Introduction to Linux for Bioinformatics

Working the command line

Joachim Jacob
5 and 12 May 2014

Short recapitulation of last week

```
joachim@joachim-VirtualBox ~ $ bowtie --version
bowtie version 0.12.7
64-bit
Built on allspice
Thu May  5 12:19:01 UTC 2011
Compiler: gcc version 4.6.1 20110503 (prerelease) (Ubuntu 4.6.0-6ubuntu2)
Options: -O3   -Wl,--hash-style=both -g -O2  -g -O2
Sizeof {int, long, long long, void*, size_t, off_t}: {4, 8, 8, 8, 8, 8}
joachim@joachim-VirtualBox ~ $ which bowtie
/usr/bin/bowtie
```

Bash only looks at certain directories for commands/software/programs ...

The location of a tool you can find with 'which'.

We can install and run software

- E.g. commands for mapping NGS data on the wiki: http://wiki.bits.vib.be/index.php/GenomeView_Workshop:_Mapping_excercises#Mapping

Mapping

Mapping is the name of aligning each read to the genome sequence. The purpose is to align the read to

```
./bwa aln genome.fasta transcript.fastq > transcript.sai
```

This command will use BWA to map all reads in *transcript.fastq* to the genome (more specifically, the seq

Linux commandline explanation: the '>' in above command means that the output of the program is writte

we know what parameters we can/have to pass on: just type *./bwa aln* to see the possibilities. You can re

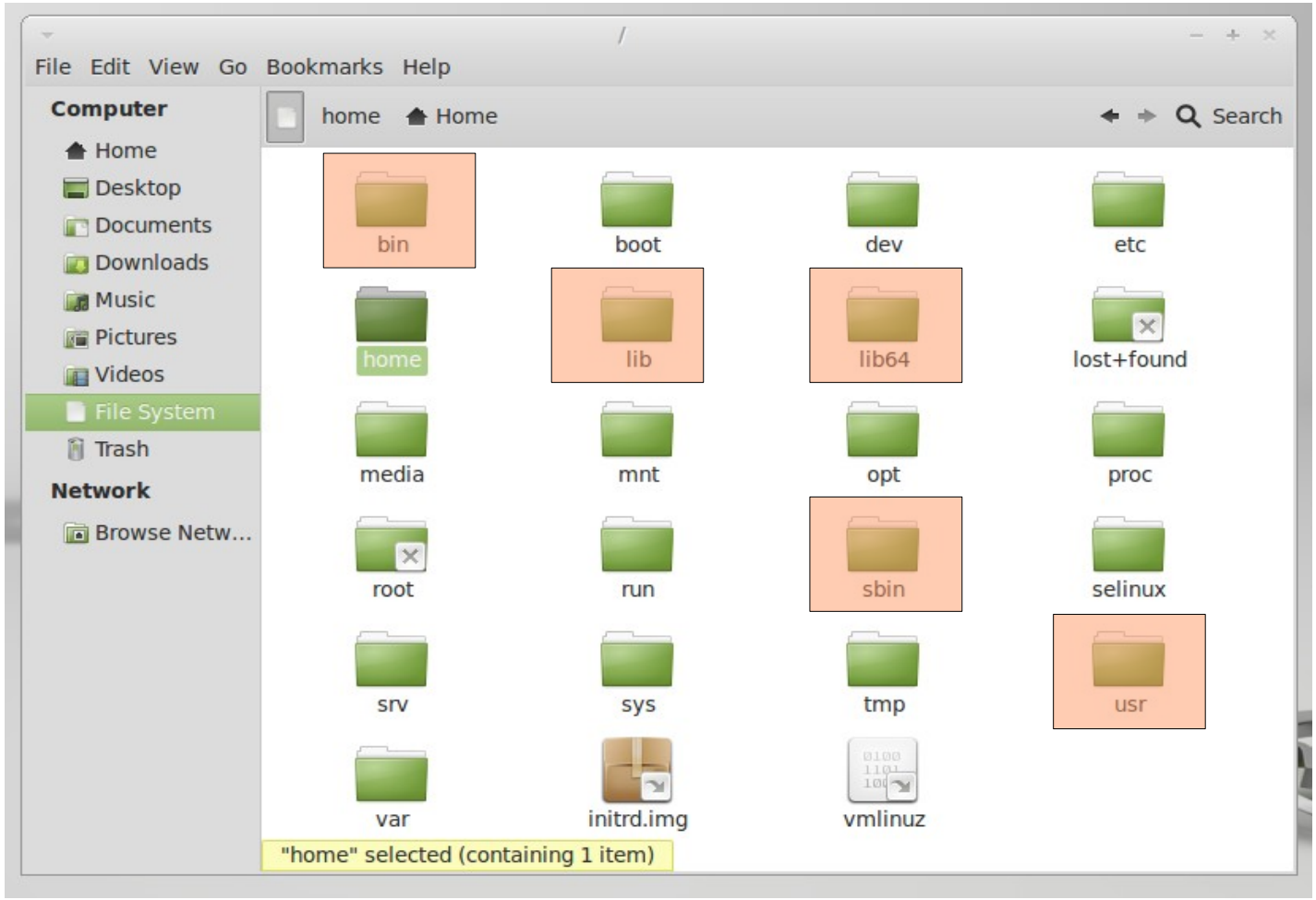
The next few steps reformat the data so that it can actually be used in further analyses

```
./bwa samse genome.fasta transcript.sai transcript.fastq > transcript.sam
```

Transforms the native file format of BWA to the general purpose format SAM.

```
./samtools view -bS -t genome.fasta.fai transcript.sam -o transcript.bam
```

Software installation directories



- Contain the commands we can execute in the terminal

Software installation directories

```
joachim@joachim-VirtualBox ~ $ bowtie --version
bowtie version 0.12.7
64-bit
Built on allspice
Thu May  5 12:19:01 UTC 2011
Compiler: gcc version 4.6.1 20110503 (prerelease) (Ubuntu 4.6.0-6ubuntu2)
Options: -O3   -Wl,--hash-style=both -g -O2  -g -O2
Sizeof {int, long, long long, void*, size_t, off_t}: {4, 8, 8, 8, 8, 8}
joachim@joachim-VirtualBox ~ $ which bowtie
/usr/bin/bowtie
```

How does the terminal know **where to look** for executables? (e.g. how does it know bowtie is in /usr/bin?)

Environment variables

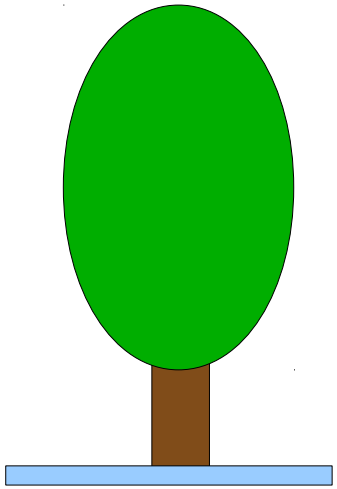
A **variable** is a word that represents/contains a value or string. Environment variables describe your system. Fictive example:

Environment variable

TREES are green on my system

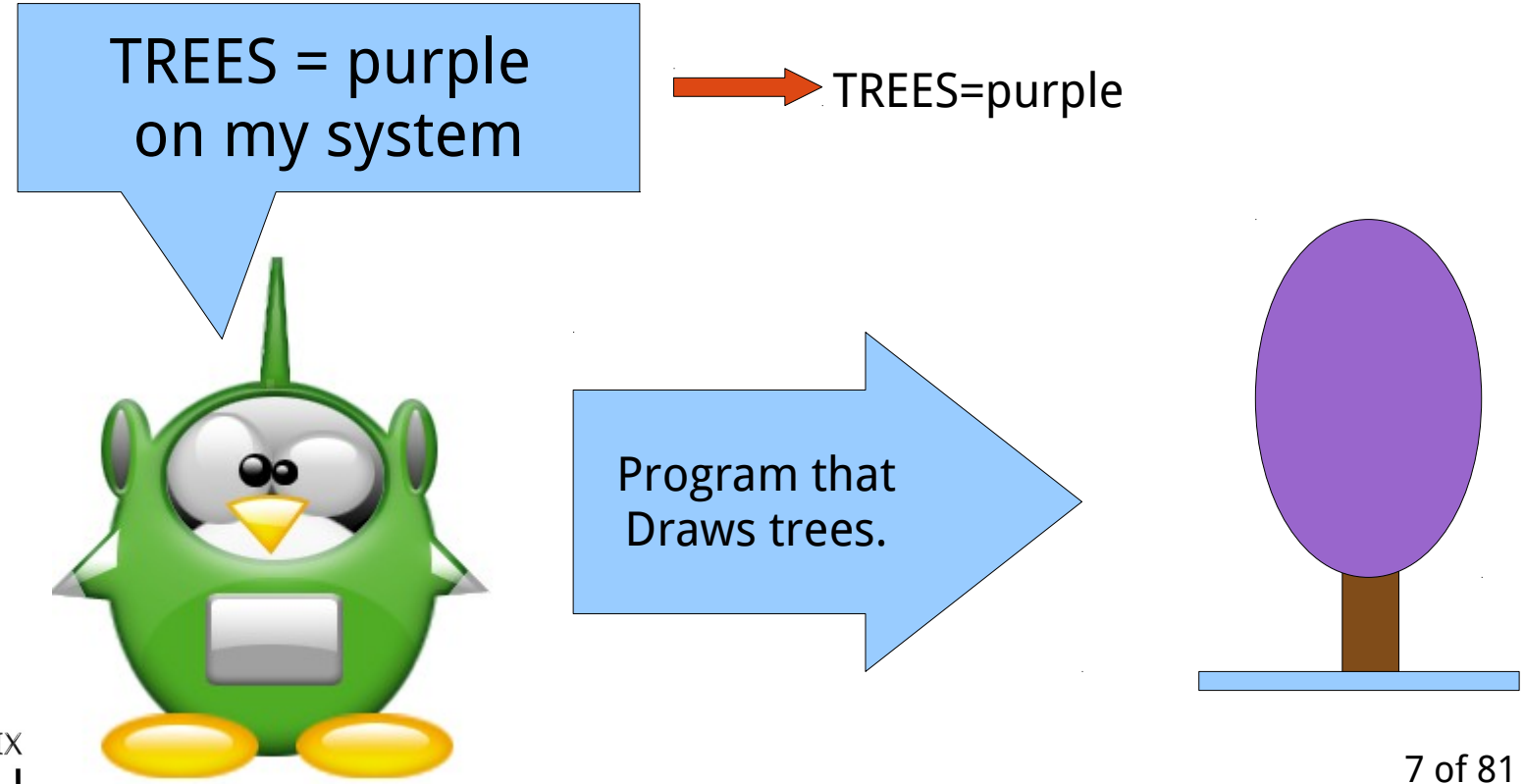
TREES=green

Program that Draws trees.



Programs use env variables

Depending on how **environment variables** are set, programs can change their behaviour.



'env' displays environment vars

```
joachim@joachim-VirtualBox ~ $ env
SSH_AGENT_PID=1252
GPG_AGENT_INFO=/tmp/keyring-h6IMRu/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=1fe9d2682cdf8e0b84c644ae00000005-1350300009.579787-965072362
WINDOWID=71303173
GNOME_KEYRING_CONTROL=/tmp/keyring-h6IMRu
USER=joachim
SSH_AUTH_SOCK=/tmp/keyring-h6IMRu/ssh
SESSION_MANAGER=local/joachim-VirtualBox:@/tmp/.ICE-unix/1164,unix/joachim-VirtualBox:/tmp/.ICE-unix/1164
USERNAME=joachim
DEFAULTS_PATH=/usr/share/gconf/default.desktop.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-default.desktop:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
DESKTOP_SESSION=default.desktop
PWD=/home/joachim
LANG=en_US.UTF-8
MANDATORY_PATH=/usr/share/gconf/default.desktop.mandatory.path
MDM_XSERVER_LOCATION=local
SHLVL=1
HOME=/home/joachim
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=joachim
XDG_DATA_DIRS=/usr/share/default.desktop:/usr/share/gnome:/usr/local/share/:/usr/share/:/usr/share/mdm/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-19ZwuvREwJ,guid=8e7ffef9257d0b74c747f6bb0000001b
MDMSESSION=default.desktop
WINDOWPATH=8
DISPLAY=:0.0
MDM_LANG=en_US.UTF-8
XDG_CURRENT_DESKTOP=GNOME
COLORTERM=gnome-terminal
XAUTHORITY=/home/joachim/.Xauthority
_=/usr/bin/env
```

Programs need to be in the PATH

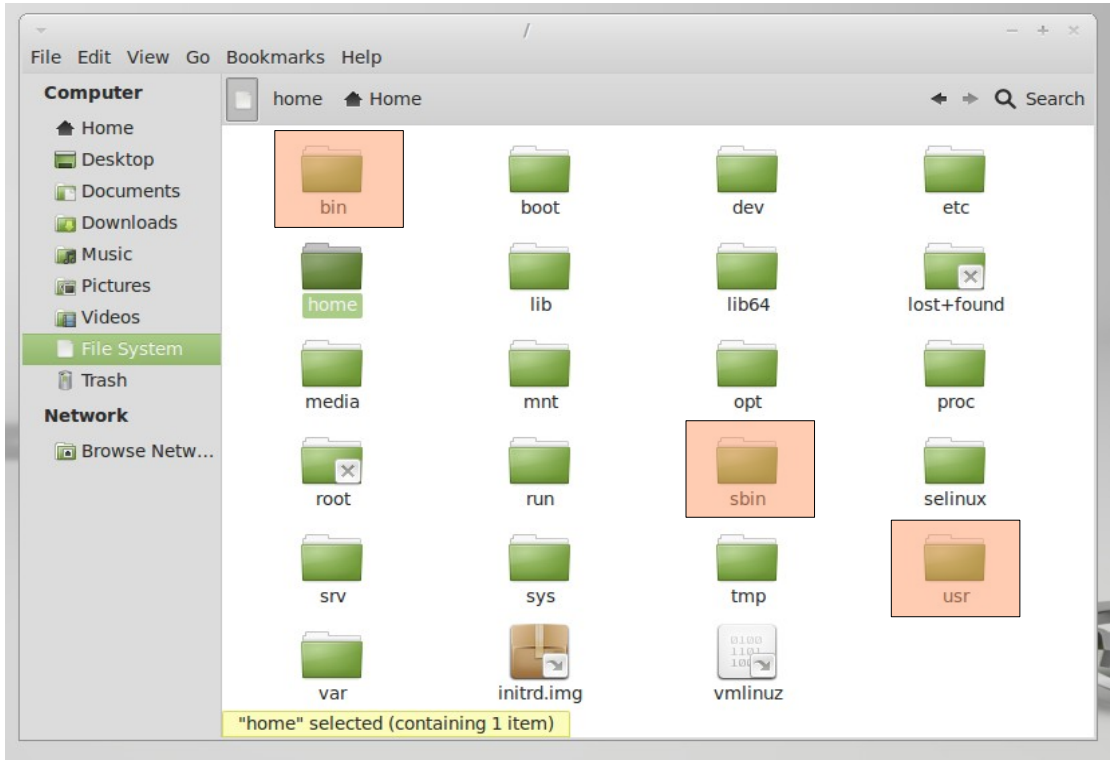
```
joachim@joachim-VirtualBox ~ $ env
SSH_AGENT_PID=1252
GPG_AGENT_INFO=/tmp/keyring-h6IMRu/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=1fe9d2682cdf8e0b84c644ae00000005-1350300009.579787-965072362
WINDOWID=71303173
GNOME_KEYRING_CONTROL=/tmp/keyring-h6IMRu
USER=joachim
SSH_AUTH_SOCK=/tmp/keyring-h6IMRu/ssh
SESSION_MANAGER=local/joachim-VirtualBox:@/tmp/.ICE-unix/1164,unix/joachim-VirtualBox:/tmp/.ICE-unix/1164
USERNAME=joachim
DEFAULTS_PATH=/usr/share/gconf/default.desktop.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-default.desktop:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
DESKTOP_SESSION=default.desktop
PWD=/home/joachim
LANG=en_US.UTF-8
MANDATORY_PATH=/usr/share/gconf/default.desktop.mandatory.path
MDM_XSERVER_LOCATION=local
SHLVL=1
HOME=/home/joachim
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=joachim
XDG_DATA_DIRS=/usr/share/default.desktop:/usr/share/gnome:/usr/local/share/:/usr/share/:/usr/share/mdm/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-19ZwuvREwJ,guid=8e7ffef9257d0b74c747f6bb0000001b
MDMSESSION=default.desktop
WINDOWPATH=8
DISPLAY=:0.0
MDM_LANG=en_US.UTF-8
XDG_CURRENT_DESKTOP=GNOME
COLORTERM=gnome-terminal
XAUTHORITY=/home/joachim/.
_=/usr/bin/env
```

https://help.ubuntu.com/community/EnvironmentVariables

The PATH environment variable

PATH contains a set of directories, separated by ':'

```
$ echo $PATH
/home/joachim/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games
```



