Introductory metaprogramming

Mark Armstrong

November 23, 2020

Contents

1 Metaprogramming 1

2 Disclosure 1

3 Ruby 1

4 Clojure 3

1 Metaprogramming

Metaprogramming, as the name implies, regards programs *that are constructed dynamically*, that is, programs whose construction is (at least partially) carried out at runtime.

2 Disclosure

I am not an expert on metaprogramming; far from it.

This material is intended as a *very introductory* taste of what metaprogramming offers.

It is *not* intended to show particularly *good* uses of metaprogramming. In particular, the Ruby examples might be somewhat dangerous and ill-advised (the Clojure examples should be a little higher in quality.)

3 Ruby

Suppose we want to have a way to run a piece of code on an object twice. We can do this for a particular class fairly easily.

As a first attempt:

```
class MyClass
  def to_s
    "I am an object"
  end

  def twice(&block)
    yield
    yield
  end
end

x = MyClass.new
puts x.twice { puts self }
```

In this first attempt, the `self` in the block refers to `self` at the point the block is written.

We can make use of the `instance_eval` method instead of `yield` to run the block *within the method we are defining*.

```
class MyClass
  def to_s
    "I am an object"
  end

  def twice(&block)
    instance_eval &block
    instance_eval &block
  end
end

x = MyClass.new
puts x.twice { puts self }
```

This is fine for this kind of object. But I can't use it more generally; for instance, I can't do

```
5.twice { puts self }
```

What I need to do is add this method to the `Object` class that all other classes inherit from.

```
class Object

  def twice(&block)
    instance_eval &block
    instance_eval &block
  end
end

5.twice { puts self }
```

We can go further. For instance, we can modify how Ruby responds if we call a method that does not exist for a given object.

```
class Object

  def method_missing(method_name, *args, &block)
    print "Uh-oh; I couldn't find the "
    print method_name
    puts " method!"
    return nil
  end
end

5.do_something
puts "Look ma, I didn't crash!"
```

4 Clojure

In Clojure, we can use *macros* to perform metaprogramming.

For instance, we might miss the `unless` construct which is present in many languages, but not in Clojure. We could try to define it as a function:

```
(defn unless [test then else]
  (if test
    else
    then))

(unless (= 0 0)
        (println "Equality is broken!")
        (println "Nevermind, it's okay."))
```

We can specify the behaviour of `unless` more precisely with a macro.

```
(defmacro unless [test then else]
  `(if ~test
     ~else
     ~then))
```