About This Course

Program, March 21st

09:00 – 09:30 Morning coffee & registration
09:30 – 09:45 Introduction to the course and recap of previous courses
09:45 – 11:00 Bash scripting
11:00 – 12:00 Regular expressions & sed
12:00 – 13:00 Lunch
13:00 – 14:00 awk
14:00 – 14:30 Coffee
14:30 – 17:00 Hands-on session

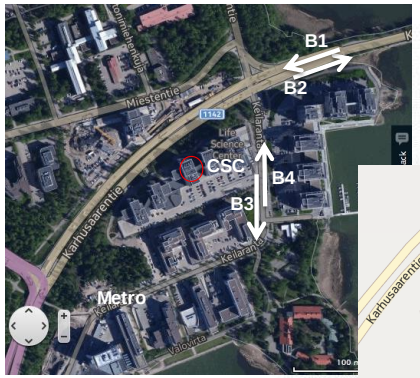
How We Teach

- All topics are presented with interactive demonstrations.
  - Please, indicate immediately, if pace is too fast. We want to have everyone with us all the time.
- Additionally, exercises to each of the sections will be provided.
- The Exercises and Wrap-up sections are meant for personal and group interaction and are (with a time-limit to 17:00 or 17:30) kept in an open end style.

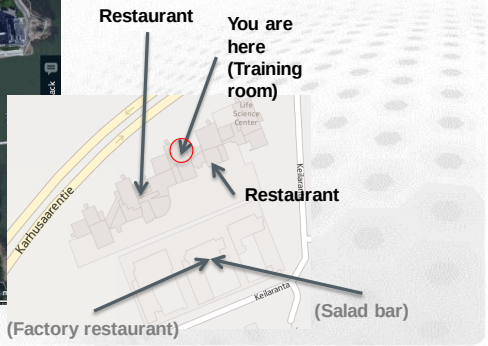
Practicalities

- Keep the name tag visible.
- Lunch is served in the same building.
  - Room will be locked during lunch but lobby is open, use the lockers.
- Toilets are in the lobby.
- Network:
  - Wi-Fi: eduroam, HAKA authentication
  - Ethernet cables on the tables
  - CSC-Guest accounts upon request
- Transportation:
  - Bus stops to Kamppi/Center are located at the other side of the street (102, 103).
  - Bus stops to Center/Pasila are on this side, towards the bridge (194, 195, 503-6).
  - To arrive at CSC the stops are at the same positions, just on opposite sides on the street.
  - If you came by car: parking is being monitored – ask for a temporary parking permit from the reception (tell which workshop you're participating).
- Visiting outside: the doors by the reception desks are open.
- Username and password for workstations: given on-site.

Around CSC



B1 (551, 552)→ Tapiola, Otaniemi
B2 (551, 552)→ Pasila, Malmi
B3 (555)→ Lauttasaari
B4 (555)→ Martinlaakso



Bash scripting

Introduction

- Shell scripting is the art of writing scripts that can automate repetitive tasks
- BASH is presented and it is discussed how it can be used as general programming language

Different shells

- sh – Bourne shell
- csh/tcsh – C shell
- ksh – Korn shell
- zsh – Z shell
- …
- ***bash*** – Bourne again shell
  - sh successor that includes csh and ksh features

When NOT to use a shell script?

- Resource or compute intensive
- Large-scale applications
- Complex datastructures
- Lot's of I/O, need libraries, floating point arithmetic, …

➡ use Python, C/C++ or Fortran, or even Java or Perl instead

When to use a shell script?

- Glue together Unix commands to automate repetitive tasks
  - awk, bc, more, less, paste, …
- Simple applications that do not need lots of resources
- Fancy scripts for batch jobs

How to write a script?

- Put commands that you would execute on the command-line in a file

```
$ cat hello.sh

echo "Hello world!"
echo "... this is my first script."

$ bash hello.sh
Hello world!

... this is my first script.
```


Pipe magic

- Commands can be chained together using a pipe |

```
$ cat hello.sh | grep my
echo "... this is my first script."

$ ./configure --help | less
```

- The output of the 1st command (left) is used as the input of the 2nd command (right)
- Arbitrary number of pipes can be used

Pipe magic with files

- Output can be redirected to a file with
  - > overwrite
  - >> append
- Input can be read from a file with
  - < read

```
$ echo "Hello world!" > hello
$ echo "... what's up?" >> hello
$ cat < hello
Hello world!
... what's up?
```

Pipes & streams

- Three special streams always opened:
  - `stdin` `0<` keyboard
  - `stdout` `1>` screen
  - `stderr` `2>` error messages (on screen)
- `stdout` and `stderr` can be joined with `&>` or `&>>`
- `M>&N` redirects file descriptor `M` to `N`
  - e.g. `2>&1` joins stderr and stdout

Pipes & streams

```
# separate stdout & stderr
$./myprogram > output 2> errors
# joined stdout & stderr
$./myprogram &> output+errors

# one-time redirect
$./myprogram > filename 2>&1
# permanent redirect
$exec 2>&1
$./myprogram > filename

# swap stdout and stderr
$./myprogram 3>&2 2>&1 1>&3
```


