Comparative Prosodic Analysis of Latin Satire

Lee Butterman

Green's introduction to his translation of Juvenal does not have many comments about his style. He says that 'few Roman poets can equal his absolute control over the pace, tone, and texture of a hexameter', and shows one example from Satire 6.434ff, but leaves it at that. (p 52) He cites Dubrocard, who claims that 2130 words of the 4790 in the Satires are hapax legomena. (p 51; this number turns out to be low.) His response is rather puzzling: 'Seldom can one man's body of work have had less spare fat on it.' Is that an ancient view? Are Homer's formulaic epics obese and ungainly? (And should Vergil's Aeneas no longer be *pius* throughout?) Green further asserts that 'very often his work reads like a series of paragraphs [sic] that bear only a token relationship to each other.' (p 44)

Green acknowledges the beauty of Juvenalian verse, but most of his analysis focuses on themes rather than presentation. In antiquity, though, poetry required artistry, and its criticsm required mention of it. Dionysius of Halicarnassus quotes Sappho 1 in his On Literary Composition, and he remarks immediately after that 'the euphony and charm of this passage lie in the cohesion and smoothness of the joinery; words are juxtaposed and interwoven according to certain natural affinities and groupings of the letters.' Horace's Art of Poetry also emphasizes style: he values smoothness of content, brevity, and clarity (14, 24ff) and is ambivalent towards hapax legomena: ‘you will express yourself wonderfully, if a skillfull setting renders a known word new.’ (47-48)

This paper attempts to explore the similarities and differences among Juvenal, Horace, and Persius, in their vocabulary, meter, and word rhythms, from six similarly-sized selections. Juvenal’s first through third satires comprise `juva`, and his tenth through twelfth comprise `juvb`. Persius’ complete Satires comprises `pers`. Horace’s first through fifth satires from his first book comprise `horacea`, and his first through fourth from his second comprise `horaceb`. For out-of-genre comparison, `vae1` is the first book of Vergil’s Aeneid.

Analysis

One interesting difference between Juvenal, Perseus, and Horace is in their word density per line. Overall, Juvenal is just under 6.5 words per line, whereas Horace is just over 6.7 words/line, and Persius is slightly higher than both. But Juvenal sticks fairly close to the 6- and 7-word line, whereas Persius’s line distribution has less severe of a dropoff on the high end (he even ventures an 11-word line, 1.104). Horace is in between Persius and Juvenal, but he too has extravagantly long lines (a smooth talking pimp in 2.3.232) and extravagantly short (obstacles in 1.2.98 and mock-tragedy in 1.2.1). As a comparison, Book 1 of Vergil’s Aeneid averages 6.45 words per line, mostly staying between 5 and 8.

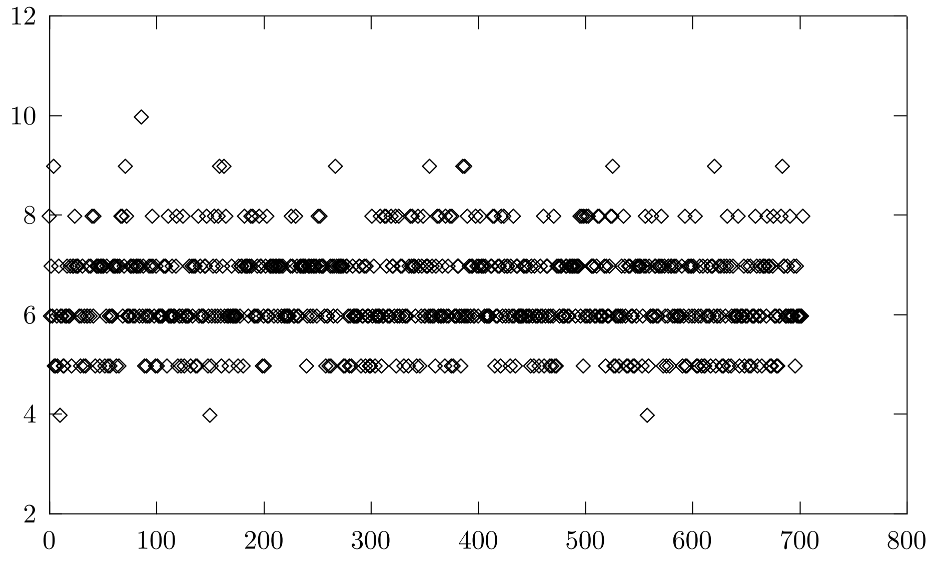

Figure 1: Words per line vs line number, `juvb`.

