Lecture 4:
You'll see she has a very tight grasp on Jack.

Last time
Prof. Neal Richman presented a number of map-related projects that allow communities to share data objects; Comments?
What did we see in Prof. Richman’s methodology? How does this connect to your projects?

... and the time before that
A short review of the functioning of an operating system; by discussing six common features of these programs, we learned a bit about our own Unix environment
We explained the idea of job control and saw how to be a good citizen on a multi-user system
We then learned the basics of regular expressions, a language for dealing with patterns; our data consisted of two hours of chat, sampled from IRC networks and “threaded” in time

Today
15 minutes of Unix
Shell programming and a closer look at the filesystem (mmm, permission bits)
From identifying patterns to programming with patterns; we move from grep to the basics of Perl
Homework related to today’s data
Unix shell(s)

Last time we identified user interaction as one of the features of an operating system; your (by now familiar) shell is one example of a text-based interface to the OS

Your default shell is tcsh which, as we show on the right, is just one flavor of shell

The computers in our department by and large run Mac OS X; they have a number of shells to choose from

How can we find programs?

As you might expect, Unix can help you find programs available to you

% which sh
/bin/sh
% which csh
/bin/csh
% which tcsh
/bin/tcsh
% which ksh
ksh: Command not found.

% ls /bin/*sh
/bin/bash /bin/csh /bin/sh
/bin/tcsh /bin/zsh

Why the choices?

A shell program was originally meant to take commands, interpret them and then execute some operation

Inevitably, one wants to collect a number of these operations into programs that execute compound tasks; at the same time you want to make interaction on the command line as easy as possible (a history mechanism, editing capabilities and so on)

The original Bourne shell is ideal for programming; the C-shell and its variants are good for interactive use; the Korn shell is a combination of both
New data

Last month, the Senate Judiciary Committee held confirmation hearings to decide if Judge John Roberts was a suitable candidate for Chief Justice of the Supreme Court.

As with all our data so far, transcripts from the proceedings are available on lab-compute:

scp -r lab-compute.stat.ucla.edu:/Data/roberts .

copy recursively (a directory and all its contents)
copy it to a directory of the same name (in this case "roberts") on your local machine

copy the directory /Data/roberts from the computer lab-compute.stat.ucla.edu

SPECTER: Good afternoon, ladies and gentlemen.

We begin these hearings on the confirmation of Judge John Roberts to be chief justice of the United States with first the introduction by Judge Roberts of his beautiful family, and then a few administrative housekeeping details before we begin the opening statements, which will be 10 minutes in length, by each senator.

At the conclusion of the opening statements, we will then turn to the introductions by Judge Lugar, Judge Warner — actually, Senator Lugar, Senator Warner and Senator Bayh, and then the administration of the oath to Judge Roberts and his opening statement.

So, Judge Roberts, if you would at this time introduce your family we would appreciate it.

ROBERTS: (OFF-MIKE) Peggy Roberts and Barbara Burke. Barbara’s husband Tim Burke is also here.

My uncle, Richard Podrasky (ph). Representing the cousins, my cousin, Jeannie Podrasky (ph).

My wife, Jane is right here, front and center, with our daughter, Josephine and our son, Jack. You’ll see she has a very tight grasp on Jack.

(LAUGHTER)

SPECTER: Thank you very much, Judge Roberts.

Judge Roberts had expressed his appreciation to have the introductions early. He said the maximum time of the children’s staying power was five minutes. And that is certainly understandable.

Thank you for doing that, Judge Roberts.

And now before beginning the opening statements, let me yield to my distinguished ranking member, Senator Leahy.

LEAHY: Well, Mr. Chairman, I want to thank you for all the consultations. I think we have had each other’s home phones on speed dial, we’ve talked to each other so often. And I have every confidence our chairman will conduct a fair and thorough hearing.

You know, less than a quarter of those of us currently serving in the Senate have exercised the Senate’s advice-and-consent responsibility in connection with a nomination to be chief justice of the United States. I think only 23 senators have actually been involved in that.

Structure of the transcripts

What regular expression indicates that a new person has begun speaking?

On this first page, there are other notations from the transcriber that represent non-spoken commentary; what regular expression describes these?
Counting the number of turns at the mic

First, let’s count the number of times each person speaks during the hearings; for the second day this might look like

```bash
% egrep '^[A-Z]+:' transcript2.txt
```

Next, how do we find out the number of times someone speaks?