Executable script

- Add a shebang to the script to define the shell that executes it
  `#!shell-binary`

```
hello.sh:
#!/bin/bash
echo "Hello world!"

$ chmod u+x hello.sh
$ ./hello.sh
Hello world!
```


Comments

- Anything after # is treated as a comment
- Can be added as separate line, or at the end of a line

```
hello.sh:
#!/bin/bash
# let's greet the world...
echo "Hello world!" # cliche

$ bash hello.sh
Hello world!
```

Variables

- Like other programming languages bash has variables
- Variables can be set with `name=value`
- To use the variable precede the name with `$`, e.g.:

  `$name`

  – You may need to enclose the variable name in curly brackets:
  `${name}`

Variable expansion

```
$ name=Max
$ echo "Hi $name!"
Hi Max!

$ name="Sir Max"; d=Sun
$ echo "Dear $name,"
$ echo "today is ${d}day."
Dear Sir Max,
today is Sunday.
```


Variable types

- There are two types of variables
  - Strings
  - Integers (no native floating point type!)
- Additionally variables can be
  - Read-only
  - An array
- By default variables can hold any data we assign to it
  - This can be limited with "declare"

declare

- Declare an integer variable

  `declare -I name=value`
- Declare a read-only variable

  `declare -r name=value`
- Print the variable attributes

  `declare -p name`

Arrays

- bash supports 1D arrays
- Defining an array
  - Element-by-element: `myarray[123]=456`
  - As a list: `myarray=( 456 4 apple 1 )`
- Elements do not need to be contiguous
- Value accessed as `${myarray[123]}`
  - Does NOT work without the curly brackets!

Arrays

- Special array syntax

`${myarray[*]}`	all items
`${!myarray[*]}`	all indexes
`${#myarray[*]}`	number of items
`${#myarray[0]}`	length of item 0
`${#myarray[1]}`	length of item 1

Command line arguments

- Command line arguments are stored as special variables

`$#`	number of arguments
`$@`	all arguments
`$0`	name of script
`$1`	first argument
`$2`	second argument
…	

Command line arguments

cmd-args.sh:

```
#!/bin/bash
echo "args: $@"
echo "name: $0"
echo "1st argument: $1"
echo "3rd argument: $3"

$ bash cmd-args.sh one two three
args: one two three
name: cmd-args.sh
 1st argument: one
 3rd argument: three
```

Quotations

- Single quotes (') preserve literal values

```
$ a='hello!'
$ echo '$a'
$a
```

- Double quotes (") preserve literal values, except for: $ ` \

```
$ echo "$a"
hello!
$ echo "$a \" "
hello! "
```


Command substitution

- The output of a command can be used in a script with `$(command)` (or `` `command` ``)
- `$(command)` is substituted with the output of `command` just like a variable would be

```
$ echo "Today is $(date)."
Today is Fri Aug 26 10:58:03 EEST 2011.

$ b=20.1
$ a=$(echo "$b*10.0" | bc -l)
$ echo $a
201.00
```


Arithmetics

- Simple arithmetics performed using:

  `(( expression ))`

  `let "expression"`
- Variable and command expansion done on *expression*
- `$` not needed for variables in *expression*

```
$ a=10              $ let "a += 3"      $ echo $a
$ (( c = a - 4 ))   $ echo $a           $ (( a++ ))
$ echo $c           13                  14
6
```


Arithmetic expansion

- `$(( expression ))` replaced by value of `expression`

```
$ a=10; b=20
$ echo $(( a + b ))
30
$ echo "$a x $b = $(( a * b ))"
10 x 20 = 200
```

- `(( expression ))` used for `if` and `for` constructs

```
$ if (( $DEBUG || $MODE == "d" ))
then
  echo "debug mode ON"
fi
```

Arithmetic operators

Operator	*Meaning*
`+ -`	addition, subtraction
`* / %`	multiplication, division, remainder
`**`	exponentiation
`VAR++ VAR--`	variable post-increment and post-decrement
`++VAR --VAR`	variable pre-increment and pre-decrement
`<= >= < >`	comparison operators
`== !=`	equality and inequality
`&&`	logical AND
`||`	logical OR
`! ~`	logical and bitwise negation
`&`	bitwise AND
`^`	bitwise exclusive OR
`|`	bitwise OR
`<< >>`	left and right bitwise shifts
`expr ? expr : expr`	conditional evaluation
`= *= /= %=`	assignment operations
`+= -= <<= >>=`	
`&= ^= |=`	

Conditional if statements

- Basic form:

```
if test-commands
then
  do-stuff
elif test-commands
then
  do-stuff
else
  do-stuff
fi
```

```
$ if [[ 1 -eq 1 ]]
then
   echo A
else
   echo B
fi
A
```

```
$ [[ 1 -eq 1 ]] && echo A || echo B
A
```


Test commands

- Four ways to do this (yes really…)
  - `test` command or `[ ]` single brackets
  - `[[ ]]` double brackets
  - `(( ))` arithmetic expansion
- Double brackets are safer than `test` or single brackets
- Operators for strings, integers, files and Boolean algebra

