Programming for Evolutionary Biology
March 21 – April 6 2014
Leipzig, Germany

Introduction to Unix systems Part 4 – awk & make

Giovanni Marco Dall'Olio
King's College of London, UK

Schedule

- 9.30 – 10.30: “What is Unix?”
- 11.00 – 12.00: Introducing the terminal
- 13:30 – 16:30: Grep, & Unix philosophy
- **17:00 – 18:00: awk, make, and question time**

awk

- awk is the Unix command to work with tabular data

awk

- Things you can do with awk:
  - Extract all the lines of a file that match a pattern, and print only some of the columns (instead of “grep | cut”)
  - Add a prefix/suffix to all the element of a column (instead of “cut | paste”)
  - Sum values of different columns

awk and gawk

- Today we will use the GNU version of awk, called gawk
- In most systems, awk and gawk are the same

Awk exercise

- Let's go to the “bed” directory
- cd .homes/evopserver/lectures/unix_intro/bed

The BED format

- The BED format is used to store annotations on genomic coordinates
- Example:
    - Annotate gene position
    - Annotate Trascription Factor binding sites
    - Annotate SNP genotypes
    - Annotate Gene Expression

Example awk usage

- Select only the lines matching chromosome 7, and print only the second column, summing 100 to it
  - awk '$1 ~ chr7 {print $2+100}' annotations.bed

Example of BED file

Columns in a BED file:

- **Chromosome**
- **Start**
- **End**
- **Label**
- **Score**
- **Strand** (+ or -)
- ….

```
chr7    127471196  127472363  gene1  0    +
chr7    127472363  127473530  gene2  2    +
chr7    127473530  127474697  gene3  20   +
chr7    127474697  127475864  gene4  3    +
chr7    127475864  127477031  gene5  100  -
chr7    127477031  127478198  gene6  3    -
chr7    127478198  127479365  gene7  5    -
chr7    127479365  127480532  gene8  1    +
chr7    127480532  127481699  gene9  3    -
```

Basic awk usage

- awk '<pattern to select lines> {instructions to be executed on each line}'

Selecting columns in awk

- In awk, each column can be accessed by $<column-number>
- For example,
    - $1 → first column of the file
    - $2 → second column
    - $NF → last column
- $0 matches all the columns of the file

Printing columns in awk

- The basic awk usage is to print specific columns
- The syntax is the following:
  - awk '{print $1, $2, $3}' annotations.bed → prints the first three columns

Printing columns in awk

- The basic awk usage is to print specific columns
- The syntax is the following:
  - `awk '{print $1, $2, $3}' annotations.bed` → prints the first three columns
  - `awk '{print $0}' annotations.bed` → print all the columns

Adding a prefix to a column with awk

- A common awk usage is to add a prefix or suffix to all the entries of a column
- Example:
    - awk '{print $2 “my_prefix”$2}' annotations.bed

Summing columns in awk

- If two columns contain numeric values, we can use awk to sum them
- Usage:
  - `awk '{print $2 + $3}' annotations.bed` → sums columns 2 and 3

Filter lines and select columns with awk

- Awk can apply a filter and print only the lines matching a pattern
  - Like grep, but more powerful
- Print all the lines that have “chr7” in their first column:
  - awk '$1 ~ “chr7” {print $0}' myfile.txt

Advanced: redirecting to files

- The following will split the contents of annotations.bed into two files, according to the 1st column:
- awk '{print $0 > $1".file"}' annotations.bed

Advanced: print something before reading the file

- The BEGIN statement can be used to execute commands before reading the file:
- awk 'BEGIN {print "position"} {print $2}' annotations.bed

More on awk

- awk is a complete programming language
- It is the equivalent of a spreadsheet for the command line
- If you want to know more, check the book “Gawk effective AWK Programming” at http://www.gnu.org/software/gawk/manual

A streaming file editor: sed

- sed is a command-line tool to edit files, without opening their full contents in memory
- Things you can achieve with sed:
  - Find&Replace words in huge text files
  - Remove determinate lines from a file

sed – basic usage

- The following will replace all the occurrences of the word 'gene' with 'GENE' in the BED file:
  - sed 's/gene/GENE/' annotations.bed
- Explanation:
  - sed → name of the command
  - 's/gene/GENE/' → substitute (s) the word 'gene' (first pair of slashes) with the word 'GENE' (second pair of slashes)

Sed – saving results to file

- The previous command printed the result of the substitution, but did not save it to a file
- If you open the file annotations.bed, it has not been changed
- To make your changes permanents, you have two options:
  - Output the result to a file:
    ```
    sed 's/gene/GENE/' annotations.bed > annotations_changed.bed
    ```
  - Use the -i option:
    ```
    sed -i 's/gene/GENE/' annotations.bed
    ```

Other sed options: removing a pattern of lines

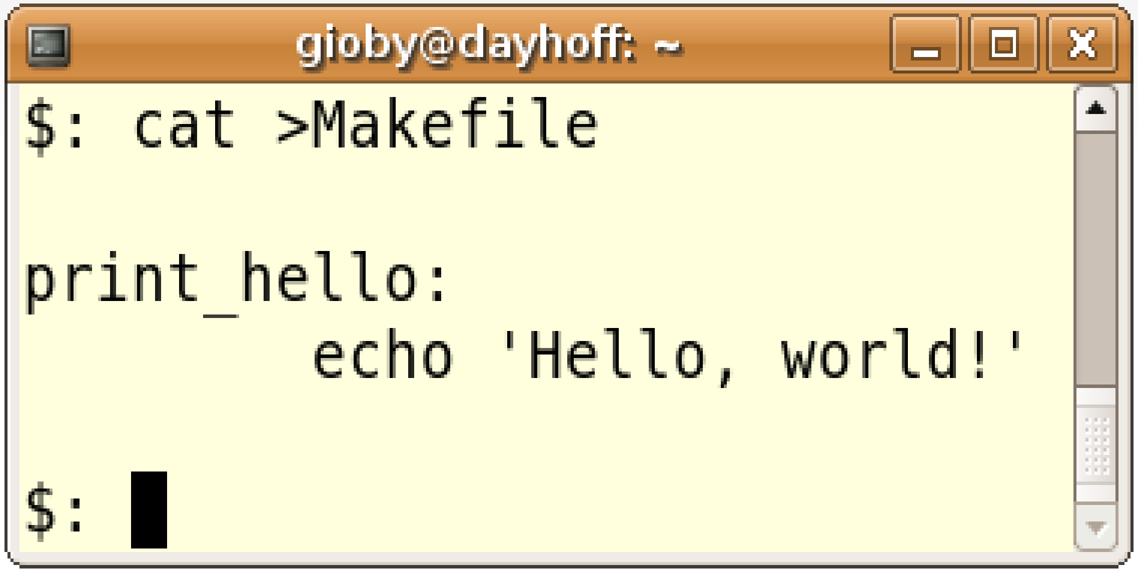
- Let's remove all the 1,3,5,7, etc.. lines from the bed file:
    - sed -n '1~2d' annotations.bed

