Synthesis of Reading Editions for Latin
Lee Butterman May 2006

Reading Latin at the collegiate level has stayed the same for 100 years: text+dictionary+critical notes=translation. Dictionaries are inevitable for texts from an age of nuance and *variatio*—even Homer's formulaic vocabulary is one-third hapax. Repetitive dictionary lookups remove much of the joy of decoding a language. Alternately, there are picture books full of notes on vocab, grammar, syntax, culture, and such. These remove much of the effort of decoding the language. This research explores how an edition of Latin text can be optimized for fast comprehension, but still stay academically honest. The editions improve comprehension speed, and invite interdisciplinary analogies.

History

SRE's design comes from the best features of many different editions.

When I was in 9th grade, we started to read Caesar, from the textbook, to xeroxes of a critical edition for half a month, back to the textbook. The critical edition was my first introduction to authentic Roman texts and writing, difficult initially. When we moved back to the book, it was faster going. I compared the book's Caesar with the xeroxed Caesar, to see how far off it was, and it was identical. The textbook was undeniably easier, even with its few notes.

In 11th grade, we used Pharr's Aeneid. The on-page vocab and notes were very helpful, though re-inking the same plates for decades made the tiny type of some notes difficult to read, and the vocabulary at the back was partitioned by popularity, so a word more frequent than expected could take two lookups. And many other editions had their benefits and drawbacks: Garrison's Catullus had copious whitespace for marking notes, Cambridge's font was easy to read in any light, etc.

This reading edition synthesis has been for my own benefit thus far, so it is primarily designed to accomodate my own weaknesses: short attention span, low vocabulary retention rate, poor vision. A dictionary is fantastically distracting—a list of a language's words is an index of the thoughts expressed in the culture, each a garden path of diversion. English (and Latin) vocabulary is learned best through reading good books, but taught through sitting down with lists of words.[1] The texts have been designed to be very legible, using a thicker font than the default, and arranging notes for easiest visual searching.

The following is a sample input to SRE, with its output.

[1] An SAT coach came to my high school, and told of a girl who had recently come in from Russia, having read Dickens in Russian, who then read all of Dickens in English, for several months, and aced the Verbal portion of the SAT. He said that her case was unusual, and advised memorizing word tables. Cf. steroids.

Sicelides Musae, paulo maiora canamus.
non omnis arbusta iuvant humilesque myricae;
si canimus silvas, silvae sint consule dignae.@Improbable.@
Ultima Cumaei venit iam carminis aetas;
magnus ab integro saeclorum nascitur ordo.
iam redit et Virgo, redeunt Saturnia regna,
iam nova progenies caelo demittitur alto.
tu modo nascenti puero, quo ferrea primum
desinet ac toto surget gens aurea mundo,
casta fave Lucina; tuus iam regnat Apollo.
Teque adeo decus hoc aevi, te consule, inibit,
Pollio,@A blatant politicization of the pastoral.@ et incipient magni procedere menses;
te duce, si qua manent sceleris vestigia nostri,
inrita perpetua solvent formidine terras.

Sicelides Musae, paulo maiora canamus.
non omnis arbusta iuvant humilesque myricae;
si canimus silvas, silvae sint consule dignae.[1]
Ultima Cumaei venit iam carminis aetas;
magnus ab integro saeclorum nascitur ordo.
iam redit et Virgo, redeunt Saturnia regna,
iam nova progenies caelo demittitur alto.
tu modo nascenti puero, quo ferrea primum
desinet ac toto surget gens aurea mundo,
casta fave Lucina; tuus iam regnat Apollo.
Teque adeo decus hoc aevi, te consule, inibit,
Pollio,[2] et incipient magni procedere menses;
te duce, si qua manent sceleris vestigia nostri,
inrita perpetua solvent formidine terras.