[[ ]] – file test operators

`[[ -e name ]]`	file exist
`[[ -f name ]]`	is a regular file
`[[ -d name ]]`	is a directory
`[[ -s name ]]`	is size of file not zero
`[[ -r name ]]`	file has read permission
`[[ -w name ]]`	file has write permission
`[[ -x name ]]`	file has execute permission

…

[[ ]] – string operators

```
[[ "s1" == "s2" ]]   string equality
[[ "s1" != "s2" ]]   string inequality
[[ "s1" < "s2" ]]    string lexiographic before
[[ "s1" > "s2" ]]    string lexiographic after
[[ "s1" =~ "s2" ]]   regular expression match
[[ -z "s1" ]]        string has zero length
[[ -n "s1" ]]        string has non-zero length
```


[[ ]] – integer operators

```
[[ 1 -eq 2 ]]   equality
[[ 1 -ne 2 ]]   non-equality
[[ 1 -lt 2 ]]   less than
[[ 1 -gt 2 ]]   more than
[[ 1 -le 2 ]]   less than or equal
[[ 1 -ge 2 ]]   greater than or equal
```


[[ ]] – Boolean algebra

```
[[ A || B ]]   (logical) A or B
[[ A && B ]]   (logical) A and B
[[ ! A ]]      not A
```


Conditional example

```
xor.sh:
#!/bin/bash
if [[ ($1 != $2) && \
   ( $1 || $2 ) ]]
then
   echo True
else
   echo False
fi
```

backslash can be used to continue the command on the next line

```
$bash xor.sh 1 0
True
$bash xor.sh 1 1
False
```

Case

- Branch code by matching the value of a variable against expression(s)

```
case "$variable" in
  expr1)
    commands...
    ;;
  expr2)
    commands...
    ;;
  *)
    default commands...
    ;;
esac
```

each block terminated with ;;

always true!

Case example

case.sh:

```
#!/bin/bash
DEBUG=0
case "$1" in
   -d|--debug)
      DEBUG=1
      ;;
   [0-9]*)
      echo "number"
      ;;
   [[:upper:]])
      echo "uppercase letter"
      ;;
   *)
      echo "something else"
      ;;
esac
(( $DEBUG )) && echo "debug mode ON"
```

```
$ bash case.sh Z
uppercase letter
$ bash case.sh 77
number
$ bash case.sh f!
something else
$ bash case.sh --debug
debug mode ON
```


Loops

- `for` loops over words in a string

```
for name in Alice Bob Charlie
```

- … or over numerical indeces

```
for ((i=0; i<10; i++)) HUOM: {0..10}
```

- `continue` jumps to next iteration
- `break` steps out of the loop
- `while`, `until`

Loops – over words

loop.sh:

```
#!/bin/bash
for name in Alice Bob "Charlie D"
do
  echo "Who? $name"
done

$ bash loop.sh
```

```
Who? Alice
Who? Bob
Who? Charlie D
```

Loops – over files

```
loop-files.sh:
#!/bin/bash
for f in *.sh
do
  l=$(wc $f | awk '{ print $1 }')
  echo "$f has $l lines"
done
```

```
$ bash loop-files.sh
loop-args.sh has 7 lines
loop-files.sh has 8 lines
…
```


Loops – over arguments

```
loop-args.sh:
#!/bin/bash
for arg in "$@"
do
  echo $arg
done
```

quotes needed to handle word splitting correctly!

```
$ bash loop-args.sh one two "3 4"
one
two
3 4
```


Loops – over integers

```
loop-int.sh:
#!/bin/bash
for ((i=0; i<4; i++))
do
  echo part-$i
done
```

```
$ bash loop-int.sh
part-0
part-1
part-2
part-3
```


Loops – while

```
loop-while.sh:
#!/bin/bash
i=0
while [[ $i -lt 4 ]]
do
  echo part-$i
  let "i += 1"
done
```

```
$ bash loop-while.sh
part-0
part-1
part-2
part-3
```

Functions

- (Limited) support for functions

```
funcs-hello.sh:
#!/bin/bash
hello () {
  echo "Funky hello!"
}
echo "calling function hello() ..."
hello
```

declaration

call

```
$ bash funcs-hello.sh
calling function hello() ...
Funky hello!
```

- Declaration must be before call!

Variables – scope

- bash has three scopes for variables:
  local, global and environment variables
- Local variables are local to current codeblock
  - set with: `local var=value`
  - use local variables in functions!
- Global variables are global in current shell
  - set with: `var=value`

Variables – scope

- Environment variables are inherited by all launched sub-shells
  - Set with: `export var=value`
  - Changes in value are not passed back!
  - By convention, ALL_CAPS used for names

Variables – scope

```
variables.sh:
#!/bin/bash
scopetest() {
   local lvar=20
   echo "---in function---"
   echo "lvar: $lvar"
   echo " var: $var"
   echo "EVAR: $EVAR"
}
var=30
export EVAR=40
scopetest
echo "---in main scope---"
echo "lvar: $lvar"
echo " var: $var"
echo "EVAR: $EVAR"
bash print-variables.sh
```

```
print-variables.sh:
#!/bin/bash
echo "---in next shell---"
echo "lvar: $lvar"
echo " var: $var"
echo "EVAR: $EVAR"
```

Variables – scope

```
$bash variables.sh
---in function---
lvar: 20
 var: 30
EVAR: 40
---in main scope---
lvar:
 var: 30
EVAR: 40
---in next shell---
lvar:
 var:
EVAR: 40
```

undefined!