GNU/make

- make is a tool used by programmer to define how software should be compiled and installed
- It is also used as a way to store a set of commands, to recall them later

Simplest Makefile example

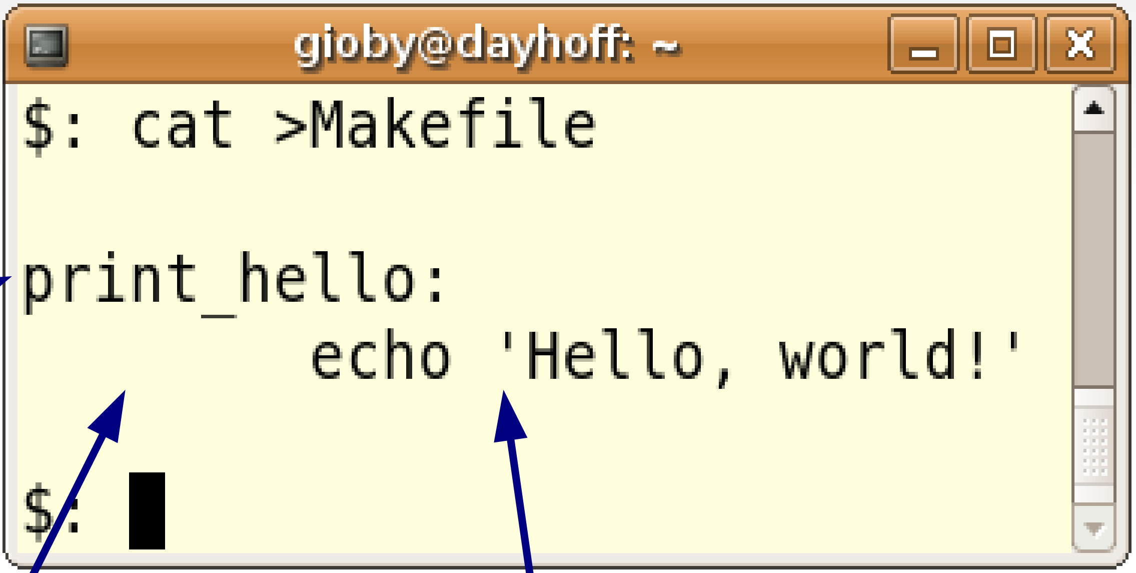
- The simplest Makefile contains just the name of a task and the commands associated with it:



```
$: cat >Makefile

print_hello:
        echo 'Hello, world!'

$: 
```

- print_hello is a makefile 'rule': it stores the commands needed to say 'Hello, world!' to the screen.

Simplest Makefile example



```
$: cat >Makefile

print_hello:
	echo 'Hello, world!'

$: 
```

Target of the rule

This is a tabulation (not 8 spaces)

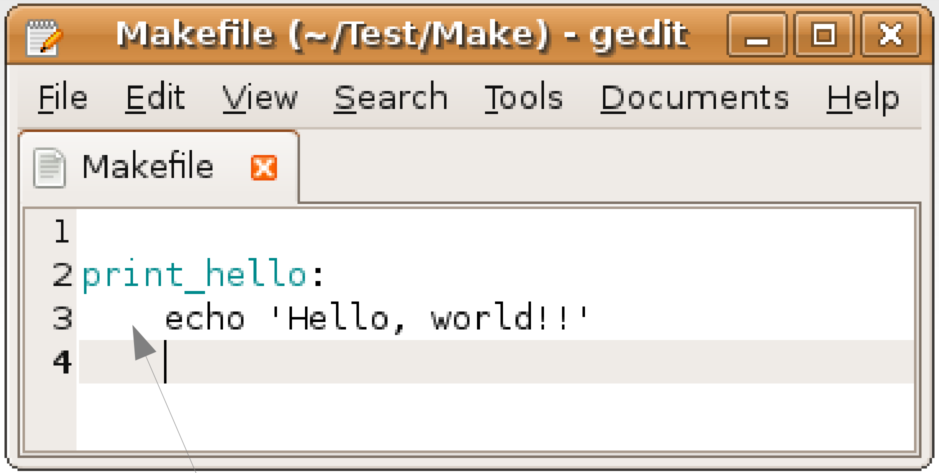
Commands associated with the rule

Makefile rule

Simplest Makefile example

- Create a file in your computer and save it as 'Makefile'.
- Write these instructions in it:

```
print_hello:
	echo 'Hello, world!!'
```


Makefile (~/Test/Make) - gedit
File Edit View Search Tools Documents Help
Makefile

