Operating systems fundamentals - B09

Michael Brockway, David Kendall

Northumbria University

Outline

- Wild-card characters and pattern-matching
- Regular expressions, *Extended* regular expressions
- Regex syntax and examples
- Using regular expressions in text editors
- `grep`, `egrep`
- Other text processing tools
    - `tr`, `cut`, `sort`, `uniq`

Pattern-matching, Wild Cards

There are many computing problems that involve finding a *set of strings* that *match a pattern* in some *piece of text*
For example, we have seen that *wild card* characters are useful when we use the unix `ls` command

- $ `ls -l *.c` - lists all files in working folder with names ending '.c'
- $ `ls -l *.c*` - lists all files in working folder with names containing '.c'
- $ `ls -l *.c??` - lists all files in working folder with names ending with '.c' followed by two additional characters

The `*` character *matches* any string of 0 or more characters; the `?` character matches any single character.

- $ `ls -l` *string*

lists all files with names that match *string*; this "search string" may contain wild-card (`*`, `?`) characters.

Pattern-matching, Wild Cards

These wild-cards are very powerful and can be used with all the file management commands

- `ls`, `rm`, `mv`, `cp`, etc;
- Seach strings containing wild-card characters are sometimes called “globs” and searching for a match with them is “globbing”.

The underlying idea is that the program or command is

- *searching* a body of text ...
- *not* just for a particular word or substring;
- but rather, for words (substrings) that *match* a *pattern*.

Without wild-cards, the match has to be an exact match with the string that specifies the pattern. With wild-cards, a more powerful search is possible.

Regular Expressions

Regular Expressions are a more powerful version of this idea. They can express patterns that it is not possible to express using “glob” expressions.

- A *regular expression* is a *pattern* comprising a sequence of *literal characters* and *meta-characters*
- The literal characters, e.g. lowercase and uppercase letters and digits, just stand for themselves in a pattern, e.g. `d` and `4`
- The meta-characters are characters that don’t stand for themselves but have a special meaning, e.g. $\star$ and $+$
- A text is searched for one or more strings that *match* the pattern;

The details of the meta-characters and how they can be used to create patterns are coming up.
There are different versions regular expression syntax
The POSIX standard for *extended* regular expressions is described in these slides. There are minor variations to the definitions in some non-Unix implementations – eg in the Java, Perl, Python languages.

Regular Expressions - Syntax

- A `.` in the pattern string will match any single character
- A `?` *after a character* in the pattern string will match zero or one occurrences of the character (it's an *optional* character);
- A `*` *after a character* in the pattern string will match zero or more occurrences of the character;
- A `+` *after a character* in the pattern string will match one or more occurrences of the character;

Examples

- `.at` matches `cat`, `bat`, `mat`, ...
- `colou?r` matches `color`, `colour`;
- `c*at` matches `at`, `cat`, `ccat`, `cccat`, etc;
- `c+at` matches `cat`, `ccat`, `cccat`, etc;

Regular Expressions - More Syntax

Regular expressions are more powerful than this. The matching characters can work with *elements*. An element is

- a single character, or
- a string of characters or other elements enclosed in `()`, or
- a string of characters enclosed in `[]`, or
- strings of characters separated by `|`

Examples

- `(cat)?` matches zero or one occurrences of `cat`
- `(cat)*` matches zero or more occurrences of `cat`
- `(cat)+` matches one or more occurrences of `cat`– eg `cat`, `catcat`, `catcatcat`, ...
- `[aeiou]` matches any one of `a`, `e`, `i`, `o`,
- `[aeiou]+` matches a string of one or more lower-case vowels;
- `abc|def|ghi` matches *either* `abc` *or* `def` *or* `ghi`

Regular Expressions - More Examples

- `[hc]at` matches `hat`, `cat`;
- `[hc]+at` matches `hat`, `cat`, `hhat`, `chat`, `hcat`, `ccat`, `hhcat`, `hccat`, etc – but not `at`;
- `[hc]*at` matches all these AND `at`;
- `[hc]?at` matches `hat`, `cat`, `at`;
- `(cat|dog)s?` matches `cat`, `cats`, `dog`, `dogs`;
- `cent(re|er)` matches `centre`, `center`

Regular Expressions - Yet More Syntax

- `\t` matches a TAB character, `\n` a NEWLINE character, etc;
- `\[` matches a literal `[` and similarly with all the other pattern-specifying characters; `\\` matches a `\`
- `^` matches the beginning of a line and `$` matches the end of a line.
  - `^Chapter 1$` matches a heading, “Chapter 1” on a line by itself.
  - NB this ^ is different usage from the one defined next!!!
- a string of characters enclosed between `[^` and `]` defines an element that matches any character *not* in the string; for example,
  - `[^bc]at` matches any 3-letter string ending ’`at`’ *except* `bat`, `cat`
  - `[^ \t\r\n]` matches any non-space character
- *Ranges* can be used inside `[ ]` and `[^ ]` - eg
  - `[0-9]` matches any digit
  - `[A-Za-z]` matches any alphabetic character
  - `[a-dx-z]` matches any alphabetic character not in range e-w
  - `[^A-Za-z]` matches any *non*-alphabetic character

Matching an exact number of elements

A number, *n*, in braces immediately after an element matches exactly *n* occurrences of the element.
It is possible to specify a *range* instead of an exact number.
So

- *element*{*n*} matches a string of exactly *n* occurrences of *element*
- *element*{*n*,} matches a string of at least *n* occurrences of *element*
- *element*{, *n*} matches a string of at most *n* occurrences of *element*
- *element*{*m*, *n*} matches a string of *m* to *n* occurrences of *element*

Examples

- `[01]{16}` matches a string of 16 bits (0s and/or 1s)
- `[01]{8,16}` matches a string of 8 to 16 bits, as many as possible

Using Regular Expressions in Tools

Many text editors do find and find/replace using regular expressions.

