Pattern Matching

An Introduction to File Globs and Regular Expressions

The danger that lies ahead

- Much to your disadvantage, there are two different forms of patterns in UNIX, one used when representing file names, and another used by commands, such as **`grep`**, **`sed`**, **`awk`**, and **`vi`**. ***You need to remember that the two types of patterns are different.***
- Still worse, the textbook covers both of these in the same chapter, and I will do the same, so as not to digress from the order in the book. This will make it a little harder for you, but with practice you will get the hang of it.

File globs

- In card games, a ***wildcard*** is a playing card that can be used as if it were any other card, such as the Joker. Computer science has borrowed the idea of a wildcard, and taken it several steps further.
- All shells give you the ability to write patterns that represent sets of filenames, using special characters called ***wildcards.*** (These patterns are not regular expressions, but they look like them.) The patterns are called ***file globs***. The name "***glob***" comes from the name of the original UNIX program that expanded the pattern into a set of matching filenames.
- A string is a ***wildcard*** pattern if it contains one of the characters '?', '*' or '['.

File glob rules

- ***Rule 1***: a character always matches itself, except for the wildcards. So **`a`** matches '**`a`**' and '**`b`**' matches '**`b`**' and so on.
- ***Rule 2:*** A sequence of characters that does not contain any wildcards matches itself, so **`hello`** matches '**`hello`**'.
- ***Rule 3:*** **`?`** matches exactly one character, including blanks and wildcard characters. It matches itself as well. So **`??`** matches any filename with exactly two characters in it, such as '**`aa`**' or '**`bb`**' or '**`b?`**' or '**`_t`**'. **`?`** is an example of a wildcard.
- ***Rule 4:*** **`?`** will not match a '**`.`**' when it is the first character in the file name.

File globs: character classes

- [*list-of-characters*] matches any single character in the list. The *list-of-characters* can be specified as a *range*, which is of the form *c*`-`*d*, where *c* and *d* are characters and no space is between. Examples:

`[a-zA-Z]`	matches any single letter
`[0-9]`	matches any single digit
`[a-zA-Z0-9]`	matches any letter or digit
`[[]`	matches left bracket '`[`'
`[-a]`	matches '`a`' or '`-`'
`[]]`	matches '`]`'

File globs: character class complements

- Putting a `!` as the first character in the list forms the complement list. [! ***list-of-characters***] matches any character ***NOT*** in the list. Examples:

  `[! ]` matches any character that is not a space

  `[!a-zA-Z]` matches any character except letters

File glob wildcards: *

- '*' matches *0 or more characters*. Examples:

  `s*` matches any filename starting with `s`

  `bin*` matches any filename starting with `bin`

  `t*c` matches any filename starting with `t` and ending with `c`.

- *But*

  `*` matches all filenames except those starting with '`.`'.

  `.*` matches only filenames starting with '`.`'

File glob examples

`hwk[0-9].???`

matches all files whose names start with **`hwk`** and are followed by a ***single*** digit then a '**`.`**' then 3 characters, such as **`hwk1.bak`**.

`w*.[a-z][a-z][a-z]`

matches all filenames starting with '**`w`**' having a '**`.`**' somewhere after **`w`** after which are 3 ***lowercase*** letters.

`[!a-zA-Z]*_*`

matches all files whose names start with a character other than a letter, and have an underscore somewhere in them.

More file glob examples

`/data/biocs/b/student.accounts/*/.bashrc`

matches all **`.bashrc`** in all user home directories, provided all home directories are in **`student.accounts`**.

NOTE – You cannot match a slash "/" in a pathname with a wildcard. File globs are only used to match what goes in between the slashes in the pathname.

`[!.]*` matches all filenames ***not*** starting with a '**`.`**'.

- For more details on using globs, consult the man page for **`glob`** in section 7. (Type man 7 glob.)

Empty file globs

If a file glob does not match any filenames, bash does not replace it with an empty string; instead it treats the pattern as the filename. For example, if there are no files in the current working directory that end with a "." then

```
ls *.
```

matches will try to list a file whose actual name is '*.' which will be an error. The bash variable nullglob can be set (using **`shopt -s nullglob`**) to replace the name by a null string.

Regular expressions in filters

- The next several slides introduce ***regular expressions***. These are a special kind of pattern used by **`grep`**, its two cousins, **`egrep`** and **`fgrep`**, as well as **`vi`**, **`sed`**, **`ed`**, and **`awk`**.
- They are used within the **`vi`** editor for searching and replacement of strings.
- They are also partly the foundation of pattern-matching in **`Perl`**.
- Therefore, they are of fundamental importance in using UNIX efficiently.

Regular expressions in filters

- In the remainder of these slides you will learn the rules for constructing regular expressions. The best way to understand them is to see what they do when given as patterns to **`grep`**.
- For example, suppose that you are curious what the regular expression "**`[acgt][acgt]*`**" matches. If you type the command

```
grep -w "[acgt][acgt]*"
```

without a file name after it, then **`grep`** will use whatever lines you type on the keyboard to find a match. If what you type matches, then when you press the Enter key, it will echo it below. If not it will not echo it.

