Creating a DSL for Ploticus

Scott Patten

What's a DSL?

Domain Specific Language

A custom language designed to solve a specific problem

Types of DSLs

- External DSL
  - A friendly way to provide data (configuration or otherwise) to a program
  - Typically text files, XML or YAML
  - Easy to use syntax
  - Examples: Make, Sendmail, YAML files like Rails test fixtures or database.yml
  - Ploticus scripting language

Types of DSLs

- Internal DSL
  - uses the constructs of the programming language itself to define the DSL
  - No clear dividing line between “DSL” and “API”. Synonymous in many ways.
  - This is the kind I'll be talking about today.
  - Examples: Rake, Capistrano, perhaps Rails

Paul Graham:

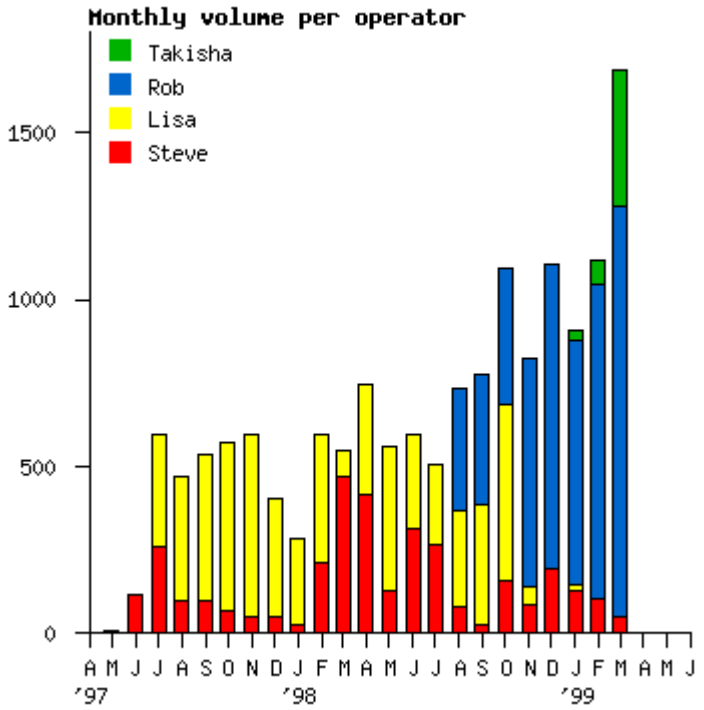
You can magnify the effect of a powerful language
by using a style called bottom-up programming

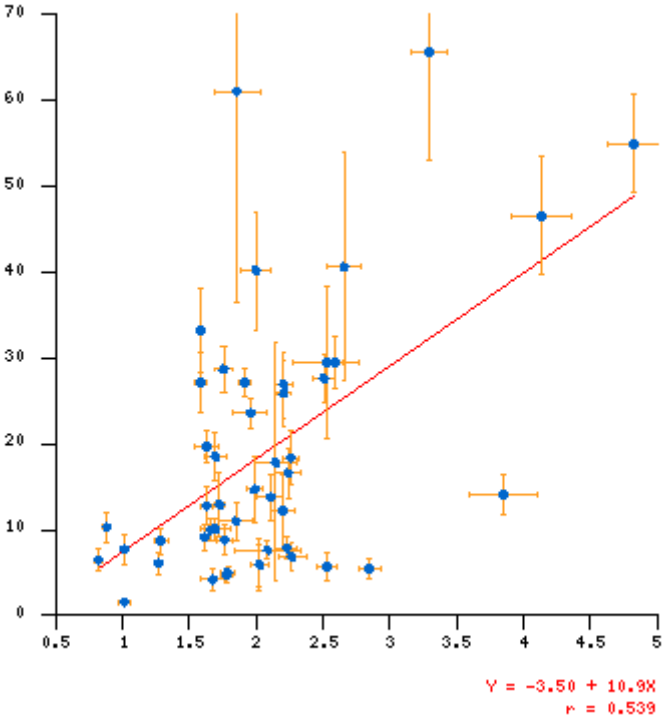
... you write programs in multiple layers,
the lower ones acting as programming languages for those above.