---

aevum neverending time, eternity
Apollo Apollo
arbustum a place where trees are planted, plantation, vineyard planted with trees
arbustus set with trees
aureus of gold, golden
caelo to engrave in relief, make raised work, carve, engrave
carmen a song, poem, verse, oracular response, prophecy, form of incantation, tune, air, lay, strain, note, sound
castus morally pure, unpolluted, spotless, guiltless, virtuous
decus grace, glory, honor, dignity, splendor, beauty
demitto to send down, let down, drop, lower, put down, let fall, sink
desino to leave off, give over, cease, desist, forbear
faveo to be favorable, be well disposed, be inclined towards, favor, promote, befriend, countenance, protect
favus a honey-comb
ferreus made of iron, iron
formido fearfulness, fear, terror, dread, awe
humilis low, lowly, small, slight
ineo to go into, enter
integer untouched
integro to make whole
Lucina she that brings to light, goddess of childbirth
mano to flow, run, trickle, drop, drip
mundus clean, cleanly, nice, neat, elegant
mundus toilet ornament, decoration, dress
Musa a muse, one of the nine Muses
myrica the tamarisk
paulum a little, somewhat
paulus little, small
perpetuus continuous, unbroken, uninterrupted, constant, entire, whole, perpetual
perpetuo to cause to continue, perpetuate
procedo to go before, go forward, advance, proceed, march on, move forward, go forth
progenies descent, lineage, race, family
saeculum a race, generation, age, the people of any time
Saturnius of Saturn, Saturnian
Sicelis Sicilian
ultimum finally, for the last time
vestigium the bottom of the foot, sole

[1] Improbable.
[2] A blatant politicization of the pastoral.

Characteristics of SRE

SRE is obivously ignorant of the meanings of words. *Apollo* is defined as Apollo. And *arbusta* is lemmatized to *arbustum*, the correct form, as well as *arbustus*, a derived adjectival form that is irrelevant in context. But its thoroughness means that it includes *faveo* for *fave*, as well as *favus*, a bee association, relevant in Georgic 4 and Aeneid 1. A print dictionary would not facilitate finding such an allusion, when looking up a word that scansion and context predict is an imperative verb. One might miss the sound-allusion. Indeed, this is an important first step to enumerating wordplay, especially in poetry, where the meaning is larger than the sum of the words. Note that it is not necessarily beneficial to include everything a computer can determine about a word: good design is as much about exclusion as inclusion, especially for a static fixed-size medium like a printed page; also, as a computer analyses forms in greater detail, this approaches a Loeb, and Loebs are not designed for a reader to focus on Latin text.

The reading editions, above all, are wonderfully usable.

I was the sole usability tester; still, I read 45-60 lines an hour with SRE's texts. I was reading 15-20 lph with paper books, 25-30 lph with Perseus online. I achieved a three-fold increase in reading speed from any other physical book, with perceptibly greater satisfaction after reading a passage, and a much better ability to sightread.

Perseus' morphological analysis cannot resolve all words, and I did use my dictionary. But I was mostly able to use the reading editions as is, jotting down notes, completely focused on the page. Part of it comes from reading at a higher level, and part of it comes from preservation of flow.

Interdisciplinarities

Alan Perlis, a famous computer scientist, observed:

> *A programming language is low-level*
> *when its programs require attention*
> *to the irrelevant.*

Similarly, a reading edition is low-level when its readers require attention to the irrelevant. Translation is looking at a page of Latin text and generating a mental parse tree in colloquial English of the meaning. Thumbing through many different books is irrelevant. Computers are much better at extracting data from an ordered table. If a computer creates a text with useful vocab marked, a reader who knows the grammar can focus on how the words fit together.

The effect of focus while reading and understanding Latin is similar to the effect of focus while reading and understanding computer code. Reading one line of code means understanding the lines of code preceding it, the lines of code following it, the variables it accesses, the data

structures that it works on, the functions that it calls, the library routines it uses, the peculiarities of its language's syntax, the comments around it — this is all relevant to the meaning of one line of code.

A well-designed function has one specific task, which each of its lines of code collaboratively achieve. It in turn calls other functions (even perhaps itself) and, understanding how the system works from the ground up, a programmer can grasp the nuances of how the code works, how the parts join together.

This is a difference in degree, not in kind, from translating Latin. The *faveo*/*favus* wordplay, the Theocritean *Sicelides Musae*, the metrical identity of line 1 and line 3, the triple repetition of *iam* just like the triple repetition of the names of the dead: poetry synthesizes beauty with meaning, and grasping the meaning as soon as possible means a reader can focus on enjoying the beauty, ultimately having more fun.

