



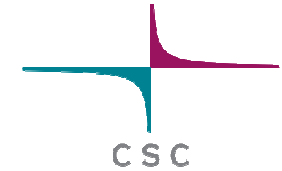
# BASH scripting

Sebastian von Alfthan

Scripting techniques 25.10.2010

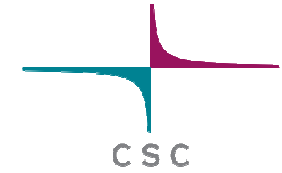
CSC – Tieteen tietotekniikan keskus Oy  
CSC – IT Center for Science Ltd.

# Introduction



- Shell scripting is the art of writing scripts that can automate repetitive tasks
- Here we present BASH and discuss how it can be used as a general programming language
- Command line usage is not the focus of this talk

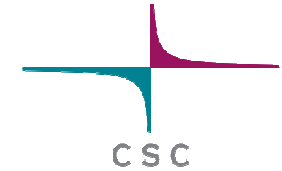
# Different shells



- Bourne shell (SH)
- Korn shell (KSH)
- C shell (CSH)
- Z shell (ZSH)
- And others.....
- BASH (Bourne again shell)
  - SH successor that includes csh and ksh features

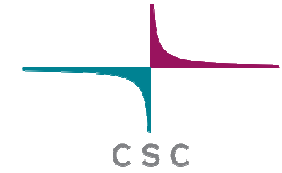


# BASH versions



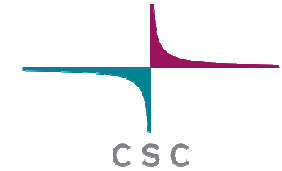
- Show version
  - BASH\_VERSION environment variable
  - /bin/bash –version
- Version 4 (2009)
- Version 3 (2004)
  - Installed on CSC's machines
  - Talk is limited to version 3 features
- Changes: <http://wiki.bash-hackers.org/scripting/bashchanges>

# When to NOT use BASH



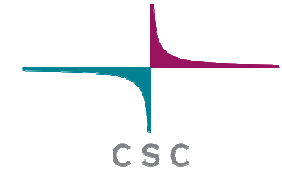
- Resource or compute intensive
- Large scale applications
- Complex datastructures
- Lot's of I/O, need libraries, Floating point arithmetic,...
- Use Perl, Python or even C or Fortran instead

# When to use BASH



- Form a glue for 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

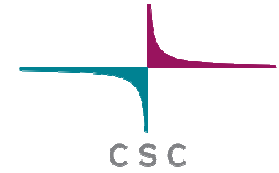
# How to write a script



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

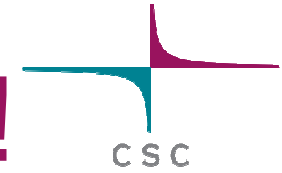
```
>cat hello1.sh
echo "Hello world"
>bash hello1.sh
Hello world
```

# Printing out information



- echo
  - Limited to simple strings
- printf
  - Similar to C printf command
  - Can use “\n”, “\t” and so on
  - Can print value of variables
- Here documents
  - For larger chunks of text, presented later on!

# Executable script, Sha-Bang!



- Add a shebang to the script (man magic) to define the shell that executes it

```
#!/shell-binary
```

```
hello.sh:  
#!/bin/bash  
printf "Hello world\n"
```

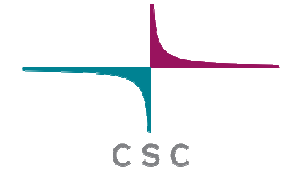
```
>chmod u+x hello.sh
```

```
>./hello.sh
```

```
Hello world
```

↖  
\n is newline, printf does not add it automatically unlike echo

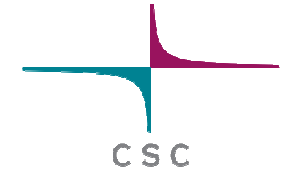
# Comments



- To write comments precede line with #
- Can be added as separate line, or at the end of a line

