

05 - Expansions and Regular Expressions

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

Stephen McDowell

February 5th, 2016

Cornell University

Table of contents

1. Shell Expansion
2. Sets, Regular Expressions, and Usage
3. More Git

- The **assignments** repository on GitHub.

Some Logistics

- The `assignments` repository on GitHub.
- Course pacing...

Some Logistics

- The **assignments** repository on GitHub.
- Course pacing...
- HW1 tonight.

Shell Expansion

There are various special characters you have access too in your shell to expand phrases to match patterns, such as `*`, `?`, `^`, `{`, `}`, `[`, `]`.

There are various special characters you have access too in your shell to expand phrases to match patterns, such as `*`, `?`, `^`, `{`, `}`, `[`, `]`.

- Any string.

There are various special characters you have access too in your shell to expand phrases to match patterns, such as `*`, `?`, `^`, `{`, `}`, `[`, `]`.

- Any string.
- A single character.

There are various special characters you have access too in your shell to expand phrases to match patterns, such as `*`, `?`, `^`, `{`, `}`, `[`, `]`.

- Any string.
- A single character.
- A phrase.

There are various special characters you have access too in your shell to expand phrases to match patterns, such as `*`, `?`, `^`, `{`, `}`, `[`, `]`.

- Any string.
- A single character.
- A phrase.
- A restricted set of characters.

Shell Expansion: Example

- The `*` matches any string, including the null string (e.g. 0 or more characters).

Input	Matched	Not Matched
<code>Lec*</code>	<code>Lecture1.pdf Lec.avi</code>	<code>AlecBaldwin/</code>
<code>L*ure*</code>	<code>Lecture2.pdf Lectures/</code>	<code>sure.txt</code>
<code>*.tex</code>	<code>Lecture1.tex Presentation.tex</code>	<code>tex/</code>

Shell Expansion: Example

- The `?` matches a single character.

Input	Matched	Not Matched
<code>Lec?.pdf</code>	<code>Lec1.pdf Lec2.pdf</code>	<code>Lec11.pdf</code>
<code>ca?</code>	<code>cat can cap</code>	<code>ca cake</code>

Shell Expansion: Example

- Brace enumerations `[...]` match any character inside the square brackets.
 - Use a dash to indicate a range of characters.
 - Can put commas between characters / ranges.

Input	Matched	Not Matched
<code>[SL]ec*</code>	Lecture Section	Vector.tex
<code>Day[1-3]</code>	Day1 Day2 Day3	Day5
<code>[A-Z,a-z][0-9].mp3</code>	A9.mp3 z4.mp3	Bz2.mp3 9a.mp3

Shell Expansion: Example

- The `^` character represents *not*.
 - E.g. `[^...]` matches any character *not* inside the square brackets.

Input	Matched	Not Matched
<code>[^A-P]ec*</code>	<code>Section.pdf</code>	<code>Lecture.pdf</code>
<code>[^A-Za-z]*</code>	<code>9Days.avi</code>	<code>vacation.jpg</code>

Shell Expansion: Example

- **Brace Expansion:** `{...,...}` matches any phrase inside the comma-separated braces.
- Supports ranges as well!
- Brace expansion needs at least two options to choose from.

Input	Matched
<code>{Hello,Goodbye}\World</code>	<code>Hello World Goodbye World</code>
<code>{Hi,Bye,Cruel}\World</code>	<code>Hi World By World Cruel World</code>
<code>{a..t}</code>	Expands to the range <code>a ... t</code>
<code>{1..99}</code>	Expands to the range <code>1 ... 99</code>

Note: NO SPACES. We haven't covered loops yet...but this is most useful when you want to do something like

```
• for x in 1..99; do echo $x; done
```


Combining Them

Of course, you can combine all of these!

Input	Matched	Not Matched
<code>*h[0-9]*</code>	<code>h3 h3llo.txt</code>	<code>hello.txt</code>
<code>[bf][ao][row].mp?</code>	<code>bar.mp3 foo.mpg</code>	<code>foo.mpeg</code>

Interpreting Special Characters

The special characters are

\$ * < > & ? { } []

Interpreting Special Characters

The special characters are

`$ * < > & ? { } []`

- The shell interprets them in a special way unless we escape them (`\$`), or place them in quotes (`"$"`).

Interpreting Special Characters

The special characters are

`$ * < > & ? { } []`

- The shell interprets them in a special way unless we escape them (`\$`), or place them in quotes ("`$`").
- When we first invoke a command, the shell first translates it from a string of characters to a Unix command that it understands.

