PERL Workbook

Instructor: Lisa Pearl

March 14, 2011

Contents

1 Useful reference books & websites

Book:

- O'Reilly's Learning Perl, 5th edition (which we will refer to extensively, and abbreviate as **LP5**):

  http://www.amazon.com/Learning-Perl-5th-Randal-Schwartz/dp/0596520107

Websites (useful for looking up examples of specific functions, control structures, etc. especially if you don't have your book handy):

- http://perldoc.perl.org
- http://www.comp.leeds.ac.uk/Perl/
- http://www.tizag.com/perlT/

2 Installing & executing PERL programs

2.1 MacOS & Unix

Fortunately, PERL should already be installed. Hurrah!

To run perl scripts in MacOS or Unix, first make sure the script has executable permissions. To check permissions on `yourscriptname.pl`, type the following at a terminal window command prompt to LiSt the detailed status of the file:

```
ls -l yourscriptname.pl
```

This will pull up the current permission status for your file, which may look something like this if your username is `lspearl`:

```
-rw-r--r-- 1 lspearl lspearl 919 Feb 4 2010 yourscriptname.pl
```

The first part (`"-rw-r--r--"`) tells you whether the file is a directory (`d`) or not (`-`), and about the permissions at the individual user level (`rw-` here), group level (`r--` here), and world level (`r--` here): `r` = permission to read from this file, `w` = permission to write to this file, and `x` = permission to execute this file. In this example file, the individual user (`lspearl`) has permission to write to the file while the individual user, group, and world all have permission to read from the file. Unfortunately, no one currently has permission to execute the file, which is what we need. To change permissions and add permission to eXecute for everyone, type the following:

```
chmod +x yourscriptname.pl
```

If you now `ls -l` the file, you'll notice the permissions look different:

```
-rwxr-xr-x 1 lspearl lspearl 919 Feb 4 2010 yourscriptname.pl
```

This shows that the file is executable. Once you're sure the file is executable, the only thing you need to do to run your perl script is type the name of the perl script in at the terminal command prompt, assuming you're currently in the directory where the script is:

```
yourscriptname.pl
```

For convenience of identification, perl scripts tend to have the `.pl` extension, though they don't actually need to since they're executable. You can use your favorite text editor to create perl scripts (I recommend Aquamacs for MacOS users, and Emacs for Unix users). Just make sure to save your script in plain text format so no strange invisible character formatting gets inserted.

An additional note for the creation of perl scripts: The very first line of any perl script you write needs to indicate where perl executables are located on your file system. For most users, this will be /usr/bin/perl (though it may also be /usr/local/bin/perl). Assuming your perl executables are located in the more usual place, you need this line at the beginning of any script:

```
#!/usr/bin/perl
```

All the perl scripts included in the bundled course file have this as their first line since they were written on a MacOS machine. Try running `shiny.pl` on your machine now. Then try creating a new perl script of your own that prints out some other message of your choice.

2.2 Windows

Sadly, PERL may not already be installed on most windows machines. Fortunately, this can be remedied fairly easily. A simple how-to is available at:

http://www.gossland.com/course/install_perl.html

Excerpts from this website are summarized below:

1. Go to http://www.activestate.com/activeperl/downloads to download Active Perl for Windows in MSI (Microsoft Installer) format.
2. Install it on your machine. Once you have this perl installation file downloaded, let it rip. It's best to accept all the default installation settings. After the installation, you'll find it in the `c:\perl` directory. You'll rarely need to go there directly though.

3. You need to reboot after a new installation to pick up the new path in the autoexec.bat file before you can run PERL. It doesn't matter how you installed it. Just reboot before going any further.

4. To test your installation once you've installed perl and rebooted, bring up a DOS window. It doesn't matter what directory you are in for now. Just type in:

   ```
   perl -v
   ```

   This should tell you what version of PERL you now have installed.

To create scripts, you can use your favorite text editor (I hear Notepad++ is good) and save your file in simple text format - though make sure to have a .pl extension to make it easy to identify your file as a perl script. Unlike the MacOS and Unix users, you don't need a special initial line at the beginning of your perl scripts. To run the sample scripts in the bundles course files, you may need to comment out or remove that initial line in order to get them to work. In general, you should be able to run perl scripts by typing this into the command window:

```
perl yourscriptname.pl
```

If you're interested in the particulars of what's going, refer to pp.15-16 in **LP5** where it talks about **Compiling Perl**. Try running `shiny.pl` on your machine. Remember, you may need to remove the first line to get it to work on your machine. Then try creating a new perl script of your own that prints out some other message of your choice.

2.3 Practicalities for all users: Where to get help.

Let's be honest - we all need help sometimes. Fortunately, there are a couple of good places to get help when you're contemplating a new perl script.

- Specific functions: the `perldoc` command typed in at the command prompt can help you out if you want to know what a specific function does and perhaps see an example. For example, you might be wondering about perl's built-in `log` function. Type this at the command prompt to find out if perl's `log` function uses base e or base 10:

  ```
  perldoc -u -f log
  ```

- General processes: google is definitely your friend when you're trying to figure out how to do something in perl (and are perhaps wondering if there's already a function that does that out there already). It will often pull up useful reference websites in addition to the main perl documentation website. For example, suppose you want to print out all the permutations of a particular sequence of numbers. Googling

"perl permutation" will likely lead you to the knowledge that there's some existing function out there called `List::Permutor`. Following this information trail, you'll likely discover something about a Permutor "module" you might like to use.

- CPAN: This stands for the Comprehensive Perl Archive Network. As its name suggests, it's a one-stop shop for all kinds of PERL goodies. (See p.9 in **LP5** for a bit more detail.) Go to http://www.cpan.org to check it out. For example, if you were interested in getting that Permutor module, you might try clicking on CPAN Search and then typing Permutor into the search box. This would lead you to the Permutor module (http://search.cpan.org/~phoenix/List-Permutor-0.022/Permutor.pm). CPAN also tells you how to install modules on your machine (http://www.cpan.org/modules/INSTALL.html). We'll talk more about using modules later in the input & output section, but if you can't wait, check out http://www.webreference.com/programming/perl/modules/index.html.

3 Scalar Data

3.1 Background & Reading

LP5 covers scalar data and basic manipulation of scalar data very well, so you should go ahead and become familiar with the following before trying the exercises for this section:

- what scalars are (p.19)
- numbers: floating-point literals (p.20), integer literals (pp.20-21)
- basic numeric operators (pp.21-22). Note also that ** is the exponential operator.
- strings: single-quoted string literals (pp.22-23), double-quoted string literals (p.23), string operators (pp.24-25), converting between numbers and strings (p.25)
- scalar variables: naming, assignment, binary assignment operators (pp.27-29)
- output with the `print` function (pp.29-30). Note that you can use parentheses for clarity, as the scripts in the course bundle do.
- basic control structures & control expressions: `if` (pp.33-34), boolean values (p.34), basic logical operators (top of p.164) [note: && = `and`, ||= `or`, and ! = `not` on p.32], `elsif` (pp.153-154), `while` (p.36), `undef` & `defined` (pp.36-38), `for` (pp.155-157), autoincrement & autodecrement (pp.154-155)
- basic user input functions (pp.34-36)

3.2 Exercises

1. What's the output difference between programs that differ only on the following line? If you're not sure, create a script that prints both and see what the difference is.

   program 1:

   ```
   print('Nothing?\nNothing?!\nNothing, tra la la?\n');
   ```

   program 2:

   ```
   print("Nothing?\nNothing?!\nNothing, tra la la?\n");
   ```

   What would the output look like if you changed all the instances of \n to \t in both programs above?

2. What does the following script do (`modulus_ex.pl`)? Note that this involves some concepts we haven't talked about yet, like the `split` command and array control loops like `foreach`. However, you should recognize the % operator.

```
#!/usr/bin/perl

$count = 0; # used to count letters

$word = "supercalifragilisticexpialidocious"; # scalar variable

# quick way to grab each character in a string

foreach $letter(split(//, $word)){

 $count++; # auto-increment the counter

 if($count % 2 == 0){ # check if counter has the right index value

   print("$count:  $letter\n"); # if so, print out counter & letter

 }

}
```

   How would you alter this script to print every fifth letter? What about if you wanted to print a . out every time it had processed 3 letters? (This sort of thing can be useful for tracking a program's progress on a large dataset.)

3. The program `names.pl` (code shown below) cycles through pre-defined lists of first and last names. Alter it so that the first and last names of each individual are printed out with a space between the first and last name, and a tab is inserted between each full name. At the end, a new line should be printed out. The output should look like this:

```
firstname1 lastname1     firstname2 lastname2 ...
```

   Code for `names.pl`

```
#!/usr/bin/perl

# list of first names

@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");

# list of last names

@lastnames = ("Williams", "King", "Beast", "Dwarf");

$index = 0; # counter for index in list
```

```
while($index <= $#firstnames){ # checking to see if at end of list
  $first = $firstnames[$index]; # get first name at current index
  $last = $lastnames[$index]; # get equivalent last name
  # probably want to add something here
  $index++; # auto-increment the index
}
```

How would you alter this program so that it assigns each full name (firstname and lastname separated by a space) to a new variable `$fullname`, before printing it out? What about if you needed to assign each full name to `$fullname` as last name, firstname (ex: Williams, Sarah)?

Below is a version (`names_with_bug.pl`) that tries to take a shortcut with the autoincrement operator. Unfortunately, it doesn't quite work. Try to figure out what the bug is and fix it, while still using the autoincrement operator in the position that's used in this script.

Code for `names_with_bug.pl`

```
#!/usr/bin/perl
# list of first names
@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");
# list of last names
@lastnames = ("Williams", "King", "Beast", "Dwarf");
$index = 0; # counter for index in list
while($index++ <= $#firstnames){ # auto-increment & end-of-list check
  $first = $firstnames[$index]; # get first name at current index
  $last = $lastnames[$index]; # get equivalent last name
  print("$first $last\t")
}
print("\n");
```

4. The `number_guess_with_bugs.pl` program is meant to have the following behavior:

- Print a welcoming message for the user.
- Ask the user to type in a numerical guess for the magic number.
- Tell the user if the guess is too high, too low, or just right.
- Give the user the option to type a character that means the user is tired of guessing and wants to quit instead of guessing.
- Quit with a closing message if the user guesses the magic number or types the quitting signal.

