# CS214 Advanced UNIX
# Lecture 4

March 2, 2005

## Passing arguments to scripts

When you invoke a command like

```
> cat f1 f2 f3
```

**f1**, **f2**, **f3** are all arguments you pass to the **cat** command. Many times in your script you will also want to be able to run the script with various arguments. Take for example the **wsc.sh** script from the assignment; you can invoke it as

```
>wsc.sh newdatafile
```

and this will have the effect of sorting all the words in **newdatafile**, and then update the file **wordstat.txt** so that it shows all words in alphabetical order and how frequent they are in all the files considered so far, including **newdatafile**. It also makes perfect sense to invoke your script with more than one file; for example

```
>wsc.sh datafile1 datafile2
```

The result should the same as running your script on the first file, and then again on the second file:

```
>wsc.sh datafile1
>wsc.sh datafile2
```

You can actually call **wsc.sh** with any number of files. These files will be passed as arguments to the script. The shell allows you to access the arguments inside the script in very simple way:

- **$#** gives you the number of arguments passed to your script (in the case above, **$#** equals 2)

- **$1** is the value of the first argument, **$2** the value of the second argument, and in general, **$i**, for some $i$ from 1 to **$#**, is the value of the $i$-th argument (in our case, **$1** is **datafile1**, **$2** is **datafile2**)
- **$*** and **$@** both give you all arguments; there is however a difference in the way the shell expands them: "**$***" expands to "**$1 $2...$n**", while "**$@**" expands to "**$1**" "**$2**" ... "**$n**"; we will see that in general it is better to use "**$@**".

It is also useful to know that **$?** is exit code of whatever program was last executed and **$$** is current process id.

Let's look now at two examples of how arguments can be used in your scripts. Suppose for example that you want to write your own calculator; in your script you will perform a number of standard operations. For example, you will want to pass the script two numbers and get back their product. Here is how you can do this:

```
#!/bin/bash/
echo  $(( $1 * $2 ))
```

It is good practice to write some code to deal with the situation when, by mistake, you pass less (or more) then the intended number of arguments:

```
#!/bin/bash/
if [ $# -ne 2 ]
then
  echo "Default 0 * 0 =0"
else
   echo  $(( $1 * $2 ))
fi
```

Here is a script that takes a filename as an argument and does two things: displays it and counts the number of lines in it:

```
#!/bin/bash/
if [ "$#" != 1 ]
then
  echo "Wrong number of arguments!"
  exit 1
else
  read -p "Display file? (y/n) " ans
  if [  "$ans" == "y"  -o  "$ans" == "Y"  -o "$ans" == "yes" -o "$ans" == "Yes" ]
  then
     less $1
     wc -l < $1
  fi
fi
```

# Loops

A **while** loop has the form

```
while cmd
  do
     cmd1
     cmd2...
  done
```

and has the effect of executing the sequence of commands **cmd1**, **cmd2**... as long as command **cmd** is successful (i.e. its exit code is 0). Here is a simple **while** loop that prints all numbers from 1 to 10:

```
i="1"
while [ $i -le 10 ]
 do
     echo "$i "
     i=`expr $i + 1`
 done
```

If you want to execute a sequence of commands until the command **cmd** is successful, use the following construct:

```
until cmd
  do
     cmd1
     cmd2...
  done
```

Here is how you can print all numbers from 1 to 10, this time using an **until** loop:

```
i="1"
until [ $i -ge 11 ]
  do
     echo i is $i
     i=`expr $i + 1`
  done
```

If you want to execute a sequence of commands multiple times, use a **for** loop:

```
for var in string1 string2 ... stringn
  do
       cmd1
       cmd2...
  done
```

Look at the following simple **for** loop:

```
for filename in f1 f2
do
    cat $filename
done
```

It has the same effect as executing the **cat** command twice, first for **f1**, and then for **f2**. You can now write a script that counts the total number of lines in the files passed as arguments:

```
#!/bin/bash/
i="0"
for f in "$@"
  do
       j=`wc -l < $f`
       i=`expr $i + $j`
  done
echo $i
```

Name this file **lcount.sh**. To count the number of lines in all files in your current directory, run

```
>lcount.sh *
```

Notice that this will not descend in subdirectories in your current directory. We will see in the next lecture how to modify your script to do that.

If you are familiar with Latex, you know that you can write a file with the extension **.tex** and then run the command **latex foo.tex** and generate a number of files: a **.dvi** file that can be later on transformed into a **.ps** or **.pdf** file, and can be seen with **.xdvi**, an auxiliary file **foo.aux**, a log file and so on. Latex is very useful for writing articles and presentations, and it is sometimes handy to have a script that will show you what **.tex** files you have in your current directoy, what **.dvi** files, and so on. You can see that such a script is a specialized **ls** utility; call this script **lstex.sh**. You want to run the script with one option **.tex**, **.dvi** and so on, and see as a result all the files of that type in your current directory. In addition, you might want to update each **.tex** file, or transform each **.dvi** file into a **.ps** and so on. Here is how you can write your script (do not worry if you do not understand what each command is doing, just notice how you can do different things for different kind of files, in the same script):

```
!/bin/bash/
#Man entry
# Script that lists different types of files generated by Latex
```

```
#
# syntax: lstex.sh  -opt
# opt may be: tex, dvi, ps, pdf, bib, aux, log,  cls, sty
#End man entry

if [ "$#" -ne "1" ]
then
   echo "1 argument!
          Try -tex, -ps, -pdf, -dvi, -bib, -sty, -cls, -log or -aux"
   exit 1
else
   case "$1" in
    -tex)
      for f in `ls  | grep '\.tex$'`
         do
           echo $f
           read -p "Update? (Y/N) " ans
           if [ "$ans" == "Y" -o "$ans" == "y" ]
           then
              latex $f
              bibtex $f  #update the bibliography
              latex $f   #needed for correct labelling
           fi
         done
      ;;
    -ps)
      for f in `ls  | grep '\.ps$'`
        do
          echo $f
          read -p "Transform to pdf? (Y/N) " ans
          if [ "$ans" == "Y" -o "$ans" == "y" ]
          then
             ps2pdf $f
          fi
        done
      ;;
    -pdf)
      ls -all | grep '\.pdf$'
      ;;
    -dvi)
      for f in `ls  | grep '\.dvi$'`
         do
```

```
            echo $f
            read -p "Transform to ps? (Y/N) " ans
            if [ "$ans" == "Y" -o "$ans" == "y" ]
            then
                read -p "Output file " ans
                dvips $f  -o $ans.ps
            fi
          done
      ;;
    -bib)
      ls  | grep '\.bib$'
      ;;
    -sty)
      ls  | grep '\.sty$'
      ;;
    -cls)
      ls  | grep '\.cls$'
      ;;
    -log)
      for f in `ls  | grep '\.log$'`
        do
          rm -i $f
        done
      ;;
    -aux)
      for f in `ls  | grep '\.aux$'`
        do
          rm -i $f
        done
      ;;
    *)
      echo "Invalid option.
            Try -tex,  -ps, -pdf, -dvi, -bib, -sty, -cls, -log or -aux"
      exit 1
      ;;
   esac
fi
```

Notice that we have used a **case** statement; its general form is

```
case  var in
     p1)
         cmd1
```

```
        ;;
    p2)
        cmd2
        ;;
    ...
    pn)
        cmdn
        ;;
esac
```

Here **p1**, **p2**, up to **pn** are patterns that are first expanded by the shell. The statement has the effect of trying to first match **var** with **p1**; if it succeeds, it executes **cmd1**, if not it tries to match **var** with **p2** and so on.