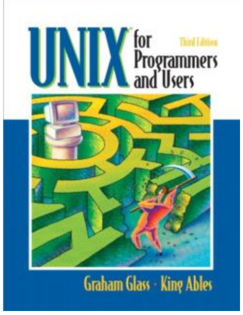
EECS2031 Software Tools

F 2021

Dec 1,2 Lecture 23

Contents



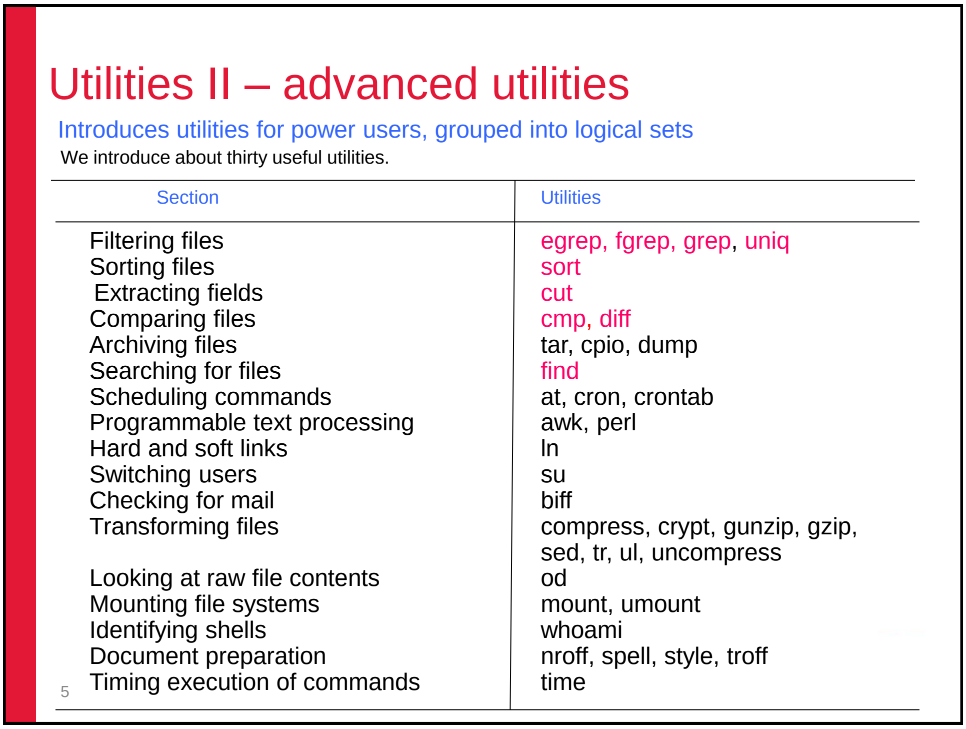
- Overview of UNIX
  - Structures
  - File systems
    - Pathname: absolute vs relative
    - Security -rwx-rw--x
  - Process:
    - Exit code ≥ 0
    - IPC: Pipes

      who | sort   who | sort | head -3   who | grep Wang | wc -l

- Utilities/commands
  - Basic: pwd, ls, rmdir, mkdir, cat, more, mv, cp, rm, wc, chmod
  - Advanced: **grep/egrep, sort, cut, find ….**
- **Shell (common shell functionalities)**

Previous class

- Bourn (again) Shell
  - scripting language



Utilities II – advanced utilities

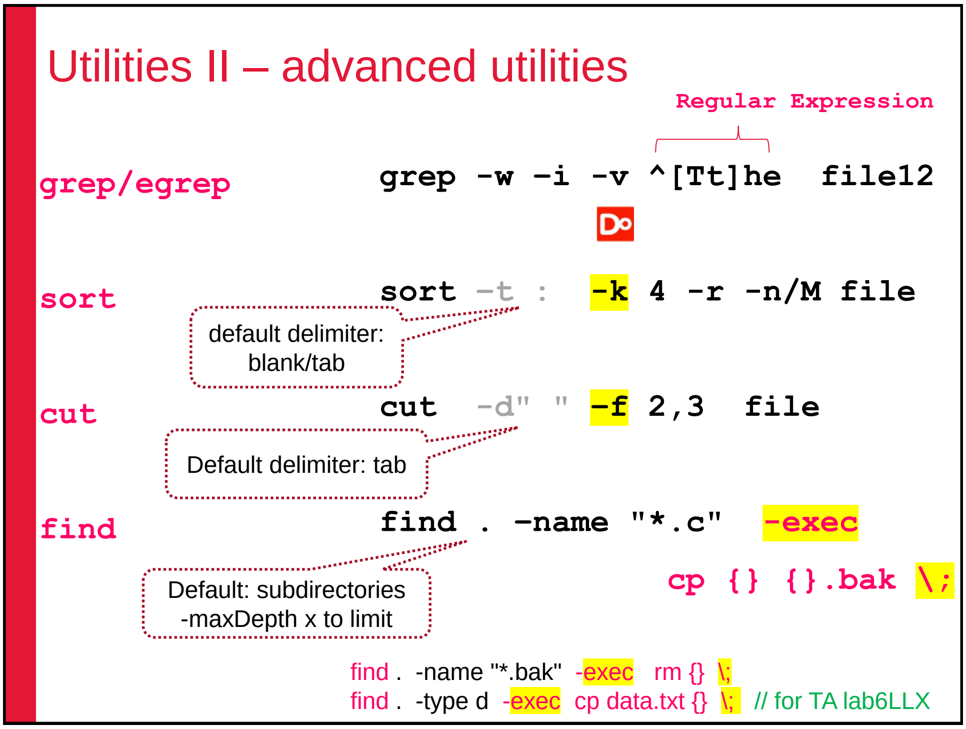
Introduces utilities for power users, grouped into logical sets

We introduce about thirty useful utilities.

Section	Utilities
Filtering files	egrep, fgrep, grep, uniq
Sorting files	sort
Extracting fields	cut
Comparing files	cmp, diff
Archiving files	tar, cpio, dump
Searching for files	find
Scheduling commands	at, cron, crontab
Programmable text processing	awk, perl
Hard and soft links	ln
Switching users	su
Checking for mail	biff
Transforming files	compress, crypt, gunzip, gzip, sed, tr, ul, uncompress
Looking at raw file contents	od
Mounting file systems	mount, umount
Identifying shells	whoami
Document preparation	nroff, spell, style, troff
Timing execution of commands	time

5

5



Utilities II – advanced utilities

Regular Expression

grep/egrep

```
grep -w -i -v ^[Tt]he  file12
```

sort

```
sort -t :  -k 4 -r -n/M file
```

default delimiter: blank/tab

cut

```
cut  -d" " -f 2,3  file
```

Default delimiter: tab

find

```
find . -name "*.c"  -exec
                      cp {} {}.bak \;
```

Default: subdirectories
-maxDepth x to limit

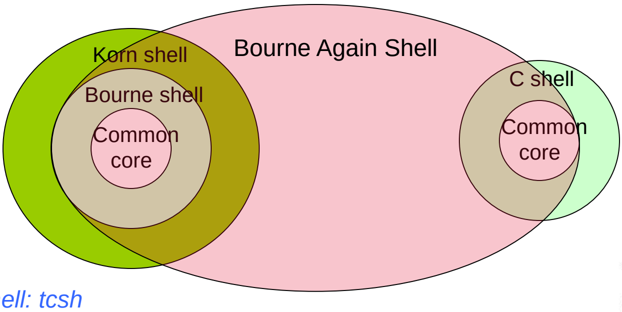
```
find .  -name "*.bak"  -exec  rm {} \;
find .  -type d  -exec  cp data.txt {}  \;  // for TA lab6LLX
```

6

SHELL FUNCTIONALITY

- This part describes the common core of functionality that all four shells provide
  - E.g., pipe who | sort
  - E.g., filename wildcards ls *.c ls a?.c

- The relationship among the four shells:



Login shell: tcsh

An enhanced but based on and completely compatible version of the C shell, *csh*

7

METACHARACTERS

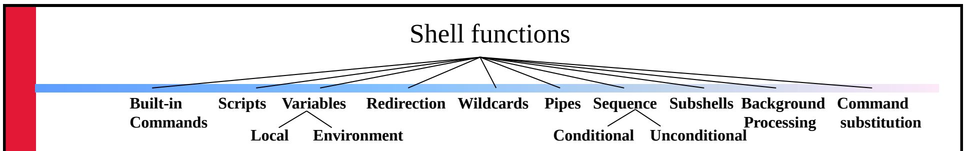
Some characters are processed specially by a shell and are known as metacharacters.

All four shells share a core set of common metacharacters, whose meanings are as follow:

Symbol	Meaning
>	Output redirection; writes standard output to a file.
>>	Output redirection; appends standard output to a file.
<	Input redirection; reads standard input from a file.
<<	Input redirection; reads standard input from script up to tok.
*	Filename-substitution (wildcard); matches zero or more characters.
?	Filename-substitution (wildcard); matches any single character.
[...]	Filename-substitution (wildcard); matches any character between the brackets.

Don't confuse with Regex

8



Symbol	Meaning
\`command\`	Command substitution; replaced by the output from command
$	Variable substitution. Expands the value of a variable.
&	Runs a command in the background. jedit&
|	Pipe symbol; sends the output of one process to the input of another
;	Used to sequence commands. echo hello; wc lyrics
||	Conditional execution; executes a command if the previous one fails.
&&	Conditional execution; executes a command if the previous one succeeds.
(…)	Groups commands.
#	All characters that follow up to a new line are ignored by the shell and program (i.e., used for a comment)
\	Prevents special interpretation of the next character.
' ' " "	quoting
Scripts	

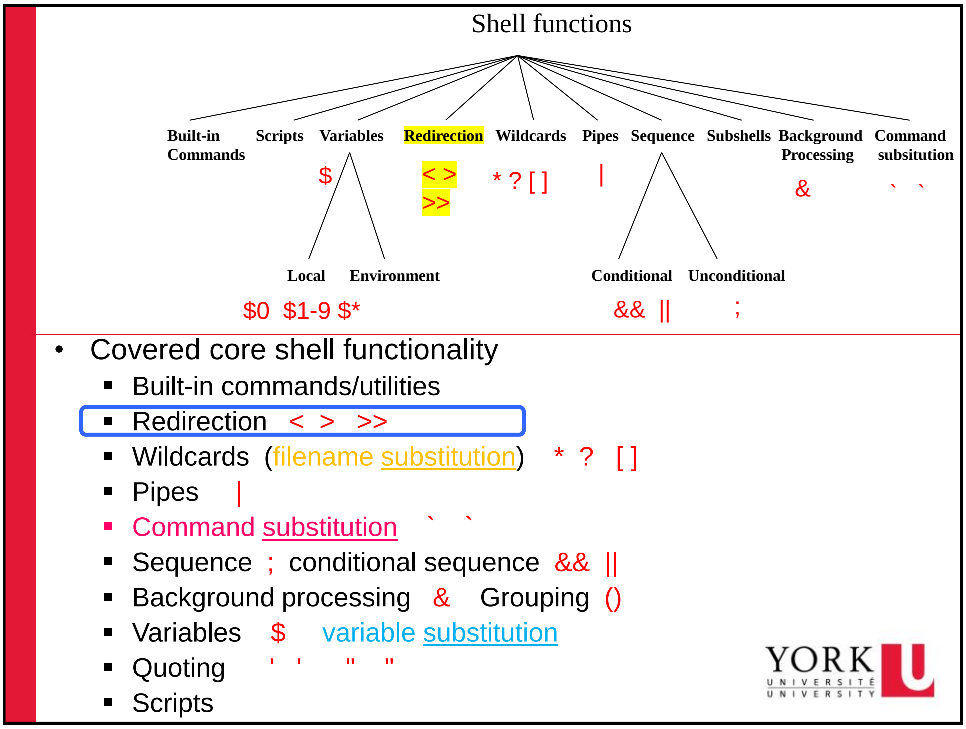
- **When you enter a command, the shell scans it for metacharacters and (if any) processes them specially**

  When all metacharacters have been processed, the command is finally executed.

- To turn off the special meaning of a metacharacter, precede it by a backslash(\) character. # Also " " ' ' (later)

- Here's an example:

```
$ echo  hi > file      #   store output of echo in "file".
$ cat  file            #   look at the contents of "file".
hi

$ echo hi \> file      #   inhibit > metacharacter.
hi > file              #    > is treated like other characters.
$ cat file             #   look at the file again.  Not written
ls: cannot access file: No such file or directory such a file

$ echo 3 + 2  = 5
$ echo 3 \* 2 = 6
```



Shell functions

Built-in Commands | Scripts | Variables | Redirection | Wildcards | Pipes | Sequence | Subshells | Background Processing | Command subsitution

Variables: $ → Local, Environment — $0 $1-9 $*

Redirection: < > >>

Wildcards: * ? [ ]

Pipes: |

Sequence: Conditional, Unconditional — && || ;

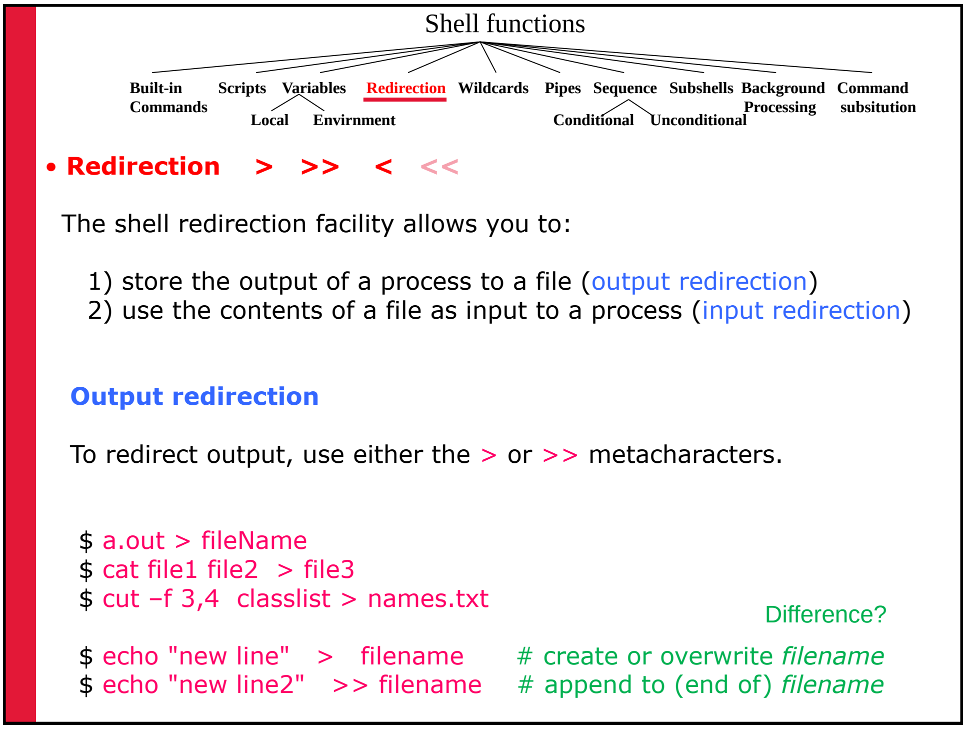
Background Processing: &

Command subsitution: ` `

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts

YORK U UNIVERSITÉ UNIVERSITY

11



Shell functions

Built-in Commands | Scripts | Variables | Redirection | Wildcards | Pipes | Sequence | Subshells | Background Processing | Command subsitution

Variables: Local, Envirnment

Sequence: Conditional, Unconditional

- **Redirection > >> < <<**

The shell redirection facility allows you to:

1) store the output of a process to a file (output redirection)
2) use the contents of a file as input to a process (input redirection)

Output redirection

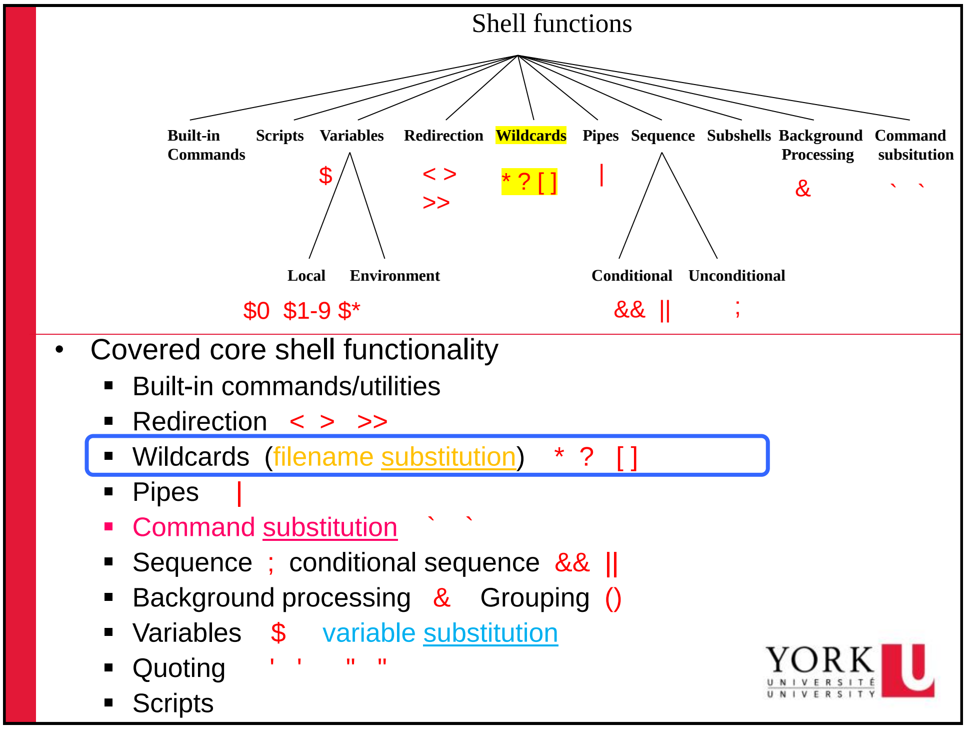
To redirect output, use either the > or >> metacharacters.

```
$ a.out > fileName
$ cat file1 file2  > file3
$ cut –f 3,4  classlist > names.txt
```

Difference?

```
$ echo "new line"   >   filename      # create or overwrite filename
$ echo "new line2"  >> filename       # append to (end of) filename
```

12



Shell functions

Built-in Commands | Scripts | Variables $ | Redirection < > >> | Wildcards * ? [ ] | Pipes | | Sequence | Subshells | Background Processing & | Command subsitution ` `

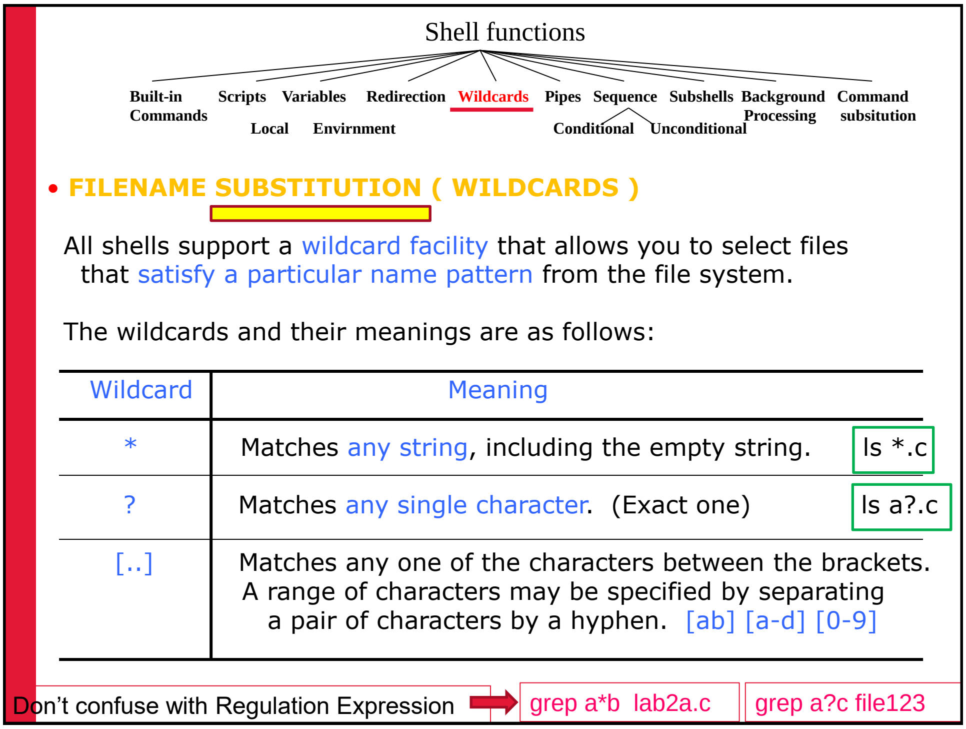
Variables: Local, Environment — $0 $1-9 $*

Sequence: Conditional && ||, Unconditional ;

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts

YORK UNIVERSITÉ UNIVERSITY

13



Shell functions

Built-in Commands | Scripts | Variables (Local, Envirnment) | Redirection | Wildcards | Pipes | Sequence (Conditional, Unconditional) | Subshells | Background Processing | Command subsitution

- **FILENAME SUBSTITUTION (WILDCARDS)**

All shells support a wildcard facility that allows you to select files that satisfy a particular name pattern from the file system.

