Wildcards Gone... Wild

By Jerry Peek

When you type a filename on the command line (at a shell prompt), you may use *filename completion* to save time and typing. A popular feature, filename completion lets you type the first few letters of a file or directory name and use the TAB key to ask the shell to fill in the rest.

The shell can also complete file and pathnames noninteractively by using *wildcards* to do *pathname expansion*. For example, the command line `rm *.o` removes (with the wildcard `*`) all files in the current directory whose name ends with `.o`. Wildcards are much more powerful and flexible than filename completion, and most shells have their own advanced wildcards that do powerful and sophisticated things.

This article assumes that you've used the basic wildcards `*` (matches anything), `?` (matches any single character), `[xyz]` (matches any of *x*, *y*, or *z*, and `[v-z]` (which matches any letter in the range *v* through *z*, inclusive). Let's look what happens underneath the surface, so you can be confident of getting the result you want. Along the way, let's also take a look at a handy function that makes wildcards work more like filename completion, letting you see an expanded command line before you run it. Time to dig in!

How Wildcards Work

When a shell reads a command line (either from a shell prompt or a shell script), it goes through a series of steps to interpret what it finds. One of the steps is to look for wildcard characters — like `*`, `?`, and others — and replace them with one or many file or directory names that match the wildcards.

Let's see an example. Let's list what's in the current directory, then remove some of the files and see what's left:

```
% ls
a.cc a.o j.cc j.o s.cc s.o subdir
% rm *.o
% ls
a.cc j.cc s.cc subdir
```

What happened? The shell saw the wildcard `*` in the word `*.o`, searched the current directory for all file or directory names that ended with `.o`, and passed the resulting name(s) to the *rm* program. It's exactly as if you'd typed `rm a.o j.o s.o` on the command line. *rm* sees only those three arguments; it doesn't know that you gave the shell a wildcard. (There's an exception in some shells, though, when a wildcard doesn't match anything. See the sidebar "If a Wildcard Doesn't Match.")

Did the shell match any filenames ending with *.o* in the subdirectory named *subdir*? We can be sure that it didn't. Why? Because except for a few advanced wildcards, *a wildcard never matches the / in a pathname*. To remove the "`.o` files" in another directory, you have to build a pathname to them.

For instance, we could clean out *subdir* with a command like `rm s*/*.o`. That would expand into something like `rm subdir/b.o subdir/d.o`. Notice that each argument is a complete pathname that follows the pattern of the wildcarded word. Because we knew that there was only one subdirectory whose name started with an `s`, we could use `s*` to match its name. (We've also included the tricky filename *s.cc* in this sample directory to show that `s*` doesn't match *s.cc* in this case. That's because *s.cc* is a file, and files do not contain subdirectories.)

What's in the (Wild)cards?

To see what the shell does with wildcards, set its "echo" option. In *tcsh* and *csh*, type the command `set echo`; in Bourne-type shells like *bash*, type `set -x`. (To cancel this mode, type `unset echo` or `set +x`, respectively.)

Let's see an example from *bash*.

```
$ set -x
$ rm *.cc s*/*.cc
rm a.cc j.cc s.cc subdir/b.cc subdir/d.cc
$ set +x
```

IF A WILDCARD DOESN'T MATCH

What happens if a wildcard doesn't match any files or directories? The answer depends on your shell.

By default, if no wildcarded words match, the *csh* and *tcsh* shells won't run the command line at all; instead, they print a "No match" error.

Bourne-type shells, such as *bash*, pass the *unexpanded* command line with the wildcarded words to the program, and let the program decide what to do.

If you've used several wildcarded words on a command line (like `rm *.o *.cc`) and only some of them match, the shell may remove the words that don't match and expand the words that do.

There are other possibilities and your shell may have options to control all of this. For instance, *bash* has the `nullglob` option and *tcsh* has `nonomatch`.

