Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Software Systems

Lectures Week 4

Bash 2

Prof. Joseph Vybihal
Computer Science
McGill University

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Week 4 Lecture 1

Bash Functions

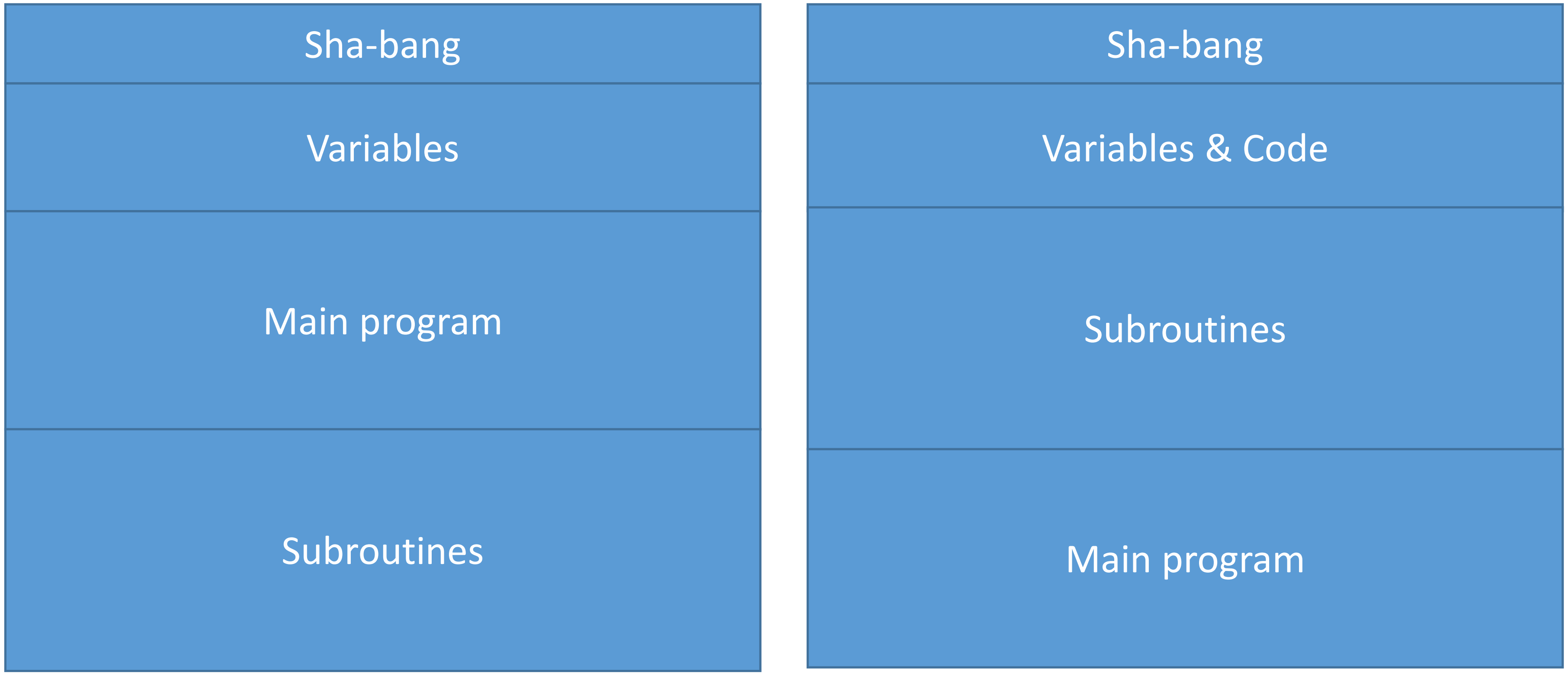
Functions

- Bash functions are incomplete
  - Passing parameters is possible through positional variables
  - Returning results is limited
  - Score rules apply
    - Scope rules work as in other languages
    - Global variables can be used as a means to return values and pass values to a function

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Bash File Structure



Scope Rules

A variable is identified in the following order based on where is was used and where is was created:

