Mastering LINUX/UNIX Regular Expressions

Presenter: Glenn Martin Stafford
stafford_g@bellsouth.net

As strange as regular expressions sounds, it is nothing more than a pattern matching tool. This extremely powerful tool allows anyone to formulate powerful queries!

By mastering regular expressions (regex), one can perform complex search and replace operations, validate input, and parse text with ease. This include cleaning data, extracting information, or building sophisticated search features, regex will undoubtedly become an invaluable part of your programming toolkit.

This presentation will demonstrate the powerful use of regular expressions with the commands vi, sed, grep, and awk.

Anyone, from novices to highly skilled in the Unix/Linux professional can benefit from this presentation.

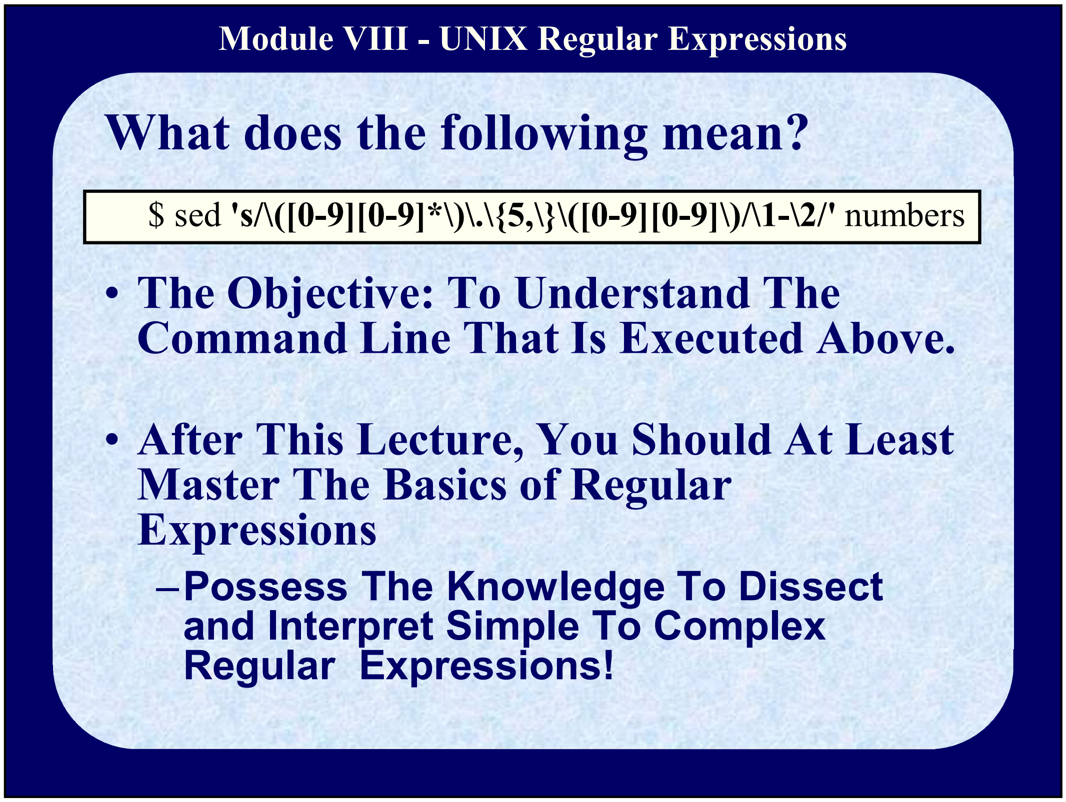
<u>Speaker Bio</u>

Glenn Stafford is a veteran Unix/Linux professional best known for automating server installations, software code conversions, version control environments, shell programming, and automating system administration tasks.

His notable accomplishments include:

- Automating installations where servers were ready for full accreditation within 30 minutes without the use of Commercial Off The Shelf configuration tools.
- Automated the migration of VmWare virtual servers to he AWS Government Club services.

When Glenn lived in the Chicagoland area, he was a dedicated member of Uniforum presenting on various topics and facilitating the UNIX Bootcamp and UNIX Top Gun workshops. Glenn is also known for his success as a top-gun instructor for Information Technology Development Systems and Sun Microsystems.

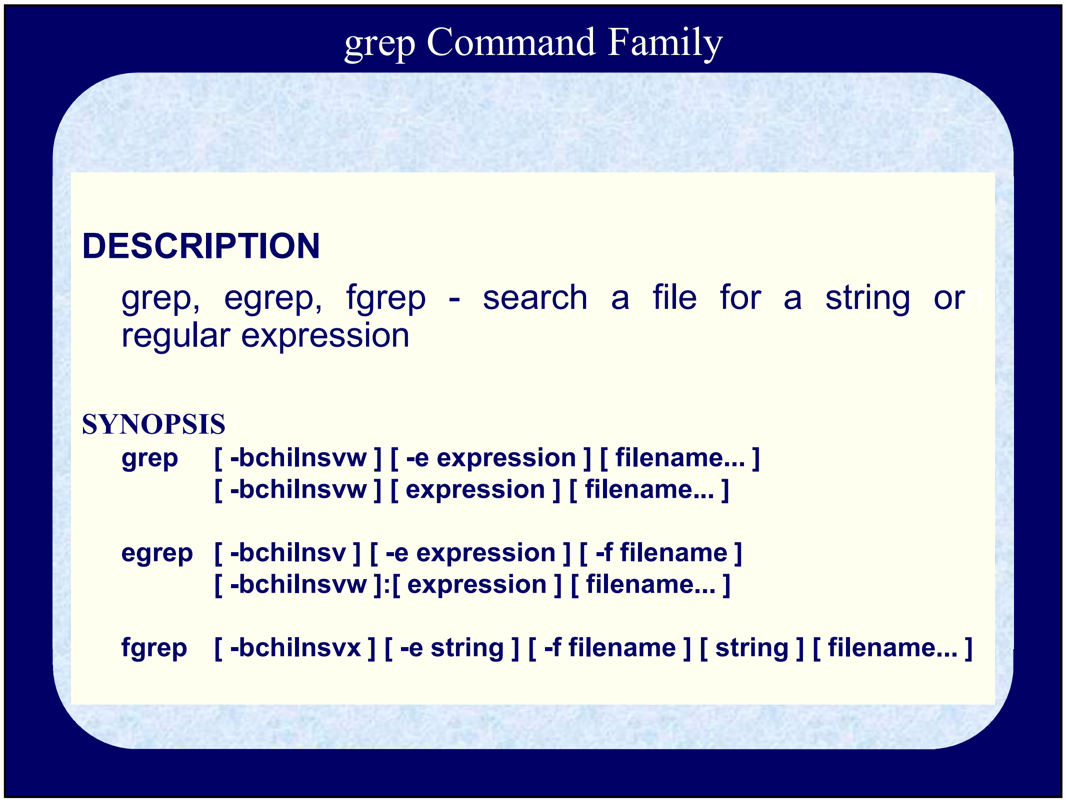


The objective here is simple: Understand the command line in the above example. Upon the completion of this lecture, you should have the basic knowledge that will you to dissect UNIX regular expressions and interpret what's happening! With the examples provide, you should realize the importance of UNIX regular expressions and include this as part of your skill sets.

Regular Expressions Defined:

UNIX Regular Expressions is nothing more than describing a text pattern (a sequence of characters). In other words, regular expressions define a sequence of characters to match. In addition to matching these patterns, there are tools that allow us to perform powerful string substitutions.

If you fear regular expressions, you are normal! However, regular expressions are equivalent to idiomatic expressions.



The purpose of this slide is to educate the users that there is a family of grep commands. These commands are used to perform very powerful pattern matches. However, there is confusion to what these commands are capable of.

The fgrep command is fixed string pattern matching command. It does not support regular expressions.

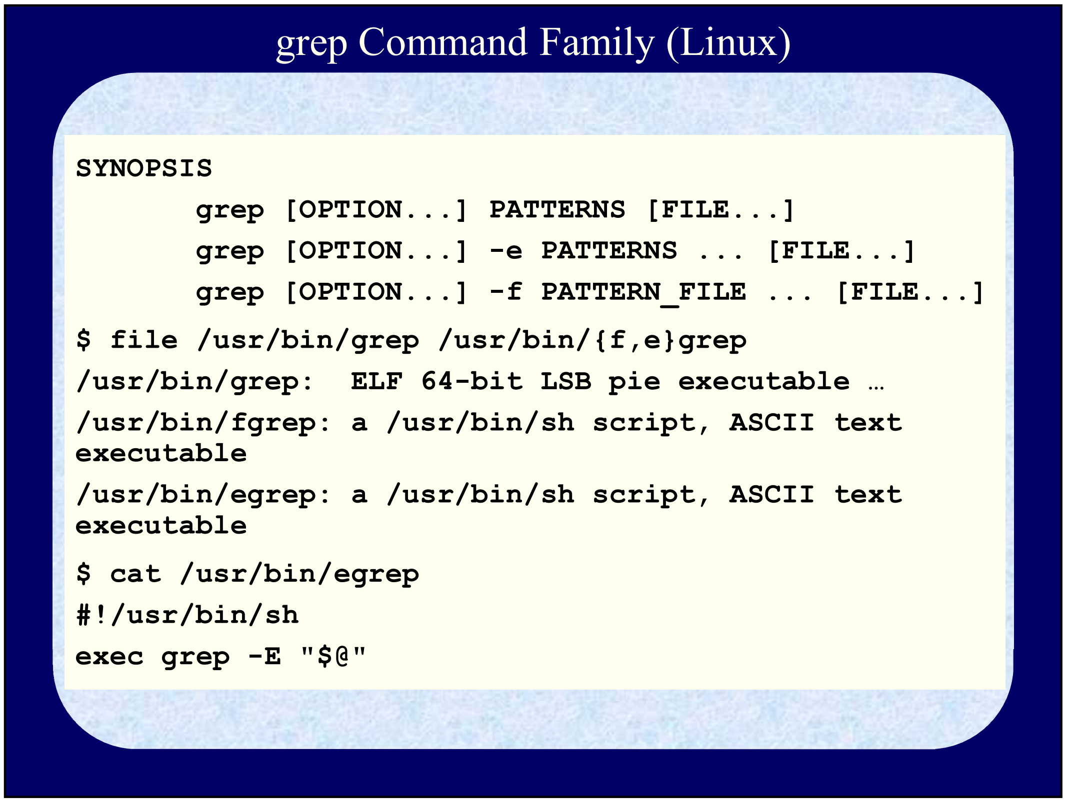
The grep command can accept the common set of regular expressions. Common character sets used include:

^	Match The Expression At The Beginning Of The Line.
$	Match The Expression At The End Of Line.
\	Escape The Special Meaning of The Next Character.
-	Indicates A Range When Not In The First/Last Position When Specifying Ranges With The "[" and "]" To Specify A List.
[LIST]	Match A Single Character Specified In The List

The egrep command is the extended grep command. It supports the extended set set of regular expressions such as:

|	Union of regular expressions
()	Group expressions

In addition, extended regular expressions support closure notations that that stipulates the frequency to repeat specified regular expression (examples provide later).



In original implementations of the Unix system, the **fgrep**, **egrep**, and **grep** commands were 3 separate files. However, there were some subtle variations. For example, for HP's UNIX version, **HPUX**, these three commands had the same **inode** number which means they are thre same file.

In this slide, you will the grep command synopsis highlights two –**e** and –**f** options:

> **-e** interpret PATTERNS as extended regular expressions
>
> **-f** Interpret PATTERNS as fixed strings, not regular expressions.

In addition, notice the file command reports, the **fgrep** and **egrep** commands are shell scripts, whereas **grep** is a compiled program.

Notice with the **cat** command displays the contents of the **egrep** script. Notice the execution of the **grep** command using the –**E** option to interpret PATTERNS as extended regular expressions.

fgrep/grep Examples

```
$ fgrep fish fortunes
A woman without a man is like a fish without a bicycle.
No one can feel as helpless as the owner of a sick goldfish.
Time is about the stream I go a-fishing in.
$ fgrep '^[AEIOU]' fortunes.

$ grep '^[AEIOU]' fortunes
 A modem is a baudy house.
 A truly wise man never plays leapfrog with a Unicorn.
 A woman without a man is like a fish without a bicycle.
 An elephant is a mouse with an operating system.
 Any time things appear to going better, you have overlooked something.
 Armadillo: Supplying weapons to a Spanish pickle.
 E Pluribus UNIX
 Every program is a part of some other program, and rarely fits.
 If at first you don't succeed, give up, don't be a darn fool.
 Old programmers never die. They just branch to a new address.
 On a clear disk you can seek forever.
 I must have slipped a disk; my pack hurts.
```

In the first example (**fgrep fish**), we're performing a pattern match on the exact character string sequence "fish" which can appear anywhere on the line of text.

In the second example, we're attempting match a capitalizes vowel at the beginning of the line. Unfortunately, fgrep doesn't support regular expressions. This pattern match specified with the fgrep command is trying to match the character string '^[AEIOU]' exactly.

Notice in the third example, we can satisfy attempting to match an line that begins with a vowel at the beginning of a line that is capitalized. The expression, "^[AEIOU]" says match a capital vowel at the beginning of a line. We're not interested with what follows in the pattern match.

Please note the circumflex character (^) has two purposes. Outside square braces it must be list first to match at the beginning of a line. Inside square braces - don't match the single character values listed in the square braces.

egrep example

```
$ grep ' paper | people ' fortunes

$ egrep ' paper | people ' fortunes
Line printer paper is strongest at preforations.
Man Year: 730 people working feverishly until noon.
Nudists are people who wear one-button suits.

$ egrep '^stu(01|02)' /etc/passwd
stu01a:x:7001:7001:Student 1a:/home/stu01a:/bin/bash
stu02a:x:7002:7002:Student 2a:/home/stu02a:/bin/bash
stu01b:x:8001:8001:Student 1b:/home/stu01b:/bin/ksh
stu02b:x:8002:8002:Student 2b:/home/stu02b:/bin/ksh
stu01c:x:9001:9001:Student 1c:/home/stu01c:/bin/csh
stu02c:x:9002:9002:Student 2c:/home/stu02c:/bin/csh
```

This slide illustrates how grep doesn't support extended regular expressions.

In the first example, we're attempting to match the strings ***paper*** or ***people*** anywhere on the line. However, don't expect the grep command to use the **logical or(ing)** of pattern matching.

However, in the second we are successful in using the pipe character for the logical or condition to match ***paper*** or ***people***.

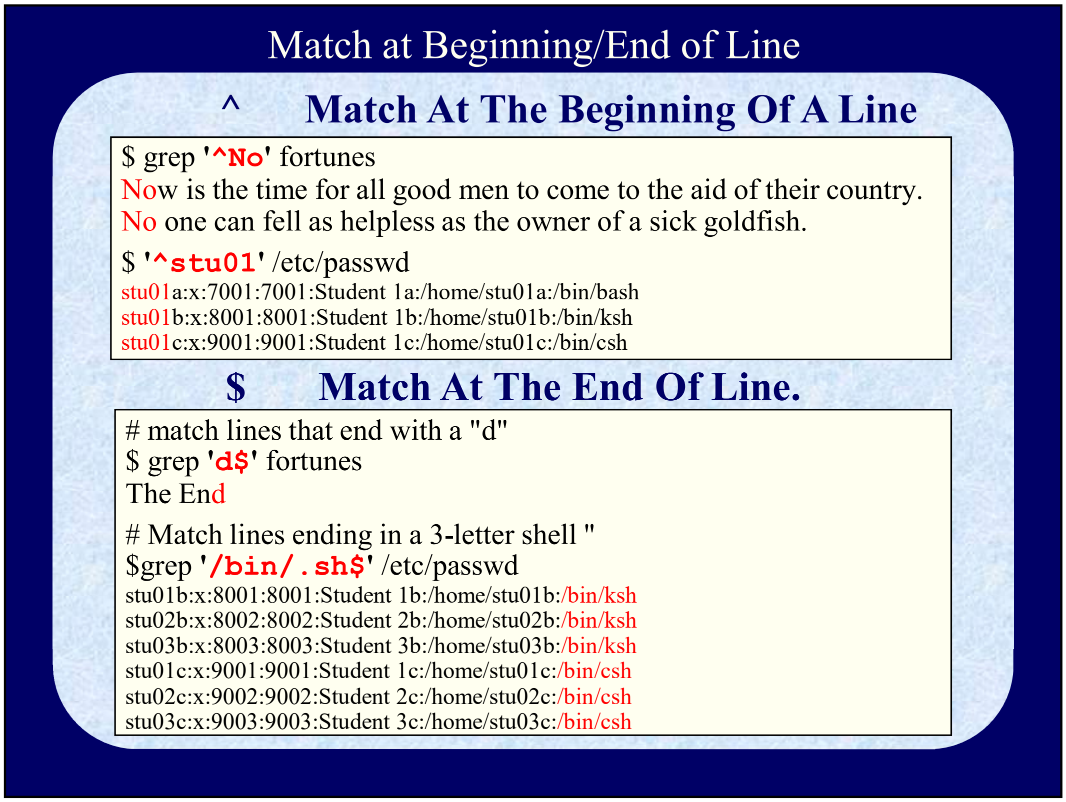
In the third example we are successful in matching any character string that:

> starts with the string **stu** at the beginning of the line.
> The next two characters must be **01**, **02**, or **03**.

Thus, the about illustrates text records that can satisfy the match:

> stu01a
> stu02a
> stu01b
> stu02b
> stu01c
> stu02c

Can you think of other possibilities that would satisfy the match?



In this slide we illustrate matching at the beginning of the line and at the end of a line.