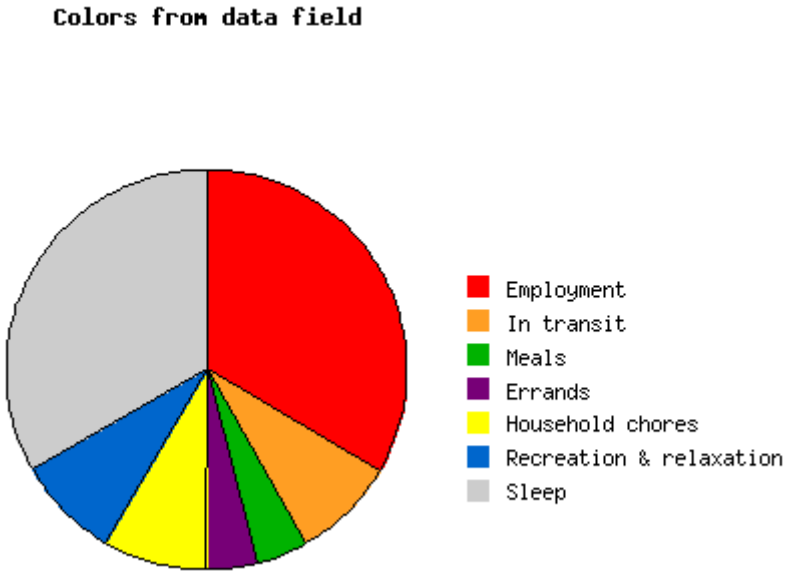
If you do this right, you only have to keep the topmost layer in your head.

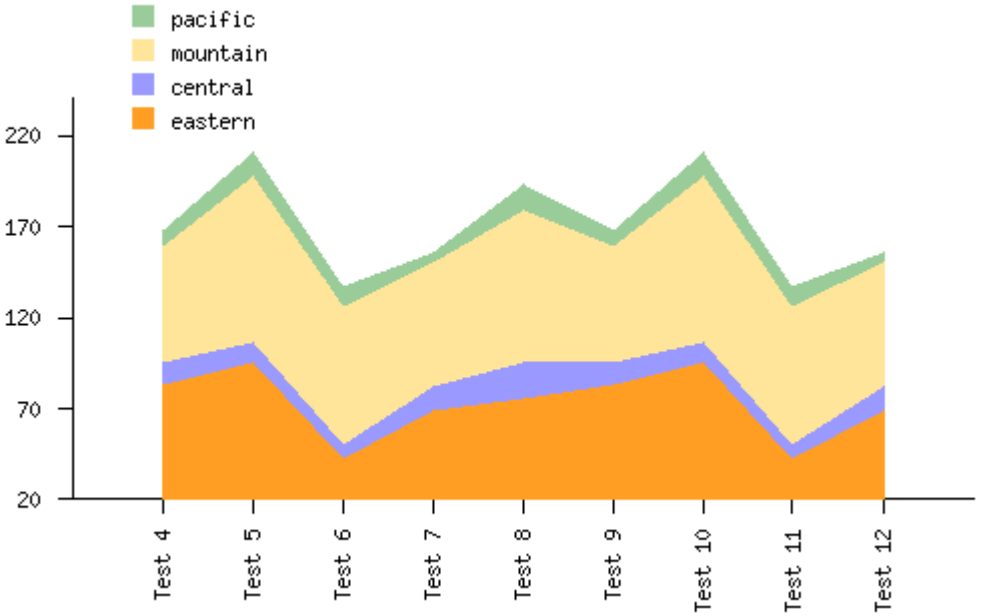
Ploticus

- Command line driven graphing software
- GPL
- Has a very nice script interface
- http://ploticus.sourceforge.net


Monthly volume per operator
Takisha
Rob
Lisa
Steve
1500
1000
500
0
A M J J A S O N D J F M A M J J A S O N D J F M A M J
'97
'98
'99


70
60
50
40
30
20
10
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Y = -3.50 + 10.9X
r = 0.539


