24.964 Fall 2004 A. Albright
Modeling phonological learning 9 Sept 2004

Class 1 Supplement: Pattern matching, and dealing with files

In class 1, we saw some basic elements of Perl syntax: printing, using scalar and array variables, for loops, and conditionals (if, else, unless). This was enough to do some phonologically-relevant tasks (generated CV syllables, filtering items that meet certain conditions, etc.). There are just a couple other pieces that are needed to do the first week's assignment: pattern matching, and some tools to read and write files.

1 Pattern matching

A tool that is very important for linguistic applications is the ability to search for text that matches abstract patterns. The basic syntax for testing whether a string contains a pattern is:

(1) `if ($sometext =~ m/pattern/) { ... }`

This tells Perl to look for the pattern inside the slashes as a string; that is, it searches inside the $sometext variable for the text "pattern" (i.e., a `p` followed by a `a` followed by ...). The 'm' is optional before the search string, if all you're doing is testing whether the pattern is matched. So, `if (‘‘blah’’ =~ /bl/)` tests if "blah" contains the sequence "bl" (it does). In addition to searching for a literal string, there are also many other ways that you can refine your search by searching for more abstract patterns, involving wildcards or classes or sounds. Some useful ones include:

- `[ab]` means "either a or b" (a, b); this can be expanded, so `[abc]` = either a, b, or c, etc... So, `"blah" =~ /[ab]/` tests whether *blah* contains either an *a* or a *b* (it does—twice, in fact)
- `[^a]` means "anything other than a"; `[^ab]` means "anything other than an a or a b", etc. (set negation). So, `"blah" =~ /[^ab]/` tests whether *blah* contains something that is neither an *a* nor a *b* (this is also true twice)
- `a*` means "any number of a's (from 0 to infinity)" (nothing, a, aa, aaa, aaaa, aaaaa, ...). So, `"blah" =~ /a*/` tests for 0 or more a's (true). `"blah" =~ /x*/` tests for 0 or more x's (also true).
- `a+` means "one or more a's" (a, aa, aaa, aaaa, aaaaa, ...) `"blah" =~ /a+/` tests for 1 or more a's (true). `"blah" =~ /x+/` is *not* true.
- `ab*` means "*a*, followed by any number of *b*'s" (a, ab, abb, abbb, abbbb, ...). `"blah" =~ /ab*/` is true.
- `ab+` means "an a, followed by one or more b's" (ab, abb, abbb, abbbb, ...). `"blah" =~ /ab+/` is false.
- `(ab)*` means "any number of consecutive occurrences of ab" (ϵ, ab, abab, ababab, abababab, ...) `"blah" =~ /(ab)*/` is vacuously true.
- `(ab)+` means "one or more consecutive occurrences of ab" (ab, abab, ababab, abababab, ...). `"blah" =~ /(ab)+/` is false. (It has an *a* and a *b*, but not together in that order.)
- `a?` means "an optional a". `"blah" =~ /a?b/` is true, because it has a *b* (in this case, with no *a* before it. `"blah" =~ /bl?a/` and `"bah" =~ /bl?a/` are both true.
- `^a` means "an *a* at the beginning of the string". `"blah" =~ /^b/` is true, but `"blah" =~ /^b/` is not. Be careful to distinguish this from set negation, when the ^ is inside the brackets. `"blah" =~ /^[^a]/` is true, but `"blah" =~ /^[^b]/` is false (be sure you understand why!)
- `a$` means "an *a* at the end of the string"
- `.` (period) means "any character". So, `"blah" =~ /b.a/` is true.
- `.` can be combined with `*` and `+`. `"blah" =~ /b.*h/` is true, as is `"blah" =~ /b.+h/`. `"blah" =~ /b.*lah/` is also true, but `"blah" =~ /b.+lah/` is not.

There are also a few built-in "shorthand" notations for useful classes of characters that you might often want to refer to. Some of these are listed in (18) on the class 1 handout. One of the most useful ones is the `\s` notation, which is matched by any "white space": a space, a tab, or a carriage return. (You don't need to memorize these, but remember that they exist.) We will cover other aspects of pattern matching later; for now, this should be enough to understand the scripts `cvcv3.pl` and `cvcv4.pl` from the class 1 handout (found in the perlscripts directory). More information about pattern matching can be found at:

- http://www.wdvl.com/Authoring/Languages/Perl/PerlfortheWeb/perlintro2_table1.html
- http://etext.lib.virginia.edu/helpsheets/regex.html

In addition to searching for text, it is also possible to do search and replace. The syntax for doing this is: `sometext =~ s/find/replace/`. There are two useful variants of this:

- `blah =~ s/x/y/` searches `blah` for `x` and replaces with `y` (1st instance only)
- `blah =~ s/x/y/g` searches `blah` for `x` and replaces with `y` (all instances)

We'll see these in operation once we learn a little bit about files.

