Filenames, globbing, greping, and regexp

In these notes we deal with some (slightly) interrelated matters: conventional filenames in Unix; how the shell can handle multiple filenames by using special characters (globbing); and the family of *grep* programs for extracting lines from files, which also use special characters to represent ranges of possible combinations. The Unix term for such a use of special characters is *regular expressions*, often shortened to *regexp*. These can be compact and powerful–and as a consequence very obscure.

File Names

We start with file names. By design, there are not many restrictions on this, but there are some:

1. Names (on the Mac) can include spaces, but this is still something to avoid, since most Unix routines assume that a space ends a name.

2. Likewise, names should not contain any characters that have special meaning to the shell, for example `?*-()|[]`. The safest approach is to stick to letters and numbers, and use `.` or `_` for separators.

3. The Mac OS has a very strange approach to upper and lower case: it shows case differences, but does not believe in them. That is, if you create a file called `tmp`, it will display that way; likewise if, in another directory (say the one above) you create a file called `Tmp`, it will display that way. However, if you attempt to create files with these two names in the same directory, you will get:

   ```
   % cat file1 > tmp
   ```

```
% ls -l tmp
-rw-r--r--    1 agnew    admin          71 Oct  1 23:57 tmp
% cat file1 > Tmp
tcsh: Tmp: File exists.
% ls -l Tmp
-rw-r--r--    1 agnew    admin          71 Oct  1 23:57 Tmp
% ls -l tmp
-rw-r--r--    1 agnew    admin          71 Oct  1 23:57 tmp
% ls -l Tmp tmp
-rw-r--r--    1 agnew    admin          71 Oct  1 23:57 Tmp
-rw-r--r--    1 agnew    admin          71 Oct  1 23:57 tmp
% ls
  file1   tmp
%
```

and just running `ls` shows only the file `tmp`. So even if two files are displayed there is really only one.

4. There are a very large number of standard suffixes for files. None are required – you can name any file anything – but often the system uses these to decide what a file is. Table 1 gives some examples.

globbing: Character Expansion by the Shell

When you run a program such as`rm` or`ls`, you can control which files it can act upon; aside from the special case that a single`ls` will list all the files, what files are acted on will depend on which ones you list after the program. But you need not explicitly name each one; given certain characters in the name you type[1] the shell will perform what is called *filename expansion*; before the line is executed, the shell will process it and replace what you typed with all the filenames that match it, allowing for the "inexplicit" characters. This is also called *globbing*, presumably because you can specify a whole glob of files very easily.

An example will help; for this one we use the`?` character, which has the meaning "match any single character"

```
% ls
aa  aa.x  aa.y  ab  ab.x  ac  ac.x  ba  ba.y  bb  bb.y
```

[1] These are called *metacharacters*.

	Graphics/Display
.gif	GIF (raster)
.jpeg	JPEG (raster)
.jpg	JPEG (raster)
.kml	Google Earth uncompressed
.kmz	Google Earth compressed
.pdf	Portable Document (vector)
.png	Portable Network Graphics (both)
.ps	Postscript (vector)
.svg	Portable Document (vector)
	Documents
.doc	MS Word
.docx	MS Word, new format
.html	hypertext markup (Web)
.rtf	rich text format
.tex	Tex/LaTex source
.wpd	Word Perfect

	Source Code
.c	C source code
.cc	C++ source code
.f	Fortran 77 source code
.f90	Fortran 90 source code
.m	MatLab script
.mat	MatLab ???
.o	object file
.pl	perl
	Compression
.bz2	compressed with bzip2
.gz	compressed with gzip
.Z	compressed with adaptive Lempel-Ziv
	Storage
.tar	archive of files and directories
.tgz	tar archive, compressed with gzip
.zip	archived/compressed with zip
	Miscellaneous
.xls	Excel spreadsheet

```
% ls ?a
aa  ba
% ls a?
aa  ab  ac
% ls aa.?
aa.x  aa.y
```

The other globbing character that is frequently used (in fact, the one that gets almost all of the use) is *, which means "any character string, except an initial .". To continue with our example

```
% ls *
aa  aa.x  aa.y  ab  ab.x  ac  ac.x  ba  ba.y  bb  bb.y
% ls a*
aa  aa.x  aa.y  ab  ab.x  ac  ac.x
% ls *.x
aa.x  ab.x  ac.x
% ls a*y
aa.y
```

```
%
```

The initial . is not included in this expansion is to protect files that begin with a . from being removed by mistake. This type of file name is used for files that might be needed but which are in some way not something the user needs to know about – for which reason they are invisible to `ls` unless the `-a` flag is used. If you run `rm *` in a directory and then cannot remove it with `rmdir`, usually there turn out to be some of these invisible files still in the directory.