Functions – input

```
funcs-input.sh:
#!/bin/bash
addup() {
  local total=0
  while [[ $# -gt 0 ]]
  do
    case "$1" in
      [0-9]*)
        let "total += $1"
        ;;
      *)
        echo "invalid argument: $1" >&2
        ;;
    esac
    shift
  done
  echo "Grand total: $total"
}
addup 4 6 12
addup 3 8 foo
```

- Input parameters work like command-line arguments

```
$ bash funcs-input.sh
Grand total: 22
invalid argument: foo
Grand total: 11

$ bash funcs-input.sh 2>
/dev/null
Grand total: 22
Grand total: 11
```


Functions – output

- No direct way to get output
- Return value of function is
  - 0 success
  - nonzero failure
- Options are
  - Global variable (not recommended but fast)
  - Command substitution, i.e. echo output (slow?)
  - Output variable name passed as input

Functions - output

```
# set global
out_a(){
  val_a=$1
}
# command subst
out_b(){
  echo $1
}
# pass output var name
out_c(){
  eval $1='$2'
}
```

```
$ out_a 123
$ echo $val_a
123
$ val_b=$(out_b 123)
$ echo $val_b
123
$ out_c val_c 123
$ echo $val_c
123
```

`eval` will evaluate the line after shell expansions

Here documents

- Pipe an arbitrary length text block spanning over multiple lines to a command

```
$ command << END
... multiple lines
of text ...
END
```

- Same as `command < file` where `file` includes the input above
- `END` is an arbitrary tag that marks the end of input

Here documents

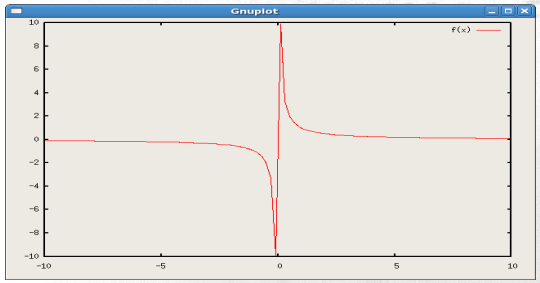
heredoc.sh:

```
cat << EndBlock
A here document can be used to print out
 instructions using cat. It can also be used to
 steer an interactive program as shown below...
EndBlock

gnuplot << EOF
f(x)=1/x
plot f(x)
pause 2
EOF
```


Here documents

```
$ bash heredoc.sh
A here document can be used to print out
 instructions using cat. It can also be used to
 steer an interactive program as shown below...
```



Further information

- Online resources
  - Bash guide for Beginners
    http://tldp.org/LDP/Bash-Beginners-Guide/html/
  - Advanced Bash-Scripting Guide
    http://tldp.org/LDP/abs/html/
  - BASH FAQ
    http://mywiki.wooledge.org/BashFAQ
  - GNU Bash Reference Manual
    http://www.gnu.org/software/bash/manual/bashref.html

Matching Text

- A number of Unix text-processing utilities let you search for, and in some cases change, text strings.
  - These utilities include the editing programs `ed`, `ex`, `vi` and `sed`, the `awk` programming language, and the commands `grep` and `egrep`.
- *Regular expressions*, or *regexes* for short, are a way to match text with patterns.
- Regular expressions are a pattern matching standard for string parsing and replacement.

The Most Simple Regex

- In it's simplest form, a regular expression is a string of symbols to match "as is".

Regex	Matches
abc	abcabcabc
234	12345

- A simple example:
  `$ grep '234'`

Quantifiers

- To match several characters you need to use a quantifier:
  - * matches *any number* of what's before it, from zero to infinity.
  - ? matches *zero or one* of what's before it.
  - + matches *one or more* of what's before it.

Regex	Matches
23*4	1245, 12345, 123345
23?4	1245, 12345
23+4	12345, 123345

- A simple example:
  `$ grep '23*4'`

Basic Regexes vs. Extended Regexes

- The *Basic Regular Expressions* or *BRE flavor* standardizes a flavor similar to the one used by the traditional UNIX grep command.
  - Only supported metacharacters are `.` (dot), `^` (caret), `$` (dollar), and `*` (star). To match these characters literally, escape them with a `\`.
  - Some implementations support `\?` and `\+`, but they are not part of the POSIX standard.
- Most modern regex flavors are extensions of the *ERE flavor*. By today's standard, the POSIX ERE flavor is rather bare bones.
- We will be using extended regexes, so:
  `$ alias grep='grep --color=auto -E'`

Regexes Are Hoggish

- By default, regexes are greedy. They match as many characters as possible.