In the first set of examples, we illustrate matching at the beginning of a line. The expression, **'^No'**, states we want to matching at the beginning of a line a capital ***N*** followed by a lowercase ***o***. This is why *Now* and *No* for the phrases match this pattern. In the second part, the expression, '**stu0**', is to match the character string ***stu01*** at the beginning of each record in the password file.

In the second set of examples, the expression, '**d$**', states we want to match a line that has the character ***d*** at the very end of the line. The record ***The End*** is a match. In the pattern match example, the expression '**/bin/.sh$**' states to match ***/bin/,*** followd by any single character, followed by sh. This string is to be matched at the end of the line. We are simply matching shells that are 3 characters long ending is **sh**.

[LIST] Match A Single Character In A Specified List.

 # Match Disk or disk

===> grep '**[Dd]isk**' fortunes
Disk crisis, please clean up!
On a clear disk you can seek forever.
I must have slipped a disk; my pack hurts.

Match csh or ksh

===> grep '**[ck]sh$**' /etc/passwd
stu01b:x:8001:8001:Student 1b:/home/stu01b:/bin/ksh
stu02b:x:8002:8002:Student 2b:/home/stu02b:/bin/ksh
stu03b:x:8003:8003:Student 3b:/home/stu03b:/bin/ksh
stu01c:x:9001:9001:Student 1c:/home/stu01c:/bin/csh
stu02c:x:9002:9002:Student 2c:/home/stu02c:/bin/csh
stu03c:x:9003:9003:Student 3c:/home/stu03c:/bin/csh

This slide illustrates features for using a single character match inside square braces.

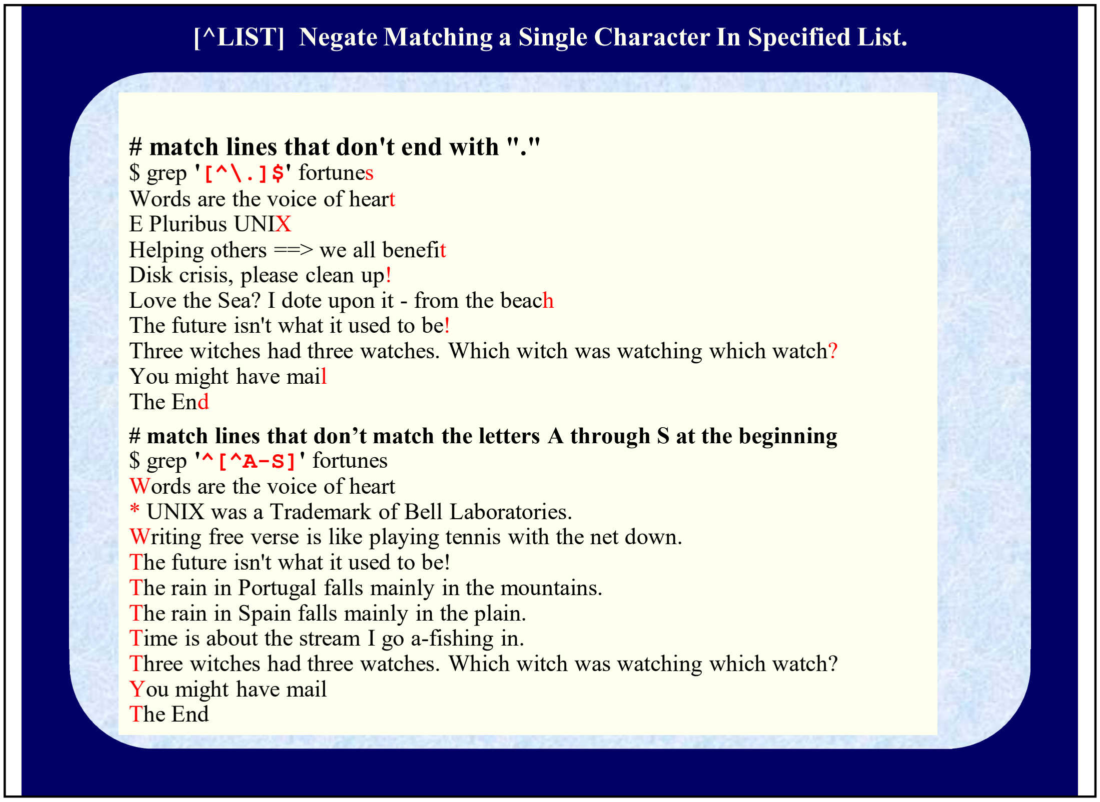
In example one, we want to match an uppercase or lowercase ***d***; followed by the string ***isk***. We're basically looking for the word *Disk* or *disk* that can appear anywhere in the text record. Notice no space was specified.

In the second example we're looking for pattern matches in the /etc/passwd file, that begins with a ***c*** or ***k***; followed by ***sh*** that appears at the end of line. We're looking for ***csh*** or ***ksh*** matches at the end of line.

Other examples of ranges include:

[A-Za-z]	Match the range A thru Z and a thru z
[0-9]	Match the single digit 0-9
[a-mt-z]	Match a thru m and t thru z

Matching single characters is a very powerful expression matching patterns.



This slide illustrates two things. First it shows how to use the negation of a single character rangeinside square braces. Then it illustrates using the circumflex (^) character to match at the beginning; then not to match a single character in a range.

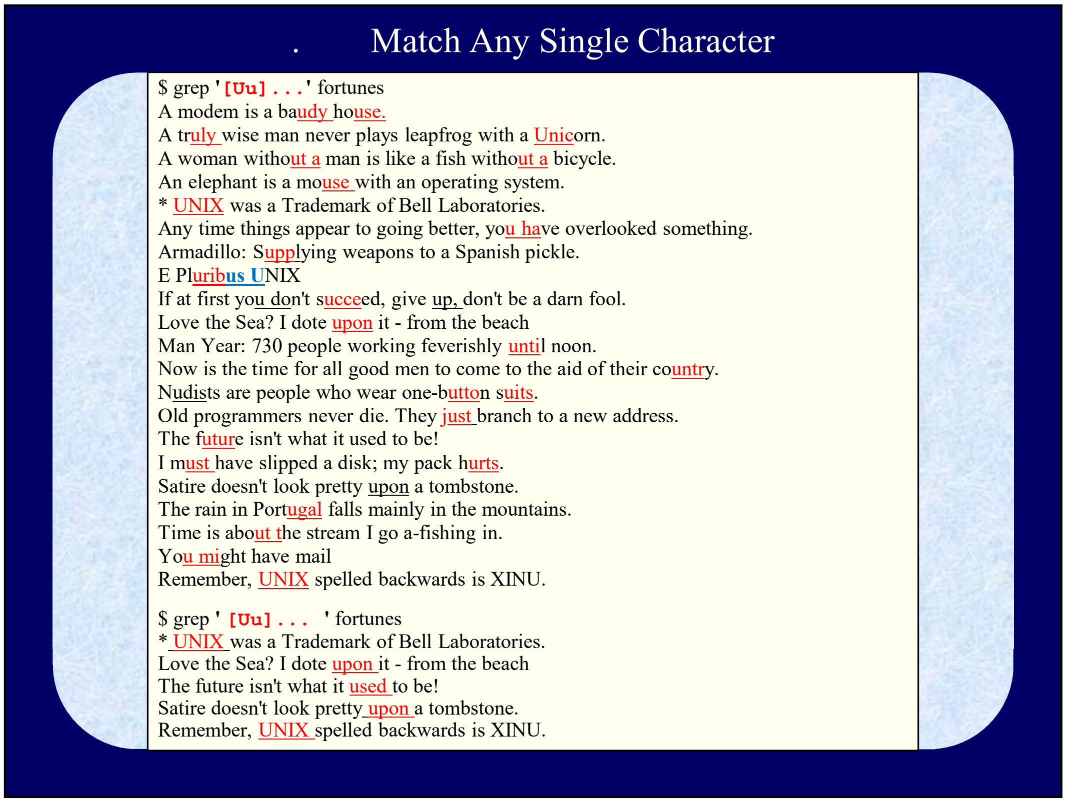
In the first example we want to not match any line which does not end with the punctuation period (**.**). To be safe I use the backslash to escape the special meaning of the period (match a printable charachter. Some would argue that this isn't necessary. The issue here is to test your results. The circumflex character states not to match any period; followed by **$** to specifically state no period at the end of the line (or record).

In the second example the expression, '**^[^A-S]**', states to match from the beginning of a line any character not in the uppercase character range **A thru S**. Thus, the characters that can satisfy this match are lowercase characters, uppercase characters in the range of T thru Z, numbers, and other characters not in the range of uppercase A thru S.

Consider the example below:

```
$ grep '^stu0[^2-6]a' /etc/passwd
stu01a:*:501:501:Student Account:/u/students/stu01a:/bin/ksh
stu07a:*:507:507:Student Account:/u/students/stu07a:/bin/ksh
stu08a:*:508:508:Student Account:/u/students/stu08a:/bin/ksh
stu09a:*:509:509:Student Account:/u/students/stu09a:/bin/ksh
```

This expression states to match all the string ***stu0*** at the beginning of the line, then the fifth character can not be a digit in the range of ***2*** thru ***6***, and the sixth character must be an ***a***.



In this example we're illustrating how the period character works. The period states to match any printable character.

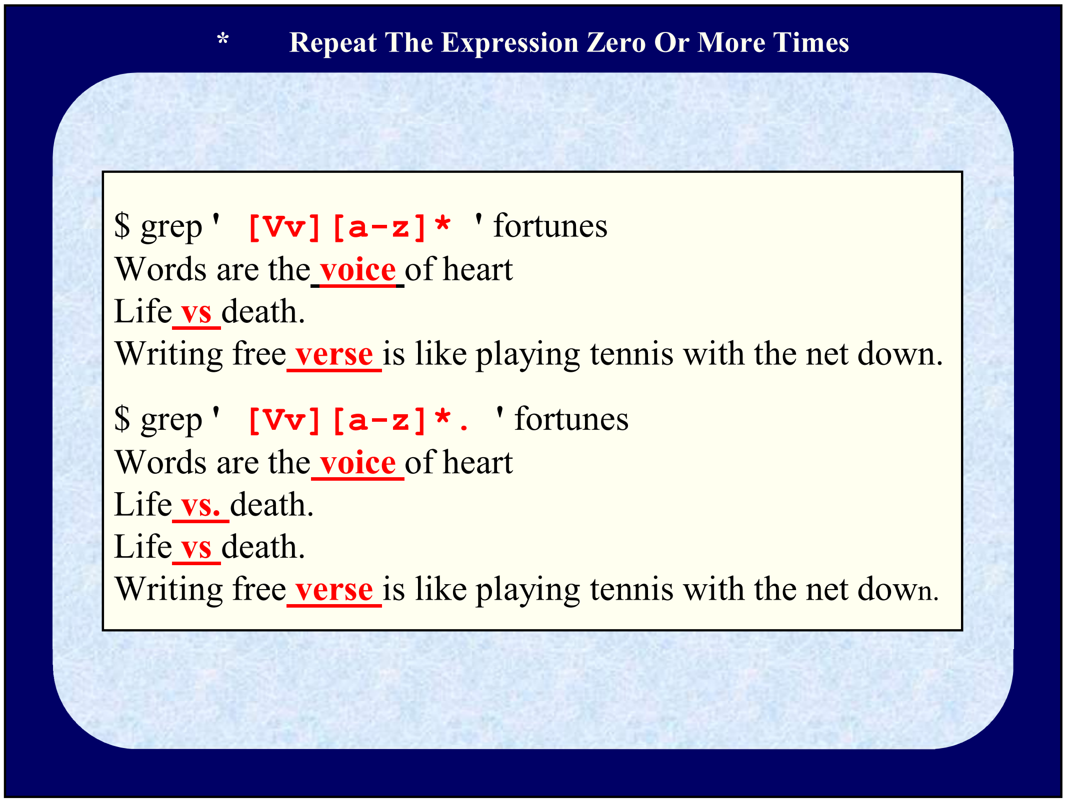
The first example, with the expression '**[Uu]…**', states to match an uppercase or lowercase ***U***, followed by three printable characters. This match can occur anywhere on the line. I have underlined what satisfied the match in the first example.

The second example has the expression ' **[Uu]…** ' to states match a space; followed by an uppercase or lowercase ***U***, followed by three printable characters, then the next character must be a space. The pattern matches have been underlined in this slide to highlight the matches. The RegEx specifies space in the beginning and end of the pattern match.

Although these examples are bizarre, notice the power of pattern matching using UNIX regular expressions. It is important to note how effective pattern matches can be when specific instructions are provided:

- Match single printable character
- Match RegEx pattern zero or more times
- Match RegEx pattern zero or one time
- Match RegEx pattern one or more times

The ability to use spanning of RegEx patterns is another powerful feature!

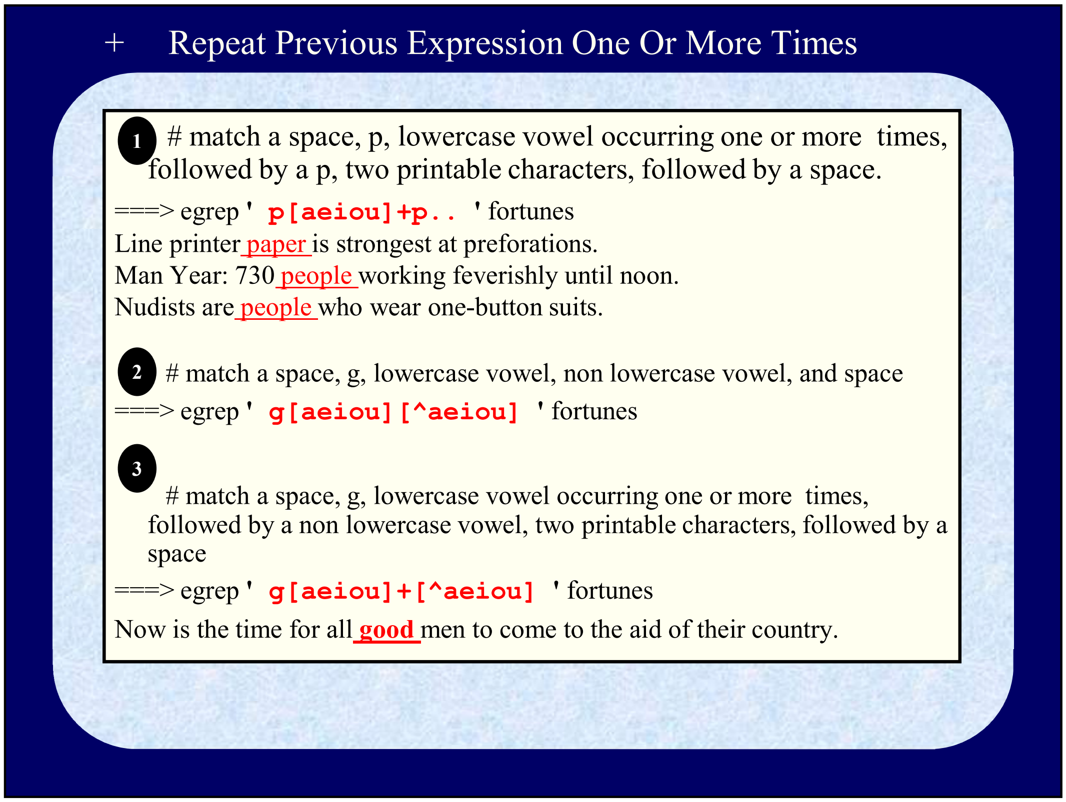


This slide illustrates the strength of the asterisk character (*). The interpretation of this character is to match zero of more occurrences of a pattern.

In the first example, the expression ‘ **[Vv][a-z]*** ‘, states to match a space, then an uppercase or lowercase *V*, then match the the lowercase character range of a thru z that can occur zero or more times, followed by a space. Notice that three lines match this expression.

However, by slightly modifying the expression to include a period after the asterisk alters the space.

Life vs. Death satisfies this match because in addition to the previous records because RegEx **[a-z]*** matches the **s** one time followed by the period which matches the criteria of a printable character which happens to be a period.

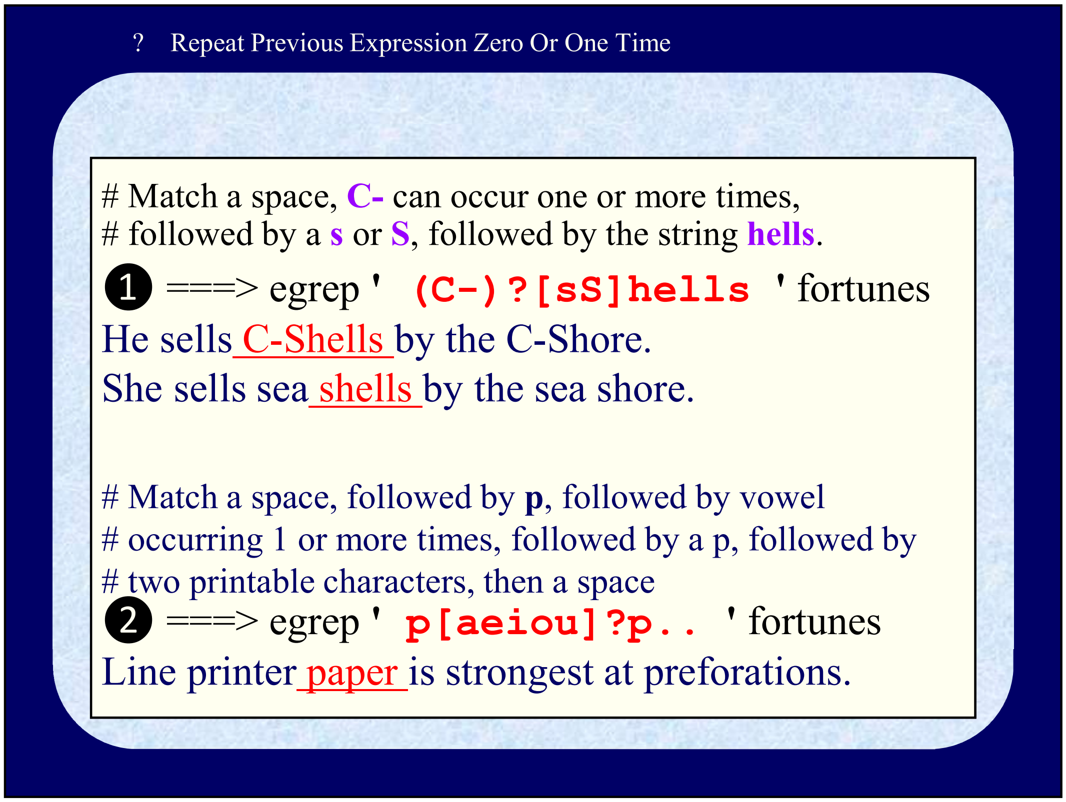


This slide illustrates the special interpretation of the `'+'` character which has the special meaning to repeat the previous expression one or more times.