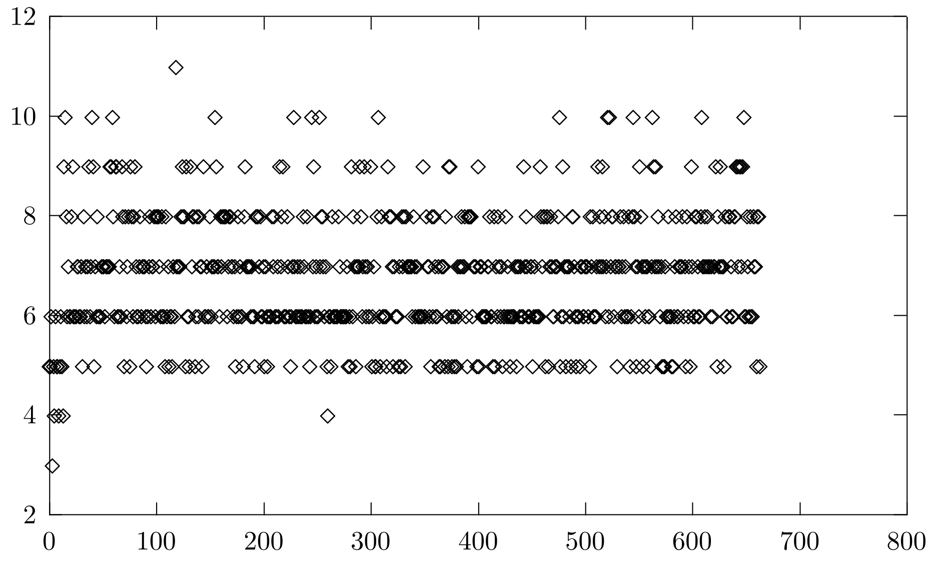

Figure 2: Words per line vs line number, `pers`.

For the vocabulary distribution, the lemmatizer used Perseus's XML dictionary lookup, and was unsuccessful on roughly 2 percent of queries. (On Vergil, only 0.7%; this is probably because dictionaries spend most of their time on the most popular authors.) All unlemmatizable words were counted as hapax legomena. The histograms of vocabulary distribution are computed from every possible lemmatized form of a word: *virum* adds a hit for *vir*, man, and a hit for *virus*, poison, even though *vir* is usually the intended form. But this is a problem throughout, so while each individual individual histogram might be incorrect, they are off by roughly the same amount, just under a factor of 2.

Roughly speaking, a larger written vocabulary lends itself to more variety in grammatical constructions. And also, roughly speaking, the fewer words per line, the easier a text is to get through. This intuition is not far off: the Aeneid Book 1 uses the most repetitive vocabulary (45% of words appear more than twice) and the lines are the least dense, and it is the easiest of all six to read. Horace's Satires are tougher, his lines are denser, and around 38% of his words appear more than twice. Juvenal has more hapax legomena than Persius, but Persius's lines are unusually dense; both are more difficult to read than Horace.

The stability of Vergil's subject matter gives his a clump around 0.01 on the X axis, and this seems like the core vocabulary. Juvenal's (and Persius's) wealth of hapax legomena means their core vocabularies are visibly smaller.