Colors from data field
Employment
In transit
Meals
Errands
Household chores
Recreation & relaxation
Sleep

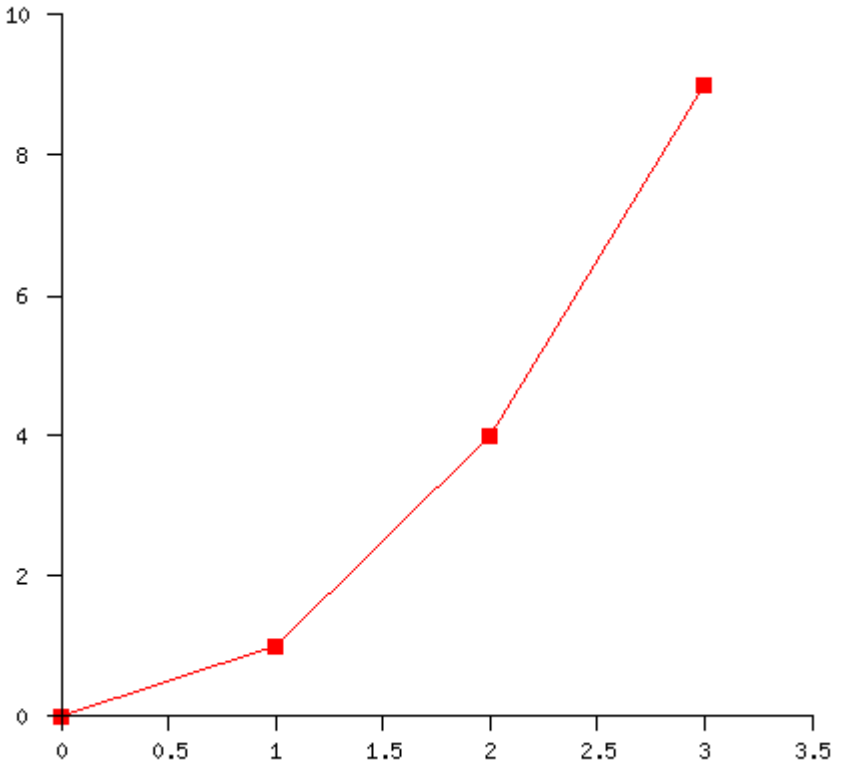

pacific
mountain
central
eastern
220
170
120
70
20
Test 4
Test 5
Test 6
Test 7
Test 8
Test 9
Test 10
Test 11
Test 12

pl -png script.pls

or

pl -png -stdin

```
#proc page
    pagesize: 4.75 4.5

#proc getdata
    data:
            0       0
            10      1
            22      4
            33      9

#proc areadef
    box: 3.75 3.5 //size in inches
    location: 0.5 0.5 // xoffset yoffset of graph
    xautorange: datafield=1
    yautorange: datafield=2,3

#proc lineplot
    xfield: 1
    yfield: 2
    linedetails: color=red width=0.3
    pointsymbol: shape=square color=red

#proc xaxis
    stubs: inc

#proc yaxis
    stubs: inc
```



The ugly way to do it

```
#proc areadef
      box: 3.75 3.5 //size in inches
      location: 0.5 0.5 // xoffset yoffset of graph
      xautorange: datafield=1
      yautorange: datafield=2

#proc lineplot
      xfield: 1
      yfield: 2
      linedetails: color=white width=0.3
      pointsymbol: shape=nicecircle color=green
```

```
ploticus = []
ploticus.push("#proc areadef")
ploticus.push("  box: #{width} #{height}")
ploticus.push("  location: #{xoffset} #{yoffset}"
ploticus.push("  xautorange: datafield=#{xdatafield}
ploticus.push("  yautorange: datafield=#{ydatafield}

ploticus.push("#proc lineplot")
ploticus.push("  xfield: #{xfield}")
ploticus.push("  yfield: #{yfield}")
ploticus.push("  linedetails: color=#{color} width=...
ploticus.push("  pointsymbol: shape=#{shape} ...

ploticus.join("\n")
```