Unfortunately, as its name suggests, this program has a variety of bugs. See if you can fix them and get the program to do what it's supposed to do. If you get stuck, a working version of the program is `number_guess.pl`. However, you may (and probably will) find other ways to get it to work.

Once you get it working, how can you easily change it so that the quitting signal is 'no' instead of 'n'? What about changing what the magic number is?

The built-in `rand` function can be used to generate random numbers. Go here to see a very helpful description of it:

http://perlmeme.org/howtos/perlfunc/rand_function.html

Note that typing the following command at the command prompt will also get you a useful description of the `rand` function:

```
perldoc -f rand
```

How would you alter the (working) number guessing script so that it randomly generates an integer between 0 and 100 for the user to guess?

5. Write a perl script that generates a random sequence of 10 coin flips, where the user enters the probability of a heads coin flip. Make sure the user enters a probability between 0.0 and 1.0. If the user doesn't, re-ask until the probability is in this range. Also output the likelihood of generating that particular sequence of coin flips.

4 Lists & Arrays

4.1 Background & Reading

LP5 also covers lists & arrays very well, so you should go ahead and become familiar with the following before trying the exercises for this section:

- what arrays and lists are (pp.39-40)
- accessing elements of an array as scalar variables (p.40)
- special array indices, such as the last index [using the `$#` prefix] and negative indices (p.41)
- list & array assignment (pp.43-44)
- `foreach` control structure and special variable `$_` (pp.47-48)
- `reverse` and `sort` operators (pp.48-49)
- list vs scalar context and forcing scalar context (pp.50-52)
- reading multiple lines in at once from `STDIN` (pp.52-53)

4.2 Exercises

1. Below is a slow way to get the number of participants in the `@participants` list, especially if there are many participants (say, a million). What's a faster way? Write a script demonstrating that your way works.

```
$count = 0;

foreach (@participants){

  $count++;

}

print("There are $count participants.\n");
```

2. Suppose that the variable INPUTFILE refers to a file that has been opened for reading (the way that STDIN refers to standard input). How would you determine the number of lines in the file? (Check `count_file_lines.pl` for one way to do this if you get stuck, and try your own script out on `inputfile.txt`)

3. Suppose you're given the following arrays containing participant information:

```
@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");

@lastnames = ("Williams", "King", "Beast", "Dwarf");
```

Write a script that asks the user whether the names should be sorted by first or last names, and whether the names should be sorted alphabetically or reverse alphabetically. Then, sort the participant list this way, and print out the sorted list of participants. Look at `sort_names_reverseABC.pl` for an example of sorting these names reverse alphabetically.

4. Suppose you're given the following arrays containing participant information:

```
@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");

@lastnames = ("Williams", "King", "Beast", "Dwarf");

@ages = (15, 39, 33, 43);

@nativelanguages = ("English", "English", "Romanian", "English");

@performancescores = (85, 99, 35, 75);
```

Write a script that calculates the average performance score and prints out the members of the groups meeting the following criteria (each criterion should produce one group, rather than identifying a group that meets all four criteria):

(a) native language is English

(b) age is greater than 20

(c) age is greater than 20 and native language is English

(d) performance score is greater than 70

If you get stuck, sneak a peek at `group_stats_natlang.pl` for an example of using the "native language is English" criterion.

5 Input & Output

5.1 Background & Reading

LP5 covers basic input and output from files, so make sure to read over the following before trying the exercises for this section:

- opening a file to use as input, or to output data to (pp.83-84)
- recognizing bad file names, closing files, and the `die` function (pp.85-87)
- reading from and writing to files/file handles (pp.88-89)

In addition, one thing you might often wish to do is to allow the user to specify options for the program when the program is called (these are sometimes called command line options). This is usually better than explicitly asking the user what options to use with some sort of querying sequence at the beginning of a program. For instance, instead of asking a user what value some variable ought to be, you might want them to be able to call your perl program with that value already specified as a command line option (in the example below, calling `myperlprogram.pl` with `variable_value` set to 3):

```
myperlprogram.pl --variable_value 3
```

Fortunately, PERL already has built-in functions that will allow you to do this. However, these functions require using a perl module, so we'll now briefly discuss modules enough so that you can use these functions (and whatever others may turn out to be useful for the programs you intend to write).

5.1.1 PERL modules

Modules are bits of code that other people have already written to accomplish certain tasks. Your basic installation of perl will come with certain core modules that are there by default. CPAN (http://www.cpan.org) is a comprehensive archive of those and all the rest that people have submitted.

Go to `http://perldoc.perl.org/index-modules-A.html` to see a list of core modules in the latest release of PERL (navigate through the list alphabetically to see ones that begin with letters besides 'A'). If you navigate to 'G', you'll notice that Getopt::Long is a core module, and is the one we're going to use to get user options from the command line.

However, you may wish to use a module that's not a core module. CPAN has a detailed list of instructions for how to install PERL modules, which you should refer to if you need to install a PERL module on your system:

http://www.cpan.org/modules/INSTALL.html

Also, PC users will want to check out PPM, which is the package manager installed with ActivePerl:

http://docs.activestate.com/activeperl/5.10/faq/ActivePerl-faq2.html

Also, pp.169-171 in **LP5** talk about modules and installing them.

In fact, there may come a time when you want to write your own module, particularly if you want to reuse code you've already written. Keep the following web reference in mind for more details on how to do this:

http://www.webreference.com/programming/perl/modules/index.html

Meanwhile, suppose the module you're interested in is already installed on your machine, either because it's a core module or you've installed it from CPAN. You should be able to use the perldoc command to access information about that module. For example, try using perldoc to find out more about Getopt::Long:

perldoc Getopt::Long

This should actually be the same content as what you find at the website below:

http://perldoc.perl.org/Getopt/Long.html

5.1.2 Getopt::Long

Now let's look at how we use a perl module, in particular the Getopt::Long module. Using the `perldoc` command or going to the perldoc.perl.org entry for Getopt::Long (both mentioned above) should get you a lot of information about this module.

The synopsis is probably the first very useful piece of information, since it gives you an example usage:

```
use Getopt::Long;
  my $data = "file.dat";
  my $length = 24;
  my $verbose;
  $result = GetOptions ("length=i" => \$length, # numeric
                        "file=s" => \$data, # string
                        "verbose" => \$verbose); # flag
```

Breaking this down, the first thing to notice is that the `use` command is required to be able to use the functions in the Getopt::Long module. After this, we see some variables declared that will contain the user input, and currently contain default values (ex: $length = 24). The variable $result contains the output of the `GetOptions` function, which specifies the options that are defined at the command line and what kind of values are expected for them (numeric (`i`), string (`s`), or simply that they are used (the $verbose variable). If the GetOptions function executed correctly, the value of $result should be true (so this can be a good way to check if the function executed properly). Note the particular syntax that is used in the GetOptions function call:

- command option names used without a $ in front of them
- an optional = and letter following the command option name that indicate what kind of value to expect
- the => and then \ preceding the variable name prefaced by a $ (ex: => \$length)

So if this code was executed, the variable $length would now have the value the user entered for the `length` option, the variable $data would have the value the user entered for the `file` option, and the $verbose variable would have the value 1 if the user included `--verbose` in the command line call to the program and 0 if the user did not (as noted under the "Simple Options" section of the documentation).

The next section of documentation that's particularly useful to us is "Options with values". Here, we find that options that take values (like `length` and `data` in the above example code) must specify whether the option value is required or not, and what kind of value the option expects. To specify a value as required, use the = after the command option name (as in the example above: `length=i`); to specify a value as optional, use a : after the command option name instead (so to make the `length` command option not required, rewrite it as `length:i`). Expected value types for command options are `s` for strings, `i` for integers, and `f` for floating points. So, if we wanted to change the `length` command line option so it required a floating point value instead of an integer, we would write it as `length=f`.

If we look at the CPAN documentation for Getopt::Long, we can also find out what happens if we don't assign a command line option to a specific variable (look under "Default destination"):

http://cpan.uwinnipeg.ca/htdocs/Getopt-Long/Getopt/Long.pm.html

When no destination is specified for an option, GetOptions will store the resultant value in a global variable named opt_XXX, where XXX is the primary name of this option.

So, in the above example, if the `=> \$variable` notation makes you unhappy, you can rewrite it like this:

```
use Getopt::Long;

 $result = GetOptions ("length=i", # numeric
                       "file=s", # string
                       "verbose"); # flag
```

Now, when this code is run (and assuming GetOptions executes successfully), `$opt_length` should contain the integer value for the command line option `length`, `$opt_file` should contain the string value for the command line option `file`, and `$opt_verbose` should contain either 0 or 1 (depending on whether the user used it or not).

5.2 Exercises

1. Write a program that has the following behavior:

   - includes a description of the program's behavior and command line options allowed, accessible when the user enters `--help` after the program name at the command line
   - allows the user to specify that they want to read a file of names in (such as `names.txt`), with the default input file as `names.txt`
   - allows the user to specify an output file they wish to print the results to (default = STDOUT)
   - checks to see if the file has been successfully opened, stopping execution of the program if it has not been and printing out a message to the user indicating that the file was not able to be opened
   - sorts the list of names alphabetically
   - prints the sorted list to the user-specified output file

   If you get stuck on using the command line options or checking that the file has been opened successfully, look to `sort_names_begin.pl` for one way to do these parts.