1st Block distance

Defined within a control structure

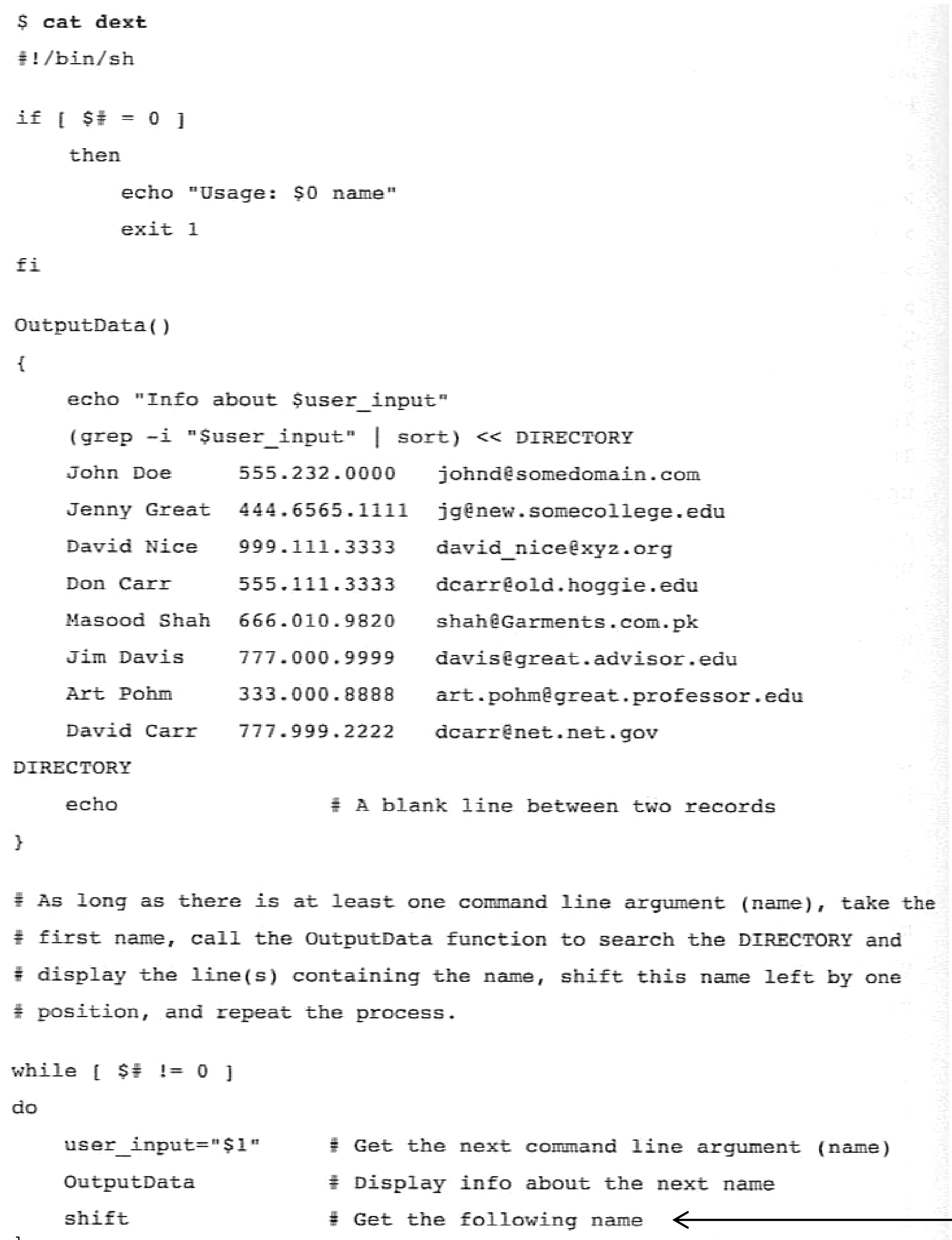
2nd Local distance

Defined within a function

3rd Global distance

Defined at the top of the program

Unix
Bash
C
GNU
Systems



```
$ cat dext
#!/bin/sh

if [ $# = 0 ]
    then
        echo "Usage: $0 name"
        exit 1
fi

OutputData()
{
    echo "Info about $user_input"
    (grep -i "$user_input" | sort) << DIRECTORY
    John Doe      555.232.0000   johnd@somedomain.com
    Jenny Great   444.6565.1111  jg@new.somecollege.edu
    David Nice    999.111.3333   david_nice@xyz.org
    Don Carr      555.111.3333   dcarr@old.hoggie.edu
    Masood Shah   666.010.9820   shah@Garments.com.pk
    Jim Davis     777.000.9999   davis@great.advisor.edu
    Art Pohm      333.000.8888   art.pohm@great.professor.edu
    David Carr    777.999.2222   dcarr@net.net.gov
DIRECTORY
    echo                    # A blank line between two records
}

# As long as there is at least one command line argument (name), take the
# first name, call the OutputData function to search the DIRECTORY and
# display the line(s) containing the name, shift this name left by one
# position, and repeat the process.

while [ $# != 0 ]
do
    user_input="$1"         # Get the next command line argument (name)
    OutputData              # Display info about the next name
    shift                   # Get the following name
done
```

Functions

Notice that the function is in the middle of the program.

Notice that global variables are being used to pass parameters.

Notice how the function is called.

Notice the “shift” command at the end of the while loop.

```
#!/bin/bash
# gloabal x and y
x=200
y=100

math(){
   # local variable x and y with passed args
   local x=$1
   local y=$2
   echo $(( $x + $y ))
}

echo "x: $x and y: $y"
echo "Calling math() with x: $x and y: $y"

# call function
math 5 10

# x and y are not modified by math()
echo "x: $x and y: $y after calling math()"
echo $(( $x + $y ))
```

Notice positional variables used to pass parameters.

Notice the use of the local command to define scope.

Notice the function call.

What do the echo commands print out?