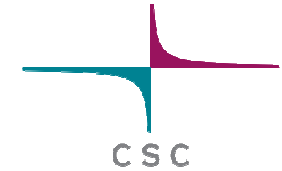
```
>cat hello3.sh
#!/bin/bash
#print out line
printf "Hello world\n" # \n new line
>./hello3.sh
Hello world
```

# Variables



- Like other programming languages Bash has variables
- Variables can be set with `var=value`
- Variable substitution (expansion) gives the value
  - Precede variable name with `$`
  - May need to enclose variable name in `{ }`

# Variables - substitution



```
a= "Hello"  
printf "%s world\n" a  
printf "%s world\n" $a  
printf "%s world\n" ${a}b
```

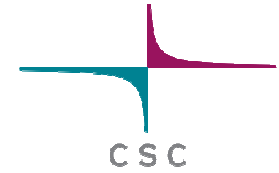
```
a world  
hello world  
hellob world
```

# Variable types

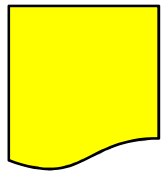


- There are two types of variables
  - Strings
  - Integers (No native floating point type!)
- Additionally variables can be
  - read-only
  - An array (see end of talk)
- By default variables can hold any data we assign to it. Can be limited with “declare”

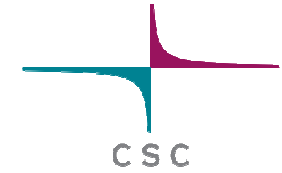
# declare



- Declare integer variable
  - declare -i var=value
- Declare a read-only variable
  - declare -r var=value
- Print the variable attributes
  - Declare -p var

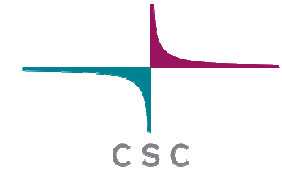


# Command line arguments




- 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



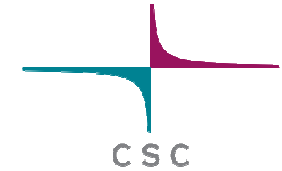
```
cmdargs.sh
#!/bin/bash
printf "arguments:\t%d\n" $#
printf "script name:\t%s\n" $0
printf "first argument:\t%s\n" $1
```

\t tab  
%d integer  
%s string

An arrow points from the first printf statement in the code block to the \t escape sequence in the format string.

```
> ./cmdargs_a.sh Hello!
arguments:          1
script name:       ./cmdargs_a.sh
first argument:   Hello!
```

# “Quotations”

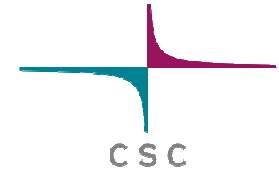


- Single quotes ('): Preserves literal values

```
>a='Hello! '  
>echo '$a'  
$a
```
- Double quotes ("): Preserves literal values, except for \$ ` and \

```
>echo "$a"  
Hello!  
>echo "$a \" \"  
Hello! "
```
- Backtick (`): Replaced by \$( ) in modern bash, see next slide

# Command substitution



- Using command substitution the output of a command substitutes the **\$(command)** construct
- Can be used to store a value in a variable that can be used

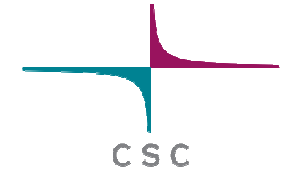
```
>b=20.1
```

```
>a=$(echo "$b*10.2" |bc -l)
```

```
>echo $a
```

```
201.0
```

# Arithmetic expansion

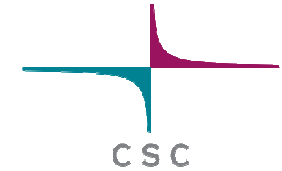


- `$(( expression ))` replaced by value of expression
- Variable & command substitution performed on expression
- `$` not needed for variables inside `()`
- C - like operators supported

; can be used to combine commands on one line

```
>a=10; b=20
>echo $(( a + b ))
30
>echo $(((a+b)/(b-a)))
3
>(( a++ ))
>echo $a
11
>echo $(( a == b ))
0
>echo $(( a < b ))
1
```