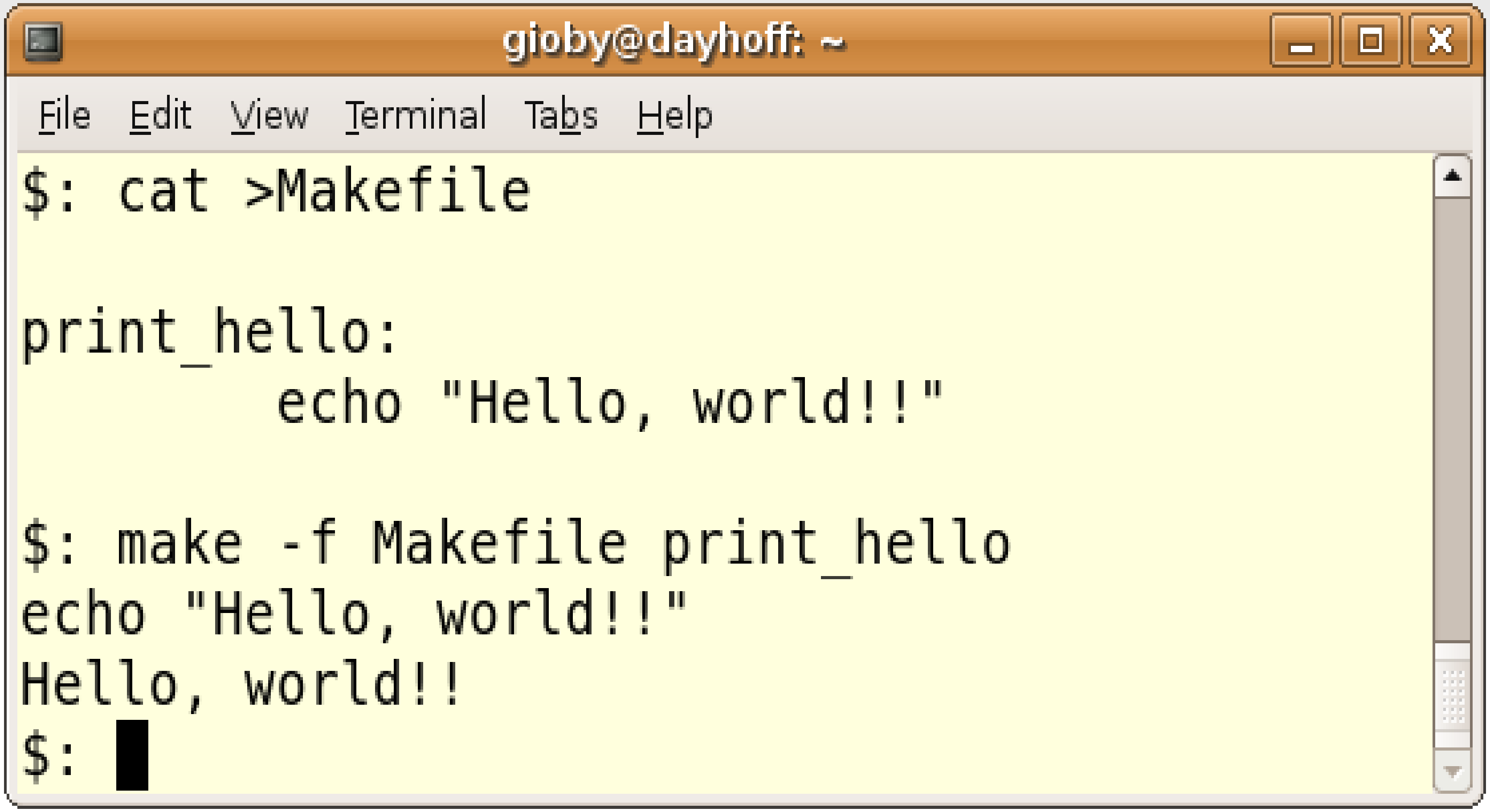
```
print_hello:
	echo 'Hello, world!!'
```

This is a tabulation (<Tab> key)

- Then, open a terminal and type:

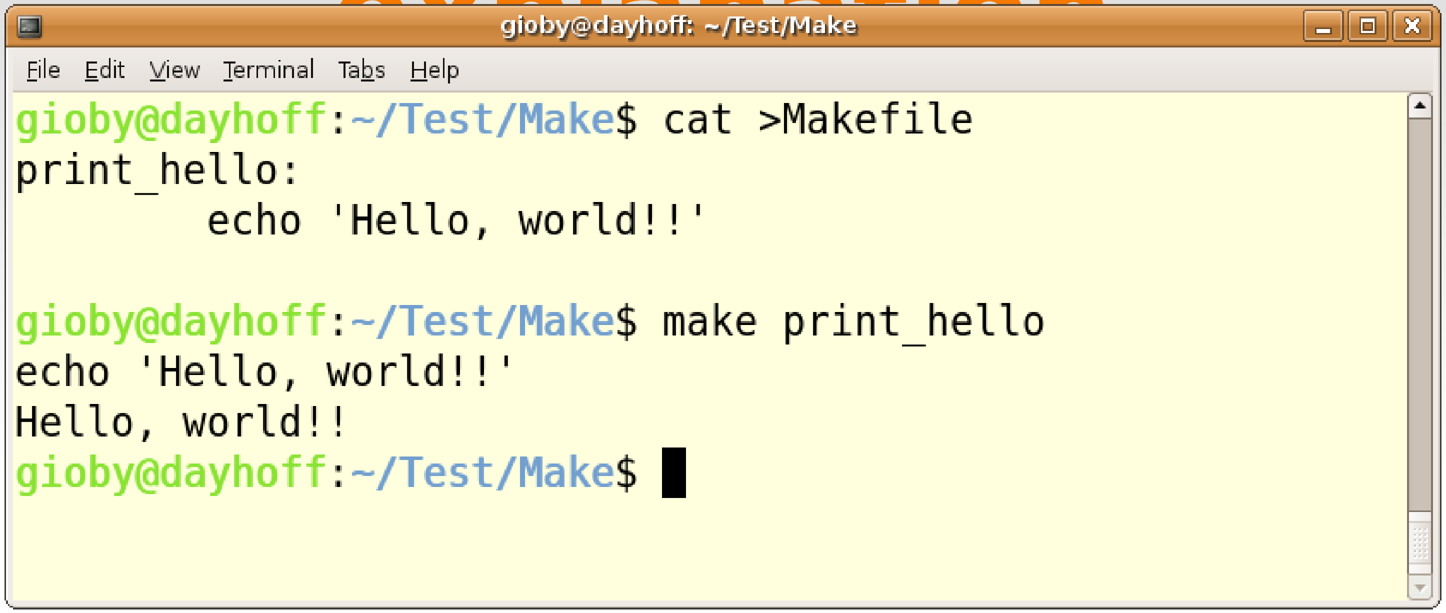
```
make -f Makefile print_hello
```

Simplest Makefile example



```
$: cat >Makefile

print_hello:
        echo "Hello, world!!"

$: make -f Makefile print_hello
echo "Hello, world!!"
Hello, world!!
$: 
```