2. Let's look at the exercise from the previous section, where we wanted to sort a list of names by either first or last names and sort them either alphabetically or reverse alphabetically. For example, suppose you're given this participant information:

   ```
   @firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");

   @lastnames = ("Williams", "King", "Beast", "Dwarf");
   ```

   You should rewrite your sorting script so it has the following behavior:

- print out a useful help message, accessible with the `--help` option
- allow the user to specify which name to sort by (default = last name) and what order to sort in (default = alphabetically, alternative = reverse)
- print the sorted results out to a user-specified file (default = STDOUT)

3. There are existing PERL modules for many different functions having to do with language. For example, the Lingua::EN::Syllable module can be found at

   http://search.cpan.org/~gregfast/Lingua-EN-Syllable-0.251/Syllable.pm

   This provides a quickie approximation of how many syllables are in an orthographically represented English word, which can be useful if you want a fast mostly-correct count of how many syllables are in a text.

   To do this exercise, you'll first need to install this module on your machine. Follow the CPAN installation instructions for your platform (Unix/Linux/PC - note that the current macosx system is equivalent to Unix). Then, write a script that has the following behavior:

   - prints out a useful help message, accessible with the `--help` option
   - reads in words one per line from a user-specified file (default = `words.txt`)
   - outputs the number of syllables for each word, one per line, as well as the total number of syllables in the file and the average number of syllables per word to a user-specified file (default = STDOUT)

   If you're having trouble figuring out how to use the Syllable.pm module, have a look at `call_syl_mod.pl` for an example call.

   How good is this syllable-counting algorithm? Can you find words where it isn't so accurate?

6 Subroutines

6.1 Background & Reading

LP5 covers the major aspects of subroutines very well, so read about the following before trying the exercises below:

- what a subroutine is (p.55)
- defining & invoking a subroutine (pp.55-56)
- subroutine return values (pp.56-58), the `return` operator (pp.65-66)
- subroutine arguments (pp.58-60)
- private variables in subroutines with `my` (pp.60, 63), persistent private variables with `state` (p.68-69)
- advanced sorting techniques using a sort subroutine, the <=> operator, and the `cmp` operator (pp.214-217)

6.2 Exercises

1. Suppose you're given the following arrays containing participant information:

   ```
   @usernames = ("Sarah1", "Sarah2", "sarah3", "sArah4");

   @scores = (10, 7, 42, 3);
   ```

   Write a program that outputs the participant information, sorted in one of the following ways:

   (a) ASCII-betical by participant username

   (b) case-insensitive ASCII-betical by participant username

   (c) numerical order by participant score (lowest to highest)

   (d) reverse numerical order by participant score (highest to lowest)

   The user should be able to choose which sorting order is preferred (default ASCII-betical on username) using a command line option. If you get stuck, have a peek at `sort_revnum.pl` for an example of sorting this information reverse numerically.

2. An exercise we did previously (listed again below) likely involved some repetitive code having to do with which criterion was being used to output participants.

Suppose you're given the following arrays containing participant information:

```
@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");

@lastnames = ("Williams", "King", "Beast", "Dwarf");

@ages = (15, 39, 33, 43);

@nativelanguages = ("English", "English", "Romanian", "English");

@performancescores = (85, 99, 35, 75);
```

Write a script that calculates the average performance score and prints out the members of the groups meeting the following criteria:

(a) native language is English

(b) age is greater than 20

(c) age is greater than 20 and native language is English

(d) performance score is greater than 70

Rewrite your script so that it uses a subroutine call to figure out which participants to output, based on the subroutine's argument, which will specify the criterion.

If you get stuck, have a look at `group_stats_critsub.pl` for an example of using subroutines to evaluate a particular criterion.

7 Larger Exercise 1: English Anaphoric *One* Learning Simulation

7.1 Background

The representation of the English referential pronoun `one` is a language example that is sometimes used to defend the necessity of prior language-specific knowledge for language acquisition. Here is an example of `one` being used referentially:

(1) "Look! A red bottle. Oh, look - there's another one!"

Most adults would agree that `one` refers to a red bottle, and not just a bottle (of any color). That is, the "one" the speaker is referring to is a bottle that specifically has the property red and this utterance would sound somewhat strange if the speaker actually was referring to a purple bottle. (Note: Though there are cases where the "any bottle" interpretation could become available, having to do with contextual clues and special emphasis on particular words in the utterance, the default interpretation seems to be "red bottle".) Lidz, Waxman, & Freedman (2003) ran a series of experiments with 18-month-old children to test their interpretations of `one` in utterances like this, and found that they too shared this intuition. So, Lidz, Waxman, & Freedman (2003) concluded that this knowledge about how to interpret `one` must be known by 18 months.

But what exactly is this knowledge? Well, in order to interpret "one" appropriately, a listener first has to determine the **antecedent** of "one", where the antecedent is the string that "one" is replacing. For example, the utterance above could be rewritten as

(2) "Look! A red bottle. Oh, look - there's another *red bottle*!"

So, the antecedent of "one" in the original utterance is the string "red bottle". According to common linguistic theory, the string "red bottle" has the following structure:

(3) [$_{N'}$ red [$_{N'}$ [$_{N0}$ bottle]]]

This bracketing notation indicates that the string "bottle" can be labeled as both syntactic category N':

(4) [$_{N'}$ [$_{N0}$ bottle]]

and syntactic category N0

(5) [$_{N0}$ bottle]

where N0 refers to a basic noun and N' is a label that applies to both nouns ("bottle") and nouns that contain modifiers ("red bottle"). Linguistic theory also states that referential elements (like `one`) can only replace strings that have the same syntactic category as they

do. So, since `one` can replace the string "red bottle", and "red bottle" is labeled with the syntactic category N', then `one` should also be labeled with syntactic category N'. If the syntactic category of `one` were instead N0, "one" could never replace a string like "red bottle" (it could only replace noun-only strings like "bottle") - and we could not get the interpretation that we do for the original utterance above.

So, under this view, adults (and 18-month-olds) have apparently learned that `one` should be syntactic category N', since they do get that interpretation. We can represent this knowledge state as the following:

- Syntactic Structure: The referential element `one` is syntactic category N' (rather than N0).
- Semantic Referent: When the potential antecedent of `one` contains a modifier (such as "red" above), that modifier is relevant for determining the referent of `one`. More specifically, the larger of the two N' string options should be chosen as the intended antecedent ("red bottle" instead of just "bottle"). This means that the intended referent is a referent that has the property indicated in the antecedent string (ex: "red" indicates a RED BOTTLE instead of any BOTTLE).

The question is how this knowledge is acquired, and specifically acquired by 18 months. The problem is the following. Suppose the child encounters utterance (1) from above ("Look! A red bottle. Oh, look - there's another one!"). Suppose this child is not sure which syntactic category `one` is in this case - N' or N0. If the child thinks the category is N0, then the only possible antecedent string is "bottle", and the child should look for the referent to be a BOTTLE. Even though this is the wrong representation for the syntactic category of `one`, the observable referent will in fact be a BOTTLE. How will the child realize that in fact this is the wrong representation, since the observable referent (a BOTTLE that also happens to be red) is compatible with this hypothesis?

Fortunately, these aren't the only referential "one" data around. There are cases where it's clear what the correct interpretation is. For example, suppose Jack has two bottles, a red one and a purple one. Suppose a child hears the following utterance:

(6) "Look! Jack has a red bottle. Oh, drat, he doesn't have another one, and we needed two."

Here, if the antecedent of "one" was "bottle", this would be a very strange thing to say since Jack clearly does have another bottle. However, if the antecedent of "one" was "red bottle", then this makes sense since Jack does not have another red bottle. Given the reasoning above, if the child realizes the antecedent is "red bottle", then the child knows that the category of `one` is N' (rather than N0). This kind of **unambiguous data** for `one` turns out to be pretty rare, though - Lidz, Waxman, & Freedman (2003) estimate that it comprises 0.25% of the data points that use `one` referentially in this way.

Lidz et al. thought this seemed like too few data points for children to learn the correct representation by 18 months, given other estimates of how much unambiguous data is required to learn specific linguistic knowledge (cf. Yang (2004)). So, nativists took this as an argument that children must somehow already know that `one` cannot be category N0. If they start by knowing the `one` in (1) must be category N', they can at least rule out the interpretation that relies on `one` being N0. (Though this still leaves two interpretations for `one` as category N': `one` replaces [$_{N'}$ red [$_{N'}$ [$_{N0}$ bottle]]] or `one` replaces [$_{N'}$ [$_{N0}$ bottle]]).

Regier & Gahl (2004) subsequently explored if it was really necessary for children to already know that `one` was not category N0, or if they could somehow learn this representational information from other data besides the unambiguous data. In particular, Regier & Gahl noted that more data than the unambiguous data could be relevant for learning `one`'s representation. Specifically, ambiguous data like (1) could actually be useful if children paid attention to how often the intended referent (ex: a RED BOTTLE) had the property in the potential antecedent (ex: "red bottle"). The more often this happens, the more of a "suspicious coincidence" (Xu & Tenenbaum 2007) it becomes. In particular, why does it keep happening that the potential antecedent includes the property as a modifer (ex: "red bottle") and the referent keeps having that property (ex: BOTTLE that is RED) if the property actually isn't important?

Using Bayesian inference, a child can leverage these continuing suspicious coincidences to decide that the potential antecedent containing the modifier (ex: "red bottle") is more likely to be the actual antecedent. If it's the actual antecedent of `one`, then `one` must be category N', since the string "red bottle" cannot be category N0. These ambiguous data actually comprise 4.6% of the child's referential `one` input, which is a number much more in line with the quantity of data researchers believe children need to learn linguistic knowledge by 18 months (Yang (2004)). Regier & Gahl simulated an incremental Bayesian learner that learned from both unambiguous data and these ambiguous data, and was able to learn the correct representation of `one`. They concluded no language-specific prior knowledge was required.

Pearl & Lidz (2009) took up the viewpoint that more data were relevant for learning the representation of `one` than had previously been considered. In addition to the unambiguous and ambiguous data considered by Regier & Gahl (2004), they considered a learner that was "equal opportunity" (EO) and used an additional kind of ambiguous data:

(7) "Look! A bottle. Oh, here's another one!"