The DSL way to do it

```
#proc areadef
      box: 3.75 3.5
      location: 0.5 0.5
      xautorange: datafield=1
      yautorange: datafield=2


#proc lineplot
      xfield: 1
      yfield: 2
      linedetails: color=white width=0.3
      pointsymbol: shape=nicecircle
                   color=green
```

```
Ploticus.new do

  areadef do
    box [3.75, 3.5]
    location [0.5, 0.5]
    xautorange :datafield => 1
    yautorange :datafield => [2,3]
  end

  lineplot do
    xfield 1
    yfield 2
    linedetails :color => 'red',
                :width => 0.3
    pointsymbol :shape => 'square',
                :color => 'red',
                :style => 'solid'
  end

end
```

The Magic Ingredients:

- method_missing
- Blocks
- instance_eval

```
def method_missing(method, *args)
```

Blocks: chunks of code

```
do |a, b, c|
  code goes in here
end
```

```
{|a, b, c| code goes in here}
```

Blocks

You can send a block to a method when you call it

```
greeting(“Scott”) {|g| puts g}
```

Blocks

You can send a block to a method when you call it

```
greeting("Scott") {|g| puts g}

def greeting(name)
  yield "Hello, #{name}"
end
```

Blocks

You can send a block to a method when you call it

```
greeting(“Scott”) {|g| puts g}

def greeting(name)
  yield “Hello, #{name}”
end

def greeting(name, &block)
  @block = block #save it for later
   block.call(name)
end
```