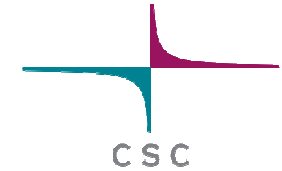
# Arithmetic expansion



- `(( expression ))` can also in itself be used for computation as already seen
- `((expression))` also used for “for” and “if” constructs
- In old bash `$(expression)` was used instead of `$((expression))`, avoid it!

```
>a=10; b=20
>echo $(( a + b ))
30
>echo $(((a+b)/(b-a)))
3
>(( a++ ))
>echo $a
11
>echo $(( a == b ))
0
>echo $(( a < b ))
1
```

# Arithmetic operators



## Operator

## Meaning

VAR++ and VAR--

variable post-increment and post-decrement

++VAR and --VAR

variable pre-increment and pre-decrement

- and +

unary minus and plus

! and ~

logical and bitwise negation

\_\*\*

exponentiation

\*, / and %

multiplication, division, remainder

+ and -

addition, subtraction

<< and >>

left and right bitwise shifts

<=, >=, < and >

comparison operators

== and !=

equality and inequality

&

bitwise AND

^

bitwise exclusive OR

|

bitwise OR

&&

logical AND

||

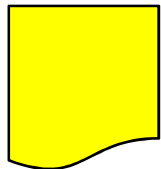
logical OR

expr ? expr : expr

conditional evaluation

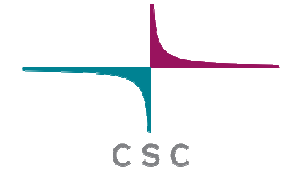
=, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^= and |=

assignments



List from: [http://tldp.org/LDP/Bash-Beginners-Guide/html/sect\\_03\\_04.html](http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_04.html)

# Conditional if statements



- Basic form:

```
if testcommands; then
```

```
    do_stuff
```

```
elif testcommands; then
```

```
    do_stuff
```

```
else
```

```
    do_stuff
```

```
fi
```

```
if [[ 1 -eq 1 ]];
```

```
then
```

```
    echo A
```

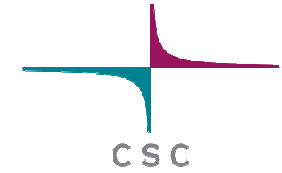
```
else
```

```
    echo B
```

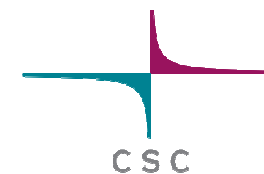
```
fi
```

```
A
```

# Test commands

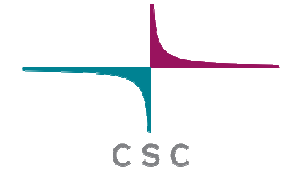


- Four ways to do this (yes really...)
  - test command or [ ] single bracket command
  - **[[ ]] double brackets (recommended!)**
  - (( )) expansion (also ok, but not so common)
- Double brackets [[ is safer than “test” or “[“
- Operators for strings, integers, files and Boolean algebra



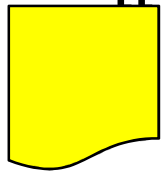
# [[ ]] – file test operators

- [[ -e fname ]] File exist
- [[ -f name ]] Is name a regular file
- [[ -d name ]] Is name 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
- And others.....

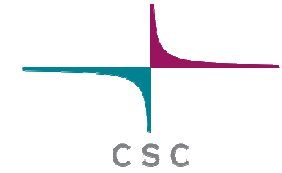


# [[ ]] – string operators

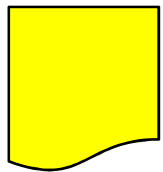
- [[ "s1" == "s2" ]] String equality
- [[ "s1" != "s2" ]] String inequality
- [[ "s1" < "s2" ]] String lexicographic before
- [[ "s1" > "s2" ]] String lexicographic after
- [[ "s1" =~ "s2" ]] Regular expression match
- [[ -z "s1" ]] String has zero length
- [[ -n "s1" ]] String has non-zero length



