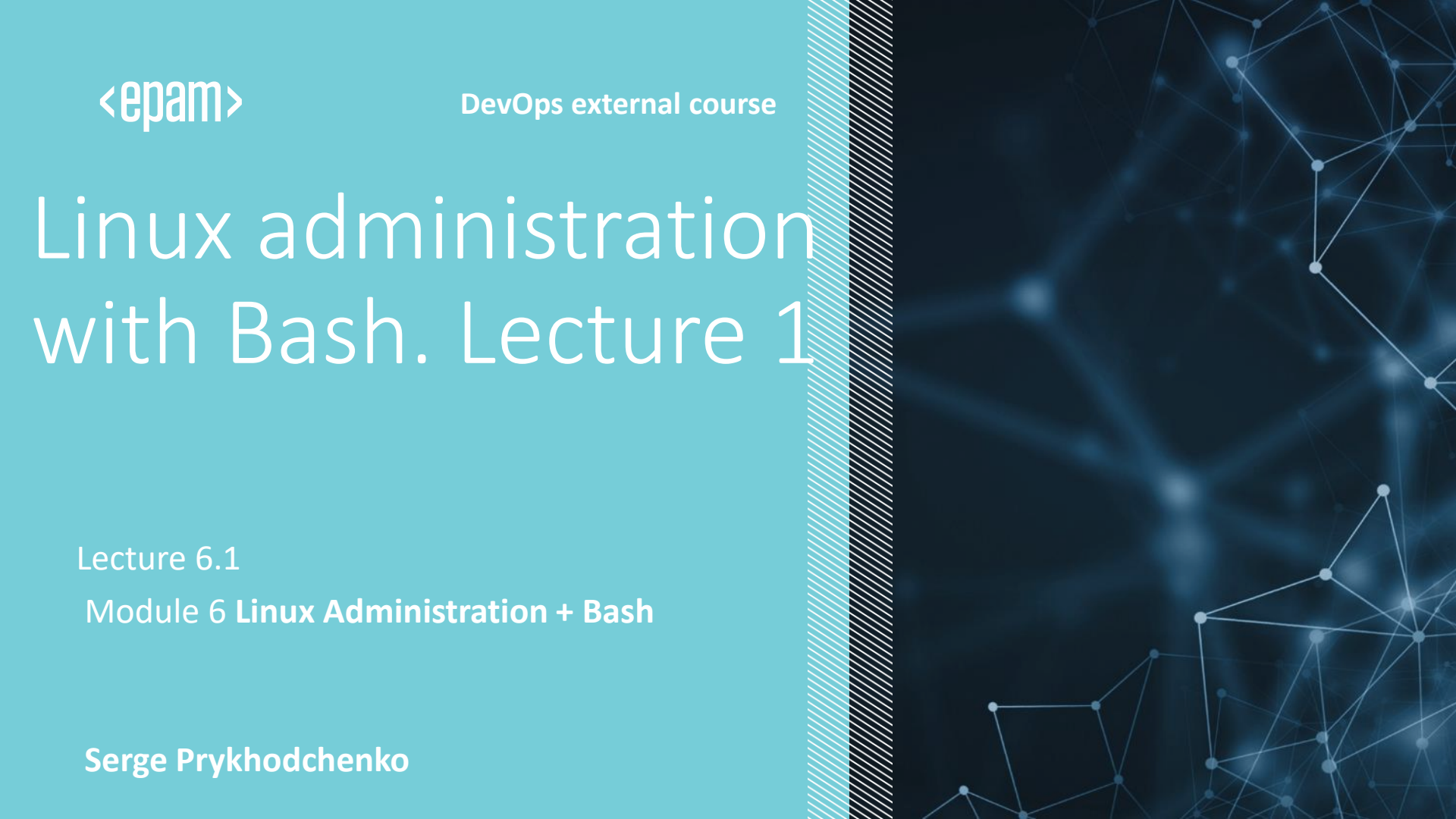

epam
DevOps external course
Linux administration with Bash. Lecture 1
Lecture 6.1
Module 6 Linux Administration + Bash
Serge Prykhodchenko

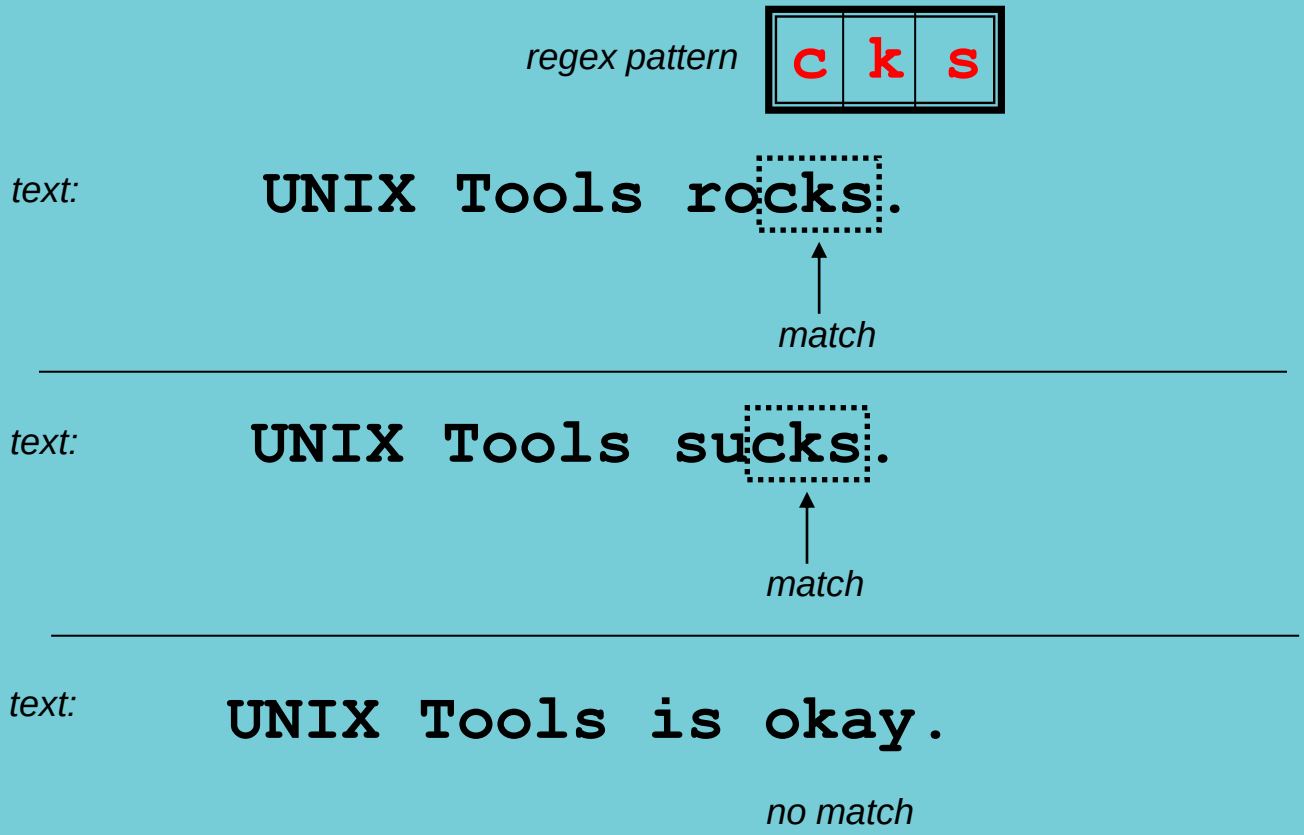
REGULAR EXPRESSIONS

1. Regular Expressions

- A regular expression (RE or *regex*) is a **pattern** used to **match** against **text** when searching inside a file.