While the semantic referent of this data point is clear (BOTTLE), it is ambiguous with respect to the syntactic category of `one`. If the child does not yet know the syntactic category of `one`, the antecedent can be either category N'

(8) [$_{N'}$ [$_{N0}$ bottle]]

or category N0

(9) [$_{N0}$ bottle]

So, these data are informative as well. Unfortunately, it turned out that they were informative in the wrong direction - favoring the N0 hypothesis over the N' hypothesis. This has to do with the nature of the strings in each category. The only strings in N0 are noun-only strings like "bottle". In contrast, N' strings can be noun-only strings and also noun+modifier strings like "red bottle". If the category of `one` is really N', it becomes a suspicious coincidence if noun-only strings keep being observed. Instead, it is more likely that the category is really N0. Also unfortunately, it turns out these kind of data make up a large portion of referential `one` data of this kind: 94.7% (and so they keep being observed).

Pearl & Lidz (2009) discovered that an incremental Bayesian learner sensitive to "suspicious coincidences", when learning from the unambiguous data and both ambiguous data types, fails to learn the proper representation of `one`. They concluded that a child must ignore this second type of ambiguous data, perhaps by using a filter that causes the child to ignore any data where the semantic referent isn't in question (in (7), the referent is clearly BOTTLE, so there is no semantic ambiguity). They argued that this learning filter was language-specific since it operated over language data, and placed priority on semantic clarity over syntactic clarity. So, while children did not need to have prior knowledge excluding the hypothesis that `one`'s syntactic category was N0, they still needed some prior knowledge in the form of a language-specific learning filter.

Pearl & Mis (in prep) took up the "there's more data out there" banner, and considered that there are other referential pronouns besides `one`, such as `it`, `him`, `her`, etc. When these pronouns are used referentially, their category label is NP (that is, they replace an entire noun phrase (NP)):

(10) "Look! The cute penguin. I want to hug it/him/her."

Here, "it" or "him" or "her" stands for "the cute penguin", which is an NP:

(11) [$_{NP}$ the [$_{N'}$ cute [$_{N'}$ [$_{N0}$ penguin]]]]

In fact, `one` can be used as an NP as well:

(12) "Look! A red bottle. I want one."

Here, "one" stands for "a red bottle", which is also an NP:

(13) [$_{NP}$ a [$_{N'}$ red [$_{N'}$ [$_{N0}$ bottle]]]]

So, the real issue of `one`'s category only occurs when it is being used in a syntactic environment where it cannot be the category NP. If `one`'s category must be something smaller than NP, and the question is which smaller category it is.

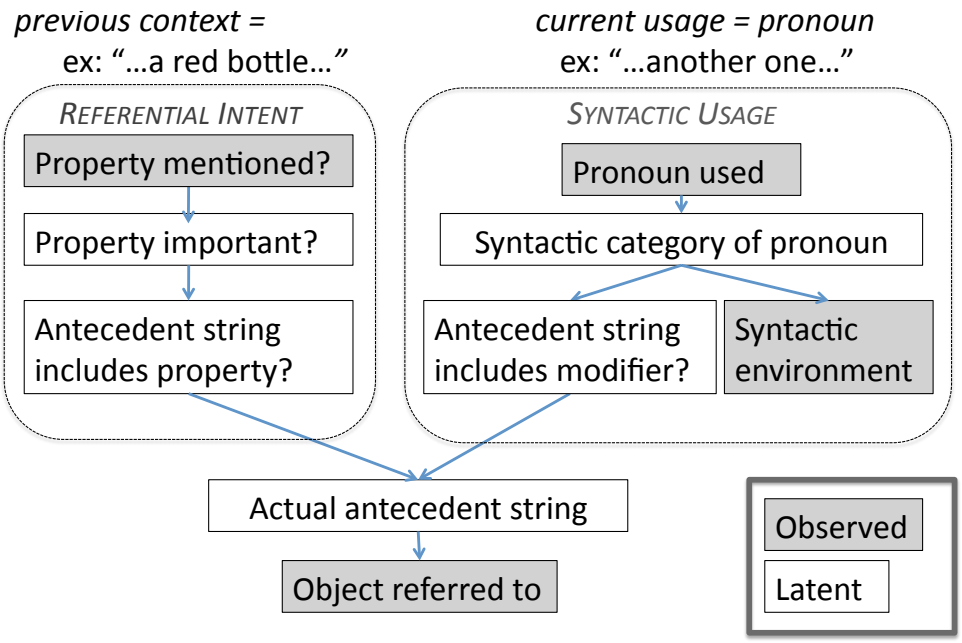

Figure 1: Information dependency graph of referential pronoun uses.

Getting back to the other pronoun data, one important piece of information for learning about one was whether the property included in the potential antecedent (ex: "red") was actually important for identifying the intended referent of the pronoun (ex: RED BOTTLE or just BOTTLE). This "is the property important" question can be informed by data like (10) and (12) - in these cases, the referent has the property mentioned. (Since the property (RED) is described by a modifier ("red"), and this modifier is always part of any NP string it's in, this should always be true.) Given this, Pearl & Mis conceived of a way to represent the observed and latent information in the situation where a referential pronoun is used and the potential antecedent mentions a property (which includes utterances like (1), (10), and (12)), shown in Figure 1.

Looking at the far right, we can see the dependencies for how the pronoun is used. Given a particular pronoun, that pronoun is labeled with a particular syntactic category that we do not observe (ex: NP for it, NP or something smaller than NP for one). Given that pronoun's category, the pronoun can be used in a certain syntactic environment (specifically, as an NP (ex: "...want it/one") or as something smaller than an NP (ex: "...want another one")), and we can observe that syntactic environment. Also dependent on the pronoun's category is whether the actual antecedent string is allowed to include a modifier like "red". If the syntactic category is N0, the antecedent cannot contain a modifier; otherwise, the antecedent can contain a modifier.

Looking back to the left of figure 1, we can see the dependencies for the referential intent of the pronoun used. We can observe if the previous context contains a potential antecedent that mentions a property (ex: "...a red bottle....want it..."). If it does, the question arises whether the property is important for picking out the intended referent (something we do not observe). This then determines if the actual antecedent string must include the mentioned property (also unobserved): if the property is important, it must be included; if the property is not important, it must not be included.

Turning now to the bottom of figure 1, the actual antecedent depends on whether this antecedent must include the mentioned property (from the referential intent side) and if it is allowed to include a modifier (from the syntactic category side). The object referred to by the speaker (which we can observe) depends on the actual antecedent (which we cannot observe).

Pearl & Mis then used this dependency graph to create a learner that would learn/infer the following things:

- p_{Imp}: the probability that the property is important for identifying the intended referent, given that the potential antecedent mentions that property
- $p_{N'}$: the probability that the syntactic category of the referential pronoun is N', given that the pronoun used is "one" and the syntactic environment in which it is used indicates that it is smaller than the NP category
- p_{corr_beh} (correct behavior in the Lidz, Waxman, & Freedman (2003) child study): the probability that the learner would look to the object that has the property mentioned in the potential antecedent when given the experimental prompt (ex: RED BOTTLE, given "Look! A red bottle. Do you see another one?")
- $p_{corr\_rep\_corr\_beh}$ (the probability that the learner has the correct representation of `one`, given that the learner has displayed the correct behavior in the Lidz, Waxman, & Freedman experimental setup): The correct behavior occurs when the learner has looked at the object that has the property mentioned in the potential antecedent. The correct representation for `one` in this scenario is that its category is N' and its antecedent is "red bottle" (indicating the property RED is important). This is basically a test of the assumption Lidz, Waxman, & Freedman made when they claimed that children looking to the red bottle meant that these children had the correct representation for `one`.

The equations that calculate these quantities are below (see Pearl & Mis (in prep) for details on how they were derived).

The general form of the update equations is in (14), which is based on an adapted equation from Chew (1971).

(14) $$p_x = \frac{\alpha + data_x}{\alpha + \beta + totaldata_x}, \alpha = \beta = 1$$

α and β represent a very weak prior when set to 1. $data_x$ represents how many informative data points indicative of x have been observed, while $totaldata_x$ represents the total number of potential x data points observed. After every informative data point, $data_x$ and $totaldata_x$ are updated as in (15), and then p_x is updated using equation (14). The variable ϕ_x indicates the probability that the current data point is an example of an x data point. For unambiguous data, $\phi_x = 1$; for ambiguous data $\phi_X < 1$.

(15a) $$data_x = data_x + \phi_x$$

(15b) $$totaldata_x = totaldata_x + 1$$

Equations for updating $\mathbf{p}_{Imp}$

The following equation is the general one for updating p_{Imp}:

(16) $$p_{Imp} = \frac{\alpha + data_{Imp}}{\alpha + \beta + totaldata_{Imp}}$$

where α and β are priors for the learner (both equal to 1 here), $data_{Imp}$ is the estimated number of informative data points seen so far that definitely indicate that the property mentioned in the antecedent is important, and $totaldata_{Imp}$ is the total number of informative data points for p_{Imp}.

- For unambiguous NP data like (10) and (12) (unamb_NP) and unambiguous "smaller than NP" data like (6) (unamb_lessNP):

  Since this data type unambiguously indicates the mentioned property is important, $\phi_{Imp} = 1$. Since this data type is informative for p_{Imp}, $totaldata_{Imp}$ is incremented by 1. Initially, d_{Imp} and tot_{Imp} are 0, since no data have been observed yet.