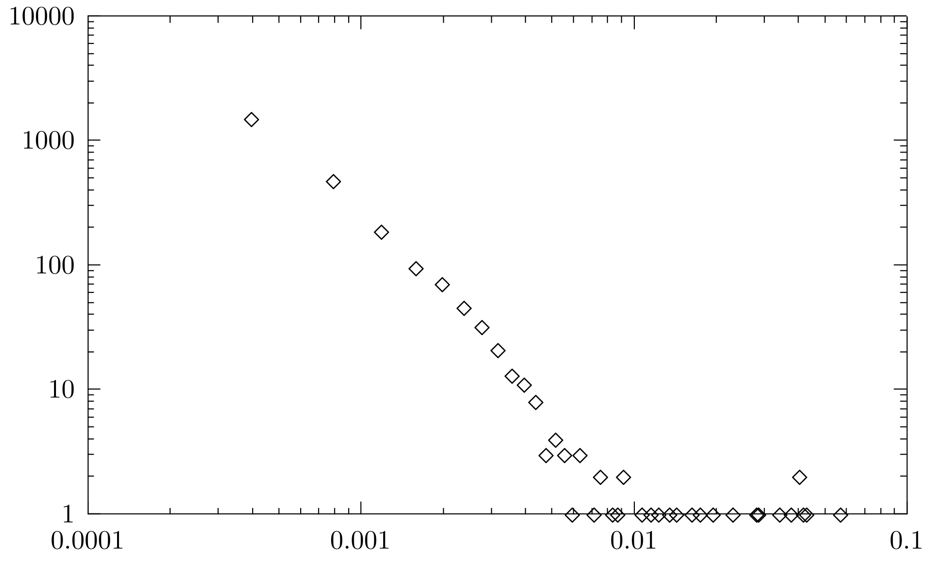

Figure 3: Number of lemmata versus probability of each lemma, `pers`.

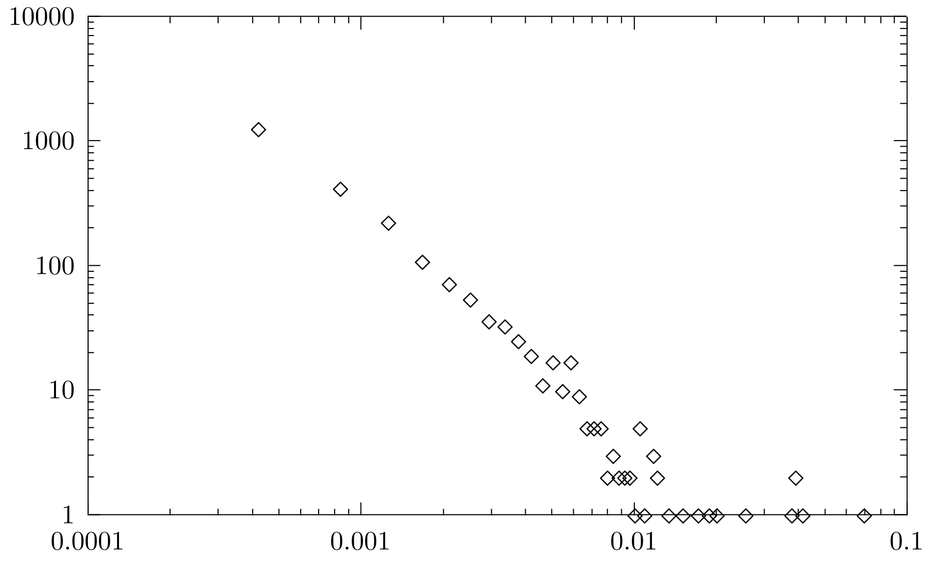

Figure 4: Number of lemmata versus probability of each lemma, `vae1`.

The metrical analyses use the code of Poeta ex Machina. Poeta ex Machina is a text-to-speech engine for Latin poetry, creating MP3s given a passage and its meter. The scansion engine is generic, and is designed to work on all meters. The whole code base is in development; overall, scansion worked on two thirds of all lines.

Vergil's hendecasyllables are strongly against final monosyllables, only 3.1%. Catullus's are not so strict, and neither are Horace's, but Juvenal's and Persius's seem to have adhered to that restriction. Juvenal, also, is more consistent than Horace, whose final monosyllables decreased in frequency from Sermones 1 to 2.