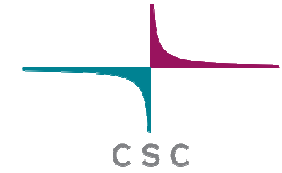
# [[ ]] – integer operators



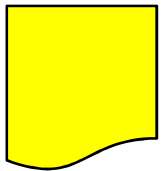
- [[ 1 -eq 2 ]] Equality
- [[ 1 -ne 2 ]] non-equality
- [[ 1 -lt 2 ]] less than
- [[ 1 -gt 2 ]] more than
- [[ 1 -le 2 ]] less than or equal
- [[ 1 -ge 2 ]] less than or equal



# [[ ]] – Boolean algebra



- `[[ a || b ]]`      a or b
- `[[ a && b ]]`      a and b
- `[[ ! A ]]`      Not a





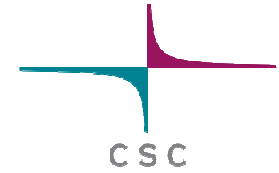
# Conditional example

```
cmdargs.sh
#!/bin/bash
if [[ ($1 == "first") && \
      ( ! $2 == "third" ) ]]
then
    echo True
else
    echo False
fi
```

Backslash (\) followed by  
newline can be used to  
continue command on next  
line

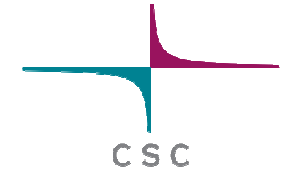
```
> ./cmdargs.sh first second
True
> ./cmdargs.sh first third
False
```

# Loops in bash



- for
  - Can loop over words in a string
  - Can loop over indexes using (( )) construct
  - Use “continue” to jump to next iteration
- while
- until

# Loops – loop over words



```
Loop.sh
#!/bin/bash
for var in "a" "b" "and c"
do
    echo "This is ${var}"
done
```

```
> ./loops.sh
This is a
This is b
This is and c
```

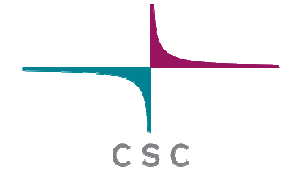
# Loops – over files



```
Loop.sh
#!/bin/bash
for f in $( ls *.sh )
do
    echo $f
done
```

```
> ./loops.sh
loops.sh
hello.sh
...
```

# Loops – over command line arguments



```
Loop.sh
```

```
#!/bin/bash
```

```
i=1
```

```
for v in "$@"
```

```
do
```

```
    printf "argument %d is %s\n" $i "$v"
```

```
    ((i++))
```

```
done
```

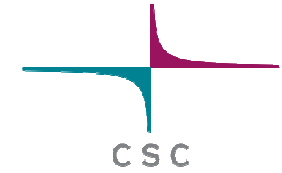
```
> ./loops.sh a "b c"
```

```
argument 1 is a
```

```
argument 2 is b c
```

Quotes “ ” needed to handle word splitting correctly!

# Loops – over command line arguments



```
Loop.sh
```

```
#!/bin/bash
```

```
i=1
```

```
for v in $@
```

```
do
```

```
    printf "argument %d is %s\n" $i "$v"
```

```
    ((i++))
```

```
done
```

```
> ./loops.sh a "b c"
```

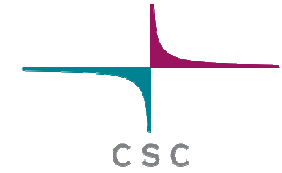
```
argument 1 is a
```

```
argument 2 is b
```

```
argument 3 is c
```

No quotes, b and c seen as separate arguments

# Loops – over integers (c-like)



```
Loop.sh
#!/bin/bash
max=3
for ((i=1;i<=max;i++))
do
    echo $i
Done
```

```
> ./loops.sh
1
2
3
```

Alternative:  
for in \$(seq 1 \$max )

# While loop



```
Loop.sh
#!/bin/bash
max=3; i=1
while [[ $i -le $max ]]
do
    echo $((i++))
done
```

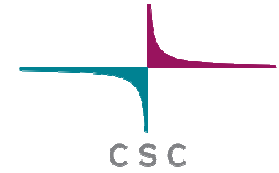
```
> ./loops.sh
```

```
1
```

```
2
```

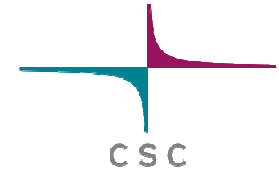
```
3
```

# Arrays



- Bash supports 1D arrays
- Declaring an array
  - By defining an element:  
`myarray[123]=456`
  - Via a list initialization  
`myarray=( 456 4 5 1 )`
  - Using declare:  
`declare -a myarray`
- Elements do not need to be contiguous

# Arrays



- Value can be obtained as before:

```
{myarray[123]}
```

- Remember braces!