In the first example we want to match a space; followed by matching a ***p***. followed by character matching a lowercase vowel that can occur one or more times; followed by a p; followed by a space.

In the second example, notice the match achieved with the "+" interpreted match the previously defined pattern match. The first character is a space; followed by matching a ***g***; followed by matching a non-lowercase vowel; followed by a space character. However, no match was satisfied.

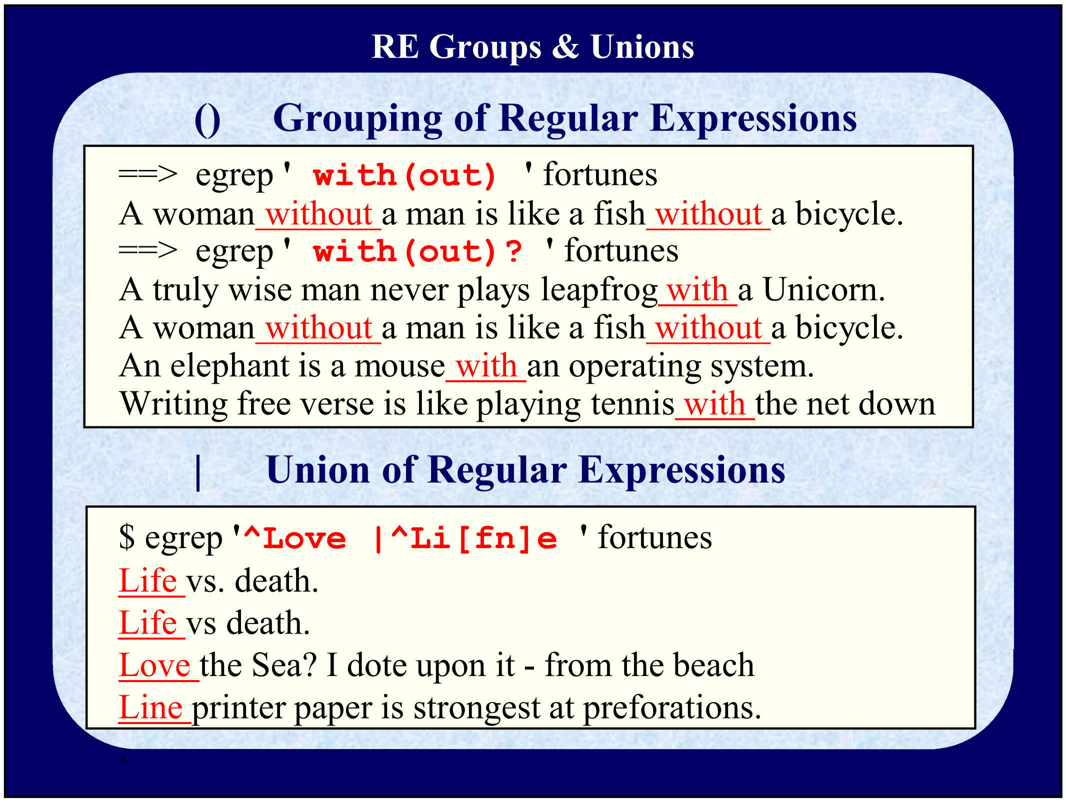
In the third example, notice the difference when a `'+'` is appended to the **[aeiou]** pattern match that is defined as match a lowercase vowel that can occur one or more times. Thus, a match is made with **`' good '`**.



This example illustrates the use of the question mark which means repeat the previous expression zero or one time. This question mark is an extended regular expression and should be used with egrep.

The first example wants to match the expression **`' (C-)?[sS]hells '`** which states to match a space. Match two characters ***C-*** occurring zero or one time. The next character must be an uppercase or lowercase ***S***. The remaining characters must be the sting *hells;* followed by a space. This is why the words ***C-Shells*** and ***shells*** satisfies the match. The RegEx patterns matched are *C-shells, Shells*, and *shells* are the pattern defined. If the RegEx is defined as **' (C-)[sS]hells '**, only ***C-Shells*** and ***C-shells*** would the only strings to satisfy the match.

In the second example, we want to match the expression **`' p[aeiou]?p.. '`**which states match a space character; followed by a *p*; followed by a lowercase vowel that can occur zero or one time; followed by a ***p****;* followed by two printable characters; followed by a space. In this example, the word **paper** satisfies this match.

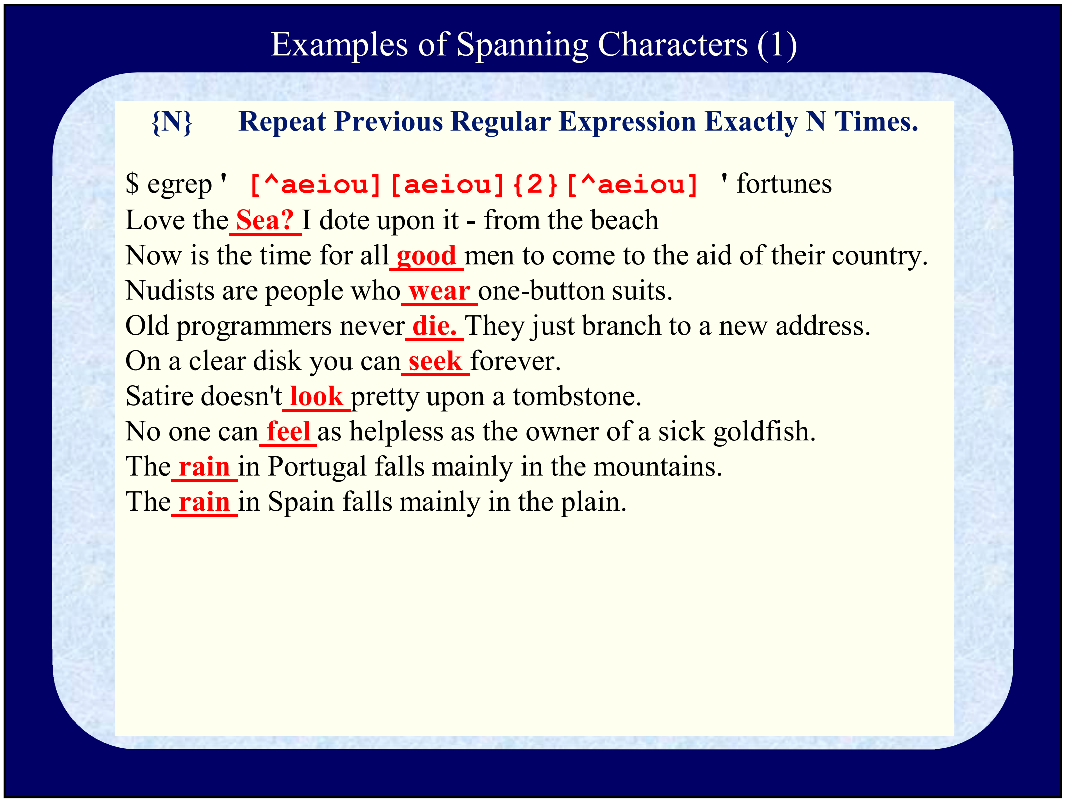


The use of parentheses is very powerful for grouping expressions. Especially when we use the special characters.

In the first example, we're illustrating how the **?** character would work on an expression. The first expression `' with(out) '` is defined as matching a space character; followed by the string *with*. The next three characters is the string *out* grouped by parentheses. The final character is a space. This pattern matches the word *without*. This example illustrates potential mistakes.

This slide illustrates the effect of using a **?** after the grouping of the string **out**. We're now able to match the words ***with*** or ***without***.

In the next example we are illustrating the ability of **Unions** (or logical oring). Notice that combinations of expressions are being used; better known as a *compound regular expression*. The first expression is `'^Love '` which is to match the word **Love**; followed by a space at the beginning of the line. The second RegEx pattern states match ***Li*** at the beginning of a line; followed by the third character to be a ***f*** or a ***n***. The fourth character is an ***e***; followed by the fifth character being a space at the beginning of the line. In summary, we're attempting to match `'Love '`, `'Life '`and `'Line '` appearing at the beginning of a line.



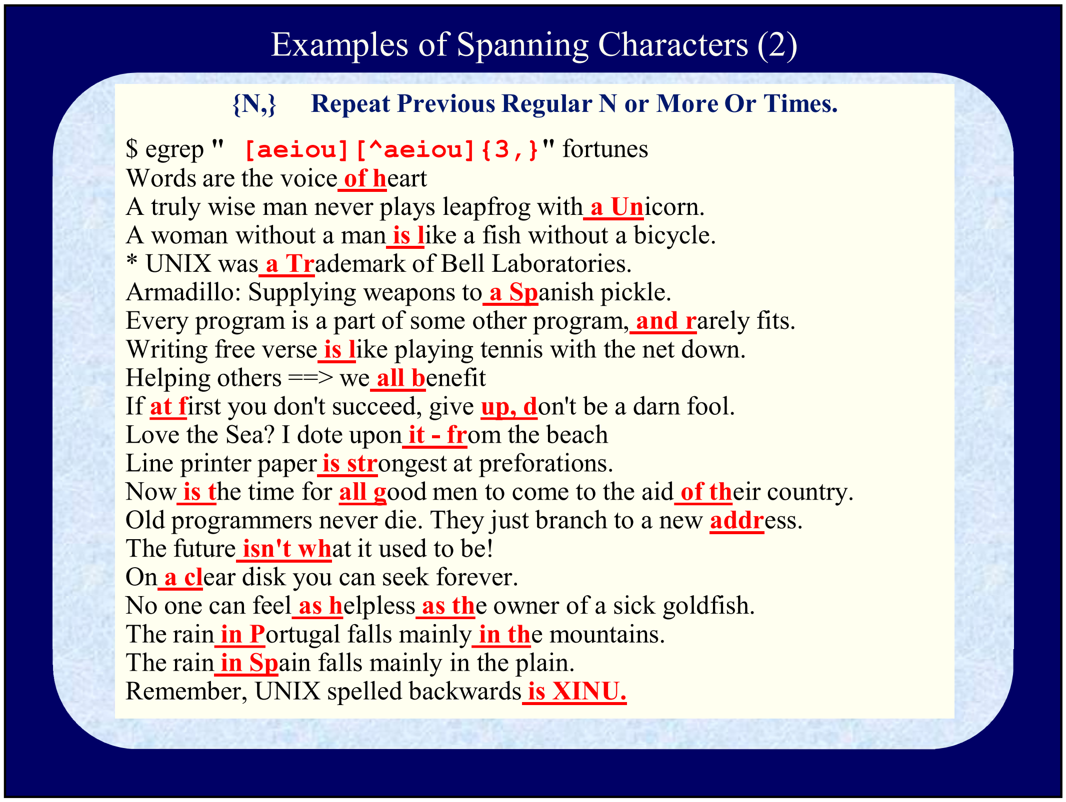
{N} Repeat previous regular expression exactly N times

This slide illustrates the spanning of characters to match the pattern exactly N times.

In this example, were illustrating how to match exactly N times. The RegEx matches a space character; followed by a non-lowercase vowel; followed by lowercase vowel occurring exactly two times; followed non-lowercase vowel; followed by a space. The pattern match will not occur at the beginning or the end of the line. Most of the lines are obvious. However, why did **Sea?** and **die.** satisfy the match?

List below are the shorthand notations for spanning charachters:

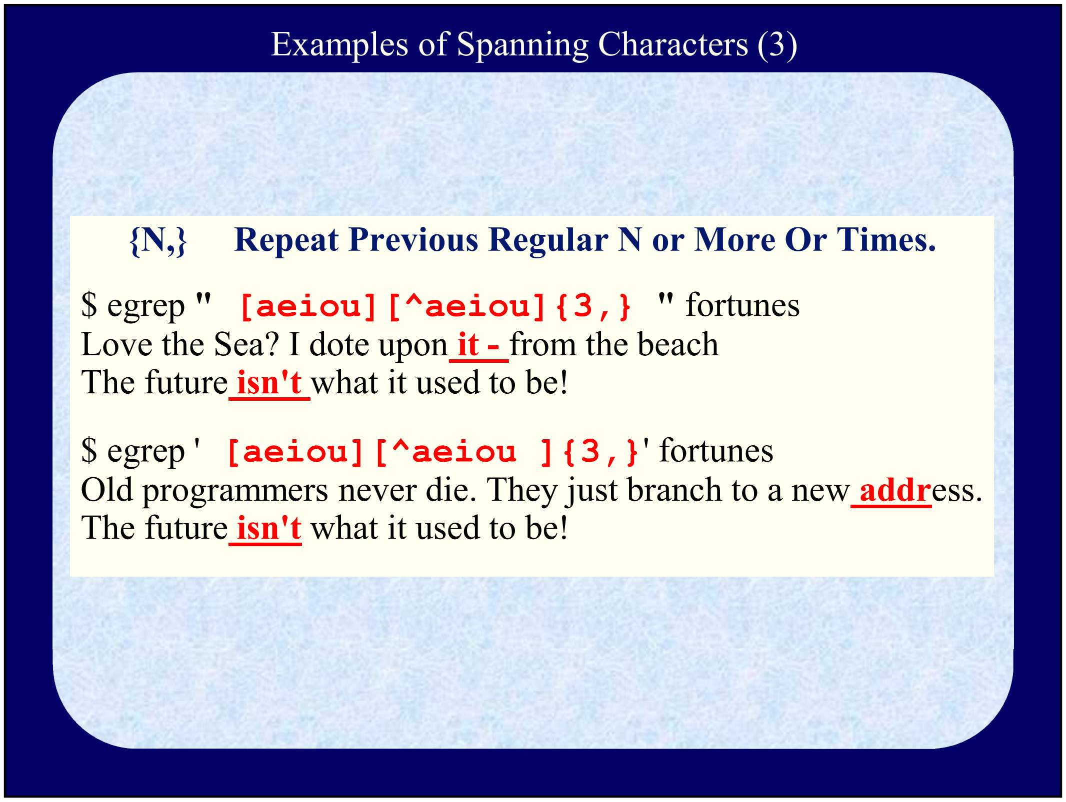
*	Shorthand notation for \{0,\}
+	Shorthand notation for \{1,\}
?	Shorthand notation for \{0,1\}



{N,} **Repeat the previous expression N or more times**

This slide illustrates repeating the previous expression N or more times

This example we want to illustrate matching N or more times. The expression defined states to match a space, followed by a vowel, the next character must be a non lowercase vowel occurring three or more times.

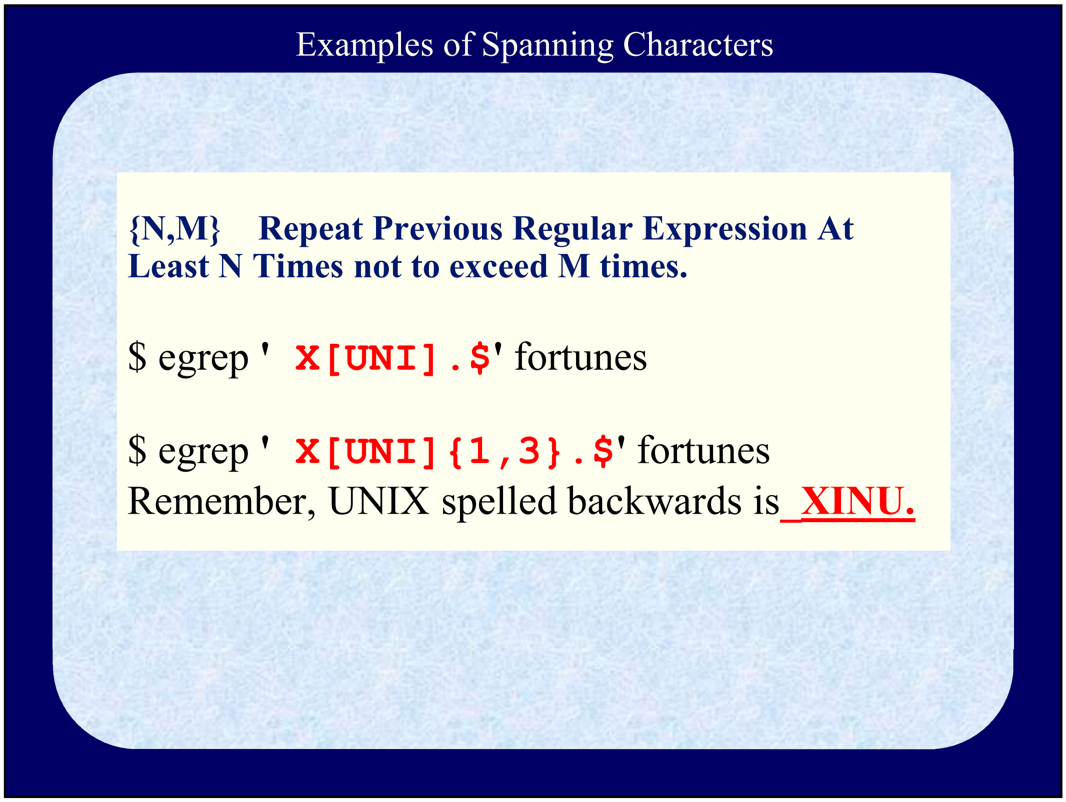


{N,} **Repeat the previous expression N or more times**

The examples in this slide include:

- First example, match a space; followed by a vowel; followed by a non lowercase vowel occurring three or more times; followed by a space. It should be obvious this pattern match can take place in the beginning or ending of a line.
- Second example, match a space; followed by a vowel; followed by a non lowercase vowel or space occurring three or more times.