Unix
Bash
C
GNU
Systems

```
#!/bin/sh
adduser()
{
  USER=$1
  PASSWD=$2
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}
## Main script starts here
adduser bob letmein Bob Holness from Blockbusters
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
elif [ "$?" -eq "2" ]; then
   echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

Question

Write a Bash script that asks the user for their age from the main program. It then uses a function to calculate the number of days they have been alive assuming 365 for every year (no leap years). Finally it prints the number of days from the main program.

Returning Values

- You cannot return values from a function
- When a bash function ends its return value is its status: zero for success, non-zero for failure.
- To return values:
    - you can set a global variable with the result, or
    - use command substitution, or
    - you can pass in the name of a variable to use as the result variable.

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Use a global variable

```
myresult=“”

function myfunc()
{
    myresult='some value'
}

myfunc
echo $myresult
```

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Use command substitution

```
function myfunc()
{
   local  myresult='some value'
   echo "$myresult"
}

result=$(myfunc)                    # or result=`myfunc`
echo $result
```

Use eval

```
function myfunc()
{
   local  __result=$1
   local  myresult='some value'
   eval $__result="'$myresult'"
}

myfunc result
echo $result
```

The double underscore variable name convention is used to make it unlikely that the caller used the same variable name when they called the function.

The command eval and the $ symbol acks like a pointer referencing the thing that $1 was pointing to. This results in the value being assigned to the function call argument.

Exit Status

- Functions and Scripts terminate with an Exit Status
  - This is analogous to the exit status returned by a command.
  - The exit status may be explicitly specified by a **return** statement or the **exit** statement, otherwise it is the exit status of the last command in the function or script (0 if successful, and a non-zero error code if not).
  - This exit status may be used in the script by referencing it as $?.
  - This mechanism effectively permits script functions to have a "return value" similar to C functions but the integer value range is limited from 0 to 255.

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Example with **return**

```
max2 ()                    # Returns larger of two numbers.
{                          # Note: numbers compared must be less than 254.
if [ -z "$2" ]; then
  return 255   # we are using the number 255 to mean ERROR
fi

if [ "$1" -eq "$2" ]; then
  return 254   # we are using the number 254 to mean EQUAL
else
  if [ "$1" -gt "$2" ];   then
    return $1
  else
    return $2
  fi
fi
}
max2 33 34

return_val=$?
```

Example with **exit**

```
max2 ()                # Returns larger of two numbers.
{                      # Note: numbers compared must be less than 254.
if [ -z "$2" ]; then
  exit 255   # we are using the number 255 to mean ERROR
fi

if [ "$1" -eq "$2" ]; then
  return 254   # we are using the number 254 to mean EQUAL
else
  if [ "$1" -gt "$2" ];   then
    return $1
  else
    return $2
  fi
fi
}
max2 33 34

return_val=$?
```

The command **exit** not only return the value but it also terminates the script. The returned value is sent to the command-line or the external program that called your script.

A note about **exit**

The exit command is used to terminate a stript, therefore it is used, normally, in two ways:

- As the last instruction in your script returning the error status of your script to the command-line.
  - 0 for all is well. A positive integer number for error.
  - As the developer you determine what meaning is given to each positive integer error status value
- Anywhere within your script as part of an if-statement. The if-statement checks to see if something bad has happened requiring the termination of the script.

Week 4 Lecture 2

Bash Developer Techniques

Techniques

- Shell memory to control script processing
- Backup and Archive scripts
- Development environment setup scripts

Shell memory to control script processing

- Methods of passing data to a script
  - Positional parameters via the command-line
  - Bash read command to prompt user for input
  - Pre-initialized within the shell memory, example:

```
Bash-prompt $ set x=10       OR       setenv x 10
Bash-prompt $ ./script
```

Within the script:

```
#! /bin/bash
if ($x .eq. 10)
   etc.
```

PATH & CLASSPATH

- PATH and CLASSPATH contain paths to folders that can be used by programs:
  - PATH is used by the shell when you type ./script or ./program
  - CLASSPATH is used by Java when you run a Java program

  (at the command-line type set to see the PATH or type
    echo $PATH to print it out)

- It is common for developers to setup the "run-time environment" by defining these kids of variables so that user's can define parameters without editing the code.

Shell memory to control script processing

For example:

- WORKINGDIR – the program's working directory
  - No matter what folder you are in it will default to that working directory
  - Eg:
    - Bash-prompt $ set WORKINGDIR="/home/jack/project1" OR
    - Bash-prompt $ setenv WORKINGDIR "/home/jack/project1"
    - Script: cd $WORKINGDIR
- DATAPATH – the path to important data
  - The user can change this information from the command-line without needing to open up the script to reprogram
  - Eg:
    - Bash-prompt $ set DATAPATH="/home/mary/project1/pictures" OR
    - Bash-prompt $ setenv DATAPATH "/home/mary/project1/pictures"
    - Script: files=$DATAPATH/dog.jpg

Backup and Archive Scripts

It is common that developers want to make backups on a regular basis but get tired of inputting all of the commands each time.

It is common for a developer to have:

- A master backup script
- A project specific backup script

Backup and Archive Scripts

Master backup script

- In the HOME directory
- A simple script listing all the files to backup with an optional variable destination

Project specific backup script

- Similar to the master backup script, but
- Contained in the project's top directory

Backup and Archive Scripts

```
#! /bin/bash
#default destination for backup
destination="../backup"
# check for environment variable
if (test –n "$BACKUPPATH"); then
    destination=$BACKUPPATH
fi
# check for command-line argument
if (test –n "$1"); then
    destination=$1
fi
cp file1 $destination
cp file2 $destination
#etc…
```

Commands to use:
- cp files to destination
- tar files and then mv
- zip files and then mv

Create a long list of everything you want to backup. Add to it and remove from it as things change.

Development environment setup scripts

Good practices:

- Standardizing your technique
- Automating best practices

Developers will often create a script to initiate the environment they want to work under.

- Or, the team leader will create a script that defines the standard working environment for the team members.
- This would include:
  - The project directory structure
  - Backup procedures
  - Start and end project procedures
  - Compiling procedures (we will see later)
  - Repository procedures (we will see later)

Development environment setup scripts

Six common scripts:

• NewProject	- setup the OS environment for the project
• Backup	- setup the way backups will happen
• EnterProject	- the beginning of your work day
• ExitProject	- the end of your work day
• Compile	(we will see later)
• Repository	(we will see later)

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

NewProject Script