Simplest Makefile example – explanation

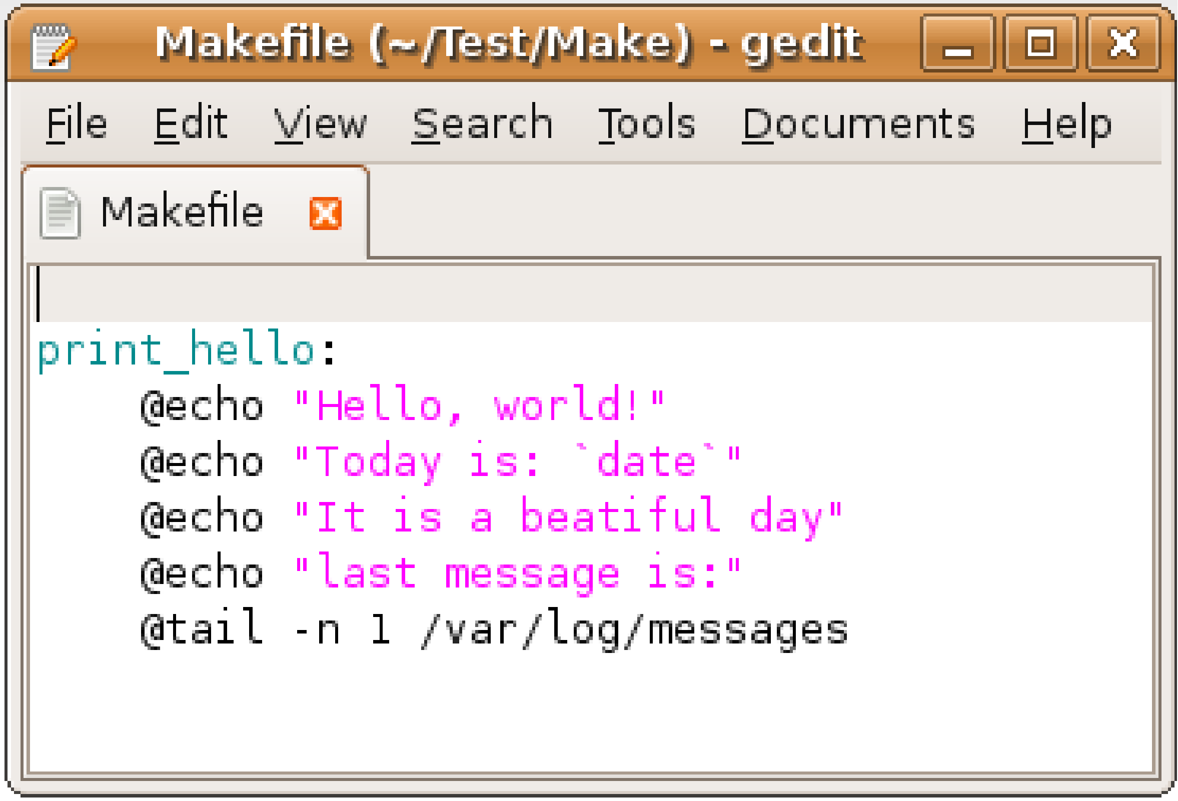


```
gioby@dayhoff:~/Test/Make$ cat >Makefile
print_hello:
        echo 'Hello, world!!'

gioby@dayhoff:~/Test/Make$ make print_hello
echo 'Hello, world!!'
Hello, world!!
gioby@dayhoff:~/Test/Make$ 
```

- When invoked, the program 'make' looks for a file in the current directory called 'Makefile'
- When we type 'make print_hello', it executes any procedure (target) called 'print_hello' in the makefile
- It then shows the commands executed and their output

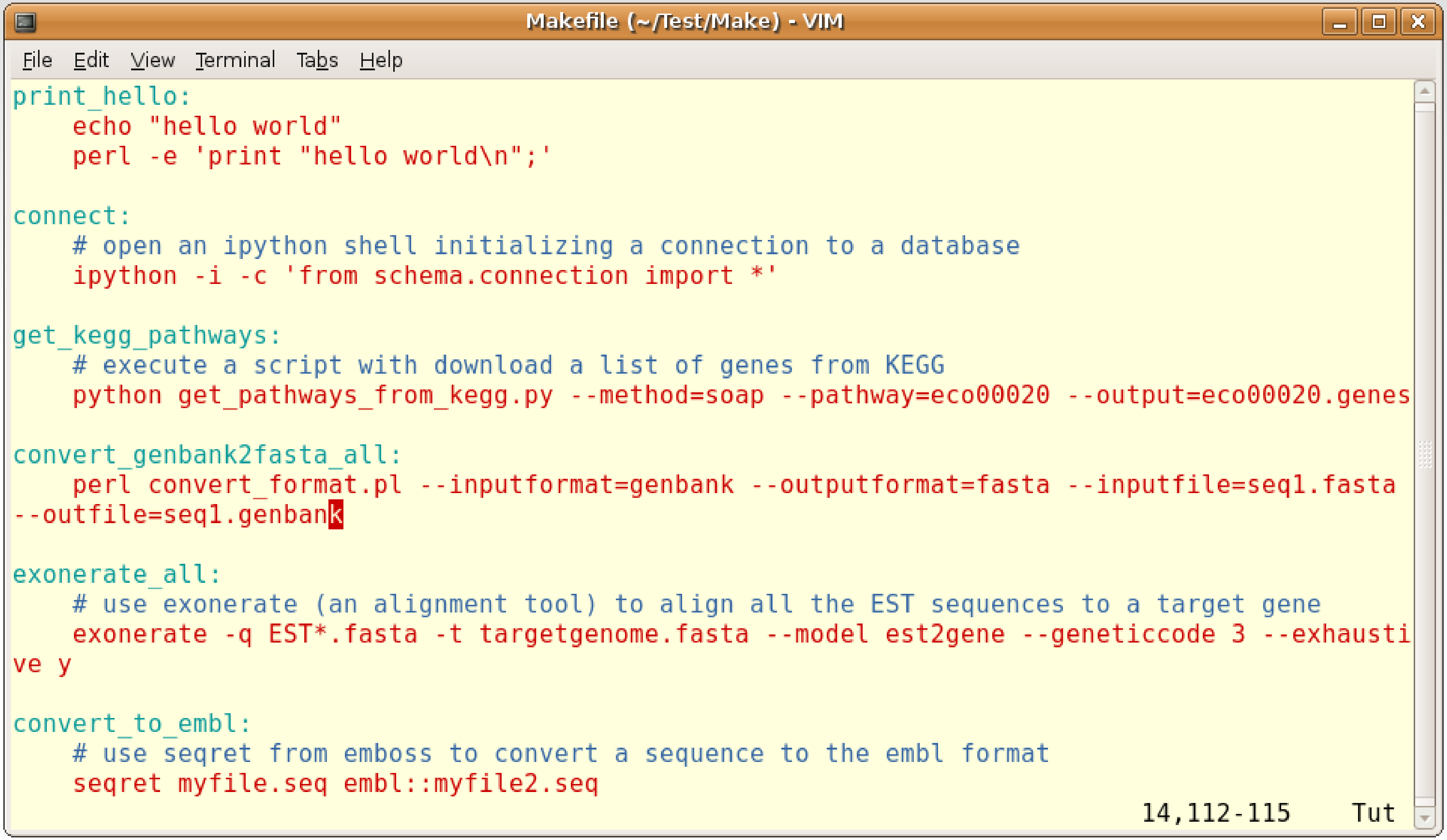
A sligthly longer example

- You can add as many commands you like to a rule
- For example, this 'print_hello' rule contains 5 commands
- Note: ignore the '@' thing, it is only to disable verbose mode (explained later)