The wildcards and their meanings are as follows:

Wildcard	Meaning	
*	Matches any string, including the empty string.	ls *.c
?	Matches any single character. (Exact one)	ls a?.c
[..]	Matches any one of the characters between the brackets. A range of characters may be specified by separating a pair of characters by a hyphen. [ab] [a-d] [0-9]	

Don't confuse with Regulation Expression → grep a*b lab2a.c | grep a?c file123

14

Used for filename wildcard, in ls, cp, mv, rm, cat, more, wc, chmod ….
grep, find operating on multiple files

```
$ ls   EECS2031*       # list files whose name beginning with EECS2031
EECS2031O                  EECS2031N             EECS2031N.LAB03
EECS2031O.LAB01            EECS2031N.LAB01
EECS2031O.LAB02            EECS2031N.LAB02       EECS2031N.LAB03

$ ls  EECS2031?    # files whose name is EECS2031 by exactly one char
EECS2031O     EECS2031N                 # if also EECS2031 match?

$ ls  EECS2031O*
EECS2031O      EECS2031O.LAB01    EECS2031O.LAB02

$ ls EECS2031O?
ls: No match.

$ ls EECS2031O.*          #   EECS2031O.?
EECS2031O.LAB01    EECS2031O.LAB02

$ ls EECS2031?.LAB?2
EECS2031O.LAB02     EECS2031N.LAB02

$ ls EECS2031[ABOX].LAB??    #  EECS2031[ABOX].LAB?
EECS2031O.LAB01     EECS2031O.LAB02
```

Same for other commands

16

Used for filename wildcard, in ls, cp, mv, rm, cat, more, wc, chmod ….
grep, find operating on multiple files

- $ cp /eecs/dept/course/2019-20/W/2030tmp/xFile?   ./
- $ cp /eecs/dept/course/2019-20/W/2030tmp/xFile*   ./
- $ cp /eecs/dept/course/2019-20/W/2030tmp/xFile[23] .
- $ rm ../*.bak   # "*.bak"   remove all bak files in parent directories
- $ find . -name 'a??.c'   # "a??.c"   search for all aXX.c

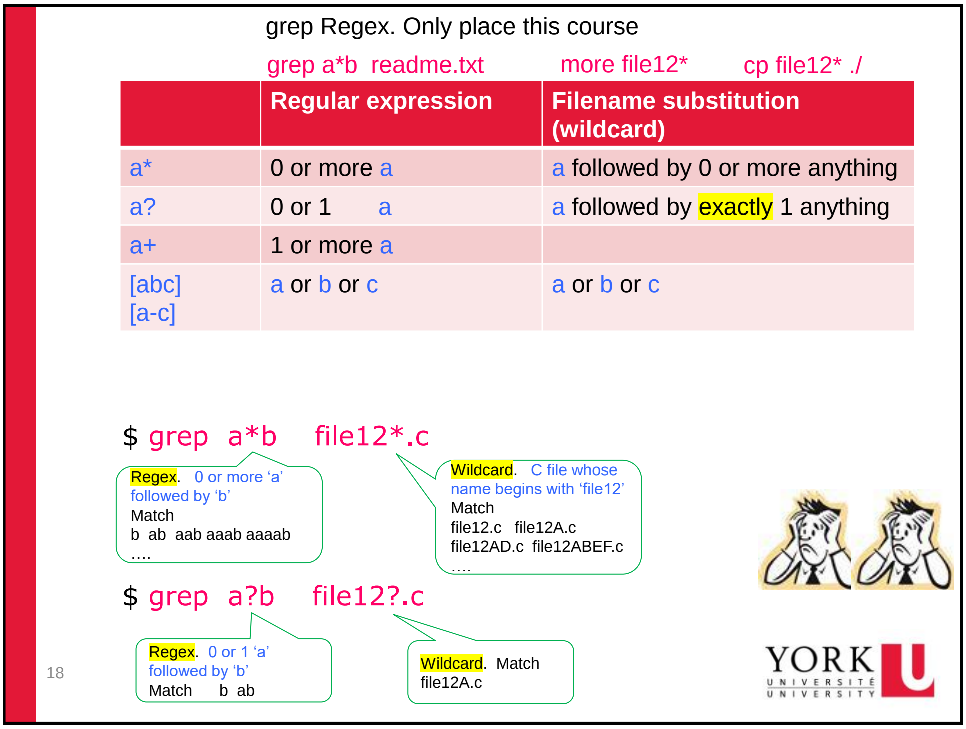
```
ab1.c
ab2.c
aXc.c          #  abc.c  does not match
```

- $ find . -name '*.c'   -exec mv {} {}.2031A \;

# find all c files and then rename it to filename.2031A

```
mv  a1.c  a1.c.2031A
mv  lab3a.c  lab3a.c.2031A
…..
```

17



grep Regex. Only place this course

grep a*b readme.txt   more file12*   cp file12* ./

	Regular expression	Filename substitution (wildcard)
a*	0 or more a	a followed by 0 or more anything
a?	0 or 1 a	a followed by exactly 1 anything
a+	1 or more a	
[abc] [a-c]	a or b or c	a or b or c

```
$ grep  a*b   file12*.c
```

Regex. 0 or more 'a' followed by 'b'
Match
b ab aab aaab aaaab
….

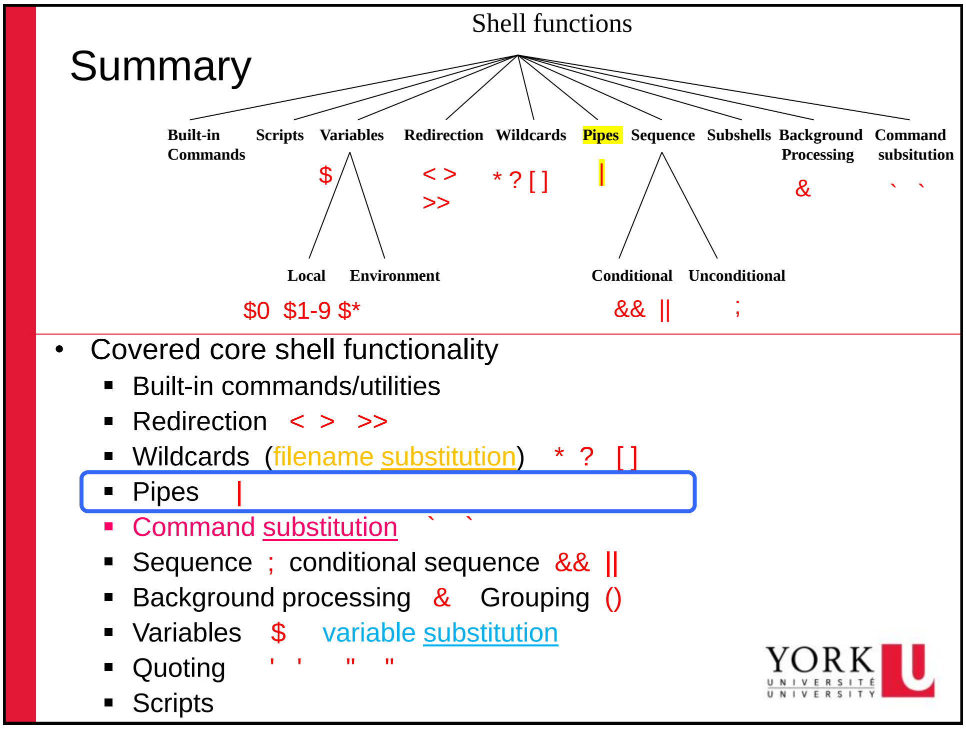
Wildcard. C file whose name begins with 'file12'
Match
file12.c file12A.c
file12AD.c file12ABEF.c
….

```
$ grep  a?b   file12?.c
```

Regex. 0 or 1 'a' followed by 'b'
Match   b ab

Wildcard. Match
file12A.c

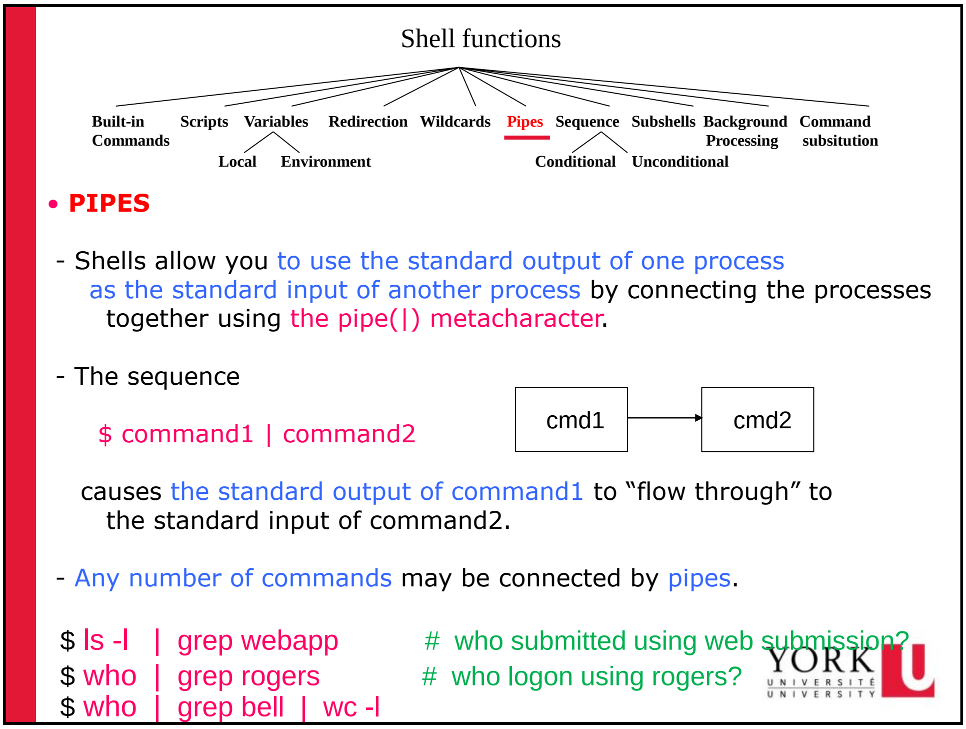
18



Shell functions

Summary

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts



Shell functions

Built-in Commands
Scripts
Variables
Local
Environment
Redirection
Wildcards
Pipes
Sequence
Conditional
Unconditional
Subshells
Background Processing
Command subsitution

- **PIPES**

- Shells allow you to use the standard output of one process as the standard input of another process by connecting the processes together using the pipe(|) metacharacter.

- The sequence

  $ command1 | command2

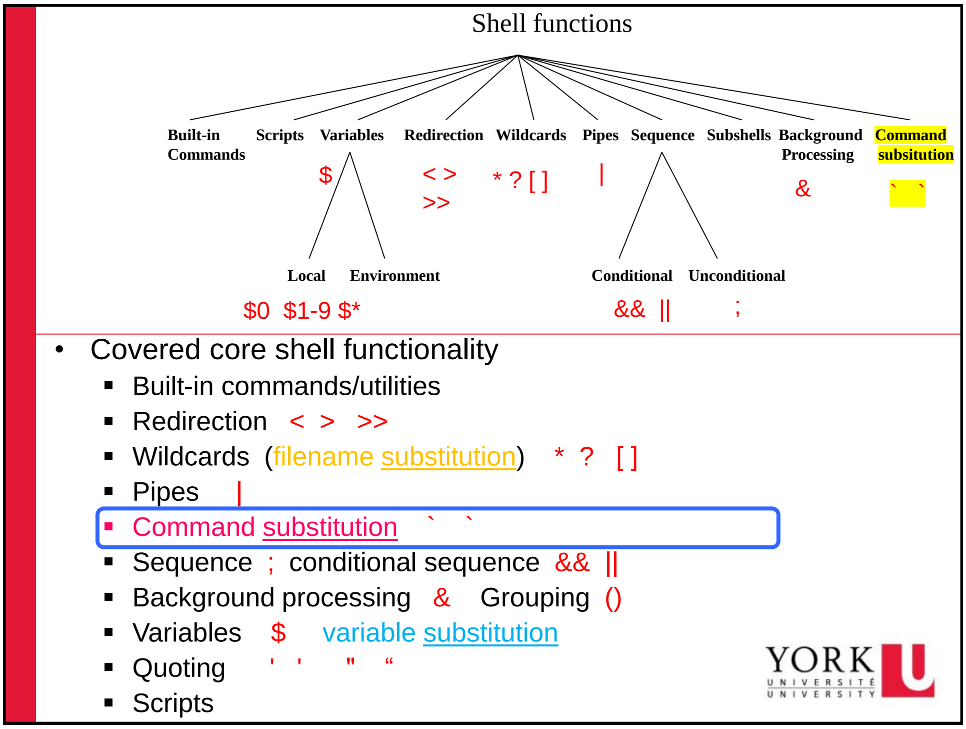
cmd1
cmd2

causes the standard output of command1 to “flow through” to the standard input of command2.

- Any number of commands may be connected by pipes.

```
$ ls -l  | grep webapp        # who submitted using web submission?
$ who  | grep rogers          # who logon using rogers?
$ who  | grep bell  |  wc -l
```

20

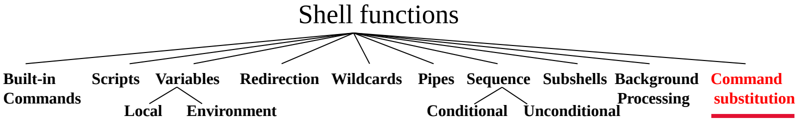


Shell functions

Built-in Commands
Scripts
Variables
$
Local
Environment
$0 $1-9 $*
Redirection
< >
>>
Wildcards
* ? [ ]
Pipes
|
Sequence
Conditional
Unconditional
&& ||
;
Subshells
Background Processing
&
Command subsitution
` `

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " “
  - Scripts

21



COMMAND SUBSTITUTION used very very … heavily in script!

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

`command`

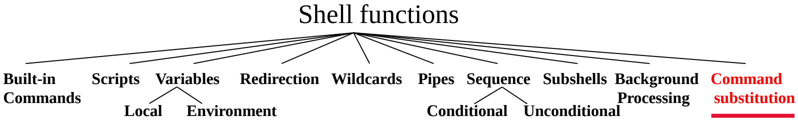
For example:

```
$ echo  the date today is `date` , right?
the date today is Sun Mar 28 08:57:44 EDT 2020, right?
$
$ echo  there are `who | wc -l` users on the system
there are 31 users on the system
```

Bash: **$(command)**

```
echo  the date today is $(date), right?
```

22



COMMAND SUBSTITUTION used very very … heavily in script!