Notice the difference between having a space and not having a space in the negation pattern match **[^aeiou]**



{N,M} **Repeat the previous expression at least N times; not to exceed M times**

This slide illustrates matching a pattern at least N times not to exceed M times. The first egrep command:

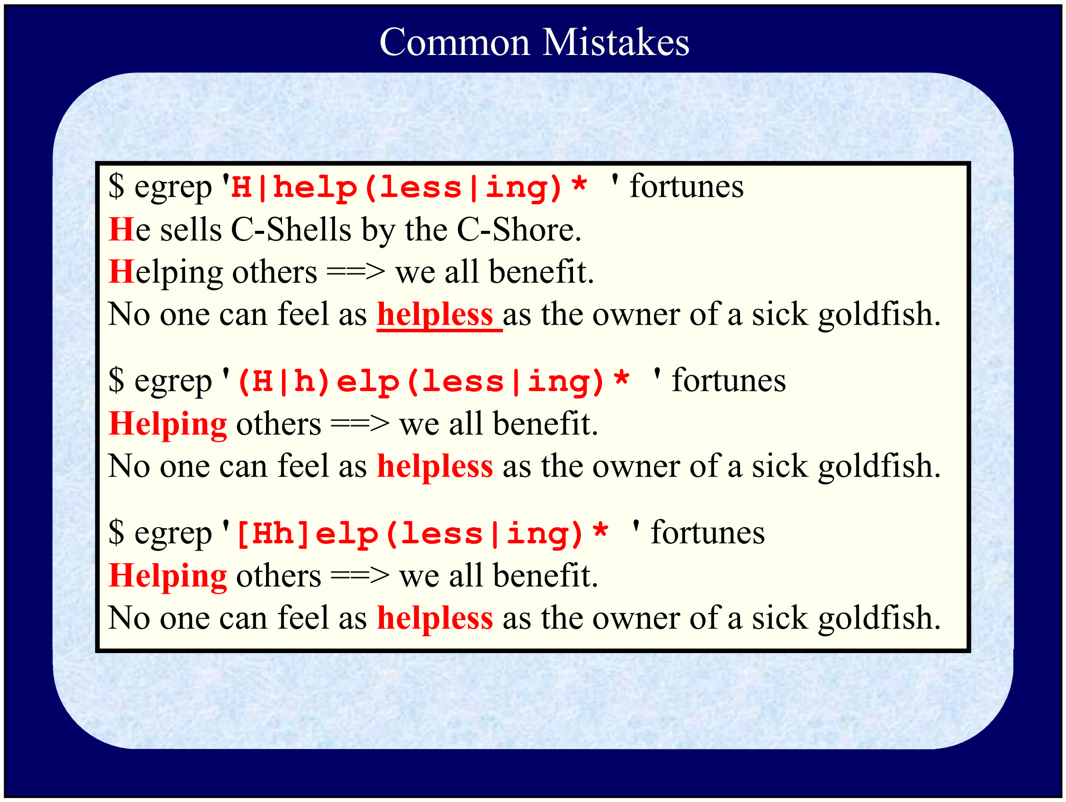
- Match a space; followed by a capital **X**; followed by a **U**, **N**, or **I** that occurs one time; followed by any printable character at the end of the line at the ed of the line.

Notice nothing matches.

With the next egrep command:

- Match a space; followed by a capital **X**; followed by a **U**, **N**, or **I** that occurs at least one time, not to exceed three times; followed by any printable character at the end of the line.

Notice the difference between the two when we span the **U**, **N**, or **I** up to three times and not spanning charachters.



This slide illustrates mistakes that can be refined and improve upon the expression.

We're attempting to match the words *Help*, *help*, *Helpless*, *helpless*, *Helping*, or *helping*. Our first attempt was trying the compound expression **'H|help(less|ing)*** ' that:

- Match H or help anywhere on a line
- Followed by the character string **less** or **ing** that can occur zero or more times

"He" was not a match we were looking for!

The second attempt works better since ' **(H|h)elp(less|ing)*** ' performs a parentheses grouping to match an *H* or *h*; followed by the string *elp*, and the suffix *less* or *ing* that can occur zero or more times. This pattern match is much better than the first works.

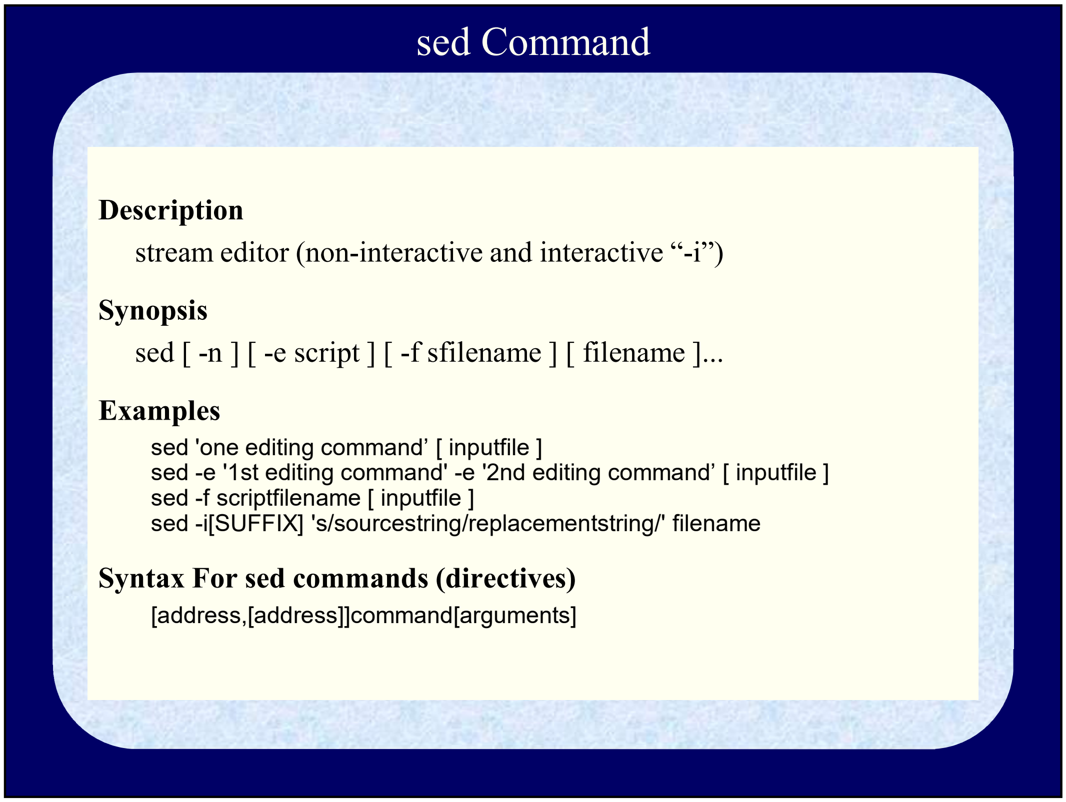
However, there is debate to, whether or not, the third expression ' **[Hh]elp(less|ing)*** ' is be better. It may be better since it is more succinct:

- Match an H or h followed by
- the character string **elp**; followed by
- the character string **less** or **ing** that can occur zero or more times

This brings up to the process of fine tuning or debugging pattern matches:

- Test: Ensure Regular Expressions Work As Expected!
- Evaluate Results With The Following Guidelines:
  - *Hits That Should Be Misses
  - Hits
  - Misses
  - *Misses That Should Be Hits
- Perfect Your Descriptions: by working opposite ends
  - Working at Opposite Ends. Eliminate:
    - »"Hits That Should Be Misses"
    - »"Misses That Should Be Hits."!

Remember to archive regular expressions: <u>They Could Be Reusable!</u>



The sed command is a very powerful streamline editor. It began as a non-interactive editor. However, it now includes an interactive feature.

This slide highlights the command synopsis. The examples provided demonstrate completing edit changes on matching lines and outputting the remaining lines without modification.

You can have one edit directive or multiple edit directives. Just remember to use the **-e** option with multiple edit directives.

Edit directives passed by reading a file containing edit directives.

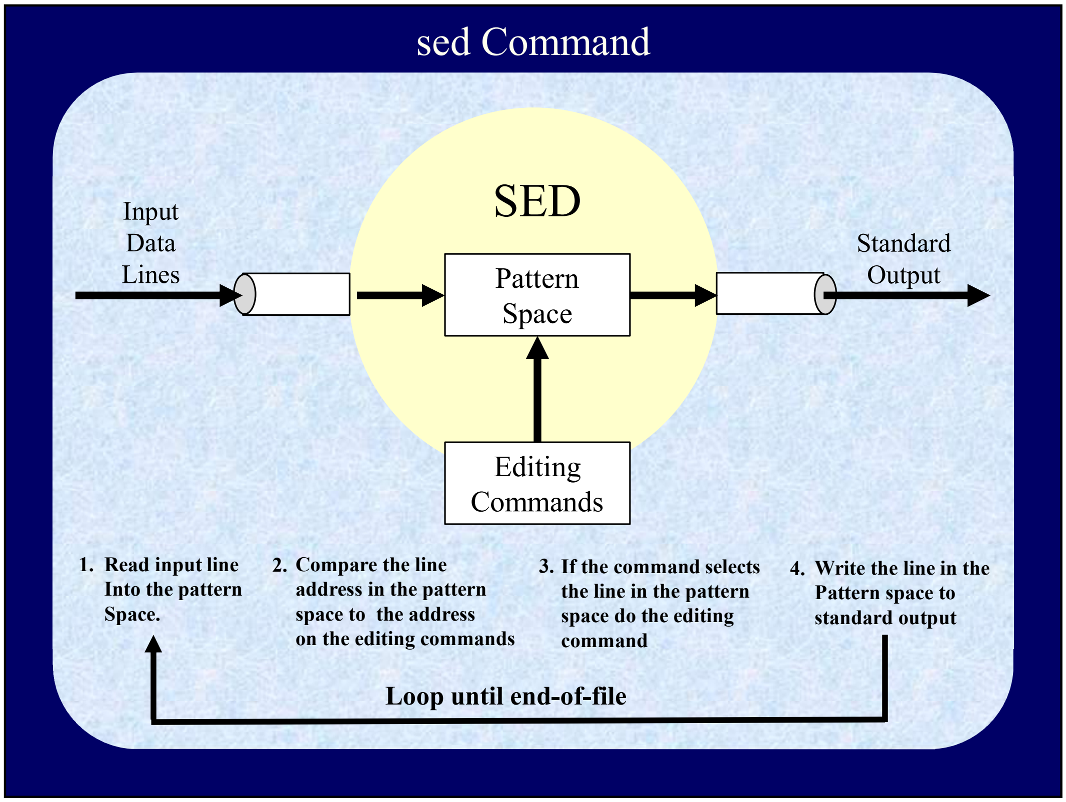
It should also be noted, the **–i** option is known as the **in-place** editing option. It allows in-place editing of files creating a temporary output file in the background. Afterwards, the original file is replaced by the temporary file.

The in-place option has two available options:

- **-i[SUFFIX]**
- **--in-place[=SUFFIX]**

If a **SUFFIX** is supplied, a backup copy of the original made with the suffix appended to the original filename. **IMPORTANT NOTE:** When using the **in-place editing** option, you must specify a file for input; as you cannot pipe standard output to a sed command. If you need to update multiple files, then embed the sed command in a loop (**for-loop** or **while-loop**)

Notations combined with commands have two methods. One may specify these editing commands on a **sed** command line; or you can place these editing commands in a ASCII text file.



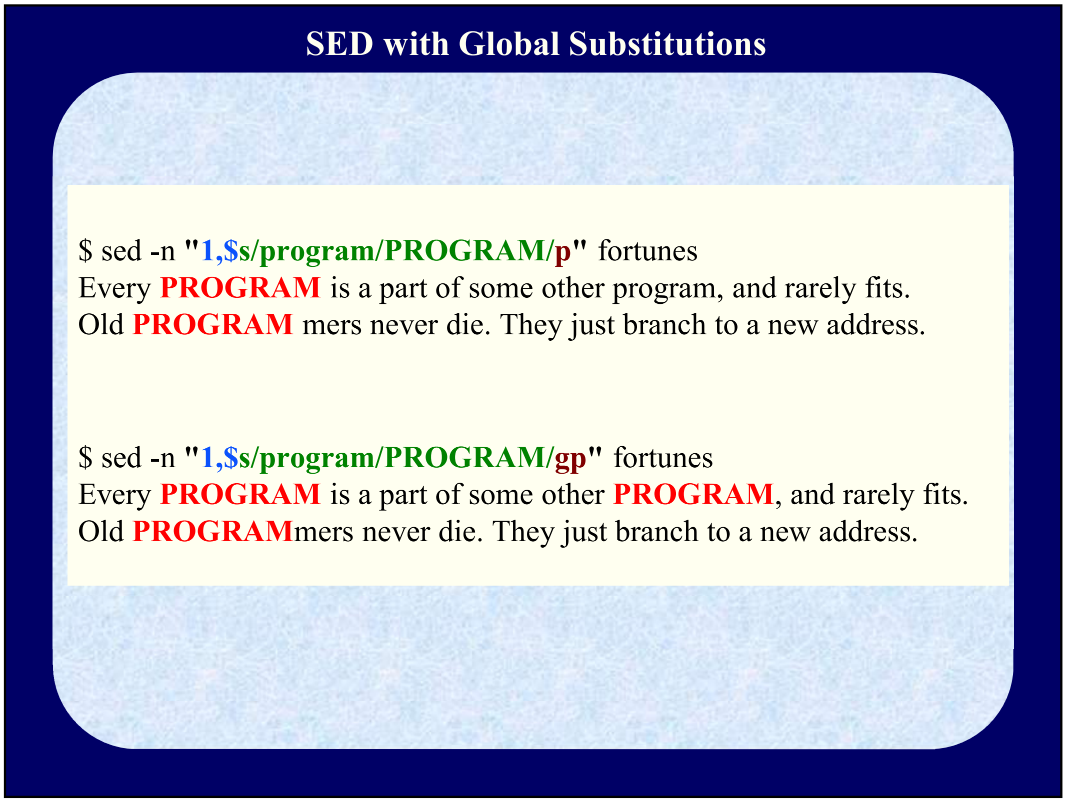
This slide paints the image of how the sed command functions as a line editor.

Edit directives passed by reading a file containing edit directives. The **sed** editing commands are almost identical to commands (directives) used by **ed**.

Edits can occur referencing address lines or pattern-match notations.

The **sed** command is a very powerful editing tool with a wide variety editing command capabilities to modify data streams in many different ways.

The sed process involves four step, highlighted in slide, until the end-of-file is reached.



This slide illustrates how we can make text edit changes quickly without invoking an interactive text editor.

The substitution mechanism works on the concept of:

/ regular expression/substitution string/

In the first example, we're substituting the string *program* with *PROGRAM*. Notice the first line will only substitute the first match.

If we want to substitute all occurrences on the line, we must include a ***g*** directive after the substitution directing to perform a global substitution of all occurrences qfor each record process.

The **-n** option should be and the **p** directive should be explained. This **-n** flag tells sed not to print to standard output. In essence no lines are to be printed. However, the **p** directive tells sed to print the lines it did process.

SED – Replace Tabs with Vertical Bar Globally

```
# Display the contents Autos file
$ cat -tve Autos
Accura^IIntegra^I1987^I35987$
Dodge^IAries^I1982^I1035987$
Jeep^ICherokee^I2021^I48000$
Chevy^IMalibu^I2004^I35999$
Ford^IMustang^I2015^I120125$
Mercury^IMonarch^I2020^I55987$
Plymouth^IShadow^I1994^I125888$
Nissan^IMaxima^I1994^I125888$
Subaru^IOutback^I2023^I5209$
Toyota^IRav4^I2025^I467$
# Substitute globally tab characters with the pipe (vertical bar)
$ sed 's:\t:|:g' Autos
Accura|Integra|1987|35987
Dodge|Aries|1982|1035987
Jeep|Cherokee|2021|48000
Chevy|Malibu|2004|35999
Ford|Mustang|2015|120125
Mercury|Monarch|2020|55987
Plymouth|Shadow|1994|125888
Nissan|Maxima|1994|125888
Subaru|Outback|2023|5209
Toyota|Rav4|2025|467
```

The Autos file contains four fields for each record:

1. Car Manufacturer
2. Model
3. Year
4. Mileage

The **cat** command uses 3 options:

-v: show non-printing characters
-t: display TAB characters as ^I
-e: display $ at end of each line

Thus notice the tab **^I** output from the cat command.