Makefile (~/Test/Make) - gedit
File Edit View Search Tools Documents Help
Makefile

```
print_hello:
	@echo "Hello, world!"
	@echo "Today is: `date`"
	@echo "It is a beatiful day"
	@echo "last message is:"
	@tail -n 1 /var/log/messages
```

A more complex example

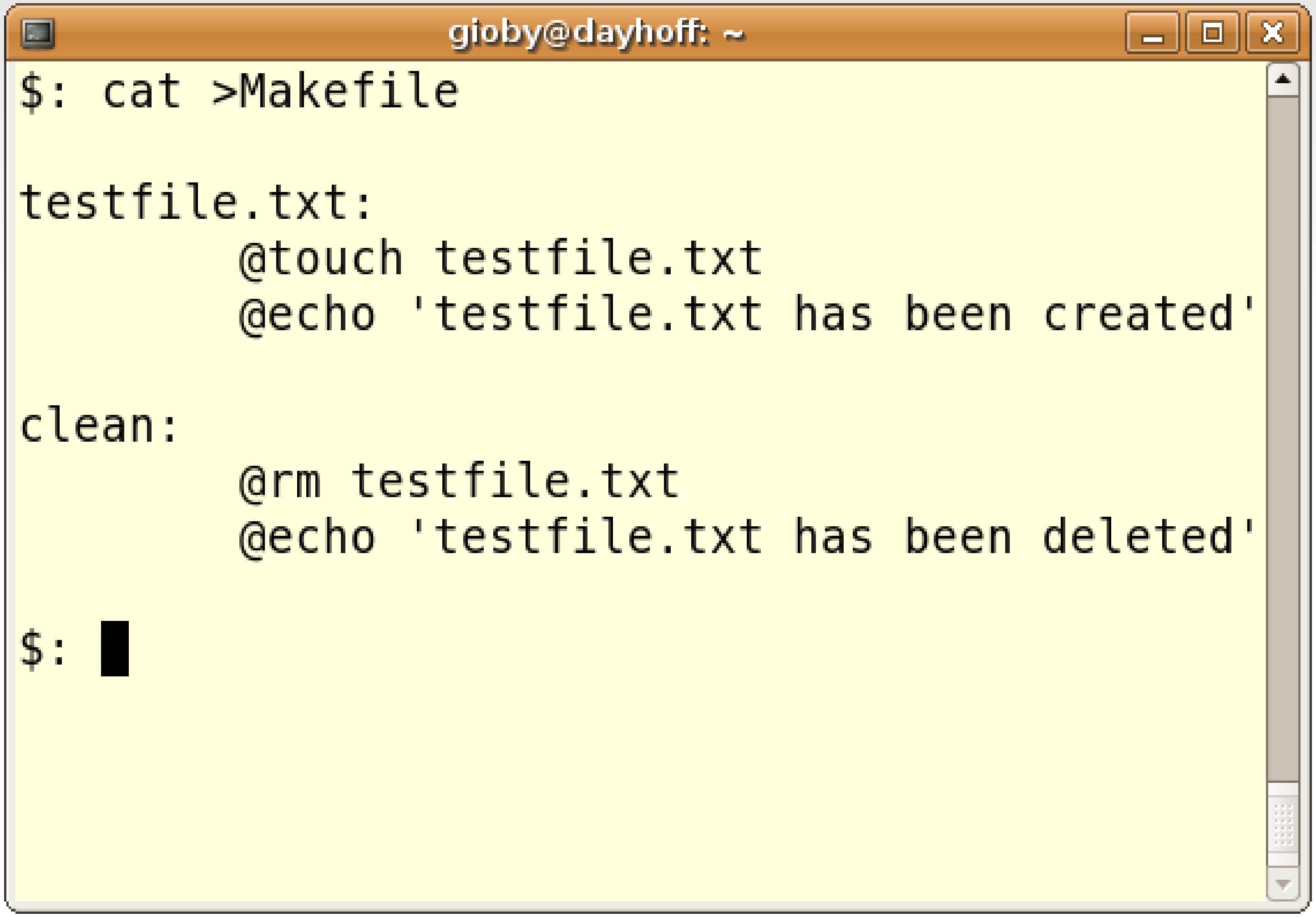

Makefile (~/Test/Make) - VIM
File Edit View Terminal Tabs Help

```
print_hello:
    echo "hello world"
    perl -e 'print "hello world\n";'

connect:
    # open an ipython shell initializing a connection to a database
    ipython -i -c 'from schema.connection import *'

get_kegg_pathways:
    # execute a script with download a list of genes from KEGG
    python get_pathways_from_kegg.py --method=soap --pathway=eco00020 --output=eco00020.genes

convert_genbank2fasta_all:
    perl convert_format.pl --inputformat=genbank --outputformat=fasta --inputfile=seq1.fasta
--outfile=seq1.genbank

exonerate_all:
    # use exonerate (an alignment tool) to align all the EST sequences to a target gene
    exonerate -q EST*.fasta -t targetgenome.fasta --model est2gene --geneticcode 3 --exhausti
ve y

convert_to_embl:
    # use seqret from emboss to convert a sequence to the embl format
    seqret myfile.seq embl::myfile2.seq
                                                                             14,112-115    Tut
```

The target syntax

- The target of a rule can be either a title for the task, or a file name.
- Every time you call a make rule (example: 'make all'), the program looks for a file called like the target name (e.g. 'all', 'clean', 'inputdata.txt', 'results.txt')
- The rule is executed only if that file doesn't exists.