Installing is just placing the executable

1. You **copy** the executable to one of the folders in PATH

```
$ sudo cp /home/joachim/Downloads/tSNE /usr/local/bin
$ sudo chmod +x /usr/local/bin/tSNE
```

2. You **create a sym(bolic) link** to an executable in the one of the folders in PATH
(see previous week)

3. You **add a directory to the PATH variable**

3. Add a directory to the PATH

Export <environment_variable_name>=<value>

```
joachim@joachim-VirtualBox ~/soft/snap-0.13.4-linux $ echo ${PATH}
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
joachim@joachim-VirtualBox ~/soft/snap-0.13.4-linux $ export PATH=/home/joachim/soft/snap-0.13.4-linux:$PATH
joachim@joachim-VirtualBox ~/soft/snap-0.13.4-linux $ snap
Welcome to SNAP version 0.13.4.

Usage: snap <command> [<options>]
Commands:
   index    build a genome index
   single   align single-end reads
   paired   align paired-end reads
Type a command without arguments to see its help.
joachim@joachim-VirtualBox ~/soft/snap-0.13.4-linux $ echo $PATH
/home/joachim/soft/snap-0.13.4-linux:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
joachim@joachim-VirtualBox ~/soft/snap-0.13.4-linux $
```

Env variables are stored in a text file

```
$ cat /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games"
```

/etc is the directory that contains configuration text files. It is only owned by root: system-wide settings.

A 'normal' user (session-wide settings) can create the file **~/.pam_environment** to set the vars with content

```
joachim@joachim-VirtualBox ~ $ cat .pam_environment
PATH    DEFAULT=${PATH}:~/soft/snap-0.13.4-linux:/opt/bin
TESTVAR         DEFAULT="123"
joachim@joachim-VirtualBox ~ $
```

Recap: editing files

Create a text file with the name .pam_environment and open in an editor:

```
$ nano .pam_environment
```
→ quit by pressing ctrl-x

```
$ gedit .pam_environment
```
→ graphical

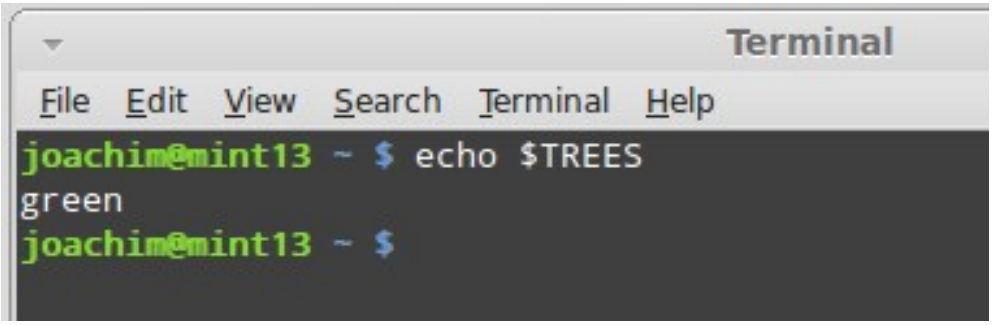
```
$ vim .pam_environment
```
→ quit by pressing one after the other :q!

Create .pam_environment

In ~/.pam_environment, type:

TREES DEFAULT=green

Save the file. Log out and log back in.



Bash variables are limited in scope

You can assign any variable you like in **bash**, like:

```
joachim@joachim-VirtualBox ~ $ myname="joachim"
joachim@joachim-VirtualBox ~ $ echo $myname
joachim
joachim@joachim-VirtualBox ~ $ echo ${myname}
joachim
```

The name of the variable can be any normal **string**. This variable exists only in this terminal. The command **echo** can display the value assigned to that variable. The value of a variable is referred to by **$ {varname}** or **$varname**.

It can be used in scripts!

All commands you type, you can put one after the other in a **text file**, and let bash execute it.

Let's try!

Make a file in your ~ called '**space_left**':

Enter two following bash commands in this file:

```
df -h .
du -sh */
```

Running our first bash script

```
joachim@mint13 ~ $ bash space_left
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        12G  5.4G  5.6G  50% /
udev            490M  4.0K  490M   1% /dev
tmpfs           200M  916K  199M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            498M   76K  498M   1% /run/shm
520K    bin/
281M    Compression_exercise/
4.0K    Desktop/
4.0K    Documents/
7.5M    Downloads/
4.0K    Music/
4.0K    Pictures/
4.0K    Public/
451M    Rice Example/
4.0K    Templates/
4.0K    test/
114M    ugene-1.12.2/
4.0K    Videos/
joachim@mint13 ~ $
```

The shebang

Simple text files become Bash scripts when adding a **shebang** line as first line, saying which program should read and execute this text file.

```
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/python
```

(see our other trainings for perl and python)

Things to remember

- Linux determines files types based on its content (not extension).
- Change permissions of scripts to read and execute to allow running in the command line:

```
$ chmod +x filename
```

```
joachim@joachim-VirtualBox ~ $ ll hello
-rw-r--r-- 1 joachim joachim 81 Oct 17 11:17 hello
joachim@joachim-VirtualBox ~ $ chmod +x hello
joachim@joachim-VirtualBox ~ $ ./hello
First line
Second line
No one messes with joachim
joachim@joachim-VirtualBox ~ $
```

Exercise

→ A simple bash script

Can you reconstruct the script?

```
joachim@joachim-VirtualBox ~ $ ll hello
-rw-r--r-- 1 joachim joachim 81 Oct 17 11:17 hello
joachim@joachim-VirtualBox ~ $ chmod +x hello
joachim@joachim-VirtualBox ~ $ ./hello
First line
Second line
No one messes with joachim
joachim@joachim-VirtualBox ~ $
```

One slide on permissions

```
$ chown user:group filename

$ chmod [ugo][+-][rwx] filename
```

or

```
$ chmod [0-7][0-7][0-7] filename
```

1 stands for execute
2 stands for write
4 stands for read
→ any number from 0 to 7 is a unique combination of 1, 2 and 4.

JAKONIX VIB

Passing arguments to bash scripts

We can pass on **arguments** to our scripts: they are subsequently stored in variables called $1, $2, $3,...

Make a file called 'arguments.sh' with following contents (copy paste is fine – be aware of the “):

```
#!/bin/bash
firstarg=$1
secondarg=$2
echo “You have entered \”$firstarg\”
and \”$secondarg\””
```

Passing arguments to bash scripts

Make your script executable.