This is similar to what Eastern thought would term 'being at one with a text', and what Western philosopher Mihaly Csikszentmihalyi called 'flow'. Flow is full immersion in a task, with a limited field of attention, a loss of conscious self-awareness, a control over the task and rewards from the task. Turning pages, frequently changing between languages, grasping different physical books, frustration in lack of note or dictionary entry: these break flow. They expand the field of attention by requiring attention to the irrelevant, they make one self-aware of the act of translation, they take control away from the act of translating,

they (besides the rare notes that change one's understanding of a text) are ultimately without reward.

"A grad student who's too lazy to use a dictionary? Back in my day..." is a valid response. But programming has similar parallels. Consider the problem of taking the sum of the numbers in a file. A program has to ask the operating system for a file handle to read in the file, save the file handle, allocate a buffer, read the bytes from the file into the buffer until encountering one of three possible newline sequences into the buffer, make sure that the buffer does not overflow which is the source of 70% of all serious security holes, make sure that the file was read successfully, convert the list of bytes read into a machine number, close the file handle. And in that list, the only thing left out was the only important step: add a value to an accumulator value. The details are irrelevant to the task at hand. Little tricks to optimize on a case-by-case basis (fractional numbers take much longer to process than integers, usually the newline character stays constant in a file) are vastly outweighted by the ease of writing `{ sum += $0 }`. A study from 30 years ago, about a project 10 years earlier, still holds true: humans have a fixed tolerance for complexity, programmers generally code the same number of lines per hour regardless of language, and the higher the level at which one works, the more efficient one can be. Similarly, there is no need to temporarily stop thinking about the parse tree of the sentence being read, visually search for a dictionary, grasp it in hand and open it, flip quickly to the general area of the lemma,

sequentially scan a page for the lemma, follow all **iu-=ju-** entries and such and repeat until finding the lemma, even if the spine is broken to the section around *quantuluscumque*, or even if a rolodex of cards serves as a dictionary.

Programmers also love to have notes about the language as they read and write it. Most programming languages have documentation that can easily be imported to a development environment. But the most authoritative answer to what some code means is to evaluate it, to type it into a Read-Evaluate-Print Loop; the LISP family of languages has been able to do this for a long time, and many newer languages like Python and Ruby have the same capability. Almost all of the languages used in successful software startup companies (which, by necessity, work quickly to deliver the most features in the least time) have REPLs and online documentation—the equivalent of vocabulary notes and commentary on the page.

Coding at a higher level improves flow, limiting the attention field to the specific problem, and REPLs improve flow, almost creating a dialogue between programmer and computer in the specific programming language. Once a language becomes a medium of thought, this greatly reduces self-awareness while programming. And I felt enormously productive reading—my attention was only on the page, eyes darting back and forth, jotting down notes beside the words, hexameters gliding along.

SRE is primarily a framework for annotating text; it automatically marks vocabulary, and anything delimited by ampersands is inserted as a numbered footnote below the lemma-definition notes. This summer, SRE will have input filters to automatically fetch texts from Perseus and thelatinlibrary.com, and, integrated somehow into Perseus, it will extract all cross-references to a particular text and include the most helpful ones. Also, there is the possibility of a Wikipedia-style collaborative generation of a commentary, tying into Perseus' grammar marking experiment ('voting' on the morphological analyses of a word). These ampersand-annotations are not restricted to simple phrases, though; the footnotes pass through SRE untouched, and can include all sorts of accentuation, formatting, and mathematical symbols. (TEX was initially designed as a typesetting system for math papers; this document is typset in TEX.)

Graphic Design

SRE necessitated several design decisions that strayed from TEX defaults. These are decisions that no editor of an edition of Latin poetry would have had to make; they arose from new limitations on fonts, page shapes, and usage.

SRE does not create any printable documents. It creates input to TEX, which in turn creates PDFs. TEX's default font is Computer Modern, which has an air of elegance at first, but after hours of reading, start to

look fragile and spindly. Further, almost all author-prepared documents in computer science use TEX, because the creator used it to typset his Art of Computer Programming, and it is particularly good for such things; one consequence is that the font has gotten very boring to read. SRE caters to my own weaknesses, so I decided against Computer Modern. (For a time, I was thinking of using Computer Modern Sans Serif; the letters are a bit thicker, and individually the shapes are cleaner than with serifs. But serifs guide the eyes from letter to letter, like trolley tracks, and for running text, it's simply awful. Look how much longer it's taking you to speedread through this sentence as your eyes fixate on each word, as opposed to with Vera. And Vera is wider, which helps at smaller sizes. Nevertheless, it is straightforward to change fonts.