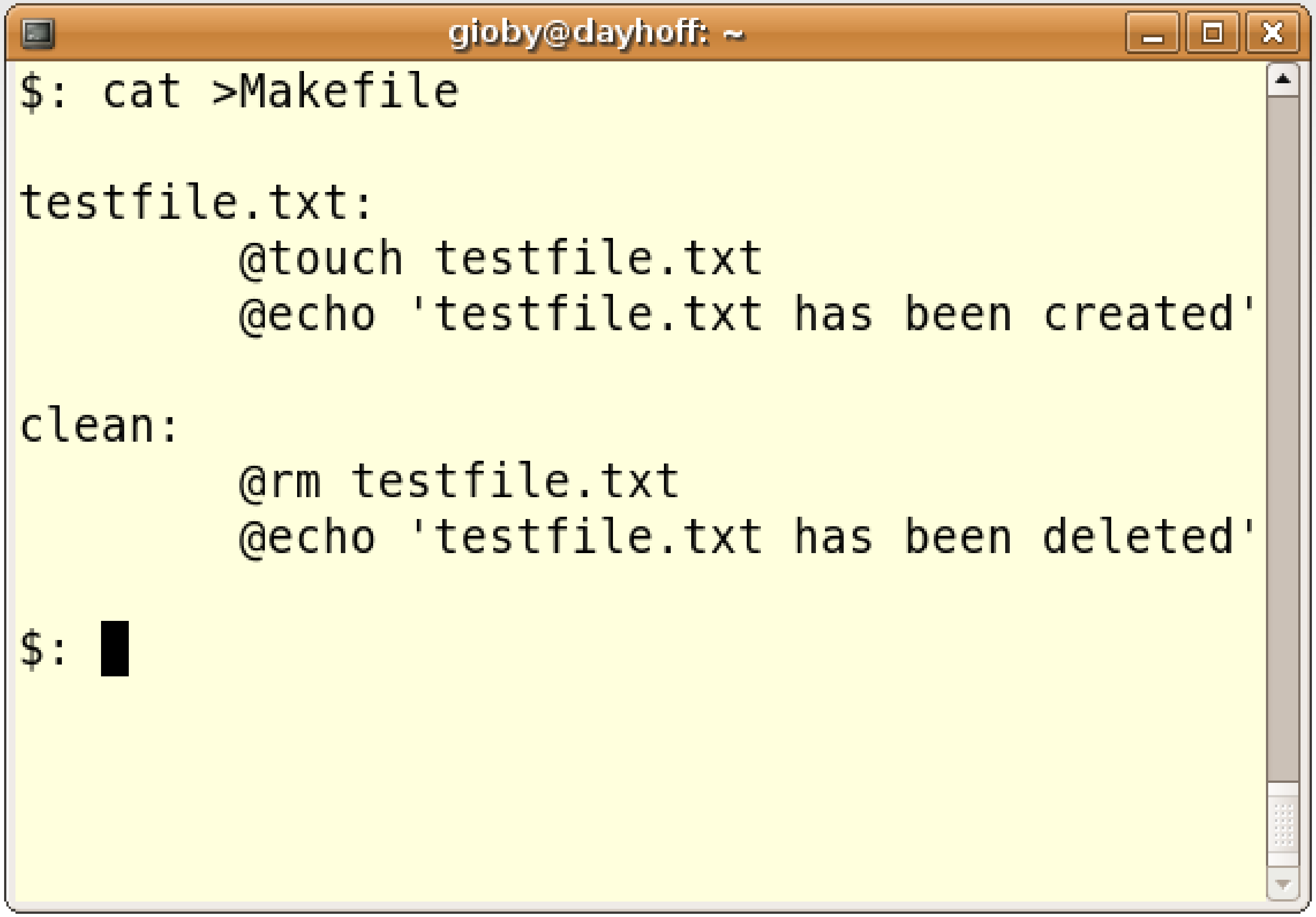
Filename as target names



```
$: cat >Makefile

testfile.txt:
        @touch testfile.txt
        @echo 'testfile.txt has been created'

clean:
        @rm testfile.txt
        @echo 'testfile.txt has been deleted'

$:
```

- In this makefile, we have two rules: 'testfile.txt' and 'clean'

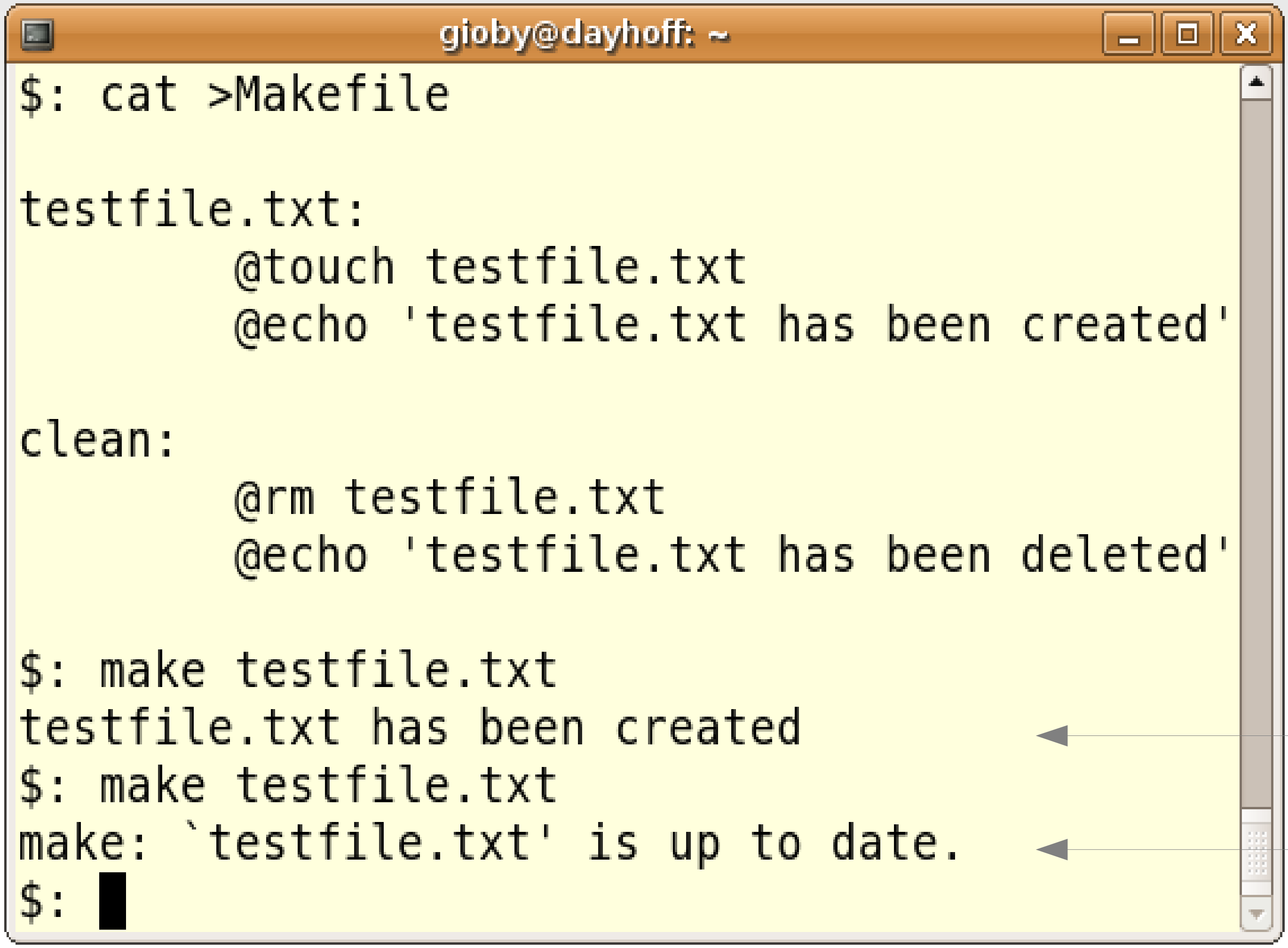
Filename as target names



```
$: cat >Makefile

testfile.txt:
	@touch testfile.txt
	@echo 'testfile.txt has been created'

clean:
	@rm testfile.txt
	@echo 'testfile.txt has been deleted'

$: 
```