$ chmod +x arguments.sh

Passing arguments to bash scripts

Let's try to look at it, and run it.

```
joachim@joachim-VirtualBox ~ $ cat arguments.sh
#!/bin/bash
firstarg=$1
secondarg=$2
echo "You have entered \"$firstarg\" and \"$secondarg\""

joachim@joachim-VirtualBox ~ $ ./arguments.sh first and second
You have entered "first" and "and"
```

Arguments are separated by white spaces

The string after the command is chopped on the **white spaces**. Different cases (note the “ and \):

```
joachim@joachim-VirtualBox ~ $ cat arguments.sh
#!/bin/bash
firstarg=$1
secondarg=$2
echo "You have entered \"$firstarg\" and \"$secondarg\""

joachim@joachim-VirtualBox ~ $ ./arguments.sh first and second
You have entered "first" and "and"
```

Arguments are separated by white spaces

The string after the command is chopped on the **white spaces**. Different cases (note the “ and \):

```
joachim@joachim-VirtualBox ~ $ ./arguments.sh first and second
You have entered "first" and "and"
joachim@joachim-VirtualBox ~ $ ./arguments.sh "first and" second
You have entered "first and" and "second"
joachim@joachim-VirtualBox ~ $ ./arguments.sh first\ and\ second
You have entered "first and second" and ""
joachim@joachim-VirtualBox ~ $ ./arguments.sh "first and second"
You have entered "first and second" and ""
joachim@joachim-VirtualBox ~ $ ./arguments.sh first second
You have entered "first" and "second"
```

Useful example in bioinformatics

For example, look at the script on our wiki:
http://wiki.bits.vib.be/index.php/Bfast

Lines starting with **#** are ignored.

```
#!/bin/bash
# This script creates an index of a genome for bfast with highest accuracity for reads <40bp
# Based on the sets of masks reported in the manuscript of bfast
# Usage: pass the reference genome file in fasta (.brg needs to be in same dir) as an argument

genomefile=$1
numThreads=16

bfast index -f $genomefile -n $numThreads -i 1 -A 0 -m 1111111111111111111 -w 14
bfast index -f $genomefile -n $numThreads -i 2 -A 0 -m 111101001101111101010101111 -w 14
bfast index -f $genomefile -n $numThreads -i 3 -A 0 -m 1111111111111001111 -w 14
bfast index -f $genomefile -n $numThreads -i 4 -A 0 -m 111101110110010100111111 -w 14
bfast index -f $genomefile -n $numThreads -i 5 -A 0 -m 1111011100010101000001010111 0111 -w 14
bfast index -f $genomefile -n $numThreads -i 6 -A 0 -m 1011001101011110100110010010111 -w 14
bfast index -f $genomefile -n $numThreads -i 7 -A 0 -m 1110110010100001000101100111001111 -w 14
bfast index -f $genomefile -n $numThreads -i 8 -A 0 -m 1111011111111111111 -w 14
bfast index -f $genomefile -n $numThreads -i 9 -A 0 -m 1101111110001011011110110 1 -w 14
bfast index -f $genomefile -n $numThreads -i 10 -A 0 -m 1110100011100011101000110111 11 -w 14

echo "Done!"
```

Chaining command line tools

This is the ultimate power of Unix-like OSes. The philosophy is that every tool should do **one small specific task**. By combining tools we can create a bigger piece of software fulfilling our needs.

How **combining** different tools?

1. writing scripts

2. pipes

Chaining the output to input

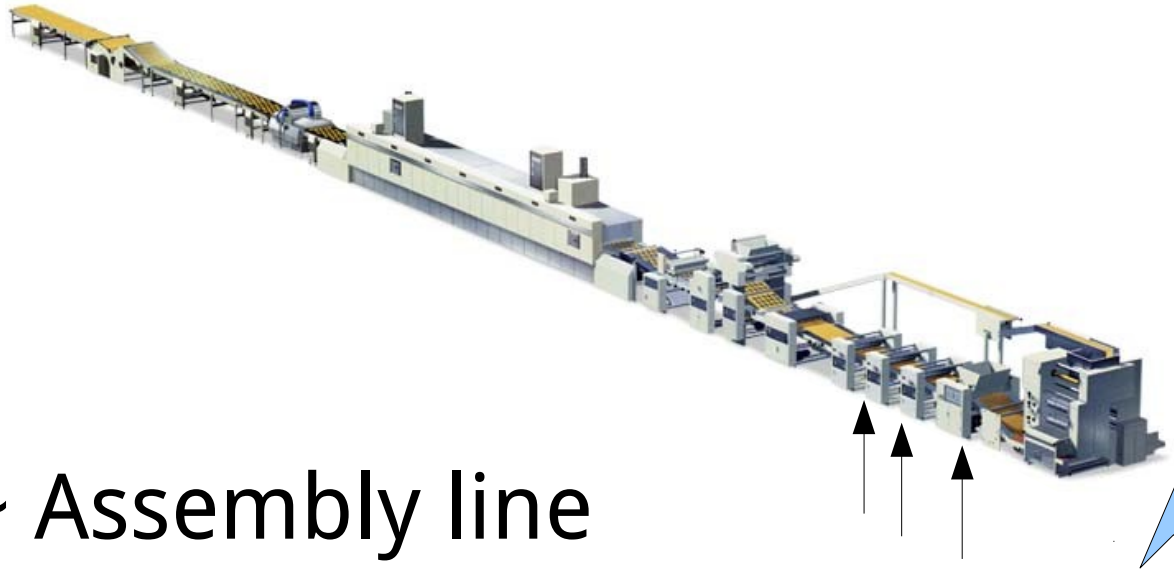
What the programs take in, and what they print out...

```
joachim@joachim-VirtualBox ~ $ cat arguments.sh
#!/bin/bash
firstarg=$1
secondarg=$2
echo "You have entered \"$firstarg\" and \"$secondarg\""

joachim@joachim-VirtualBox ~ $ ./arguments.sh first and second
You have entered "first" and "and"
```

Chaining the output to input

We can take the output of one program, store it, and use it as input for another program



~ Assembly line

Deliverance through channels

When a program is executed, 3 *channels* are opened:

- **stdin**: an input channel – what is read by the program
- **stdout**: channel used for functional output
- **stderr**: channel used for error reporting

In UNIX, open files have an identification number called a file descriptor:

- **stdin** called by 0
- **stdout** called by 1
- **stderr** called by 2

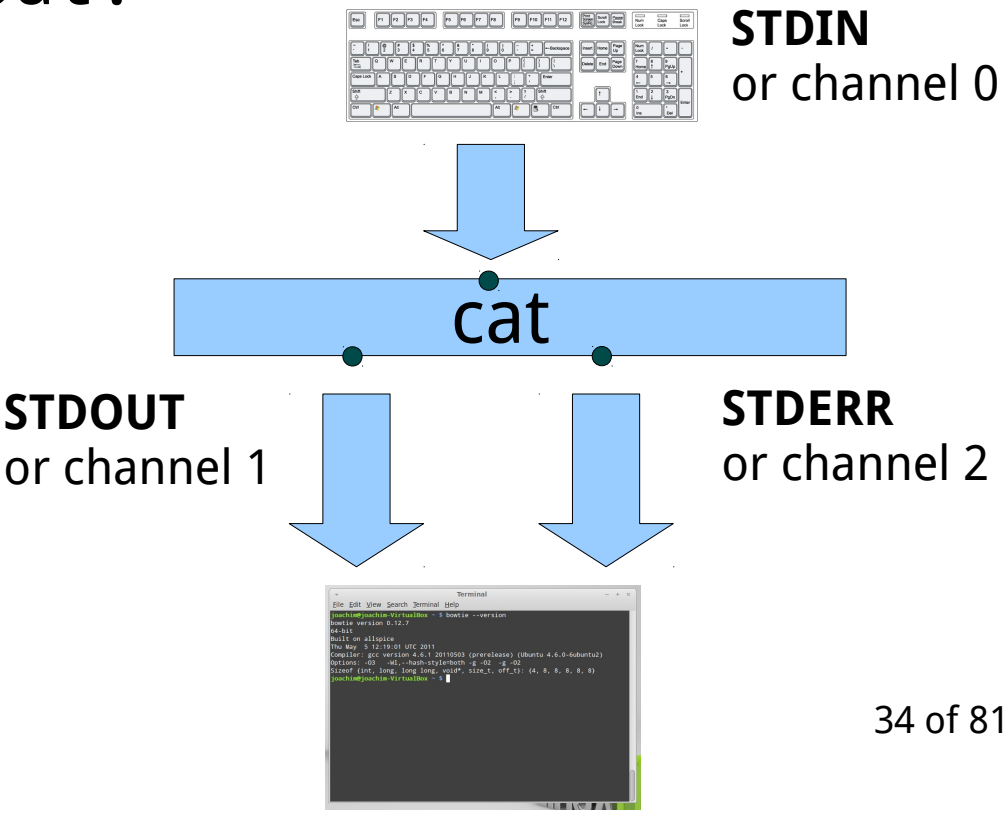
(*) by convention

http://www.linux-tutorial.info/modules.php?name=MContent&pageid=21

I/O redirection of terminal programs

```
$ cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s), or standard input,
to standard output.
```

"When cat is run it waits for input. As soon as an enter is entered output is written to STDOUT."