Regular expressions: What are they?

- ***A regular expression (re, for short) is a pattern that represents a set of character strings***. A ***character string***, or a ***string*** for short, is any sequence of characters, including blanks, newlines, punctuation, and control characters.
- For example, if we invented a rule that '**#**' represents any single digit from 0 to 9, then the pattern **##** would represent all strings consisting of exactly two digits, such as **00**, **01**, **02**, **03**, …, **10**, **11**, …, **20**, …, **30**, … **97**, **98**, and **99**.
- We say that a ***re matches a string s*** if ***s*** is in the set that the ***re*** defines. Thus **##** would match **56** in our ***fictitious*** regular expression language.

Regular expression form

- The rules that define the form of ***re***'s may vary from one operating system to another or from one application to another, so there is no single set of rules that defines how they look.
- Versions of UNIX that conform to the POSIX 1003.2 standard will have the same regular expressions. In spite of the standard, there will be small differences in form.
- The **`regex`** man page in section 7 defines the POSIX-compliant regular expressions. The ***re***'s described in these slides are a subset that are common to all UNIX systems.

Regular expression building blocks

- Basic regular expressions are built up from operands and operators in much the same way that arithmetic expressions are constructed.
- The fundamental building block of a regular expression is a single character. Most single characters match themselves (not all do.) E.g.

  `a` matches '`a`'

  `b` matches '`b`'

  `1` matches '`1`'

  and so on.

Basic *re* operations: Concatenation

- ***Concatenation*** is the juxtaposition of two strings.
- ***The concatenation of two regular expressions r and s is the set of all possible strings xy, where r matches x and s matches y.***

  `ab` matches '**`ab`**'

  `11` matches '**`11`**'

- ***Concatenation is associative:***

  `abc` is really **`(ab)c`** and so it matches '**`ab`**' concatenated with '**`c`**' which is '**`abc`**'.

- Concatenation is really called ***product***.

Basic *re* operations: Closure (*)

- The only explicit, basic operator is *, which is the ***closure operator***.
- ***A regular expression followed by * matches the concatenation of 0 or more strings each of which is matched by the regular expression.***
- For example:

  a* matches 0 or more **a**'s: , **a**, **aa**, **aaa**, **aaaa** …

  ca* matches **c** followed by whatever matches **a***, so it matches **c**, **ca**, **caa, caaa**, **caaaa,** …

  ca*t matches **ct**, **cat**, **caat**, **caaat**, **ccaaaat**, …

Basic *re* operations: Closure (**`*`**)

- **`cc*aa*`** is the product of **`cc*`** and **`aa*`**. It matches all strings formed in all possible ways by concatenating a string from **`cc*`** to one from **`aa*`**. The best way to list these is by writing all strings of length 2, then length 3, then 4 and so on:

  `ca`, **`cca`**, **`caa`**, **`ccca`**, **`ccaa`**, **`caaa`**,

  `cccca`, **`cccaa`**, **`ccaaa`**, **`caaaa`** , ...

- Because a pattern like **`a*`** matches zero **`a`**'s as well as 1 or more **`a`**'s, if you want to match one or more **`a`**'s, you need to use the re **`aa*`**, which matches **`a`**, **`aa`**, **`aaa`**, and so on.

More Examples of *

If you want to apply the * operator to more than one character, you have to enclose it in \(\) brackets. For example, an re that matches all strings of the form **abababababab**, i.e., **ab** repeated any number of times, is

\(ab\)*

This also matches the empty string. If you want to match only lines containing at least one **ab**, you should use either

\(ab\)*ab or **ab\(ab\)***

Basic character classes

The period '`.`' matches any single character.

There are other one character regular expressions.

`[`*list-of-characters*`]` matches any single character in the list. This is the same rule as file globs:

`[a6j&]`	matches `a`, `6`, `j`, or `&`
`[0-9]`	matches any single digit
`[a-zA-Z0-9]`	matches any letter or digit

- The `^` inside brackets means the complement:

`[^a6j&]`	*matches anything BUT* `a`, `6`, `j`, or `&`
`[^0-9]`	matches anything but a digit.

Basic character classes

`[]]`	matches `]`
`[0-9-]`	matches any single digit and hyphen `-`
`[-0-9]`	matches any single digit and hyphen `-`

Character classes combined with *

- You can use character classes with the `*` operator to create useful patterns:

`\(c[acgt]g\)*`	matches 0 or more sequences of **`cag`**, **`ccg`**, **`cgg`**, or **`ctg`**
`[1-9][0-9]*`	matches any decimal numeral except **`0`**
`[A-Z][a-z]*`	matches words that start with an uppercase letter.
`\(...\)*`	matches any string whose length is a multiple of 3..