(17a) $$data_x = data_x + 1$$

(17b) $$totaldata_x = totaldata_x + 1$$

- For ambiguous "smaller than NP" data like (1) (ambig_lessNP_1):

(18a) $$data_{Imp} = data_{Imp} + \phi_{Imp}$$

(18b) $$totaldata_{Imp} = totaldata_{Imp} + 1$$

The equation for ϕ_{Imp} is:

(19) $$\phi_{Imp} = \frac{\rho_1}{\rho_1 + \rho_2 + \rho_3}$$

where

(20a) $$\rho_1 = p_{N'} * \frac{m}{n+m} * p_{Imp}$$

(20b) $$\rho_2 = p_{N'} * \frac{n}{n+m} * (1 - p_{Imp}) * \frac{1}{t}$$

(20c) $$\rho_3 = (1 - p_{N'}) * (1 - p_{Imp}) * \frac{1}{t}$$

and m (mod_strings) = the proportion of N' strings that contain modifiers (ex: "red bottle"), n (noun_strings) = the proportion of N' strings that contain only nouns (ex: "bottle"), and t (prop_types) = number of potential properties the learner is aware of (and that the intended referent could possibly have). The quantities in (20) correlate with anaphoric one representations. For ρ_1, the syntactic category is N' ($p_{N'}$), a modifier is used ($\frac{m}{n+m}$), and the property is important (p_{Imp}). For ρ_2, the syntactic category is N' ($p_{N'}$), a modifier is not used ($\frac{n}{n+m}$), the property is not important (1- p_{Imp}), and the object has the mentioned property by chance ($\frac{1}{t}$). For ρ_3, the syntactic category is N0 (1-$p_{N'}$), the property is not important (1- p_{Imp}), and the object has the mentioned property by chance ($\frac{1}{t}$).

- For all other data types:
  No update is done to p_{Imp}.

Equations for updating $p_{N'}$

The following equation is the general one for updating $p_{N'}$:

(21) $$p_{N'} = \frac{\alpha + data_{N'}}{\alpha + \beta + totaldata_{N'}}$$

where α and β are priors for the learner (both equal to 1 here), $data_{N'}$ is the estimated number of informative data points seen so far that indicate the syntactic category of one is N' when it is smaller than NP, and $totaldata_{N'}$ is the total number of informative data points for $p_{N'}$.

- For unambiguous "smaller than NP" data like (6) (unamb_lessNP):

(22a) $$data_{N'} = data_{N'} + 1$$

(22b) $$totaldata_{N'} = totaldata_{N'} + 1$$

Since this data type unambiguously indicates the category is N', $\phi_{N'}$=1. Since this data type is informative for $p_{N'}$, 1 is added to $totaldata_{N'}$. Initially, $data_{N'}$ and $totaldata_{N'}$ are 0, since no data have been observed yet.

- For ambiguous "smaller than NP" data like (1) (ambig_lessNP_1):

(23a) $$data_{N'} = data_{N'} + \phi_{N'1}$$

(23b) $$totaldata_{N'} = totaldata_{N'} + 1$$

where $\phi_{N'1}$ represents the probability that this particular data point indicates that the category of one is N'. The equation for $\phi_{N'1}$ is:

(24) $$\phi_{N'1} = \frac{\rho_1 + \rho_2}{\rho_1 + \rho_2 + \rho_3}$$

where ρ_1, ρ_2, and ρ_3 are the same as they were in (20) for calculating ϕ_{Imp}.

- For ambiguous "smaller than NP" data like (7) (ambig_lessNP_2):

(25a) $$data_{N'} = data_{N'} + \phi_{N'2}$$

(25b) $$totaldata_{N'} = totaldata_{N'} + 1$$

where $\phi_{N'2}$ represents the probability that this particular data point indicates that the category of one is N'. The equation for $\phi_{N'2}$ is:

(26) $$\phi_{N'2} = \frac{\rho_4}{\rho_4 + \rho_5}$$

where

(27a) $$\rho_4 = p_{N'} * \frac{n}{n+m}$$

(27b) $$\rho_5 = 1 - p_{N'}$$

The quantities in (26) intuitively correspond to representations for anaphoric one when no property is mentioned in the previous context. For ρ_4, the syntactic category is N' ($p_{N'}$) and the N' string uses only a noun ($\frac{n}{n+m}$). For ρ_5, the syntactic category is N0 (1-$p_{N'}$).

- For all other data types:
  No update is done to $p_{N'}$.

Equation for calculating $\mathbf{p}_{corr\_beh}$:

$$p_{corr\_beh} = \frac{\rho_1 + \rho_2 + \rho_3}{\rho_1 + 2 * \rho_2 + 2 * \rho_3} \tag{28}$$

where ρ_1, ρ_2, and ρ_3 are calculated as in (20) and $t = 2$ (since there are only 2 looking choices in the Lidz, Waxman, & Freedman (2003) experiment). As before, these quantities intuitively correspond to the different outcomes. The numerator represents all the outcomes where the learner looks to the correct object (ρ_1, ρ_2 and ρ_3 looking at the RED bottle), while the denominator includes the two additional outcomes where the learner looks to the incorrect object (ρ_2 and ρ_3 looking at the non-RED bottle).

Equation for calculating $\mathbf{p}_{corr\_rep\_corr\_beh}$:

$$p_{corr\_rep\_corr\_beh} = \frac{\rho_1}{\rho_1 + \rho_2 + \rho_3} \tag{29}$$

where ρ_1, ρ_2, and ρ_3 are calculated as in (20), again with $t = 2$. This is the same as taking the probability of the correct representation (represented by ρ_1, which stands for category = N', property = important), and dividing it by the probability of looking to the object that has the mentioned property (whether on purpose or by accident). You'll probably notice that this is the same equation as (19) (the only difference is the value of t). This has good intuitive appeal since ρ_1 in (20) corresponds to the correct representation where one's category is N' and the mentioned property is important.

7.2 The Exercise

Your task is to create a perl script that will simulate the Pearl & Mis learner. The learning process is as follows:

for each data point to be encountered

1. randomly generate a data point, based on the learner's estimated input distribution for referential pronouns
2. based on the data point's type, update p_{Imp} and/or $p_{N'}$ appropriately

At the end of learning (all data points have been encountered for the learning period), calculate p_{corr_beh} and $p_{corr\_rep\_corr\_beh}$. Your script should output these two probabiltiies, along with the final p_{Imp} and final $p_{N'}$.

You should allow the user to adjust the following quantities, via command line prompt (defaults come from the Pearl & Mis corpus analysis and Pearl & Lidz corpus analysis):

- data_points_to_run: number of data points the learner will encounter throughout the learning period [default = 36500, from Pearl & Mis's corpus analysis]
- unamb_NP: number of unambiguous NP data points (like (10) and (12)) [default = 3073, from Pearl & Mis's corpus analysis]
- unamb_lessNP: number of unambiguous data points where the syntactic environment indicates `one`'s category is smaller than NP (like (6)) [default = 0, from Pearl & Mis's corpus analysis]
- ambig_lessNP_1: number of ambiguous data points where `one`'s category is smaller than NP and the potential antecedent mentions a property (like (1)) [default = 242, from Pearl & Mis's corpus analysis]
- ambig_lessNP_2: number of ambiguous data points where `one`'s category is smaller than NP and the potential antecedent does not mention a property (like (7)) [default = 2743, from Pearl & Mis's corpus analysis]
- junk: number of uninformative data points (due to ungrammatical usage in Pearl & Lidz's model and/or due to being an uninformative data type in Pearl & Mis (in prep)'s model) [default = 30442, from Pearl & Mis's corpus analysis]
- mod_strings: m, number of noun+modifier string types (like "red bottle") in the N' category [default = 1, from Pearl & Lidz's corpus analysis]
- noun_strings: n, number of noun-only string types (like "bottle") in the N' category [default = 3, from Pearl & Lidz's corpus analysis]
- prop_types: t, number of properties the learner is aware of [default = 5, from Pearl & Lidz's model]

Your script should also output the values used by the learner for that simulation, so that the user knows what data distribution and what update equation quantities were used.

Do your learners always converge on the same final probabilities? Try the data distributions used by the unambiguous data learners, the Regier & Gahl learners, the Pearl & Lidz "Equal Opportunity" learners, and the Pearl & Mis learners:

Table 1: Input sets for different anaphoric one proposals, assuming the Brown/Eve distribution and 36,500 data points.

Data type	**Unamb**	**R&G**	**P&L's EO**	**P&M**
Unamb <NP	0	0	0	0
Ambig I	0	242	242	242
Ambig II	0	0	2743	2743
Unamb NP	0	0	0	3073
Uninformative	36500	36258	33515	30442

What kind of variation do you get? Try to vary the prop_types value (a higher number makes ambig_less_NP_1 data benefit the correct interpretation more). Do you find that the learner's behavior changes significantly between different ranges of values for prop_types? Does the P&M dataset seem to lead to correct infant looking behavior? What about the correct representation in general (category = N', property is important)?

8 Hashes & References

8.1 Background & Reading

Chapter 6 of **LP5** provides a nice introduction to the hash data structure, so make sure to read over these sections:

- what a hash data structure is, and why you would want to use one (pp.93-96)
- accessing elements and assigning values to elements in a hash (pp.96-98)
- copying entire hashes and inverting hashes (p.99)
- big arrow notation in hashes (p.100)
- functions useful for manipulating hashes: `keys`, `values`, `each`, `exists`, `delete` (pp.100-104)

In addition, you may also find it handy to pass larger data structures like hashes to subroutines as an argument. Suppose for example that we have two hashes, one for subject1 and one for subject2:

```
%subject1 = (
  first_name => 'Sarah',
  last_name => 'Williams',
  age => 15,
  native_language => 'English',
  performance_score => 85,
);
%subject2 = (
  first_name => 'Jareth',
  last_name => 'King',
  age => 39,
  native_language => 'English',
  performance_score => 99,
);
```

Suppose we want to pass these hashes to a subroutine `print_subject_info` that will print out their contents.

```
sub print_subject_info{
  my(%hash_to_print) = @_;
  while(($key, $value) = each (%hash_to_print)){
    print("$key is $value\n");
  }
}
```

We might then call it on each hash, one at a time:

```
print("Subject1 information is:\n");
print_subject_info(%subject1);
print("Subject2 information is:\n");
print_subject_info(%subject2);
```

To verify that this works, check out the first code block in `hash_refs_ex.pl`.