Regex	Matches
2	122223

- You can define how many instances of a match you want by using ranges:
  - `{m}` matches only *m* number of what's before it.
  - `{m,n}` matches *m* to *n* number of what's before it.
  - `{m,}` matches *m* or more number of what's before it.

Special Characters

- A lot of special characters are available for regex building. Here are some of the more usual ones:
  - `.` the dot matches any single character.
  - `\w` matches an alphanumeric character, `\W` a non-alphanumeric.
  - `\` to escape special characters, e.g. `\.` matches a dot, and `\\` matches a backslash.
  - `^` matches the beginning of the input string.
  - `$` matches the end of the input string.

Special Character Examples

Regex	Matches	Does not match
1.3	1234, 1z3, 0133	13
1.*3	13, 123, 1zdfkj3	
\w+@\w+	a@a, email@oy.ab	,.-!"#€%&/
^1.*3$	13, 123, 1zdfkj3	x13, 123x, x1zdfkj3x

Character Classes

- You can group characters by putting them between square brackets. This way, any character in the class will match *any one* character in the input.
  - `[abc]` matches any of a, b, and c.
  - `[a-z]` matches any character between a and z.
  - `[^abc]` matches anything other than a, b, or c.
    - Note that here the caret ^ at the beginning indicates "not" instead of beginning of line.
  - `[+*?.]` matches any of +, *, ? or the dot.
    - Most special characters have no meaning inside the square brackets.

Character Class Examples

Regex	Matches	Does not match
[^ab]	c, d, abc, sadvbcv	a, b, ab
^[1-9][0-9]*$	1, 45, 101	0123, -1, a1, 2.0
[0-9]*[,.]?[0-9]+	1, .1, 0.1, 1,000, 0,0,0.0	

Grouping and Alternatives

- It might be necessary to group things together, which is done with parentheses (and).

Regex	Matches	Does not match
(ab)+	ab, abab, aabb	aa, bb

  - Grouping itself usually does not do much, but combined with other features turns out to be very useful.
- The OR operator | may be used for alternatives.

Regex	Matches	Does not match
(aa|bb)+	aa, bbaa, aabb	abab

Subexpressions

- With parentheses, you can also define subexpressions to store the match after it has happened and then refer to it later on.

Regex	Matches	Does not match
(ab)\1	ababcdcd	ab, abcabc
(ab)c.*\1	abcabc, abcdefabcdef	abc, ababc

- You can store up to nine matches and refer back to the matches using \1, \2,… \9 notation.

Some Practical Examples

- Check for a valid format for email address:
  `$ grep '[A-Za-z0-9_-][A-Za-z0-9_.-]*[^.]@[A-Za-z0-9][A-Za-z0-9.-]+\.[A-Za-z]{2,}'`
  - `[A-Za-z0-9_-][A-Za-z0-9_.-]*[^.]` matches a positive number of acceptable characters not starting or ending with dot.
  - `@` matches the @ sign.
  - `[A-Za-z0-9][A-Za-z0-9\.-]+` matches any domain name, incl. dots.
  - `\.[A-Za-z]{2,}$` matches a literal dot followedby two or more characters at the end.
- Check for a valid format for Finnish social security number:
  `$ grep '[0-9]{6}[+-A][0-9]{3}[A-Z0-9]'`

The Stream Editor – sed

- Sed is a non-interactive – or ***s***tream oriented – ***ed***itor.
- Sed comes standard with every POSIX-compliant Unix.
- Mastering sed can be reduced to understanding and manipulating the four spaces of sed. These four spaces are:
  - Input stream
  - Pattern space
  - Hold buffer
  - Output Stream

How sed Works

- Sed reads data from the *input stream* until it finds the newline character `\n`.
- Then it places the data read so far, without the newline, into the *pattern space*. Most of the sed commands operate on the data in the pattern space.
  - The *hold buffer* is there for your convenience. You can copy or exchange data between the pattern space and the hold buffer.
- Once sed has executed all the commands, it outputs the pattern space to *output stream* and adds a newline character `\n` at the end.

Syntax of Sed Commands

- Sed commands have the general form:
  `[address [,address] [!]] command [arguments]`
  - The optional *addresses* specify to which input lines the command will be applied to, the default is every line.
  - The *command*s consists of a single letter or symbol.
  - The *arguments* are optional and only a few commands accept them.

A Simple sed Program

- This program replaces text "foo" with "bar" on every line:
  `$ sed 's/foo/bar/'`
- Sed opens the file as the input stream and starts reading the data.
  - After reading the first line it finds a newline `\n` and it then places the text read in to the pattern space without the newline.
- Next sed applies the `s/foo/bar/` command for the pattern space.
  - If there is no "foo" anywhere, sed does nothing to the pattern space.
  - If there is "foo", it will be replaced by "bar".
- The default action when all the commands have been executed is to print the pattern space, followed by a newline.