Pagination is useless, except maybe if the pages fall on the floor. One possibility is to have no page numbers, and push the bottom margin down to fill out each column. Another option is to use the page footer to mark the line numbers of the text on the page, and remove the numbers from the left margin. This suddenly becomes practical if there are 15 lines of text a page instead of 50. Intuitively, it seems difficult, but interesting to see the results of.

SRE creates reading editions optimized for Flow. To glance away from the text breaks concentration, so the goal is to minimize the time spent finding a lemma. Cassell's dictionary has bold lemmata, with principal parts, and scansions, and a dot indicating the ending of the

root of a word, and genders, and parts of speech, all of which are unnecessary most of the time. Is there any imminent need to know the gender of *myricae*, or the past participle of *demitto*? A dictionary does not consider that: it is for reference, authoritative (my dictionary even calls itself authoritative), non-text-specific, a tour-de-force of all Golden Age vocabulary. SRE's vocab is locally chosen, and non-authoritative (it takes whatever Perseus has got)—it is intentionally throw-away, because the context of a word colors its meaning better than a dictionary can. But the bold lemmata is a good idea. I have mixed feelings about eliminating the footnote-separation dash; significant work, potentially little gain.

Source Code

All of the source code is divided into three parts. The `bigfoot` footnote package is responsible for shuffling footnotes, the TEX code is responsible for establishing the typesetting environment, and the Ruby program transforms the text.

Easiest things first. The footnote package's style file, `bigfoot.sty`, has the code

```
\stepcounter{FN@totalid}
```

which increments the number of the footnote each time one pops up in the text. If you override this counter, you could put any footnote in

running text, give it a number, and it's in that order at the bottom of the page. And each word can be read as a base 26 number (a–z → 0–9a...).[2] The downside is that numbers in TEX have to be less than a billion or so, so we can only use the first six letters to determine sorting order: $26^6 \approx 300\text{M}$. The other option, much more difficult, would be to count the lines of poetry, to count the number of footnotes passing through, doing some funny math with how far separated the baselines are, giving some leeway for lines that are too long, putting the footnotes in all in one shot, starting a new column, and before you know it, it's an re-implementation of TEX and the footnote package, which is not the goal of the exercise.

Therefore, the line becomes

```
\setcounter{FN@totalid}{\value{vocab}}
```

which sets it to whatever value the counter vocab has, and we assume that we'll make a command to set the counter vocab and call a footnote defintion at the same time. (All variables are globally accessible.)

(They are only 16 letters different, but the expressive power is increased tremendously. TEX is a high-level language for writing papers, and a language should promote much more thought than typing.)

Thus the footnote package. The TEX template is a bunch of page definitions and a few functions.

[2] `tr('a-z','0-9a-z').to_i(26)` is valid Ruby code.

```
\documentclass[10pt]{article}
\pdfpageheight=11in\pdfpagewidth=8.5in
\usepackage{bigfoot}
\usepackage[margin=0.5in,top=0.35in,bottom=0.95in]{geometry}
\usepackage{bera}
\DeclareNewFootnote{df}\DeclareNewFootnote{cr}
\newcounter{vocab}\newcounter{notes}\setcounter{notes}{0}
\def\df#1#2{\setcounter{vocab}{#1}\Footnotedf{\kern-1ex}{\kern-3.55ex#2}}
\def\comm{\stepcounter{notes}\setcounter{vocab}{\value{notes}}\footnotecr}
\begin{document}
\obeylines\twocolumn\parindent=0pt
\input sreout
\end{document}
```