I/O redirection

When `cat` is launched without any arguments, the program reads from **stdin** (keyboard) and writes to **stdout** (terminal).

Example:

```
$ cat
```

type:

```
DNA: National Dyslexia Association↵
```

result:

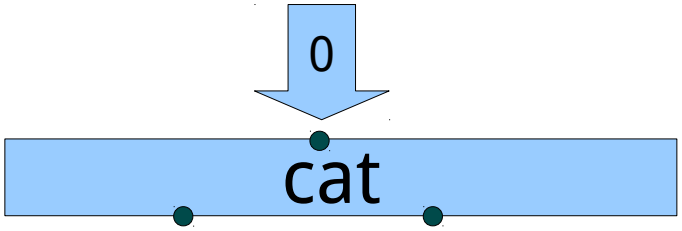
```
DNA: National Dyslexia Association
```

You can stop the program using the '*End Of Input*' character CTRL-D

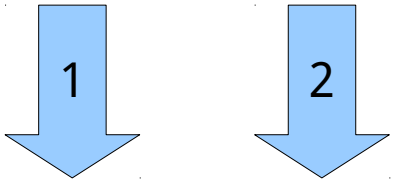
I/O redirection of terminal programs

Takes input from the **keyboard**, attached to STDIN.

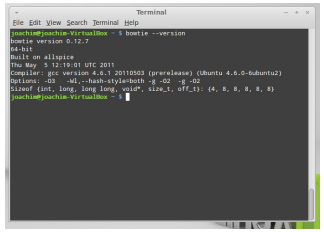
STDIN
or channel 0



STDOUT
or channel 1

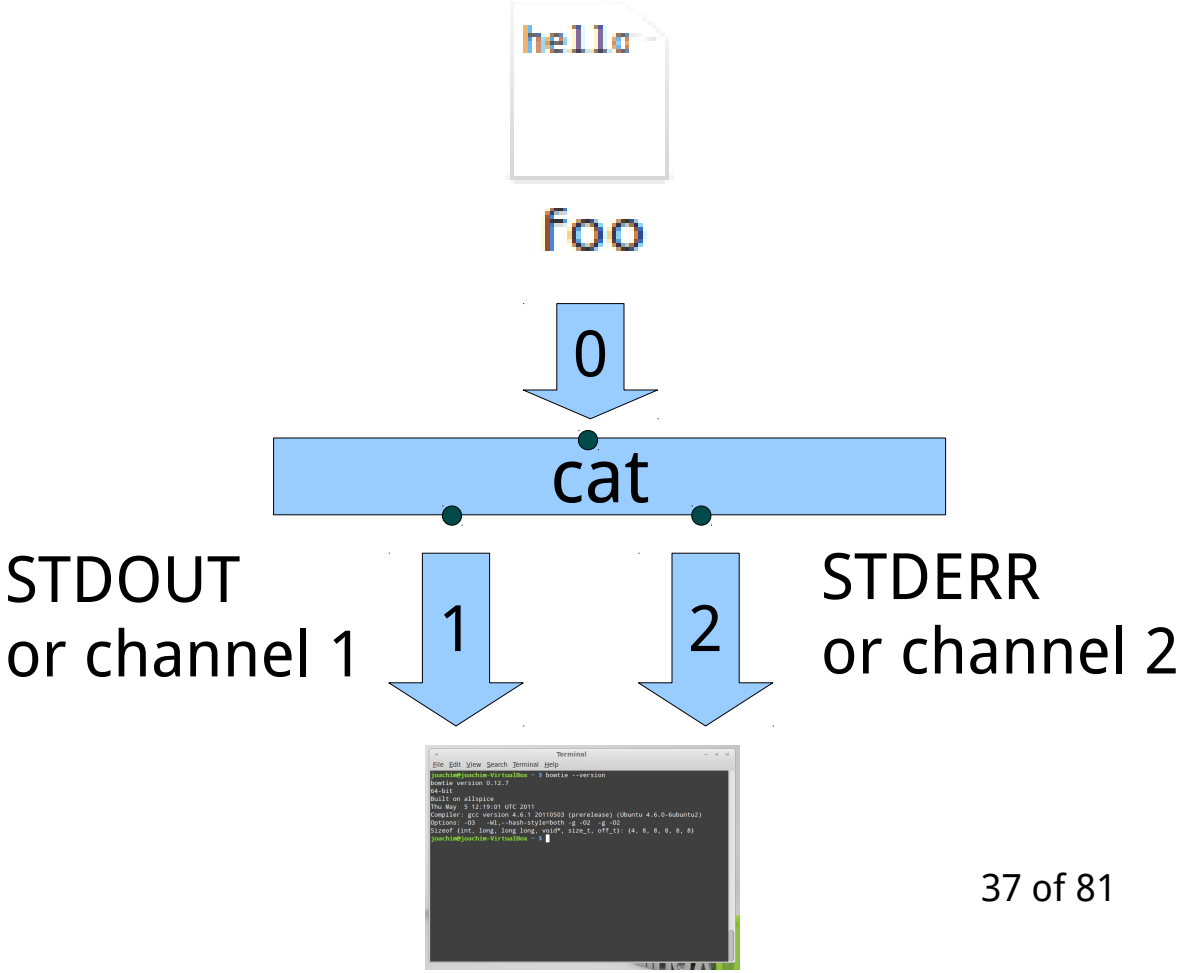


STDERR
or channel 2



I/O redirection of terminal programs

Takes input from files, which is attached to STDIN



I/O redirection of terminal programs

Connect a file to STDIN:

```
$ cat 0< file
```

or shorter:

```
$ cat < file
```

or even shorter (and most used – what we know already)

```
$ cat file
```

Example:

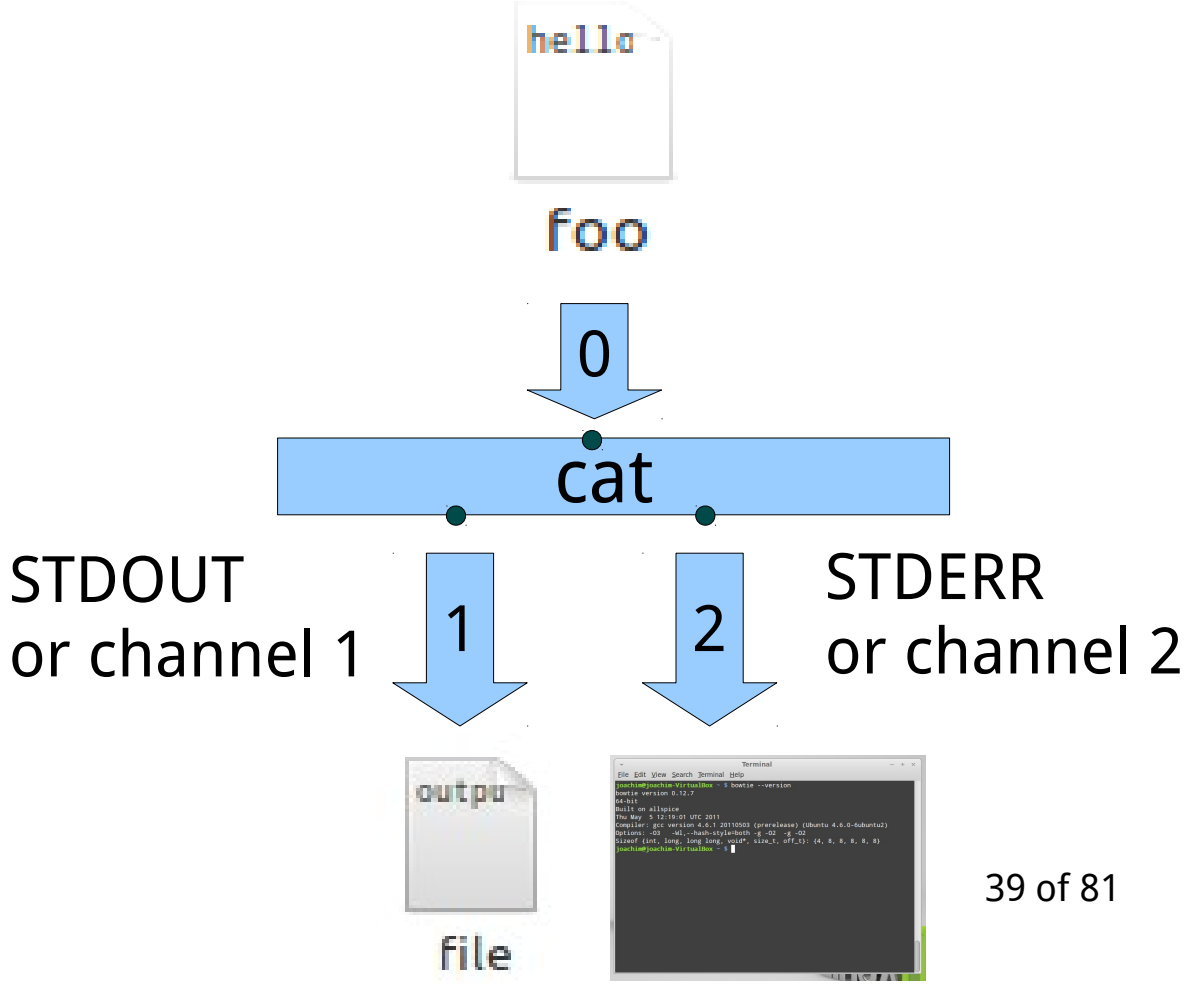
```
$ cat ~/arguments.sh
```

Try also:

```
$ cat 0<arguments.sh
```

Output redirection

Can write output to files, instead of the terminal



Output redirection

- The **stdout** output of a program can be saved to a file (or device):
  ```
  $ cat 1> file
  ```
  or short:
  ```
  $ cat > file
  ```

- Examples:
  ```
  $ ls -lR / > /tmp/ls-lR

  $ less /tmp/ls-lR
  ```

Chaining the output to input

You have noticed that running:

```
$ ls -lR / > /tmp/ls-lR
```

outputs some warnings/errors on the screen: this is all output of **STDERR** (note: channel 1 is redirected to a file, leaving only channel 2 to the terminal)

```
ls: cannot open directory /root: Permission denied
ls: cannot open directory /run/cups/certs: Permission denied
ls: cannot open directory /run/udisks: Permission denied
ls: cannot open directory /sys/kernel/debug: Permission denied
ls: cannot open directory /tmp/pulse-PKdhtXMmr18n: Permission denied
ls: cannot open directory /var/cache/ldconfig: Permission denied
```

Chaining the output to input

Redirect the errors to a file called 'error.txt'.

```
$ ls -lR /
```

?

Chaining the output to input

Redirect the error channel to a file error.txt.

```
$ ls -lR / 2 > error.txt

$ less error.txt
```

Beware of overwriting output

IMPORTANT, if you write to a file, the contents are being **replaced** by the output.

To **append** to file, you use:

```
$ cat 1>> file
```

or short

```
$ cat >> file
```

Example:

```
$ echo “Hello” >> append.txt
$ echo “World” >> append.txt
$ cat append.txt
```

Chaining the output to input

	INPUT	OUTPUT	
Input from file	< filename	> filename	**Output to a file**
Input until string EOF	<< EOF	>> filename	**Append output to a file**
Input directly from string	<<< “This string is read”		

```
joachim@joachim-VirtualBox ~ $ cat <<EOF >cattest.txt
> Adding to cattest.txt
> until following string is encountered:
> EOF
```

Special devices

- For input:

`/dev/zero` all zeros

`/dev/urandom` (pseudo) random numbers

- For output:

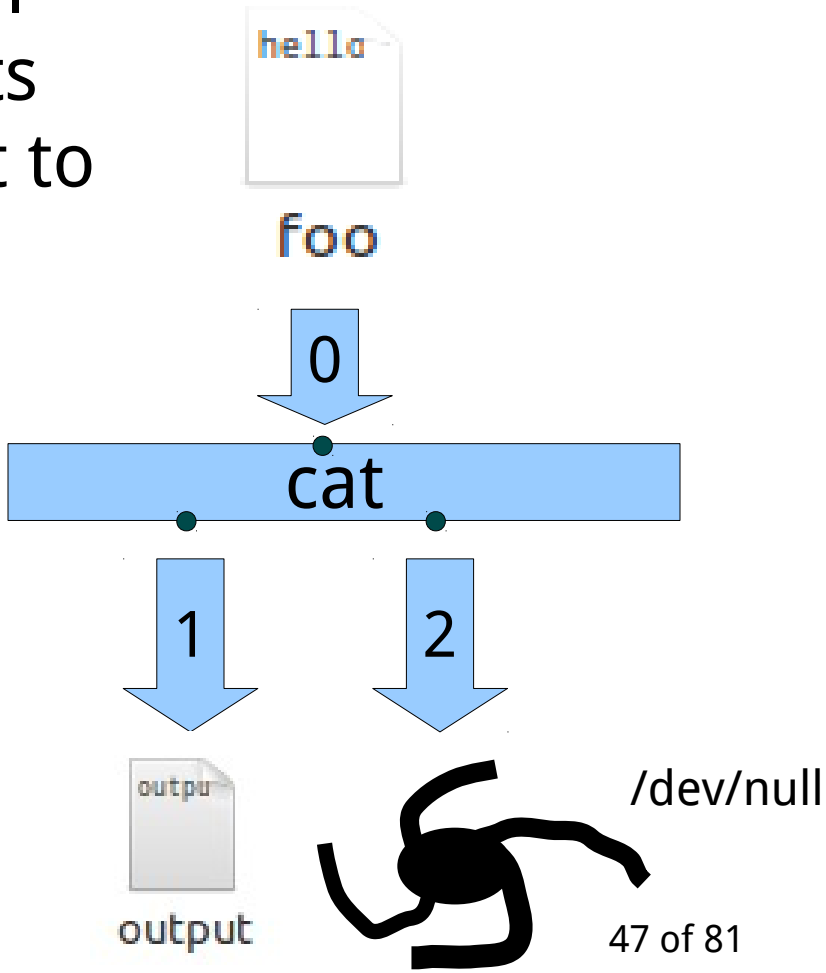
`/dev/null` 'bit-heaven'

Example:
You are not interested in the errors from the a certain command: send them to /dev/null !

```
$ ls -alh / 2> /dev/null
```

Summary of output redirection

- Error direction to /dev/null
- The program can run on its own: input from file, output to file.



Plumbing with in- and outputs

- Example:

```
$ ls -lR ~ > /tmp/ls-lR

$ less < /tmp/ls-lR ('Q' to quit)

$ rm -rf /tmp/ls-lR
```

Plumbing with in- and outputs

- Example:

```
$ ls -lR ~ > /tmp/ls-lR

$ less < /tmp/ls-lR ('Q' to quit)

$ rm -rf /tmp/ls-lR
```

can be shortened to:

```
$ ls -lR ~ | less
```

(Formally speaking: the stdout channel of `ls` is connected to the stdin channel of less)

Combining pipe after pipe

Pipes can pass the output of a command to another command, which on his turn can pipe it through, until the final output is reached.

```
$ history | awk '{ print  $2 }'  \
| sort | uniq -c | sort -nr | head -3

    237 ls
    180 cd
    103 ll
```

Text tools combine well with pipes

UNIX has an extensive toolkit for **text** analysis:

- **Extraction**: head, tail, grep, awk, uniq
- **Reporting**: wc
- **Manipulation**: dos2unix, sort, tr, sed

But, the UNIX tool for heavy text parsing is **perl** (see https://www.bits.vib.be/index.php/training/175-perl)

JAKONIX
VIB

Grep: filter lines from text

grep extracts lines that match a string.