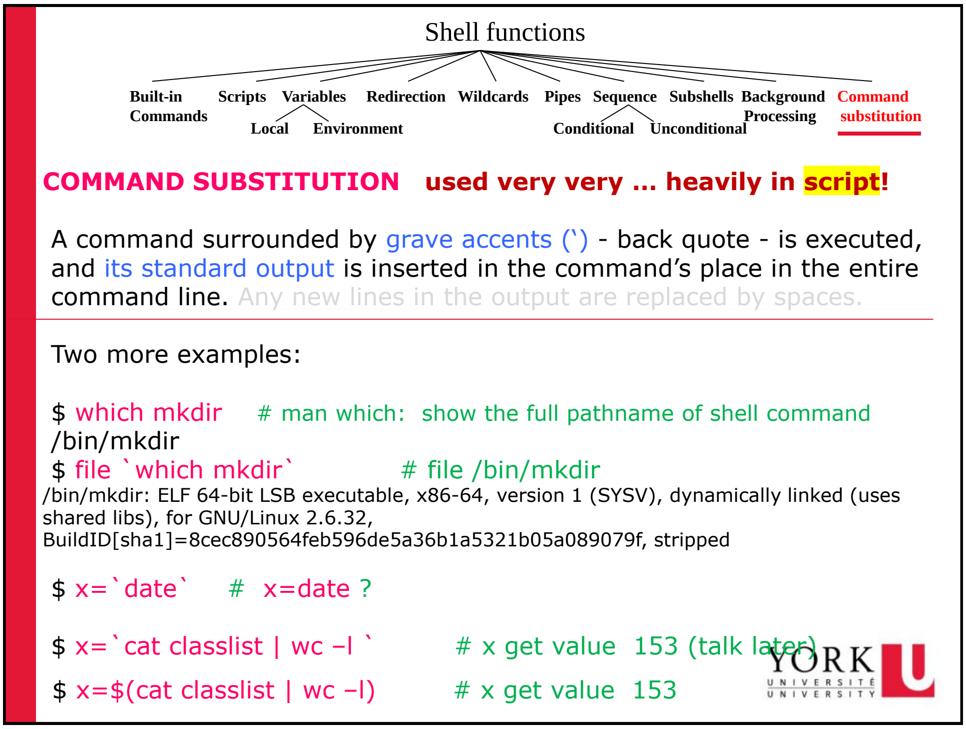
A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

For example:

```
$echo  there are `cat classlist | wc –l` students in the class
there are 153 students in the class

$echo  has `cat classlist | grep –w Wang | wc -l` students name Wang
has 3 students name Wang
```

Bash: `echo  there are $(cat classlist | wc -l) students in the class`

23



Shell functions

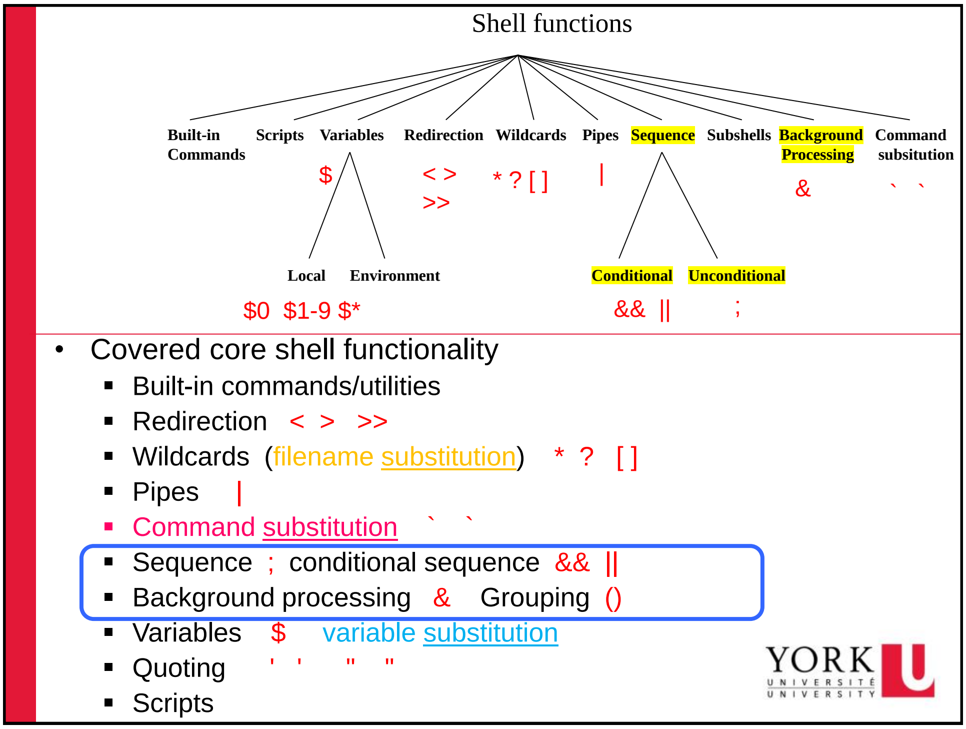
Built-in Commands
Scripts
Variables
Local
Environment
Redirection
Wildcards
Pipes
Sequence
Conditional
Unconditional
Subshells
Background Processing
Command substitution

COMMAND SUBSTITUTION used very very … heavily in script!

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line. Any new lines in the output are replaced by spaces.

Two more examples:

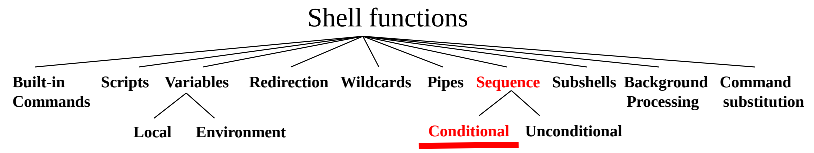
```
$ which mkdir    # man which:  show the full pathname of shell command
/bin/mkdir
$ file `which mkdir`          # file /bin/mkdir
/bin/mkdir: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=8cec890564feb596de5a36b1a5321b05a089079f, stripped

$ x=`date`    #  x=date ?

$ x=`cat classlist | wc –l `        # x get value  153 (talk later)

$ x=$(cat classlist | wc –l)        # x get value  153
```



Shell functions

Built-in Commands
Scripts
Variables
$
Local
Environment
$0 $1-9 $*
Redirection
< >
>>
Wildcards
* ? [ ]
Pipes
|
Sequence
Conditional
Unconditional
&& ||
;
Subshells
Background Processing
&
Command subsitution
` `

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts



- **Conditional Sequences** && ||

- Every UNIX process terminates with an exit value.
By convention, an exit value of 0 means that the process completed successfully, and a > 0 exit value indicates failure.
(opposite to C)

- You may construct sequences that make use of this exit value:

1) If you specify a series of commands separated by && tokens,

```
cmd 1  &&  cmd2
```

cmd2 is executed only if cmd1 returns exit code of **0** i.e. cmd1 succeeded

2) If you specify a series of commands separated by || tokens,

```
cmd 1  ||  cmd2
```

"Lazy evaluation"

cmd2 is executed only if cmd1 returns **nonzero** exit code. i.e., cmd1 fails

- For example,
if gcc compiles a program without fatal errors,
it creates an executable program called a.out and returns an exit code of 0;
otherwise, it returns a non-zero exit code.

```
$ gcc  myprog.c  &&  a.out     # (only) if gcc successful, then run a.out
                               # otherwise, no run a.out

$ gcc  myprog.c  ||  echo  "compilation  failed."
                              # if gcc is not successful, then echo
```

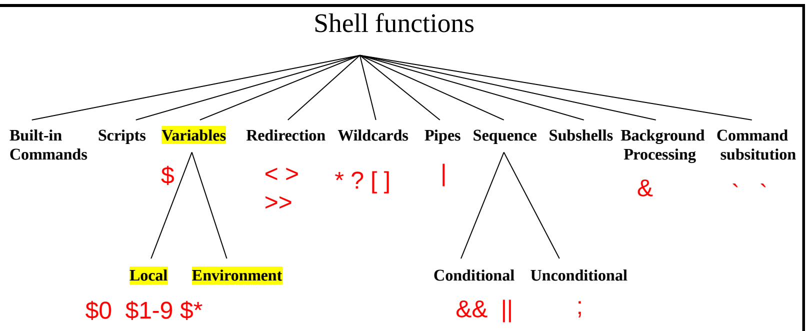
return 0 if match, return 1 otherwise

```
$ grep –w Wang classlist  && echo "found someone in class"
$ grep –w WangXXX classlist || echo "not found in class"
```

return 0 if match, return 1 otherwise

```
$ diff f1  f2  ||  echo "not same"     $ diff f1  f2  &&  echo "same"
```

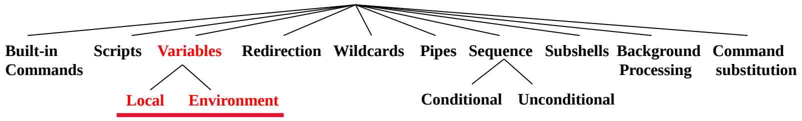
Shell functions


Built-in Commands
Scripts
Variables
Redirection
Wildcards
Pipes
Sequence
Subshells
Background Processing
Command subsitution
$
< >
>>
* ? [ ]
|
&
` `
Local
Environment
Conditional
Unconditional
$0 $1-9 $*
&& ||
;

- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts

28

Shell functions


Built-in Commands
Scripts
Variables
Redirection
Wildcards
Pipes
Sequence
Subshells
Background Processing
Command substitution
Local
Environment
Conditional
Unconditional

- **VARIABLES and variable substitution $**

- A shell supports two kinds of variables:
local/shell and environment variables.

local:
user defined
positional

Both kinds of variables hold data in a string format.

The child shell gets a copy of its parent shell’s environment variables, but not its local variables.

A set of predefined/built in special variables
- Some are Environment variables
- Some are local variables

29

For your information

- **Environment VARIABLES**

- Here is a list of the predefined environment variables that are common to all shells:

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$MAIL	the full pathname of your mailbox
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

To display your environment variables, type “set”.

Built-in local variables

- several common built-in local variables that have special meanings:

Name	Meaning
$$	The process ID of the shell.
$?	Exit code of last command execution
$0	The name of the shell script (if applicable).
$1..$9	$n refers to the n’th command line argument (if applicable).
$*	A list of all the command-line arguments.

```
$ myscript paul  ringo  george  john
     $0     $1    $2     $3      $4        …    $9
                     $*
```

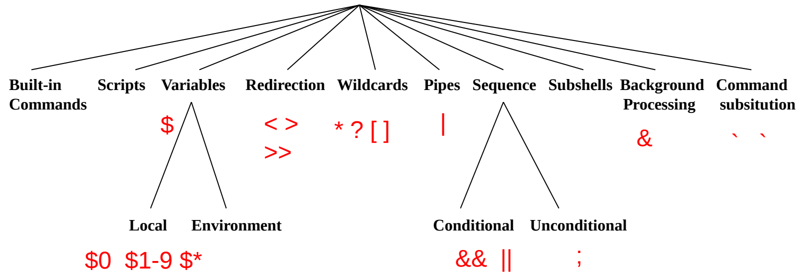
User defined local variables

Accessing variable values:
variable substitution $

```
$ echo $?
$ echo call me $USER

$ x=5    # bash
$ echo x
x
$ echo value of x is $x          # value of x is 5

$ name=Graham
$ echo Hi, I am $name          # Hi, I am Graham


$ x=`cat classlist | wc –l `     # x get value  153
$ echo $x   # 153
$ echo there are $x students   # there are 153 students
```

33

Shell functions



- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts

34

QUOTING

There are often times when you want to inhibit the shell’s wildcard-substitution * ? [], command-substitution ` ` and/or variable-substitution $ mechanisms.

The shell’s quoting system allows you to do just that.

- Here’s the way that it works:

1) Single quotes (' ') inhibits both wildcard substitution, command substitution and variable substitution.

2) Double quotes(" ") inhibits wildcard substitution only.

Single quotes (' ') inhibits both wildcard substitution, variable substitution, and command substitution.

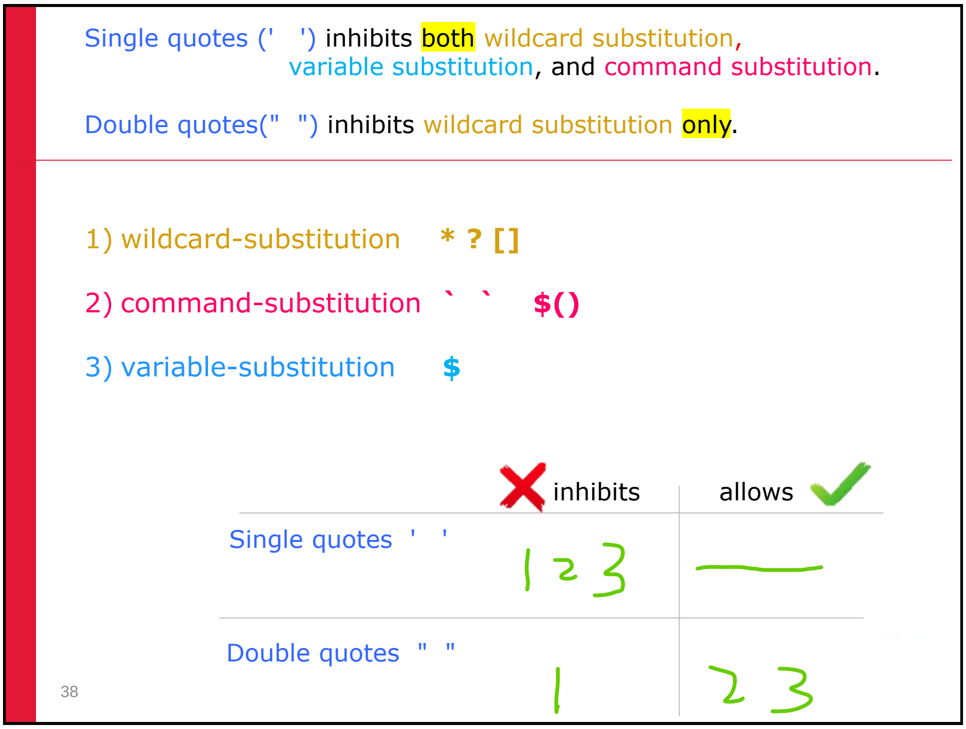
Double quotes(" ") inhibits wildcard substitution only.

1) wildcard-substitution *** ? []**

2) command-substitution **` ` $()**

3) variable-substitution **$**

	✗ inhibits	allows ✓
Single quotes ' '		
Double quotes " "		



Single quotes ('   ') inhibits both wildcard substitution,
variable substitution, and command substitution.

Double quotes("   ") inhibits wildcard substitution only.

1) wildcard-substitution   * ? []

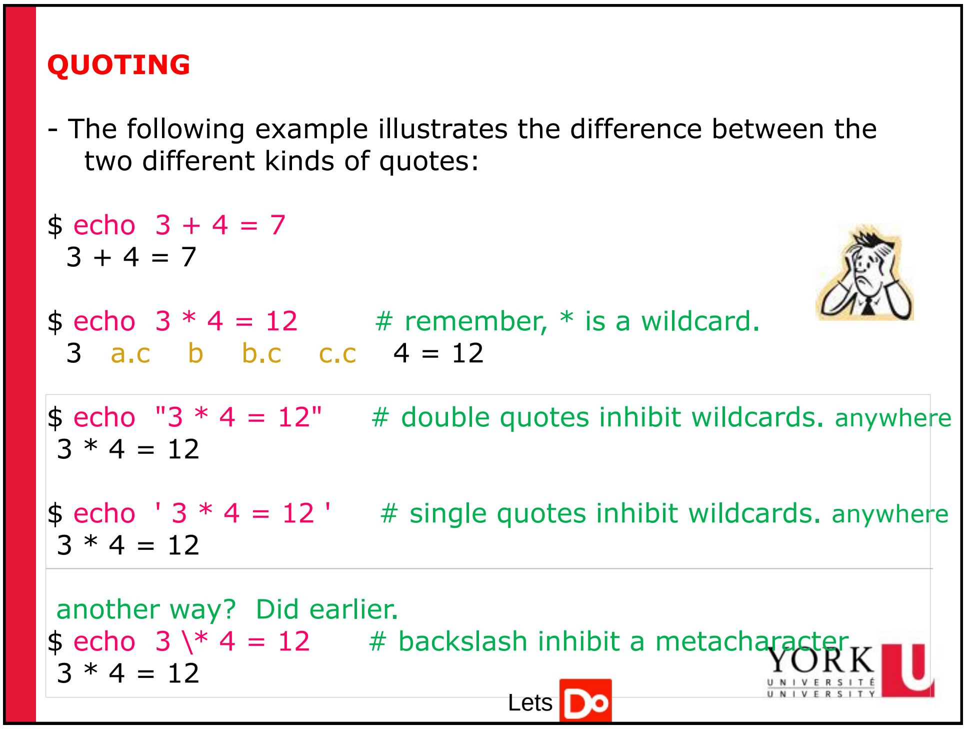
2) command-substitution   ` ` $()

3) variable-substitution $

	❌ inhibits	allows ✓
Single quotes ' '	1 2 3	—
Double quotes " "	1	2 3

38

38



QUOTING

- The following example illustrates the difference between the two different kinds of quotes:

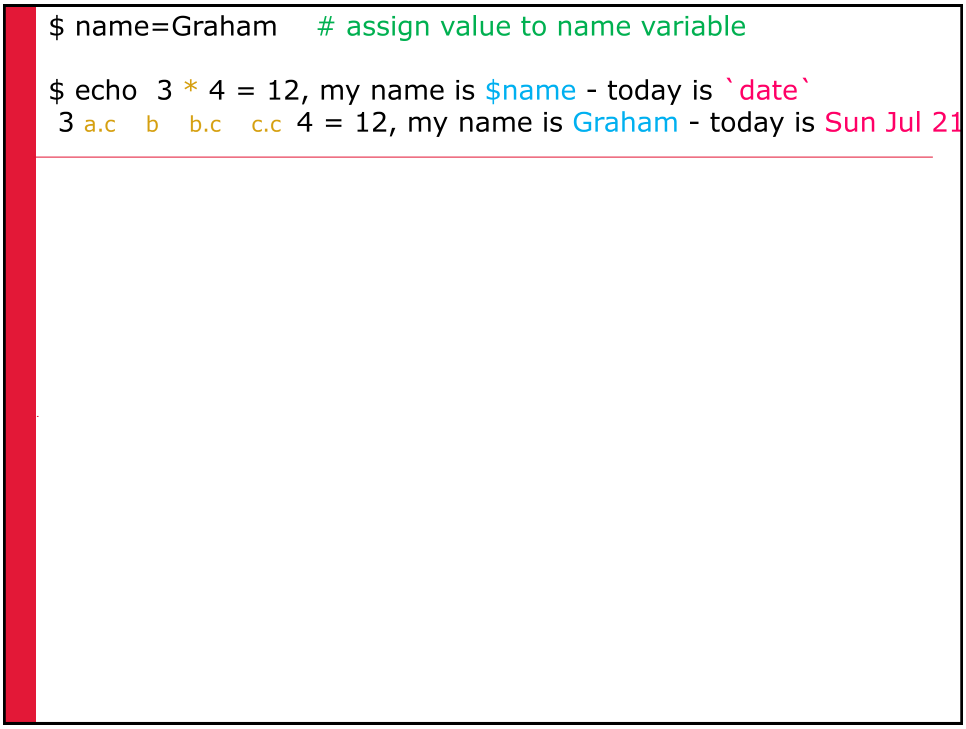
```
$ echo  3 + 4 = 7
  3 + 4 = 7

$ echo  3 * 4 = 12        # remember, * is a wildcard.
  3   a.c    b    b.c    c.c    4 = 12

$ echo  "3 * 4 = 12"     # double quotes inhibit wildcards. anywhere
 3 * 4 = 12

$ echo  ' 3 * 4 = 12 '     # single quotes inhibit wildcards. anywhere
 3 * 4 = 12
```

another way? Did earlier.

```
$ echo  3 \* 4 = 12      # backslash inhibit a metacharacter
 3 * 4 = 12
```