The sed command is searching for the TAB character with a vertical bar (pipe sign).

SED – Insert/Change/Append Text

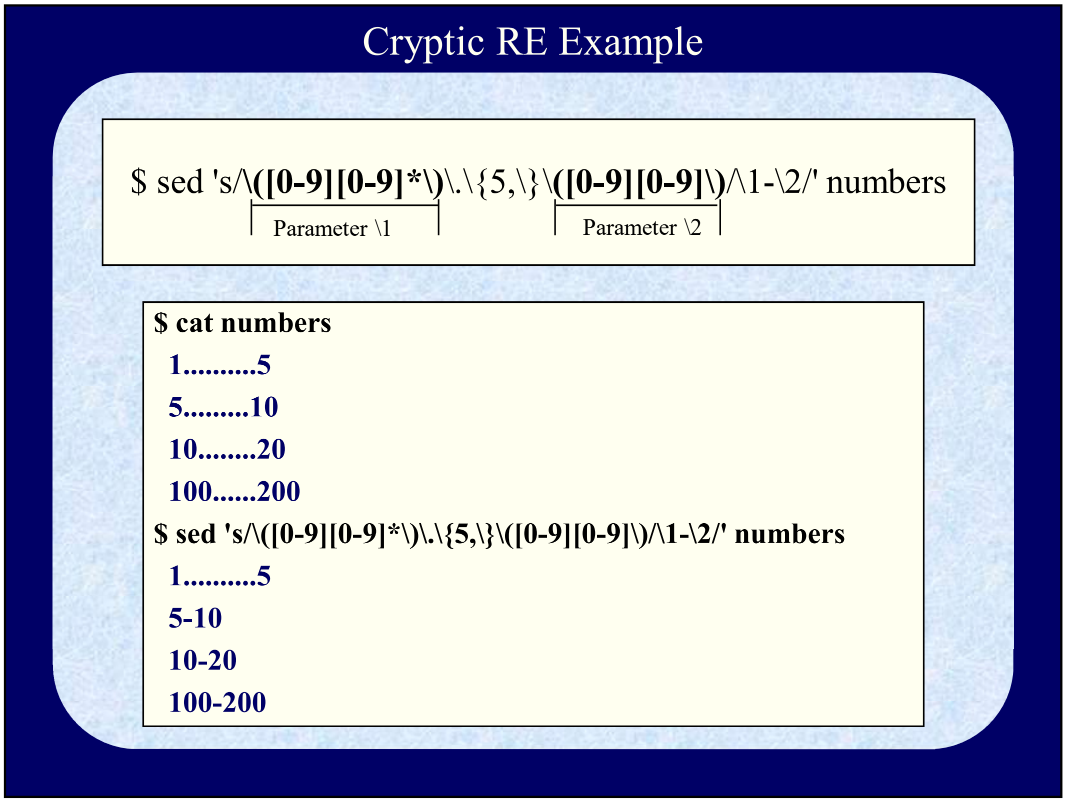
```
== Display contents of Autos with line numbers ==
$ nl Autos2
     1  Accura  Integra 1987    35987
     2  Dodge   Aries   1982    1035987
     3  Jeep    Cherokee        2021    48000
     4  Chevy   Malibu  2004    35999
     5  Ford    Mustang 2015    120125
     6  Mercury Monarch 2020    55987

== Contents of sedscript file ==
$ cat sedscript
/Accura/iInsert Above Line 1
/Chevy/cReplacing Chevy record (record 4)
/Ford/aAppending after line 5

== Results of stream edit ==
$ sed -f sedscript Autos2 | nl
     1  Insert Above Line 1
     2  Accura  Integra 1987    35987
     3  Dodge   Aries   1982    1035987
     4  Jeep    Cherokee        2021    48000
     5  Replacing Chevy record (record 4)
     6  Ford    Mustang 2015    120125
     7  Appending after line 5
     8  Mercury Monarch 2020    55987
```

Notice the Autos2 file has 6 records. It is helps displaying the records with line numbers. The field separator for each record is the tab character.

The sed script has the following instructions:

- Insert the text **"Insert Above Line 1"** above the Acura record.
- Replace the Chevy record with the text **"Replacing Chevy record (record 4)"** on line 4.
- Append the text **"Appending after line 5"** below the Ford record

The command **`sed -f sedscript Autos2`** produces the following results illustrated in the slide.



Now getting back to the terse beginning I introduced at the beginning.

Sed supports most of the regular expression grep does. However, parentheses are used to establish.

With sed, we can group pattern matches with parentheses to create positional parameters. Notice that the parentheses are preceded with a backslash. This is necessary to define the positional parameters. These parameters are grouped as follows:

- First positional (\1) field matches a digit; followed by a period that can occur 5 or more times; followed by a digit can occur zero or more times; followed second positional (\2) parameter is matching a digit; followed by a second digit.

The first redord

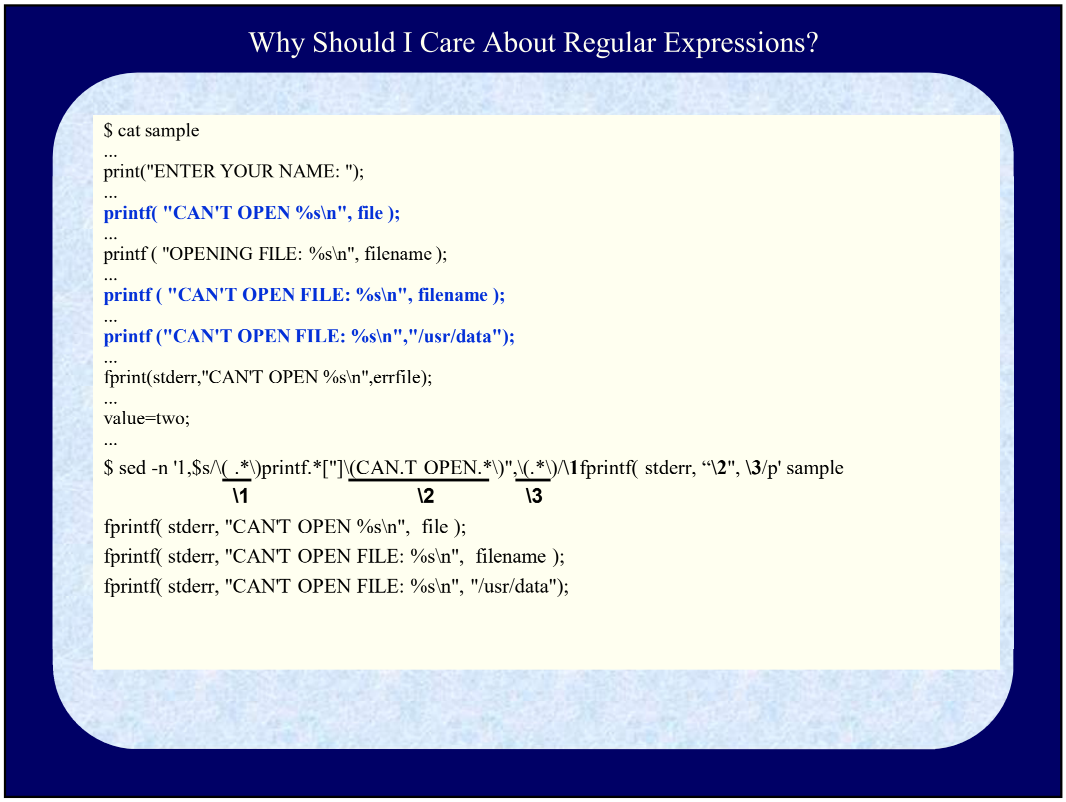
In the substitution, we will replace the periods with a dash. Notice the results from executing the command (based upon the contents of the *numbers* file. The first line failed because we didn't the * special character when establishing a second positional parameter. The second expression only satisfies the match for a two-digit value! A better expression would be:

```
sed 's/\([0-9][0-9]*\)\.\{5,\}\([0-9][0-9]*\)/\1-\2/' numbers
```

producing:

5-10
10-20
100-200

New GNU **sed** command may have difficulty with the + character for the shorthand notation matching the regular expression 1 or more times.



Most people make think regular expressions are useless. However, in the eighties knowing this skill paid big dividends.

Many COBOL source code files could have syntax changes done through filtering. I remember filtering 1000 program files in three days with very good results.

In the scenario listed here. A programmer made the mistake not to use *fprintf()* function call to include file descriptors. Instead, he would use *printf()* that would always print to standard out.

This sed program illustrates how we can match can’t open file messages and replace:

1. Replace ***printf*** function call with ***fprintf***.
2. Create **positional parameter 1** that include all the characters leading up to ***printf***.
3. Store **"Can’t Open"** and any additional character just before the double quote and store the string in **positional parameter 2**.
4. All characters after the double quote are stored in **positional parameter 3**.

In the replacement string involved the following instructions:

1. Write the character string stored in **positional parameter 1.**
2. Write the character string **"*fprintf (stderr,* "**.
3. Write the character string stored in **positional parameter 2**.
4. Write a double quote; followed by a comma and space character
5. Write the character string stored in **positional parameter 3**.

This sed command took 25 minutes to write. Think of the time savings when it was estimated 900+ programs needed to be modified.

In-place substiution

```
$ cat df-data
/dev/mapper/ol_vbox-root    3080192  166152   2914040   6% /
/dev/mapper/ol_vbox-usr    10420224 5813800   4606424  56% /usr
/dev/mapper/ol_vbox-tmp     2031616   47324   1984292   3% /tmp
/dev/mapper/ol_vbox-local  46063616  396304  45667312   1% /local
/dev/mapper/ol_vbox-home    5177344  322688   4854656   7% /home
/dev/sda1                   2031616  770948   1260668  38% /boot
/dev/mapper/ol_vbox-opt    15663104  159064  15504040   2% /opt
/dev/mapper/ol_vbox-var     5177344 1677068   3500276  33% /var
$ sed -ri.$(date "+%Y%m%d_%H%M%S_")$$ 's: +:|:g' df-data
$ ls df-data*
df-data  df-data.20250403_141153_133139
$ cat df-data
/dev/mapper/ol_vbox-root|3080192|166152|2914040|6%|/
/dev/mapper/ol_vbox-usr|10420224|5813800|4606424|56%|/usr
/dev/mapper/ol_vbox-tmp|2031616|47324|1984292|3%|/tmp
/dev/mapper/ol_vbox-local|46063616|396304|45667312|1%|/local
/dev/mapper/ol_vbox-home|5177344|322688|4854656|7%|/home
/dev/sda1|2031616|770948|1260668|38%|/boot
/dev/mapper/ol_vbox-opt|15663104|159064|15504040|2%|/opt
/dev/mapper/ol_vbox-var|5177344|1677068|3500276|33%|/var
```

Prior to the **in-place editing** feature:

1. First make a backup copy of the file or files.
2. Perform the the sed edits needed and redirect the output to new files
3. Replace the original files with their corresponding new files

With the **GNU sed** with i**n-place edits**:

1. The edits are stored in a temporary file.
2. A backup file is created when an **suffix** extension is provided
3. After the edit is complete, overwrite the file with the temporary file.

In this example, of **in-place substitution**. This feature was made available with the **GNU distribution** of ***sed***. The /RegEx/ReplacementString/ is **`'s: +:|:'`** is stating to replace all spaces with a pipe sign (vertical bar).

Notice the contents of **root-dirlist** before the edit and the contents after the edit.

The **SUFFIX** is **`.$(date "+%Y%m%d_%H%M%S_")$$`** translates to:

`.YYYYMMDD_HHMMSS_PID`

PID represents **$$** is a special parameter being the value current **process ID**. Notice the extension

Substituting with ex directives

```
/dev/mapper/ol_vbox-root    3080192  166152   2914040   6% /
/dev/mapper/ol_vbox-usr    10420224 5813800   4606424  56% /usr
/dev/mapper/ol_vbox-tmp     2031616   47324   1984292   3% /tmp
/dev/mapper/ol_vbox-addon  41877504  325020  41552484   1% /addon
/dev/mapper/ol_vbox-local  46063616  396304  45667312   1% /local
/dev/mapper/ol_vbox-home    5177344  322688   4854656   7% /home
/dev/sda1                   2031616  770948   1260668  38% /boot
/dev/mapper/ol_vbox-opt    15663104  159072  15504032   2% /opt
/dev/mapper/ol_vbox-var     5177344 1676944   3500400  33% /var
~
:1,$s: \{1,\}:|:g
```

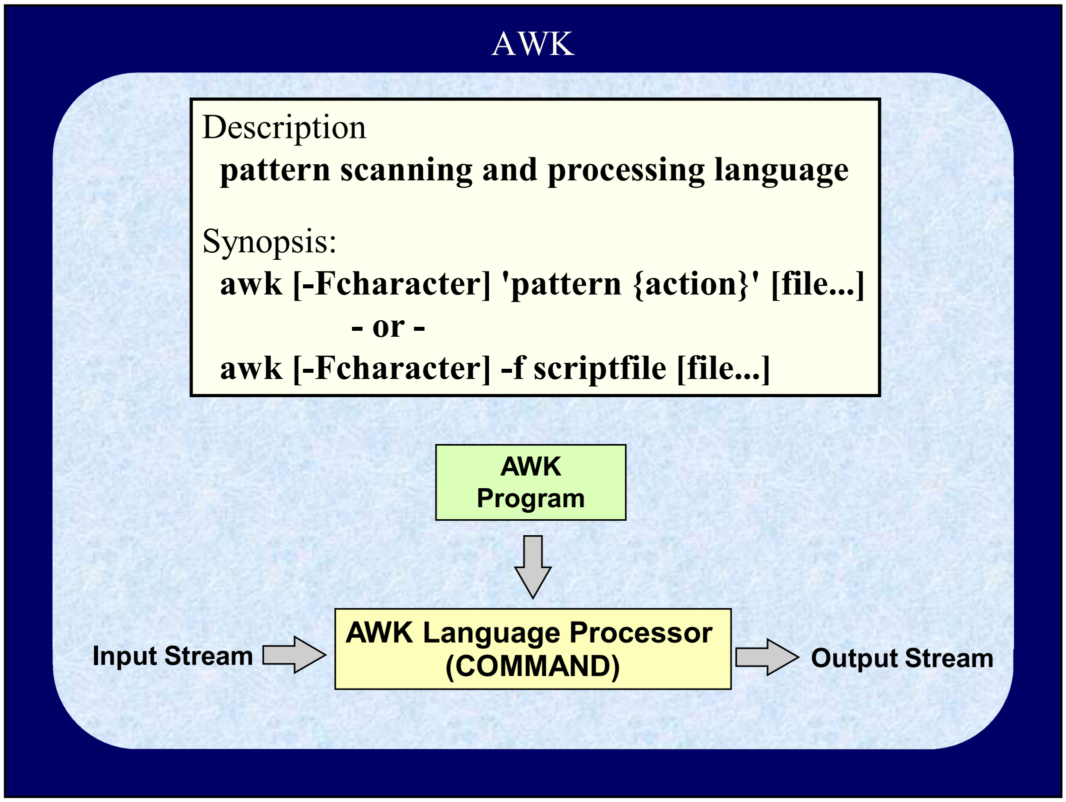
Results of ex directive edit:

```
/dev/mapper/ol_vbox-root|3080192|166152|2914040|6%|/
/dev/mapper/ol_vbox-usr|10420224|5813800|4606424|56%|/usr
/dev/mapper/ol_vbox-tmp|2031616|47324|1984292|3%|/tmp
/dev/mapper/ol_vbox-addon|41877504|325020|41552484|1%|/addon
/dev/mapper/ol_vbox-local|46063616|396304|45667312|1%|/local
/dev/mapper/ol_vbox-home|5177344|322688|4854656|7%|/home
/dev/sda1|2031616|770948|1260668|38%|/boot
/dev/mapper/ol_vbox-opt|15663104|159072|15504032|2%|/opt
/dev/mapper/ol_vbox-var|5177344|1676944|3500400|33%|/var
~
45 substitutions on 9 lines
```

This example illustrates how to make text edits using the vi ex directive feature. This is drastically faster than the manual edits.

The ex directive **`'1,$s: \{1,\ }:|:g'`** states:

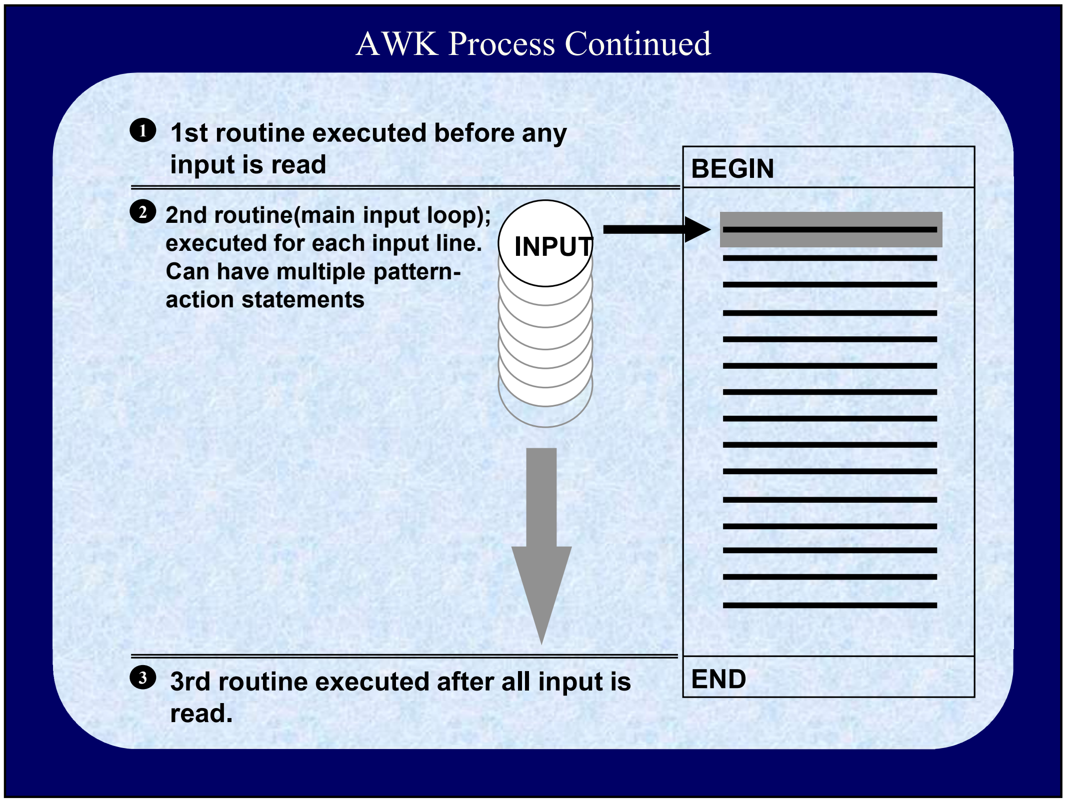
1. Match a space that can occur **<u>1 or more times</u>** and replace it with a single vertical bar (or pipe character) globally.



