Ling 5801: Lecture Notes 3
From Regular Expressions to Scripting

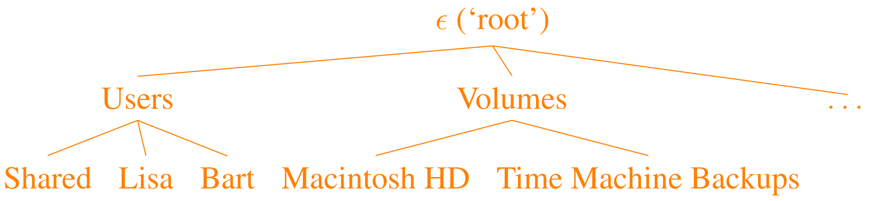
We generally run corpus experiments, etc., by typing **unix commands** into a **terminal window**.

Practical: If you're new to this, here's how to open a **terminal window**:
- On Mac/OSX: find the Terminal app in Finder under Applications/Utilities.
- On Windows: enable 'Windows Subsystem for Linux', then find Terminal in the Start menu. You should then type `cd /mnt/c/Users/`⟨yourname⟩ to be in your PC home directory.

Contents

3.1 Notation for directory structure

Commands usually manipulate **files**, organized in your computer's **directories** like a family tree:



(Here, and elsewhere in these notes, 'ϵ' indicates an empty name, consisting of no characters.)

Files are identified using **paths**: sequences of directories, delimited by the slash character '`/`'.

For example, if you are at the root (topmost) directory, the path:

```
Users/Lisa/sample.txt
```

denotes the `sample.txt` in the `Lisa` directory, which is in the `Users` directory.

Paths may also refer to **parent** directories '`..`' to ascend back up the tree.

For example, if you are in the `Lisa` directory:

```
../Bart
```

denotes the `Bart` directory, which is a sibling to the `Lisa` directory.

3.2 Unix commands for basic navigation

The general format of a command is: ⟨operator⟩ ⟨argument⟩, like a verb followed by a direct object.

Some useful commands to start with:

1. `pwd` prints the working directory.

   For example, assuming the Terminal app starts you in your home directory, type:

   ```
   pwd
   ```

   and unix should respond with a path, like:

   ```
   /Users/yourname
   ```

2. `curl -O` ⟨web-address⟩

   downloads the html of the ⟨web-address⟩ argument.

   For example:

   ```
   curl -O https://www.asc.ohio-state.edu/schuler.77/courses/5801/sample.txt
   ls
   ```

   should copy the file '`sample.txt`' from the web into the current directory:

   ```
   sample.txt
   ```

3. `ls` lists the contents of the current directory.

   For example, if you curled the sample txt file from the course web site, type:

   ```
   ls
   ```

   and unix should respond with something like:

   ```
   sample.txt
   ```

4. `cd` ⟨path⟩ changes the current directory to the ⟨path⟩ argument.

   For example:

   ```
   cd ..
   pwd
   ```

   should show that you're in the Users directory:

   ```
   /Users
   ```

   and:

```
cd yourname
pwd
```

should show that you are back in your home directory:

```
/Users/yourname
```

5. `mkdir` creates a new subdirectory.

   For example:

   ```
   mkdir PS1
   ls
   ```

   should print:

   ```
   sample.txt
   PS1
   ```

6. `open` ⟨path⟩

   edits an existing file (mac only!).

   For example:

   ```
   open sample.txt
   ```

   (It chooses the editor based on the file extension: the part of the filename after '`.`', if any.)

3.3 Unix commands with path patterns using regular expressions

Regular expressions make their way into some useful unix commands.

Many commands use regexp-like **path patterns** to match filenames as paths from the current directory ('`*`' is repeated wildcard; '`[a-z]`' is character range; '`{.tex,.bib}`' is disjunction, '`/`' delimits directories, and '`..`' backs up a directory — e.g. '`../*/[0-9]*{.h,.o}`' matches files in siblings to the current directory that begin with a number, end with .h or .o):

7. `ls` ⟨path-pattern⟩

   lists all files matching ⟨path-pattern⟩.

   For example:

   ```
   ls ../*
   ```

   prints a list of the files in directory above ('..') the current directory.

8. `mv` ⟨path-pattern⟩ ⟨path⟩

   moves file(s) matching ⟨path-pattern⟩ to directory (or new file name) ⟨path⟩.

   For example:

   ```
   mv sample.txt myfile.txt
   ```

changes the name of ‘sample.txt’ to ‘myfile.txt’, and this:

```
mv myfile.txt sample.txt
```

changes it back, and this:

```
mv sample.txt PS1/sample.txt
```

moves ‘sample.txt’ into the ‘PS1’ directory without changing its name.

9. `cp` ⟨path-pattern⟩ ⟨path⟩

   copies file(s) matching ⟨path-pattern⟩ to directory (or new file name) ⟨path⟩ (same as `mv`, but preserves the old file).