... and now this should be familiar

We can finally count up each person’s contributions with our familiar flow through some `sorts` and `uniq’s`

```bash
egrep '^[A-Z]+:' transcript2.txt | cut -d"" -f1 | sort | uniq -c | sort -rn
```

423 ROBERTS
84 SPECTER
47 BIDEN
44 GRASSLEY
40 SESSIONS
34 SCHUMER
34 FEINGOLD
33 LEAHY
33 KENNEDY
32 FEINSTEIN
27 KOHL
25 DURBIN
21 DEMING
20 HATCH
19 CORNYN
17 KYL
14 GRASSLEY

Shell programs

Let’s try this for every day and not just the second; given that there are only four days of hearings, we could just repeat the above commands one at a time with the different transcript files

But assuming you have more files...

You can collect a series of Unix commands into a shell program; this allows you to repeat commands over different inputs
A simple shell program

At the right we have a short program contained in a file doit.sh

The first line tells the system which program to use to execute the file; otherwise we see a familiar command or two and some extra looping

The command echo simply prints out the line we've given it

```
#!/bin/sh
for i in `ls *.txt`
do
   echo "file: $i"
done
```

A simple shell program

• The symbol i is a variable in the shell; it is referenced by $i in our shell script

Variables are used by the shell to remember information; for example, when you start a shell, a number of variables get set by default

And now let’s run the program...

There are two ways to run a shell script; you can either execute it within a new shell

```
% sh doit.sh
```

Or you can make the script executable; that is, it becomes just like any command Unix knows about

Let’s see how this is done; it requires looking a little into how the filesystem specifies permissions, who can do what to a file
Permission bits

Unix can support many users on a single system and each user can belong to one or more groups.

Every file in a Unix filesystem is owned by some user and one of that user’s groups; each file also has a set of permissions specifying which users can:

- r: read
- w: write (modify) or
- x: execute

the file; these are specified with three "bits" and we need three sets of bits to define what the user can do, what their group (that owns the file) can do and what others can do.

In general...

The command chmod changes the permissions on a file; here are some examples:

- `% chmod g+x doit.sh`
- `% chmod ug-x doit.sh`
- `% chmod a+w doit.sh`
- `% chmod go-w doit.sh`

You can also use binary to express the permissions; so if we think of bits ordered as rwx, then:

- `r-x`: $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$
- `rwx`: $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$
- `r--`: $1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4$

and we can specify permissions with these values:

- `% chmod 755 doit.sh`

An example

- `% ls -al`

```bash
total 2240
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
A simple shell program

After that long detour, we could do the following

```bash
% chmod a+x doit.sh
% ./doit.sh
```

If we add a bit to our file, what do we expect to see after we run the following script?

```bash
#!/bin/sh
for i in `ls *.txt`
   do
      echo "file: $i"
      egrep '^[A-Z]+:' $i | cut -d"":" -f1 | sort | uniq -c | sort -rn > $i.cnt
done
```

From the four transcript files

What pattern do we see? What roles do each of these people play in the proceedings?

```makefile
30 SPECTER
  423 ROBERTS
  392 ROBERTS
  90 ROBERTS
4 ROBERTS
2 LUCAS
2 LEAHY
2 FEINSTEIN
1 WARNER
1 SESSIONS
1 SCHUMER
1 KYL
1 KOHL
1 KENNEDY
1 HATCH
1 GRASSLEY
1 GRAHAM
1 FEINGOLD
1 DURBIN
1 DEWINE
1 CORNYN
1 BROWNBACK
1 BIDEN
1 BAHN
```
Going a little deeper

The NY Times (September 4, 2005) quotes something called the 80-20 rule; if the senators are talking 80% of the time, you’re winning.

How would we investigate this? What about peering deeper and compare participants based on the words they used?

Perl: The practical extraction and report language

In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible.

And what are these “easy jobs” that ought to be easy? The ones you do every day, of course. You want a language that makes it easy to manipulate numbers and text, files and directories, computers and networks, and especially programs. It should be easy to run external programs and scan their output for interesting tidbits. It should be easy to send those same tidbits off to other programs that can do special things with them. It should be easy to develop, modify, and debug your own programs too. And, of course, it should be easy to compile and run your programs, and do it portably, on any modern operating system.

Perl does all that and a whole lot more.

Larry Wall

Much of the explosive growth of Perl has been fueled by the hankerings of former UNIX programmers who wanted to take along with them as much of the “old country” as they could. For them, Perl is the portable distillation of UNIX culture, an oasis in the wilderness of “can’t get there from here.”

Larry Wall
The obligatory step back... Awk

*A awk was originally designed and implemented by its authors in 1977, in part as an experiment to see how the Unix tools grep and sed could be generalized to deal with numbers as well as text.

Aho, Kernighan, Weinberger

Perl basics: The call

Perl commands can be invoked from the Unix shell

```
% perl -e 'print "hello\n"
```

Or Perl commands can be collected into a file