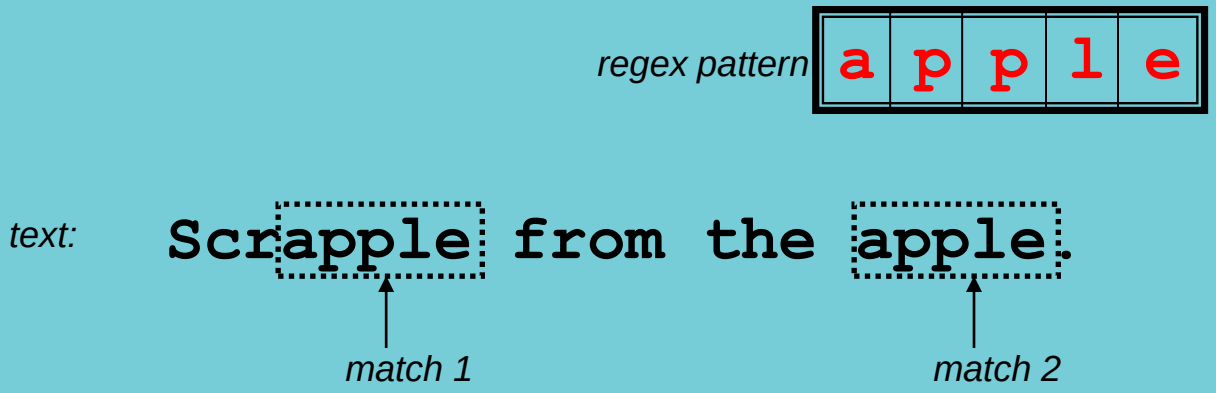
- Regexs are used everywhere in Linux:
  - Editors: ed, ex, vi
  - Utilities: grep, egrep, sed, and awk

String Regex



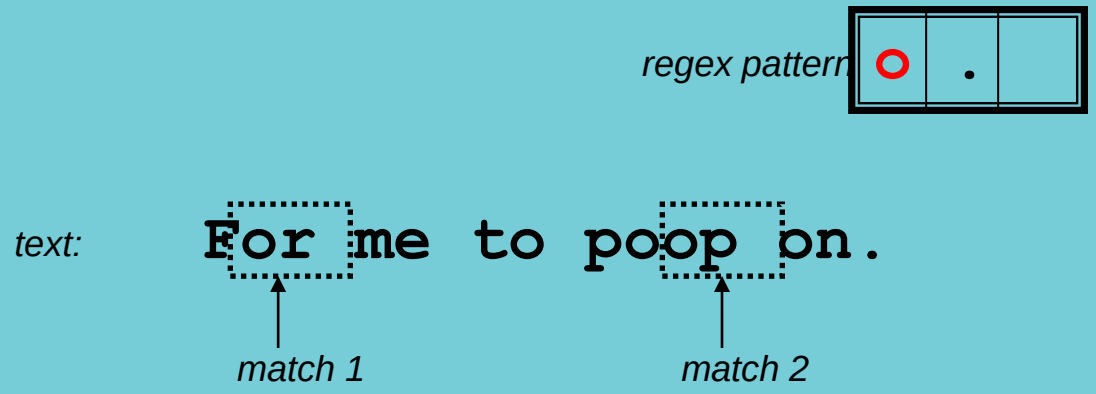
Multiple Matches

A regex pattern can match text in more than one place.



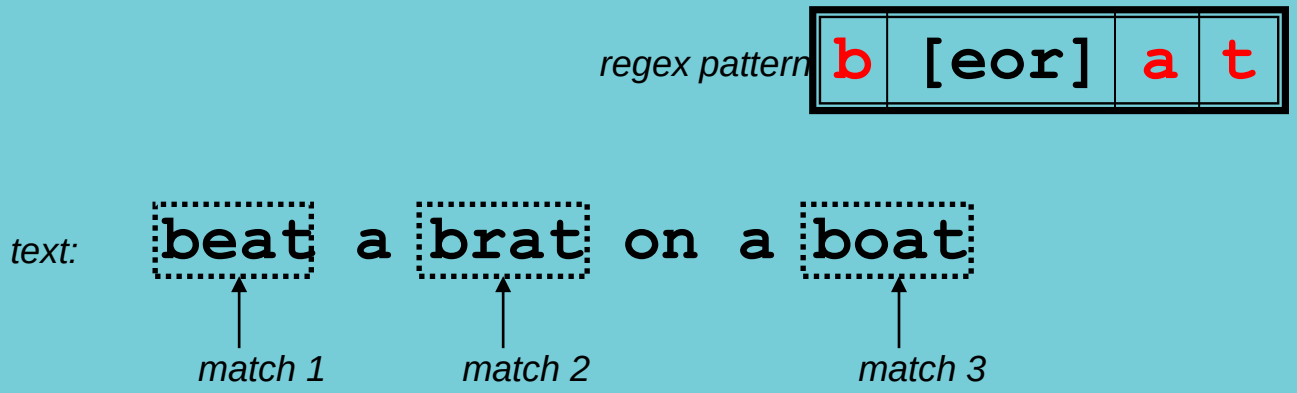
The . (dot) Regex

The `.` regex pattern can be used to match **any character** in the text.