- `vi/vim`
- `emacs`
- `gedit`
- `atom`

Command-line tools use regexes too -

$ `grep -E` *search-term target-file*; or

$ `egrep` *search-term target-file*

  - the `-E` option specifies *extended* regular expressions
  - *search-term* is a regular expresson; the target file is searched and all lines containing a match are output.
  - Usually a good idea to enclose the search term in quotes ’...’.

- Can be used in a pipe too: eg

```
$ ls -l | egrep 'michael'
```

- `sed` and `awk` – more on these later

Translating characters - `tr`

```
$ tr "string1" "string2"
$ tr "string1" "string2" < input-file
```

The basic form translates input characters that occur in *string1*, substituting the corresponding character in *string2*, and outputting on `stdout`. Input is from `stdin` by default, but you usually redirect input from a file or pipe it from another process. Eg -

```
$ tr "abc" "XYZ" < data.txt
```

- Every ‘a’ in the file is replaced by ‘X’, ‘b’ by ‘Y’ and ‘c’ by ‘Z’. Output is `stdout`, original file is unchanged.

You might redirect or pipe the output too -

```
$ tr "abc" "XYZ" < data.txt > result.txt
```

`tr` - Variations

Other versions of `tr` just take a single string of characters. The `-d` option deletes every occurrence of a character listed in the string; the `-s` option replaces every repeated occurrence of a listed character with a single occurrence.

```
$ tr -d "abc" < data.txt > result.txt
```

- Delete every occurrence of ‘a’, ‘b’, ‘c’. Output is `stdout` unless redirected or piped.

```
$ tr -s "abc" < data.txt > result.txt
```

- “Squeeze”: Replace repeated occurrence of ‘a’ with a single ‘a’, similarly with ‘b’, ‘c’.

So to replace a repeated white space made of SPACEs and TABs by a single space,

```
$ tr -s "  " < data.txt > result.txt
$ ls -l | tr -s "  " > result.txt
```

Filtering a line of text - `cut`

This command works on a file of lines of text.

- Each line is divided into *fields* by *delimiters*
- The default delimiter is TAB, `\t`
- You can choose a different delimiter with the `-c` option
- You can output a subset of fields with the option `-f` *LIST*
  - *LIST* is a list (or range) of field numbers
- Output is to `stdout` but can be redirected or piped elsewhere
- Input is can be redirected from a pipe

Examples

```
$ cut -d "," -f 2,3 data.txt
```

- Data is a file of lines of items separated by commas. Output second and third item of each line.

```
$ ls -l | cut -d " " -f 5-
```

- Use SPACE as delimiter of output of `ls -l`. Output from file size onwards.

`sort`

Sorts lines of one or more files, ouputting on `stdout` (which can be redirected or piped, of course):

$ `sort` *file1*

$ `sort` *file1 file2 file3*

$ `sort` *file1 file2 file3* > `output.txt`

- By default, the whole line is its *sort key*.
- Lines break by white-space into *words*. Option `-k` *n* uses word number *n* as the sort key. Example: to sort on 3*rd* word -

  ```
  $ sort -k 3 data.txt
  ```

- Option `-d` sorts into dictionary order (ingore other than letters, numerals blanks);
- Option `-f` ignores upper/lower case;
- Options `-n`, `-g`, sorts into numeric order. The `-g` option is slower, but handles floating-point numbers.

sort (ctd)

The sort command can take redirected or piped input: eg

```
$ ls -l | sort -n -k 5 > output.txt
```

  - sorts output of `ls -l` in order of size of files

- Option `-b` ignores leading blanks;
- Option `-r` sorts into reverse order

uniq

Usually used after sort ...

```
$ sort -k 3 data.txt | uniq > output.txt
```

- removes duplicate lines in sorted text
- Options
  - `-f` *n* skips first *n* fields in comparison of lines;
  - `-s` *n* skips first *n* characters in comparison of lines;
  - `-i` ignores case in comparison of lines;

References

- http://linuxcommand.org/tlcl.php
    - Shotts, W., The Linux Command Line, Chp. 19 gives an excellent introduction to the use of regular expressions in Linux
- http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html
    - The POSIX standard reference on regular expressions
- http://en.wikipedia.org/wiki/Regular_expression
    - the Wikipedia page has a good summary of basic definitions (some beyond our scope) and underlying theory.
- http://www.regular-expressions.info/reference.html
    - a reference page on regex syntax
- http://www.regular-expressions.info/grep.html
    - a reference page on `grep`
- http://www.cyberciti.biz/faq/grep-regular-expressions/
    - a page of frequently-asked questions.