Options -n and -e, and Command p

- If you specify option `-n` then sed will no longer print the pattern space when it reaches the end of the script:
  `$ sed -n 's/foo/bar/'`
  - If you run this program, there will be no output.
- You now must use sed's `p` command to force sed to print the line:
  `$ sed -n 's/foo/bar/; p'`
  - Sed commands are separated by the semicolon `;` character.
  - You can also use the option `-e` to separate the commands:
    `$ sed -n -e 's/foo/bar/' -e 'p'`

Danger: the Option -i

- Option `-i` forces sed to do in-place editing of the file:
  `$ sed -i 's/foo/bar/' file`
  - This means sed reads the contents of the *file*, executes the commands, and places the new contents back into the *file*.
- Be very careful when using `-i` as it's destructive and it's not reversible!
  - It's always safer to run sed without `-i`, and then replace the file yourself.

Addresses

- The simplest form of an *address* is a single number that limits sed commands to the given line number:
  `$ sed -n '5p'`
  - Only the 5th line of the input stream will be printed.
- The addresses can be inverted with the `!` after the address:
  `$ sed -n '5!p'`
  - Print all but 5th line.
- You can also limit sed commands to a range of lines by specifying two numbers, separated by a comma:
  `$ sed -n '5,10p'`
  - Print only lines 5 – 10, inclusive.

Regular Expressions as Addresses

- There is also a *regular expression address* match, `/regex/`.
- If you specify a regular expression as an address, then the command will only get executed on those lines matching the regex:
  `$ sed -n '/-/p'`
  - Print only lines containing hyphen (or minus) character `-`.
- You can also use expressions to match a range between two regexes:
  `$ sed -n '/May/,/Aug/p'`
  - This matches all lines between the first line that matches "May" regex and the first line that matches "Aug" regex, inclusive.

The Last Line and Mixing Address Types

- The special address `$` matches the last line of the input stream:
  `$ sed -n '$p'`
  - Prints only the last line of the input stream.
- Addresses can also be combined:
  `$ sed -n '5,$p'`
  - From 5th line to the end of file.
  `$ sed -n '/regex/,$p'`
  - From line matching *regex* to the end of file.
  `$ sed -n '5,/regex/p'`
  - From 5th line to the line matching *regex*.

Command Grouping

- With *command grouping* `{…}` it is possible to apply one set of commands to specific lines and another set of commands for all the rest of the input lines.
  - It says, execute all the commands in `{…}` on the line(s) that matches the restriction.
- E.g. to print the line after *regex*, but not the line containing the *regex*:
  `$ sed -n '/regex/{n; p}'`
  - The `n` command will empty the current pattern space and read in the next line of input.

The Hold Buffer

- With hold buffer you can save the current line to the hold buffer, and then let sed read in the next line.
- The command for copying the current pattern space to the hold buffer is `h` and the command for copying the hold buffer back to pattern space is `g`.
  - The command for exchanging the contents of the hold buffer and the pattern space is `x`.
- E.g. to print the line before the line that matches *regex*:
  `$ sed -n '/regex/{x; p; x}; h'`

Text Substitution

- The *substitute* command `s///` is used to find and replace text:
  `$ sed 's/foo/bar/'`
  - Change the first occurrence of "foo" with "bar".
- Substitute the 4th occurrence of "foo" with "bar" on each line:
  `$ sed 's/foo/bar/4'`
  - With a numeric flag like `/1`, `/2`, etc. only that occurrence is substituted.
- Substitute all occurrences of "foo" with "bar" on each line:
  `$ sed 's/foo/bar/g'`
  - With global flag `/g` set, substitute command does as many substitutions as possible, i.e., all.

Find and Replace vs. Substitute

- You can use any regex to match (i.e. find) the text you want:
  `$ sed 's/\(.*\)foo/\1bar/'`
  - Replace only the last occurrence of "foo" with "bar".
  - Note that sed uses basic regular expressions, hence `\(` and `\)`!
    - You can use extended regexes with the `-r` option:
      `$ sed -r 's/(.*)foo/\1bar/'`
- You can also refer back to the matched string with the `&` character:
  `$ sed 's/^\(.*\)$/(&)/'`
  - Adds parenthesis around the line.

Translate Characters

- The *translate* command `y/src/dst/` does transliteration, a 1:1 mapping of symbols in *src* to symbols in *dst*:
  `$ sed 'y/abc/xyz/'`
  - Change every instance of *a* to *x*, *b* to *y*, and *c* to *z*.
- The command always acts on the whole pattern space so you can't use regexes to match the parts of the line you would want to.
  - You can still use addressing to limit the command to specific lines:
    `$ sed '/regex/,$y/src/dst/'`
    - Translate only from the first line matching *regex* to the end of the file.

Deleting Text

- The d command
  - deletes the current pattern space;
  - reads in the next line into the pattern space; and
  - aborts the script execution and starts the execution at the first sed command.

`$ sed '/^$/d'`

- Deletes all blank lines from the input stream.

`$ sed 'n; n; n; d'`

- Deletes every 4th line.