Interpreting Special Characters

The special characters are

`$ * < > & ? { } []`

- The shell interprets them in a special way unless we escape them (`\$`), or place them in quotes ("`$`").
- When we first invoke a command, the shell first translates it from a string of characters to a Unix command that it understands.
- A shell's ability to interpret and expand commands is one of the powers of shell scripting.

Interpreting Special Characters

The special characters are

`$ * < > & ? { } []`

- The shell interprets them in a special way unless we escape them (`\$`), or place them in quotes ("`$`").
- When we first invoke a command, the shell first translates it from a string of characters to a Unix command that it understands.
- A shell's ability to interpret and expand commands is one of the powers of shell scripting.
- These will become your friends, and we'll see them again...

Sets, Regular Expressions, and Usage

tr Revisited

The **tr** does not understand regular expressions per se (and really for the task it is designed for they don't make sense), but it does understand ranges and **POSIX** character sets:

tr Revisited

The **tr** does not understand regular expressions per se (and really for the task it is designed for they don't make sense), but it **does** understand ranges and **POSIX** character sets:

Useful Sets

<code>[:alnum:]</code>	alphanumeric characters
<code>[:alpha:]</code>	alphabetic characters
<code>[:digit:]</code>	digits
<code>[:punct:]</code>	punctuation characters
<code>[:lower:]</code>	lowercase letters
<code>[:upper:]</code>	uppercase letters
<code>[:space:]</code>	whitespace characters

If you Leave this Class with Anything...

Quite possibly the two most common things anybody uses in a terminal:

If you Leave this Class with Anything...

Quite possibly the two most common things anybody uses in a terminal:

- **find**: searching for files / directories by name or attributes.

If you Leave this Class with Anything...

Quite possibly the two most common things anybody uses in a terminal:

- **find**: searching for files / directories by name or attributes.
- **grep**: search contents of files.

If you Leave this Class with Anything...

Quite possibly the two most common things anybody uses in a terminal:

- **find**: searching for files / directories by name or attributes.
- **grep**: search contents of files.
- Used in conjunction with expansions, sets, and regular expressions.

find

`find [where to look] criteria [what to do]`

- Used to locate files or directories.
- Search any set of directories for files that match a criteria.
- Search by name, owner, group, type, permissions, last modification date, and *more*.
- Search is recursive (will search all subdirectories too).
 - Sometimes you may need to limit the depth.

Some Find Options

- **name**: name of file or directory to look for.
- **maxdepth num**: search at most **num** levels of directories.
- **mindepth num**: search at least **num** levels of directories.
- **amin n**: file last access was **n** minutes ago.
- **atime n**: file last access was **n** days ago.
- **group name**: file belongs to group **name**.
- **path pattern**: file name matches shell pattern **pattern**.
- **perm mode**: file permission bits are set to **mode**.

Of course...a lot more in **man find**.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.
 - The variable name is **{}**.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.
 - The variable name is **{}**.
 - You have to end the command with either a

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.
 - The variable name is **{}**.
 - You have to end the command with either a
 - **;** to execute the command on each individual result as you *find* them.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.
 - The variable name is **{}**.
 - You have to end the command with either a
 - **;** to execute the command on each individual result as you *find* them.
 - **+** to execute on all results once *at the end*.

Some Details

- This command is extremely powerful...but can be a little verbose. That's normal.
- Normally all modifiers for **find** are evaluated in conjunction (a.k.a AND). You can condition your arguments with an OR by passing the **-o** flag *for each* modifier you want to be an OR.
- You can execute a command on found files / directories by using the **-exec** modifier, and **find** will execute the command for you.
 - The variable name is **{}**.
 - You have to end the command with either a
 - **;** to execute the command on each individual result as you *find* them.
 - **+** to execute on all results once *at the end*.
 - **Note:** You have usually to escape them, e.g. **\;** and **\+**

Some Examples

Find all files accessed at most 10 minutes ago

```
find . -amin -10
```

Find all files accessed at least 10 minutes ago

```
find . -amin +10
```

Display all the contents of files accessed in the last 10 minutes

```
find . -amin -10 -exec cat +
```

Accidentally did **git add** on a Mac and ended up with
.DS_Store Everywhere?

```
find . -name .DS_Store -exec git rm -rf
```


Time for the Magic

Globally Search a Regular Expression and Print

`grep <pattern> [input]`

- Searches **input** for all lines containing **pattern**.
- As easy as specifying a **string** you need to find in a **file**.
- Or it can be much more.
- Common:

```
<some_command> | grep <thing you need to find>
```

Understanding how to use **grep** is really going to save you a lot of time in the future!