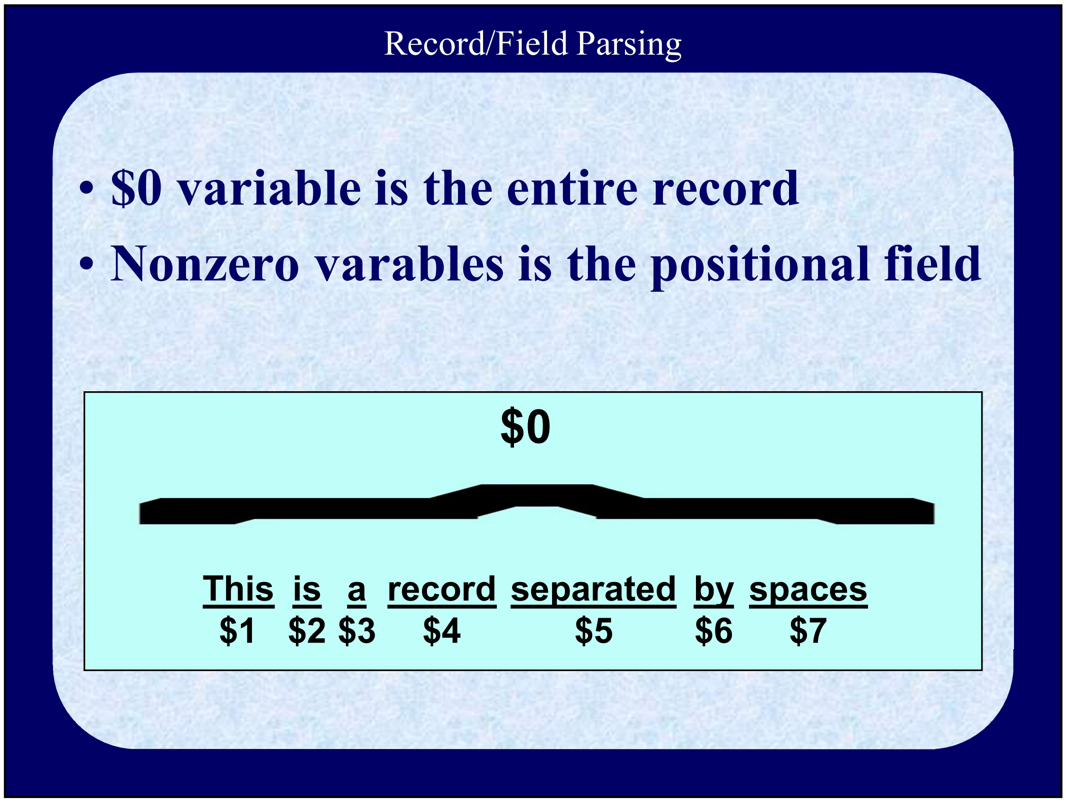
Awk is another useful command for text filtering. The nice part is it includes useful programming constructs. Like grep, it's a text filter command.

AWK Program is the series of **' pattern {action} '** statements**.** The AWK language Processor **is the command itself** (awk, gawk, and nawk)**.** The input/output Streams are ASCII Text Records. The text can be treated as strings, numeric quantities, individual characters fields.

It should be patterns can include relation expressions in addition to regular expressions.



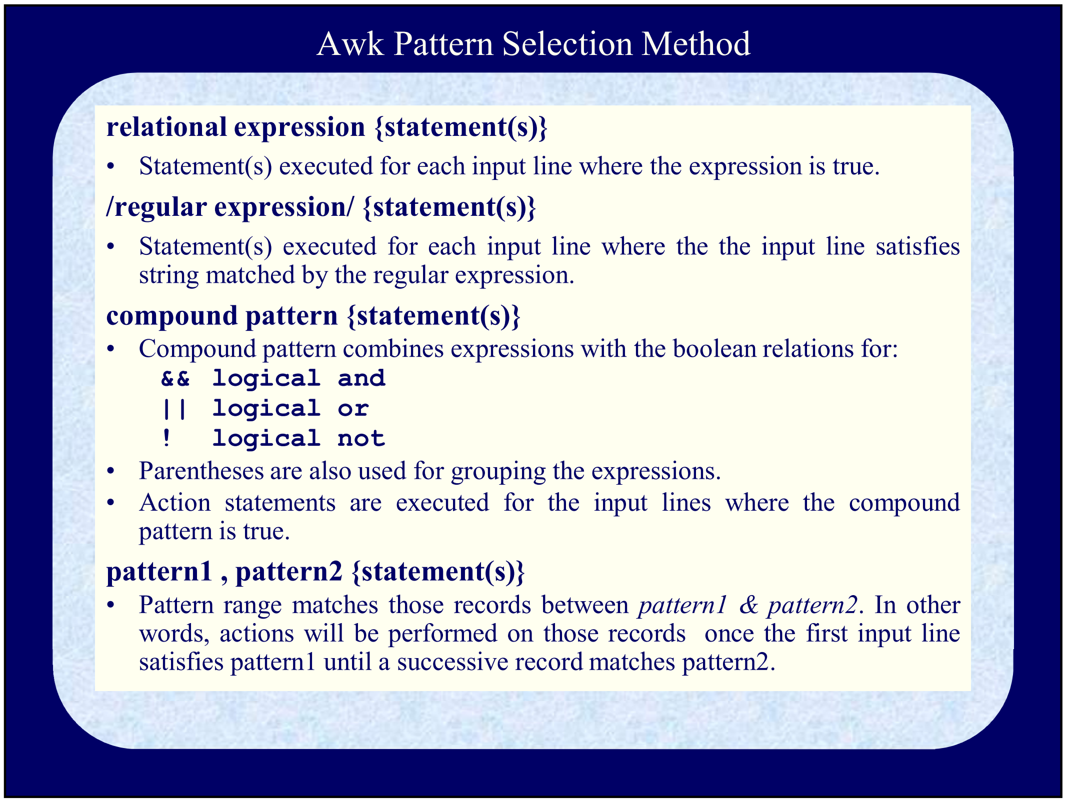
Since **awk** can be very intimidating to those first learning this program. It's important to note that is method of processing data is rather simple. Before any data is processed, **awk** will process an existing **BEGIN statement**. Then it will process text for the files it's instructed to process. The lines of data read will be compared to each pattern action statement defined. After all the data is processed, **awk** will then process an **END statement**.



Awk splits input stream into <u>records</u> based upon the value of the the built-in variable for the <u>record separator (RS)</u>. The default is generally the newline (\n).

Then it splits input stream into <u>fields</u> based upon the value of the the built-in variable for the <u>field separator (FS)</u>. The default value is white-space (combination of spaces and tabs).

This slide illustrates the concept of the entire record; and the fields that make up that record.

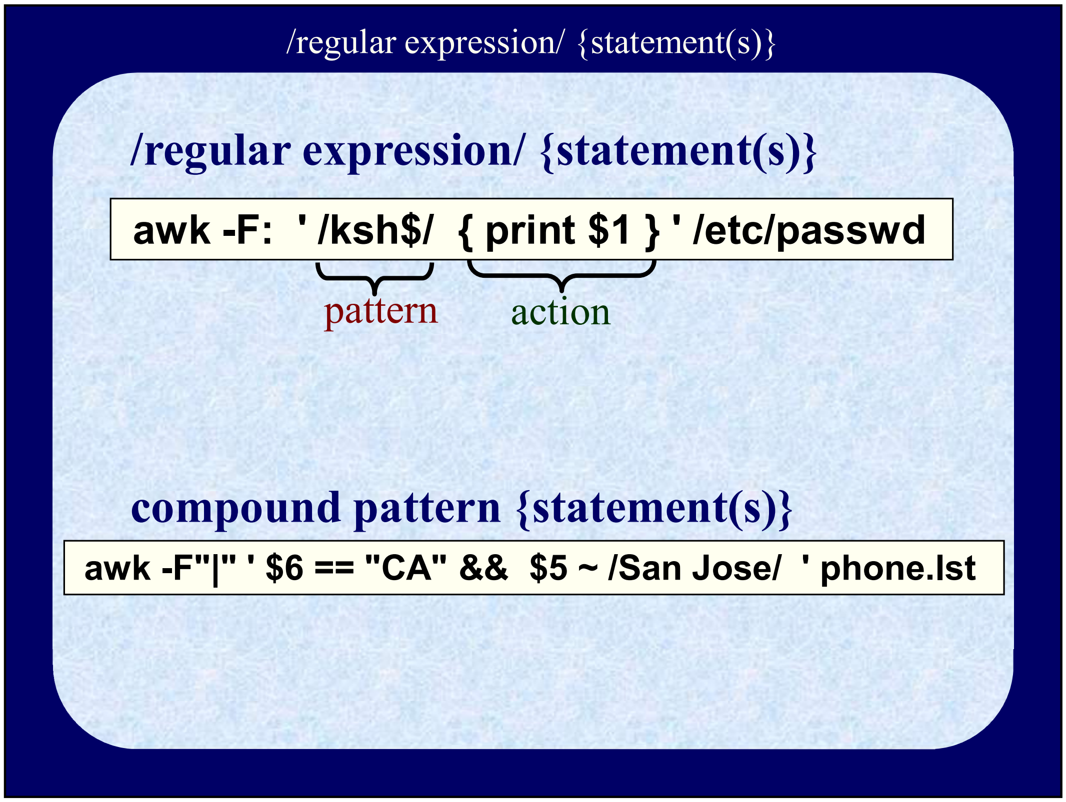


Although awk can not be covered in complete detail, this slide is here for informational purposes to let you know the various pattern statements that awk supports.

Relational operators are supported such as the if statement. The more common pattern statement is the regular expression statement.

Awk does support compound pattern expressions. These compound expressions can be a combination of regular expressions with relational expressions.

We pattern1, pattern2 expressions are very interesting when wanting to retrieve data within a certain range.



The slide illustrates a regular expression example and a compound expression example that uses a regular expression.

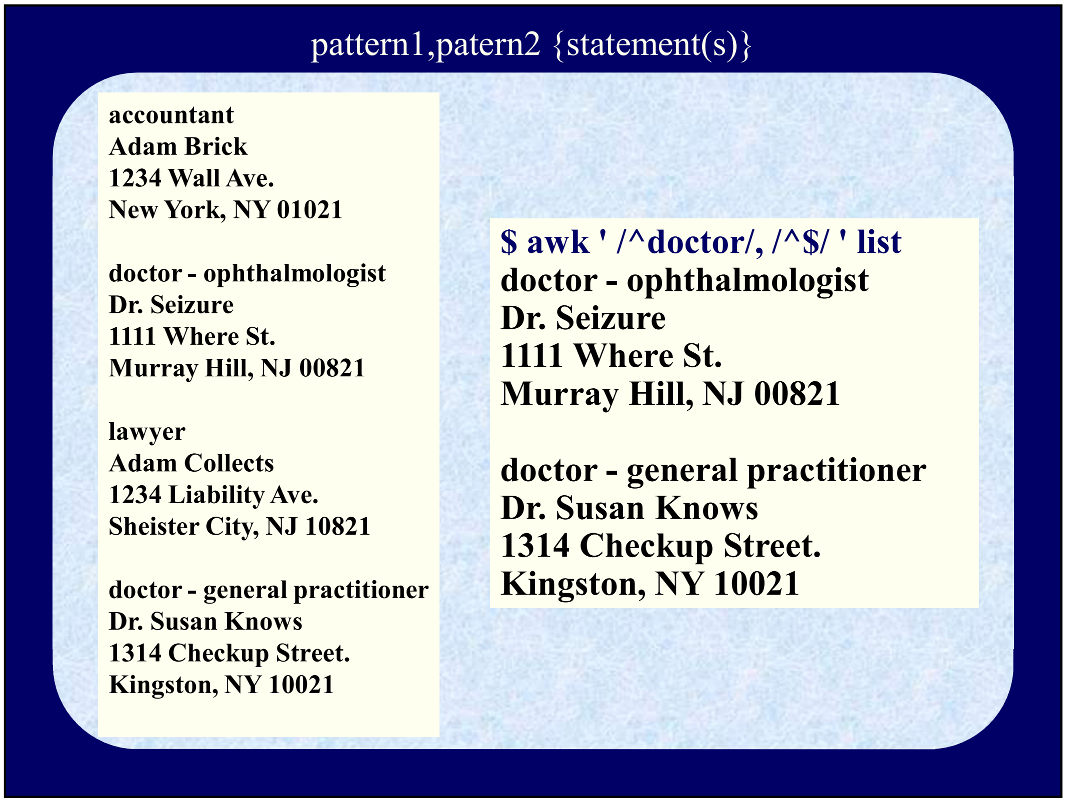
Before we explain the two examples, it's necessary to discuss what to do when the field separator is not white spaces (space(s) or tab(s)). The -F option with the colon(:) indicates our field separator is to be colon.

In the first regular expression, we want to match *ksh* at the end of each password record. The action statement says print positional filed 1 ($1), which is the username or login name field of the password record. This example is useful for finding those user accounts that have /bin/ksh as its startup program.

In the second example, the field separator is the UNIX pipe character(|). The pattern selection:

- Determines if <u>field six equals (==)</u> the character string "CA" and
- Determines if <u>field five is like (~)</u> "San Jose"

These comparisons are done with records read from the *phone.lst* file.



This example illustrates the pattern1, pattern2 matching capability of awk. A brief contents of the text data is on the left. The program executed and the output is listed to the right.

The program wants to start match a line that starts with *doctor* at the beginning of a line up to an include an immediate newline. Notice the two records that match this pattern.

An important note about ***awk*** regard pattern-action statements: When no action is specified, the entire record is printed. If no pattern statement, then all the records are printed.

Manipulating Text via AWK

```
== Print the user name for those accounts using ksh
==
$ awk -F: ' /ksh$/ { print $1 } ' /etc/passwd
stu01b
stu02b
stu03b
$
== Print the user name and shells ==
$ awk -F: ' /sh$/ { print $1 ": " $7 } ' /etc/passwd
root: /bin/bash
gstafford: /bin/bash
stu01a: /bin/bash
stu02a: /bin/bash
stu03a: /bin/bash
stu01b: /bin/ksh
stu02b: /bin/ksh
stu03b: /bin/ksh
stu01c: /bin/csh
stu02c: /bin/csh
stu03c: /bin/csh
```

The ability to pattern match text then format data output is an appealing capability of the ***awk*** command.

In the first example. Match those records that use passwd records ending in ksh. For those records that match, print the user name that is the first field.

The second example, match those records that end in **sh**. For the matching rccords that match, print username, colon (:) and a space the the login shell.

AWK – field separator trick

```
== cat the numbers file ==
nl -w2 numbers
 1      1.........5
 2      5........10
 3      10.......20
 4      100......200
== print the nuber of fields for each line ==
$ awk -F. ' { printf "%d: %2d\n", NR,NF } ' numbers
Record 1: 11
Record 2: 10
Record 3:  9
Record 4:  7
== Now use a formatted print (printf) ==
$ awk -F. '{printf "%3d - %3d\n",$1,$(NF)}' numbers
  1 -  5
  5 - 10
 10 - 20
100 - 200
```

This slide illustrates that using regular expressions are not always necessary.

-F. stipulates the period is the field separator.

The awk command, awk -F. ' { printf "Record %d: %2d\n", NR,NF } ' numbers yields:

- Record 1: 11
- Record 2: 10
- Record 3: 9
- Record 4: 7

NR is the record of the current record being processed. **NF** is the number of fields of the current record being processed.

The command: `awk -F. ' { printf %3d-%3dn, $1,$(NF) } ' numbers` yields:

```
  1 -   5
  5 -  10
 10 -  20
100 - 200
```

This awk command uses the printf command to print the first and last field that are integer numbers. The integer number will have a width of 3 characters. A string, ‘ – ‘, is printed between the digits.

