Here's the Julia version of a script we saw a while ago in Python:

```
function writerule(selector; options...)
  println("$selector {")
  for (prop, value) in options
    println("  $(replace(string(prop), "_", "-")): $value;")
  end
  println("}")
end

writerule("h1", font_family="Helvetica", size="20px")
writerule("p.error", color="red", margin="16px", padding="0")
```

Rather than having only global or function scopes, Julia's scoping regions coincide with the following constructs: function bodies, `while` and `for` loops, and blocks introduced with `try`, `catch`, `finally`, `let`, and `type`. Variables introduced in a scope are visible to nested scopes, and shadowing is permitted. We introduce new variables into the current scope with `local` or `const`; function parameters are automatically made part of the function body scope. If you assign to a variable without marking it local, a new local is introduced in the current scope *unless* it has been marked `local` or `global` in an enclosing scope. Marking a variable `global` is required to allow writing to a global variable in an "inner" (non-global) scope.

```
a, b, c = 1, 2, 3       # three globals
(function ()
  a = 10                # introduces local and shadows
  global b = 20         # overwrites global
  d = 10c + 10          # new local, reading global
  local e = 5
  while true
    e = 50              # outer e because marked local
    f = 60              # local to while loop!
    break
  end
  @assert (a,b,c,d,e) == (10,20,3,40,50)
  @assert (try f catch -1 end) == -1
end)()
@assert (a,b,c) == (1,20,3)
```

Finally, Julia has a special rule for variables in comprehensions: new variables are created for each iteration. CoffeeScript, in contrast, iterates with existing variables! Let's compare:

```
# Julia
julia> x = 0
julia> [x^2 for x in 0:9]
10-element Array{Int64,1}
julia> x
0
```

```
# CoffeeScript
coffee> x = 0
coffee> (x**2 for x in [0..9])
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
coffee> x
10
```

That's it for the basics. Julia's type system goes well beyond anything we've seen up to this point, so we'll devote the next section to it.