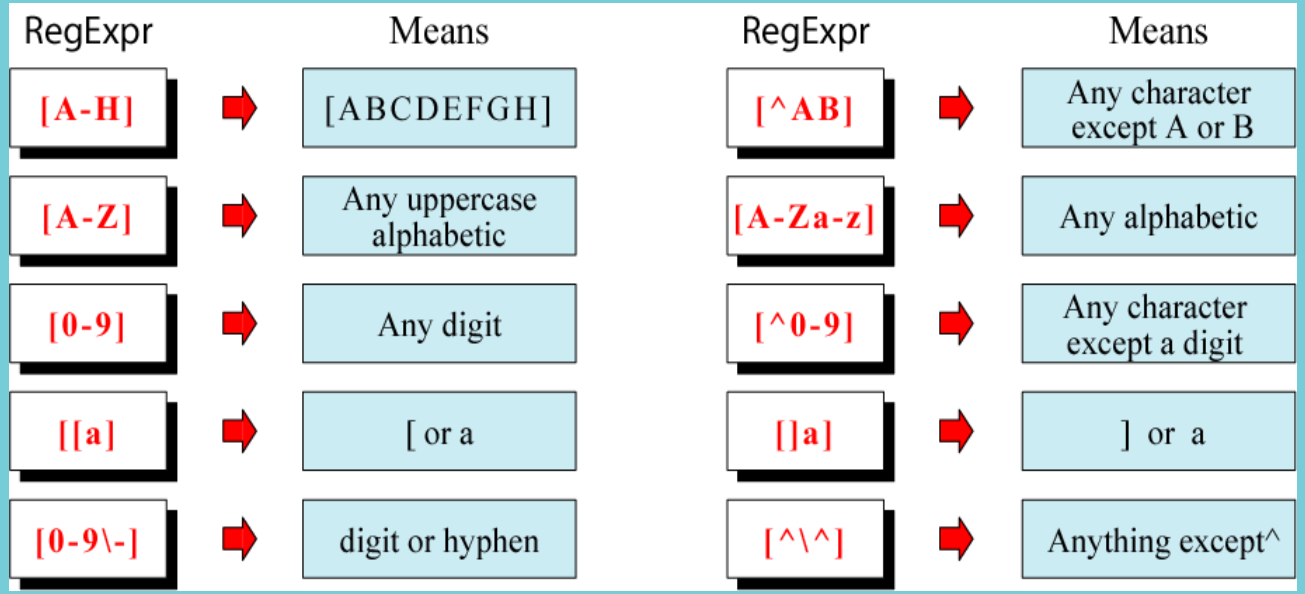


The Character Class Regex

A character class `[]` can match **any set of characters** in the text.

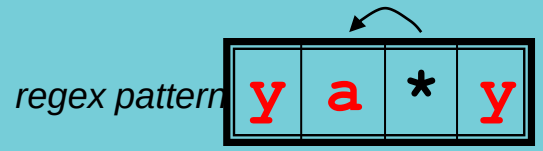


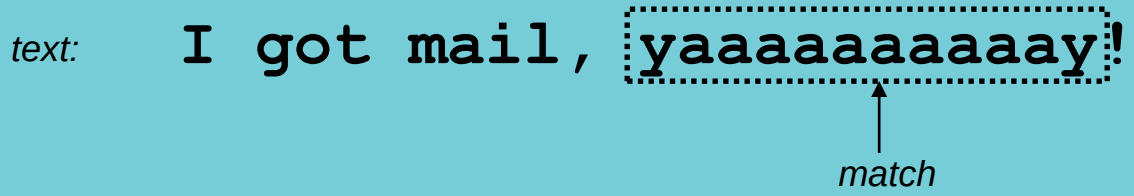
Character Class Examples

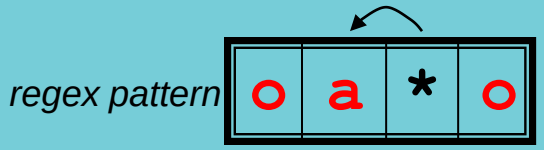


Repetition Regex: * (star)

- The **`*`** defines **zero or more** copies of the letter before it.

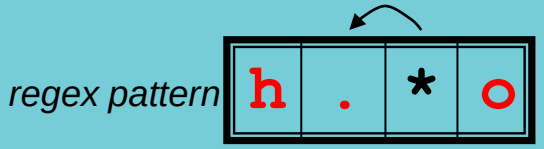






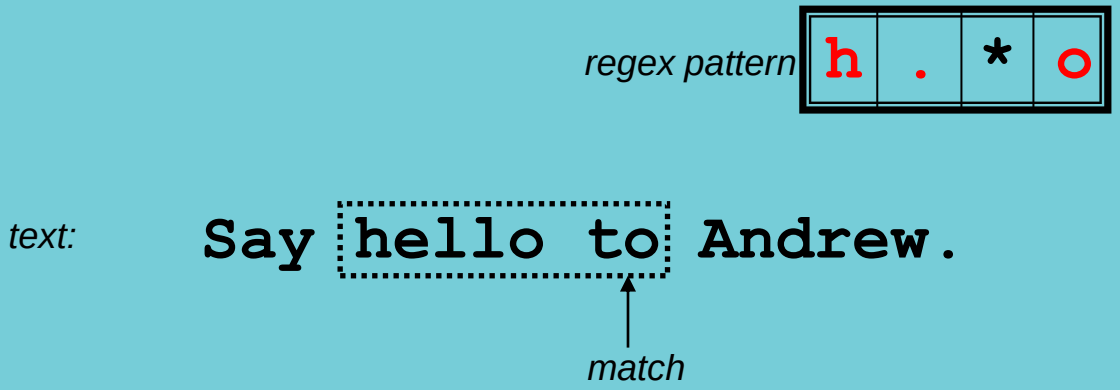
text: I like the zoo.

match



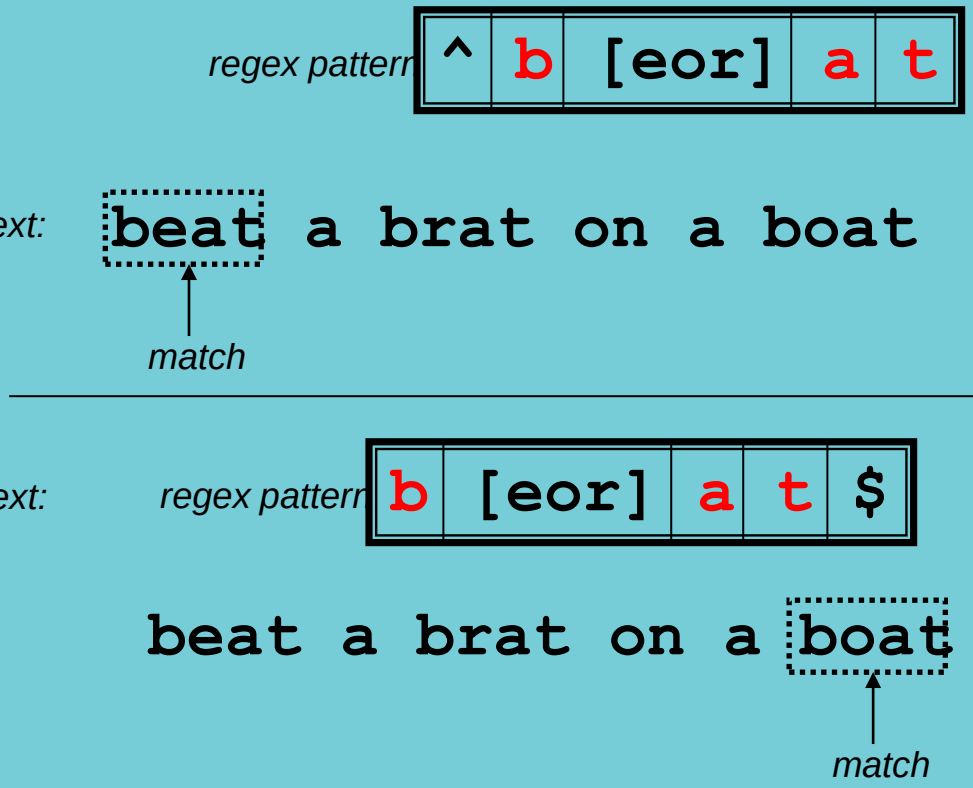
text: Say hello Andrew.