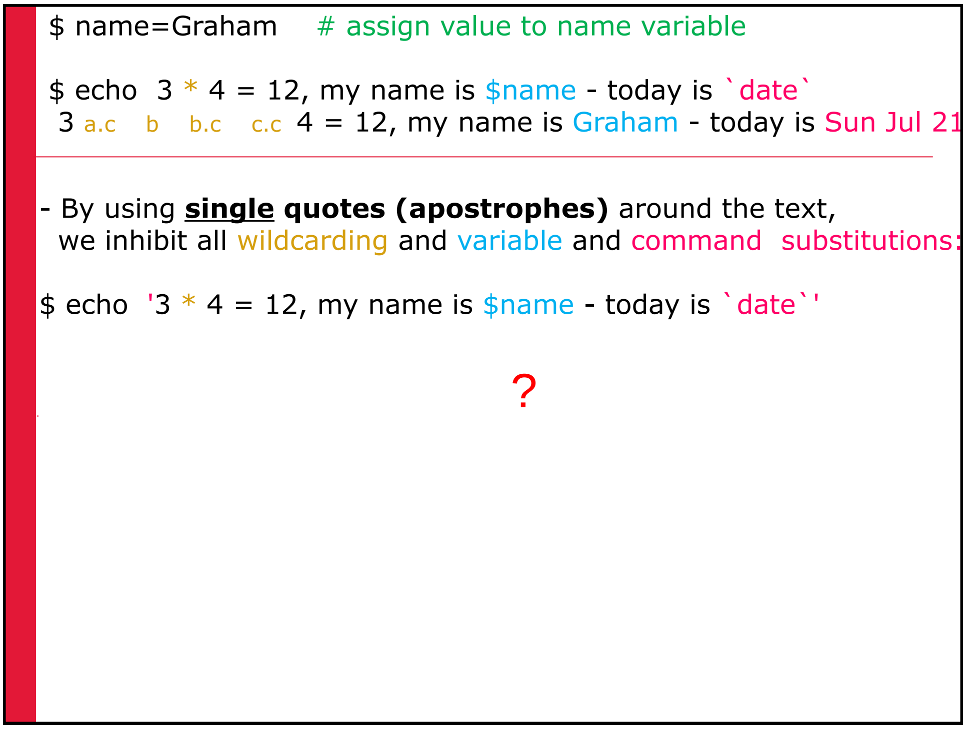
Lets Do

40



```
$ name=Graham    # assign value to name variable

$ echo  3 * 4 = 12, my name is $name - today is `date`
 3 a.c   b   b.c   c.c  4 = 12, my name is Graham - today is Sun Jul 21
```

41



```
$ name=Graham    # assign value to name variable

$ echo  3 * 4 = 12, my name is $name - today is `date`
 3 a.c   b   b.c   c.c  4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **<u>single</u> quotes (apostrophes)** around the text, we inhibit all wildcarding and variable and command substitutions:

```
$ echo  '3 * 4 = 12, my name is $name - today is `date`'
```

?

42

```
$ name=Graham     # assign value to name variable

$ echo  3 * 4 = 12, my name is $name - today is `date`
 3 a.c   b   b.c   c.c  4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **<u>single</u> quotes (apostrophes)** around the text, we inhibit all wildcarding and variable and command substitutions:

```
$ echo  '3 * 4 = 12, my name is $name - today is `date`'
 3 * 4 = 12, my name is $name - today is ‘date`
$ _
```

inhibited

```
$ name=Graham     # assign value to name variable

$ echo  3 * 4 = 12, my name is $name - today is `date`
 3 a.c   b   b.c   c.c  4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **<u>single</u> quotes (apostrophes)** around the text, we inhibit all wildcarding and variable and command substitutions:

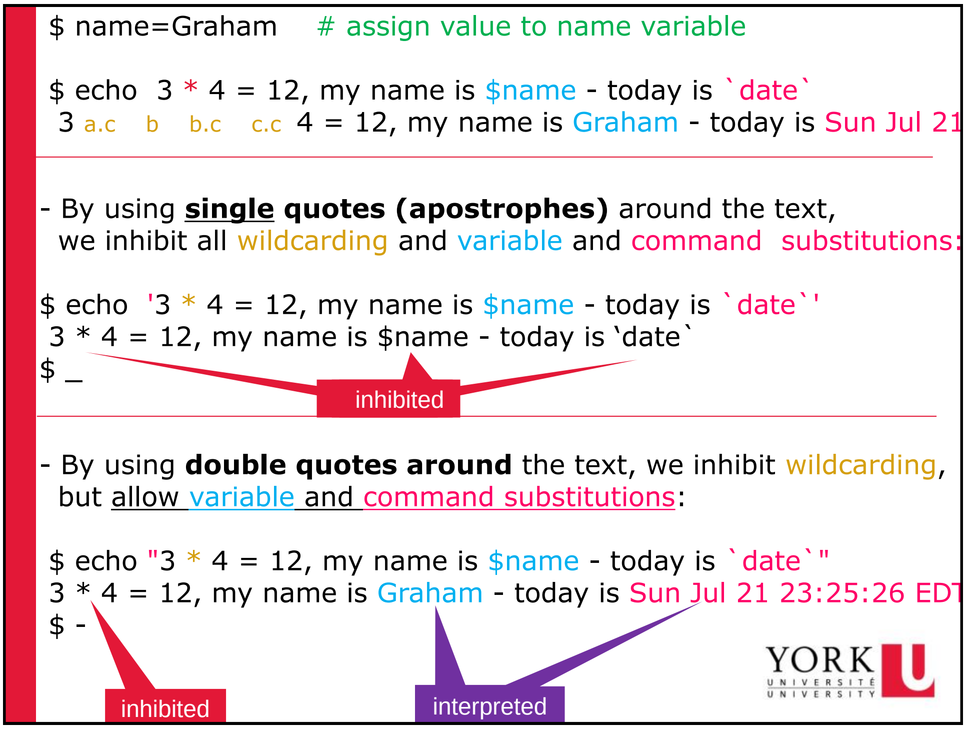
```
$ echo  '3 * 4 = 12, my name is $name - today is `date`'
 3 * 4 = 12, my name is $name - today is ‘date`
$ _
```

inhibited

- By using **double quotes around** the text, we inhibit wildcarding, but <u>allow variable and command substitutions</u>:

```
$ echo "3 * 4 = 12, my name is $name - today is `date`"
```

?



```
$ name=Graham     # assign value to name variable

$ echo  3 * 4 = 12, my name is $name - today is `date`
 3 a.c   b   b.c   c.c  4 = 12, my name is Graham - today is Sun Jul 21
```

- By using **single quotes (apostrophes)** around the text, we inhibit all wildcarding and variable and command substitutions:

```
$ echo  '3 * 4 = 12, my name is $name - today is `date`'
 3 * 4 = 12, my name is $name - today is `date`
$ _
```

inhibited

- By using **double quotes around** the text, we inhibit wildcarding, but allow variable and command substitutions:

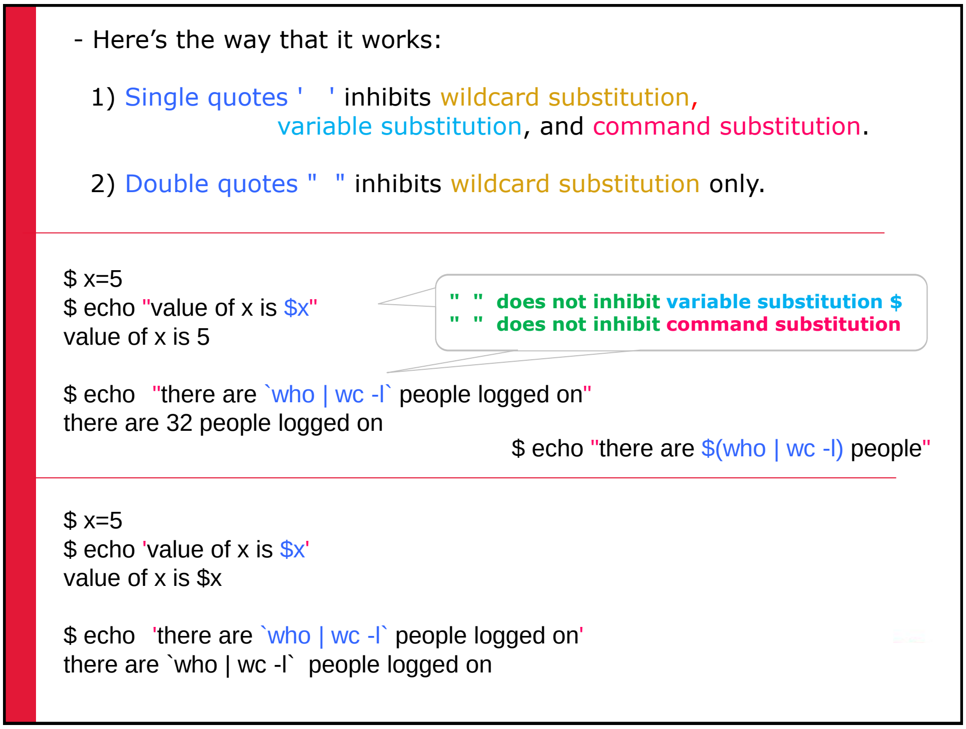
```
$ echo "3 * 4 = 12, my name is $name - today is `date`"
3 * 4 = 12, my name is Graham - today is Sun Jul 21 23:25:26 EDT
$ -
```

inhibited

interpreted

YORK U UNIVERSITÉ UNIVERSITY

45



- Here's the way that it works:

1) Single quotes ' ' inhibits wildcard substitution, variable substitution, and command substitution.

2) Double quotes " " inhibits wildcard substitution only.

```
$ x=5
$ echo "value of x is $x"
value of x is 5

$ echo  "there are `who | wc -l` people logged on"
there are 32 people logged on
```

" " does not inhibit variable substitution $
" " does not inhibit command substitution

```
$ echo "there are $(who | wc -l) people"
```

```
$ x=5
$ echo 'value of x is $x'
value of x is $x

$ echo  'there are `who | wc -l` people logged on'
there are `who | wc -l`  people logged on
```

46

- Here's the way that it works:

1) Single quotes '   ' inhibits wildcard substitution, variable substitution, and command substitution.

2) Double quotes "   " inhibits wildcard substitution only.

$ x=5
$ echo "value of x is $x"
value of x is 5

" " does not inhibit variable substitution $
" " does not inhibit command substitution

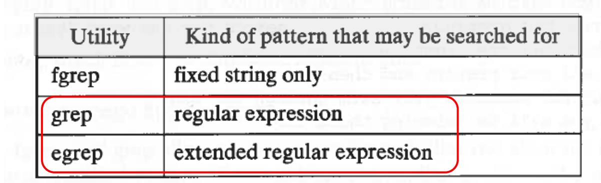
$ echo "there are `who | wc -l` people logged on"
there are 32 people logged on

Both '   ' and "   " inhibit wildcard substitution * ?

Needed for Some shell e.g., tcsh

$ egrep the lyrics — $ egrep 'the' lyrics — $ egrep "the" lyrics
~~$ egrep ab? lyrics~~ — $ egrep "ab?" lyrics — $ egrep 'ab?' lyrics
~~$ egrep ab*c lyrics~~ — $ egrep "ab*c" lyrics — $ egrep 'ab*c' lyrics

Better use quote on Regex to prevent (some) shell from interpreting Regex repetition symbol * ? as filename wildcards.



Utility	Kind of pattern that may be searched for
fgrep	fixed string only
grep	regular expression
egrep	extended regular expression

- Since many of the special characters used in regexs (e.g., * ? |) also have special meaning to the shell, it's a good idea to get in the habit of quoting your regexs
  - This will protect any special characters from being operated on by the shell
  - If you habitually do it, you won't have to worry about when it is necessary

Needed for Some shell e.g., tcsh

$egrep the lyrics — $egrep 'the' lyrics — $egrep "the" lyrics
~~$egrep ab? lyrics~~ — $egrep "ab?" lyrics — $egrep 'ab?' lyrics
~~$egrep ab*c lyrics~~ — $egrep "ab*c" lyrics — $egrep 'ab*c' lyrics

Explained next chapter

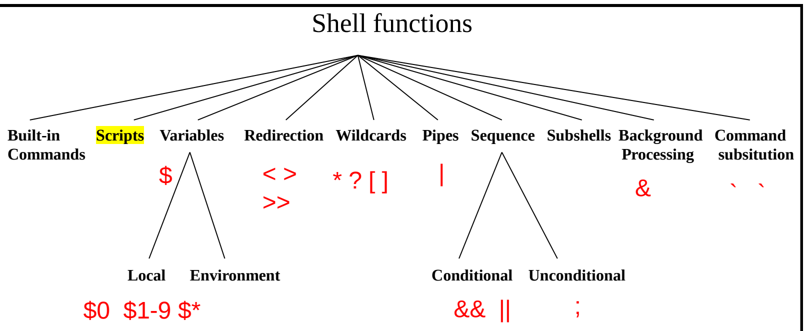
Needed even in sh bash

~~$egrep -w Chan|Chen classlist~~
$egrep -w "Chan|Chen" classlist — $egrep -w 'Chan|Chen' classlist

Always Needed !

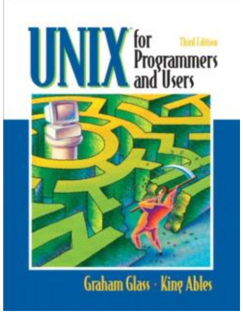
$ find . -name lyrics — $ find . -name 'lyrics' — $ find . -name "lyrics"
~~$ find . -name a?.c~~ — $ find . -name 'a?.c' — $ find . -name "a?.c"
~~$ find . -name *.c~~ — $ find . -name '*.c' — $ find . -name "*.c"

Needed to prevent interpretation of * ?, giving to find as is



- Covered core shell functionality
  - Built-in commands/utilities
  - Redirection < > >>
  - Wildcards (filename substitution) * ? [ ]
  - Pipes |
  - Command substitution ` `
  - Sequence ; conditional sequence && ||
  - Background processing & Grouping ()
  - Variables $ variable substitution
  - Quoting ' ' " "
  - Scripts

Contents



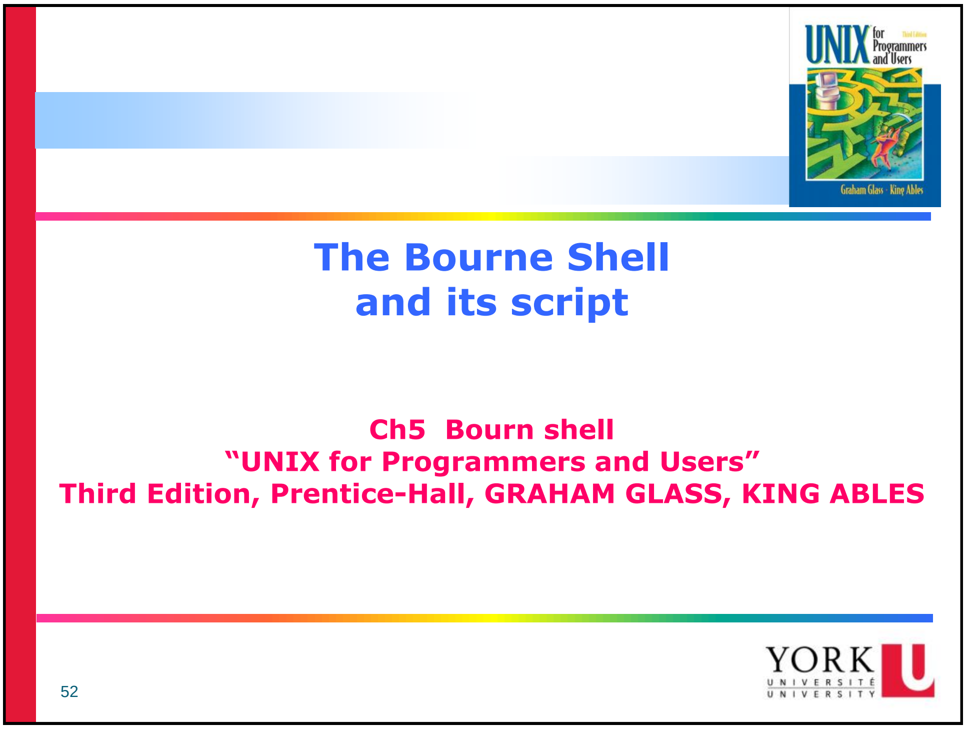
- Overview of UNIX
  - Structures
  - File systems
    - Pathname: absolute vs relative
    - Security -rwx-rw--x
  - Process:
    - Exit code ≥ 0
    - IPC: Pipes

      who | sort   who | sort | head -3   who | grep Wang | wc -l

- Utilities/commands
  - Basic: pwd, ls, rmdir, mkdir, cat, more, mv, cp, rm, wc, chmod
  - Advanced: grep/egrep, sort, cut, find ….

- Shell (common shell functionalities)

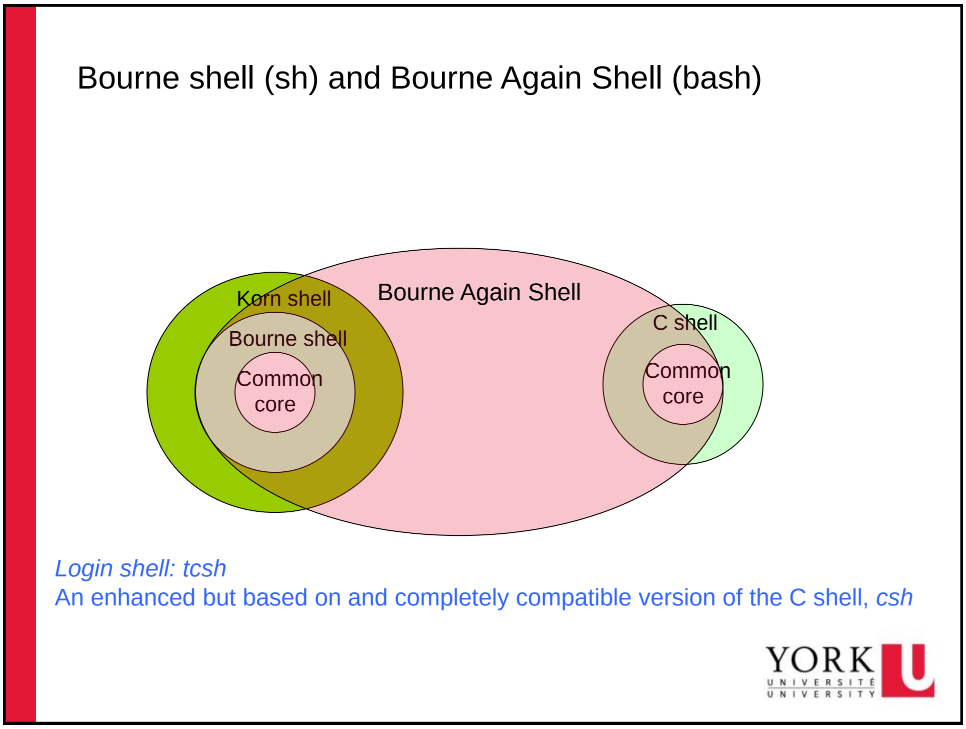
- Bourn (again) Shell
  - scripting language



UNIX for Programmers and Users
Third Edition
Graham Glass · King Ables

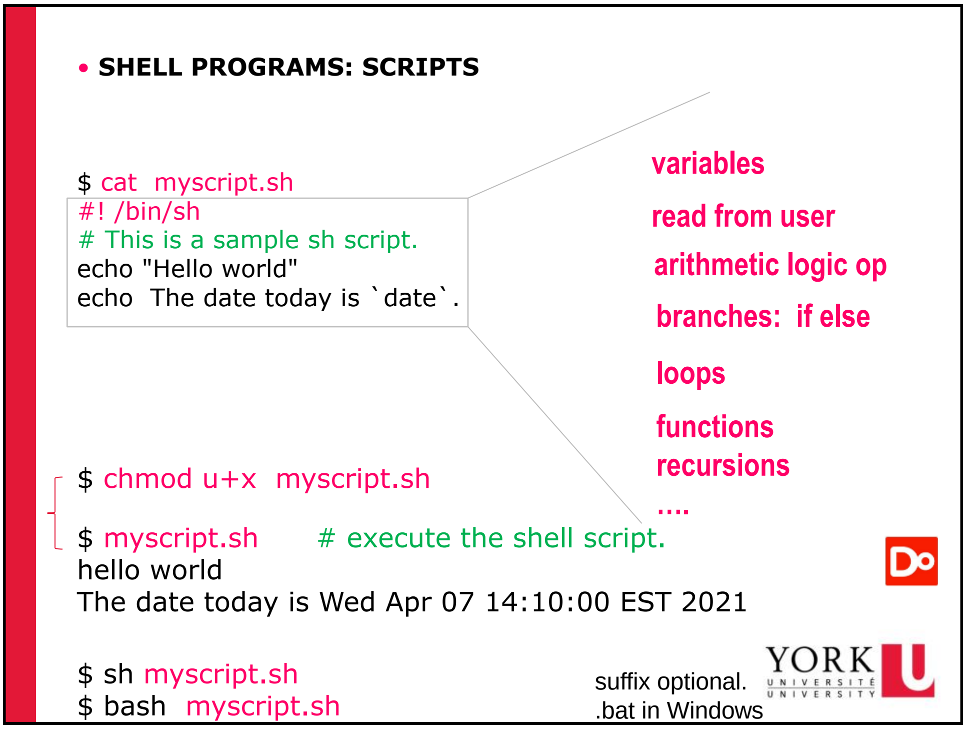
The Bourne Shell
and its script

Ch5 Bourn shell
"UNIX for Programmers and Users"
Third Edition, Prentice-Hall, GRAHAM GLASS, KING ABLES



Bourne shell (sh) and Bourne Again Shell (bash)

Login shell: tcsh
An enhanced but based on and completely compatible version of the C shell, *csh*



- **SHELL PROGRAMS: SCRIPTS**