The use of weak caesura is very interesting. Strong caesura gives a decisive pause in a line, and oppositely, weak caesura causes the line to run quickly together. Comparatively, the Aeneid has longer words and a higher coincidence of ictus and accent, so it makes sense that Vergil would use weak caesura to not slow down the lines. And Persius has shorter words and a much lower coindcidence of ictus and accent in the middle of the line, so he uses weak caesura just above a third the frequency that Vergil does. Juvenal's lines are no denser than Vergil's, but his accents coincide less with his ictus, so he could be employing strong caesura to emphasize the halfway point in the line. Horace is less consistent than Juvenal. When his accent and ictus for syllable 2 coincide more, they coincide less in syllable 3 (three in a row would be too much), and thus he uses less weak caesura. Percentage of weak caesura is strongly correlated to coincidence of ictus and accent in syllable three, and this is intuitive: if there is a caesura, it will probably be between the two short syllables, because the accented long vowel would be followed by one short vowel. (Monosyllables were uncomfortable to say, and forms evolved to avoid them: cf *on va* from *unus vadit*, instead of *quis it.*)

The beauty of having a large collection of Satires from Juvenal and Horace is the ability to see a progression of style. As time progressed, both men used longer words, both used more words twice or more, and both had a greater coincidence of ictus and accent at the beginning and ending of the line. Horace dropped his use of final monosyllable from four times Vergil's average to three times, and dropped his use of weak caesura by two thirds, whereas Juvenal's use of weak caesura rose slightly.

Another distinctive characteristic of the poetry is the metrical patterns of the words. Reading and talking were inextricably connected in antiquity, and everyone talks with slightly different accentuation and cadences; all four authors each have slightly different word-patterns. Of course there will be some similarity: the probability of a cretic in hexameter, —◡—, is zero.

Most words have longer vowels in the beginning than at the end: in the top twelve word-patterns, only one (◡◡—) is popular. Horace has ◡—◡ might be so uncommon is probably because it must get the accent on its ictus, and it must follow a word with the accent on its ictus (if the word is polysyllabic), and any word under three morae following it would get the verse accent on its ictus. But within the strictures of the hexameter, there is significant flexibility.

A single short monosyllable long by position ($\overset{\text{P}}{—}$) is most common in them all, and second is a long monosyllable (—); *et* and *atque* and *qui* and *hic* and *est*/*sunt* are all quite popular words. Horace's third favorite is ⏑⏑, almost twice as common as in Juvenal. Juvenal's third favorite is $\overset{\text{P}}{—}$⏑, as is Persius's; this is Horace's fourth. And they all employ $\overset{\text{P}}{—}$— as the fourth or fifth most frequent word-pattern. Horace is slightly more consistent than Juvenal, but the differences are close to the scansion engine's margin of error.

These results are summed up in the following tables.

	`vae1`	`juva`	`juvb`	`pers`	`horacea`	`horaceb`
lines	756	663	704	664	644	643
words	4873	4310	4525	4514	4355	4299
words per line	6.45	6.50	6.42	6.80	6.76	6.68
unlemmatizable	35	95	81	101	77	71
total lemmata	2369	2492	2594	2508	2167	2316
non-hapax	45.1%	33.4%	35.5%	35.8%	37.2%	38.8%
non-dis	27.5%	16.0%	16.9%	16.8%	20.7%	19.5%
non-tris	18.0%	9.35%	9.33%	9.37%	12.6%	11.7%
scannable lines	76%	63%	66%	68%	70%	71%
final monosyll	3.1%	7.9%	7.1%	7.1%	16.6%	11.5%
strong caesura	84.3%	89.6%	87.7%	93.8%	84.5%	90.2%
weak caesura	15.7%	10.4%	12.3%	6.2%	15.5%	9.8%
$\acute{—}$ syllable 1	38.3%	37.8%	37.2%	40.0%	35.1%	40.5%
2	20.7%	12.0%	12.7%	9.3%	12.4%	15.7%
3	16.4%	8.0%	10.3%	7.7%	11.9%	8.5%
4	25.4%	33.7%	44.1%	44.2%	33.6%	34.4%
5	95.5%	88.0%	91.4%	88.5%	79.7%	79.7%
6	95.3%	92.0%	91.0%	93.2%	85.4%	89.1%