match

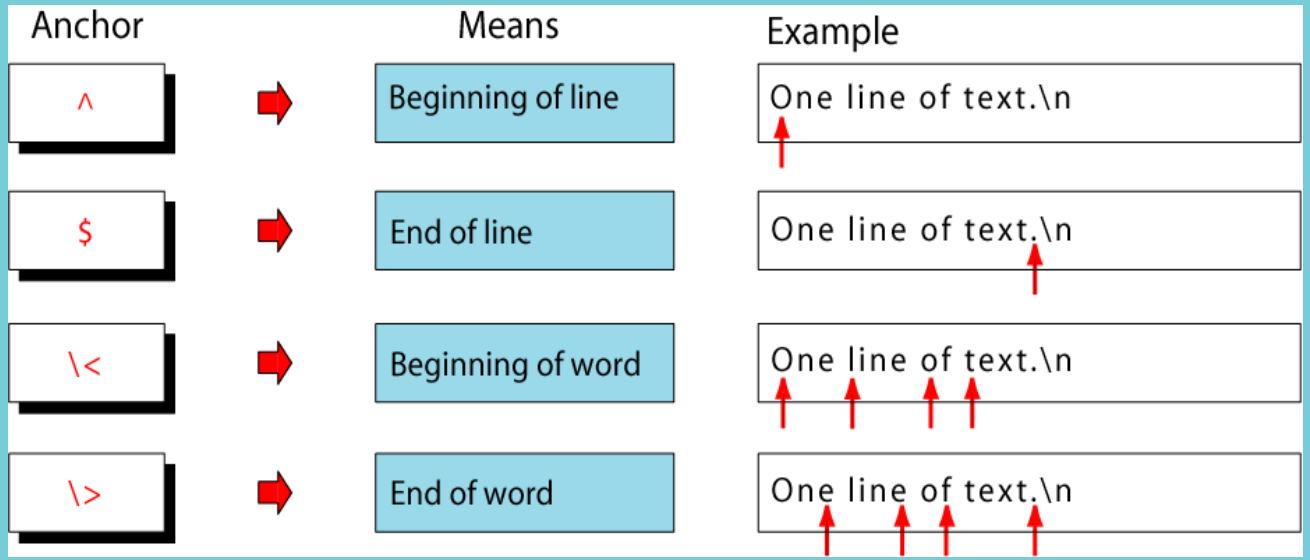


Regex are **greedy** – they match as much of the text as they can.

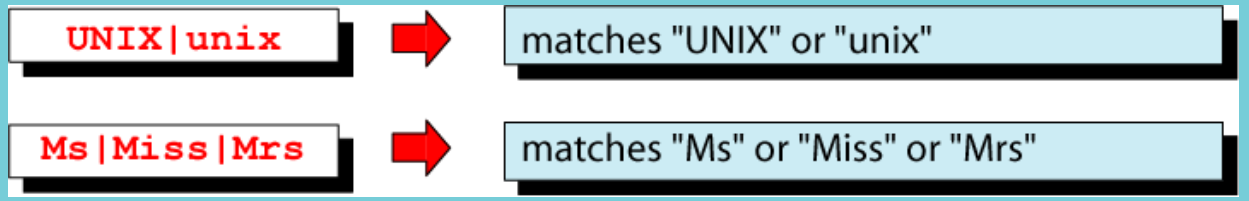
Anchors: ^ $


regex pattern
^ b [eor] a t
text:
beat a brat on a boat
match
^ matches the **beginning** of the text **line**
text:
regex pattern
b [eor] a t $
beat a brat on a boat
match
$ matches the **end** of the text **line**

More Anchors



The | (or) Regex


UNIX|unix
matches "UNIX" or "unix"
Ms|Miss|Mrs
matches "Ms" or "Miss" or "Mrs"

More Repetition Regexs: * + ?

Formats

`*`	➡	special case: matches previous atom zero or more times
`+`	➡	special case: matches previous atom one or more times
`?`	➡	special case: matches previous atom 0 or one time only

Examples

`BA*`	➡	B, BA, BAA, BAAA, BAAAA, . . .
`B.*`	➡	B, BA . . . BZ, BAA . . . BZZ, BAAA . . . BZZZ, . . .
`.*`	➡	zero or more characters
`.+`	➡	one or more characters
`[0-9]?`	➡	zero or one digit

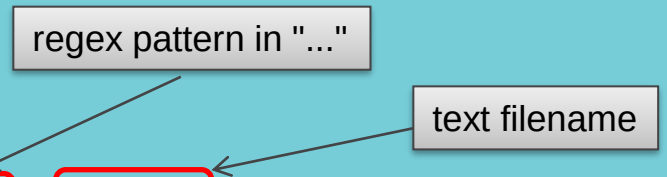
More Regex Operations

See the regular expressions "cheat-sheet"
https://cheatography.com/davechild/cheat-sheets/regular-expressions/
over 80 operators!!

2. grep

“grep” uses a regex pattern to search a text file

all the lines containing a match (or matches) are printed



Examples:

```
% grep "root" test1
% grep "r..t" test1
% grep "ro*t" test1
% grep "r[a-z]*t" test1
```

The Grep Family

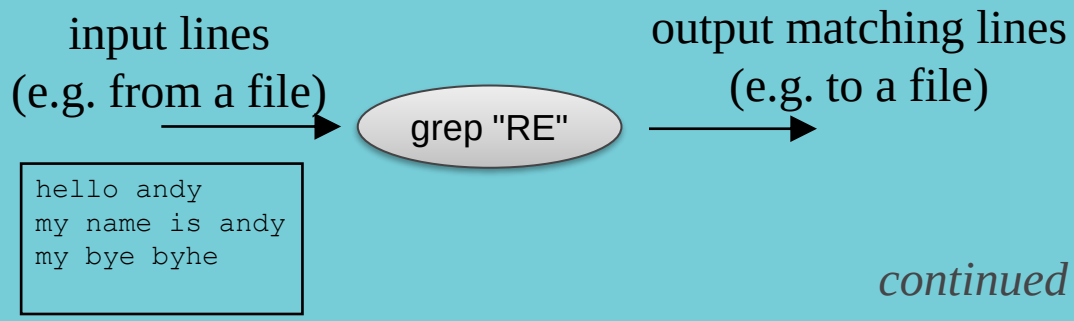
grep	usual version
egrep	**extended** REs | + ? don’t need backslash)
fgrep	only strings, i.e. is **faster**

Common “grep” Options

-c	Print a **count** of matched lines.
-i	**Ignore** uppercase/ lowercase
-l	**List** filenames that contain matches
-n	Print matched lines *and* line **numbers**
-s	Work **silently**; only display error messages.
-v	Print lines that do **not match** the pattern.