```
$ cat  myscript.sh
#! /bin/sh
# This is a sample sh script.
echo "Hello world"
echo  The date today is `date`.
```

variables

read from user

arithmetic logic op

branches: if else

loops

functions

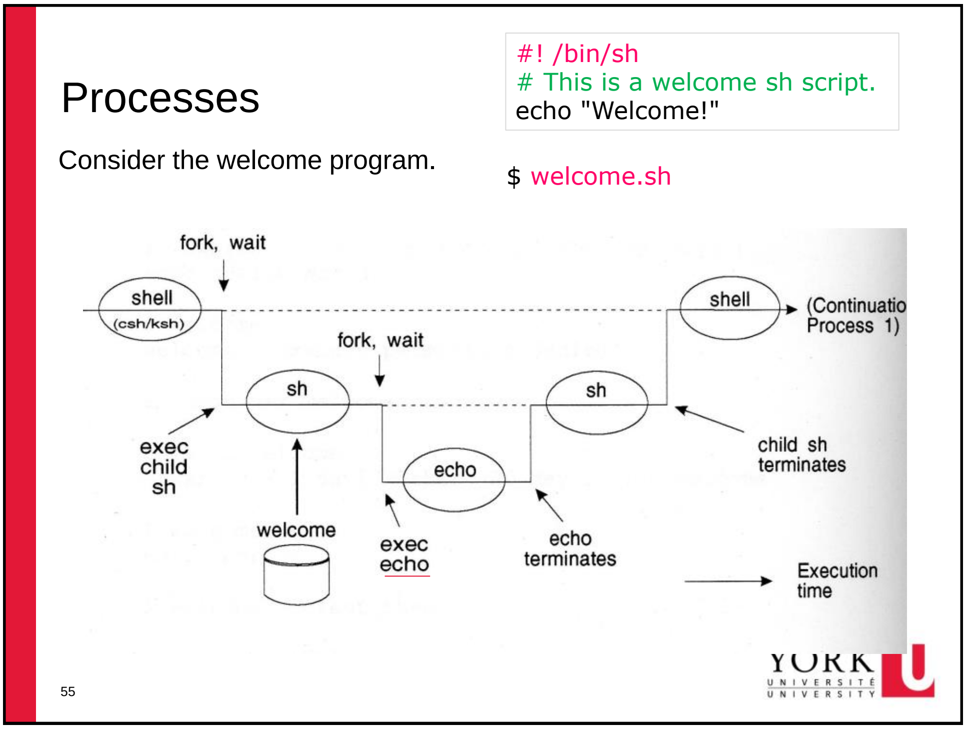
recursions

....

```
$ chmod u+x  myscript.sh

$ myscript.sh       # execute the shell script.
hello world
The date today is Wed Apr 07 14:10:00 EST 2021

$ sh myscript.sh
$ bash  myscript.sh
```

suffix optional.
.bat in Windows

54



Processes

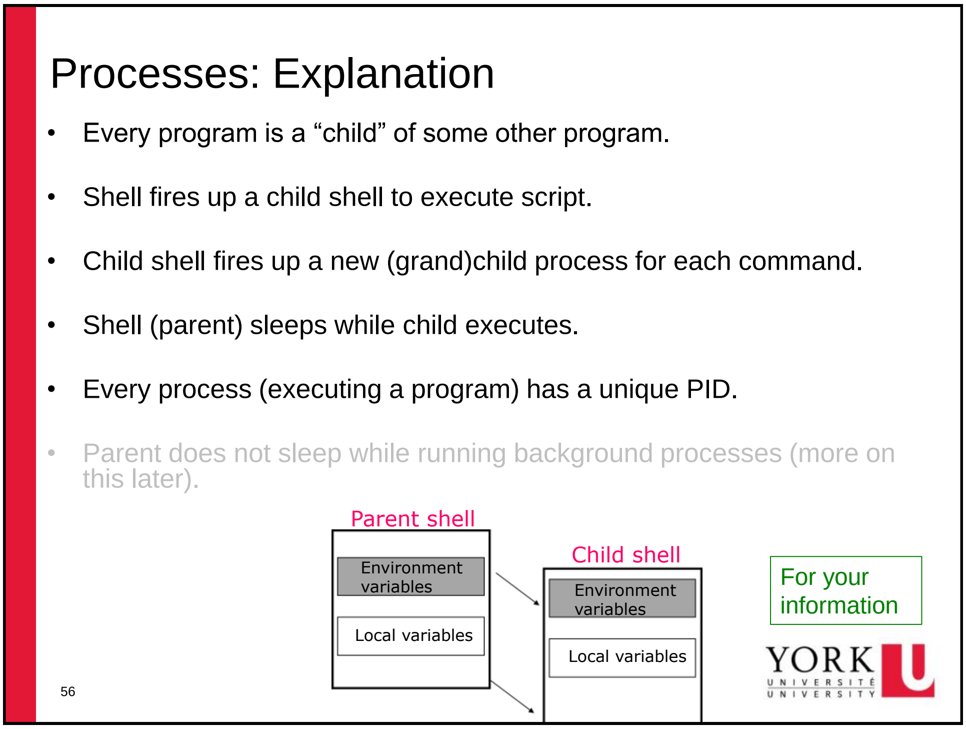
Consider the welcome program.

```
#! /bin/sh
# This is a welcome sh script.
echo "Welcome!"
```

```
$ welcome.sh
```

55

55



Processes: Explanation

- Every program is a “child” of some other program.
- Shell fires up a child shell to execute script.
- Child shell fires up a new (grand)child process for each command.
- Shell (parent) sleeps while child executes.
- Every process (executing a program) has a unique PID.
- Parent does not sleep while running background processes (more on this later).

Parent shell

Environment variables

Local variables

Child shell

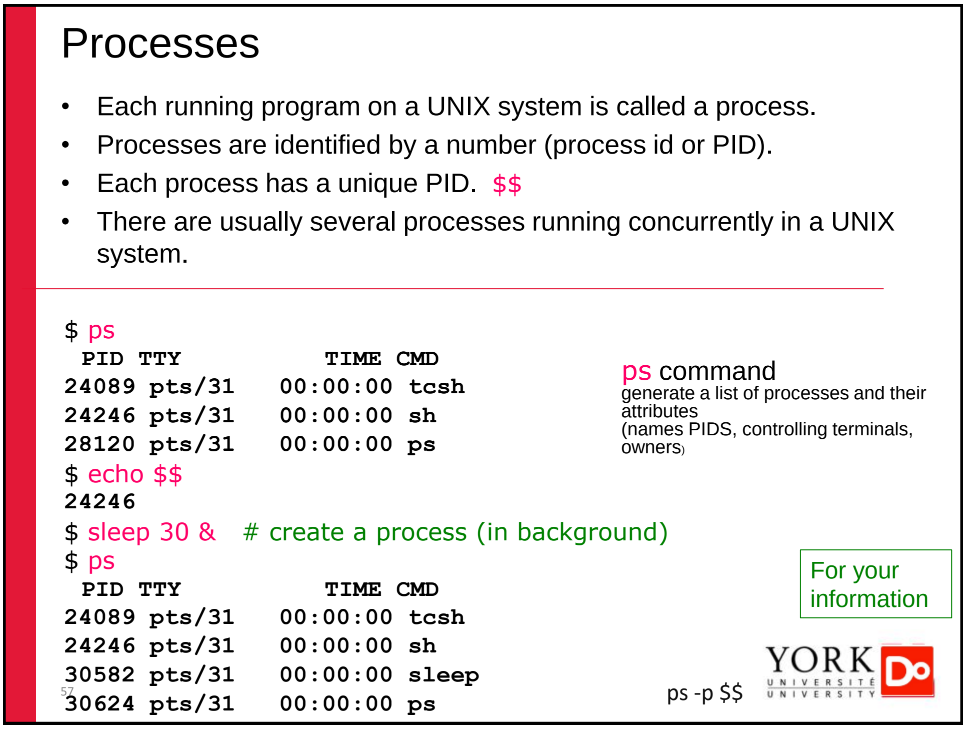
Environment variables

Local variables

For your information

56

56



Processes

- Each running program on a UNIX system is called a process.
- Processes are identified by a number (process id or PID).
- Each process has a unique PID. $$
- There are usually several processes running concurrently in a UNIX system.

```
$ ps
  PID TTY          TIME CMD
24089 pts/31   00:00:00 tcsh
24246 pts/31   00:00:00 sh
28120 pts/31   00:00:00 ps
$ echo $$
24246
$ sleep 30 &   # create a process (in background)
$ ps
  PID TTY          TIME CMD
24089 pts/31   00:00:00 tcsh
24246 pts/31   00:00:00 sh
30582 pts/31   00:00:00 sleep
30624 pts/31   00:00:00 ps
```

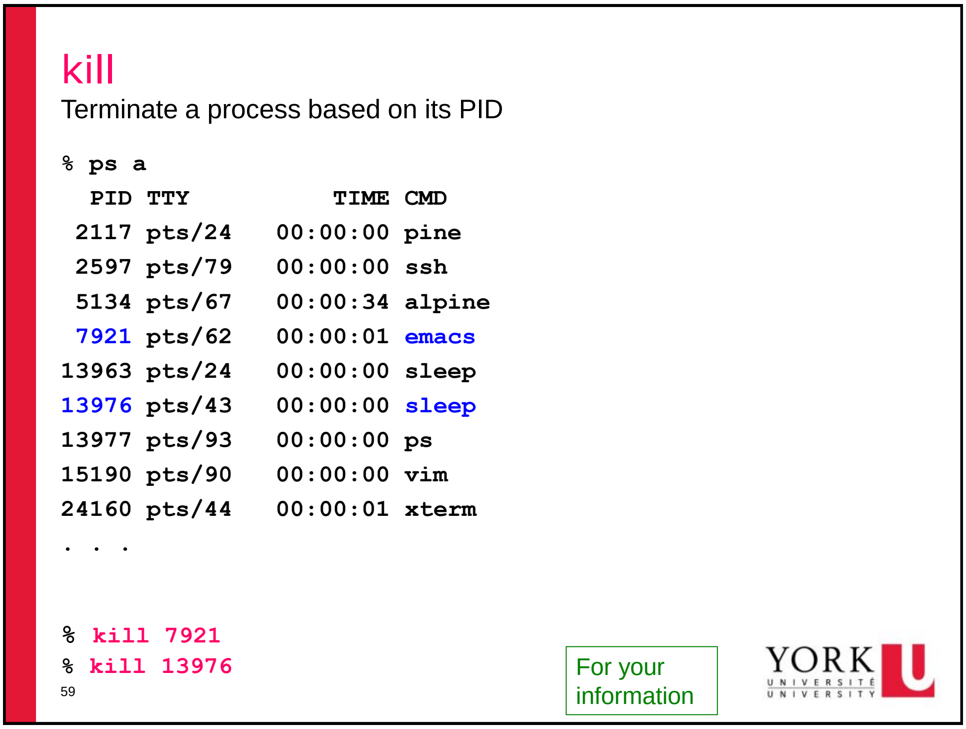
ps command
generate a list of processes and their attributes
(names PIDS, controlling terminals, owners)

ps -p $$

For your information

57

57



kill

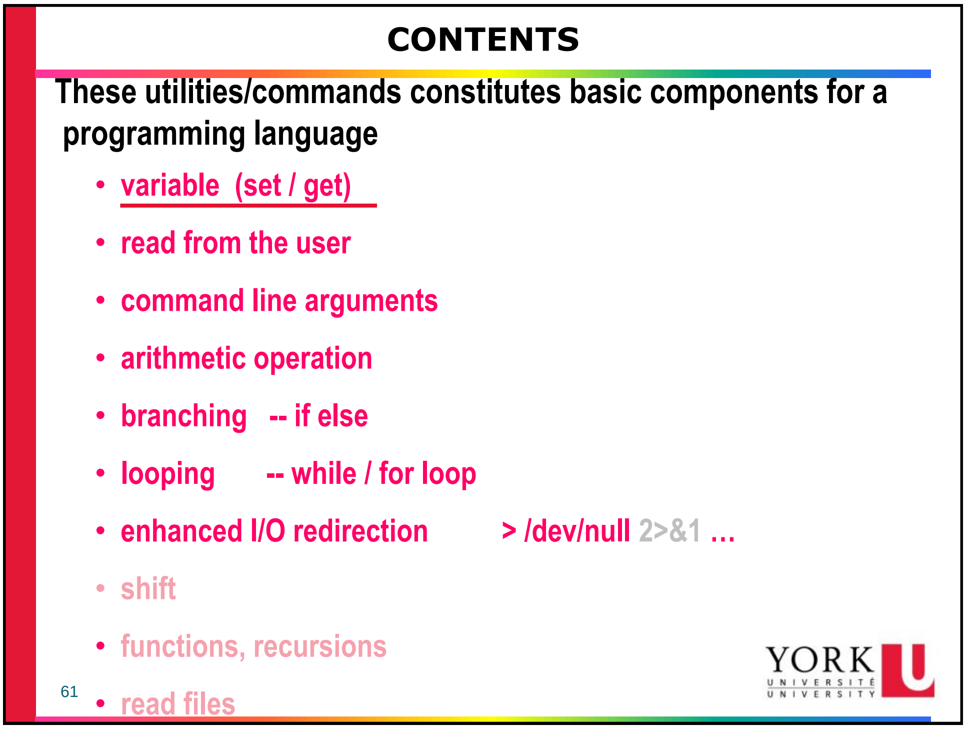
Terminate a process based on its PID

```
% ps a
  PID TTY          TIME CMD
 2117 pts/24   00:00:00 pine
 2597 pts/79   00:00:00 ssh
 5134 pts/67   00:00:34 alpine
 7921 pts/62   00:00:01 emacs
13963 pts/24   00:00:00 sleep
13976 pts/43   00:00:00 sleep
13977 pts/93   00:00:00 ps
15190 pts/90   00:00:00 vim
24160 pts/44   00:00:01 xterm
. . .