A good way to see what will happen is to use the *echo* command with your wildcards, like `echo *.o *.cc`. The shell will expand (or not expand) the wildcards, then show either the result or the error message.

The shell shows the expanded command line before executing it. This wouldn't have been easy with filename completion! You'd need to type more than half of the command line and hit TAB several times.

If you'd like confirmation before a wildcarded command line runs, use the little homemade shell function named `x()`. *Listing One* shows the `x()` function as written for *bash;* the script may need tweaking on other Bourne-type shells. For example:

```
$ x rm /a/b/0?
rm /a/b/00 /a/b/01 /a/b/06
x: execute that command line? [yn](y) n
x: NOT executing: rm /a/b/00 /a/b/01 /a/b/06
```

`x()` outputs the command line with long lines neatly wrapped by *fmt*, and then prompts (by not outputting a newline at the end) whether you want to execute the command or not. If you answer `y` or simply press RETURN, the command runs.

Wilder Wildcards

Every shell supports the basic wildcards `*`, `?`, and `[xyz]`, and all shells support wildcard ranges — such as `[0-9]*`, which matches any filename that starts with a digit — and lists like `*[_.]*`, which matches any filename that contains an underscore (`_`) or a dot (`.`).

(Astute readers may be scratching their heads at this point, wondering, "Doesn't `[0-9]*` mean a filename made up of zero or more digits?" In *egrep* or *perl,* that *regular expression* indeed matches *strings* composed of zero of more digits; however, in the shell, `[0-9]` is associated with the first character in the filename, and the `*` is applied to the rest of the characters in the filename. You can create a wildcard pattern that matches all filenames composed of zero or more digits, but the syntax is different. See below.)

In many shells, you can also match any character that's not in a range or a list: just put an exclamation point (`!`) or a caret (`^`) after the opening square bracket. So, `rm [^0-9]*` would remove all files whose names don't start with a digit. The `!` or `^` qualifier is often referred to as "except", as in "except the following characters."

Most shells also have their own, more-advanced wildcards. For instance, *zsh* has `**`, which can match entries in multiple directories. It's worth investigating advanced wildcards for all shells — not just the shell you're using. If another shell has a wildcard you need, you can start that shell (as a subshell, by typing its name at a prompt), execute the command lines you want, then end the shell by typing `exit`. You may need to set a shell option before some advanced wildcards will work.

LISTING ONE: `x()` function for confirming wildcards

```
x() {
  local ans
  echo "$@" | fmt
  echo -n "x: execute that command line? [yn](y) "
  read ans
  case "$ans" in
    [yY]*|"") "$@" ;;
    *) echo "x: NOT executing: $*" | fmt ;;
  esac
}
```

For example, you can start *bash* and use one of its advanced wildcard patterns this way:

```
% ls
1  11  999  1e  9e
% bash
bash$ shopt -s extglob
bash$ rm *([0-9])
  ... command line runs...
bash$ exit
% ls
1e  9e
```

The bash extended wildcard pattern `*([0-9])` expands to all filenames made up of zero or more digits. ("Zero matches" means an empty string with no characters.) As you can see, it expanded to `1`, `11`, and `99`, and omitted `1e` and `9e` because those two filenames contain an alphabetic character, `e`.

For a shell you use often, you can also store the advanced wildcards option in the shell's setup file. For instance, you can add `shopt -s extglob` to the *.bashrc* file in your home directory. Whenever you start *bash*, even for occasional use, the *shopt* command "remembers" to set the *extglob* option for you.

Given "except" and other special wildcard patterns, can you get the names of all filenames that don't contain a dot or underscore? Would `*[^_.]*` or `*[!_.]*` do the job? No, it wouldn't: that expression would match a name that starts with zero or more of any character, followed by any character that's not an underscore or a dot, followed by zero or more of any character. Instead, you could use a pattern like `!(*[._]*)`. The pattern `*([0-9])` and `!(*[._]*)` are two simple examples of *composite patterns*.