However, suppose you want to pass both hashes as arguments to a subroutine, perhaps to compare their `performance_score` values and print out the name and value of the subject who has the higher performance score. You might try calling a subroutine the following way:

```
print_higher_score_1(%subject1, %subject2);
```

However, try out the second code block in `hash_refs_ex.pl`. You'll find that this way of passing hashes doesn't work very well. In fact, you'll find you have the same problem passing multiple arrays - try the third code block in `hash_refs_ex.pl` to verify that passing multiple arrays as arguments doesn't work the way you expect.

The strangeness that's happening has to do with the way PERL processes subroutine arguments, which we won't go into here. A simple way to get around it (and probably better programming practice in general) is to use something called a reference to pass non-scalar variables like arrays and hashes to subroutines.

A reference to a variable is the numerical value associated with the variable's location in memory, sometimes called the variable's address. This means that a reference is always a scalar value (which is good, since PERL subroutines don't have problems with confusing multiple scalar arguments the way that they do for multiple array/hash arguments). The way to create a reference to a variable is by preceding it with a backslash (\):

```
@my_array = (1, 2, 3, 4, 5);

$ref_to_my_array = \@my_array;

$ref_to_subject1 = \%subject1;

$ref_to_subject2 = \%subject2;
```

It's then easy to pass a bunch of these references to a subroutine (check out the fourth code block in `hash_refs_ex.pl`):

```
print_higher_score_2($ref_to_subject1, $ref_to_subject2);
```

In fact, we can also skip making an explicit variable to hold the reference to our array/hash of interest, and make the references in the subroutine call (again, check out the fourth code block in `hash_refs_ex.pl`):

```
print_higher_score_2(\%subject1, \%subject2);
```

It's great that we can pass references to variables instead of the variables themselves, but how do we get the original information from the variable back out again? This is where we dereference. Dereferencing is exactly what it sounds like - getting rid of the reference, in this case by using the memory address stored in the reference to go find the original variable. So, if we want to recover the information from a variable reference, we use curly brackets { ...} around the reference:

```
@my_original_array = @{$ref_to_my_array};

%my_original_subject1 = %{$ref_to_subject1};
```

You'll notice in the fourth code block and the `print_higher_score_2` subroutine in `hash_refs_ex.pl` that we use this notation in order to call subroutines that expect arrays and hashes as their arguments, rather than references to arrays and hashes.

However, suppose you want to access just a single element in the data structure using the reference, such as the element in position 2 in `my_array` above, when you only have a reference to the data structure. One way to do it would be to use the curly brackets:

```
$second_pos_array_val = ${$ref_to_my_array}[2];
```

And we can do the same for accessing hash elements, when we only have a reference to the hash:

```
$first_name_subj1 = ${$ref_to_subject1}{"first_name"};
```

However, this looks rather messy. A prettier and more intuitive notation for dereferencing involves the arrow (`->`) operator. Place the arrow operator after the reference, and then use the normal notation for accessing array/hash elements (either square brackets for arrays or curly brackets for hashes).

```
$second_pos_array_val = $ref_to_my_array->[2];

$first_name_subj1 = $ref_to_subject1->{"first_name"};
```

To see these dereference options in action, run the fourth code block in `hash_refs_ex.pl`.

8.2 Exercises

1. Suppose you're given the following hash:

   ```
   %subject_performance = (
        Sarah => 85,
        Jareth => 99,
        Ludo => 35,
        Hoggle => 75,
       );
   ```

   Write a script that will print out the subject performance information in the following format:

   ```
   subject_name1:  subject_performance1
   subject_name2:  subject_performance2
   subject_name3:  subject_performance3
   subject_name4:  subject_performance4
   ```

   in the following four orders:

   (a) sorted alphabetically by name

   (b) sorted reverse alphabetically by name

   (c) sorted in ascending order by performance

   (d) sorted in descending order by performance

   If you get stuck, check out `hash_sort_abc.pl` for an example of how to print out the information in the first requested order.

2. Write a script that takes two hashes representing subject information (such as the two below), compares their performance scores, and prints out the information of the subject that has the higher performance score:

```
%subject1 = (
  first_name => ‘Sarah’,
  last_name => ‘Williams’,
  age => 15,
  native_language => ‘English’,
  performance_score => 85,
);
%subject2 = (
  first_name => ‘Jareth’,
  last_name => ‘King’,
  age => 39,
  native_language => ‘English’,
  performance_score => 99,
);
```

If you get stuck, have a look back at the example code in `hash_refs_ex.pl` for some ideas.

3. Remember (from the section on subroutines) that we looked at a problem of selecting participant information by certain criteria (such as “native language is English”). We represented participant information about four different participants by using arrays:

```
@firstnames = ("Sarah", "Jareth", "Ludo", "Hoggle");
@lastnames = ("Williams", "King", "Beast", "Dwarf");
@ages = (15, 39, 33, 43);
@nativelanguages = ("English", "English", "Romanian", "English");
@performancescores = (85, 99, 35, 75);
```

This representation forced us to do rather strange things while processing the data, such as concatenating a subject’s information together. A more intuitive way might be to use an array of hash references, with one hash per participant:

```
%subject1 = (
  first_name => ’Sarah’,
```

```
 last_name => 'Williams',
 age => 15,
 native_language => 'English',
 performance_score => 85,
);
%subject2 = (
 first_name => 'Jareth',
 last_name => 'King',
 age => 39,
 native_language => 'English',
 performance_score => 99,
);
%subject3 = (
 first_name => 'Ludo',
 last_name => 'Beast',
 age => 33,
 native_language => 'Romanian',
 performance_score => 35,
);
%subject4 = (
 first_name => 'Hoggle',
 last_name => 'Dwarf',
 age => 43,
 native_language => 'English',
 performance_score => 75,
);
@subjects = (\%subject1, \%subject2, \%subject3, \%subject4);
```

Write a script that will calculate the average performance score and print out the members of the groups meeting the following criteria, using the data representation above (an array of hash references):

(a) native language is English

(b) age is greater than 20

(c) age is greater than 20 and native language is English

(d) performance score is greater than 70

You'll probably want to take the script you wrote before when the information was all in arrays and modify it so it deals with this new data representation. If you get stuck, have a look at `array_hashrefs_firstcrit.pl` for one way to check the first criterion with this data representation.

9 Regular Expressions: Matching & Processing

9.1 Background & Reading

One of the things PERL is known for is `regular expressions`. As it does with many other topics, **LP5** covers aspects of regular expressions quite well, so you should read about the following before trying the exercises in this section:

- what regular expressions are (pp.107-108)
- matching simple patterns (pp.108-109)
- metacharacters and the backslash \(p.109)
- simple quantifiers: *, +, ? (pp.109-110)
- using parentheses, back references, and relative back references (pp.110-112)
- the vertical bar: | (pp.112-113)
- character classes (pp.113-115)
- option modifiers: /i, /s (pp.118-119)
- anchors: ^, $, \b (pp.120-121)
- binding operator: `=~` (pp.121-122)
- match variables: $1, $2, etc. (pp.123-124)
- non-capturing parentheses: (?:) (pp.125-126)
- general quantifiers (pp.129-130)
- precedence (useful in case strange errors are occurring with your regular expression) (pp.130-131)
- substitutions using s/// (pp.135-136)
- the global replacement modifier: /g (p.136)
- the case shifting escape characters: \U, \L, \E, \u, and \l (pp.137-138)
- the `split` and `join` functions (pp.138-140)
- nongreedy quantifiers: +?, *? (pp.141-142)

9.2 Exercises

1. Often, you will find that the output of the CLAN tool from CHILDES is something you'll want to format or otherwise extract information from. In this exercise, you should be able to use your newfound knowledge of regular expressions to help you pull out the necessary information.

   The file adam-verbs.adult contains the output of a query that pulls out a large number of verb uses in the child-directed speech in the Brown/Adam corpus. Lines like the following indicate which verb was identified in each utterance:

   ```
   *** File "adam01.cha":  line 95.  Keyword:  v|get
   ```

   Here, the verb get is contained in the utterance. As you'll notice by looking through the rest of that file, more than one verb can be contained in an utterance.

   (a) Write a script that counts how many times the verb `want` appears in adam-verbs.adult.

   (b) Write a script that counts how many times an utterance in adam-verbs.adult contains at least two verbs. (Hint: The `Keyword` section should indicate how many verbs are in the utterance.)

   (c) Write a script that tracks how many different verbs are used in the utterances (for example, `want`, `get`, `see` are all different verbs that are used) and outputs the list of verbs and how often each verb is used in numerically descending order (i.e., if `want` has 600 entries and `get` has 500, `want` should appear above `get`).

   If you get stuck on any of the above exercises, check out calculate_verbs_hints.pl for some helpful hints.

2. Another very useful thing you'll want to do is process a file containing a bunch of information and either convert it into a format that's easier to look at (especially for annotation purposes) and/or extract some simple information from it (in the case of annotation, perhaps once it's already marked up).

   (a) For the first part of this exercise, you will take a file that's in a format that is not so easy to look at, and extract the relevant portions of it in order to output a file that's much easier on the eyes. The file valian_thenps.out contains the output of a program that searches syntactically annotated utterances and identifies utterances that contain noun phrases including the word *the*. Suppose that we're only interested in the utterances themselves, rather than the syntactic annotation (and any of the other information available in the file). Fortunately, the utterances are easily identifiable, since they're preceded by /~* and followed by *~/, such as in the utterance below:

```
/~*
oh, that 's nice, a hug for the birdie, that 's nice.
*~/
```

You should write a perl script called `get_utterances.pl` that extracts all the utterances from valian_thenps.out that are delimited this way, and outputs them to a file called valian_thenps.utt, which should have the following format:

```
1
[first utterance]
2
[second utterance]
3
[third utterance]
```

etc.

Your script should work on any file that's formatted this way. How many utterances are there for the following files (all of which are child-directed speech from the CHILDES database to American English children of different ages)?

i. valian_thenps.out

ii. brownsarahfours_thenps.out

iii. gleasonfours_thenps.out

iv. hallfours_thenps.out

v. kuczajfours_thenps.out

vi. vanhoutenthrees_thenps.out

vii. vanhoutentwos_thenps.out

If you get stuck, look at `id_utterance.pl` for one way to identify when you've found an utterance you want to print out to the designated output file. You may also wish to explicitly output the number of utterances separately from the output file you're creating in order to help you answer the questions above.