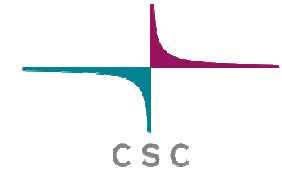
```
>echo $myarray[123]  
[123]
```

```
>echo ${myarray[123]}  
456
```

- One can mix integers and string values

```
myarray=( 123 "onetwothree" )
```

# Arrays



- Special array syntax

- `${myarray[*]}`

All items

- `${!myarray[*]}`

All indexes

- `${#myarray[*]}`

Number of items

- `${#myarray[0]}`

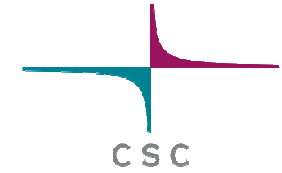
Length of item 0

- `${#myarray[1]}`

Length of item 1

- ...

# Functions – simple case



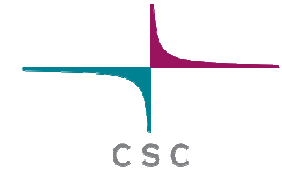
```
Function.sh
#!/bin/bash
function myfunction()
{
echo "In function myfunction"
}
myfunction
```

Declaration

Calling

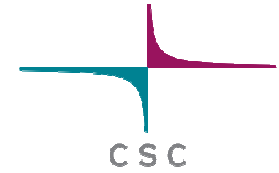
```
> ./function.sh
In function myfunction
```

# Variables -scope



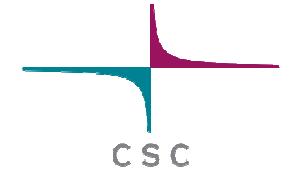
- Bash has three scopes for variables, local, global and environment variables
- Local variables local to current codeblock
  - Set with: `local var=value`
  - **Use local variables in functions!**
- Global variables shown previously
  - Can be set with `var=value`
  - Global in current shell

# Variables



- Exported (environment) variables
  - Can be set with `export var=value`
  - Inherited by all launched bash shells
    - Changes in value are not passed back!
  - Style dictates the variables should be in uppercase

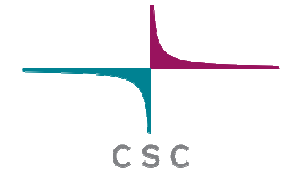
# Variables – scope



```
variable.sh:
function testfunc(){
    local lvar=20
    echo "---In function---"
    echo "lvar:" $lvar
    echo "var:" $var
    echo "EVAR:" $EVAR
}
var=30
export EVAR=40
testfunc
echo "---In main scope---"
echo "lvar:" $lvar
echo "var:" $var
echo "EVAR:" $EVAR
./printvar.sh
```