```
#! /bin/bash
# Step 1: define the directory structure
 projectPath="projects"
 projectName="/project"
 project=$projectPath$projectName

 if (test –n $1); then
      project=$projectPath$1
 fi
 if (! test –d $projectPath); then
      mkdir $projectPath
 fi
 mkdir $project
```

```
mkdir $project/source
mkdir $project/backups
mkdir $project/docs
mkdir $project/archives
```

NewProject Script

```
# Step 2: create the extra scripts
# if they are stored somewhere then copy otherwise generate them
 if (test –r $projectPath/backup.sh); then
      cp $projectPath/backup.sh $project
      cp $projectPath/enterproject.sh $project         # assuming present
      cp $projectPath/exitproject.sh $project          # assuming present
      exit
 fi
# otherwise generate these scripts
 echo “#! /bin/bash” > backup.sh
 echo “cp *.sh $project/backup” >> backup.sh
 echo “exit 0” >> backup.sh
# create the other scripts as needed
```

Enter Project Script

Things you may want to standardize at the beginning of your workday on a project:

- Make sure you have the latest version of the source code from your team members
- Check to see if there are any special instructions from someone
- Define shell variables (if needed)
- Setup shell environment (like: alias and prompt)

Enter Project Script

Things you may want to standardize at the beginning of your workday on a project:

- Make sure you have the latest version of the source code from your team members
  - GIT commands (we will see later)
- Check to see if there are any special instructions from someone
  - chrome https://www.hotmail.com
  - cat $project/todayreadme.txt
- Define shell variables (if needed)
  - setenv workingdir "/home/jack/testing"
- Setup shell environment (like: alias and prompt)
  - alias ll ls –l
  - export PS1 "I am the greatest>"

The Alias and PS1 options

The alias command

- Lets you rename commands
- Syntax:
  - alias NEWNAME OLDEXPRESSION
- Example:
  - alias dir ls -l-a

The PS1 Shell Variable

- Lets you redefine the prompt. (Check it out, type SET)
- Different shell versions: PS1, PS2, prompt
- Syntax:
  - set prompt=STRING
  - export PS1 STRING

Exiting a project

Things you may want to standardize at the end of your workday:

- Make sure to backup everything
- Make sure to update the repository