	vae1		juva		juvb		pers		horacea		horaceb
$\overset{P}{—}$	0.124460	$\overset{P}{—}$	0.137533	$\overset{P}{—}$	0.152996	$\overset{P}{—}$	0.165911	$\overset{P}{—}$	0.131169	$\overset{P}{—}$	0.132616
⌣$\overset{P}{—}$—	0.075594	—	0.081625	—	0.069635	—	0.092029	—	0.099371	—	0.097426
⌣—	0.069924	$\overset{P}{—}$⌣	0.065598	$\overset{P}{—}$⌣	0.061600	$\overset{P}{—}$⌣	0.068697	⌣⌣	0.073534	⌣⌣	0.067449
⌣$\overset{P}{—}$⌣	0.065065	$\overset{P}{—}$—	0.063735	$\overset{P}{—}$—	0.060596	⌣⌣	0.064809	$\overset{P}{—}$⌣	0.065585	$\overset{P}{—}$⌣	0.066471
⌣⌣⌣	0.050756	⌣⌣	0.046962	$\overset{P}{—}$⌣⌣	0.050218	$\overset{P}{—}$—	0.060920	$\overset{P}{—}$—	0.055979	$\overset{P}{—}$—	0.052460
⌣——	0.044276	⌣⌣$\overset{P}{—}$	0.046590	——	0.045531	$\overset{P}{—}$⌣⌣	0.045690	——	0.038755	—⌣	0.044640
⌣$\overset{P}{—}$⌣⌣	0.043197	⌣⌣—	0.039508	—⌣	0.040844	⌣⌣—	0.041154	⌣	0.038092	$\overset{P}{—}\overset{P}{—}$	0.038775
⌣—⌣	0.038337	$\overset{P}{—}$⌣⌣	0.038763	⌣⌣—	0.039170	——	0.039857	$\overset{P}{—}$⌣⌣	0.036436	⌣⌣$\overset{P}{—}$	0.038449
⌣⌣⌣—	0.037797	—⌣	0.037644	⌣⌣	0.035152	⌣⌣$\overset{P}{—}$	0.033377	—⌣	0.034780	——	0.038123
⌣—⌣⌣	0.031587	$\overset{P}{—}\overset{P}{—}$	0.034290	—$\overset{P}{—}$	0.031470	—⌣	0.032728	$\overset{P}{—}\overset{P}{—}$	0.034117	⌣⌣—	0.034539
⌣$\overset{P}{—}\overset{P}{—}$	0.031587	——	0.030563	$\overset{P}{—}\overset{P}{—}$	0.031135	—⌣⌣	0.031756	—$\overset{P}{—}$	0.033786	⌣	0.032910
⌣⌣⌣$\overset{P}{—}$	0.029158	⌣	0.027954	⌣⌣$\overset{P}{—}$	0.030800	⌣	0.030784	⌣⌣$\overset{P}{—}$	0.031799	$\overset{P}{—}$⌣⌣	0.032258
⌣—$\overset{P}{—}$	0.028618	⌣$\overset{P}{—}$	0.027954	—⌣⌣	0.028791	$\overset{P}{—}\overset{P}{—}$	0.029812	⌣⌣—	0.025174	—$\overset{P}{—}$	0.029977
⌣⌣$\overset{P}{—}$	0.026998	—$\overset{P}{—}$	0.026463	⌣	0.027787	—$\overset{P}{—}$	0.020091	⌣—⌣	0.023186	⌣$\overset{P}{—}$	0.026067
⌣⌣—	0.025648	—⌣⌣	0.025717	⌣$\overset{P}{—}$	0.024439	⌣$\overset{P}{—}$	0.019119	⌣$\overset{P}{—}$	0.021862	—⌣⌣	0.025090
⌣⌣	0.023488	⌣—⌣	0.021245	⌣—⌣	0.018748	⌣—⌣	0.016526	—⌣⌣	0.018218	⌣—⌣	0.022809
⌣⌣—⌣	0.019708	⌣——	0.016772	$\overset{P}{—}$—⌣	0.014396	⌣$\overset{P}{—}$⌣	0.013610	$\overset{P}{—}$—$\overset{P}{—}$	0.016562	$\overset{P}{—}$—⌣	0.013034
⌣⌣——	0.015929	$\overset{P}{—}$—⌣	0.014536	⌣——	0.013726	⌣——	0.012638	⌣$\overset{P}{—}$⌣	0.015568	⌣⌣—⌣	0.012382
⌣$\overset{P}{—}$—⌣	0.014849	⌣$\overset{P}{—}$⌣	0.012672	⌣—	0.012722	⌣$\overset{P}{—}$—	0.012638	⌣—	0.013912	⌣$\overset{P}{—}$⌣	0.012382
⌣——⌣	0.012959	——⌣	0.012300	$\overset{P}{—}$⌣⌣—	0.011717	⌣—	0.010045	$\overset{P}{—}$—⌣	0.012256	⌣—	0.012056