(b) For the second part of this exercise, you will write a script called `get_counts.pl` that searches through an annotated file for certain tags that indicate that the tagged utterance contains certain properties. For example, someone might wish to annotate when there are plural noun phrases that contain the word `the`, given a file like the one output in the previous part of this exercise. A file so annotated might have lines that look like this:

```
63-PL
that 's right, at Halloween the children come knock on the door
```

since the noun phrase "the children" is a plural noun phrase, and the PL tag can indicate the utterance contains a plural noun phrase.

Your script should take as input the name of the file to be searched and the tag to search for. It should output the number of times the tag occurs in the input file. You can assume that the output file has the same format as the files you output in the first part of this exercise (see `valian_thenps.utt_ann` for an example). How many times do the following tags appear in the following files?

i. `PL` in valian_thenps.utt_ann (finding plural noun phrases)

ii. `All` in valian_thenps.utt_ann (finding noun phrases containing the word `all`)

iii. `DefPP` in valian_thenps.utt_ann (finding noun phrases containing the word `the` (sometimes called definite noun phrases) that are modified by preposition phrases)

iv. `DefCP` in valian_thenps.utt_ann (finding noun phrases containing the word `the` (sometimes called definite noun phrases) that are modified by entire clauses)

v. `FR` in hallfours_potFRs.utt_ann (finding structures called free relative clauses)

vi. `WHAT` in hallfours_potFRs.utt_ann (finding structures called free relative clauses that begin with the wh-word `what`)

If you get stuck, check out `id_tag.pl` for a regular expression that will help you identify the right lines to count.

10 Larger Exercise 2: Input Data Conversion

10.1 Background

One problem in language acquisition where computational models are often used is word segmentation (the identification of meaningful units in fluent speech). Models will often draw input sets from the CHILDES database. Usually, the model will want to operate over input that's in some sort of **phomemic** form (words written as they sound), rather than input that's still in **orthographic** form (that is, words written as they are spelled). For example, we spell *enough* with 6 letters but pronounce it using four sounds such as"ih", "n", "uh", "f". Since we think children are hearing input data rather than seeing its written form, a model should use input that's in some kind of phonemic form rather than input that's in orthographic form.

Unfortunately, most available data sets (with the exception of the derived corpora in CHILDES) are in orthographic form. Before a word segmentation model can be run, the input set needs to be converted to phonemic form. For example, one common encoding type that is used to create some of the derived corpora in the CHILDES database is shown in figure 2 below. Each ASCII symbol maps to one sound.

Consonants

ASCII	Ex.	ASCII	Ex.
D	THe	h	Hat
G	Jump	k	Cut
L	bottLe	l	Lamp
M	rhythM	m	Man
N	siNG	n	Net
S	SHip	p	Pipe
T	THin	r	Run
W	WHen	s	Sit
Z	aZure	t	Toy
b	Boy	v	View
c	CHip	w	We
d	Dog	y	You
f	Fox	z	Zip
g	Go	~	buttON

Vowels

ASCII	Ex.
&	thAt
6	About
7	bOY
9	flY
A	bUt
E	bEt
I	bIt
O	lAW
Q	bOUt
U	pUt
a	hOt
e	bAY
i	bEE
o	bOAt
u	bOOt

Rhotic Vowels

ASCII	Ex.
#	ARe
%	fOR
(	hERE
)	lURE
*	hAIR
3	bIRd
R	buttER

Figure 2: Example encoding system for orthographic forms in English.

The process of converting data from orthographic from to phonemic form usually goes something like this:

1. Use the CLAN tool to pull out all child-directed speech utterances from a data set of interest.

2. Create a list of words that are in those utterances.
3. Create the phonemic "translations" for the orthographic forms. For example, using the phonemic encoding in figure 2, we might translate *enough* as `InAf`.
4. Once the translation dictionary is ready, translate the orthographic forms to their phonemic equivalents in the utterances.
5. Once the translation is complete, compute some statistics on the input set such as
   - total number of word tokens in the input
   - total number of word types in the input
   - total number of utterances
   - average number of phonemes per word used
   - average number of phonemes per utterance
   - average number of words per utterance

10.2 The Exercise

1. Use the CLAN tool to extract the child-directed speech utterances from the first twelve files of the Brown/Eve corpus in the American English section of CHILDES (downloadable from here: http://childes.psy.cmu.edu/data/Eng-USA/Brown.zip), where Eve is younger than 2 years old (2;0). (*Hint: Try using the kwal command with the wildcard * character.*)
2. Write a PERL script to take the output of that CLAN search and extract just the content of the utterances, making sure to remove all the extraneous non-word characters and punctuation. Each utterance should be on its own line. For example,

   you xxx more cookies ? should become
   you more cookies

   how_about another graham+cracker ? should become
   how about another graham cracker

   oh (.) I took it . should become
   oh I took it

   peep@o [/] peep@o . should become
   peep peep

can you say +... should become
can you say

Name the file that contains this output eve1to12utt.txt.

3. The file dict_eng.txt contains orthographic to phonemic translations for a large number of words. If you've removed the extraneous characters from the utterances in eve1to12utt.txt correctly, all the words in the file should already be in dict_eng.txt. The dictionary file has the following format on each line:

   [word1_orth_form][some white space like tabs or spaces][word1_phon_form]

   Write a PERL script that translates the utterances in eve1to12utt.txt into their phonemic form, using the dictionary file dict_eng.txt. Call the translated file eve1to12utt.pho. For example, lines in eve1to12utt.txt that look like this:

   ```
   how about another graham cracker

   would that do just as well

   here
   ```

   should be translated in eve1to12utt.pho to something that looks like this:

   ```
   hQ 6bQt 6nADR gr&m kr&kR

   wUd D&t du GAst &z wEl

   h(
   ```

   (*Hint: You'll probably find the hash data structure useful for representing the information in the dictionary file.*)

4. Write a PERL script that computes the following statistics about the utterances in eve1to12utt.pho: (*Hint: You'll probably find the split command useful.*)

   - total number of word tokens in the input (Note: two instances of "cat" counts as 2 word tokens.)
   - total number of word types (lexicon items) in the input (Note: two instances of "cat" counts as 1 type/lexicon item)
   - total number of utterances
   - average number of phonemes per word used (Note: Calculate this over word types. This means we want the average number of phonemes of the words in the lexicon used to generate the observable utterances in the input. So, if "cat" occurs two times, it still only counts as one word type/lexicon item, and that type (if represented as `k&t`), has 3 phonemes.)

- average number of phonemes per utterance (Note: Calculate this over word tokens. This means that if "cat" appears twice in an utterance, we count the phonemes in it twice.)
- average number of words per utterance (Note: Calculate this over word tokens. This means if "cat" appears twice in an utterance, we count it twice.)

11 Larger Exercise 3: Calling a part-of-speech tagger and parser

11.1 Background: Freely available tagging & parsing software

Something you may find yourself wanting (or needing) to do with a data set consisting only of text is to identify the grammatical categories (like noun, verb, adjective, etc.) of the words in the data set, and/or identify the syntactic structure of the utterances in the data set. If you're working with data from the CHILDES database, you may be fortunate enough to have this information - the `%mor` line gives the grammatical category (sometimes called part-of-speech) while the `%gra` line provides one way of representing the syntactic structure of an utterance (called dependency trees).

However, what if you're working with a data set that doesn't already have the information? Fortunately, there is freely available software that will give a good approximation of part-of-speech and syntactic structure. One example of this kind of software comes from the Stanford NLP group (`http://nlp.stanford.edu/software/`). Among other things, the Stanford NLP group provides a free implementation of both a part-of-speech (POS) tagger and a syntactic parser that outputs the syntactic structure of an utterance:

Stanford POS Tagger:
`http://nlp.stanford.edu/software/tagger.shtml`

Stanford Parser:
`http://nlp.stanford.edu/software/lex-parser.shtml`

Go to each of these websites and download the latest version of both:

POS Tagger (English-only: basic English Stanford Tagger version 3.0 (re-entrant version): `stanford-postagger-2010-05-26.tgz`)

Stanford Parser (Stanford Parser version 1.6.5:
`stanford-parser-2010-11-30.tgz`)

The .tgz files should be able to be unzipped using your standard unzip tools (WinZip for Windows users, GNU tar for unix users, etc.) In general, double-clicking on the .tgz file should unzip it into a folder containing the necessary files.

11.1.1 Running the POS tagger from the command line

Once you've unzipped the POS tagger tgz file, you should have a folder containing all the necessary files needed to invoke the POS tagger. Have a look at the provided `README.txt` file for useful information, particularly the QUICKSTART instructions for running the

tagger on a file containing input utterances ("To tag a file using the pre-trained bidirectional model"):

```
java -mx300m -classpath stanford-postagger.jar edu.stanford.nlp.tagger.maxent.MaxentTagger -model
models/bidirectional-distsim-wsj-0-18.tagger -textFile sample-input.txt > sample-tagged.txt
```

This command runs a `java` program called `stanford-postagger.jar`, including a number of different arguments to the java program, such as the provided `sample-input.txt` input file as the `textFile`. In addition, it outputs the tagged file to `sample-tagged.txt`, using the > (the output redirect command).

If you have java installed on your computer, you should be able to execute this command successfully when you are in the folder that you unzipped. (That is, you should be able to execute this command from the command prompt where you execute perl scripts, provided that you have changed your current directory to be `stanford-postagger-2010-05-26`. If you are a Mac OSX or Unix user, you will likely already have java installed on your computer (one place it might be is in `\usr\bin\java`). If you are a Windows user, you may not. To check to see if you have java installed (and what version if so), type the following at the command line:

```
java -version
```

If you have java installed, this should pop up a message telling you what version you have installed, such as the following:

```
java version "1.6.0_22"

Java(TM) SE Runtime Environment (build 1.6.0_22-b04-307-10M3261)

Java HotSpot(TM) 64-Bit Server VM (build 17.1-b03-307, mixed mode)
```

If you do not have java installed, this command should cause a "command not found" error, or something equivalent. In that case, go to

```
http://www.java.com/en/download/help/download_options.xml
```

and follow the instructions to download and install java for your operating system.