- Clear all environment variables
- Change you working directory to HOME

Exiting a project

Things you may want to standardize at the end of your workday:

- Make sure to backup everything
  - backup.sh
- Make sure to update the repository
  - GIT (we will see later)
- Clear all environment variables
  - setenv workingdir “”
- Change your working directory to HOME
  - cd $HOME   # if this is defined by your shell

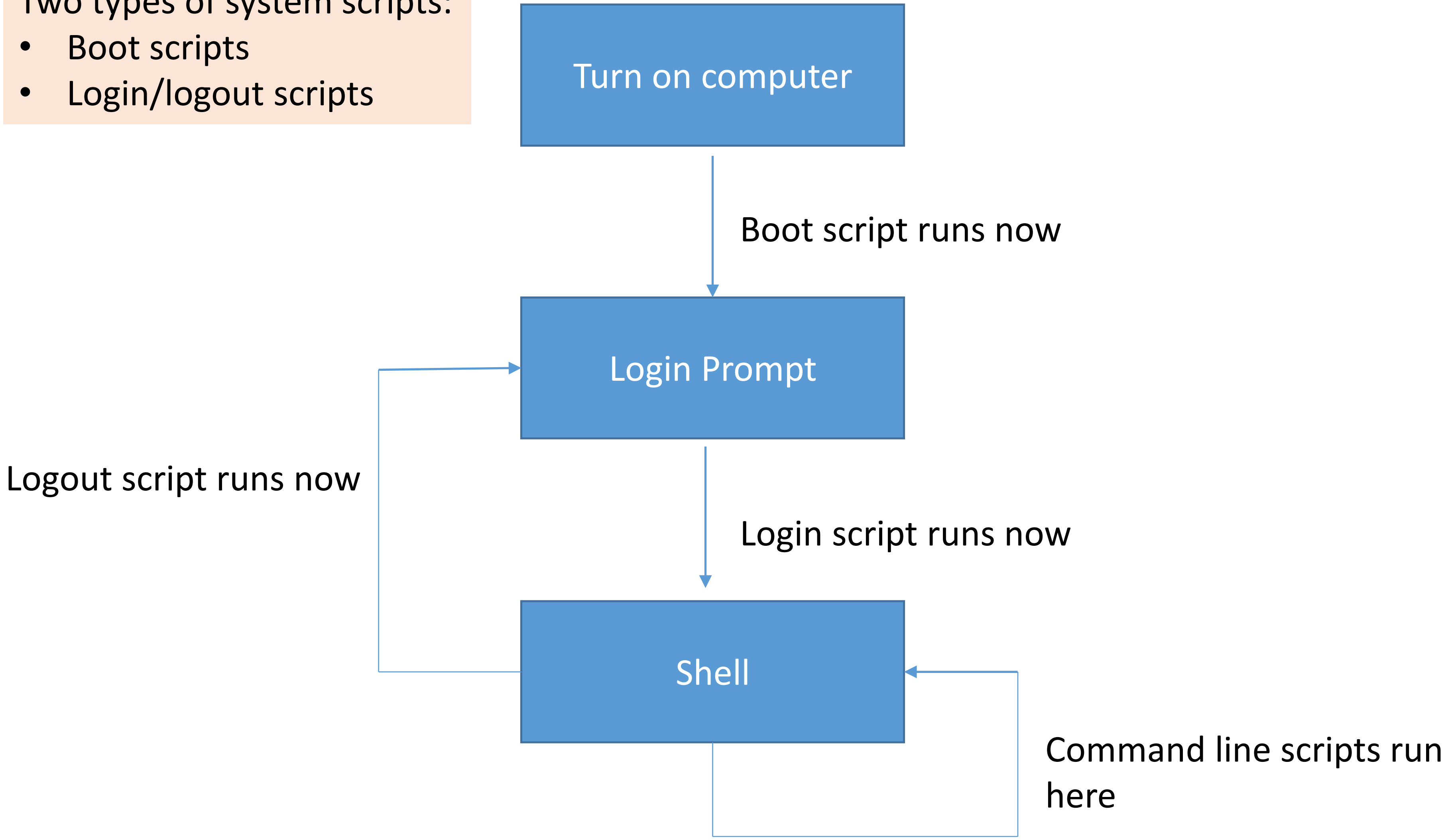
Week 4 Lecture 3

System Scripts

System Scripts

Two types of system scripts:

- Boot scripts
- Login/logout scripts



System Scripts

System scripts are used by the operating system or shell for configuration purposes.

Since they are similar to regular scripts all the regular script commands also work.

System Scripts

Boot scripts:

- Are created and managed by the root (or system operator).
- When the computer is turned on this script is executed.

Login scripts:

- Similar to a boot script but is managed by the user.
- When the user logs in this script is executed.

Logout scripts:

- Similar to the login script but not always supported by shells.
- When the user logs out this script is executed.

Default Script Files

- Session Script Files

.cshrc	csh login script
.kshrc	ksh login script
.login	sh login script
.bash_profile	bash login script
.bashrc	
.logout	sh and csh logout

- Not Script files

.plan	extra finger info
.forward	email forwarding

Note!

- Login scripts only run when you login, not from the command-line prompt.
- Login scripts require you to use **setenv** or **export** when changing the run-time environment.

Default script files are hidden

Syntax: .name

The dot makes the file hidden.

To see them:

ls

ls -a

You must list with the -a option to see hidden files.
Many of the system files are hidden files.

They are hidden files:

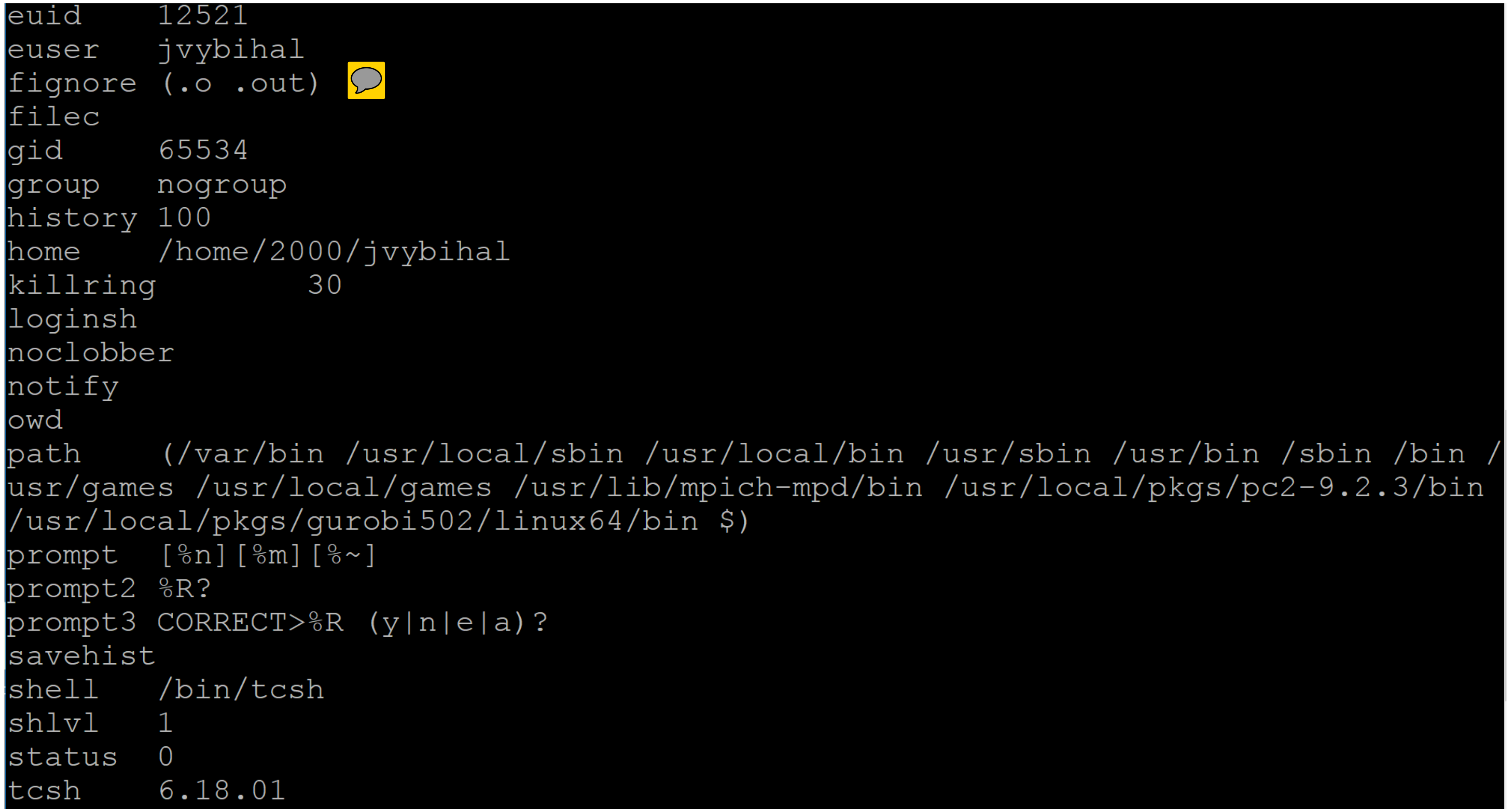
.cshrc

.bashrc

Session & Environment Variables

- These variables are created when you login.
- They contain information about your session.
  - Your IP address
  - The shell you are using
  - Your username
- Environment values can be changed later on, for example:
  - Your prompt
  - Text and background colours
  - The terminal type to process special keyboard keys, eg. F-keys

Example Session Variables