2 Dealing with files

One thing that makes Perl convenient is the ease of opening and creating files. There are two basic ways to open a file: for reading (an input file, that you don't want to change), and for writing (an output file, that you are creating). The syntax for these is nearly the same:

- Reading: `open (HANDLE, "filename")`
- Writing: `open (HANDLE, ">filename")`

The "handle" is a label that you give the file, and it is how you will refer to the file throughout the program. (Once you tell the program that the file you want is actually called `filename` on the disk, you'll never use that name again.) This is probably easier to understand with an example:

(2) readfile1.pl

```
#Read a file, print its line to the screen.
$input_file = "sample.txt";
open (INFILE, $input_file) or die "The file $input_file could not be found\n";

# Loop, continuing as long as lines can be read from the file
while ($line = <INFILE>)
{
  $line_count++;
  print "$line_count  $line";
}

close INFILE;
```

This program reads in a file called `sample.txt`, which is assumed to exist already and reside in the same directory as `readfile1.pl`. In this case, I have created a file `sample.txt` which contains a short passage of pseudo-Latin that is often used as dummy text by typesetters. (It starts: `Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod.`) If the program can't find this file (or can't open it for some reason), we need to know this. The `or die` clause serves this function; if the `open` command is unsuccessful (technically, if returns a value of false when we try to invoke it), then the `or die` part instructs the program to print a message warning us that the file couldn't be found. (The program then quits, or dies, without trying to do anything else.)

Now that the file is open for reading, how do we actually read it? This is accomplished by the command `<HANDLE>`, which reads a single line from the file called HANDLE. Usually, we want to do something to the text that we read, so it's handy to assign it to a variable: `$line = <INFILE>` gets the next line and stores it in the `$line` variable. When the file runs out of lines (i.e., once we've read the whole thing), the `<HANDLE>` command returns a value of "false".

The actual reading part is embedded in a `while` loop. The command `while` is like a shorthand for `for`. It means "keep looping through a block of code until something isn't true any more". When combined with the `<HANDLE>` command, this has the effect of continuing to read lines from the file until there are no more lines (until `<HANDLE>` is false). Each time we enter the `while` loop, a new line is read, and stored in the `$line` variable. Within the loop, we can do whatever we want to the line, before moving on to the next line. In the `readfile1.pl` program, all that we do is count the lines, and print the current line number and line to the screen. (That is, we do not actually manipulate the lines in any way.)

The program `readfile2.pl` shows how to write to a file, rather than printing the results to the screen. The key is opening a second file (here, given the handle OUTFILE), using the `open (HANDLE, ">name")` command.

*****WARNING: if there is already a file with the same name on the disk, this command will replace it, destroying the previous contents. You should be careful when creating new files with perl*****

Once the file is open for writing, we can print things to it with the `printf` command: `printf HANDLE "some text"`

(3) `readfile2.pl`

```
#Read a file, print its line to the screen.
$input_file = "sample.txt";
$output_file = "sample-output.txt";

open (INFILE, $input_file) or die "The file $input_file couldn't be found\n";
open (OUTFILE, ">$output_file") or die "The file $output_file couldn't be written\n";

# Loop, continuing as long as a line can be read successfully from the file
while ($line = <INFILE>)
{
  $line_count++;
  printf OUTFILE "$line_count  $line";
}

close INFILE;
close OUTFILE;
```

The file `readfile3.pl` (on the class handout) combines reading a file with pattern matching. Try to read it and guess what it should do, before you read further.

As with the previous program, this program reads in the file `sample.txt`, and opens an output file for writing (`sample-output.txt`). It reads in lines from the input file, and for each line, tries to count some things. Each time a line is read, the variable `$count` is set to zero, and the variable `$lines` has 1 added to it (keeping a running total of the number of lines seen so far). We attempt to do something to the `$count` variable in another `while` loop: `while ($line =~ m/[aeiou]/)`. From the previous section, we know that `[aeiou]` refers to a set of characters (vowels), and `=~ m//` is the pattern matching operation. So, you would think that `while ($line =~ m/[aeiou]/)` would continue to loop as long as there are vowels to be found in the line. Now try running `readfile3.pl` and see what actually happens—apparently nothing! (Hit control-c to quit out)

What went wrong? The `while ($line =~ m/[aeiou]/)` command does in fact keep looping as long as there are vowels in the line, but the problem is that if we're not changing the line in any way, there will ALWAYS continue to be vowels in the line. The first line starts `Lorem ipsum dolor...`, and the `while` loop looks at the line, finds the first `o`, adds one to the `$count` variable, looks at the line again and finds the first `o` again, adds one to the variable, etc. etc. etc. (This is known as an infinite loop.) This is not really what we wanted.