Predefined character classes

- Certain character classes have special names. Some of them are:

`[[:alpha:]]`	matches any letter, upper or lowercase
`[[:alnum:]]`	matches any letter or digit
`[[:lower:]]`	matches lowercase letters
`[[:upper:]]`	matches uppercase letters
`[[:punct:]]`	matches punctuation
`\w`	equivalent to `[[:alnum:]]`

- These must be typed exactly as I have written them here.

Anchors

- The caret **^** anchors a ***re*** to the beginning of a line, and the dollar sign, **$**, anchors it to the end of the line. For example:

^drwx	matches lines whose ***first 4*** characters are **drwx**
^\w	matches lines that ***begins with*** a letter or digit
abcd$	matches lines whose ***last 4*** characters are **abcd**
^abc$	matches lines that ***contain only*** **abc**
^$	matches empty lines
^[ ]*$	matches empty lines or lines containing only spaces

Metacharacters

- If you want to match one of the special characters such as `*`, `[`, `]`, `.`, or `-`, you need to put a backslash in front of it:

  `\.` matches `.`

  `\*` matches *

  `\[` matches `[`

  `\]` matches `]`

  `\\` matches `\`

- These characters are called ***metacharacters***.
- Note: there are other ways to do this. These are just the easiest to remember.

Extended regular expressions

- The set of basic regular expressions was extended to include more powerful operators and has come to be called the ***extended regular expression language***.
- The **`egrep`** filter recognizes these expressions. So does **`grep`** if you give it the **`-E`** option: **`egrep`** is the same as **`grep -E`**.
- Other programs recognize the extended regular expression language. Most notable are **`sed`** and **`vi`**.
- The **`grep`** man page describes these expressions in sufficient detail. In these slides, I will cover just a few useful operators.

Extended regular expressions: `|` and `+`

`|` ***This is the OR-operator***. If `r` and `s` are regular expressions, then `r|s` matches either strings that `r` matches or strings that `s` matches:

`acg|act`	matches either `acg` or `act`
`aa*|bb*`	matches either a sequence of 1 or more `a`'s or a sequence of 1 or more `b`'s.

`+` ***This is called positive closure***. It is identical to `*` except it matches ***1 or more*** instead of ***0 or more*** occurrences:

`a+`	is the same as `aa*`
`a+|b+`	is the same as `aa*|bb*`

Extended regular expressions: `?`

`?` *This matches 0 or 1 occurrences of its argument*.

`a?`	matches either the empty string or `a`
`ab?a`	matches either `aa` or `aba`
`..?`	matches any single symbol or two symbols
`(cc?)+`	matches 1 or more combinations of `cc` and `c`

- ***Beware: it is different than the glob `?` operator !!!***

Backreferences

- When you enclose a basic regular expression in **`\( \)`** brackets, or an extended regular expression in ordinary parentheses **`( )`**, the string that matched it is "***remembered***" for future use. The regular expression ***backreference***, **`\1`**, matches the first "**`remembered string`**."
- For example, in

  `\(aa*\)b\1`

  any string that matches **`aa*`** is saved into a memory cell named **`\1`**. Therefore the only strings that this expression matches are either **`aba`**, **`aabaa`**, **`aaabaaa`**, **`aaaabaaaa`**, etc.

Backreference examples

```
^\([acgt][acgt][acgt]\)[acgt]*\1$
```

matches any dna string that starts and ends with the same codon.

```
\([0-9]\)\1\1-\1\1\1\1
```

matches phone numbers like **111-1111**, **222-2222**, **333-3333** and so on.

Backreferences in general

- In general, the expressions **`\1`**, **`\2`**, **`\3`**, …, **`\9`** remember matches of the 1st, 2nd, 3rd, up to 9th parenthesized regular expressions.
- This complicated expression

  `^\(.*\):\(.*\)::\1:\2$`

  matches lines of the form

  x*`:`*y*`::`*x*`:`*y

  where ***x*** and ***y*** are possible empty strings, such as

  `abc:666::abc:666`

fgrep

- The third member of the **grep** family, **fgrep**, is the fixed-string version of **grep**. The way that it is intended to be used is as follows.
- Create a list of strings, one per line, in a file. Suppose the file is named **patternfile**.

  fgrep -f patternfile searchfile

  will display all lines in **searchfile** that match any of the strings that occur in **patternfile**. In a sense, **patternfile** acts like a dictionary of words whose presence you want to check in **searchfile**.

Summary

- This lesson introduced a very powerful computational tool called regular expressions. The work that goes on behind the scenes to match them is significant. Tools like **`grep`** can simplify many of the tasks you have to do, so it is worth the time to master regular expressions now.

Things to try

- Write a **`grep`** command to display all files in the PWD that are executable by user, group, and others.
- Write a command that lists all files in the PWD that are Perl scripts. (Use the file command with grep.)
- Write a command that lists all nitrogen atoms in the valine amino acids in a given PDB file. (N is the symbol for nitrogen and VAL is the symbol for valine.)
- Read the man page in section 7 for regex.