Once you have java working, you should be able to run the `java` command from above that generates POS tags for the sentence in `sample-input.txt`. You may run into an "Out of Memory" error - if this is the case, you'll need to adjust the first argument in the java command from `-mx300m` (maximum memory to use is 300M) to something bigger, such as `-max1g` (max memory to use is 1G). The README.txt file actually recommends this in the paragraph that talks about memory usage.

Compare what is output into `sample-tagged.txt` with the POS tags in `sample-output.txt`. You hopefully will find that they're the same. The part-of-speech tags that are used come

from the PennTreeBank (a very popular data resource for natural language processing), and represent the following:

CC - Coordinating conjunction, examples: and, or, but

CD - Cardinal number, examples: one, two

DT - Determiner, examples: the, a, some

EX - Existential there, example: there in "there seems to be..."

FW - Foreign word, examples: gesundheit, oy

IN - Preposition or subordinating conjunction, examples: from, to, while, after

JJ - Adjective, examples: big, good, pretty, shiny

JJR - Adjective, comparative, examples: bigger, better, prettier, shinier

JJS - Adjective, superlative, examples: biggest, best, prettiest, shiniest

LS - List item marker, examples: items in lists like "one, two, three" or "a, b, c"

MD - Modal, examples: should, might, would

NN - Noun, singular or mass, examples: penguin, jewel, cheese, water

NNS - Noun, plural, examples: penguins, jewels

NNP - Proper noun, singular, examples: Sarah, Jareth

PDT - Predeterminer, examples: all in "all those penguins", such in "such as good time"

POS - Possessive ending, example: 's in Jareth's

PRP - Personal pronoun, examples: him, her, you, I

PRP$ - Possessive pronoun, examples: his, hers, mine

RB - Adverb, examples: really, easily, out

RBR - Adverb, comparative, example: later in "I'll come by later"

RP - Particle, example: up in "pick the letter up"

TO - to, example: to in "I want to do it"

UH - Interjection, examples: um, uh, yeah

VB - Verb, base form, examples: do in "I want to do it", "Can he do it?"

VBD - Verb, past tense, examples: kicked, hugged, ate, drank

VBG - Verb, gerund or present participle, examples: kicking, hugging, eating, drinking

VBN - Verb, past participle, examples: eaten, drunk

VBP - Verb, non-3rd person singular present, examples: do in "They do it", "You do it"

VBZ - Verb, 3rd person singular present, example: does in "He does it"

WDT - Wh-determiner, example: which in "which one"

WP - Wh-pronoun, example: what

WP$ - Possessive wh-pronoun, example: whose

WRB - Wh-adverb, examples: how, when

11.1.2 Running the parser from the command line

Once you've unzipped the pasrer tgz file, you should have a folder containing all the necessary files needed to invoke the parser. Have a look at the provided `README.txt` file for useful information, particularly for using the parser that generates PCFG (probabilistic context-free grammar) structures. The QUICKSTART section includes instructions for running the parser on a file containing input utterances ("UNIX COMMAND-LINE USAGE" - note that you can run this from the command prompt in Windows):

```
./lexparser.csh testsent.txt
```

This command runs a shell script that has the following commands in it:

```
set scriptdir=‘dirname $0‘

java -mx150m -cp "$scriptdir/stanford-parser.jar:" edu.stanford.nlp.parser.lexparser.LexicalizedParser
-outputFormat "penn,typedDependencies" $scriptdir/englishPCFG.ser.gz $*
```

The shell script runs a `java` program called `stanford-parser.jar`, including a number of different arguments to the java program, such as the requested output file formats (`-outputFormat`). You may notice similar arguments from the POS-tagger, such as the maximum amount of memory to use (`-mx150m`).

If you want to output to a file instead of to standard output, use a redirect > command:

```
./lexparser.csh testsent.txt > testsent.out
```

As with the POS-tagger, you need to have java installed on your machine in order to use the parser.

If you'd like to just invoke the parser directly from the command line (assuming your current directory is stanford-parser-2010-11-30), skip the shell script and use the following on the command line:

```
java -mx150m -cp "./stanford-parser.jar:" edu.stanford.nlp.parser.lexparser.LexicalizedParser
-outputFormat "penn,typedDependencies" englishPCFG.ser.gz testsent.txt > testsent.out
```

With this, you can then change the amount of maximum memory you're using (say you want 1G max instead of 150M max) and what output format you want (say you want PennTreeBank style phrase structure trees instead of dependency trees), among other things.

```
java -mx1g -cp "./stanford-parser.jar:" edu.stanford.nlp.parser.lexparser.LexicalizedParser -outputFormat
"penn" englishPCFG.ser.gz testsent.txt > testsent.out
```

11.2 The `system` function

A perl function that can be used to call other programs from inside a perl script is `system`. **LP5** discusses the basic usage of this function on pp.233-234. This will be useful if, for example, you want to run the part-of-speech tagger or the parser from within a perl script.

In order to call the Stanford POS tagger or parser from within a perl script, you simply invoke the command you want inside the `system` function. For example, look at `test-tagger-parser.pl` and try running it to see that it does indeed call these two programs from inside the script to tag and parse the sentences in the designated input file.

11.3 The Exercise

1. For the first part of this exercise, you will be running the Stanford POS tagger and parser on some child-directed speech utterances from the Brown/Adam corpus.

   adam-sample.txt contains the first 10 child-directed speech utterances from the first file in the Brown/Adam corpus from CHILDES (adam01.cha). Run the Stanford POS tagger on the utterances in this file. How does your output compare to the information provided in the %mor line of the CHILDES files, which appears in adam-sample-mor.txt? Does most of the information seem to be the same? What differences/errors do you notice?

   Note that there are some differences between the PennTreeBank tags used by the Stanford parser and the tags used by the software that generated the CHILDES %mor line:

   CHILDES %mor tags:

   adj = adjective

adv = adverb

co = communicator

conj = conjunction

det = determiner

fil = filler

int = interjection

n = noun

n:prop = proper name

neg = negation

num = number

on = onomatopoeia

prep = preposition

pro = pronoun

pro:dem = demonstrative pronouns like "that"

pro:refl = reflexive pronouns like "himself"

ptl = verb particle

qn or quan = quantifier

rel = complementizer

v = verb

wh = interrogative word like "who"

Now run the Stanford parser over the utterances in adam-sample.txt and generate Penn Treebank phrase structure trees. Given that output format, write a script that will count how many noun phrases occur, assuming those phrase structure trees are accurate. Note that noun phrases are indicated by either NP or WHNP. The general notation is that parentheses indicate a syntactic unit. So, for example, if we have a noun phrase like "the penguin", which includes the determiner "the" and the noun "penguin", we would end up with the following phrase structure tree:

```
(NP (DET the)
    (NN penguin))
```

This can be translated to the bracket notation introduced in chapter 7:

[$_{NP}$ [$_{DET}$ the] [$_{NN}$ penguin]]

There are actually 13 noun phrases (listed below) in adam-sample.txt, but your script should find a different number. Why does it behave this way? (Hint: Look at the phrase structure trees generated by the Stanford parser. What does it recognize as NPs? Do these look like the NPs listed below?)

Noun phrases in adam-sample.txt:

big drum, horse, who, that, those, checkers, two checkers, checkers, big horn, Mommy, Shadow, those, your checkers

2. In the second part of this exercise, you will tag and parse text that we don't already have part-of-speech and syntactic structure information for, and then use that annotation to help you track various statistics in the data that could be useful for detecting authorship deception. The folder imitation-attacks contains four files. One of these, sample_mccarthy01.txt, is a sample of one author's writing. The other files, imitation_*.txt, are attempts to consciously imitate this writing style with the goal of fooling someone about the actual authorship of the writing sample.

   First, use the POS tagger to tag all the files, and then gather some basic statistics about how often the following grammatical categories occur, relative to the total number of words used by each author:

   - nouns (NN, NNS, NNP, PRP, WP)
   - verbs (VB, VBD, VBG, VBN, VBP, VBZ, MD)
   - determiners (DT, PDT, PRP$, WP$, WDT)
   - adjectives and adverbs (JJ, JJR, JJS, RB, RBR)
   - infinitival to (TO)
   - interjections (UH)
   - conjunctions (CC)

   Note: You'll probably find the following commands helpful for getting a list of all the files in a directory $input_dir that end with a txt extension.

```
opendir(DIR, "$input_dir");

@txtfiles = grep(/\.txt$/,readdir(DIR));

closedir(DIR);
```

You script should output the ratio of each category to total number of words for each author's file (e.g., noun ratio = (total noun tokens)/(total word tokens)). Do any categories stand out as distinctive (significantly higher or lower) for the author's sample compared to the imitators? (Keep in mind we're just looking for a quick eyeball estimate here - sophisticated machine learning algorithms could be leveraged over the features here for more sophisticated judgments.)

Now, use the Stanford parser to provide syntactic structure for all four files, and then gather some more sophisticated statistics about the structures that occur in the files:

- The ratio of tensed embedded clauses (indicated by SBAR) to total utterances (indicated by ROOT)
- The ratio of sentence fragments (indicated by FRAG) to total utterances (indicated by ROOT)
- The ratio of noun phrases (indicated by NP or WHNP) to total utterances (indicated by ROOT)

As with the pos tagger results, your script should output the ratios for each author's file (e.g., tensed embedded clauses = (total embedded clauses)/(total utterances)). Do any of these features distinguish the original author from the imitators? What about if you combine both these features and the part-of-speech features from before - is the actual author distinct from each of the imitators? (Again, keep in mind that we're just looking for a quick eyeball judgment.)