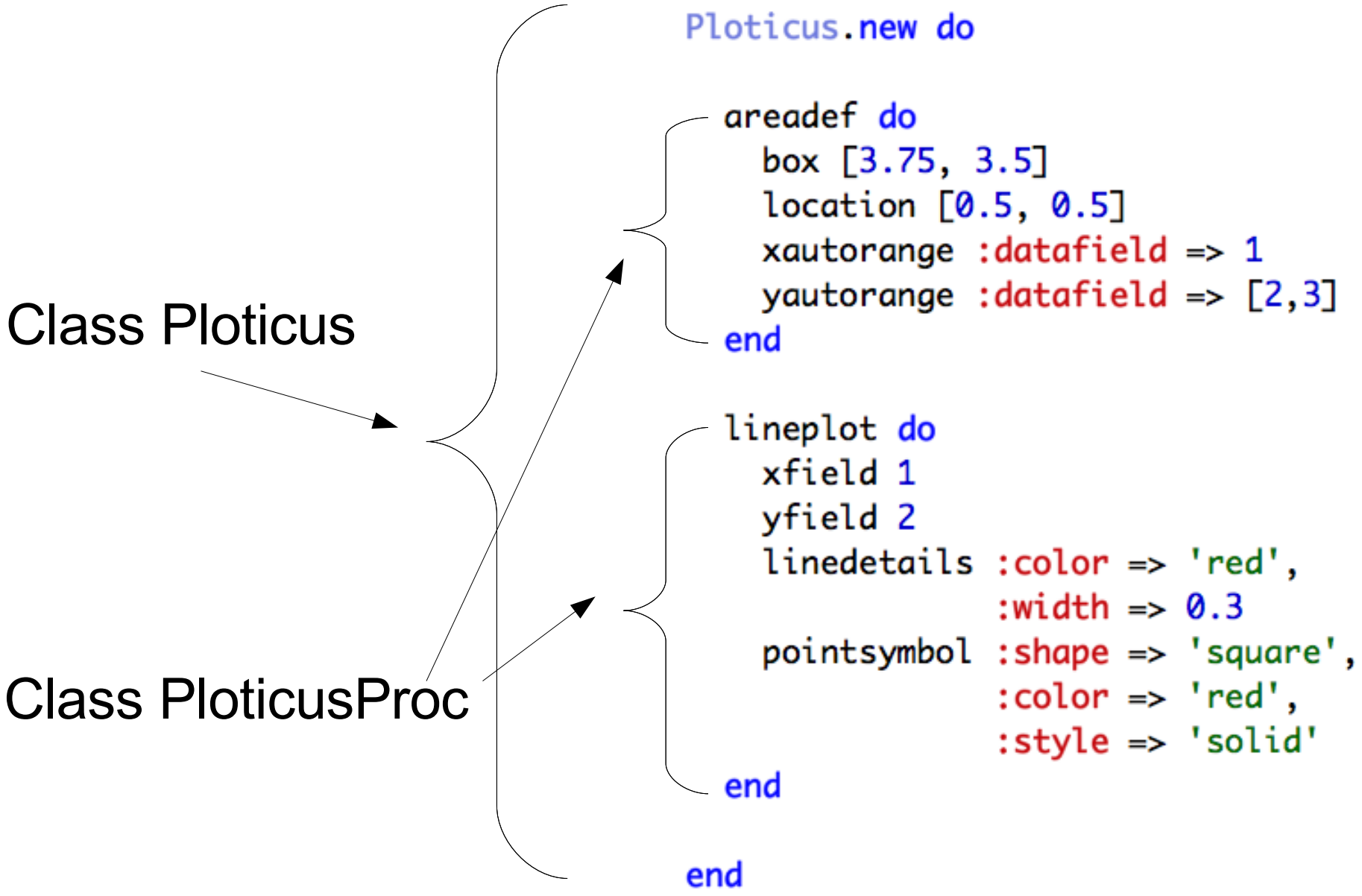
instance_eval

- Two ways of calling it:
  - *obj*.instance_eval(string) or *obj*.instance_eval(&block)
  - *obj*.instance_eval {block}
- Evaluates the string (or block) in the context of *obj*
  i.e. self = *obj,* so calling *some_method* is equivalent to calling *obj.some_method*.

Back to our example.....

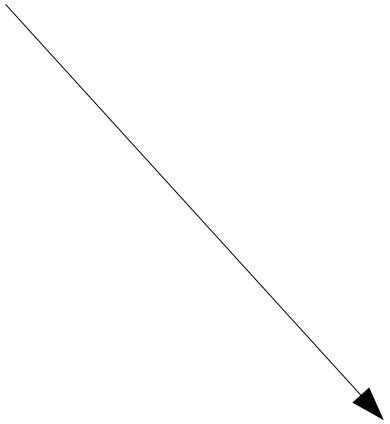
```
#proc areadef
      box: 3.75 3.5
      location: 0.5 0.5
      xautorange: datafield=1
      yautorange: datafield=2


#proc lineplot
      xfield: 1
      yfield: 2
      linedetails: color=white width=0.3
      pointsymbol: shape=nicecircle
                   color=green
```

```
Ploticus.new do

  areadef do
    box [3.75, 3.5]
    location [0.5, 0.5]
    xautorange :datafield => 1
    yautorange :datafield => [2,3]
  end

  lineplot do
    xfield 1
    yfield 2
    linedetails :color => 'red',
                :width => 0.3
    pointsymbol :shape => 'square',
                :color => 'red',
                :style => 'solid'
  end

end
```


Class Ploticus
Class PloticusProc
Ploticus.new do
areadef do
box [3.75, 3.5]
location [0.5, 0.5]
xautorange :datafield => 1
yautorange :datafield => [2,3]
end
lineplot do
xfield 1
yfield 2
linedetails :color => 'red',
:width => 0.3
pointsymbol :shape => 'square',
:color => 'red',
:style => 'solid'
end
end

```ruby
class Ploticus
  attr_reader :received, :sent, :errors, :procs

  def initialize(filetype = 'png', &block)
    @errors = @received = ""
    @procs = []
    @filetype = filetype.gsub('.','')
    if block
      instance_eval(&block)
      run
    end
  end

  def method_missing(method, &block)
    @procs.push PloticusProc.new(method, &block)
  end

  def to_s
    @procs.join("\n\n") + "\n\n"
  end

  def run
    #from http://pleac.sourceforge.net/pleac_ruby/processmanagementetc.html
    #use open4 (not in standard library) if you want to see the exit status
    Open3.popen3("pl -#{@filetype} -stdin") do |stdout, stdin, stderr|
      stdout.puts @sent = self.to_s
      stdout.close #if you don't close stdout, you'll never finish reading stdin or stderr
      @received = stdin.readlines
      @errors = stderr.readlines
    end
  end
```

```ruby
def initialize(filetype = 'png', &block)
  @errors = @received = ""
  @procs = []
  @filetype = filetype.gsub('.','')
  if block
    instance_eval(&block)
    run
  end
end
```



```ruby
Ploticus.new do

  areadef do
    ....
  end

  lineplot do
    ....
  end

end
```