The document will be an article, at 10 point body text (and 8 point footnote text). Set the pdf page size to 8.5x11; it defaults to A4, but letter-sized paper is much more common in the US. Use the bigfoot package. Use the page geometry package to shrink the margins to as little as my printer can manage. Use the Vera font (contributed by Bitstream, so to avoid name confusion, it is called Bera in TEX). Declare a new footnote, df. Declare a new footnote, cr. Make a new counter, vocab. We changed bigfoot to rely on a vocab counter, and defined the vocab counter after we included the bigfoot package. No problem—TEX is a macro language. It replaces control sequences (delimited by backslashes) as needed, so the value of the counter for vocab is not evaluated until a footnote is used. Make another new counter, `notes`, and set it to zero. Define a control sequence `df` to take two parameters immediately following each other, and replace it by a command to set a counter, `vocab`, to the value of the first parameter, and then a lower-level command to make a footnote. This lower-level `Footnotedf` takes

two parameters, a symbol to use as the number, and text to insert at the bottom of the page. A kern is a blank space, so the footnote command is to insert a negative space in the running text (to compensate for the footnote's automatic space), and to insert a negative space at the bottom of the page followed by the second parameter of our function. The next function, `comm`, is a function of no variables, and replace it with the following commands: add one to the counter `notes`, set the `vocab` counter to the value of `notes`, and start a high-level footnote for *c*ommenta*r*y. This footnote takes one parameter, so the function `comm` is actually a function of one implicit parameter. (Leave as much as possible implicit; no sense in asking an operating system for a file handle, allocating a buffer, and so on.) Begin the document. Obey the line separations, and make new paragraphs each time a new line is seen.[3] Make the output two-column; a line of hexameter is usually under 4 inches, so the page will be packed more densely, and thus less page turning, and better flow. Because each line of text is a new paragraph, and the convention is to indent a paragraph, set the paragraph indent to 0 points, 0 seventy-seconds of an inch. Input the SRE output file (which always stayed as `sreout.tex`), and end the document.

[3]The default is to make new paragraphs every time two new lines are seen—for old computers with 80x25 character screens (for which this was originally designed), and for papers with long paragraphs (for which this was originally designed), paragraphs of more than one screen line of text are the norm, and TEX is designed to be unintrusive by default, with easily changeable defaults.

```
load 'sym2proc.r'
load 'inetfile.r'
load 'texcode.r'

class String
  def firstcaps(rgx) match(rgx).captures.join end
  def cleanup() tr('_','').gsub(/&.+?;/,' ').gsub(/\n+/,"\n") end
end

$ouija = 2
Lang = "la"
LemDefInHTM = %r%<G><foreign lang="la">(.*?)[#0-9]*?</foreign></G></font></td><td>(.*?)</td><td>%
FreqInHTM = />([0-9.]+)</
TextWord = /[[:alpha:]]+/
SWindow = [nil] * 100
Gensym = memoize {|s| (0..19).map {rand 10}.join}

def defn_to_tex(lemma,meaning,freq10k)
  SWindow.shift;SWindow.push lemma+meaning
  TeX.df(word_to_num(lemma),TeX.textbf(lemma)+' '+meaning)
end

def word_to_num(word) (word.downcase + 'a' * 6).tr('a-z','0-9a-z')[0,6].to_i(26) end

def morphhtms(word)
  HTM["http://www.perseus.tufts.edu/cgi-bin/morphindex?lang=#{Lang}&embed=2&lookup="+word].split('<p>')[1..-1].grep(LemDefInHTM)
end

def avgfreq(frequencies) [0.5 * (frequencies[2].to_f + frequencies[4].to_f)] end

Defs = memoize {|word|
         morphhtms(word).map {|s|
           s.firstcaps(LemDefInHTM) + avgfreq(s.firstcaps(FreqInHTM))
         }.uniq.sort_by(&:slice[0])
       } # [lemma, definition, (max+min/2)freq/10k]

def anydefs(word)
  Defs[word].find_all {|lemma,meaning,freq10k| freq10k<$ouija && !SWindow.include?(lemma+meaning)}.map(&method(:defn_to_tex)).join
end

def maketex(file)
  wolinenums = File.read(file).gsub(/@.+?@/) {|s| Gensym[s]}.gsub(TextWord) {|w| w+anydefs(w)}.cleanup
  wolinenums.zip(1..wolinenums.count("\n").next).
             map{|line,num| (num%10==0 ? TeX.noboundary+TeX.llap(TeX.scriptsize+num.to_s+'~~') : "")+line}.join.
             gsub(/\d{20}/) {|v| TeX.comm Gensym.index(v)[1..-2]}
end

maketex('ecl4start').writeout('sreout.tex')
```

(It's easiest to follow along if you tear that last page out.)