- In this makefile, we have two rules: 'testfile.txt' and 'clean'
- When we call 'make testfile.txt', make checks if a file called 'testfile.txt' already exists.

Filename as target names



```
$: cat >Makefile

testfile.txt:
	@touch testfile.txt
	@echo 'testfile.txt has been created'

clean:
	@rm testfile.txt
	@echo 'testfile.txt has been deleted'

$: make testfile.txt
testfile.txt has been created
$: make testfile.txt
make: `testfile.txt' is up to date.
$:
```

The commands associated with the rule 'testfile.txt' are executed only if that file doesn't exists already

Real Makefile-rule syntax

- Complete syntax for a Makefile rule:

```
<target>: <list of prerequisites>
     <commands associated to the rule>
```

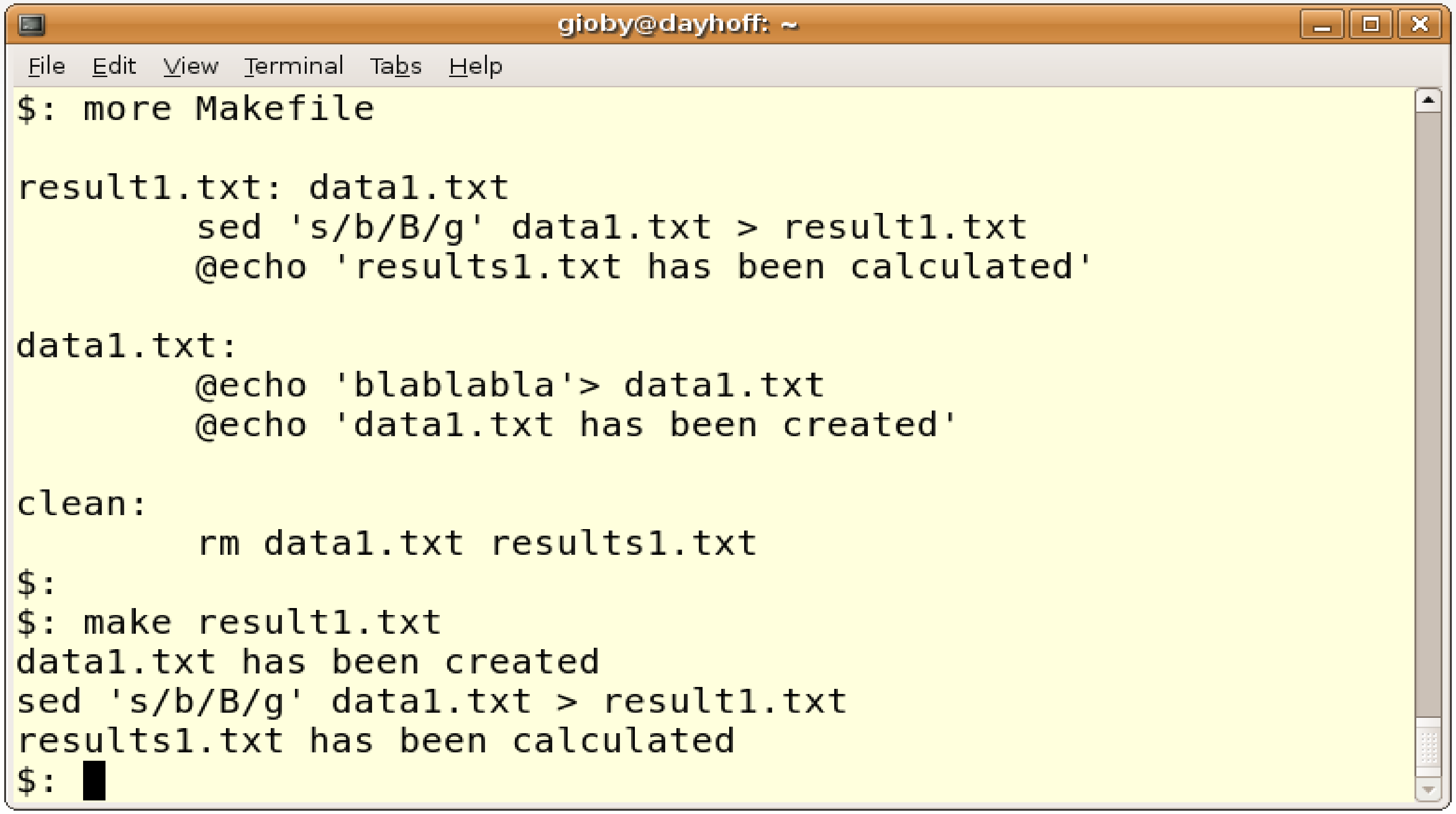
- Example:

```
result1.txt: data1.txt data2.txt
     cat data1.txt data2.txt > result1.txt
     @echo 'result1.txt' has been calculated'
```

- Prerequisites are files (or rules) that need to exists already in order to create the target file.
- If 'data1.txt' and 'data2.txt' don't exist, the rule 'result1.txt' will exit with an error (no rule to create them)

