Alternative OS – Lecture 11

1. **Case** control structure
2. Functions
3. Here document
4. **sed** – the Stream Editor**,** using in shell scripts
5. Conclusions

1. Case control structure

When a choice between several options is to be made, one cane use **if … elif …** too. But it is by far more convenient to use the **case** construction.

Syntax:

case *variable* **in**
pattern1**[**|pattern1a|pattern1b…**]**) commands **;;**
pattern2**[**|pattern2a|pattern2b…**]**) commands **;;**
…….
patternN**[**|patternNa|patternNb…**]**) commands **;;**
*****) commands**;;**
esac

Where patterns are set using *wildcards*.

Here the *variable* is matched against several cases. When match is found the commands that belong to the case are executed. The ***) case** is always matched. The list of the commands can be *empty* for some cases.

Examples:

```
#!/bin/sh
echo "Enter your grade:"
read grade
case $grade in
   Extraordinary ) echo "You get a bonus" ;;
   "Very good" | Good ) echo "You performed very well!" ;;
   Satisfactory ) echo "You passed the test" ;;
   Failed ) echo "You should pass the test one more time";;
   default ) echo "Unknown grade ($grade)" ;;
```

```
esac

#!/bin/sh
while true; do
        echo -n "list the current dir? (y/n) "
         read yn
        case $yn in
                y* | Y* ) ls -l . ; break ;;
                 [nN]* )   echo "skipping" ; break ;;
                q* ) exit ;;
                 * ) echo "unknown response.  Asking again" ;;
        esac
done
```

Exercise: Conceive a program that checks the traffic light and prints what to do.

2. Functions

As soon as your scripts grow sufficiently large, you'll feel need for *functions*. A *function* is a list of command that you can call from different places in your script, just like you call Unix shell commands.

Syntax:

```
#Declaration:
        Function_name()
        {
         command; command
        command
        …….
        }
#Usage:
        Function_name
```

All functions must be *declared* before they are used, at the beginning of the script!

The shell variables $1, $# will store the values corresponding to the arguments of the *function*

Example:

```
#!/bin/sh
help()
{
 echo "Try: $0 -g"
}
ifempty()
{
 if [ $# -eq 0 ]; then echo "No arguments were given $#"; exit
 fi
}

ifempty $@
case $1 in
  -h*|-H*) help;;
  -g) echo "Hello, World!";;
   *) echo "The argument list: $@";;
esac
```

3. Here document

Sometimes you might need to display a block of formatted text. The **echo** command can't help you there. There's a special form of **redirection** that can be used to achieve the goal. It is called **here document.**

Syntax:

```
<<SOME_WORD
The text
        You want
                To display
SOME_WORD
```

If you don't want the variables to be substituted in the text to be displayed, you can *escape* **SOME_WORD** using \:

Syntax:

```
<<\SOME_WORD
The text
        You want
```

To display
SOME_WORD

Example:

```
#!/bin/sh
does=does
not=""
cat << EOF
This here document
$does $not
do variable substitution
EOF
cat << \EOF
This here document
$does $not
do variable substitution
EOF
```

The output produced by running this script would be:

```
Both produce the output:
This here document
does
do variable substitution
This here document
$does $not
do variable substitution
```

4. sed – the Stream Editor

When a shell script is being executed the shell works in the so-called *non-interactive* mode. That means the user is limited in interacting with the shell. He receives the control over the course of actions only when the script allows it, and only as much as the script permits.

Editing task, however, is commonly attributed as an interactive one. Is it possible to edit a file from a shell script? It turns out the answer is positive!

The non-interactive, *stream editor* called **sed** edits the input stream line by line, making the specified changes and sends the result to standard output.

Syntax:

sed *[options] edit_commands [file]*

where *edit_commands* **syntax is:**

[address1[,address2]][function][arguments]

The address entries are optional. They can be separated from the commands with tabs or spaces.

- If both the **address1** and the **address2** are given, then the **address1** specifies the first line and the **address2** specifies the last line to apply the commands.
- If only the **address1** is given then the commands are applied to the lines matching the address.
- If no address is specified then the commands are applied to the whole text.

The address can be given in two forms:

- **Line-number** is a decimal number. The last line number can be specified by the **$** character.
- **Context address** is a regular expression enclosed in slashes /.

The most often command used is the *substitution command*. We shall learn it next.

Syntax:

s/*regular_expression_pattern*/*replacement_string*/*flag*

This function tells the **sed** to find all strings matching the *regular_expression_pattern* and replace them with the

replacement_string. This function must be quoted with single quotes (‘) if additional options are specified or functions.

Example:

```
#! /bin/sh
# converts arg[1].* files to arg[2].*
if [ $# -lt 2 ]
then
 echo 'Usage: rename_base base1 base2';
 exit 1;
fi
echo "Processing basenames  $1 and $2"
for f in $1.*
do
 newfile=`echo $f | sed s/$1/$2/`
 echo "renaming $f to $newfile"
 mv $f $newfile
done
```

5. Conclusions

Today we learnt a new control structure: **case.** This structure allows us to write smaller programs than we could do using only **if** for testing conditions. The advantage of **case** is that it has an arbitrary long list of actions that must take place for each value of some variable is easily implemented. The same result would be rather cumbersome when achieved with **if** only.

Next we learnt how to work with **sed** – the Stream EDitor. This tool let us perform editing tasks from within a script! The most useful function of **sed** is substitution. It is also noteworthy that **sed** supports *regular expressions*. Thus making it a very powerful text processing utility.

We also learnt to use *functions* – the way we group commands so that we can call the whole group from anywhere in the script. This feature saves our time and energy from typing the same lines several times in the program. It is similar to the way *functions* are used in Pascal programming language.

At last, we learnt a new way of displaying *formatted* text from a script. The *here document* construction.