Syntax:

```
$ grep [options] regex [file(s)]
```

The file(s) are read line by line. If the line matches the given criteria, the entire line is written to stdout.

Grep example

A GFF file contains genome annotation information. Different types of annotations are mixed: gene, mRNA, exons, ...

Filtering out one type of annotation is very easy with **grep**.

Task:
Filter out all lines from locus Os01g01070 in all.gff3 (should be somewhere in your Rice folder).

Grep example

```
joachim@mint13 ~/Rice Example/Genome data/Annotation $ grep LOC_Os01g01070 all.gff3
Chr1    MSU_osa1r7      gene    29818   34493   .       +       .       ID=LOC_Os01g01070;Name=LOC_Os01g01070;Not
Chr1    MSU_osa1r7      mRNA    29818   34493   .       +       .       ID=LOC_Os01g01070.1;Name=LOC_Os01g01070.1
Chr1    MSU_osa1r7      exon    29818   29976   .       +       .       ID=LOC_Os01g01070.1:exon_1;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    30146   30228   .       +       .       ID=LOC_Os01g01070.1:exon_2;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    30735   30806   .       +       .       ID=LOC_Os01g01070.1:exon_3;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    30885   30963   .       +       .       ID=LOC_Os01g01070.1:exon_4;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    31258   31331   .       +       .       ID=LOC_Os01g01070.1:exon_5;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    31505   31606   .       +       .       ID=LOC_Os01g01070.1:exon_6;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    32377   32466   .       +       .       ID=LOC_Os01g01070.1:exon_7;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    32542   32616   .       +       .       ID=LOC_Os01g01070.1:exon_8;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    32712   32744   .       +       .       ID=LOC_Os01g01070.1:exon_9;Parent=LOC_Os0
Chr1    MSU_osa1r7      exon    32828   32908   .       +       .       ID=LOC_Os01g01070.1:exon_10;Parent=LOC_Os
Chr1    MSU_osa1r7      exon    33277   33330   .       +       .       ID=LOC_Os01g01070.1:exon_11;Parent=LOC_Os
Chr1    MSU_osa1r7      exon    33400   33471   .       +       .       ID=LOC_Os01g01070.1:exon_12;Parent=LOC_Os
Chr1    MSU_osa1r7      exon    33543   33617   .       +       .       ID=LOC_Os01g01070.1:exon_13;Parent=LOC_Os
Chr1    MSU_osa1r7      exon    33975   34493   .       +       .       ID=LOC_Os01g01070.1:exon_14;Parent=LOC_Os
Chr1    MSU_osa1r7      five_prime_UTR  29818   29939   .       +       .       ID=LOC_Os01g01070.1:utr_1;Parent=
Chr1    MSU_osa1r7      CDS     29940   29976   .       +       .       ID=LOC_Os01g01070.1:cds_1;Parent=LOC_Os01
```

Grep

`-i`: ignore case

matches the regex case insensitively

`-v`: inverse

shows all lines that do *not* match the regex

`-l`: list

shows only the **name** of the files that contain a match

`-n`:

shows *n* lines around the match

`--color`:

highlights the match

Finetuning filtering with regexes

A **regular expression**, aka regex, is a formal way of describing sets of strings, used by many tools: grep, sed, awk, ...

It resembles wild card-functionility (e.g. `ls *`) (also called globbing), but is more extensive.

http://www.slideshare.net/wvcrieki/bioinformatics-p2p3perlregexes-v2013wimvancriekinge?utm_source=slideshow&utm_medium=ssemail&u

http://www.faqs.org/docs/abs/HTML/regexp.html

Basics of regexes

A plain character in a regex matches itself.

. = any character

^ = beginning of the line

$ = end of the line

[ ] = a set of characters

Example:

```
$ grep chr[1-5] all.gff3
```

Regex	chr	chr[1-5]	chr.	AAF12\.[1-3]	AT[1,5]G[:digit:]+/.[1,2]
Matching string set	chr1	chr1	chr1	AAF12.1	AT5G08160.1
	chr2	chr2	chr2	AAF12.2	AT5G08160.2
	chr3	chr3	chr3	AAF12.3	AT5G10245.1
	chr4	chr4	chr4		AT1G14525.1
	chr5	chr5	chr5		

Basics of regexes

Example: from TAIR9_mRNA.bed, filter out the mRNA structures from chr1 and only on the + strand.

```
$ egrep '^chr1.+\+' TAIR9_mRNA.bed  > out.txt
```

^ matches the start of a string

. matches any char

Since + is a special character (standing for a repeat of one or more), we need to **escape** it.

\+ matches a '+' symbol as such

^chr1
Matches lines
With 'chr1' appearing
At the beginning

.+
matches
any string

Together in this order, the regex filters out lines of chr1 on + strand

Finetuning filtering with regexes

```
$ egrep '^chr1.+\+' TAIR9_mRNA.bed  > out.txt

chr1    2025600 2027271 AT1G06620.10   +   2025617 2027094 0   3541,322,
chr1    16269074   16270513   AT1G43171.10   +   1626998816270327   0
chr1    28251959   28253619   AT1G75280.10   +   2825202928253355   0
chr1    693479  696382  AT1G03010.10   +   693479  696188  0   592,67,11
```


http://www.faqs.org/docs/abs/HTML/regexp.html

Wc – count words in files

A general tool for counting lines, words and characters:
`wc [options] file(s)`

`c`: show number of characters
`w`: show number of words
`l`: show number of lines

How many mRNA entries are on chr1 of *A. thaliana*?

```
$ wc -l chr1_TAIR9_mRNA.bed
```

or

```
$ grep chr1 TAIR9_mRNA.bed | wc -l
```

Translate

To replace characters:

```
$ tr 's1' 's2'
```

! tr always reads from stdin – you cannot specify any files as command line arguments. Characters in `s1` are replaced by characters in `s2`.

Example:

```
$ echo 'James Watson' | tr '[a-z]' '[A-Z]'
JAMES WATSON
```

Delete characters

To remove a particular set of characters:

```
$ tr -d 's1'
```

Deletes all characters in `s1`

Example:

```
$ tr -d '\r' < DOStext > UNIXtext
```

awk can extract and rearrange

... specific fields and do calculations and manipulations on it.

```
awk -F delim '{ print $x }'
```

- `-F delim`: the field separator (default is white space)
- `$x` the field number:
  - $0: the complete line
  - $1: first field
  - $2: second field
  - ...

NF is the cumber of fields (can also be taken for last field).

awk can extract and rearrange

For example: TAIR9_mRNA.bed needs to be converted to .gff (general feature format). See the .gff format http://wiki.bits.vib.be/index.php/.gff

With AWK this can easily be done! One line of .bed looks like:

```
chr1   2025600 2027271 AT1G06620.10   +   2025617 2027094 0   3541,322,429,  0
```

→ needs to be one line of .gff

awk can extract and rearrange

```
chr1    2025600 2027271 AT1G06620.10    +    2025617 2027094 0    3541,322,429,  0
```

```
$ awk '{print $1"\tawk\tmRNA\t"$2"\t"$3"\t" \
  $5"\t"$6"\t0\t"$4 }' TAIR9_mRNA.bed
```

chr4	awk	mRNA	7607310	7609843	0	+	0	AT4G13030.1	
chr1	awk	mRNA	26597312	26600219	0	+	0		AT1G70550.1
chr1	awk	mRNA	26597151	26600219	0	+	0		AT1G70550.2
chr5	awk	mRNA	23637706	23640920	0	-	0		AT5G58470.2
chr5	awk	mRNA	23637706	23640920	0	-	0		AT5G58470.1
chr3	awk	mRNA	3988742	3991052	0	+	0	AT3G12570.1	
chr3	awk	mRNA	3988751	3991052	0	+	0	AT3G12570.3	
chr3	awk	mRNA	3988669	3991052	0	+	0	AT3G12570.2	
chr3	awk	mRNA	3988713	3991079	0	+	0	AT3G12570.4	

Awk has also a filtering option

Extraction of one or more fields from a tabular data stream of lines that match a given regex:

```
awk -F delim '/regex/ { print $x }'
```

Here is:

- `Delim:` the delimiter in the file
- `regex`: a regular expression

The awk script is executed only if the line matches regex lines that do not match regex are removed from the stream

Further excellent documentation on awk:
http://www.grymoire.com/Unix/Awk.html

Cut selects columns

Cut extracts fields from text files:

- Using fixed delimiter

```
$ cut [-d delim] -f <fields> [file]
```

- chopping on fixed width

```
$ cut -c <fields> [file]
```

For <fields>:

`N`	the Nth element
`N-M`	element the Nth till the Mth element
`N-`	from the Nth element on
`-M`	till the Mth element

The first element is 1.