Grep Options

- **i**: ignores case.
- **A 20 -B 10**: print 10 lines before, and 20 lines after each match.
- **v**: inverts the match.
- **o**: shows only the matched substring.
- **n**: displays the line number.
- **H**: print the filename.
- **--exclude <glob>**: ignore **glob** e.g. **--exclude *.o**
- **r**: recursive, search subdirectories too.
 - **Note**: you're Unix version may differentiate between **-r** and **-R**, check the **man** page. We'll cover what that means soon.

- **grep**, like many programs, takes in a regular expression as its **input**. Pattern matching with regular expressions is more sophisticated than shell expansions, and also uses different syntax.

Regular Expressions

- **grep**, like many programs, takes in a **regular expression** as its **input**. Pattern matching with regular expressions is more sophisticated than shell expansions, and also uses different syntax.
- More precisely, a regular expression is a set of strings - these strings match the specified expression.

Regular Expressions

- **grep**, like many programs, takes in a **regular expression** as its **input**. Pattern matching with regular expressions is more sophisticated than shell expansions, and also uses different syntax.
- More precisely, a regular expression is a set of strings - these strings **match** the specified expression.
- When we use regular expressions, it is (usually) best to enclose them in quotes to stop the shell from expanding it before passing it to **grep** / other tools.

Regular Expression Notes

Some **regex** patterns perform the same tasks as the wildcards we learned:

Single Characters

Wild card: `?` Regex: `.`

- Matches any single character.

Wild card: `[a-z]` Regex: `[a-z]`

- Matches one of the indicated characters.
- Don't separate multiple characters with commas in the **regex** form (e.g. `[a,b,q-v]` becomes `[abq-v]`).

A Simple Example

`grep 't.a'` - prints lines like **tea**, **taa**, and **steap**.

A Note on Ranges

- Like shell wildcards, regexs are case-sensitive. What if you want to match any letter, regardless of case?

A Note on Ranges

- Like shell wildcards, regexs are case-sensitive. What if you want to match any letter, regardless of case?
 - If you take a look at the ASCII codes I keep mentioning in [2], you will see that the lower case letters come after the upper case letters.

A Note on Ranges

- Like shell wildcards, regexs are case-sensitive. What if you want to match any letter, regardless of case?
 - If you take a look at the ASCII codes I keep mentioning in [2], you will see that the lower case letters come after the upper case letters.
 - You should be careful about trying to do something like `[a-Z]`.

A Note on Ranges

- Like shell wildcards, regexs are case-sensitive. What if you want to match any letter, regardless of case?
 - If you take a look at the ASCII codes I keep mentioning in [2], you will see that the lower case letters come after the upper case letters.
 - You should be careful about trying to do something like `[a-Z]`.
 - Instead, just do `[a-zA-Z]`.

A Note on Ranges

- Like shell wildcards, regexs are case-sensitive. What if you want to match any letter, regardless of case?
 - If you take a look at the ASCII codes I keep mentioning in [2], you will see that the lower case letters come after the upper case letters.
 - You should be careful about trying to do something like `[a-Z]`.
 - Instead, just do `[a-zA-Z]`.
 - Note: some programs very well could accept the range `[a-Z]` correctly.

- **grep** accepts the **POSIX** sets we learned earlier!

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.
- ***** matches 0 or more occurrences of the expression.

Workarounds

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.
- `*` matches 0 or more occurrences of the expression.
- `\?` matches 0 or 1 occurrences of the expression.

Workarounds

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.
- `*` matches 0 or more occurrences of the expression.
- `\?` matches 0 or 1 occurrences of the expression.
- `\+` matches 1 or more occurrences of the expression.

Workarounds

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.
- `*` matches 0 or more occurrences of the expression.
- `\?` matches 0 or 1 occurrences of the expression.
- `\+` matches 1 or more occurrences of the expression.
- Remember that you can flip the expressions with the not signal: `^`

Workarounds

- **grep** accepts the **POSIX** sets we learned earlier!
 - e.g. `ls | grep [[:digit:]]` gives all files with numbers in the filename.
- `*` matches 0 or more occurrences of the expression.
- `\?` matches 0 or 1 occurrences of the expression.
- `\+` matches 1 or more occurrences of the expression.
- Remember that you can flip the expressions with the not signal: `^`
- The `$` can be used to match the end of the line.

To be continued...

There's a lot more going on here. We'll come back to it soon!

More Git

Syncing a Fork...

...again!

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.

Previous cornell cs 2043 course slides.

[2] A. Table.

Ascii character codes and html, octal, hex, and decimal chart conversion.

<http://www.asciitable.com/>.