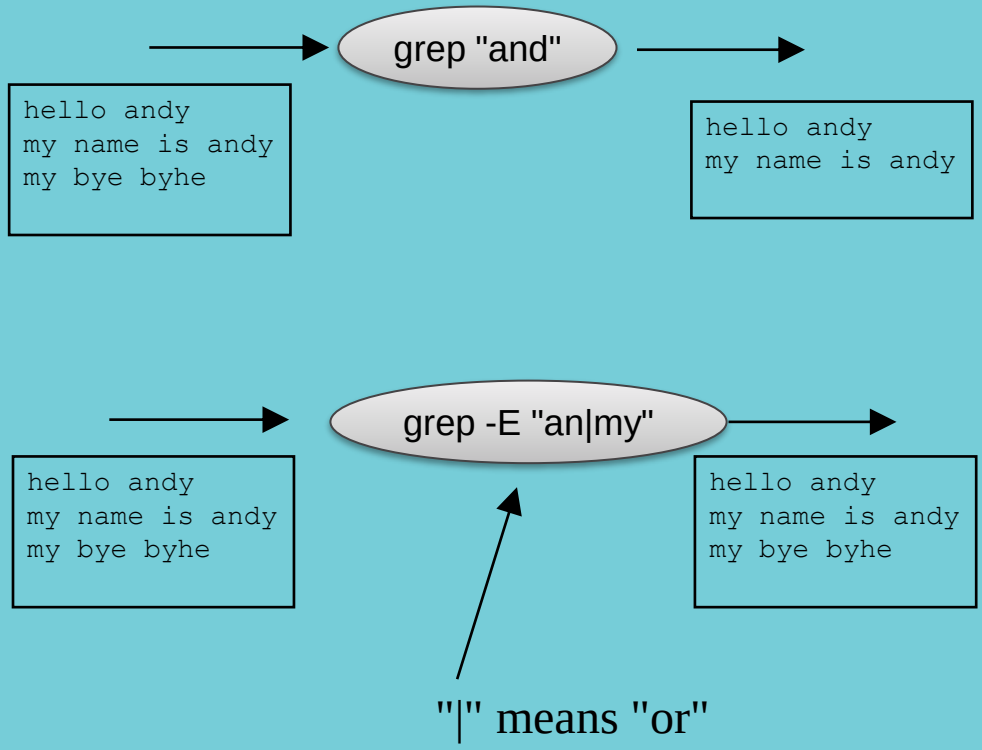
Some Simple Examples

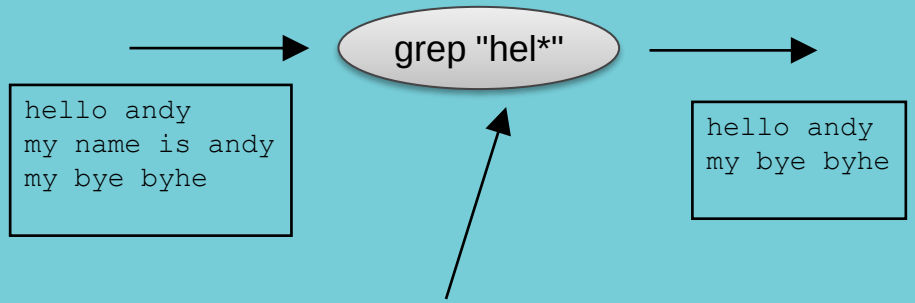
- grep searches input lines, a line at a time.
- If the line contains a string that matches grep's RE (pattern), then the line is output.



continued

Examples


grep "and"
hello andy
my name is andy
my bye byhe
hello andy
my name is andy
grep -E "an|my"
hello andy
my name is andy
my bye byhe
hello andy
my name is andy
my bye byhe
"|" means "or"


grep "hel*"
hello andy
my name is andy
my bye byhe
hello andy
my bye byhe
"*" means "0 or more"

grep with \< \>

begin and end of word

```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ grep "north" grep-datafile
northwest       NW      Charles Main            300000.00
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
ad@fivedots$ grep "\<north\>" grep-datafile
north           NO      Ann Stephens            455000.50
ad@fivedots$
```

Look for the word "north"

grep with a\|b

a or b

```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ grep "NW\|EA" grep-datafile
northwest       NW      Charles Main            300000.00
eastern         EA      TB Savage               440500.45
ad@fivedots$ egrep "NW|EA" grep-datafile
northwest       NW      Charles Main            300000.00
eastern         EA      TB Savage               440500.45
ad@fivedots$
```

egrep doesn't need backslash

grep with \+

one or more

```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ grep "30\+" grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
ad@fivedots$ egrep "30+" grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
ad@fivedots$
```

egrep doesn't need backslash

grep with .

any character

```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ egrep "\.8" grep-datafile
western         WE      Sharon Gray             53000.89
ad@fivedots$ egrep ".8" grep-datafile
western         WE      Sharon Gray             53000.89
northeast       NE      AM Main Jr.             57800.10
ad@fivedots$
```

egrep doesn't need backslash

grep with ^ and $

begin and end of line

```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ grep "^n" grep-datafile
northwest       NW      Charles Main            300000.00
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
ad@fivedots$ grep "00$" grep-datafile
northwest       NW      Charles Main            300000.00
southeast       SE      Patricia Hemenway       400000.00
ad@fivedots$
```

grep with [ ]