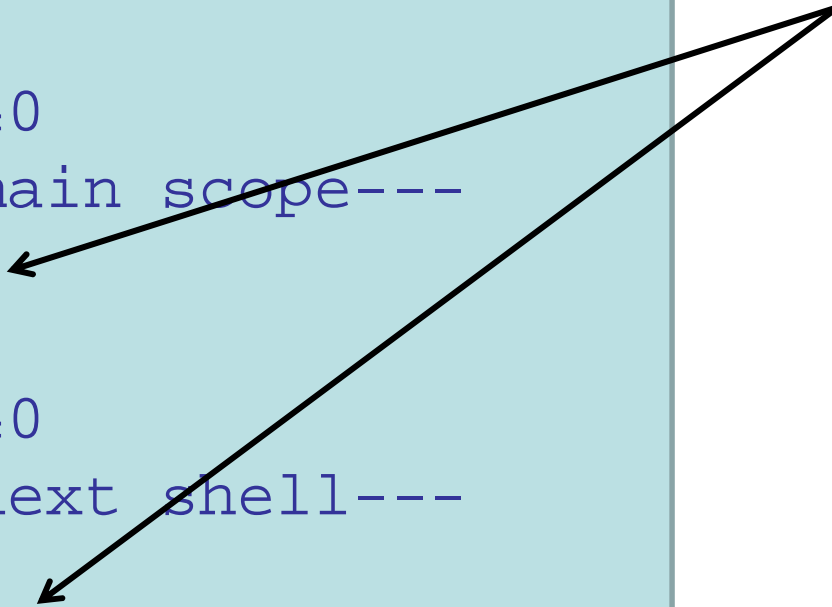
```
printvar.sh:
echo "---In next shell--
-"
echo "lvar:" $lvar
echo "var:" $var
echo "EVAR:" $EVAR
```

# Variables – scope

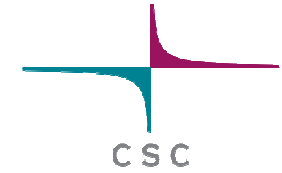


```
>./variables.sh
---In function---
lvar: 20
var: 30
EVAR: 40
---In main scope---
lvar:
var: 30
EVAR: 40
---In next shell---
lvar:
var:
EVAR: 40
```

Undefined



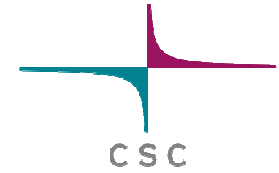
# Functions – input



- Input parameters work in the same way as command line arguments

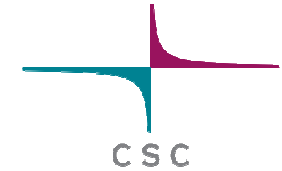
```
variable.sh:
function pars(){
  local i=1
  local v
  for v in $@
  do
    printf "arg %d is %s\n" \
          $i "$v"
    ((i++))
  done
}
pars "first" 2 "third"
```

# 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 (could be slow)
  - Input contains output variable name

# Functions - output



```
#set global
function out_a(){
    val_a=$1
}
#command subst
function out_b(){
    echo $1
}
#input has outputvar
function out_c(){
    eval $1='$2'
}
```

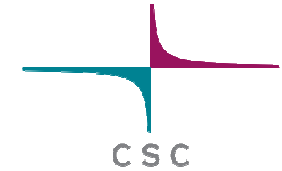
```
out_a 123
echo $val_a #prints 123

val_b=$(out_b 123)
echo $val_b #prints 123

out_c val_c 123
echo $val_c #prints 123
```

Eval will evaluate line a second time after shell expansions

# Here documents



- Special purpose code block

```
command << endblockstring
...
input
...
endblockstring
```
- Same function as

```
command < file
```

where file includes the input above
- endblockstring is arbitrary

# Here documents



```
heredocument.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 << end
```

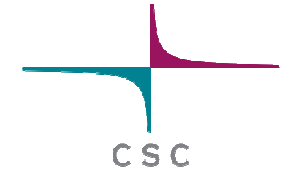
```
f(x)=1/x
```

```
plot f(x)
```

```
pause 2
```