AWK – RegEx 1 or more times

```
$ cat df-awkex
df -t xfs | awk ' NR > 1 {
                            gsub(/ +/, "|", $0)
                            print
                         } '
$

$ sh df-awkex
/dev/mapper/ol_vbox-root|3080192|166152|2914040|6%|/
/dev/mapper/ol_vbox-usr|10420224|5817888|4602336|56%|/usr
/dev/mapper/ol_vbox-tmp|2031616|47324|1984292|3%|/tmp
/dev/mapper/ol_vbox-addon|41877504|325020|41552484|1%|/addon
/dev/mapper/ol_vbox-local|46063616|396352|45667264|1%|/local
/dev/mapper/ol_vbox-home|5177344|322732|4854612|7%|/home
/dev/sda1|2031616|732384|1299232|37%|/boot
/dev/mapper/ol_vbox-opt|15663104|159108|15503996|2%|/opt
/dev/mapper/ol_vbox-var|5177344|1726580|3450764|34%|/var
```

In this script:

The **df -t xfs** command generates df output that is piped to the awk command. The gsub function has the following synopsis: `gsub(regex, sub, string).` If a string, the third parameter, is optional. If omitted, then **$0** is used.

The gsub function in the slide has a regular expression defined match a space that occurs 1 or more times and replace it with the vertical bar.

An important note about ***awk*** regard pattern-action statements: When no action is specified, the entire record is printed. If no pattern statement, then all the records are printed. In addition, the pattern **NR > 1** is a relational expression. The purpose of this e pression is to skip the of the **df** command. The gsub function will simply replace oe or spaces with a single vertical bar.

swapInt.c in-place edit

```
$ cat swapInt.c
// C Program to Swap Two Numbers using a Temporary Variable
//
#include <stdio.h>
int main() {
    int a = 5, b = 10, temp;
    printf("a = %d, b = %d\n", a, b);
    // Swapping values of a and b
    temp = a; a = b; b = temp;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}

$ cat swapsed
s: a and b: firstInt and secondInt:
s:([ "])a([ ,;]):\1firstInt\2:g
s:([ "])b([ ,;)]):\1secondInt\2:g
/^#include <stdio.h>/i\
// GMS 2025-04-22 - changes "a" to firstInt and "b" to secondInt\
//

$ sed -i$(date "+.%Y%m%d_%H%M%S") -rf swapsed swapInt.c

$ ls swapInt.c*
swapInt.c  swapInt.c.20250409_201339
$
```

The results of the sed edits is displayed below:

```
// C Program to Swap Two Numbers using firstInt Temporary Variable
//
// GMS 2025-04-22 - changes "a" to firstInt and "b" to secondInt
//
#include <stdio.h>
int main() {
    int firstInt = 5, secondInt = 10, temp;

    printf("firstInt = %d, secondInt = %d\n", firstInt, secondInt);
    // Swapping values of firstInt and secondInt
    temp = firstInt; firstInt = secondInt; secondInt = temp;

    printf("firstInt = %d, secondInt = %d\n", firstInt, secondInt);
    return 0;
}
```

The **swapsed** file has 4 edits:

- The first substitution will relace **a** and **b** variables with **firstInt** and **secondInt** in the comment line.
- Establish positional parameter with all characters preceding **a**; and all the characters after **a**.
- Establish positional parameter with all characters preceding **b**; and all the characters after **b**.
- Insert the comments regarding the code change above the include statement.

The in-place substitution **`-i$(date "+.%Y%m%d_%H%M%S")`** creates a backup file with the suffix **.YYYYMMDD_HHMMSS** appended to current filename.

Properly set a csv format file

```
$ head -5 EdAwards.csv
Toastmasters International -Education Awards,,,,
Education Program,Completion Date,Club,,
Engaging Humor 2 (EH2),"October 31, 2024",Holy City Toastmasters,,
Engaging Humor 1 (EH1),"October 24, 2024",Holy City Toastmasters,,
Strategic Relationships 2 (SR2),"November 09, 2023",Daybreak Club,,

$ awk ' { printf "%1d: %s\n", NR,$0 } ' EdAwardsFilter'
1: 3,$ s/,/|/1        # Replace the 1st comma occurrence
2: 3,$ s/,/|/2        # Replace the 2nd comma occurrence
3: 3,$ s/,/|/2        # Replace the 2nd comma occurrence
4: 3,$ s/"//g         # Globally remove the double quotes
5: 3,$ s/ *[(]/|/g    # Globally replace ' *(' with '|'
6: 3,$ s/[)]//g       # Globally replace ')' with '|'

$ sed -f EdAwardsFilter f1
Toastmasters International -Education Awards,,,,
Education Program,Completion Date,Club,,
Engaging Humor 2|EH2|October 31, 2024|Holy City Toastmasters|,
Engaging Humor 1|EH1|October 24, 2024|Holy City Toastmasters|,
Strategic Relationships 2|SR2|November 09, 2023|Daybreak Club|,
```

Another view of the sed command synopsis is:

Command Synopsis: **`sed 's/old/new/[flags]' [input-file]`**

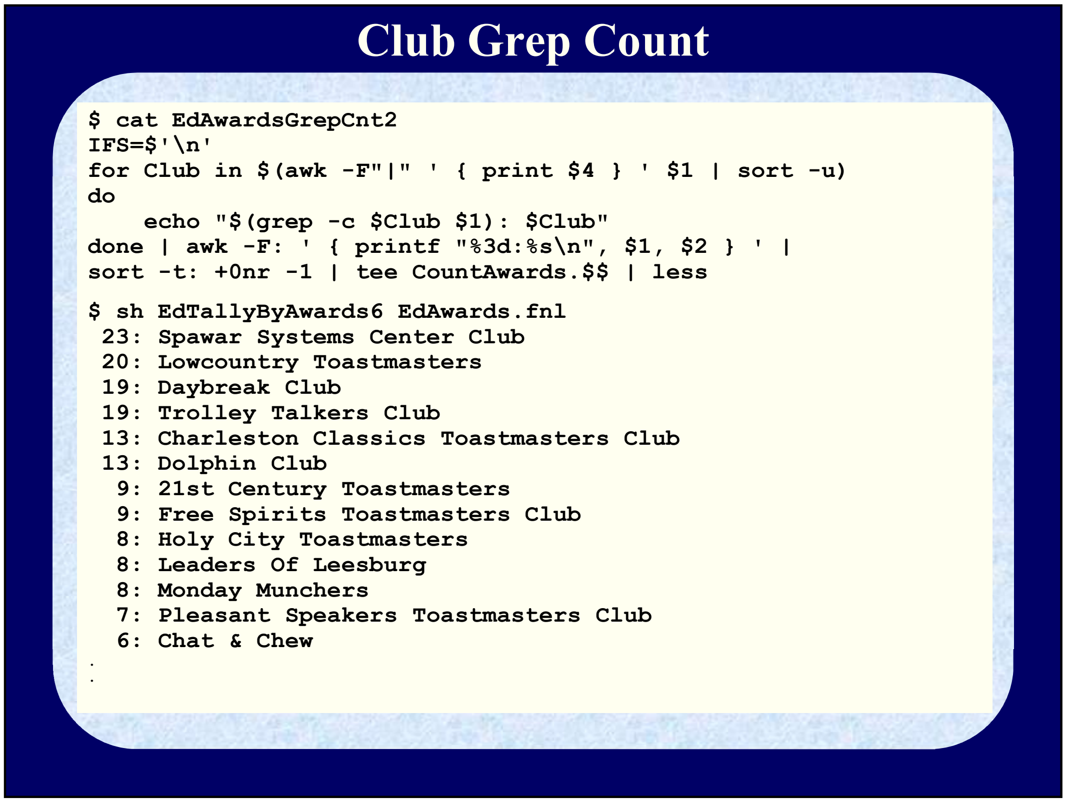
The sed flags can be any of the following:

```
g        Global substitution
1,2...   Substitute the nth occurrence
p        Print only the substituted line
w        Write only the substituted line to a file
I        Ignore case while searching
e        Substitute and execute in the command line
```

This slide illustrates replacing (or substituting) the n^th occurrence.

The substitution process has the following progression:

- Replace the 1st comma with a pipe(|). The 2nd comma is now the 1st; and the 3rd comma now becomes the 2nd.
- Replace the 2nd comma with a pipe(|). The 2nd comma is now the 1st; and the 3rd comma now becomes the 2nd.
- Replace the 2nd comma with a pipe(|). The 2nd comma is now the 1st; and the 3rd comma now becomes the 2nd.
- Globally remove the double quotes.
- Globally replace `'  *[(]'` with `'|'`
- Globally replace `'[)]'` with `'|'`



This slide illustrates the power of counting records with the grep command.

The **`IFS=$'\n'`** is critical and required if we're have fields with spaces.

In the four-loop Club will be assigned the value of a club name for each record passed. The output from the awk command is piped to sort to generate a unique list (avoiding duplicates).

The command, **`$(grep -c $Club $1)`**, will tabulate the number of records for the club.

The command, **`echo "$(grep -c $Club $1): $Club"`**, will print the club count, a colon (:) character, a space, and the name of the club. The output is then piped to the awk command. The awk command prints the club count, colon (:) character, a space, the club name. This output piped to sort to have a descending order of the count. Thus we output the highest to lowest of awards for the respective club.

AWK – Education Awards Tally (1)

```
$ cat EdTallyByAwards3
awk -F"|" '
 # Group Pathways Education Awards with string concatenation
 $2 ~ /[A-Z]{2}[12345]/ { $2 = "Level" substr($2,3,1) }

                  # Associative Array
                  { Award[$2]++ }
END {
           for (i in Award) printf ("%5d\t%s\n", Award[i], i);
    }  '  $1 | sort -nr
$ sh EdTallyByAwards3 EdAwards.fnl
   44   CC
   43   CL
   43   ALB
   20   ACB
   10   Level2
   10   Level1
    8   LDREXC
    6   Level3
    6   ACS
    2   Level4
    1   Level5
    1   DTM
    1   ALS
    1   ACG
```

Here is an example of using associative arrays, However, we will include pattern matches to modify the array index for pathway education codes. The pathway education awards have code that consists of 2 uppercase characters followed by the third character being a digit 1 thru 5.

The field separator is the vertical bar; as specified by –**F"|"**. The expression for the pattern match is a relational expression, **`$2 ~ /[A-Z]{2}[12345]/`**, states Field 2 must match an uppercase letter ranging a thru z and must be exactly two characters; followed by a digit ranging 1 thru 5.

For each pattern match, reassign the string **"Level“** and concatenate with the digit retrieved by the awk ***substr() function***. This results in generating new values for **$2** that include **Level1**, **Level2**, **Level3**, **Level4**, and **Level5**. There are no requirements to manipulate values of the other education codes.

The old education codes and the updated pathway education become the award indexes.

If the index does not exist, create it, and increment it by one. If the index does exist, increment it by one. After all the records are processed, precede to the end statement.

When all the records are processed, the **END** action will execute a for loop that prints each index with the format: print the record count and the award (index). The two fields are separated by a tab. The output stream is piped to the sort command to numerically (**-n**) and print the numbers from highest number to lowest.

AWK – Education Awards Tally (2)

```
$ EdTallyByAwards4
awk -F"|" '
  # Group Pathways Education Awards concateating with the use of
  # substr() and sprintf() functions
  $2 ~ /[A-Z]{2}[12345]/ { $2=sprintf("%s%s","Level",substr($2,3,1)) }

                 # Associative Array
                 { Award[$2]++ }
END {
           for (i in Award) printf ("%5d\t%s\n", Award[i], i);
    }  '  $1 | sort -nr

$ sh EdTallyByAwards4 EdAwards.fnl
   44   CC
   43   CL
   43   ALB
   20   ACB
   10   Level2
   10   Level1
    8   LDREXC
    6   Level3
    6   ACS
    2   Level4
    1   Level5
    1   DTM
    1   ALS
    1   ACG
```

Here is an example of using associative arrays, However, we will include pattern matches to modify the array index for pathway education. The pathway education awards have code that consists of 2 uppercase characters followed by the third character being a digit 1 thru 5.

The field separator is the vertical bar; as specified by –**F"|"**. The expression for the pattern match is a relational expression, **`$2 ~ /[A-Z]{2}[12345]/`**, states Field 2 must match an uppercase letter ranging a thru z and must be exactly two characters; followed by a digit ranging 1 thru 5.

For each pattern match, **$2** is modified by using the awk ***sprintf()*** function. The ***sprintf()*** format is **`"%s%s"`** combining the string "Level" and the ***substr()*** grabbing the digit (the third character in the pathway code. function results and concatenate with the digit retrieved The new values for **$2** that include **Level1**, **Level2**, **Level3**, **Level4**, and **Level5**. There are no requirements to manipulate values of the other education codes.

The old education codes and the updated pathway education become the award indexes.

If the index does not exist, create it, and increment it by one. If the index does exist, increment it by one. After all the records are processed, precede to the end statement.

When all the records are processed, the **END** action will execute a for loop that prints each index with the format: print the record count and the award (index). The two fields are separated by a tab. The output stream is piped to the sort command to numerically (**-n**) and print the numbers from highest number to lowest (**-r** reverse the results from high to low).

AWK – Education Awards Tally (3)

```
$ EdTallyByAwards5
awk -F"|" '
  # Group Pathways Education Awards using the substr() function
  # to tally the paths (first two characters)
  $2 ~ /[A-Z]{2}[12345]/ {
                             $2=sprintf("%s",substr($2,1,2))
                             # Associative Array
                             Award[$2]++
                         }
END {
           for (i in Award) printf ("%5d\t%s\n", Award[i], i);
    }  '  $1 | sort +0nr -1

$ sh EdTallyByAwards5 EdAwards.fnl
    8   EH
    4   MS
    3   DL
    3   PI
    3   PM
    2   EC
    2   LD
    2   SR
    2   VC
```

Here is an example of using associative arrays, However, we will include pattern matches to modify the array index for pathway education. The pathway education awards have code that consists of 2 uppercase characters followed by the third character being a digit 1 thru 5.

The field separator is the vertical bar; as specified by –**F"|"**. The expression for the pattern match is a relational expression, **`$2 ~ /[A-Z]{2}[12345]/`**, states Field 2 must match an uppercase letter ranging a thru z and must be exactly two characters; followed by a digit ranging 1 thru 5.

The **$2** is reassigned the first two characters of pathway code when the pattern is matched, otherwise, the code is from the old program.

The old education codes and the updated pathway education become the award indexes.

If the index does not exist, create it, and increment it by one. If the index does exist, increment it by one. After all the records are processed, precede to the end statement.

When all the records are processed, the **END** action will execute a for loop that prints each index with the format: print the record count and the award (index). The two fields are separated by a tab. The output stream is piped to the sort command to numerically (**-n**) and print the numbers from highest number to lowest (**-r** reverse the results from high to low).

AWK – Education Awards Tally (4)

```
$ cat EdTallyByAwards5
awk -F"|" '
  # Group Pathways Education Awards using the substr() and
length()
  # functions to extract the Full Path Description
  $2 ~ /[A-Z]{2}[12345]/ {
                             number=length($1)
                             string=substr($1,1,number-2)
                             # Associative Array
                             Award[string]++
                         }
END {
           for (i in Award) printf ("%5d\t%s\n", Award[i], i)
    }  '  $1 | sort +0nr -1

$ sh EdTallyByAwards6 EdAwards.fnl
    8   Engaging Humor
    4   Motivational Strategies
    3   Dynamic Leadership
    3   Persuasive Influence
    3   Presentation Mastery
    2   Effective Coaching
    2   Leadership Development
    2   Strategic Relationships
    2   Visionary Communication
```

Here is an example of using associative arrays, where the index is the pathway name excluding the level number.

The field separator is the vertical bar; as specified by –**F"|"**. The expression for the pattern match is a relational expression, **`$2 ~ /[A-Z]{2}[12345]/`**, matching an uppercase letter ranging a thru z and must be exactly two characters; followed by a digit ranging 1 thru 5. This example extracts the pathway name for the index.

First awk function length() calculates the length **$1**. Then the string is assigned the string extracted from the awk function ***substr()***. The calculation **number-2** strips off the space and digit.

If the index does not exist, create it, and increment it by one. If the index does exist, increment it by one. After all the records are processed, precede to the end statement.

When all the records are processed, the **END** action will execute a for loop that prints each index with the format: print the record count and the award (index). The two fields are separated by a tab. The output stream is piped to the sort command to numerically (**-n**) and print the numbers from highest number to lowest (**-r** reverse the results from high to low).

Filter Yum Update Log

```
$ vi yum-logfilter
s:^[::g           # Remove the escape character globally
s:^M::g           # Remove <ctrl><M> globally
s:^G::g           # Remove <ctrl><G> globally
s:^H::g           # Remove <ctrl><H> globally
s:\[1m::g         # Remove '[1m' globally
s:\[m::g          # Remove '[m' globally
s:[(]B::g         # Remove '[(]B' globally
s:\[32m::g        # Remove '[32m' globally
s:^\].+(#):\1:    # Discard all characters except the root prompt
s:..20041::g      # Remove any 2 printable character + '20041' globally
/exit/d           # Delete the line with exit
~
$ cat yum-massage
unset LS_COLOR
export TERM=xterm-mono
sed -rf yum-logfilter $1 |
awk '/ETA/ { printf"%15s %-50s %6s %s\n", $1,$(NF-6),$(NF-2),$(NF-1) ; next}
     /Running scriptlet:/ {printf "%s %s %s %10s\n",$1,$2,$3,$4 ; next}
     /Upgrading:$/ {print $0 ; next }
     /Upgrading/ {printf "%s: %-50s\t%10s\n", $1,$(NF-1),$(NF); next}
     /Cleanup/ { printf "%s: %-50s\t%10s\n", $1,$3,$(NF)  ; next}
     /^Installing:$/ { print $0 ; next}
     /Installing / {printf "%10s: %-50s\t%10s\n", $1,$(NF-1),$(NF) ; next}
     /Verifying/ {printf "%9s: %-50s\t%10s\n", $1,$3,$(NF) ; next}
     /Preparing/ {printf "%9s: %-70s\n", $1,$(NF) ; next}
     /#/ {printf "%s\n", substr($0,index($0,"#")) ; next }
     /$/ {printf "%s\n", substr($0,index($0,"$")) ; next }
     /x86_64 +$/ { print substr($0,1,75) ; next }
     /Last metadata/ {printf "%s\n", substr($0,index($0,"Last metadata")) ; next }
     { print } '
$
```

This is an example taking advantage of sed and awk to modify a log with embedded control characters. The sed command, referencing the yum-logfilter file, is the best means for stripping out escape and control characters; along character strings that are distracters.