We need a way to tell the pattern matching operator to look for different instances of vowels each time, or we need to alter the `$line` variable so we get rid of vowels that we've seen before. The first approach is to simply tell the matching operator that it should look sequentially at the whole line, rather than starting from the beginning every single time. This is done by adding the specification `g` (for 'global'): `while ($line =~ m/[aeiou]/g)`. The program `readfile3b.pl` uses this approach (try it out and see how it works).

Another approach is to zap each vowel as we count it, since it has served its purpose. Instead of searching with `m/ /`, we do a search and replace with `s/ / /`. In this case, we search for a vowel and replace it with nothing: `while ($line =~ s/[aeiou]//) {}`. (We could replace it with some deletion indicator, like a capital V or something, if we wanted a record of where the vowels had been; but in this case there's no need.) Now we don't get caught up on the first vowel over and over, because each time we see a vowel, we zap it. Once we've seen, counted, and deleted all of the vowels, the condition `$line =~ s/[aeiou]//)` is no longer true (the replace command `s/ / /` fails returning a false value), so the `while` loop is done, and we go on to the next line in the outer `while` loop. This is seen in the file `readfile3c.pl`.

A totally different approach, which introduces another useful operation, is seen in `readfile3d.pl`. Here, instead of scanning through the line for vowels in a loop, we start out by splitting the line up at the vowels, and seeing how many pieces we got. In particular, we consider the vowels to be "boundaries" (delimiters) between stretches of non-vowel material, and we break the line up into the pieces of non-vowel material: `Lorem ipsum dolor` becomes `L`, `r`, `m`, `ps`, `m d`, `l`, `r`. This is accomplished with the `split` command, which has the syntax `split(/delimiter/, $x)`, and splits the text `$x` into an array, starting a new item wherever it finds an instance of the delimiter.

In `readfile3d.pl`, we see this in action in the line `@non_vowels = split(/[aeiou]/, $line)`, which takes the text in `$line`, scans it for instances of the set `[aeiou]`, and puts all the pieces between the vowels into an array called `@non_vowels`. This leaves us with an array of the consonants around the vowels, but gets rid of the vowels themselves.[1] We can then infer how many vowels there were by counting how many non-vowel pieces there are.

It pays to be careful when counting things: the text `Lorem ipsum dolor` has 6 vowels, but when we split it into non-vowel pieces, we get 7 pieces, since the consonants "frame" the vowels: `L`, `r`, `m`, `ps`, `m d`, `l`, `r`. So, the number of vowels is the number of consonant pieces minus 1. The number of consonant pieces can be found by looking at the size of the array that holds them; recall that this is found with `$#` (`$#non_vowels`). This tells us the position of the last item in the array, which, because arrays start numbering at 0, is actually 1 less than the number of items that are in the array. (That is, the final `r` is item 6, and would be accessed as `$non_vowels[6]`, or `$non_vowels[$#non_vowels]`.) So, the number of vowels is equal to `$(#non_vowels + 1) - 1`, or simply `$#non_vowels`.

***Note that this program is not always exactly accurate, because there is the possibility that a line could begin or end with a vowel, in which case the consonants don't "frame" the vowels in the way that the previous paragraph supposes. We could fix this and guarantee that there really are always non-vowel pieces to the left and right of all vowels by first concatenating a non-vowel character to the beginning and end of every line: `$line = "x". $line ." x"`. This looks inelegant, but we'd never notice the difference on the surface. If the line actually begins with a consonant, it won't change the count (the first item is

[1] This technique is sort of like using plaster of paris to mold an impression of the area around something, or like the "lost-wax" casting method of sculptures.

now `xL` instead of `L`, but it's the same number of items). But if the line begins with a vowel, it provides a "bumper" consonant to ensure that the count always comes out right. I overlooked this problem when creating `readfile3d.pl` for you, but it's instructive to think about such problems.