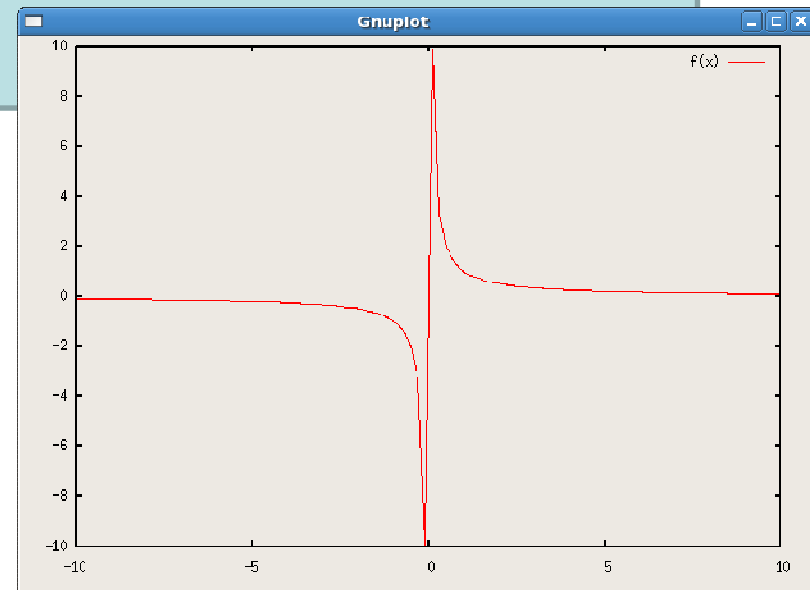
```
end
```

# Here documents



```
./heredocument.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



# Floating point arithmetic



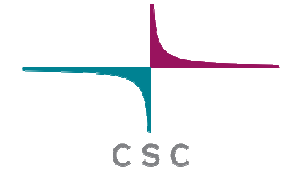
- No floating point variables, can be stored as strings
- Use `bc` or `gawk` for computing
  - Example already shown earlier in command substitution
  - For best performance and accuracy do as much of it as possible in one complete `gawk`, `bc`, ... program

# Further information



- Online free (as in beer) resources
  - Bash guide for Beginners  
<http://tldp.org/LDP/Bash-Beginners-Guide/html/>
  - Advanced Bash-Scripting Guide (good, but a bit dated)  
<http://tldp.org/LDP/abs/html/>
  - BASH FAQ (answers several common cases)  
<http://mywiki.woledge.org/BashFAQ>
  - GNU Bash Reference Manual  
<http://www.gnu.org/software/bash/manual/bashref.html>

# Further information



- Books

- Learning the bash shell (O'Reilly)

- <http://oreilly.com/catalog/9780596009656/>

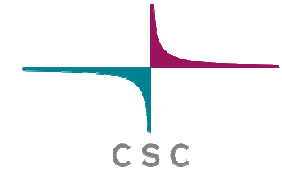
- Bash cookbook (O'Reilly)

- <http://oreilly.com/catalog/9780596526788/>

- Pro Bash Programming: Scripting the GNU/Linux Shell

- <http://cfajohnson.com/books/cfajohnson/pbp/>

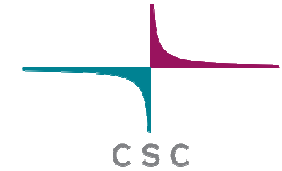
# Exercises



1. Write a “hello world” script “hello.sh” in bash. Make it executable and check that it works. The expected behaviour is given below:

```
> ./hello.sh  
Hello world!  
>
```

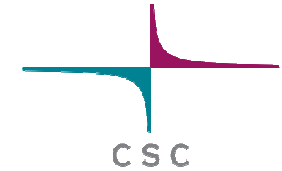
# Exercises



2. Write a script “cmdpar.sh” that takes a number of command line variables and write them out to standard out. If no variables are given then it should write a help message. It should be able to handle different amounts of input

```
> ./cmdpar.sh
Please give me some parameters!
> ./cmdpar one two three
3 parameters
1: one
2: two
3: three
```

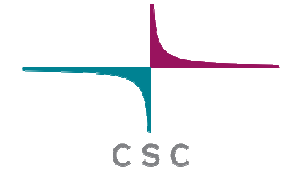
# Exercises



3. Continue to work on `cmdpar.sh`. Put the code that prints out the command line parameters in a function. The script should work as before.

```
> ./cmdpar.sh
Please give me some parameters!
> ./cmdpar one two three
3 parameters
1: one
2: two
3: three
```

# Exercises



4. Continue to work on `cmdpar.sh`. Now we assume that the command line parameters are file names. Print out the number of lines in each file. Also check if the file exists and write out if it does not. Additionally write out the total line count. Tip: The linecount can be computed with “`wc -l file`”, this can be stored in a variable...

```
> ./cmdpar one.txt two.txt three.txt
one.txt has 40 lines
two.txt does not exist
three.txt has 80 lines
-----
120 lines in total
```