Methodology

Much of the programming to generate these results was done in Ruby. All of the graphs were produced in gnuplot. Poeta ex Machina was originally programmed in AWK, with a new Ruby scansion engine as of January 2006. All of the texts came from the Latin Library.

The built-in Unix command *wc* performs a *w*ord *c*ount, a line count and a letter count of a file. This showed a small difference between Vergil at the low end of words/line and Persius at the high end, so the next step was to see how the number of words per line was distributed. This is easy with a little Ruby code:

```
File.readlines('juv').map {|line|
  line.scan(%r/[[:alpha:]]+/).size
}.join("\n").writeout("juvdensity")
```

This reads roughly as follows: From the *file*system, *read* the *lines* of the file *juv* into an list, and *map* each element of the list like follows: call it *line*, take the *line*, *scan* it looking for each match to the *r*egular expression to match an *alpha*betic character *one or more times*, which produces a new list of each word, and then take the *size* of it, replacing the line that we read with this numerical size. Looking at this newly mapped array, *join* each element together, separated by a *n*ewline character, and *write* it *out* to a file called *juvdensity*.

To visualize this, gnuplot is convenient: it can generate a plot based on data in a file. If it just reads a single column of numbers, it reads that as the vertical coordinates, and the horizontal coordinates are automatically provided integers, starting from 0.

The histograms are all based on lemmatizations of the texts. The dictionary is Lewis and Short, from the Perseus 4.0 Hopper XML interface. The function to read from the web and parse XML data is not very interesting, but let us assume there exists a function `morph` to return a list of all possible morphologies of a word. The code to find the number of unlemmatizable words goes like this:

```
%w{vae1 juva juvb pers horacea horaceb juv hor}.each {|file|
  puts File.read(file).scan(/[[:alpha:]]+/).select {|word| [] == morph(word)}.size
}
```

Make an array out of the *w*ords between braces [assuming those are the filenames], and for *each file*, go to the *File*system to *read* that *file* into one long string, *scan* it looking for each word as before [the percent-r is optional], and from the list that scanning returns, *select* each *word* such that the empty list is equal to the list of all possible morphologies of that word, and get the *size* of that reduced array, and *put* that value onto the *s*creen.