It is important note actions take for satisfied pattern matches. For each pattern match, the printf directive prints revised record based upon format specified. The **next** directive states to go to the next record. That means if we had a pattern match, the solo action **{print}** is ignored. Those records that did not match a pattern will have that record unchanged and printed.

The pattern matches in sequential order are:

- If the line has the character string `'ETA'`, print he first field, 6 fields to the left of the last field, 2 fields to the left of the last field, and the last field.
- Print the 1st, 2nd, 3rd, and 4th fields.
- Print the entire when matching `'Upgrading$'` is matched.
- If the pattern match is `'Upgrading '`, print the first field, 1 field left of the last field, the last field.
- If the pattern is `'Verifying'`, print the 1st field, the 3rd field, and the last field.
- If the pattern is `'Preparing'`, print the 1st field and the last field
- If the pattern is matching the root prompt (#), print the substring starting at the root prompt. Since no length is provided , the remainder of the character string is printed.
- If the pattern is matching the user prompt ($), print the substring starting at the user prompt. Since no length is provided , the remainder of the character string is printed

Again, those records that did not satisfy a pattern match, the entire record is printed.

Comma to Newline Conversion

```
== Contents of DisneyEmails ==
$ cat DisneyEmails
pango@dog.com,goofy@dog.com,daffy@duck.com,kirk@startrek.net,judd@defense.org

== Illustrate replacing "," with newlines "\n\" ==
$ sed s/,/\n/g DisneyEmails
pango@dog.com
goofy@dog.com
daffy@duck.com
kirk@startrek.net
judd@defense.org

$ cat DisneyEmails
pango@dog.com,goofy@dog.com,daffy@duck.com,kirk@startrek.net,judd@defense.org

== Perform in-place substitution; then display contents ==
$ sed -i s/,/\n/g DisneyEmails ; cat DisneyEmails
pango@dog.com
goofy@dog.com
daffy@duck.com
kirk@startrek.net
judd@defense.org
$
```

There are times when a comma separated list needs to have the comma replaced with the newline.

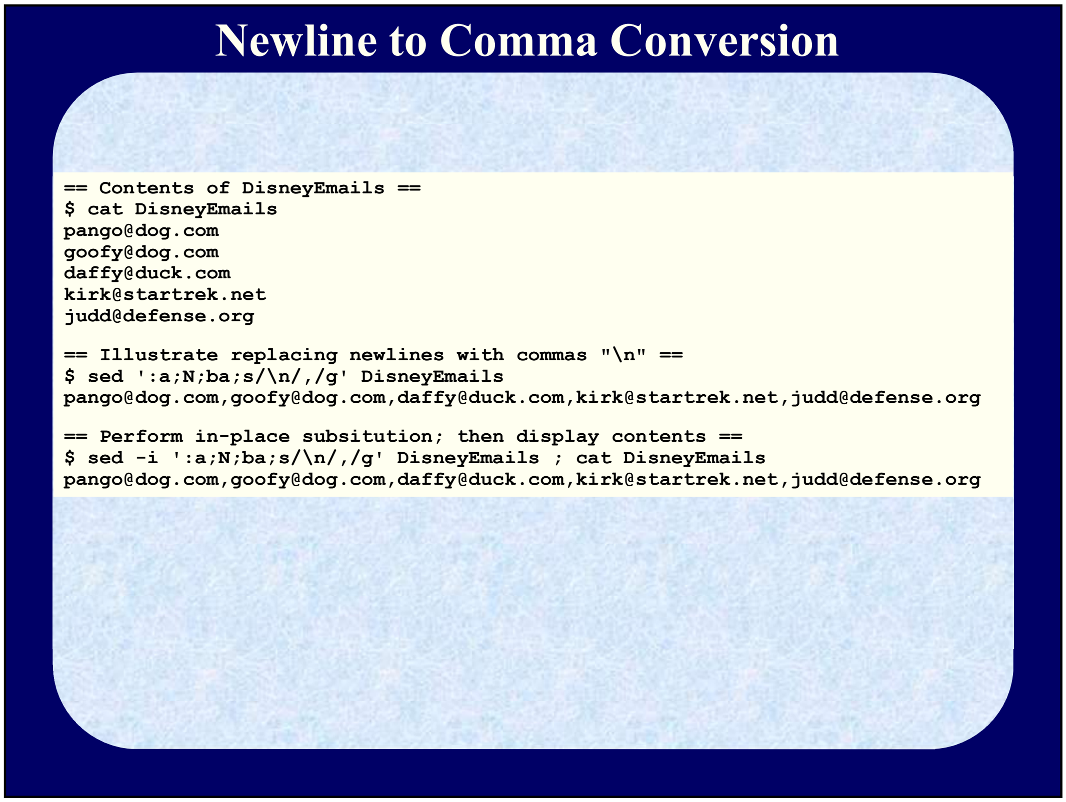
Notice the comma separated email addresses. We need to add additional email addresses. Therefore, it is a sound practice to:

1. Replace the commas with newlines.
2. Insert or remove email addresses as needed for email updates.
3. Generate a unique sorted list to avoid duplicates
4. Proceed to replace the newlines with commas.

The above slide first displays the contents of Disney Emails. The first sed command performs the pattern match and replacement, replacing commas with newlines globally and outputs the results.

The second cat confirms the file was not updated.

The next sed of command performs an in-place substitution without creating a back-up file (no suffix specified), replacing commas with newlines globally; followed by the third cat command verifying the changes were made and saved in the original file DisneyEmails.



The above slide first displays the contents of Disney Emails that have each email on a separate line. The first sed command performs the pattern match and replacement, replacing newlines with commas and outputs the results.

The second cat confirms the file was not updated.

The next sed of command performs an in-place substitution without creating a back-up file (no suffix specified), replacing the newlines with commas; followed by the third cat command verifying the changes were made and saved in the original file DisneyEmails.

The :pattern:replacement:, being **`':a;N;ba;s/\n/,/g'`**, is defined as follows:

1. sed starts by reading the first line excluding the newline into the pattern space.
2. Create a label via :a.
3. Append a newline and next line to the pattern space via N.
4. If we are before the last line, branch to the created label $!ba ($! means not to do it on the last line. This is necessary to avoid executing N again, which would terminate the script if there is no more input!).
5. Finally the substitution replaces every newline with a space on the pattern space (which is the whole file).

The single-line results can be copied and pasted appropriate email (**To:** or **CC:**).

Reformat Numbers Inserting Commas

```
$ cat NUMBERS
49
25
104
300
41261
27525
21654680
48672067

$ cat awk readableNumbers2
awk '{while (match($0, /(^[^.0123456789])[0123456789]{4,}/))
      {
        $0 = substr($0, 1, RSTART+RLENGTH-4) "," substr($0, RSTART+RLENGTH-3)
      }
        printf "%12s\n",$0
     }' NUMBERS

$ sh readableNumbers2
          49
          25
         104
         300
      41,261
      27,525
  21,654,680
  48,672,067
```

This slide illustrates a unique approach to reformatting numbers to insert appropriate commas.

modDates script

```
$ nl -w 2 modDates
 1      awk -F "|" '
 2      BEGIN { OFS="|" }
 3      /Jan/ { sub(/Jan/,"01",$6) }
 4      /Feb/ { sub(/Feb/,02",$6) }
 5      /Mar/ { sub(/Mar/,"03",$6) }
 6      /Apr/ { sub(/Apr/,"04",$6) }
 7      /May/ { sub(/May/,"05",$6) }
 8      /Jun/ { sub(/Jun/,"06",$6) }
 9      /Jul/ { sub(/Jul/,"07",$6) }
10      /Aug/ { sub(/Aug/,"08",$6) }
11      /Sep/ { sub(/Sep/,"09",$6) }
12      /Oct/ { sub(/Oct/,"10",$6) }
13      /Nov/ { sub(/Nov/,"11",$6) }
14      /Dec/ { sub(/Dec/,"12",$6) }
15            {
16                # Old $6: Mon 10 Feb 2025 05:25:17 PM EST
17                # New $6: Mon 10 02 2025 05:25:17 PM EST
18                DateString=substr($6,5,10)
19                Day=substr(DateString,1,2)
20                Month=substr(DateString,4,2)
21                Year=substr(DateString,7,4)
22                NewString=sprintf("%s-%s-%s", Year, Month, Day)
23                # No slashes (/), the variable DateString is not treated
24                # as a RegEx DateString is replaced with the value in
25                # NewString for field $6 (sixth positional field)
26                sub(DateString,NewString,$6)
27            }
28            { print }
29      ' $1
```

This slide demonstrates modifying 3 character; followed by a space, followed by a day of month, followed by a comma, followed by a four-digit year.

Lines 3-14 has the pattern substitute match of the 3 character month with its corresponding 2-digit month.

Next is the block statement, with no pattern match RegEx. At present the date format is currently 'MM DD YYYY'. This character string in **$6** is has the starting point at position 5 10 characters long. Thus `substr($6,5,10)` grabs that string and assigns it to the variable **DateString**. Now reformat the 'MM DD YYYY' to 'YYYY-MM-DD'; taking advantage of the sprintf() function and assign that modified value to **NewString**. Next substitute **DateString** with the newly created value of **NewString**; and update field **$6**.

Now that all the substitutions are complete; proceed to the next block statement, with no pattern match specified that prints the entire record.

Execution Results

```
$ cat RpmTable
sed|x86_64|9.el9|4.8|Unspecified|Mon 10 Feb 2025 05:25:26 PM EST
iw|x86_64|1.el9|6.9|Unspecified|Mon 10 Feb 2025 05:25:38 PM EST
inih|x86_64|6.el9|49|Unspecified|Mon 10 Feb 2025 05:25:43 PM EST
jq|x86_64|17.el9|1.6|Unspecified|Mon 10 Feb 2025 05:25:43 PM EST
jose|x86_64|1.el9|14|Unspecified|Mon 10 Feb 2025 05:25:55 PM EST
gom|x86_64|6.el9|0.4|Unspecified|Mon 10 Feb 2025 05:26:03 PM EST
zsh|x86_64|9.el9|5.8|Unspecified|Sun 16 Feb 2025 11:21:32 PM EST
.
.
.
$ sh modDates RpmTable
sed|x86_64|9.el9|4.8|Unspecified|Mon 2025-02-10 05:25:26 PM EST
iw|x86_64|1.el9|6.9|Unspecified|Mon 2025-02-10 05:25:38 PM EST
inih|x86_64|6.el9|49|Unspecified|Mon 2025-02-10 05:25:43 PM EST
jq|x86_64|17.el9|1.6|Unspecified|Mon 2025-02-10 05:25:43 PM EST
jose|x86_64|1.el9|14|Unspecified|Mon 2025-02-10 05:25:55 PM EST
gom|x86_64|6.el9|0.4|Unspecified|Mon 2025-02-10 05:26:03 PM EST
zsh|x86_64|9.el9|5.8|Unspecified|Sun 2025-02-16 11:21:32 PM EST
.
.
.
```

This slide illustrates converting the 'Mon Day, YYYY' to 'YYYY-MM-DD' format to after executing the script **modDates**.

Execution Results

```
$ nl -w2 rpmUpdDate
 1      for Date in $(cut -f6 -d"|" RpmTable2 | cut -c5-15 | sort -u)
 2      do
 3         count=$(grep -c "${Date}" RpmTable2)
 4         printf "%5d: %s\n" "${count}" "${Date}"
 5      done | sort -nr

$ sh rpmUpdDate
 1270: 2025-02-10
   82: 2025-03-27
   25: 2025-02-11
   20: 2025-04-03
    2: 2025-02-12
    1: 2025-04-05
    1: 2025-03-10
    1: 2025-02-16
```

One may ask why did we change the date format. Simply put the 'YYYY-MM-DD' format is a much easier key for tallying updates on a specific date.

Line 1 is the beginning of the for-loop to create a unique list of dates that will be assigned to the shell variable **Date**. Line 3 executes **grep –c** to calculate the number of matches. Line 4 will print the record count of the **$Date** match; a followed by a colon; followed by a space; followed by the date string. This output is piped to the sort command; sorting numerically in reverse order (descending order).

Notice the output from executing **sh rpmUpdDate**.

genExample

```
1   #! /usr/bin/awk
2
3   BEGIN { print "clear" }
4
5   /^prep:/ { command=substr($0,7) ; print command }
6
7   /^:/ { command=substr($0,3)
8          print "echo -n","\"" "\$",command "\""
9          print "read pause"
10         print command
11         printf "%s\n", "printf \"$ \""
12         print "read pause"
13       }
14  /^clear/  { print "clear" }
15
16  /^#/  { print "echo", "\"" "==", substr($0,3), "==" "\"" }
17  /^>:/ { print "echo", "'" ">>>", substr($0,3), "<<<" "'" }
```

This **genExample** awk script is a very powerful script to generate interctive demos to teach specific topics in the Unix or Linux environments.

On Line 3 the **BEGIN** action will print the clear command to the first command executed for the demo script.

On Line 5 the **pattern match** is **'**prep:**'** will print the preparatory command before executing any following commands.

On Lines 7 thru 13 the **pattern match** is **'**^:**'** stating to match a colon at the beginning of the record. Satisfied matches results in:

1. Line 7: Use the **substr()** to extract characters starting at position 3 up to the end of record and assign it to the variable ***command***.
2. Line 8: Print the string **'echo –n'**; followed by printing the command, embedded in double quoted, to simulate the command entered on the command line. Notice the **–n** option is used to disable outputting a newline.
3. Line 9: Have a read pause; requiring one to press the <enter> key to execute the command.
4. Line 10: Print the command that will be executed after pressing the <enter> key.
5. Line 11: Print the user prompt. Notice there s no '\n' which will preempt a newline.
6. Line 12: Print a 'read pause' to pause the script.

On Line 14, if the record has the string **'clear'**, print the string **'clear'** that will clear the screen in the script generated.

Line 16 & 17 will print comments for documentation purposes.

GenDemo

```
1  prep: cd /home/gstafford/Training/RegEx/tmp
2  prep: rm *.zip *.gz 2> /dev/null
3  prep: cp .commands commands
4  >: Print current working directory and
5  >: contents of current directory
6  : pwd; ls
7  # Use gunzip to compress the file
8  : gzip commands
9  # List the contents of $(pwd)
10 : ls
11 >: Contrasting the difference between
12 >: executing grep and zgrep commands
13 : grep '^[fg]grep$' commands.gz
14 : zgrep '^[fg]grep$' commands.gz
15 >: zgrep need the '-E' option to
16 >: support regular expressions
17 : zgrep 'fgrep$|^z(more|less|new)' commands.gz
18 : zgrep -E 'fgrep$|^z(more|less|new)' commands.gz
19 prep: cd /home/gstafford/Training/RegEx/UniForum
```

In the GenDemo file:

- Lines 1 thru 3 will change directory to **/home/gstafford/Training/RegEx/tmp**.
- Lines 4 and 5 are comments to be printed
- Line 6 has the command line to be demoed; **pwd** and **ls**.
- Line 7 is a comment.
- Line 8 demos the gzip command compressing the command file
- Line 9 is a comment stating the contents of the directory will be listed
- Line 10 is the execution to list the file(s) in the current directory
- Lines 11 and 12 are comments that the grep and zgrep command will be contrasted searching for a pattern match in a gzip'd file.
- Line 13 and 14 is the demo of executing the grep and zgrep commands respectively.
- Lines 15 and 16 are comments contrasting the requirement to have the **`'-E'`** option when the need exists to interpret extended regular expressions.
- Lines 17 and 18 is to demo the two commands.
- Line 19 is the forces the return to the **/home/gstafford/Training/RegEx/UniForum** directory.

GenDemo.sh

```
 1  clear
 2  cd /home/gstafford/Training/RegEx/tmp
 3  rm *.zip *.gz 2> /dev/null
 4  cp .commands commands
 5  echo '>>>  Print current working directory and <<<'
 6  echo '>>>  contents of current directory <<<'
 7  echo -n "$ pwd; ls"
 8  read pause
 9  pwd; ls
10  printf "$ "
11  read pause
12  echo "== Use gunzip to compress the file =="
13  echo -n "$ gzip commands"
14  read pause
15  gzip commands
16  printf "$ "
17  read pause
⋮
```

This is a portion of the **GenDemo.sh** script:

- Lines 1 – 4 executes the commands clear the screen, cd directory to the **tmp** subdirectory under **/home/gstafford/Training/RegEx**; remove the existence of any files with a **'.zip'** or **'.gz'** suffixes; copy the hidden **.command** to **commands**.
- Lines 5 thru 6 are self-explanatory comments to be printed
- Line 7 echoes the two commands to be demoed (**pwd** and **ls**).
- Line 8 pauses the script.
- Line 9 executes the two commands **pwd** and **ls**.
- Line 10 prints the user prompt '$'.
- Line 11 pause the script.
- Line 12 print the comment.
- Line 13 echoes the the command gzip command to be demoed.
- Line 13 pause the script.
- Line 14 pause the script.
- Line 15 demo the execution of the gzip command.
- Line 16 print the user prompt '$'.
- Line 17 pause the script.