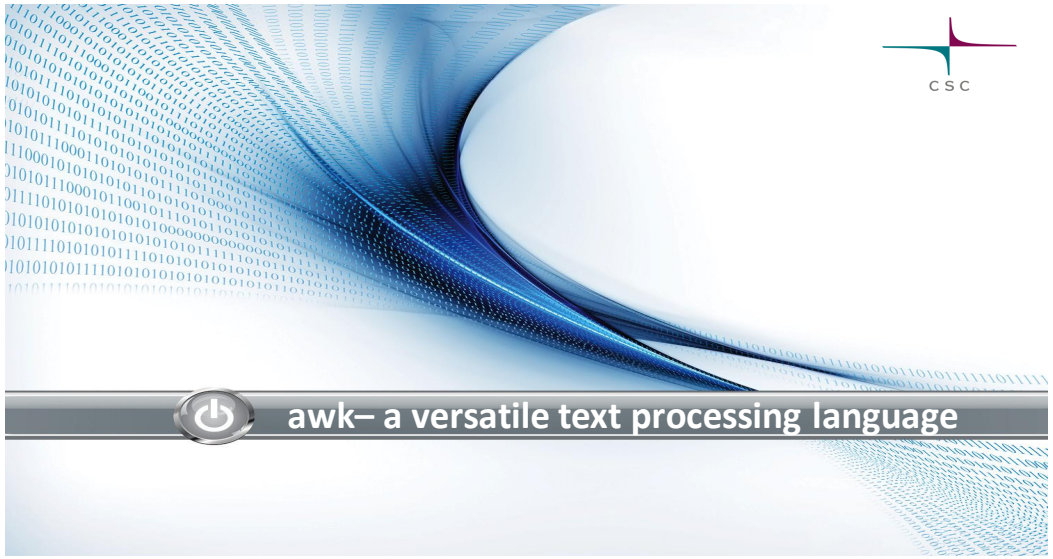
Labels and Branching

- The command `:name` creates a named label *name*, which you can branch to with the `b` command:
  `$ sed ':a $s/\n/ /g; N; ba'`
  - The `:a` command creates a named label *a*.
  - The `$` restricts the next command only to the last line of input.
  - The `s/\n/ /g` substitutes all newline character to space characters.
  - The `N` command first appends a newline to the current pattern space and then appends the next input line.
  - The `ba` command branches back to the label *a*.
  - Effectively, this will join all input lines into a single line.

Sed Is Good for…

- A *Turing machine* is a hypothetical machine thought of by the mathematician Alan Turing in 1936. Despite its simplicity, the machine can simulate ***any*** computer algorithm, no matter how complicated it is!
  - Sed has been demonstrated[1] to be Turing complete so in theory, you can use sed for any computational problem:
    `$ sed -f sedtris.sed`
    `$ sed -nf arkanoid.sed`
- Sed is mainly used for its text substitution capabilities.

[1] Blaeess, Christophe, Implementation of a Turing Machine as Sed Script, http://www.catonmat.net/ftp/sed/turing.txt



awk – text processing

- Developed at Bell Labs in 1977 by

  Aho (not Esko, but Alfred Vainö!), **W**einberger, **K**ernighan
- A versatile scripting language which resembles C (surprise! - Kernighan & Ritchie)
- Powerful with spread-sheet / tabulated data
- Typical usage perhaps in one-liners with matching/reordering/formatting/calculating fields from the existing tables of data
- awk command scripting is also available

awk line command

- To print a certain column (**$2** refers to 2nd column in row – will be explained later), type the following to the terminal

  ```
  $ awk '{print $2}' /etc/netconfig
  ```

  – by default you assume that the file is separated by blank spaces
- You can redirect the output (using the > symbol) to store the result into a new file

  ```
  $ awk '{print $2}' /etc/netconfig > netconfig2ndcolumn.txt
  ```
- You can also use it within a pipe (feeding it with stdout)

  ```
  $ cat /etc/netconfig | awk '{print $2}'
  ```


awk-scripts

- You can save your awk-directives in a text file (a.k.a. script). Why?
  - Sometimes one-liners get too long
  - You want to be able to easily reproduce your awk-command
  - May be useful if you need to declare user defined functions through command scripts
  - Not mandatory, but useful to give suffix **.awk**
  - Triggered by option **-f**
- Can be used in connection with redirected output:

  ```
  $ awk -f myscript.awk inputfile.txt > outputfile.txt
  ```

awk pattern

- awk commands essentially match a pattern from a text and apply an action to it:

```
!/ pattern /   { action }
```

(the exclamation mark inverts match)

- For example, we want to print all relevant lines in **/etc/netconfig**, i.e., exclude all commented lines that start with **#**

```
$ awk '!/#/' /etc/netconfig
```

  - Or the 2nd column (action) of all relevant lines:

```
$ awk '!/#/ {print $2}' /etc/netconfig
```


awk pattern ctd.