```
perl greeting.pl
print "hello\n";
```
Perl basics: The call

We can also assemble commands into a Perl executable

This requires adding a line to the beginning of your file; remember that on Unix systems, if the first two characters on the first line of a text file are "#!", then what follows is the name of the program that executes the rest of the file

We then tell the operating system that this file is executable using the chmod command

```
chmod +x greeting.pl
```

```
#!/usr/bin/perl
print "hello\n";
greeting.pl
```

Perl basics: A word on formatting

Perl is free-form, meaning you can add whitespace (tabs, newlines, spaces) to your program to make it easier to read

The "#" sign denotes comments; everything on a line following this character is ignored by Perl
Now that we can write a simple program...

Let's consider the the overall structure of the language

- Basic data types
- Operators
- Control structures
- Input/output
- Regular expressions

... but we'll do it in a task-oriented fashion

The easiest path out...

- The `print` command is one extremely popular route out of Perl
- We will elaborate on this command shortly

Getting data into a Perl program

- Data can either be included in the program in the form of a literal value
- Or we can also take input from the user
- Or we can scan data from a file

1. How do we get data into and out of Perl?
Let's start with the last first...

A simple program to read data from a file

```perl
#!/usr/bin/perl
open(FH,"<transcript2.txt");
while($str = <FH>){
   print $str;
}
```

If we “read past” some of the coding details, this program reads lines from the file `transcript2.txt` and simply prints them out again.

The `while` loop just keeps reading until we hit the bottom of the file; we’ll explain the `while` business shortly.

---

Getting data into a Perl program

FH is called a filehandle

Filehandles can be used to specify where Perl should read and write data.

```perl
open(DATA,"<file.txt");  # take input from file.txt
open(OUT,">results.txt"); # write to results.txt
open(OUT,">>results.txt"); # write to the end of results.txt
```

By convention, we usually choose only uppercase letters to specify filehandles.

---

Getting data into and out of a Perl program

In general, filehandles specify I/O connections and not just files.

So, we can take data from a pipes...

```perl
open(DATA,"egrep ROBERTS: transcript2.txt | ");
open(OUT,"| sort > results.txt");
```
Getting data into and out of a Perl program

Perl provides some “built-in” filehandlers

STDIN, STDOUT, STDERR

... among others.

Getting data into and out of a Perl program

The command print writes data to STDOUT by default (i.e., your terminal window)

You can use filehandles to write data to different locations

open(OUT,">results.txt");  # open a file for writing
print OUT "Hello there\n";  # write to the file

Getting data into and out of Perl

In our simple program, we are reading data from a filehandle using the line input operator <>

```
#!/usr/bin/perl
open(FH,"<transcript2.txt");
while($str = <FH>){
   print $str;
}
```

In this case, we are just reading from a file, but recall that the filehandle can read from the output of a series of piped commands

Getting data into and out of Perl

You can also slim down the program and leave the data source unspecified...

```
 simple.pl
 #!/usr/bin/perl
 while($str = <>){
    print $str;
 }
```

In this way, we can call the program using either of the following

```
./simple.pl transcript2.txt
cat transcript2.txt | ./simple.pl
```
Getting out of “Getting data into and out of Perl”

You can close `FH` with the command

```
close(FH);
```

Perl will automatically close all filehandles when your program finishes.

It will also close a filehandle if you try to open it again later in the program; that is, if you have two `open(FH,)` in a program.

Basic data types: Scalars

A single instance of something is called a scalar.

The most frequently used scalars in Perl are numbers and strings.

Scalar data: Numbers

A *literal* is the way a value is represented in the source code of a Perl program.

We can refer to the following kinds of numbers (*numeric literals*) in a Perl program:

- 1.45
- 256.000
- 7.3e45
- -1.2E-3
- 22
- 2004
- -200

2. What kinds of data does Perl know about?
Basic data types: Scalars

Great, but how do you store values?

In Perl, scalar variables are prefaced with the symbol $ (think of it as a stylized "S" for "scalar")

You can assign scalar values using "="

```perl
$year = 2004;
$dow = "wednesday";
$phrase = "today is $dow";  # interpolation
$pi = 3.14159;
$x = 2*$pi + 10;
$x = $x + 5;                # same variable twice
$pi = "Mmm, pie!\n";  # pi is now a string
```

Basic data types: Scalars

So back to our simple program

```perl
#!/usr/bin/perl
open(FH,"<transcript2.txt");
while($str = <FH>){
   print $str;
}
```

A slightly enlarged version of our simple program