% kill 7921
% kill 13976
```

For your information

YORK UNIVERSITÉ UNIVERSITY

59



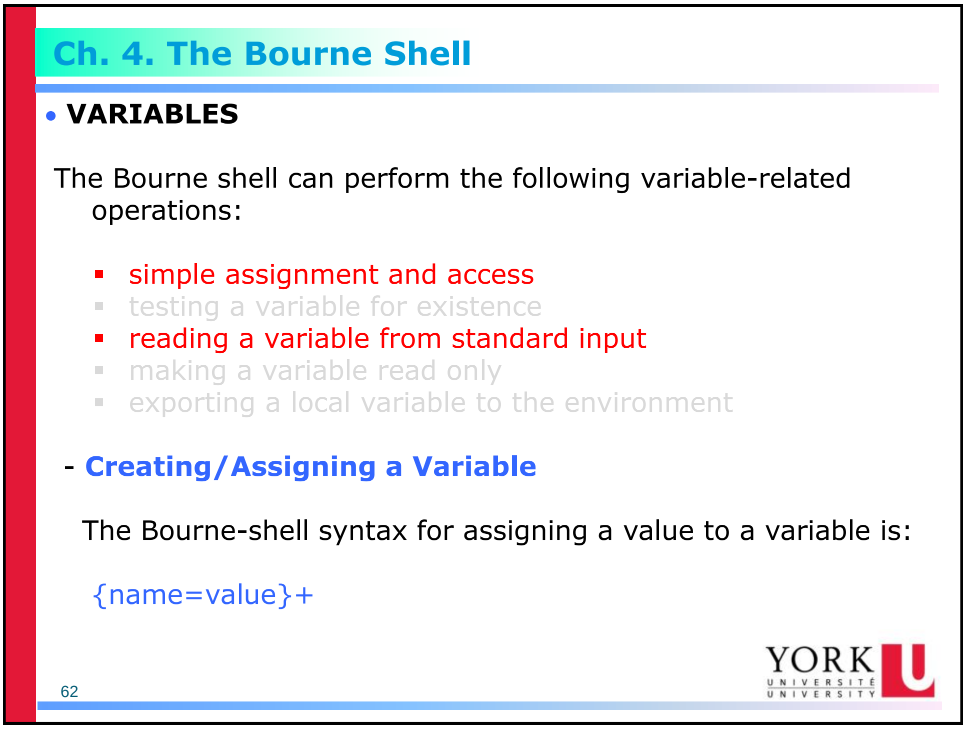
CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- enhanced I/O redirection > /dev/null 2>&1 …
- shift
- functions, recursions
- read files

YORK UNIVERSITÉ UNIVERSITY

61



Ch. 4. The Bourne Shell

- **VARIABLES**

The Bourne shell can perform the following variable-related operations:

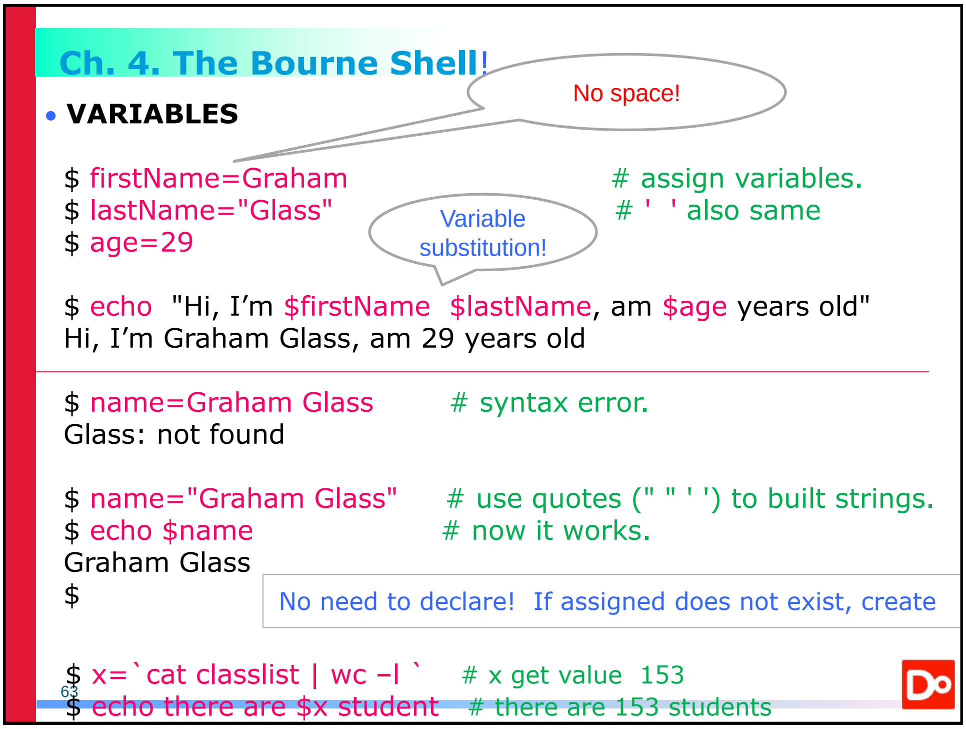
- simple assignment and access
- testing a variable for existence
- reading a variable from standard input
- making a variable read only
- exporting a local variable to the environment

- **Creating/Assigning a Variable**

The Bourne-shell syntax for assigning a value to a variable is:

```
{name=value}+
```

62



Ch. 4. The Bourne Shell!

- **VARIABLES**

No space!

```
$ firstName=Graham                         # assign variables.
$ lastName="Glass"                         # '  ' also same
$ age=29
```

Variable substitution!

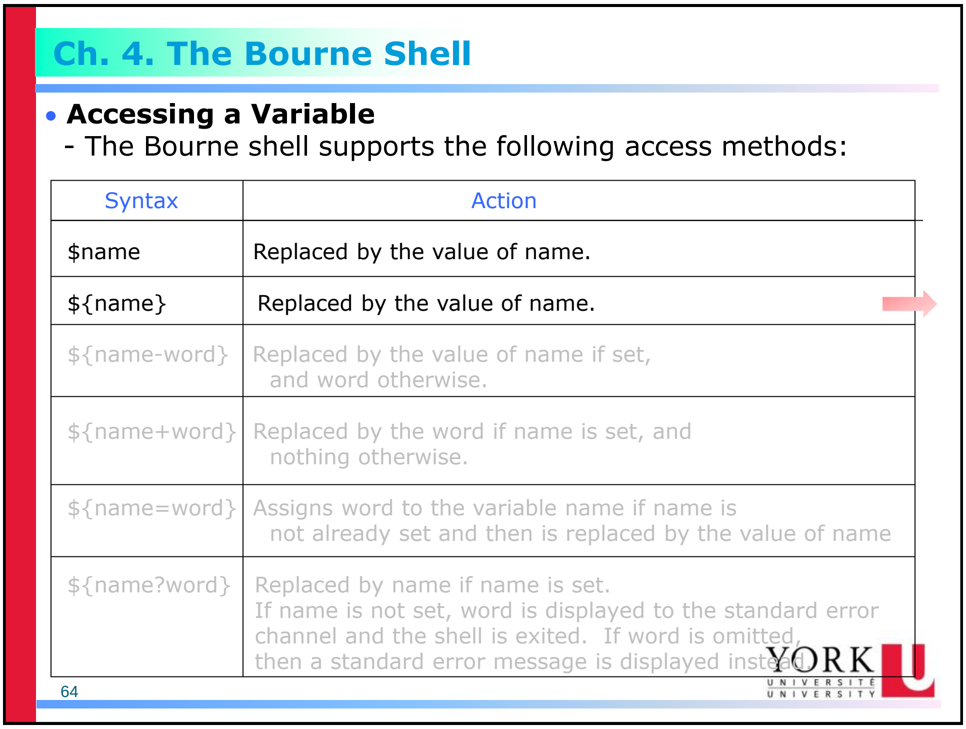
```
$ echo  "Hi, I’m $firstName  $lastName, am $age years old"
Hi, I’m Graham Glass, am 29 years old
```

```
$ name=Graham Glass        # syntax error.
Glass: not found

$ name="Graham Glass"      # use quotes (" " ' ') to built strings.
$ echo $name               # now it works.
Graham Glass
$
```

No need to declare! If assigned does not exist, create

```
$ x=`cat classlist | wc –l `     # x get value  153
$ echo there are $x student      # there are 153 students
```

63

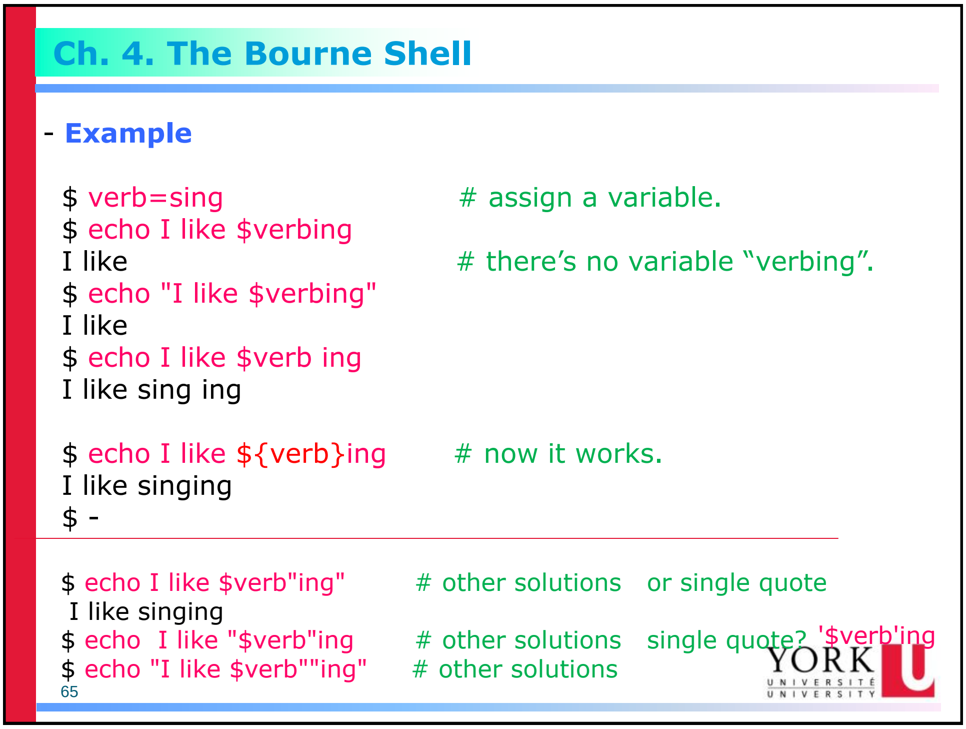


Ch. 4. The Bourne Shell

- **Accessing a Variable**

  - The Bourne shell supports the following access methods:

Syntax	Action
$name	Replaced by the value of name.
${name}	Replaced by the value of name.
${name-word}	Replaced by the value of name if set, and word otherwise.
${name+word}	Replaced by the word if name is set, and nothing otherwise.
${name=word}	Assigns word to the variable name if name is not already set and then is replaced by the value of name
${name?word}	Replaced by name if name is set. If name is not set, word is displayed to the standard error channel and the shell is exited. If word is omitted, then a standard error message is displayed instead.

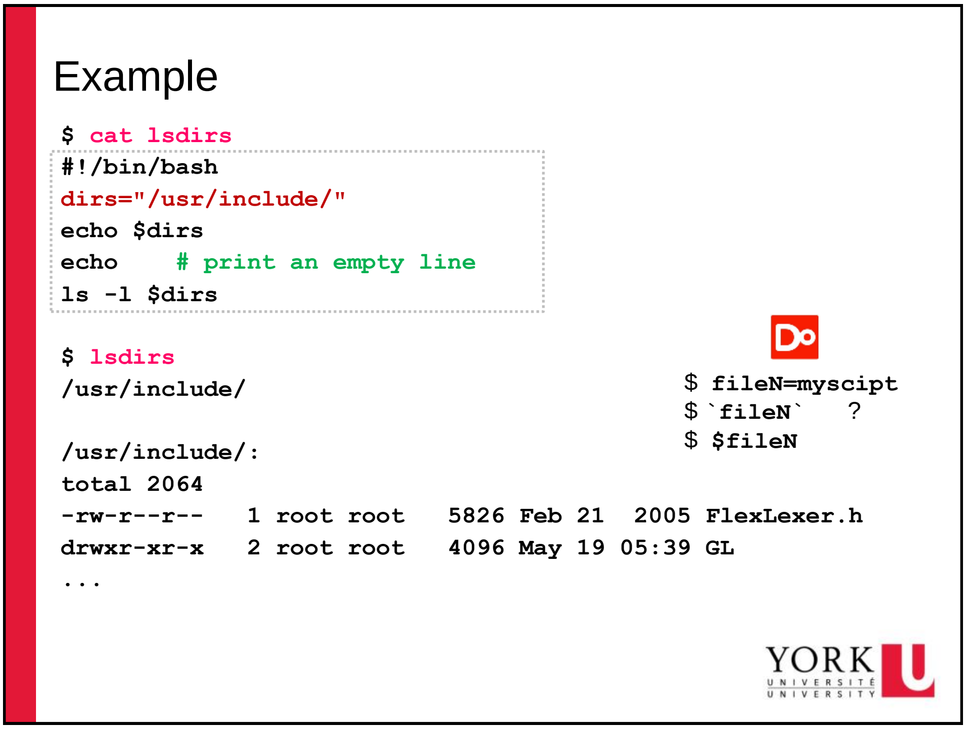


Ch. 4. The Bourne Shell

- **Example**

```
$ verb=sing                     # assign a variable.
$ echo I like $verbing
I like                          # there's no variable "verbing".
$ echo "I like $verbing"
I like
$ echo I like $verb ing
I like sing ing

$ echo I like ${verb}ing        # now it works.
I like singing
$ -
```

```
$ echo I like $verb"ing"       # other solutions   or single quote
 I like singing
$ echo  I like "$verb"ing      # other solutions   single quote? '$verb'ing
$ echo "I like $verb""ing"     # other solutions
```



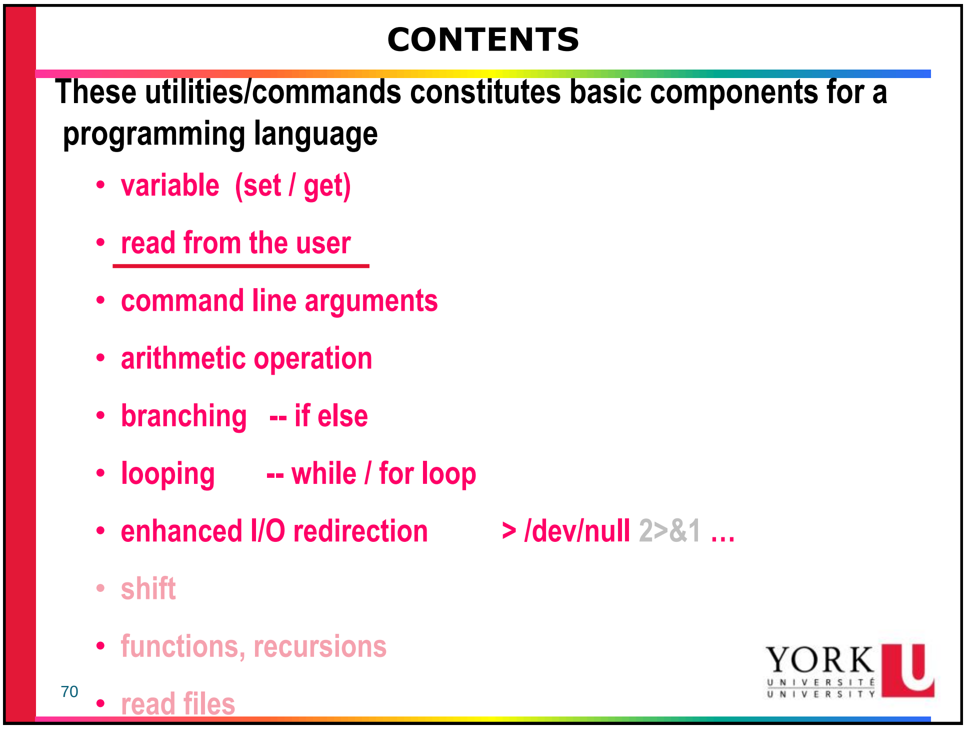
Example

```
$ cat lsdirs
#!/bin/bash
dirs="/usr/include/"
echo $dirs
echo     # print an empty line
ls -l $dirs
```

```
$ lsdirs
/usr/include/

/usr/include/:
total 2064
-rw-r--r--   1 root root   5826 Feb 21  2005 FlexLexer.h
drwxr-xr-x   2 root root   4096 May 19 05:39 GL
...
```

```
$ fileN=myscipt
$ `fileN`   ?
$ $fileN
```

67



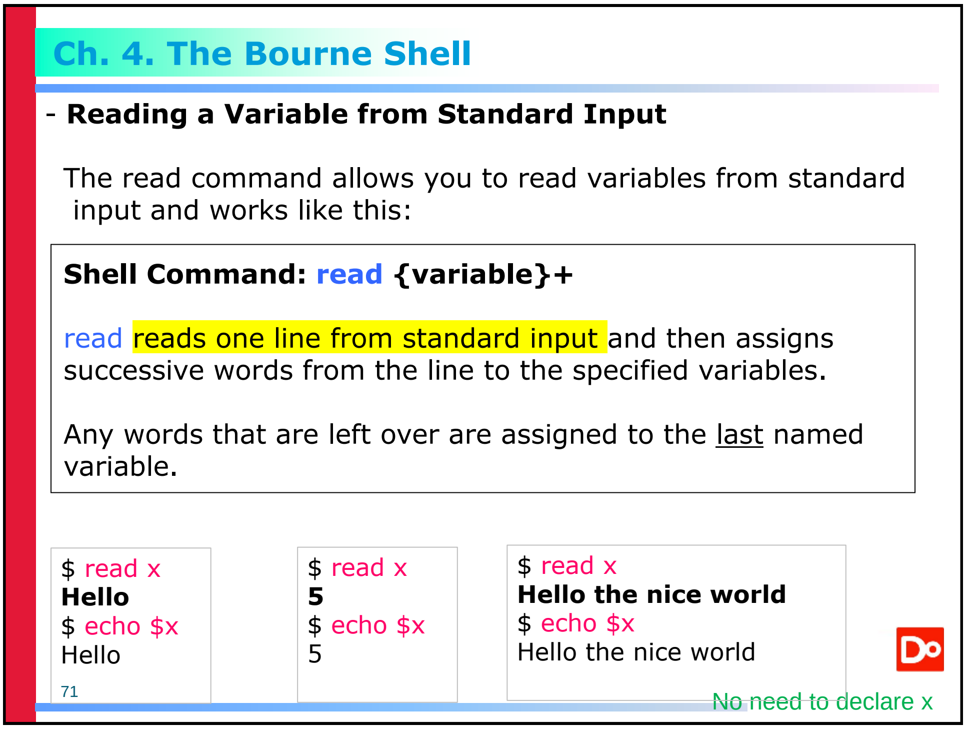
CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- enhanced I/O redirection > /dev/null 2>&1 ...
- shift
- functions, recursions
- read files

70

70



Ch. 4. The Bourne Shell

- Reading a Variable from Standard Input

The read command allows you to read variables from standard input and works like this:

Shell Command: read {variable}+

read reads one line from standard input and then assigns successive words from the line to the specified variables.

Any words that are left over are assigned to the last named variable.

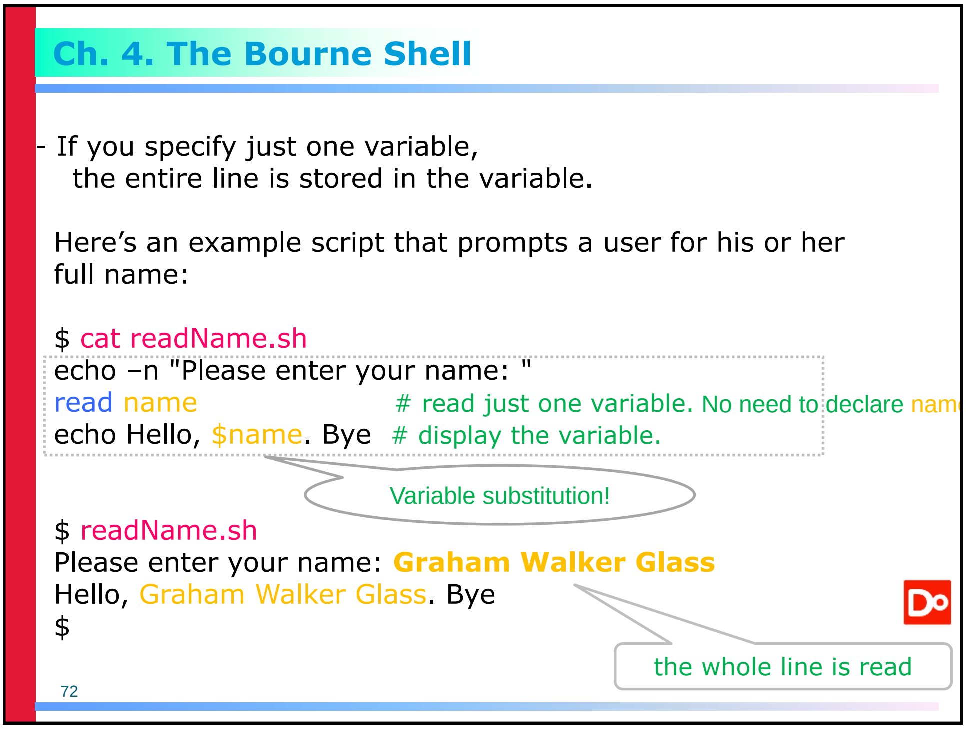
```
$ read x
Hello
$ echo $x
Hello
```

```
$ read x
5
$ echo $x
5
```

```
$ read x
Hello the nice world
$ echo $x
Hello the nice world
```

No need to declare x

71



Ch. 4. The Bourne Shell

- If you specify just one variable, the entire line is stored in the variable.

Here's an example script that prompts a user for his or her full name:

```
$ cat readName.sh
echo –n "Please enter your name: "
read name                # read just one variable. No need to declare nam
echo Hello, $name. Bye  # display the variable.
```

Variable substitution!