   For example:

   ```
   cp sample.txt sample.txt.backup
   ```

10. `rm` ⟨path-pattern⟩

    removes (i.e. deletes) file(s) matching ⟨path-pattern⟩.

    For example:

    ```
    rm *.backup
    ```

11. `rmdir` ⟨path-pattern⟩

    removes (i.e. deletes) directory(-ies) matching ⟨path-pattern⟩.

Unix commands for reading files, also using path patterns:

12. `cat` ⟨path-pattern⟩

    prints a big concatenation of the contents of all files matching ⟨path-pattern⟩.

13. `head -n`⟨num⟩ ⟨path-pattern⟩

    prints the first ⟨num⟩ lines of each file matching ⟨path-pattern⟩.

14. `tail -n`⟨num⟩ ⟨path-pattern⟩

    prints the last ⟨num⟩ lines of each file matching ⟨path-pattern⟩.

15. `sed -n` ⟨num1⟩`,`⟨num2⟩`p` ⟨path-pattern⟩

    prints lines ⟨num1⟩ through ⟨num2⟩ of each file matching ⟨path-pattern⟩.

16. `sort` ⟨path-pattern⟩

    prints a sorted list of all lines of all files matching ⟨path-pattern⟩.

3.4 Unix commands for text processing using regular expressions

These commands use regular expressions to match lines in text files, to print or substitute:

17. `egrep '⟨reg-exp⟩' ⟨path-pattern⟩`

    prints lines in file(s) ⟨path-pattern⟩ that match ⟨reg-exp⟩.

18. `egrep -o '⟨reg-exp⟩' ⟨path-pattern⟩`

    prints *strings* in file(s) ⟨path-pattern⟩ that match ⟨reg-exp⟩.

19. `grep '⟨reg-exp⟩' ⟨path-pattern⟩`

    same as egrep, but weaker regexp support.

20. `perl -pe 's/⟨reg-exp⟩/⟨string⟩/g' ⟨path-pattern⟩`

    prints contents of file(s) ⟨path-pattern⟩ with:

    - every ⟨reg-exp⟩ replaced with ⟨string⟩, and
    - any '\⟨num⟩' in ⟨string⟩ replaced with contents of the ⟨num⟩th parens in ⟨reg-exp⟩

    (the global search option `g` allows multiple matches per line – this can be omitted).

    For example:

    ```
    perl -pe 's/semprini/CENSORED/g' myfile.txt
    ```

    prints version of myfile.txt with all occurrences of 'semprini' censored out, and:

    ```
    perl -pe 's/item ([0-9]+)/the \1th item/g' myfile.txt
    ```

    prints version of myfile.txt w. cardinal items ('*item 12*') as ordinals ('*the 12th item*').

21. `sed 's/⟨reg-exp⟩/⟨string⟩/g' ⟨path-pattern⟩`

    works the same as 'perl -pe', but weaker regexp support.

3.5 Unix commands not for navigation, using no regular expressions

These commands don't use any regexps, but are still useful:

22. `uniq -c ⟨path⟩`

    given sorted lines in ⟨path⟩, prints each unique line preceded by number of occurrences.

23. `echo ⟨string⟩`

    prints ⟨string⟩. This is useful for reporting progress in unix scripts.

    For example:

    ```
    echo 'Here is some text!'
    ```

    echoes back:

```
Here is some text!
```

3.6 Chaining commands

Commands can be chained together by piping/redirecting input and output:

1. Commands `cat`, `head`, `tail`, `sort`, `egrep`, `perl`, `uniq`, `echo`, `curl` write output.
2. Commands `head`, `tail`, `sort`, `egrep`, `perl`, `uniq` read from piped/redirected input.
3. Commands writing output can **redirect** (or 'pipe') output to commands reading input (using '`|`' pipes and leaving off the path argument from commands following pipes):

   ```
   cat file.txt | sort | uniq -c
   ```

4. Commands writing output can also redirect their output to files, using '`>`':

   ```
   echo 'Here is some text!' > myfile.txt
   ```

   (This is an easy way to make a text file.)

Practice:

In one line, print an alphabetized list of all capitalized words in some file 'myfile.txt'

3.7 Makefiles

Chained commands can be generalized into 'Makefiles,' to automate projects/experiments:

Makefiles organize unix commands to:

- record how to obtain output/target files ('results') from input/source files ('data'),
- generalize these as processes from file types to file types (e.g. '.' extensions),
- figure out what's out of date and needs re-computing, using process dependencies,

essentially an artificial-intelligence production system, it figures out how to make things for you!

Makefiles contain rules for making output/target files from input/source files, of the form:

```
⟨target-path⟩ : ⟨list-of-source-paths⟩
tab ⟨chained-unix-command⟩
tab ⟨chained-unix-command⟩
⋮
```

For example, if you create a file called '`Makefile`' (e.g. using TextEdit or Notepad) containing:

```
samples.txt: sample.txt
tab cat sample.txt sample.txt > samples.txt
```

(and you create a file `sample.txt` at that same directory, containing whatever you want)
then you can create samples.txt by typing '`make samples.txt`' in the terminal at that directory.