```perl
#!/usr/bin/perl
open(FH,"<transcript2.txt");
while($str = <FH>){
   chomp $str;
   @words = split " ",$str;
   print "@words[0]\n";
}
```

A slightly enlarged version of our simple program

One command to notice: chomp

The scalar $str holds lines from transcript2.txt; that means all the characters we can read plus a newline

The command `chomp` removes that trailing newline (poetic isn't it?)

A more brutal command, `chop`, strips the last character from a string, whether it is an end of record marker or not
A slightly enlarged version of our simple program

Next notice the reference to $words[0]

#!/usr/bin/perl
open(FH, "<transcript2.txt");
while($str = <FH>){
    chomp $str;
    @words = split " ", $str;
    print "$words[0]\n";
}

Basic data types: Lists and arrays

Scalars were single items, lists represent multiple items; lists are stored in structures called arrays

We access separate entries of an array using [ and ]

$tally[0] = 3;
$tally[1] = "fancy";
$tally[2] = 3*17;

Basic data types: Lists and arrays

Arrays begin with the index zero; and $#name (where name refers to the name of the array) is used to indicate the last element

As with any variable in Perl, it is assigned a value undef before it has been assigned

$tally[0] = 3;
$tally[1] = "fancy";
$tally[99] = 3*17;
print "#$tally\n";

Basic data types: Lists and arrays

Sometimes, we make reference to an entire array and not just its elements

In that case, we preface the variable name not with $ but with @ (think of this as a fancy way to write an "a", as in "array")

@tally = 1 .. 10;  # the range operator
@tally = ("fancy", 3*17, 5);
@tallyho = (@tally, 25);
A slightly enlarged version of our simple program

In this case, the command \texttt{split} breaks the line we read from \texttt{transcript2.txt} into pieces, where the pieces are defined by one or more blank spaces (sorta like \texttt{cut} but you get to "hold" the output) and stored in the array \texttt{@words}

\begin{verbatim}
#!/usr/bin/perl
open(FH,"<transcript2.txt");
while($str = <FH>){
    chomp $str;
    @words = split " ",$str;
    print "$words[0]\n";
}
\end{verbatim}

An example

Using this program, the line

\begin{quote}
SPECTER: It is 9:30. The confirmation hearing of Judge Roberts will now proceed.
\end{quote}

is stored into \texttt{@words} as if we had typed

\begin{verbatim}
$words[0] = "SPECTER:";
$words[1] = "It";
$words[2] = "is";
$words[3] = "9:30.";
... 
\end{verbatim}

But what are we printing?

The expression \texttt{"\$words[0]\n"} takes the first element in the array \texttt{@words}, adds a newline (a carriage return) to the end and then passes the resulting string to print

This is similar to what we did in our first program \texttt{greeting.pl}

3. How do I run through the elements of an array and print them?
Control structures: while loops

So far, we’ve seen an example of a while loop

```perl
while(    ){
   ....
}
```

The loop continues to execute until the condition in () is no longer true; we’ll talk about logical comparisons of various kinds next time.

Perl has many different kinds of constructs to “control the flow” of a running program; we will talk about more of these next time.

Control structures: foreach

But to get to the question, we can do something like this

```perl
foreach $x (@some_array){
   print "$x\n";
}
```

As this code runs, it replaces $x with each element of @some_array.

A more traditional “for” loop can be written as

```perl
foreach $i (0 .. $#some_array){
   print "$some_array[$i]\n";
}
```

Perl has “for” loops also (proving that there really are a million ways to do the same thing in Perl).

An even bigger version of our simple program

```perl
#!/usr/bin/perl
while($str = <>){
   chomp $str;
   @words = split " ", $str;
   foreach $word (@words){
      print "$word\n";
   }
}
```

doit.pl
Back to the 80-20 rule

Admittedly we have introduced a lot of machinery to address a rather low purpose

But we’re within striking distance of actually doing something useful

Here’s an extended program; what is it doing?

```perl
#!/usr/bin/perl
while($str = <>){
  chomp $str;
  @words = split " ",$str;
  if($words[0] =~ m/^[A-Z]+:/){
      close(DF);
      open(DF,">$words[0].txt");
      shift @words;          # removes first element
                          # of an array
  }
  foreach $word (@words){
    print DF "$word
";
  }
}
```

What’s going on?

The new piece

```perl
$words[0] =~ m/^[A-Z]+:/
```

is a regular expression; instead of enclosing it in single quotes like we would for `egrep`, it is bound by `/`s

This expression is "true" if the variable `$words[0]` is a match for the pattern we specified
For next time

1. Try the code snippets in this lecture and produce the files of words for each senator; write a shell script that will take each file and produce a sorted list of words (most to least frequently used). Turn in the top 15 and the bottom 15 words for senators Biden, Specter and Grassley and for Judge Roberts.

2. Comment on whether this program is actually capturing all the words used by these senators or something more; if we are not selecting just words, describe what needs to be added (not literal code, just concept).

3. Create a regular expression that describes commentary added by the transcriber and create a list of the unique additions.