The program `checkmath.pl` takes on the task of reading a file of arithmetic statements (such as `2 + 3 = 5`) and checking to see if they are correct. It opens the input file, and splits each line into an operation and an answer by looking for the equals sign (flanked by optional spaces). If the line is well-formed, this should give two items:`2 + 3`, and `5`). A new bit of syntax in this program is to assign the items to separate variables, rather than to an array: `($operation, $given_answer) = split(/\s*=\s*/,$line)`. It also demonstrates a couple other points of Perl syntax:

- After reading in a line, it performs the `chomp operation on it. This just removes the carriage return at the end of the line. It is quite common to want to remove the carriage return and work on just the ‘‘real’’ text.`
- In deciding whether the operation is +, -, *, or /, it uses a sequence of conditions: `if ...elsif ...elsif ...elsif ...else` (there are four possible legal values, but it would be dangerous to assume that just because there was no +, -, or *, there must be a /. The final `else` is a failsafe, in case the line is ill-formed by not having any of these operators.)
- In keeping track of whether an answer is correct or not, the program sets a variable called `$correct` to 0 (incorrect) or 1 (correct). This is a convention to representing true and false as numbers. The `unless` condition near the end of the program demonstrates the Perl also respects this convention: when we say (`unless($correct)`), Perl interprets this as meaning "unless the value of the `$correct` variable is true"—that is, unless it does not equal 0.[2] So when `$correct` is 0, the word "NOT" is printed.

The last program from the class handout is also the most parallel to the first assignment. It is actually quite a lot simpler than the previous program: it reads a file of text that is assumed to be romanized Japanese text in the Kunrei-shiki ("Monbushō") transliteration system, and converts it to the more commonly used Hepburn system. The Kunrei-shiki (which more closely parallels actual Japanese orthography) is a more phonemic spelling system, and does not reflect some phonological rules of Japanese. For example, the sequences [tsu] and [t͡ʃi] are written as *tu* and *ti*, not reflecting the phonological assibilation/affrication of /t/ before high vowels. Similarly, the sequence [ʃi] is written *si*, [fu] is written as *hu*, and palatal consonants in general are written with *y*: *sy*, *ty*, etc. In the Hepburn system, these are written in more "English-friendly" surface forms (details at: http://en.wikipedia.org/wiki/Romaji).

The program `hepburn.pl` opens a file, reads it in line by line, and searches for sequences that need rewriting in the Hepburn system. The syntax of the replacement commands is: `$line =~ s/search/replace/g`. The `g` at the tells Perl to look for `all` occurrences of the search string, not just the first one. The program is relatively straightforward, but something to notice is that the rule changing hu → fu must be ordered first, before (at least some of) the following rules. (Why is this the case?) One of the things that you should think about as you do the Italian assignment is how you might diagnose that a certain order is crucial—or, that it might *potentially* be crucial, even if it turns out not to be. (I don't expect that your program will do this, but we will discuss it in class next week)

There is one final bit of Perl syntax that will be useful in doing the assignment. Sometimes, you want to do a replacement on a string by searching for a pattern that contains wildcards or classes of segments, but you want to leave some of the wildcard segments unchanged. For example, suppose (counterfactually) that Japanese phonology palatalized $\text{s} \rightarrow \text{ʃ} \,/\, \underline{\quad} \begin{bmatrix} -\text{syl} \\ +\text{high} \end{bmatrix}$ (that is, before all high vowels, and not just before [i]). We want to search for the pattern `/s[iu]/`—but what do we replace it with? We can't just say `$line =~ s/s[iu]/sh/g`, because this would replace the sequences *si* and *su* with merely *sh* (we want *shi* and *shu*). What we need is a way to say "replace `s[iu]` with `sh` *and a copy of whatever matched the term* `[iu]`."

[2]Strictly speaking, any non-zero value is interpreted as true; it does not have to be exactly 1.

Fortunately, Perl provides an easy way to do this. If you want to "remember" a piece of the search so you can use it later (for instance, by putting a copy of it into the replacement), you can do this by putting parentheses around it. instead of searching for `s[iu]`, we search for `s([iu])`. Each time Perl sees a pair of parentheses in a search string, it stores the material that matches the pattern inside them as a numbered variable: `$1`, `$2`, etc. For example, if a word containing *si* is found to match the pattern `s([iu])`, the *i* part is stored as `$1`, since that is what matched the `[iu]` contained within the first parentheses. Then, to get back this material later, we simply use `$1`:

(4)
```
$line = s/s([iu])/sh$1/g;
```

These numbered variables are called "back-reference" variables. Perl starts a new batch of back-references each time you use `m/ /` or `s/ / /` to search for a pattern. (So, you don't need to worry that `$1` might contain matched material from a previous search.)