Ploticus#method_missing:

```
def method_missing(method, &block)
  @procs.push PloticusProc.new(method, &block)
end
```

```
Ploticus.new do

  areadef do
    ....
  end

  lineplot do
    ....
  end

end
```

```
@procs = [<PloticusProc:0x352bfc  @name=:areadef, .....>,
          <PloticusProc:0x352850  @name=:lineplot, ..... >]
```

```ruby
class PloticusProc
  attr_reader :res, :name

  def initialize(name, &block)
    @name = name
    @res = []
    instance_eval(&block)
  end

  def method_missing(method, *args)
    @res.push({:name => method.to_s.gsub(/=/,'').to_sym,
      :value => args.collect {|a| pretty(a) }.join(' ')})
  end

  def to_s
    "#proc #{@name}\n#{@res.collect{|r| "  #{r[:name]}: #{r[:value]}"}.join("\n")}"
  end

  private

  def pretty(a, level = :first)
    if a.respond_to?(:each_pair) then a.collect{|k,v| "#{k}=#{pretty(v, level = :second)}"}.join(" ")
    elsif a.respond_to?(:join)
      separator = level == :first ? ' ' : ','
      a.join(separator)
    else a
    end
  end

end
```

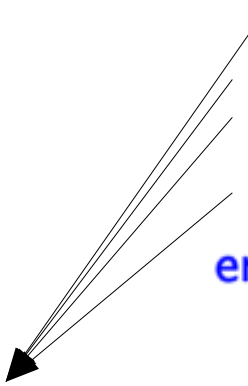
Ploticus#method_missing:

```ruby
def method_missing(method, &block)
  @procs.push PloticusProc.new(method, &block)
end
```

PloticusProc#new

```ruby
def initialize(name, &block)
  @name = name
  @res = []
  instance_eval(&block)
end
```

```ruby
areadef do
  box [3.75, 3.5]
  location [0.5, 0.5]
  xautorange :datafield => 1
  yautorange :datafield => [2,3]
end
```



```ruby
def method_missing(method, *args)
  @res.push({:name => method.to_s.gsub(/=/,'').to_sym,
             :value => pretty(args[0])})
end
```

```ruby
Res = [{:name => 'box', :value => '3.75 3.5'},
       {:name => 'location', :value => '0.5 0.5'},
       {:name => 'xautorange', :value => 'datafield=1'},
       {:name => 'yautorange', :value => 'datafield=2,3'}]
```

```ruby
def pretty(a, level = :first)
  if a.respond_to?(:each_pair) then a.collect{|k,v|
    "#{k}=#{pretty(v, level = :second)}"}.join(" ")
  elsif a.respond_to?(:join)
    separator = level == :first ? ' ' : ','
    a.join(separator)
  else a
  end
end

areadef do
  box [3.75, 3.5]
  location [0.5, 0.5]
  xautorange :datafield => 1
  yautorange :datafield => [2,3]
end
```

If it's a hash

```
#proc areadef
  box: 3.75 3.5 //size in inches
  location: 0.5 0.5 // xoffset yoffset of graph
  xautorange: datafield=1
  yautorange: datafield=2,3
```

```ruby
def pretty(a, level = :first)
  if a.respond_to?(:each_pair) then a.collect{|k,v|
    "#{k}=#{pretty(v, level = :second)}"}.join(" ")
  elsif a.respond_to?(:join)
    separator = level == :first ? ' ' : ','
    a.join(separator)
  else a
  end
end
```

```ruby
areadef do
  box [3.75, 3.5]
  location [0.5, 0.5]
  xautorange :datafield => 1
  yautorange :datafield => [2,3]
end
```

```
#proc areadef
  box: 3.75 3.5 //size in inches
  location: 0.5 0.5 // xoffset yoffset of graph
  xautorange: datafield=1
  yautorange: datafield=2,3
```