```
$ readName.sh
Please enter your name: Graham Walker Glass
Hello, Graham Walker Glass. Bye
$
```

the whole line is read

72

Ch. 4. The Bourne Shell

```
$ read a
1 2 3 4 5 6 7
$ echo $a
1 2 3 4 5 6 7
```

```
$ read a b
1 2 3 4 5 6 7
$ echo $a
1
$ echo $b
2 3 4 5 6 7
```

```
$ read a b  c
1 2  3  4 5 6 7
$ echo $a
1
$ echo $b
2
$ echo $c
3 4 5 6 7
```

```
$ read a b  c
1 2
$ echo $a
1
$ echo $b
2
$ echo $c
```

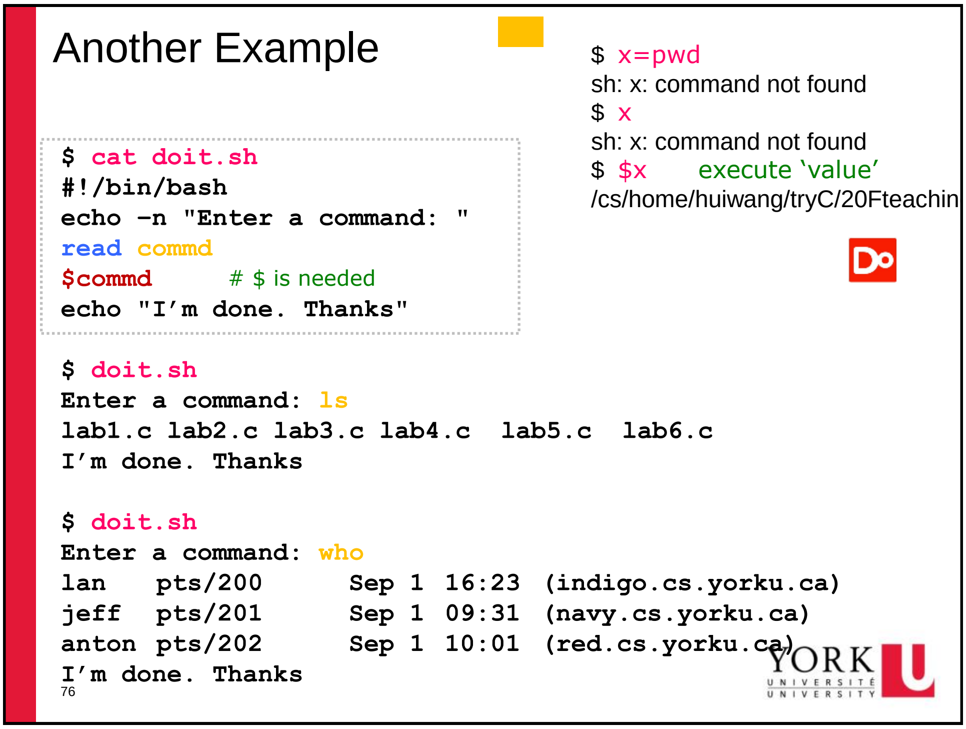
- Here's other example script

```
$ cat readNames.sh
echo –n "Please enter your name: "
read first last                 # read two variables. no need to declare first, last
echo your first name is $first      # display the variables.
echo your last name is  $last

$ readNames.sh
Please enter your name: Graham Walker Glass
your first name is Graham
your last name is  Walker Glass         # the whole rest line is read.
$
```

Ch. 4. The Bourne Shell

- Here's other example script

```
$ cat mygrep.sh
echo –n "Please enter file to search: "
read file                           # read two variables.
echo –n "Please enter search key: "
read pattern
grep –w $pattern  $file         # display the variables.


$ mygrep.sh
Please enter file to search: classlist
Please enter search key: Wong
wcad*          *********      Wong    FengC
cse***         *********      Wong    JunXiu

$ mygrep.sh
Please enter file to search: classlist
Please enter search key: Leung
$
```

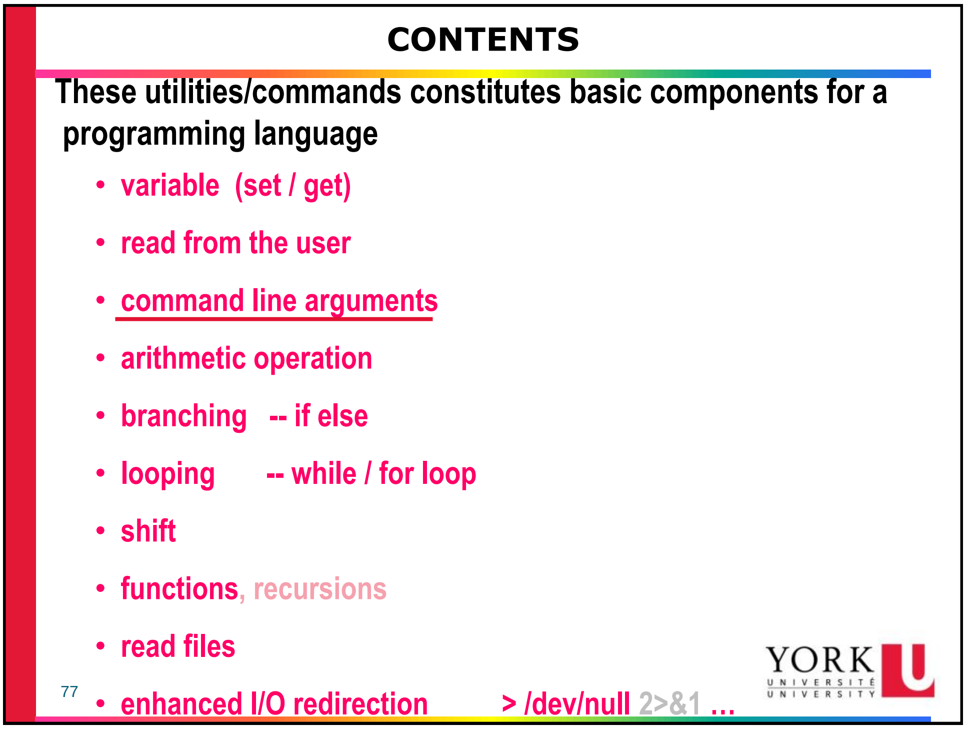


Another Example

```
$ cat doit.sh
#!/bin/bash
echo -n "Enter a command: "
read commd
$commd        # $ is needed
echo "I'm done. Thanks"
```

```
$ x=pwd
sh: x: command not found
$ x
sh: x: command not found
$ $x      execute `value'
/cs/home/huiwang/tryC/20Fteachin
```

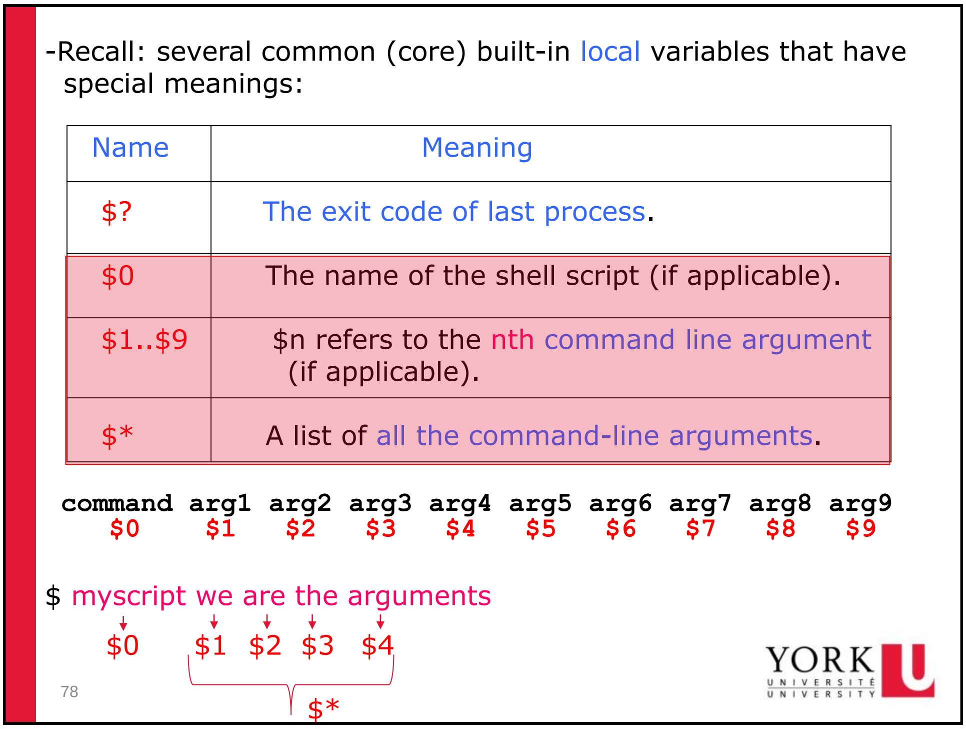
```
$ doit.sh
Enter a command: ls
lab1.c lab2.c lab3.c lab4.c  lab5.c  lab6.c
I'm done. Thanks

$ doit.sh
Enter a command: who
lan    pts/200     Sep 1 16:23  (indigo.cs.yorku.ca)
jeff   pts/201     Sep 1 09:31  (navy.cs.yorku.ca)
anton  pts/202     Sep 1 10:01  (red.cs.yorku.ca)
I'm done. Thanks
```



CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- shift
- functions, recursions
- read files
- enhanced I/O redirection > /dev/null 2>&1 ...



-Recall: several common (core) built-in local variables that have special meanings:

Name	Meaning
$?	The exit code of last process.
$0	The name of the shell script (if applicable).
$1..$9	$n refers to the nth command line argument (if applicable).
$*	A list of all the command-line arguments.

```
command arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
  $0     $1   $2   $3   $4   $5   $6   $7   $8   $9
```

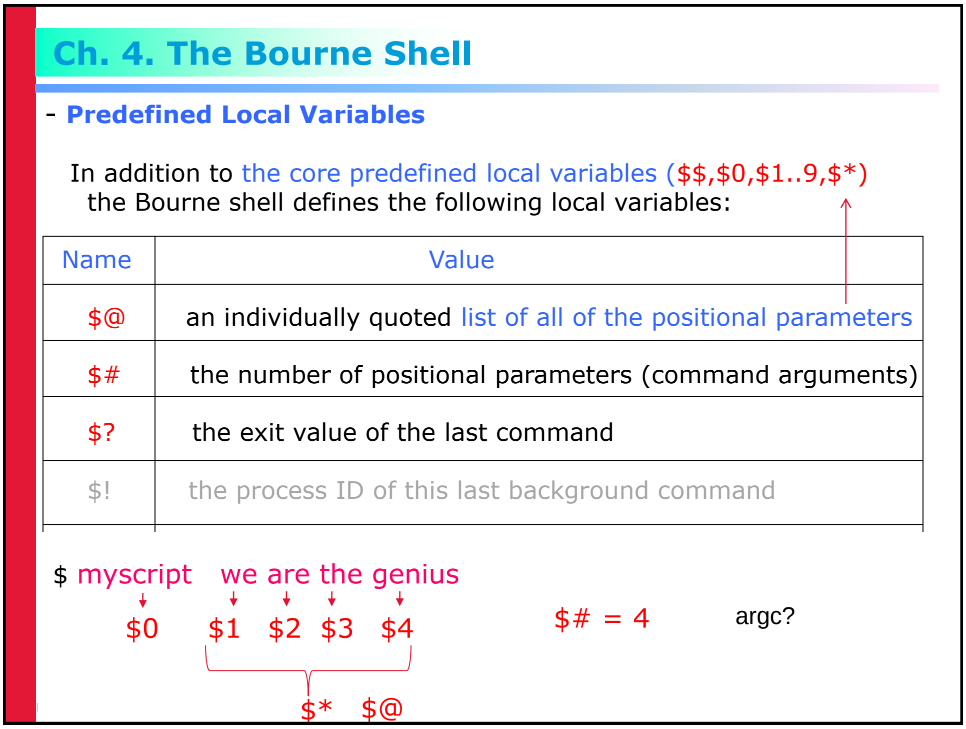
```
$ myscript we are the arguments
     $0    $1  $2  $3   $4
```

$*

78

YORK U UNIVERSITÉ UNIVERSITY

78



Ch. 4. The Bourne Shell

- Predefined Local Variables

In addition to the core predefined local variables ($$,$0,$1..9,$*) the Bourne shell defines the following local variables:

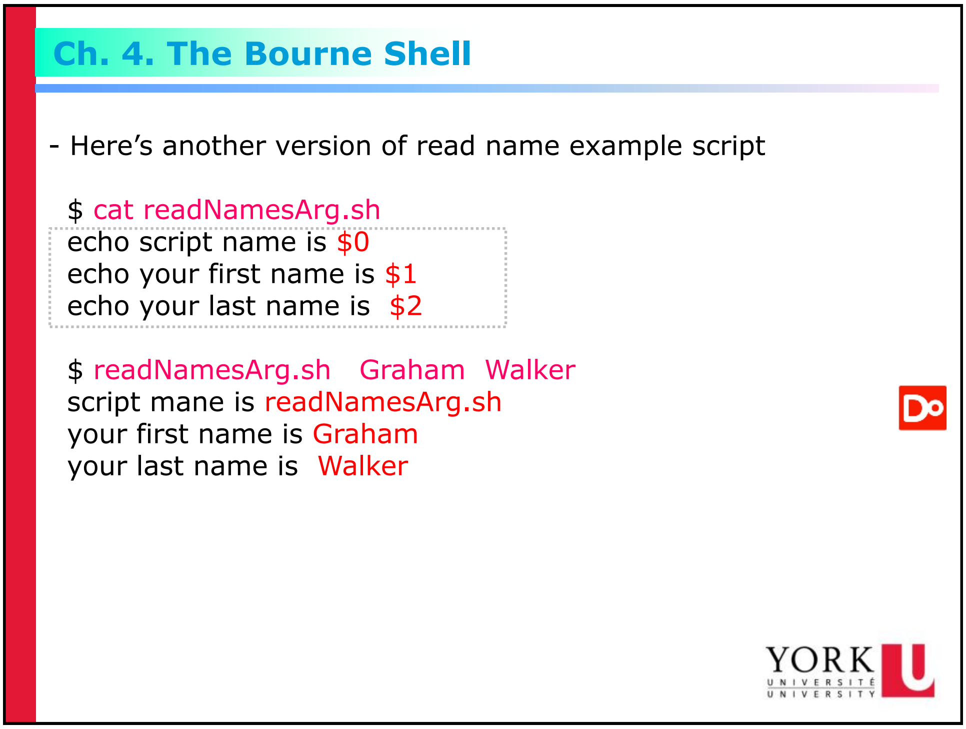
Name	Value
$@	an individually quoted list of all of the positional parameters
$#	the number of positional parameters (command arguments)
$?	the exit value of the last command
$!	the process ID of this last background command

```
$ myscript   we are the genius
     $0      $1  $2  $3   $4
```

$# = 4 argc?

$* $@

79

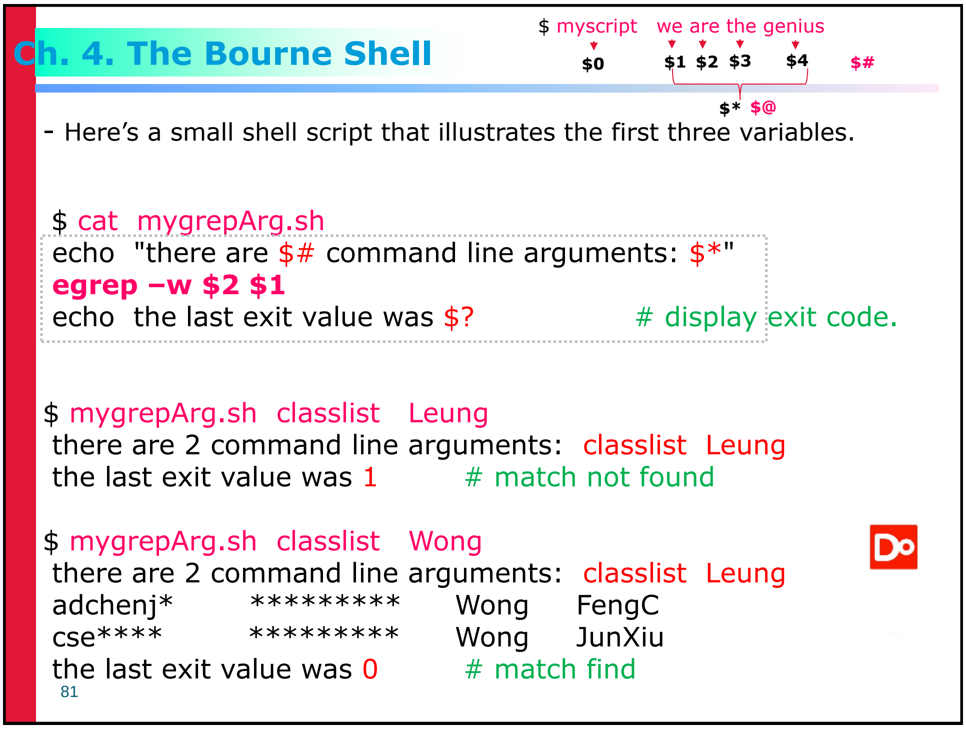


Ch. 4. The Bourne Shell

- Here’s another version of read name example script