The easiest way to read the code is backwards. We'll start at the last line. That looks like it should make TEX output, from the file `ecl4start`, and write it out to `sreout.tex`, which is exactly what it does. Just above, maketex is defined as taking one parameter, which it calls `file`. `File.read()` reads a file, taking as a parameter a filename, and returns a character string of the entire contents of the file. (`File` is a object, it is a class, and `.read` calls its `read` method. This is object-oriented programming; objects can perform certain functions on themselves.) So now we have a string, with the file contents. Tell the string to globally substitute, every time it sees in itself anything that matches the Regular Expression of an ampersand and any character 1 or more times (as few as possible though) and an ampersand, the result of the expression in braces. The function in braces takes one parameter, `s`, through the 'chute' (as it's lovingly called), and gsub passes to its associated function block the whole text that matched the Regular Expression (regexp). This function returns `Gensym` of `s`.

`Gensym` is a memoized/cached function; memoize means to store a value once computed. `memoize` takes a function, within braces, and returns a lookup table, and every time a value is looked up that does not exist, the function is run with that value, and the return value is stored in the lookup table. When `Gensym` looks up a value, its default value is a string of twenty random numbers from 0 to 9: the range `0..19`,

twenty numbers, each mapped to the value of the associated function, which is a random number less than 10, then joined together into a string. So we replace every ampersand-footnote with twenty random numbers, but Gensym's lookup table (hash table) records all incoming strings, so we can replace it back, after we've finished fiddling with the words. (`memoize` is in `sym2proc.r`[4]).

So `gsub` returns the string, with the substitutions made. Then we gsub that, matching every `TextWord` (call it `w`), to `w` plus any definitions of `w`: `anydefs(w)`. Clean up this second substituted string, and call it `wolinenums`; it is the TEX file without line numbers.

At the top, we put some definitions of functions in the `String` class; `cleanup` translates all underscores (common in Perseus' HTML) to the empty string, globally substitutes any `&ecirc;`-like HTML entities to an empty space, and globally substitutes all instances of one or more newline character to a single newline. (TEX does not create a blank paragraph when it sees a newline, but my *ad hoc* line numberer will treat it as a line, so we must take out all multiple newlines.) Once we define a function in the String class, it's callable just like the built in gsub.

[4] sym2proc.r:

```
class Symbol
 def to_proc(*args)lambda{|*a| a.first.send self,*(args+a[1..-1])}end
 alias [] to_proc
end
def memoize(&b) Hash.new {|h,nu| h[nu] = b[nu]} end
```

Now for `anydefs`. For the definitions of a word, find all (lemma, meaning, frequency per 10k) triples whose frequency per 10K is less than the global variable `$ouija`—currently set at 2 up top—and for whom it is NOT TRUE!, the answer to the question, does the Sliding Window include this specific lemma+meaning?[5] Map each of these definitions with the function method to convert a definition to TEX code, and join them into one string.

The definitions `Defns` is a memoized function in a lookup table, but we're not going to do reverse lookups. The definition of a `word` starts with the morphological analysis html fragments of this word. `morphhtms` fetches a webpage (via `HTM`, defined in `inetfile.r`[6]): the language is interpolated into the string with sharp-brace syntax (`#{Lang}`), because it is the same process for a Latin or Greek morphological analysis page, and reusing the same code to do the same thing is good. Split the page by HTML paragraph tags (each entry in the Perseus table starts

[5] Note Ruby's effective use of punctuation.

[6] `inetfile.r`:

```
require 'digest/md5'
require 'open-uri'
load 'sym2proc.r'

class String
  def writeout(f) File.open(f,'w',&:write[self]);self end #ersatz monad
  def md5() ".ip." + Digest::MD5.hexdigest(self)[0,16] end
end

HTM = memoize {|addr|
File.exist?(addr.md5)?open(addr,&:read).writeout(addr.md5):File.read(addr.md5)
}
```

with one), toss out the first (at position 0, get everything from element 1 to -1, the end), and Globally search for Regular Expressions Preserving/printing them, with the regexp matching lemma definitions in HTML. Ok, so, take these morphhtms, map each, calling it `s`, to the first captures for the lemma and definition in s, plus the average frequency of the first captures for frequency in s. The regular expressions at top parenthesize values to save them in the scan, take all the parenthesized captures, and join them. After mapping, return only the unique definitions, sorted by lemma (given two identical numbers, `bigfoot` will put the first one it sees first).