Cutting columns from text files

Fixed width example:
Suppose there is a file fixed.txt with content
`12345ABCDE67890FGHIJ`

To extract a range of characters:

```
$ cut -c 6-10 fixed.txt
ABCDE
```

Sorting output

To sort alphabetically or numerically lines of text:

```
$ sort [options] file(s)
```

When more files are specified, they are read one by one, but all lines together are sorted.

Sorting options

- n sort numerically
- f fold – case-insensitive
- r reverse sort order
- t*s* use *s* as field separator (instead of space)
- k*n* sort on the *n*-th field (1 being the first field)

Example: sort mRNA by chromosome number and next by number of exons.

```
$ sort -n -k1 -k10 TAIR9_mRNA.bed  > \
out.bed
```

Detecting unique records with uniq

- eliminate duplicate lines in a set of files
- display unique lines
- display and count duplicate lines

Very important: `uniq` always needs from **sorted** input.

Useful option:
`-c` count the number of fields.

Eliminate duplicates

- Example:

```
$ who
root      tty1          Oct 16 23:20
james     tty2          Oct 16 23:20
james     pts/0         Oct 16 23:21
james     pts/1         Oct 16 23:22
james     pts/2         Oct 16 23:22

$ who | awk '{print $1}' | sort | uniq
james
root
```

Display unique or duplicate lines

- To display lines that occur only once:

```
$ uniq -u file(s)
```

- To display lines that occur more than once:

```
$ uniq -d file(s)
```

Example:

```
$ who|awk '{print $1}'|sort|uniq -d
james
```

- To display the counts of the lines

```
$ uniq -c file(s)
```

Example

```
$ who | awk '{print $1}' | sort | uniq -c
 4 james
 1 root
```

VIB

Edit per line with sed

Sed (the stream editor) can make changes in text per line. It works on files or on STDIN.

See http://www.grymoire.com/Unix/Sed.html

This is also a very big tool, but we will only look to the substitute function (the most used one).

```
$ sed -e 's/r1/s1/' file(s)
```

- `s`: the substitute command
- `/`: separator
- `r1`: regex to be replaced
- `s1`: text that will replace the regex match

Paste several lines together

Paste allows you to concatenate every *n* lines into one line, ideal for manipulating fastq files.

We can use sed for this together with **paste**.

```
$ paste - - - - < in.fq | \
    cut -f 1,2  | \
    sed 's/^@/>/' | \
    tr "\t" "\n"    >    out.fa
```

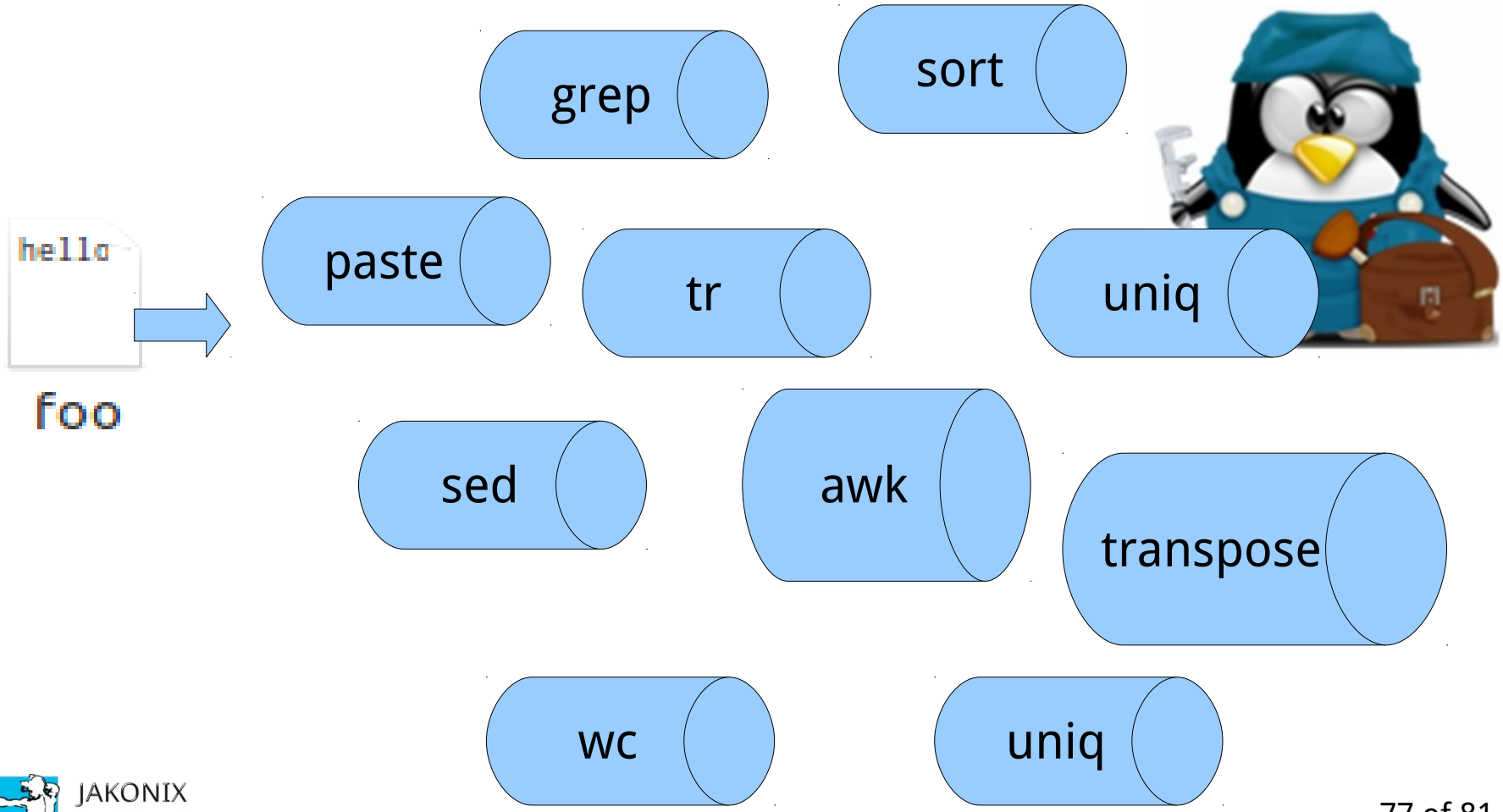

http://thegenomefactory.blogspot.be/2012/05/cool-use-of-unix-paste-with-ngs.html

Transpose is not a standard tool

http://sourceforge.net/projects/transpose/

But it is extremely useful. It transposes tabular text files, exchanging columns for row and vice versa.

Building your pipelines



Fill in the tools

Filter on lines ánd select different columns: .....

Select columns: .....

Sort lines: .....

Filter lines: .....

Merge identical fields: .....

Replace characters: .....
Replace strings: .....

Numerical summary: .....

Transpose: .....

Exercise

→ Text manipulation exercises

Keywords

Environment variable
PATH
shebang
script
argument
STDIN
pipe
comment

Break