```
euid    12521
euser   jvybihal
fignore (.o .out)
filec
gid     65534
group   nogroup
history 100
home    /home/2000/jvybihal
killring        30
loginsh
noclobber
notify
owd
path    (/var/bin /usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin /bin /
usr/games /usr/local/games /usr/lib/mpich-mpd/bin /usr/local/pkgs/pc2-9.2.3/bin
/usr/local/pkgs/gurobi502/linux64/bin $)
prompt  [%n][%m][%~]
prompt2 %R?
prompt3 CORRECT>%R (y|n|e|a)?
savehist
shell   /bin/tcsh
shlvl   1
status  0
tcsh    6.18.01
```

Use the **set** command to see your variables.

Environment Variables

- In Bash:

```
export SHELL_VAR=value
```

or

```
SHELL_VAR=value
```

- In tcsh:

```
setenv SHELL_VAR value
```

Environment Variables Example

- Setting your prompt:
  - set prompt="I am the best>>"
  - export ps1="I am the best>>"
- Setting the terminal type:
  - set TERM="VT100"
  - export TERM="VT100"

The prompt is the command-line symbol that is displayed when the shell is waiting for your next command.

The TERM, or terminal, describes your keyboard and screen. VT100 is a standard simple 256 color screen with a keyboard that has F-keys.

Login Scripting

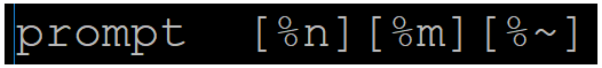
•To customize your account:

– set prompt = “Best Student $home> ”

– setenv prompt “Best Student $home> ”

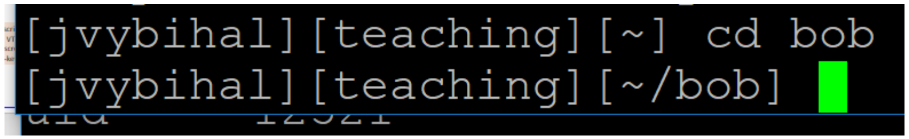
– set ps1=”Best Student $home>”

Notice the use of $variables within your configuration.



```
prompt   [%n][%m][%~]
```

%n → user name
%m → machine name
%~ → current directory



```
[jvybihal][teaching][~]  cd bob
[jvybihal][teaching][~/bob] █
```

Login Scripting

- To customize your account:

  – set history = 100

This will remember the commands you enter at the keyboard. In the example about 100 commands you type.

Using the up and down arrow keys you can cycle through the commands you have been using. Once you found the command you want pressing the enter key will execute that command.

You can directly invoke a command from the command-line prompt by using the “bang” command, the exclamation mark, !, following by the index number of the command. Using the example above, it would be the position in the list from 0 to 99, since we have 100.

Login Scripting

- To customize your account:

  – alias yourTag oneWordCommand

  – alias yourTag 'multi-word command'

  – Example:

```
alias ll 'ls -l-a'
alias dir ls
```

Login Scripting

- The PATH is a set of directories a shell searches for executables.
  - In Unix, it is a colon (:) separated list.
    - You can use the **which** command to figure out what file path is needed.
- The CLASSPATH is the set of directories the JVM searches when loading classes.

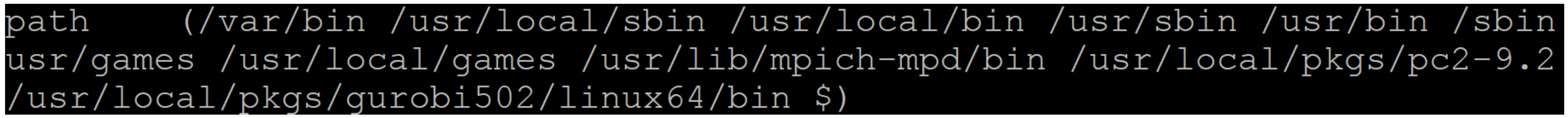
Path and Classpath

- set path=/home/foo:/bin/exe
  - set path=$path:/bla/folder:/bla2

Notice the reuse of $path in the configuration.

- export path=/home/foo:/bin/exe

- set classpath=/home/java:bin/java