Calculating the lemma-frequency histograms themselves also uses morph.

```
class String
  def writeout(f) File.open(f,'w',&:write[self]) end
end
class Array
  def hashcount() inject(Hash.new(0)) {|h,e| h[e]+=1;h} end
end
texts = %w{vae1 juva juvb pers horacea horaceb juv hor}
texts.each {|f|
  h = File.read(f).scan(/[[:alpha:]]+/).map {|w| pmm(w,:lemma)}.flatten.hashcount
  h.map {|wd,freq| freq*1.0/h.size}.hashcount.map(&:join[" "]).join("\n").
    writeout(f+".histo")
}
```

Teach a string how to write itself out to a file of a given name. Teach a list how to count its elements.[1] Store an array of the filenames of the

[1]Each list has a procedure called inject, which takes a starting accumulator value and a block of code between braces. The first iteration of inject passes the block the

texts. For each filename [as in the prior example], read the file, make a list of each word, map each word into the lemma entry of the word, take this list of lists of lemma entries and *flatten* it to make one long list, count the words in this list and return a list of pairs of each word and its count; call this *h*. Normalize the counts by dividing each by the size of h, and count these normalized counts, and map this list of normalized counts and frequency thereof into strings joined by a space, join the whole list into one long string separated by *n*ewline characters, and write it out to a file with a filename of the old filename concatenated with the string '.histo'.

The metrical analyses come from Poeta ex Machina. To generate the underlining and the circumflex accents, Poeta ex Machina saves the input text in the middle of processing, just after it has been scanned and the pitch accents have been inserted, and pipes this intermediate data into an HTML generator to visualize the scansion. For these purposes, we can use the intermediate data as is, and strip everything besides the vowel lengths and word separations. Assume that for all six files, we can create ".meter" and ".accentmeter" files. The meter files just have newline characters, spaces, and numbers, 0 for short, 6 for long by position, 8 for long by nature. To find how many lines have a final monosyllable requires code like this

```
texts.each {|f|
  puts File.readlines(f+'.meter').grep(/ \d *$/).size
}
```

For each of our files, read its '.meter', and do a linewise *g*lobal *r*egular *e*xpression search for the regexp between the slashes, which returns an array of lines that matched; get the size of this and print it.

starting value and the first value in the list, and the block returns the new accumulator value. The next iteration, the block is passed in this new accumulator value and the second element in the list, and so on, until the last value that the block returns is the value that inject returns.

The regular expression matches a space, followed by a digit, followed a space repeated 0 or more times (the asterisk signifies 0 or more times), all of this at the end of a line which is signalled by a dollar sign. Regular expression are notoriously terse, but Poeta ex Machina is based on regular expression matching, and it works well for analyzing Latin poetry. But wait – the scansion engine got over a third more lines of Aeneid Book 1 than it did for Juvenal's first three satires; the numbers would be biased. Ideally, we'd like to have a percentage of monosyllabic lines. So change the middle line to become these two:

```
m = File.readlines(f+'.meter')
puts m.grep(/ \d *$/).size * 100.0 / m.size
```

Let m be the list of lines in the meter, and write to screen the number of grepped lines of the meter times 100 divided by the entire number of lines in the meter. Change the second line to this to find percentages of weak caesurae:

```
 puts m.grep(/^( *[68] *([68]|0 *0)){2} *[68] /)*100.0/m.size
```

The same as before, just with a different regular expression. What grep is looking through is spaces, zeros, sixes, and eights. Caret is the beginning-of-line signal. Brackets around characters signify a character class, which matches any character within the class. A number inside braces after a parenthesized expression is tantamount to repeating the contents of the parenthesized expression that many number of times. So this regexp matches the beginning of the line followed by 2 of this pattern: an arbitrary amount of space, either type of long vowel, an arbitrary amount of space, and either another long vowel or a short vowel followed by some more arbitrary space and another short vowel. (That huge parenthesized pattern is one foot of dactylic hexameter, in Poeta ex Machina's eyes.) So after two feet at the beginning of the line, some more arbitrary space, and then a long vowel to start of syllable 3, followed by a space for a strong caesura.