So. After we make the definitions, after filter them, we pass them to `defn_to_tex`, which takes, a lemma, meaning, and frequency per 10k. `SWindow` is a sliding window; shift an old entry out (we start out with the null value, so it never matches anything, which means our filter lets everything in at first), push in a concatenation of this lemma and meaning. Here we generate TEX code, from `texcode.r`:

```
class TeXCode
  def method_missing(sym,*args)
    "\\"+sym.to_s+args.map {|a| "{#{a}}"}.join
  end
end
TeX = TeXCode.new
```

So remember the object paradigm, you call a method on an object. What happens if an object doesn't have that method? Ruby defines `method_missing` to receive the symbol of the undefined function, and all of the arguments passed in. `TeXCode` is designed to make backslash sequences, with curly-brace-surrounded arguments. So instead of enumerating every possible sequence that SRE needs, just let Ruby's method handling do it, so every function call with arguments will map into a TEX function call. Recalling the code from a while ago, `df` takes two parameters, one the footnote number (which we decided would be the numerical equivalent of the word), and the other the contents of the footnote.

So to make a word into a number, make it lowercase, add 6 letter 'a's (to pad it if it's too short), translate a to z into 0 to 9 and a to (as high as z corresponds), start at position 0 and take 6 characters, and convert it to an integer base 26. The text of the footnote is the lemma in bold face text plus a space plus the meaning.

All functions return the same outputs with the same inputs, except for `defn_to_tex`. In `maketex`, every time we find a `TextWord`, we call `anydefs` on it, which in turn calls `defn_to_tex`, so we are updating the state of the sliding window every time we insert a definition. If the 100-entry sliding window does not already have the definition, it will be processed, otherwise, filtered out. But besides this, all functions return the same values given the same inputs.

So then, after getting the TEX code without line numbers, we insert the line numbers. A string is an enumerable value; Ruby counts a string by the number of newline characters it has, and `zip` merges enumerable values into a list of tuples of these values, much like a zipper, joining a pair of teeth. So we have a range, from 1 to the next higher number after the count of newlines in `wolinenums`. (In a list (a,b,c), there are 3 elements, 2 commas.) Map each pair of zipped values, calling them the line and the number, to this: is the number modulo 10 equal to 0? If so, this expression is the TEX code for starting horizontal mode (after making the last paragraph) plus the code for a left overlap (spilling out legally into the left margin) of a temporary change to a smaller font plus the line number converted to a string plus two explicit blank spaces. If the number mod 10 is not, the expression returns an empty string. Take whatever we got and append the line to it, and replace the zipped pair by that. Join it all into one string. Now, after all is done, we put back in the ampersand-delimited comments. Globally substitute, for all twenty-number runs calling each run `v`, the TEX control sequence for numbered commentary and braces around the index of this run `v` which equals the string we initially passed in because the index is the string and the value at the index is the run `v`, but take the second character to the next-to-last character of this index string, stripping off the ampersands.

And voilà, we are done.

Fun Summer Reading

Bederson, Benjamin. *Interfaces for Staying in the Flow.*
hcil.cs.umd.edu/trs/2003-37/2003-37.html

Flow.
wikipedia.org/wiki/Flow_(psychology)

Garrison, Daniel H. *The Student's Catullus.*

Graham, Paul. *Great Hackers.*
paulgraham.com/gh.html

Mental State called Flow.
http://c2.com/cgi/wiki?MentalStateCalledFlow

Mihalyi Csikszentmihalyi.
wikipedia.org/wiki/Mihaly_Csikszentmihalyi

Perlis, Alan. *Epigrams on Programming.*
www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html

Pharr, Clyde. *Vergil's Aeneid.*

Spolsky, Joel. *Human Task Switches Considered Harmful.*
http://joelonsoftware.com/articles/fog0000000022.html[sic]

Virgil, R Thomas ed. *Eclogues.* Cambridge University Press, 1988.