruby borrowed it from Perl.

#4 - 01/08/2025 03:15 AM - nobu (Nobuyoshi Nakada)

- Subject changed from [DRAFt] Add in a `-g` flag, like `-s` but with a few more quality of life features to [DRAFT] Add in a `-g` flag, like `-s` but with a few more quality of life features

#5 - 01/08/2025 05:03 AM - sampersand2 (Sam Westerman)

- Description updated

#6 - 01/08/2025 05:19 AM - sampersand2 (Sam Westerman)

- Subject changed from [DRAFT] Add in a `-g` flag, like `-s` but with a few more quality of life features to Add in a `-g` flag, like `-s` but with a few more quality of life features

- Description updated

#7 - 01/08/2025 05:30 AM - sampersand2 (Sam Westerman)

- Description updated

#8 - 01/08/2025 06:13 AM - zenspider (Ryan Davis)

For the record, I love this and want it very badly.

I advocated for -ss to mean "an extension of -s", much like some CLIs do -vv to mean "more verbose". This would mean "more switches"... Since sflag is already an integer and ruby -ss is nonsensical I figured it wouldn't conflict with anything... but -g is perfectly fine.

I proposed that the values default to 1, such that -vv is easily supported and you never have to typecheck or cast anything from true, eg:

```
puts extra_debugging_info if $v > 1
```

I think this feature would fill 80% of my use cases for options processing, esp for single file scripts (which i write a lot of).

#9 - 01/09/2025 10:14 AM - matz (Yukihiro Matsumoto)

I like the basic idea of this proposal. But, I am against:

- -g option name. -s is relatively unpopular. Even with the proposed improvement, this is kinda old-fashioned way to parse command-line options. I don't want to consume precious single character option here.
- Adding up the multiple options. If you want more precise interpretation of the command line options, use optparse or something similar.
- Interpreting option values. Conversion should be done by the application, even for the basic values like nil, true, false and integers.

Matz.

#10 - 01/11/2025 12:26 AM - sampersand2 (Sam Westerman)

matz (Yukihiro Matsumoto) wrote in #note-9:

> I like the basic idea of this proposal. But, I am against:
>
> - -g option name. -s is relatively unpopular. Even with the proposed improvement, this is kinda old-fashioned way to parse command-line options. I don't want to consume precious single character option here.
> - Adding up the multiple options. If you want more precise interpretation of the command line options, use optparse or something similar.
> - Interpreting option values. Conversion should be done by the application, even for the basic values like nil, true, false and integers.
>
> Matz.

- Adding up the multiple options. If you want more precise interpretation of the command line options, use optparse or something similar. Totally agree with this. It's a bit awkward, and I think removing it is a good idea. 100% on-board.

- Interpreting option values. Conversion should be done by the application, even for the basic values like nil, true, false and integers. While a bit less of a fan of this, I'm ok dropping this, because it's not a huge burden for applications to do $foo == 'true'.

- -g option name. -s is relatively unpopular. Even with the proposed improvement, this is kinda old-fashioned way to parse command-line options. I don't want to consume precious single character option here.

Few things here:

1. I'd argue the *reason* -s is relatively unpopular is because it's cumbersome and awkward to use---IMO, if it supported -abc == -a -b -c from the outset, I think it'd see a lot more use.
2. While this is a somewhat old-fashioned way to parse arguments, I think that it's quite useful when writing shorter scripts. Compare the following, for example:

```
#!ruby -G
while line = gets
  $stdout.flush if $f
  line.chomp! if $l
  print line.dump.slice!(1..-2).gsub(/\\([#'"']), '\1')
  puts if $l
end
puts unless $l || $n
```

and

```
#!ruby
require 'optparse'
OptionParser.new do |op|
  op.on '-n' do $n = true end
  op.on '-l' do $l = true end
  op.on '-f' do $f = true end
  op.parse!
end

while line = gets
  $stdout.flush if $f
  line.chomp! if $l
  print line.dump.slice!(1..-2).gsub(/\\([#'"']), '\1')
  puts if $l
end
puts unless $l || $n
```

The fact that the option parsing in the second OptionParser example is just as long as the entire program in the -G example is quite cumbersome. I think there's definitely a niche for "small scripts which just need to support parsing a handful of flags"

1. If -g is considered precious, would -G or --switches be acceptable alternatives? We currently have 28 unused flags (bfgjkmoqtuzABDGHJLMNOPQRTVYZ), and of the 26 currently supported flags (0CEFIKSUWXacdeghilnprsvwxy), 55% of them are still shared with ruby 0.49 (FIXacdeilnpsvxy). I'd argue adding a single extra short or long flag to support this use-case would be reasonable.

I've also considered adding a singleton method on $*/ARGV (just like how $LOAD_PATH has resolve_feature_path, however using this would need to be done in BEGIN blocks for scripts using -n/`-p).