```
path     (/var/bin /usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin
usr/games /usr/local/games /usr/lib/mpich-mpd/bin /usr/local/pkgs/pc2-9.2
/usr/local/pkgs/gurobi502/linux64/bin $)
```

Other “start-up” things . . .

- You can set your default editor.
  - EDITOR=vi
- Some applications might require you to set up an environment variable.
  - PVM-ROOT=/usr/local
  - set FILE=”*.txt”
  - set DIR=”/usr/jack/backup

Using the environment variables from the prev slide:

```
#!/bin/bash
cp “$FILE $DIR”
```

We would execute the program without the need of command line argument:

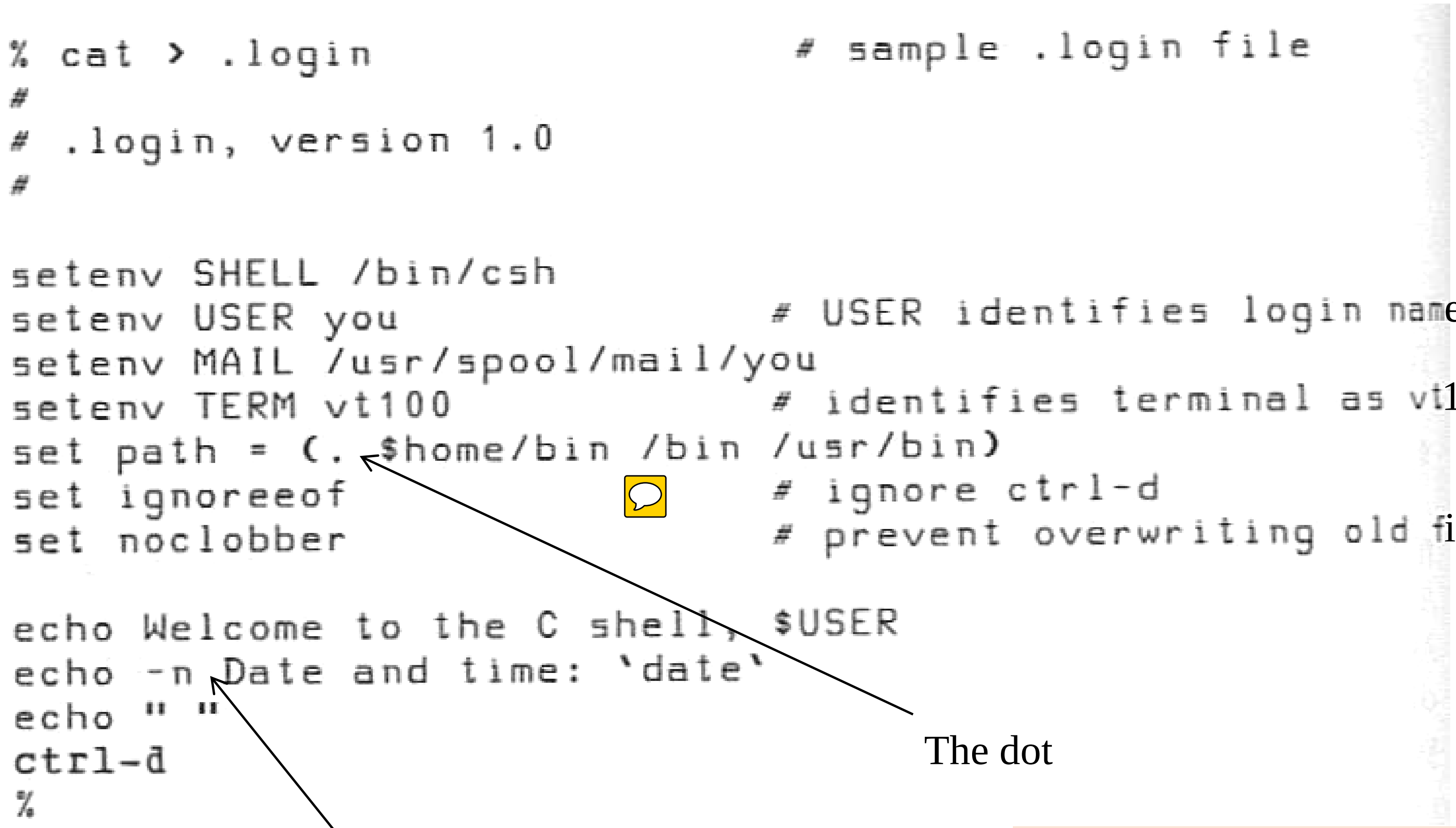
```
$ ./backup
```

The variables $FILE and $DIR are using the the ./backup script.

Other defaults…

•HOME	path to home directory
•SHELL	path to your shell
•TERM	type of terminal I/O
•USER	your user name
•PWD	your current directory

SH .login Example



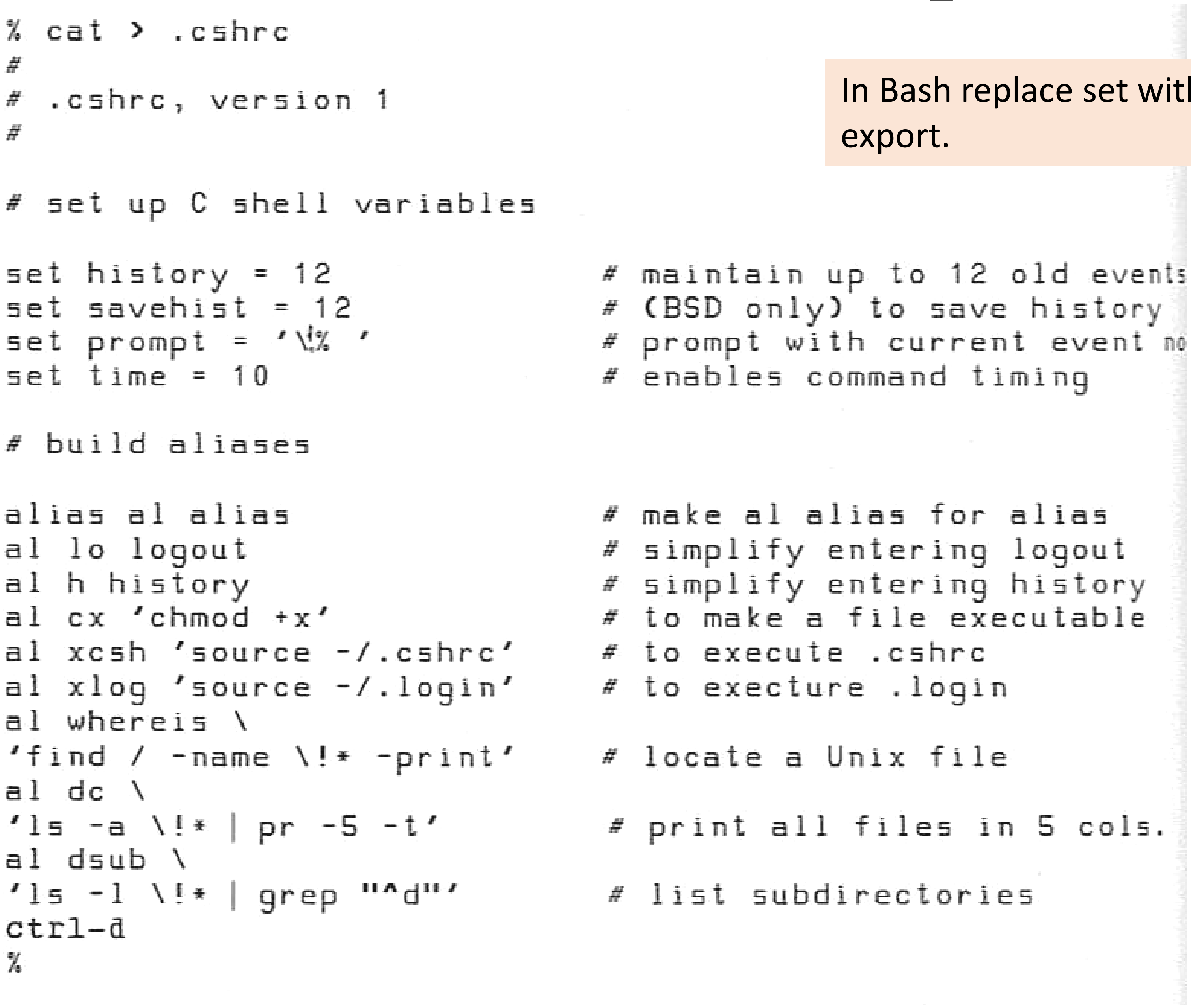
```
% cat > .login                    # sample .login file
#
# .login, version 1.0
#