The '.accentmeter' files are of a slightly different format: no spaces, this time, just a pair of 'LNH' and '068' for each vowel in the line, to mark whether it has a low, neutral, or high accent, and whether it is short$\smile$, long by position$\overset{\text{P}}{—}$, or long by nature—. Initially, we want to see whether the first foot has coincidence of ictus and accent, which is fairly simple:

```
texts.each {|f|
  m = File.readlines(f+'.accentmeter')
  s = m.size / 100.0
  puts m.grep(/^H/).size / s
}
```

The first syllable comes at the beginning of the line, just a simple H. (And for brevity's sake, because the third line will go through 3 updates, we save the size in a local variable.) The second syllable requires writing a variation of the dactylic hexameter expression:

```
puts m.grep(/^[LNH][68](([LNH]0){2}|[LNH][68])H/).size * s
```

Start of the line, any old accent, long vowel, two of any old accent + short vowel or any old accent + long vowel, and a high accent. To do this for the foot 3, the regexp would be

```
/^([LNH][68](([LNH]0){2}|[LNH][68])){2}H/
```

To save the trouble of typing it over and over with a different number every time, we can splice in the number that we want into the regexp's braces like so:

```
texts.each {|f|
  m = File.readlines(f+'.accentmeter')
  s = m.size / 100.0
  6.times {|i|
    puts m.grep(/^([LNH][68](([LNH]0){2}|[LNH][68])){#{i}}H/).size / s
  }
  puts ""
}
```

The *6.times* calls the block of code 6 times, passing in 0 to 5 as the parameter *i*. Each time, in the regular expression, the sharp-braces insert the value of the expression within the braces, so the value of i the first time is 0, which will be tantamount to matching the beginning of the line, none of the huge expression, and an H, as before. The second time, the value of i is 1, and so on. (And separate each sextuple with a blank line.)

The most involved procedure was getting the distribution for the word-shapes. The vocabulary histograms just plotted the frequencies of each lemma, without recording the lemmata themselves, and this was fine for dealing with what was on the order of thousands of words. The word shape histograms will be dealing with data an order of magnitude smaller, and whereas the histograms' region of interest was at the long tail, the region of interest of word shapes is in the top dozen or so, another order of magnitude smaller. First, to generate the complete histogram of word shapes of each text:

```
texts.each {|f|
  m=File.read(f+'.meter').tr("068","spl").scan(/[[:alpha:]]+/)
  m.histo_hash.sort_by {|shape,freq| -freq}.map {|ws,freq|
    "%s %.6f" % [ws, freq*1.0/m.size, ws]
  }.join("\n").writeout(f+'.shape')
}
```

For each filename, call it f, let m be the Filesystem's read of the file denoted by the string f followed by '.meter', 068 translated into spl respectively, scanned into a list of consecutive word characters. Take m, make a histogram hash of it (same procedure as a few pages ago) and sort each pair of shape and frequency by the negative of the frequency (for decreasing order), and map these sorted pairs of shape and frequency into a meaningful string, and join each string together and write it out.

Then to align the early Juvenal and the later Juvenal:

```
def rd(f) File.readlines(f+'.shp').map(&:strip) end
puts rd('juva').zip(rd('juvb'))[0,20].map(&:join["\t"])

p 0.137533      p 0.152996
l 0.081625      l 0.069635
ps 0.065598     ps 0.061600
pl 0.063735     pl 0.060596
ss 0.046962     pss 0.050218
ssp 0.046590    ll 0.045531
ssl 0.039508    ls 0.040844
pss 0.038763    ssl 0.039170
ls 0.037644     ss 0.035152
pp 0.034290     lp 0.031470
ll 0.030563     pp 0.031135
s 0.027954      ssp 0.030800
sp 0.027954     lss 0.028791
lp 0.026463     s 0.027787
lss 0.025717    sp 0.024439
sls 0.021245    sls 0.018748
sll 0.016772    pls 0.014396
pls 0.014536    sll 0.013726
sps 0.012672    sl 0.012722
lls 0.012300    pssl 0.011717
```

Bibliography

Peter Green. Juvenal: The Sixteen Satires. Penguin Books: London, 1974.

Perseus Digital Library 4.0. At `www.perseus.tufts.edu/hopper`.

The Latin Library. At `thelatinlibrary.com`.

Lee Butterman. Poeta ex Machina: A Text-to-Speech System for Latin Poetry. Undergraduate Thesis, Brown University, 2005.
An online interface is available at `poetaexmachina.net`.