Composite Patterns

If you've used programs like *egrep* or *Perl* and their regular expressions with alternation, you'll recognize the composite wildcard patterns in *bash*. (Composite wildcard patterns are also supported in *zsh* and in *ksh*, where they're called *pattern*

lists.) A composite pattern is a series of one or more words and wildcards, separated by | characters, inside parentheses. Before the opening parenthesis is a single character — ?, *, +, @, or ! — that tells what kind of composite pattern follows. (The shells' manual pages describe these.)

For instance, ?(*pat1*|*pat2*) starts with ?; this means that the pattern matches *zero or one* of any pattern inside the parentheses. The patterns (here, *pat1* or *pat2*) can be literal filenames and can also contain wildcards.

Wildcards are much more powerful, flexible, and sophisticated than filename completion

One of the handiest composite patterns uses the ! (or "except") matching operator to give you a list of files or directories that *don't* match certain names. Here's an example.

A directory full of files has a subdirectory named *backups* where you keep backup copies of important files. You'd like to copy all files, except filenames ending with *.o*, into the *backups* subdirectory. You need to give *cp* (the file copying program) a list of every name except filenames ending with *.o and the backups directory itself. That pattern is !(*.o|backups). You'd use it like this (assuming you have shell option extglobs set):

```
$ ls
a.cc backups c.cc
a.o crossref c.o
$ cp !(*.o|backups) backups
```

The composite pattern would match everything except *a.o*, *backups*, and *c.o*. So *bash* would execute cp a.cc crossref c.cc backups.

zsh Numeric Ranges

Filenames containing numbers can be tough to match with standard wildcards. For example, let's say you're writing a book made of 33 chapter files named *chap1* through *chap33*. You'd like to copy chapters 8 through 23 to your *backups* directory. With standard wildcards, you'd need to do something like this to match the individual digits in each filename:

```
cp chap[89] chap1[0-9] chap2[0-3] backups
```

The Z shell has a long list of handy wildcards. One is the *numeric range* that lets you write the previous example this way:

```
cp chap<8-23> backups
```

An open-ended range works, too. For instance, chap<24-> matches all chapters from 24 up.

zsh Recursive Matching

As we saw earlier, to build a pathname into another directory, most wildcards make you type each slash (/) separately. The Z shell's wildcards ** and *** are the exception to that rule. They match recursively into subdirectories.

For instance, to make a long listing of any name containing *sendmail*, in any subdirectory (or sub-sub-subdirectory), type:

```
zsh% ls -l **/*sendmail*
-rwxr-xr-x  ... rc.d/init.d/sendmail
-rw-r—r—  ... sendmail.cf
-rwxr-xr-x  ... sysconfig/sendmail
```

The *** wildcard follows symbolic links; ** doesn't.

That's not all...

Wildcards aren't the only way to get lists of files. For instance, you can pipe a list of files through a filter like *sed* or *egrep* to include or exclude files, then use command substitution or *xargs* to pass the resulting list to a program. Tricks like those were often needed when shells only had basic wildcards. Now, advanced wildcards — like the ones we've seen, and others we haven't — can build almost any list of filenames or pathnames that you need.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: Wildcards for GUIs

A typical file-open dialog box in a graphical (GUI) application makes you click on individual files from a scrolling list of all filenames. If you want to open several files from a long list, you may waste a lot of time holding down the CTRL key, scrolling and clicking. Try using wildcards, *echo*, and maybe a shell loop, to build a list of pathnames in a terminal, which you can then copy and paste into the GUI's filename box.

For example, one file-conversion application wants a series of filenames, each surrounded by double quote marks and separated by spaces. A quick shell loop, typed at a shell prompt, builds that list in a flash. (This works best for longer lists than the short one we show here):

```
$ for f in 127_27{18,24,37}.CRW
> do echo -n "\"$f\" "
> done
"127_2718.CRW" "127_2724.CRW" "127_2737.CRW"
```