Composition

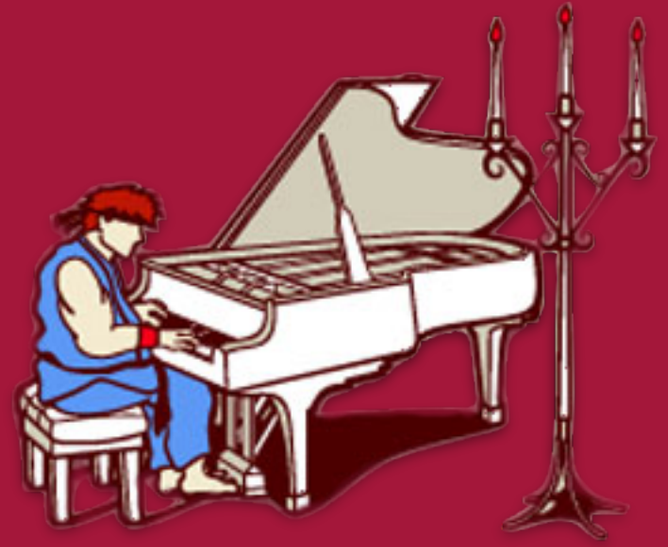
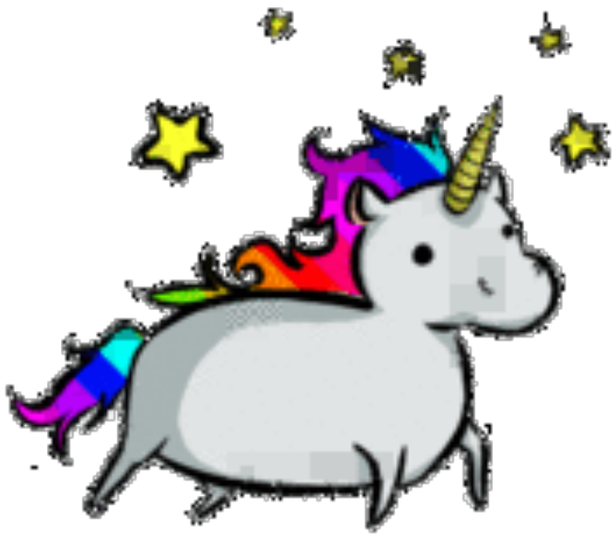