set of chars

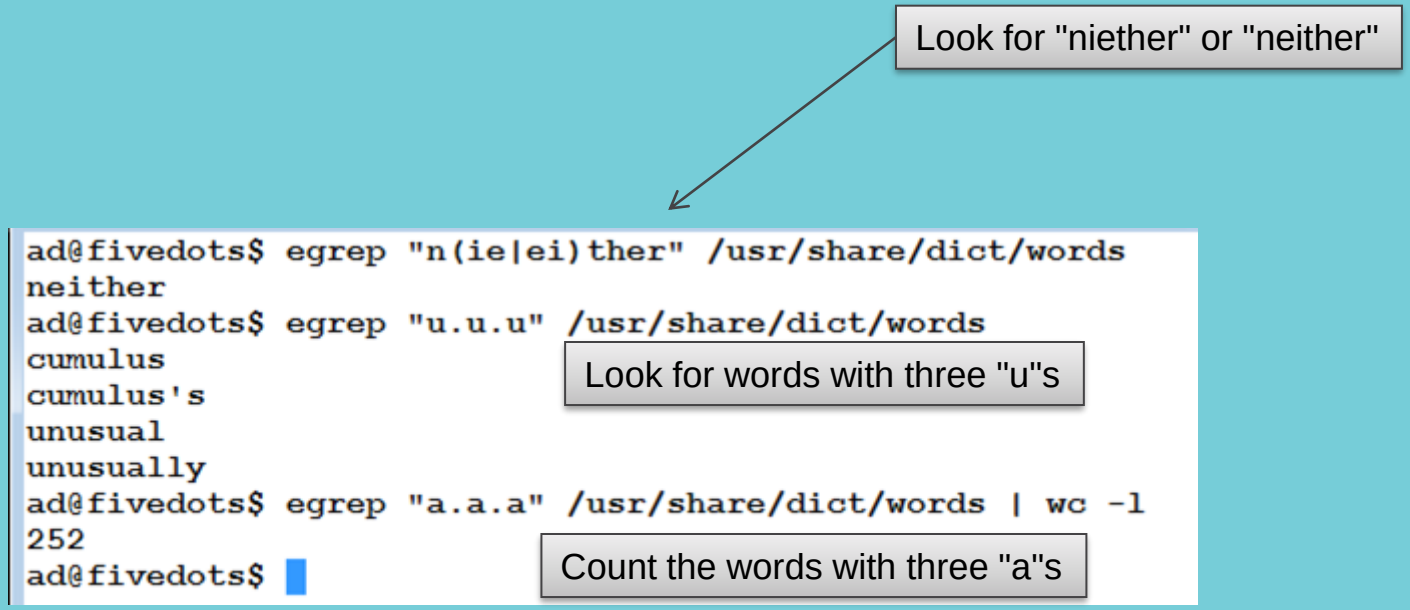
```
ad@fivedots$ cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
ad@fivedots$ grep "^[we]" grep-datafile
western         WE      Sharon Gray             53000.89
eastern         EA      TB Savage               440500.45
ad@fivedots$
```

Fun with a Linux Dictionary

```
ad@fivedots$ pwd
/home/ad
ad@fivedots$ find /usr -name words -print
find: `/usr/src/snoopy-1.3': Permission denied
/usr/share/dict/words
ad@fivedots$ grep "hh" /usr/share/dict/words
Hohhot
Hohhot's
Kirchhoff
Mashhad
Mashhad's
MÃ¼nchhausen
MÃ¼nchhausen's
Wahhabi
bathhouse
bathhouse's
bathhouses
beachhead
beachhead's
beachheads
fishhook
fishhook's
fishhooks
hitchhike
```

Find the location of the words file

List all the words containing "hh"


Look for "niether" or "neither"
ad@fivedots$ egrep "n(ie|ei)ther" /usr/share/dict/words
neither
ad@fivedots$ egrep "u.u.u" /usr/share/dict/words
cumulus
cumulus's
unusual
unusually
ad@fivedots$ egrep "a.a.a" /usr/share/dict/words | wc -l
252
ad@fivedots$
Look for words with three "u"s
Count the words with three "a"s

Complex Regex Examples

Variable names in C

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Dollar amount with optional cents

```
\$[0-9]+(\.[0-9][0-9])?
```

Time of day

```
(1[012]|[1-9]):[0-5][0-9] (am|pm)
```

HTML headers <h1> <H1> <h2> …

```
<[hH][1-4]>
```

3. The RE Language

•A RE can be defined as a pattern language (operands and operators) which matches on text strings.

Some Possible RE Operands

text characters (e.g. ‘a’, ‘1’, ‘(‘)
the symbol ε (means an empty string ‘’)
in code just use ""

variables, which can be assigned a RE
•variable = RE

The Basic RE Operators

- There are three basic operators:

union ‘|’
concatenation
closure *

Union

- S | T

use S *or* T to match strings

- Example REs:

  a | b

  a | b | c

Concatenation

- S T

use S *followed by* the T to match against strings

- Example REs:

a b	matches the string "ab"
w | (a b)	matches the strings "w" or "ab"

Closure

- S*

use S *0 or more times* to match against strings

- Example RE:

a* matches the strings:

, a, aa, aaa, aaaa, aaaaa, ...

empty string

3.1. REs for C Identifiers

- We define two RE variables, `letter` and `digit`:

```
letter = A | B | C | D ... Z |
         a | b | c | d .... z

digit = 0 | 1 | 2 | 3 | 4 | 5 |
  6 | 7 | 8 | 9
```

- `id` is defined using `letter` and `digit`:

```
id = letter ( letter | digit )*
```

continued

- Strings matched by `id` include:

  ab345 w h5g

- Strings not matched:

  2 $abc ****

3.2. REs for Integers and Floats

- We redefine `digit`:

```
digit = 0|1|2|3|4|5|6|7|8|9
```

or `digit = [0 - 9]`

- `int` and `float`:

```
int = {digit}+
float = {digit}+ "." {digit}+
```

- Integers and floats with exponents:

```
number = {digit}+('.' {digit}+ )?
  ( 'E'('+'|'-')? {digit}+ )?
```

4. More on REs

- See RE summary:
  `regular_expressions_cheat_sheet.pdf`

- I have the standard RE book:
  Mastering Regular Expressions
  Jeffrey E. F. Freidl
  O'Reilly & Associates

continued

There are many websites that explain REs:

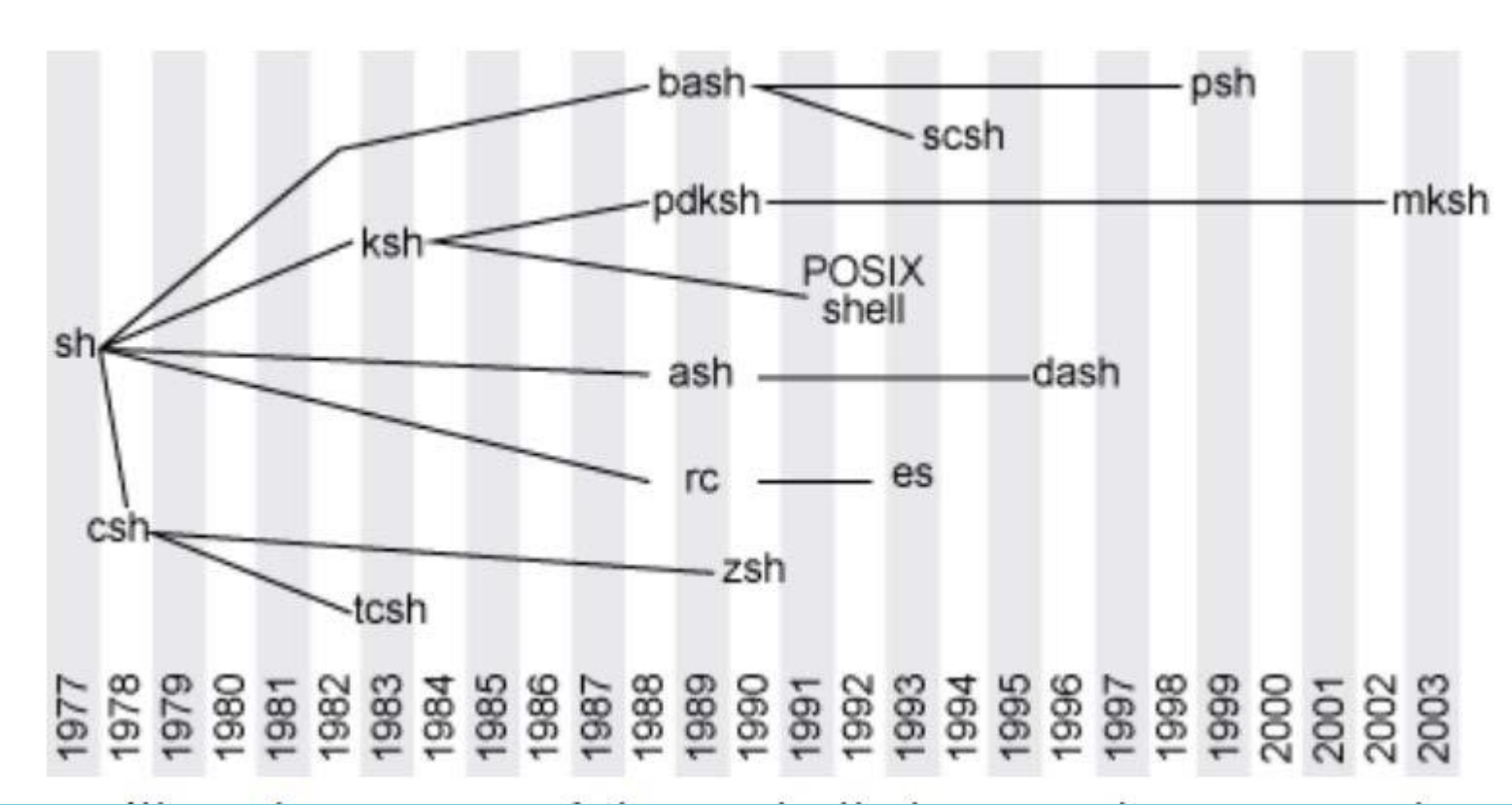
```
http://etext.lib.virginia.edu/services/
helpsheets/unix/regex.html

http://www.zytrax.com/tech/web/regex.htm

http://www.regular-expressions.info
```

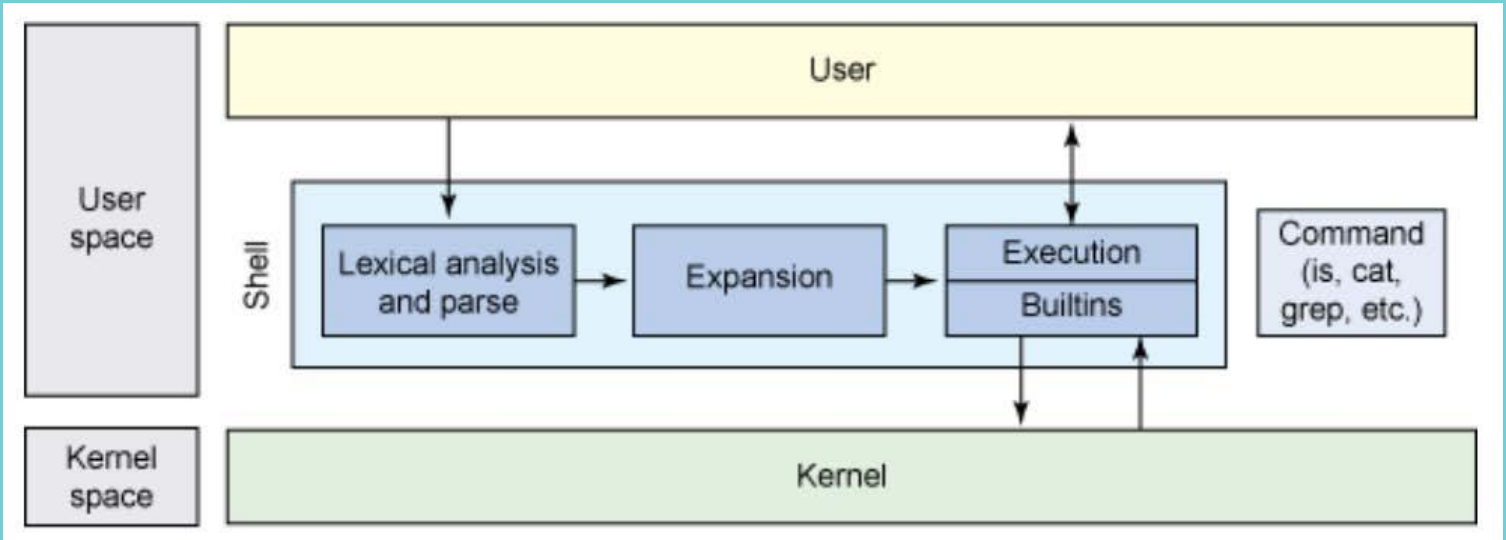
SHELL

Shell Evolution



Basic shell architecture

The fundamental architecture of a hypothetical shell is simple (as evidenced by Bourne's shell). As you can see below, the basic architecture looks similar to a pipeline, where input is analyzed and parsed, symbols are expanded (using a variety of methods such as brace, tilde, variable and parameter expansion and substitution, and file name generation), and finally commands are executed (using shell built-in commands, or external commands)



Bash. Scripting. Summary (1)

When **not** to use shell scripts:
- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion, etc.)
- Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- Cross-platform portability required (use C or Java instead)
- Complex applications, where structured programming is a necessity (type-checking of variables, function prototypes, etc.)
- Mission-critical applications upon which you are betting the future of the company
- Situations where security is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access)

Bash. Scripting. Summary (2)

When **not** to use shell scripts:
- Need native support for multi-dimensional arrays
- Need data structures, such as linked lists or trees
- Need to generate / manipulate graphics or GUIs
- Need direct access to system hardware or external peripherals
- Need port or socket I/O
- Need to use libraries or interface with legacy code
- Proprietary, closed-source applications (Shell scripts put the source code right out in the open for all the world to see.)

P.S. If any of the above applies, consider a more powerful scripting language -- perhaps Perl, Python, Ruby - or possibly a compiled language such as C, C++, or Java.
Even then, prototyping the application as a shell script might still be a useful development step.

Bash. Scripting. Summary (2)

Shells like **bash** have support for programming constructs that can be saved as scripts.