Piping Makefile rules together

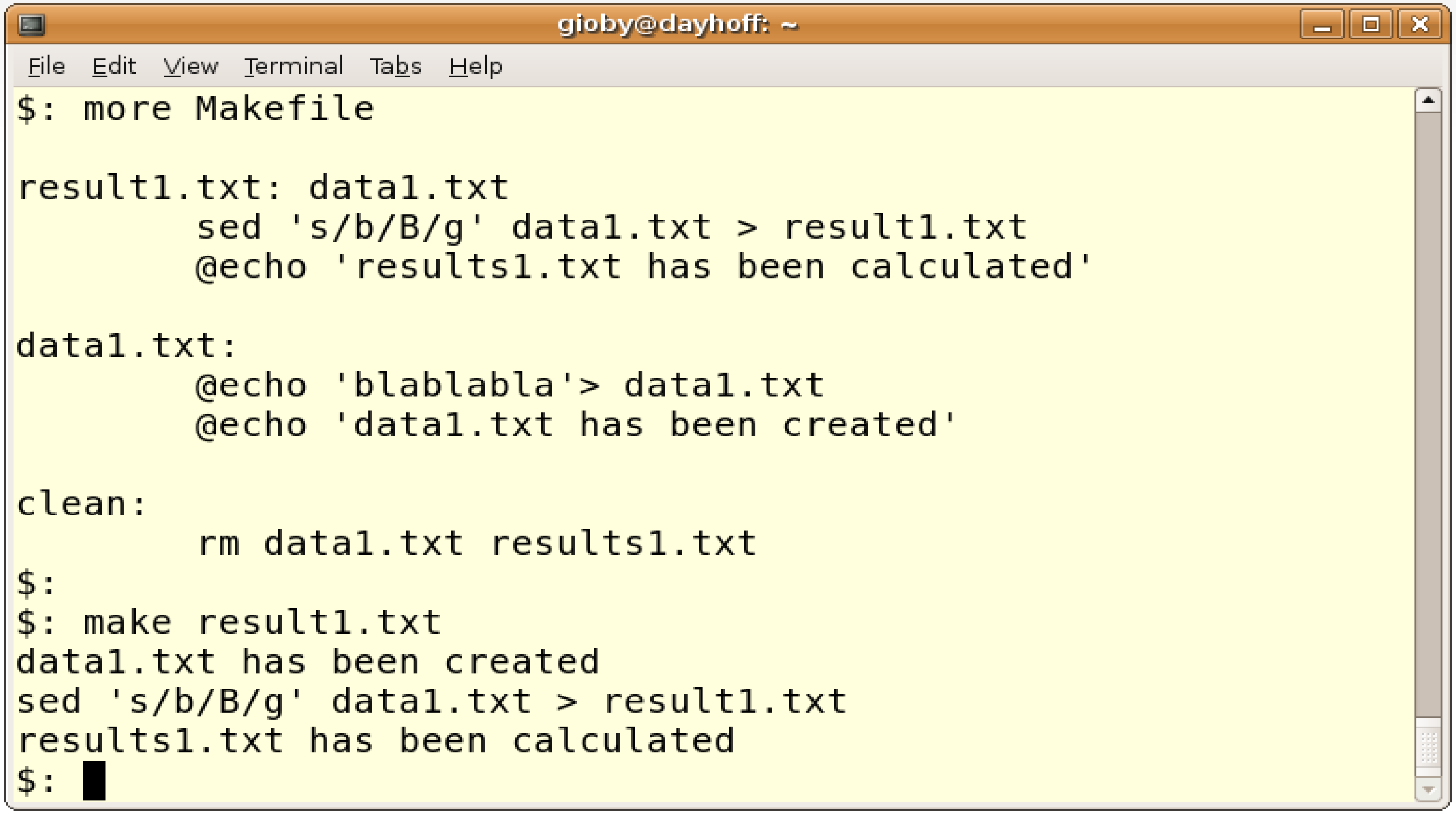
- You can pipe two Makefile rules together by defining prerequisites



```
$: more Makefile

result1.txt: data1.txt
        sed 's/b/B/g' data1.txt > result1.txt
        @echo 'results1.txt has been calculated'

data1.txt:
        @echo 'blablabla'> data1.txt
        @echo 'data1.txt has been created'

clean:
        rm data1.txt results1.txt
$:
$: make result1.txt
data1.txt has been created
sed 's/b/B/g' data1.txt > result1.txt
results1.txt has been calculated
$: 
```

Piping Makefile rules together

- The rule 'result1.txt' depends on the rule 'data1.txt', which should be executed first



```
$: more Makefile

result1.txt: data1.txt
        sed 's/b/B/g' data1.txt > result1.txt
        @echo 'results1.txt has been calculated'

data1.txt:
        @echo 'blablabla'> data1.txt
        @echo 'data1.txt has been created'

clean:
        rm data1.txt results1.txt
$:
$: make result1.txt
data1.txt has been created
sed 's/b/B/g' data1.txt > result1.txt
results1.txt has been calculated
$: 
```

Conditional execution by modification date

- We have seen how make can be used to create a file, if it doesn't exists.

```
file.txt:
    # if file.txt doesn't exists, then create it:
    echo 'contents of file.txt' > file.txt
```

- We can do better: create or update a file only if it is newer than its prerequisites

Conditional execution by modification date

- Let's have a better look at this example:

```
result1.txt: data1.txt calculate_result.py
    python calculate_result.txt --input
data1.txt
```

- A great feature of make is that it execute a rule not only if the target file doesn't exist, but also if it has a 'last modification date' earlier than all of its prerequisites

Conditional execution by modification date

```
result1.txt: data1.txt
	@sed 's/b/B/i' data1.txt > result1.txt
	@echo 'result1.txt has been calculated'
```

- In this example, result1.txt will be recalculated every time 'data1.txt' is modified

```
$: touch data1.txt calculate_result.py

$: make result1.txt
result1.txt has been calculated

$: make result1.txt
result1.txt is already up-to-date

$: touch data1.txt
$: make result1.txt
result1.txt has been calculated
```

Make - advantages

- Make allows you to save shell commands along with their parameters and re-execute them;
- It allows you to use command-line tools which are more flexible;
- Combined with a revision control software, it makes possible to reproduce all the operations made to your data;

Resume of the day

- Unix → an operating system from the '70s, which was successful for its philosophy and technical novelties
- The terminal → a way to execute commands by typing
- How to get help → man, info, --help, internet
- Grep, awk → search patterns in a file
- Other Unix tools → each specialized on a single data manipulation task