Image credit: davidsoames.com

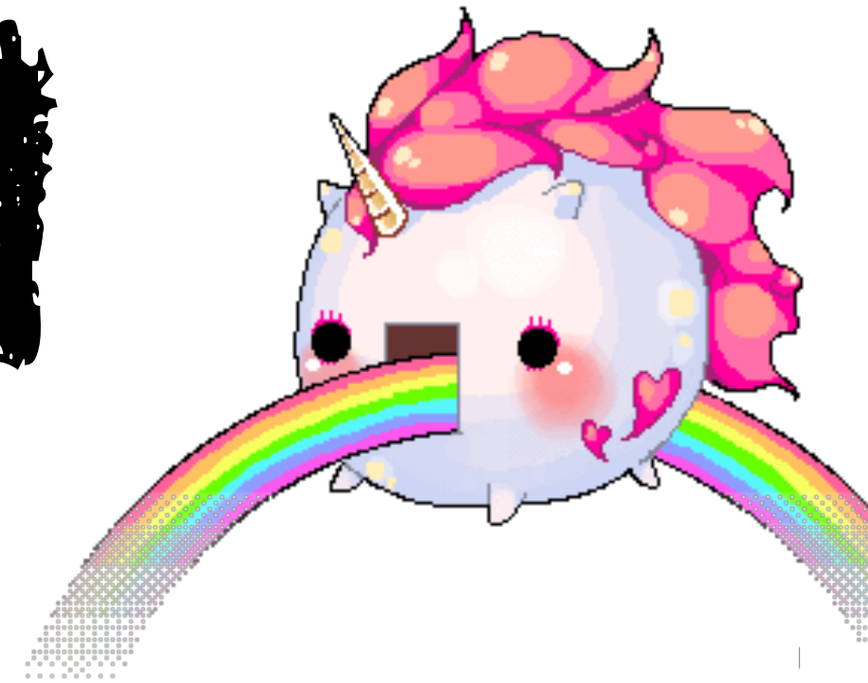
Composition

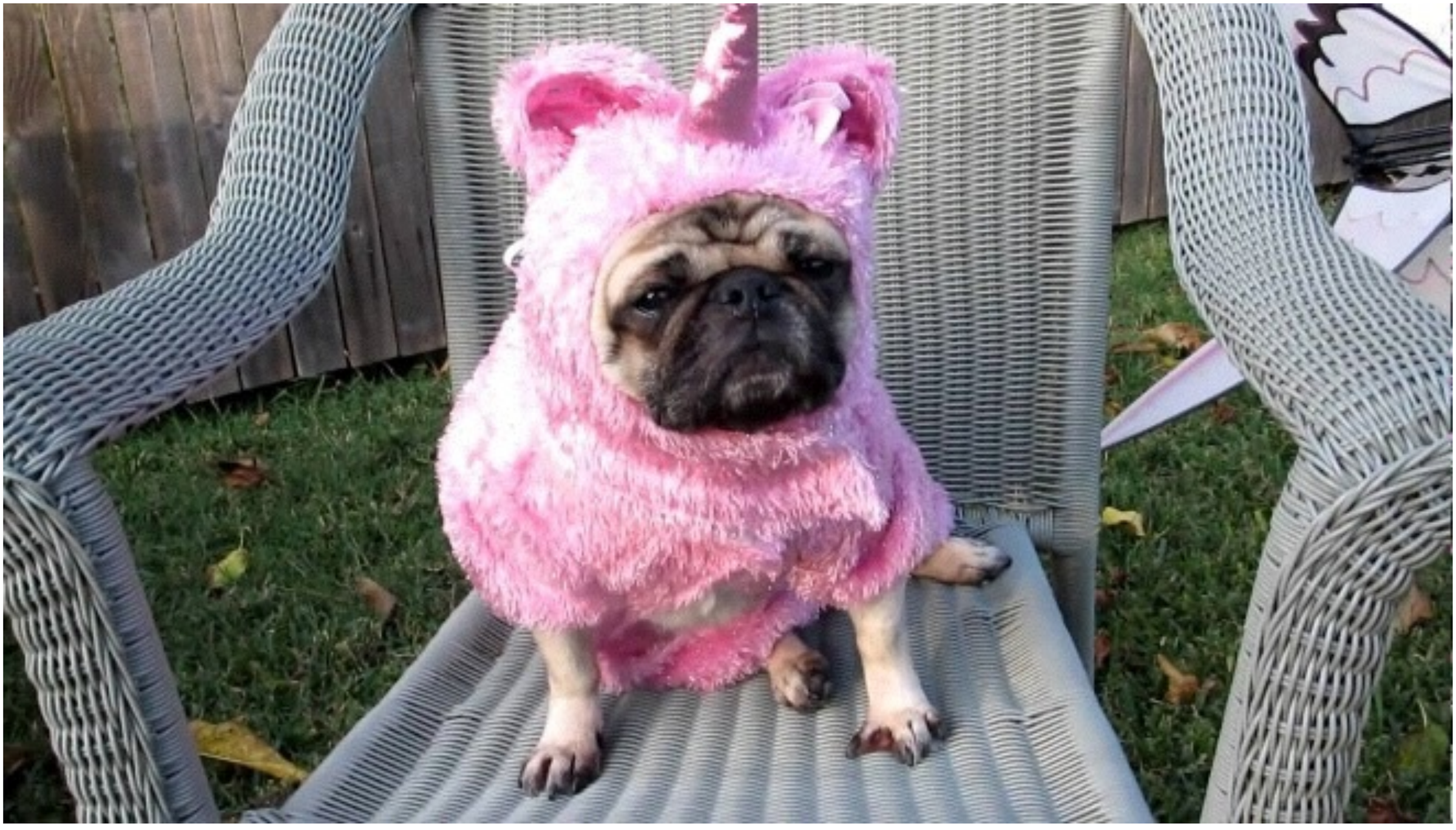
or:

Neatly Gluing Things Together



FUN WITH BLOCKS





```ruby
add_one = proc do |x|
  x + 1
end
```

```ruby
puts add_one.call(1)
# => 2
```

```ruby
add_one = lambda do |x|
  x + 1
end
```

```ruby
puts add_one.call(1)
# => 2
```

```
add_one = -> (x) {
  x + 1
}
```

```
puts add_one.call(1)
# => 2
```

```
add_one = -> (x) {
  x + 1
}
```

```
puts add_one.call(1)
# => 2
```

```ruby
class AddOne
  def call(x)
    x + 1
  end
end
add_one = AddOne.new

puts add_one.call(1)
# => 2
```

Composition

```
add_one = ->(x) { x + 1 }
```

```
add_one = ->(x) { x + 1 }
double  = ->(x) { x * 2 }
```

```
add_one = ->(x) { x + 1 }
double  = ->(x) { x * 2 }


add_one_and_double =
  compose(add_one, double)
```

```
add_one = ->(x) { x + 1 }
double  = ->(x) { x * 2 }


add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
```

```
add_one = ->(x) { x + 1 }
double  = ->(x) { x * 2 }


add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
```

```
add_one = ->(x) { x + 1 }
double  = ->(x) { x * 2 }


add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```
def compose(f1, f2)
  ->(x){ f2.call(f1.call(x)) }
end

add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```ruby
def compose(f1, f2)
  ->(x){ f2.call(f1.call(x)) }
end

add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```ruby
def compose(f1, f2)
  ->(x){ f2.call(f1.call(x)) }
end

add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```ruby
def compose(f1, f2)
  ->(x){ f2.call(f1.call(x)) }
end

add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```ruby
def compose(f1, f2)
  ->(x){ f2.call(f1.call(x)) }
end

add_one_and_double =
    compose(add_one, double)

add_one_and_double.call(1)
# => (1 + 1) * 2
# => 4
```

```
add_one_and_double =
  compose(add_one, double)

add_one_and_double.call(1)
```

```
class Proc
  def >>(f2); ...; end
end

add_one_and_double =
  (add_one >> double)

add_one_and_double.call(1)
```

```ruby
class Proc
  def >>(f2); ...; end
end

add_one_and_double =
  (add_one >> double)

add_one_and_double.call(1)

# See:
# https://bugs.ruby-lang.org/issues/6284
```