The *grep* Family of Programs

Before turning to the other use of special characters, we look at the textitgrep family of programs, since one of these, textitgrep, is the best way to illustrate these uses. There are two programs (actually there is a third, but it is not much used now), which could be run as

```
% cat file | grep expression
% cat file | fgrep string
```

The program `fgrep` reads in lines from standard input, and writes any to standard output that contain a given, fixed, character string. So, for example, taking these notes (up to here) as input:

```
% cat notes.glob.tex | fgrep something
ranges of possible combinations, something that, for these and other
Names (on the Mac) can include spaces, but this is still something to
% cat notes.glob.tex | fgrep z
% fgrep z notes.glob.tex
%
```

where the last two lines show two possible forms (and that there are no letter z's in these notes, at least before this sentence).

This is a very useful way to extract something from a file. For a fixed string, `fgrep` is the fastest way to extract lines; `grep`, however, is more general, because it can look for regular expressions, which we now turn to.

Regular Expressions

Regular expressions give special meaning to certain characters in looking for a match to a part of a character string, just as the shell does when trying to match filenames. Unfortunately, the meanings assigned to particular characters are not all the same.

First of all, `.`, which has no special meaning to the shell, has the meaning “any character” in a regular expression. For example, to find the string: “a `t`, followed by any character, followed by an `e`” the regular expression would be `t.e`; we would search for this string in these notes using `grep` by typing:

```
%cat notes.glob.tex | grep 't.e'
In these notes we deal with some (slightly) interrelated
matters: conventional filenames in Unix;
%
```

where we have shown only the first couple of lines matched). The first line matches because of `these`, and the second one, less obviously, because of `matters`: both contain a `t`, some character, and an `e`. We can have more than one such metacharacter:

```
%cat notes.glob.tex | grep 't..e'
cah% cat notes.glob.tex | grep ’t..e’
avoid, since most Unix routines assume that a space ends a name.
%
```

where the matching word is `routines`: not (at least to me) immediately obvious.

The character `*` means “zero or more of the previous character”, so `x*` matches anything, `xx*` matches one or more `x`’s, `xxx*` matches two or more `x`’s, and so on

```
%cat notes.glob.tex | grep 'xx*'
matters: conventional filenames in Unix;
and the family of grep programs for extracting lines
%cat notes.glob.tex | grep 'xxx*'
%
```

(there are no strings `xx`).

These two metacharacters have referred to types of characters; there are others than refer to positions in the line. These are `^`, for “before the first character on the line”, and `$`, for “after the last character on the line”. We can combine these with the both characters and other metacharacters; for example, `^a` returns all lines starting with `a`; `a$` returns all lines ending with `a`; `tt*$` returns all lines ending with one or more `t`’s.

```
% cat notes.glob.tex | grep '^a'
and the family of grep programs for extracting lines
avoid, since most Unix routines assume that a space ends a name.
% cat notes.glob.tex | grep 'a$'
meaning to certain characters in looking for a match to a
% cat notes.glob.tex | grep 'tt*$'
from files. These also use special characters to represent
aside from the special case that a single ls will list
%
```

Finally, the symbols `[` and `]` form a pair: if they are separated by any other characters, the expression `[`*chars*`]` is taken to mean "any of the characters between the square brackets". For example, `[ABC]` means any of these three characters:

```
%cat notes.glob.tex | grep '[ABC]'
But you need not explicitly name each one;
An example will help; for this one we use the? character,
%
```

One character is itself special within the brackets (unless it is the first one): the dash, `-`. When surrounded by characters, it fills in the ones between them. Here "between" refers to the order of the characters in the ASCII encoding, which is:

```
!"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

so that, for example, `[0-9]` means all numerals, `[A-Z]` means all uppercase letters, `[a-z]` means all lowercase letters, and `[A-Za-z]` means all letters; in the last case, we do not write `[A-z]` because this would match some other characters as well. For example,

```
%cat notes.glob.tex | grep '[V-Z]'
When you run a program such as rm or ls,
You can see this is a very useful way to extract something from a file.
%
```

We can combine all of these features to get specific, but flexible, matching. For example, to match a line that has zero or more initial blanks followed by a number, we would write `^ *[0-9]`. If the initial number had two digits, no leading zeros, and a decimal point, we would write `^ *[1-9][0-9]\.`; the backslash in front of the period is said to *escape* this character, so that it is interpreted as what it is, rather than in its metacharacter form. Without this, `^ *[1-9][0-9].` would match any line that started with zero or more blanks, followed by a numeral from 1 through 9, followed by a numeral from 0 through 9, followed by any character at all.