- Another example given by a script to display all nologin-accounts in the system (save into file *slide3.awk*):

```
BEGIN {x = 0}
/nologin/ {x = x + 1; print x," ...", $1}
END {print "------------------";print "nologins=",x}
```

- Use **-f** option to launch the script

```
$ awk  -f slide3.awk /etc/passwd
```

- Short exercise for the audience: Change the script such that all users with not nologin accounts are shown

Pre- and post-processing steps

- **BEGIN { }** and **END { }** statements are optional in awk and if present, they execute code before and after reading the input
- They are *not* tested against the input
- **BEGIN** is often used to initialize variables before the first input line has been read in
- **END** is usually used to print some summary information after input has been finished

Field separator

- Field separator (**FS**), the same as **-F** option, can be used to indicate character(s) used to separate consecutive fields:

```
$ awk  -F: -f slide3.awk /etc/passwd
```

- If you do not want to use the **–F** option, define inside the script

```
BEGIN { FS="[:,]" }
```

- Your **FS** is either colon or comma, try for instance (**NF** is number of columns – see next slide):

```
$ echo "0 1:2,3 4" | awk  -F"[:,]" '{ print NF " last
  column: "  $NF}'
```

or with blank or colon

```
$ echo "0 1:2,3 4" | awk  -F"[: ]" '{ print NF " last
```

Record separator – RS

- Similar to **FS**, the record separator (**RS**) can be used to turn any character(s) into line breaks (=new rows)
- There is no command line option for **RS** that can be passed
- The following prints out not 1, but 4 lines:

```
$ echo "AA,BB:CC;DD" |awk
  'BEGIN{RS="[,:;]"}{print}'
```


Counters of columns and rows

- **NF** is the number of fields on each line (# columns in row)

```
$ awk  -F: '{for (i=1; i<=NF; i++) print i,$i; printf
  "\n"}' /etc/passwd
```

- **NR** is the number of input records (lines)

```
$ awk  'END {print NR} ' /etc/passwd
```

Much simpler still : `$ wc -l /etc/passwd`

- awk fields are accessed through variables **$1 , $2, …, $(NF-1), $(NF)**
  - **$0** refers to the whole input row

Counters of columns and rows ctd.

- Print whole line only if number of fields (=columns in row) exceeds 7

```
$ awk '(NF > 7) {print }' /etc/netconfig
```

Try also with NF > 6 and spot the difference

- Print first 7 rows

```
  $ awk '(NR <=7) {print }' /etc/netconfig
```

this is the same as

```
$ head –n 7 /etc/netconfig
```


Print statement in awk

- Instead of using generic **print** in awk, it is possible to use C-language like **printf**
- This gives you a full spectrum of C-like formatting capabilities, e.g.

```
$ date | awk -F"[ :]" '{printf("Time= %2d hours and %2d
minutes\n",$5,$6)}'
```

  - Please do not forget to supply the newline "**\n**" in **printf** ! The generic **print** already adds that for you – automatically

Variables in awk

- awk has predefined variables, user defined variables and arrays
- **Predefined** variables are fields columns (`$1,$2,…`), the whole line (`$0`) or internal variables (kept in capital letters) like `NF, NR, FS, RS`
- **User defined** variables are usually typed in a lowercase to avoid mix-up, e.g. `a, b, tmp`
  - For instance loop counters: `{for (i=1; i<=NF; i++) print $i}`
  - Or string variables: `mytext= "jada, jada"`
- **Variables** are **set** either
  - inside the script
  - or as argument: `$ awk -F: '{ print $n }' n=1 /etc/passwd`

Variables in awk, ctd.

- awk arrays are in fact ***associative arrays***
  - This means the ***index into an array does not have to be an integer number***
- It can be anything from numerical values (even floating point) to character strings, and can be looped through:

```
BEGIN{tmp[15.6]=0; tmp["sanomalehti"]="Iltasanomat"; tmp["Saab"]="car"}
END{for (i in tmp) {print i,tmp[i]}}
```

- Save into **slide12.awk** and run: `$ awk -f slide12.awk`
  - Note: the order in which the array is scanned through is arbitrary
  - In order to see something you have to send an EOF (**Ctrl**+**D**) to stdin

Built-in functions in awk

- Some numerical functions: **`int, exp, log, sin, cos, sqrt,`** …

```
e.g.,    $ for ((x=1; x<=180; x++)); {   echo $x; } |
                    awk '{print $1,
cos($1*3.1415927/180.0)}' > cosine.dat
```

- Some string handling functions: **`substr, match, sprintf, tolower, toupper,`** …

  e.g. changing everything to upper-case,

  `$ awk '{print toupper($0)}' /etc/netconfig`
- Bit manipulation functions: **`and, or, xor, lshift, compl`**…

Control statements

- awk contains **`if-else`** statements for conditional computation

```
    $ awk '{printf "%f", $2; if ($2 > 0) { print
"positive" }  else { print "negative"} }'
```

  - You can add this to the previous cosine-pipeline (or apply to cosine.dat)
- Can also be programmed as a ladder:

```
if(condition1) {action1}; else if (condition2) {action2};…
else {actionN};
```

- Logical operators: or ||; and &&:

```
if ((condition1a || condition1b) && condition2) {action};
```

Further reading

- Please do read awk Unix manual pages :
  - `$ man awk`
  - `$ info awk`
- Web contains a plenty of additional info
  - Do google for instance on "awk tutorial"