The target path may contain wildcard '`%`' to match a substring and copy it in the source paths (in which case the rule is called an 'implicit rule').
Files created like this are then deleted, unless '`.PRECIOUS:` ⟨target-path⟩' precedes item.

The chained unix commands may contain the following variables (to allow '%' paths):

1. `$@` — the target path (with '`%`' wildcard instantiated with a string)

   For example:

   ```
   samps-uniq%.txt: sample.txt
   tab cat sample.txt sample.txt > $@
   ```

   (here, `make samps-uniqA.txt` and `make samps-uniqB.txt` files are identical).

2. `$^` — the list of source paths (with '`%`' wildcard instantiated with a string)

   For example:

   ```
   %.combined.model: %.pcfg.model %.pos.model
   tab cat $^ > $@
   ```

3. `$<` — the first source path (with '`%`' wildcard instantiated with a string)

   For example:

   ```
   %.txt: %.html scripts/remove-html.pl
   tab cat $< | perl scripts/remove-html.pl > $@
   ```

4. `$*` — the string instantiating '`%`' in an implicit rule

   For example:

   ```
   %.wikipedia.html:
   tab curl https://en.wikipedia.org/wiki/$* > $@
   ```

Practice:

Write a Makefile item to make a '%.capwords' file, containing an alphabetical list of all capitalized words in a source '%.txt'?

3.8 Advanced Makefile scripts (if there's time)

The target, source, and commands may also contain user variables, defined prior to the item:

5. set user variable: ⟨user-var⟩ `=` ⟨string⟩

   For example:

   ```
   SWAMP = Frog Snail
   ```

6. invoke user variable: `$(`⟨user-var⟩`)`

For example:

```
Swamp: $(SWAMP)
tab cat $^ > $@
```

The chained unix commands may also contain macros:

(may also appear among the source paths if ‘`.SECONDEXPANSION:`’ precedes item, in which case all dollar signs must be ‘escaped’ with an additional dollar sign: `$$`)

7. `$(word` ⟨num⟩`,` ⟨string⟩ `)`

   the ⟨num⟩-th word in the ⟨string⟩, delimited by spaces

   For example:

   ```
   %.txt: % scripts/remove-html.pl
   tab cat $(word 1,$^) | perl $(word 2,$^) > $@
   ```

   cats the first source (the html file) into the second source (the `.pl` script)

8. `$(suffix` ⟨string⟩ `)`

   the part of a string containing the last dot + everything after (‘extension’ of a filename)

9. `$(basename` ⟨string⟩ `)`

   the part of a string prior to the last dot (i.e. the part of a filename without the extension)

10. `$(subst` ⟨string1⟩`,` ⟨string2⟩`,` ⟨string3⟩ `)`

    a copy of ⟨string3⟩ with each instance of ⟨string1⟩ replaced with ⟨string2⟩

    For example, suffix, basename and subst can define a general reproducible process:

    ```
    %.parses: $$(basename %).sents parser $$(subst .,,$$(suffix %)).model
    tab cat $(word 1,$^) | $(word 2,$^) $(word 3,$^) > $@
    ```

    so, given any test set (e.g. `testset.sents`), trained model (`trainingset.model`),

    `make testset.trainingset.parses`

    will produce a file of hypothesized parse trees that identifies the model and test set.

11. `$(wildcard` ⟨path-pattern⟩ `)`

    a list of every file in the current directory matching ⟨path-pattern⟩

    For example:

    ```
    WSJSECTS = $(wildcard Corpora/penn_treebank_3/parsed/mrg/wsj/*)
    ```

generates a list of all the subdirectories in `/Corpora/.../mrg/wsj`

12. `$(foreach` ⟨varname⟩`,` ⟨string1⟩`,` ⟨string2⟩ `)`

    a list of copies of ⟨string2⟩, replacing '`$(`⟨varname⟩`)`' with each word in ⟨string1⟩

    For example:

    ```
    WSJTR = 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21
    wsjTRAIN.linetrees:  $(foreach sect,$(WSJTR),wsj$(sect).linetrees)
    tab cat $^ > $@
    ```

    concatenates files `wsj02.linetrees`, `wsj03.linetrees`, etc.

13. `$(patsubst` ⟨%-pattern1⟩`,` ⟨%-pattern2⟩`,` ⟨string⟩ `)`

    a copy of ⟨string⟩ with each instance of ⟨%-pattern1⟩ replaced with ⟨%-pattern2⟩

    For example:

    ```
    OUTPUTS = $(patsubst %.in,%.out,$(wildcard *.in))
    ```

14. `$(shell` ⟨command⟩ `)` or, for short: `` ` `` ⟨command⟩ `` ` ``

    output (w/o newlines) of executing ⟨command⟩ at unix prompt in current directory

    For example:

    ```
    CFLAGS = $(shell cat user-cflags.txt)
    CFLAGS = `cat user-cflags.txt`
    ```