setenv SHELL /bin/csh
setenv USER you                   # USER identifies login name
setenv MAIL /usr/spool/mail/you
setenv TERM vt100                 # identifies terminal as vt100
set path = (. $home/bin /bin /usr/bin)
set ignoreeof                     # ignore ctrl-d
set noclobber                     # prevent overwriting old file

echo Welcome to the C shell, $USER
echo -n Date and time: `date`
echo " "
ctrl-d
%
```

The dot

-n do not print trailing new line

In Bash replace setenv and set with export.

CSH .cshrc Example



```
% cat > .cshrc
#
# .cshrc, version 1
#

# set up C shell variables

set history = 12                # maintain up to 12 old events
set savehist = 12               # (BSD only) to save history
set prompt = '\!% '             # prompt with current event no
set time = 10                   # enables command timing

# build aliases

alias al alias                  # make al alias for alias
al lo logout                    # simplify entering logout
al h history                    # simplify entering history
al cx 'chmod +x'                # to make a file executable
al xcsh 'source ~/.cshrc'       # to execute .cshrc
al xlog 'source ~/.login'       # to execture .login
al whereis \
'find / -name \!* -print'       # locate a Unix file
al dc \
'ls -a \!* | pr -5 -t'          # print all files in 5 cols.
al dsub \
'ls -l \!* | grep "^d"'         # list subdirectories
ctrl-d
%
```

In Bash replace set with export.