```
$ cat readNamesArg.sh
echo script name is $0
echo your first name is $1
echo your last name is  $2

$ readNamesArg.sh   Graham  Walker
script mane is readNamesArg.sh
your first name is Graham
your last name is  Walker
```

80



Ch. 4. The Bourne Shell

$ myscript   we are the genius

$0 $1 $2 $3 $4 $#

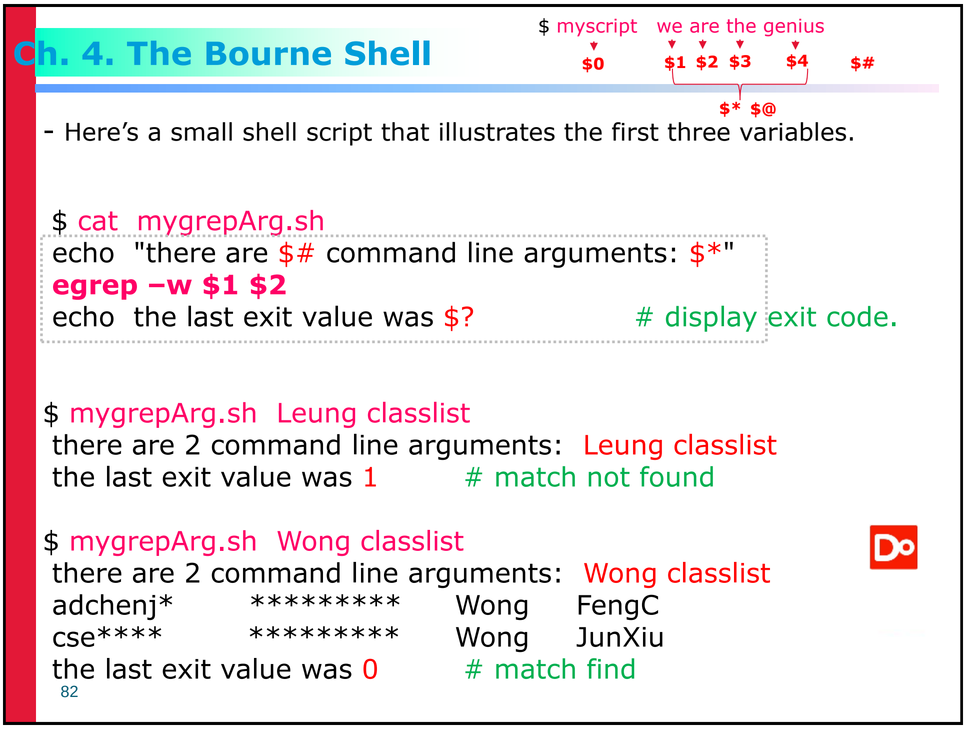
$* $@

- Here’s a small shell script that illustrates the first three variables.

```
$ cat  mygrepArg.sh
echo  "there are $# command line arguments: $*"
egrep –w $2 $1
echo  the last exit value was $?                 # display exit code.

$ mygrepArg.sh  classlist   Leung
there are 2 command line arguments:  classlist  Leung
the last exit value was 1           # match not found

$ mygrepArg.sh  classlist   Wong
there are 2 command line arguments:  classlist  Leung
adchenj*        *********      Wong     FengC
cse****         *********      Wong     JunXiu
the last exit value was 0           # match find
```

81

81



Ch. 4. The Bourne Shell

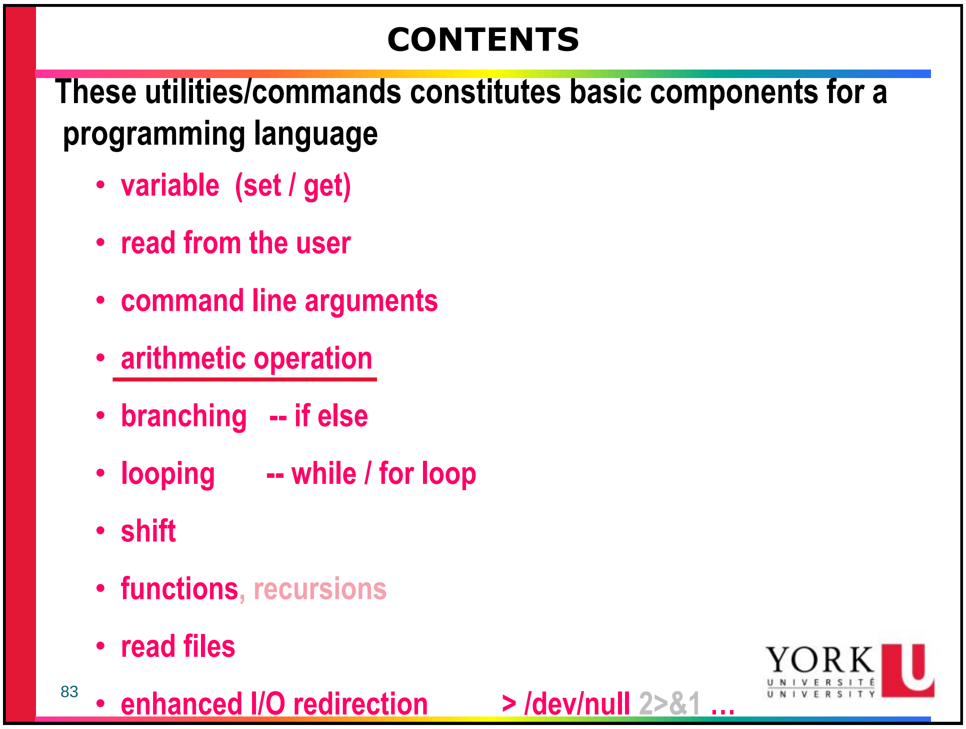
```
$ myscript   we are the genius
   $0        $1  $2  $3   $4     $#
                  $* $@
```

- Here’s a small shell script that illustrates the first three variables.

```
$ cat  mygrepArg.sh
echo  "there are $# command line arguments: $*"
egrep –w $1 $2
echo  the last exit value was $?                # display exit code.
```

```
$ mygrepArg.sh  Leung classlist
there are 2 command line arguments:  Leung classlist
the last exit value was 1          # match not found

$ mygrepArg.sh  Wong classlist
there are 2 command line arguments:  Wong classlist
adchenj*        *********     Wong     FengC
cse****         *********     Wong     JunXiu
the last exit value was 0          # match find
```

82

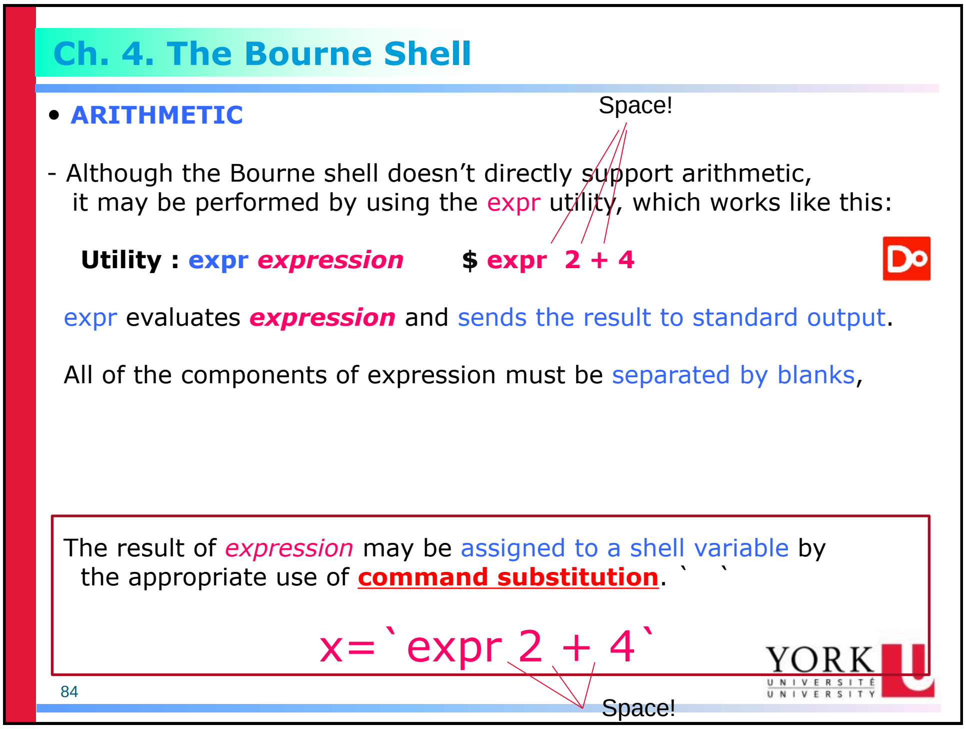


CONTENTS

These utilities/commands constitutes basic components for a programming language

- variable (set / get)
- read from the user
- command line arguments
- arithmetic operation
- branching -- if else
- looping -- while / for loop
- shift
- functions, recursions
- read files
- enhanced I/O redirection > /dev/null 2>&1 ...

83



Ch. 4. The Bourne Shell

- **ARITHMETIC**

Space!

- Although the Bourne shell doesn't directly support arithmetic, it may be performed by using the expr utility, which works like this:

Utility : expr *expression*   **$ expr 2 + 4**

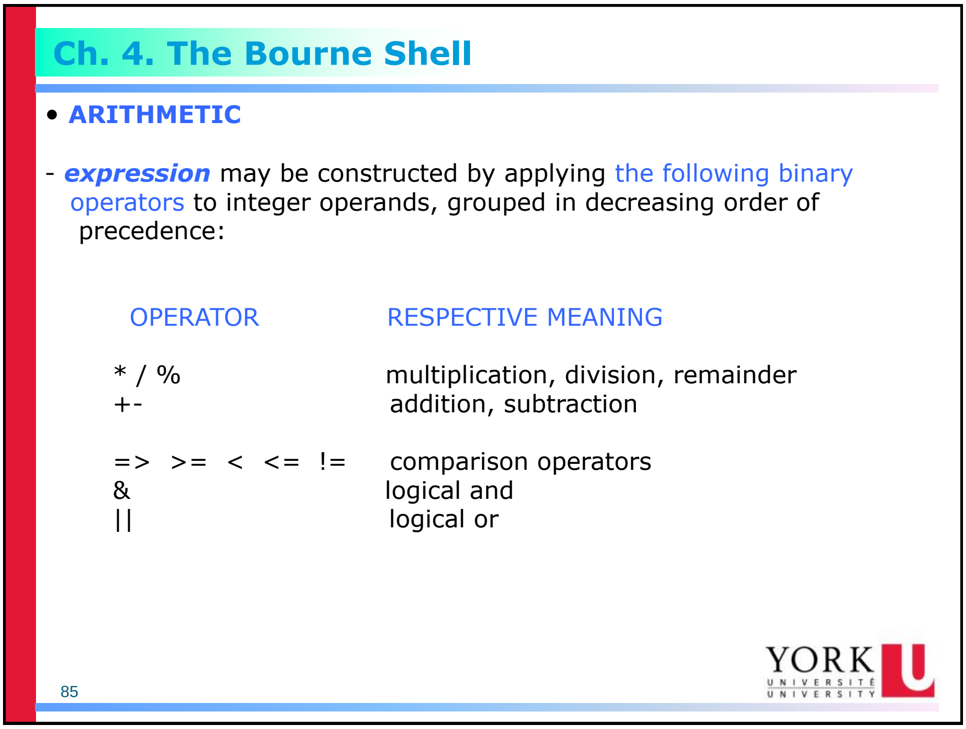
expr evaluates ***expression*** and sends the result to standard output.

All of the components of expression must be separated by blanks,

The result of *expression* may be assigned to a shell variable by the appropriate use of **command substitution**. `  `

```
x=`expr 2 + 4`
```

Space!

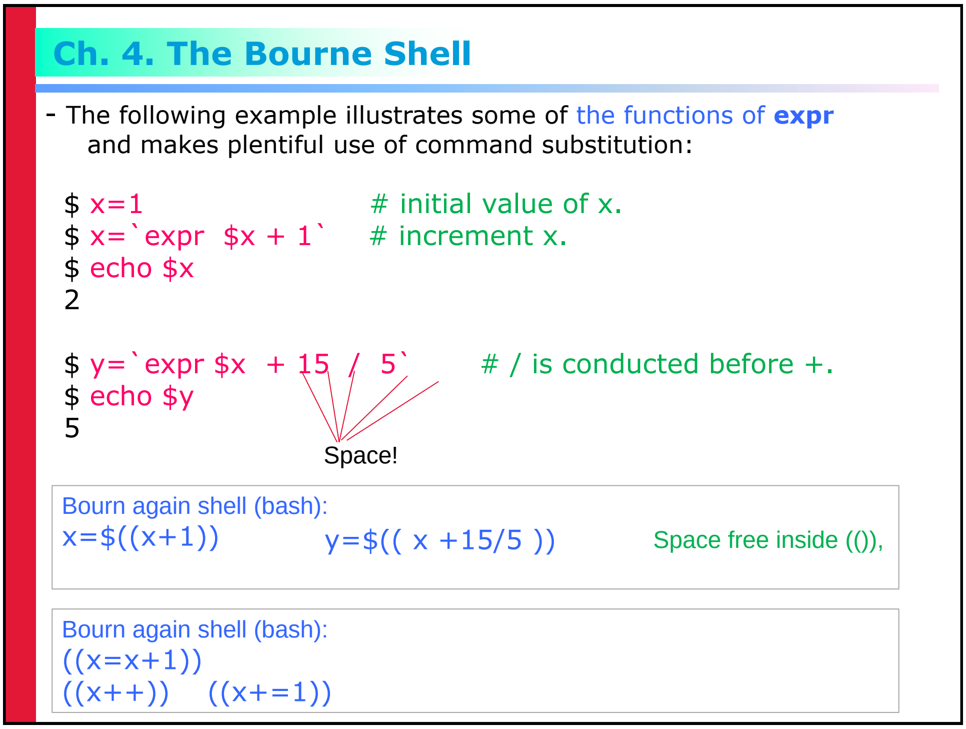


Ch. 4. The Bourne Shell

- **ARITHMETIC**

- ***expression*** may be constructed by applying the following binary operators to integer operands, grouped in decreasing order of precedence:

OPERATOR	RESPECTIVE MEANING
* / %	multiplication, division, remainder
+-	addition, subtraction
=> >= < <= !=	comparison operators
&	logical and
||	logical or



Ch. 4. The Bourne Shell

- The following example illustrates some of the functions of **expr** and makes plentiful use of command substitution:

```
$ x=1                 # initial value of x.
$ x=`expr  $x + 1`    # increment x.
$ echo $x
2

$ y=`expr $x  + 15  /  5`      # / is conducted before +.
$ echo $y
5
```

Space!

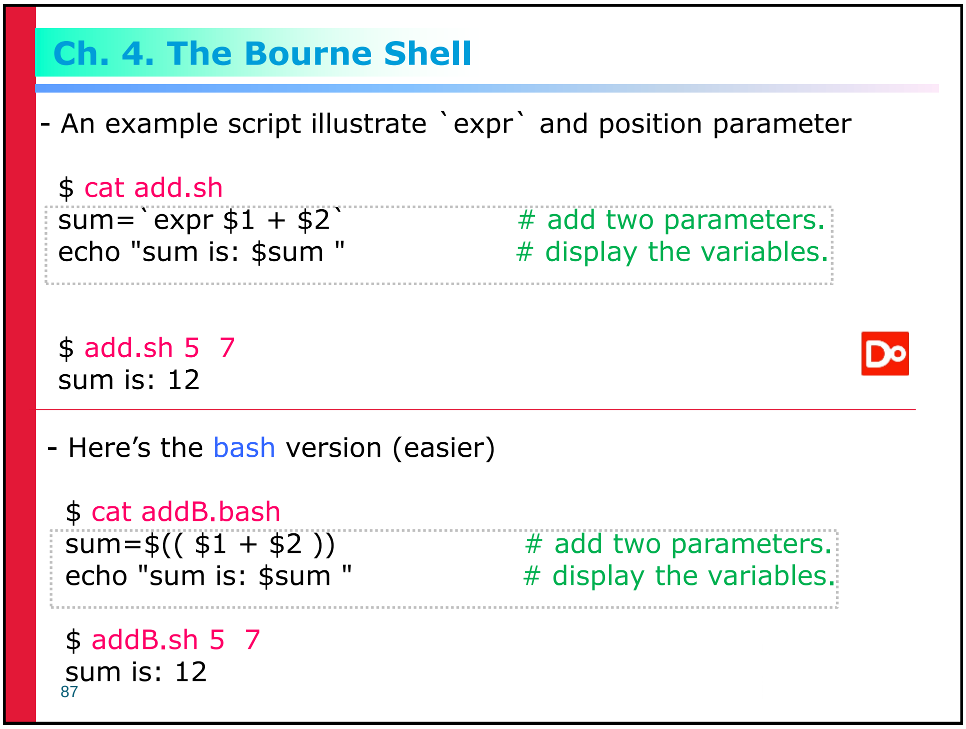
Bourn again shell (bash):

```
x=$((x+1))          y=$(( x +15/5 ))
```

Space free inside (()),

Bourn again shell (bash):

```
((x=x+1))
((x++))    ((x+=1))
```



Ch. 4. The Bourne Shell

- An example script illustrate `expr` and position parameter

```
$ cat add.sh
sum=`expr $1 + $2`                # add two parameters.
echo "sum is: $sum "              # display the variables.

$ add.sh 5  7
sum is: 12
```

- Here's the bash version (easier)

```
$ cat addB.bash
sum=$(( $1 + $2 ))                # add two parameters.
echo "sum is: $sum "              # display the variables.

$ addB.sh 5  7
sum is: 12
```