These scripts in turn then become more shell commands. Many Linux commands are scripts.

User profile scripts are run when a user logs on and init scripts are run when a daemon is stopped or started.

This means that system administrators also need basic knowledge of scripting to understand how their servers and their applications are started, updated, upgraded, patched, maintained, configured and removed, and also to understand how a user environment is built.

The goal of this module is to give enough information to be able to read and understand scripts. And to become a writer of simple scripts.

Bash. Scripting. Hello World

Just like in every programming course, we start with a simple hello_world script. The following script will output Hello World.

> echo Hello World

After creating this simple script in vi or with echo, you'll have to chmod +x hello_world to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

[student@localhost ~]$ echo echo Hello World > hello_world
[student@localhost ~]$ chmod +x hello_world
[student@localhost ~]$ ./hello_world
Hello World
[student@localhost ~]$

Bash. Scripting. She-bang

Let's expand our example a little further by putting #!/bin/bash on the first line of the script. The **#!** is called a she-bang (sometimes called sha-bang), where the she-bang is the first two characters of the script.

```
#!/bin/bash
echo Hello World
```

You can never be sure which shell a user is running. A script that works flawlessly in bash might not work in ksh, csh, or dash. To instruct a shell to run your script in a certain shell, you can start your script with a she-bang followed by the shell it is supposed to run in. This script will run in a bash shell.

```
#!/bin/bash
echo -n hello
echo A bash subshell `echo -n hello
```

Bash. Scripting. Comments. Variables

Let's expand our example a little further by adding comment lines.

```
#!/bin/bash
#
# Hello World Script
#
echo Hello World
```

Here is a simple example of a variable inside a script.

```
#!/bin/bash
#
# simple variable in script
#
var1=3
echo var1 = $var1
```

Bash. Scripting.Variables. Sourcing a script

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

[student@localhost ~]$ ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
[student@localhost ~]$

But we can force a script to run in the same shell, this is called ***sourcing*** a script (2 ways).

```
[student@localhost ~]$ source ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

```
[student@localhost ~]$ . ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

Bash. Scripting. Conditions and loops. test[]

The ***test*** command can test whether something is true or false. Let's start by testing whether 10 is greater than 55.

$ test 10 -gt 55 ; echo $?

1

The test command returns ***1*** if the ***test*** fails. And as you see in the next screenshot, ***test*** returns 0 when a test succeeds.

$ test 56 -gt 55 ; echo $?

0

If you prefer true and false, then write the test like this.

$ test 56 -gt 55 && echo true || echo false

true

$ test 6 -gt 55 && echo true || echo false

false

The test command can also be written as square brackets, the screenshot below is identical to the one above.

$ [56 -gt 55] && echo true || echo false

true

$ [6 -gt 55] && echo true || echo false

false

Bash. Scripting. Conditions and loops. test[]

Below are some example tests. Take a look at ***man test*** to see more options for tests.

[-d foo] Does the directory foo exist ?
[-e bar] Does the file bar exist ?
['/etc' = $PWD] Is the string /etc equal to the variable $PWD ?
[$1 != 'secret'] Is the first parameter different from secret ?
[55 -lt $bar] Is 55 less than the value of $bar ?
[$foo -ge 1000] Is the value of $foo greater or equal to 1000 ?
["abc" < $bar] Does abc sort before the value of $bar ?
[-f foo] Is foo a regular file ?
[-r bar] Is bar a readable file ?
[foo -nt bar] Is file foo newer than file bar ?
[-o nounset] Is the shell option nounset set ?

Tests can be combined with logical AND and OR.

```
$ [ 66 -gt 55 -a 66 -lt 500 ] && echo true || echo false
true
$ [ 66 -gt 55 -a 660 -lt 500 ] && echo true || echo false
false
$ [ 66 -gt 55 -o 660 -lt 500 ] && echo true || echo false
true
```

Bash. Scripting. Conditions *If then else*

The ***if then else*** construction is about choice. If a certain condition is met, then execute something, else execute something else. The example below tests whether a file exists, and if the file exists then a proper message is echoed.

```
#!/bin/bash
if [ -f isit.txt ]
        then echo isit.txt exists!
else echo isit.txt not found!
fi
```

If we name the above script 'choice', then it executes like this.

```
$ ./choice
isit.txt not found!
$ touch isit.txt
$ ./choice isit.txt exists!
$
```

Bash. Scripting. Conditions *If then elif*

You can nest a new ***if*** inside an ***else*** with ***elif***. This is a simple example.

```
#!/bin/bash
count=42
if [ $count -eq 42 ]
then
echo "42 is correct."
elif [ $count -gt 42 ]
then
echo "Too much."
else
echo "Not enough."
fi
```

Bash. Scripting. Loops. *for loop*

The example below shows the syntax of a classical ***for loop*** in bash:

```
for i in 1 2 4
do
echo $i
done
```

An example of a for loop combined with an embedded shell:

```
#!/bin/bash
for counter in `seq 1 20`
do
echo counting from 1 to 20, now at $counter
sleep 1
done
```

Bash. Scripting. Loops. *for loop*

The same example as above can be written without the embedded shell using the bash {from..to} shorthand.

```
#!/bin/bash
for counter in {1..20}
do
echo counting from 1 to 20, now at $counter
sleep 1
done
```

This for loop uses file globbing (from the shell expansion). Putting the instruction on the command line has identical functionality.

```
$ ls
count.ksh go.ksh
$ for file in *.ksh ; do cp $file $file.backup ; done
$ ls
count.ksh count.ksh.backup go.ksh go.ksh.backup
```

Bash. Scripting. Loops. *while loop*

Below a simple example of a ***while loop***

```
i=100;
while [ $i -ge 0 ] ;
do
        echo Counting down, from 100 to 0, now at $i;
        let i--;
done
```

Endless loops can be made with
while true or ***while :*** ,
where the ***colon*** is the equivalent of
no operation in the ***bash*** shell.

```
#!/bin/bash
# endless loop while :
do
        echo hello
        sleep 1
done
```

Below a simple example of an ***until loop***

```
let i=100;
until [ $i -le 0 ] ;
do
        echo Counting down, from 100 to 1, now at $i;
        let i--;
done
```

Bash. Scripting. Loops. *for loop*

Write a script that counts the number of files ending in ***.txt*** in the current directory

```
#!/bin/bash
let i=0
for file in *.txt
do
        let i++
done
echo "There are $i files ending in .txt"
```

Wrap an ***if*** statement around the script so it is also correct when there are zero files ending in ***.txt***

```
#!/bin/bash
ls *.txt > /dev/null 2>&1
if [ $? -ne 0 ]
then echo “Directory contains 0 *.txt files"
else
    let i=0
    for file in *.txt
    do
        let i++
    done
    echo " Directory contains $i *.txt files "
fi
```

QUESTIONS & ANSWERS


THANK YOU!