So... Why?

```
# An Asset Pipeline.
# output_css_with:
#   Takes a Proc, which takes
#   a list of CSS text to process.

pipeline.output_css_with(
  -> (files) { files.join("\n") }
)
```

```
concat = -> (x) { x.join("\n") }

pipeline.output_css_with(
  concat
)
```

```
concat = -> (x) { x.join("\n") }
minify = CSS::Minifier.new(
          strategy: :conservative
         )

pipeline.output_css_with(
  concat >> minify
)
```

```
concat = -> (x) { x.join("\n") }
minify = CSS::Minifier.new(
           strategy: :conservative
         )
# ↪ Has call(); expects a string.

pipeline.output_css_with(
  concat >> minify
)
```

```
concat = -> (x) { x.join("\n") }
minify = CSS::Minifier.new(
          strategy: :conservative
         )
prefix = CSS::Compat::Prefix.new

pipeline.output_css_with(
  concat >> prefix >> minify
)
```

```
concat >>  prefix  >> minify
```

```
 concat >>  prefix  >> minify
          ==
(concat >>  prefix) >> minify
```

```
 concat >>  prefix  >> minify
          ==
(concat >>  prefix) >> minify
          ==
   concat >> (prefix  >> minify)
```

```
"abc"  +  "def"  +  "ghi"
```

```
"abc"  +  "def"  +  "ghi"
          ==
("abc"  +  "def")  +  "ghi"
```

```
 "abc"  +   "def"   +  "ghi"
                ==
("abc"  +   "def")  +  "ghi"
                ==
"abcdef"            +  "ghi"
```

```
 "abc"  +   "def"   +  "ghi"
                ==
("abc"  +   "def")  +  "ghi"
                ==
"abcdef"            +  "ghi"
                ==
"abcdefghi"
```

```
"abc"  +   "def"    +  "ghi"
                ==
"abc"  +  ("def"    +  "ghi")
                ==
"abc"  +   "defghi"
                ==
"abcdefghi"
```

```
"abc"  +  "def"  +  "ghi"
```

==

. . . . . .

==

```
"abcdefghi"
```

"Associativity"

```
def shrinking
  standard_minify = CSS::Minifier.new(...)
  rewrite_colours = MyMagic::Shrink.colours
  standard_minify >> rewrite_colours
end

def compatibility
  prefix  = CSS::Compat::Prefix.new
  flexbox = MyMagic::Polyfills.flexbox
  grid    = MyMagic::Polyfills.grids
  grid >> flexbox >> prefix
end

pipeline.output_css_with(
  concat >> shrinking >> compatibility
)
```

```
even     = ->(x)   { x % 2 == 0 }
add      = ->(x,y){ x + y }.curry
add_ten  = ->(x)   { add.call(10) }
list     = ...

Pipeline = Filter.new(even)
         | Map.new(add_ten)
         | Reduce.new(add)

Pipeline.new(list).value
```

```elixir
# Elixir:

[[1,2],[3,4]]
  |> List.flatten
  |> Enum.map(fn x -> x + 1 end)
# => [2,3,4,5]


# See:
# https://speakerdeck.com/solnic/blending-functional-and-oo-programming-in-ruby
```

```
; Clojure

(map
 (comp C sharp major)
 (concat (range 0 8) (reverse (range 0 7))))
```

```
; See:
; http://www.infoq.com/presentations/music-functional-language
```

```
; Clojure

(map
 (comp C sharp major)
 (concat (range 0 8) (reverse (range 0 7))))
```

```
; See:
; http://www.infoq.com/presentations/music-functional-language
```

```
; Clojure

(map
 (comp C sharp major)
 (concat (range 0 8) (reverse (range 0 7))))

(->>
 bassline
  (with hook)
  (where :pitch (comp low B flat major))
  (with beat)
  (in-time (bpm 90)))))

; See:
; http://www.infoq.com/presentations/music-functional-language
```

Summary

- Convenient.

Summary

- Convenient.
- Build pipelines out of reusable parts. Composable dependency injection?

Summary

- Convenient.
- Build pipelines out of reusable parts. Composable dependency injection?
- Everyone looks at you funny when you use it.

References & Further Reading

- https://speakerdeck.com/solnic/blending-functional-and-oo-programming-in-ruby
- http://www.parsonsmatt.org/currb
- http://www.infoq.com/presentations/music-functional-language
- https://gist.github.com/damncabbage/cdf71ec519db01f2f64b

PS: ...

$$(f \circ g)(x) = f(g(x))$$

$$(f \circ g)(x) = f(g(x))$$


```
(f << g).call(x)
==
f.call(g.call(x))
```

Fin.

Rob Howard
@damncabbage
http://robhoward.id.au