If it's an array

Ploticus:

```
def to_s
  @procs.join("\n\n") + "\n\n"
end
```

PloticusProc:

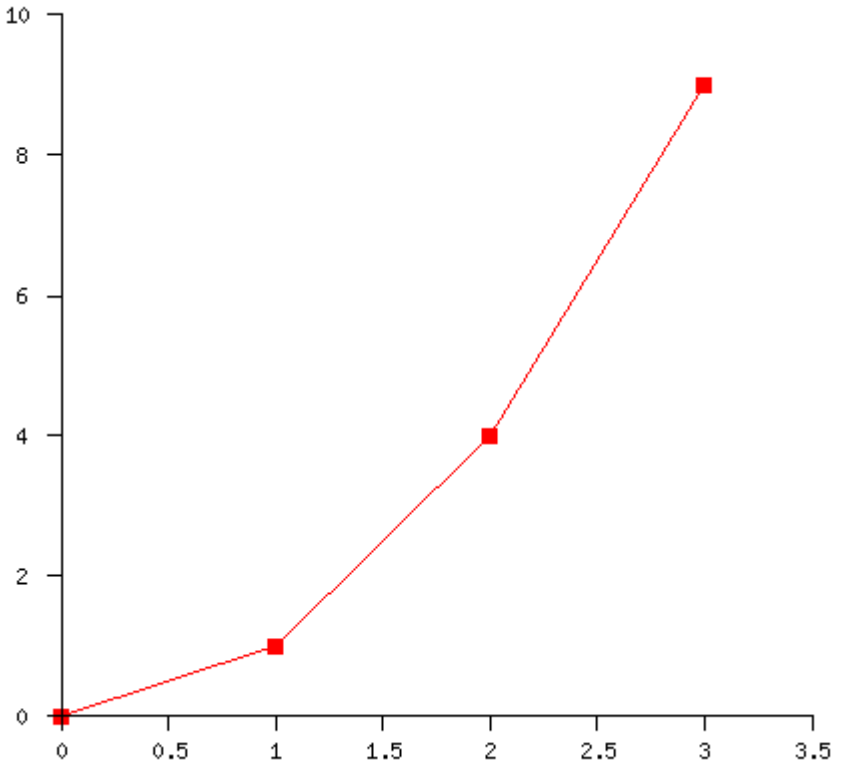
```
def to_s
  "#proc #{@name}\n
     #{@res.collect{|r| "  #{r[:name]}: #{r[:value]}"}.join("\n")}"
end
```

```
#proc areadef
  box: 3.75 3.5 //size in inches
  location: 0.5 0.5 // xoffset yoffset of graph
  xautorange: datafield=1
  yautorange: datafield=2,3
```

Ploticus#run

```ruby
def run
  #from http://pleac.sourceforge.net/pleac_ruby/processmanagementetc.html
  #use open4 (not in standard library) if you want to see the exit status
  Open3.popen3("pl -#{@filetype} -stdin") do |stdout, stdin, stderr|
    stdout.puts @sent = self.to_s
    stdout.close #if you don't close stdout, you'll never finish reading stdin
    @received = stdin.readlines
    @errors = stderr.readlines
  end
end
```

```
#proc page
    pagesize: 4.75 4.5

#proc getdata
    data:
            0       0
            10      1
            22      4
            33      9

#proc areadef
    box: 3.75 3.5 //size in inches
    location: 0.5 0.5 // xoffset yoffset of graph
    xautorange: datafield=1
    yautorange: datafield=2,3

#proc lineplot
    xfield: 1
    yfield: 2
    linedetails: color=red width=0.3
    pointsymbol: shape=square color=red

#proc xaxis
    stubs: inc

#proc yaxis
    stubs: inc
```



See it in action at
www.plotomatic.com