Bash guesses what users want to type, thanks to Perl

A MORE COMPLETE COMPLETER

The shell's completion mechanism finishes whatever you start typing when you press the Tab key. A Perl script customizes this function.

BY MICHAEL SCHILLI

almagami, 123rf

The Tab key is fairly worn on many people's keyboards. The heavy use it gets is due to the Bash shell's ability to intelligently complete command, directory, and filenames that you start typing at the command line.

Standard Repertoire

Figure 1 shows Bash's standard completion repertoire. If you type *lsm* at the keyboard, as in line 1 and press the Tab key, Bash will immediately expand this to *lsmod*, the command for querying installed kernel modules. Why? Bash knows that the three letters at the start of a command line must begin an executable command. Bash couldn't find any other command with these three letters in the *$PATH*, so it simply made the right choice.

If you look at the second command line in Figure 1, you will see that the user typed just two letters, *ls*, before pressing Tab. Again, this has to be a command, rather than a normal file, but the details are ambiguous. Dozens of commands start with *ls* in the path. In this case, Bash just sits and waits until the user presses Tab again before responding with a compact list of the possible results. Then it waits for the user to enter some more letters and reduce the number of candidates.

1731 Matches

If you press Tab while typing the second word in a line, or later – as shown in the fourth command line in Figure 1, the completion mechanism will assume that you want to specify a filename for the *ls* command. Unfortunately, options are so numerous that Bash says nothing when you press Tab once, except to ask you if you really want to see all 1731 options (line 5 of the output) should you be so bold as to press Tab again.

If you enter more letters, as in lines 6 and 7, and thus reduce the number of options, pressing Tab twice will give you a meaningful list of completions. Bash will not actually complete the line until it has an unambiguous answer.

Partial completion also happens. If a user types */etc/up*, as in line 8, and then presses Tab once, the shell immediately completes as far as */etc/update-*, although two options – *update-manager* and *update-notifier* – could fullfil the request. Such partial completion helpfully gives you the path up to the point that your entry becomes ambiguous.

The *complete* command documented in "Programmable Completion" on the Bash man page (*man bash*) lets shell users willing to invest some effort in programming enhance the standard repertoire. The Bash Completion project [1] offers a huge collection of completion rules that you can add to your local *.bashrc* file.

This brain implant for Bash adds command-specific rules and completion for command options. For example, if you enter *git com* and press the Tab key, the mechanism will expand this to *git com-*

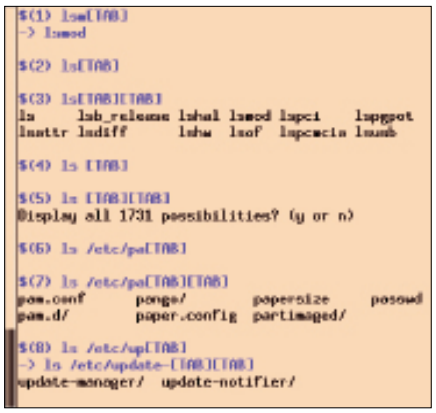

Figure 1: Bash normally completes commands and names of files to be processed when the user presses the Tab key.

mit, because this is the only subcommand starting with *com* for the version control tool. For more details on the complete function, check out the material in some books on Bash [2] [3].

Although the collection of scripts in the Bash Completion project builds on Bash functions, experienced Perl programmers will be aware that shell scripts – although quickly programmed – often turn out to be dead ends, mainly because the Shell language is rather limited and doesn't provide powerful mechanisms for abstraction. Programmers are better off implementing some shell scripts in a more complete scripting language like Perl, because sooner or later they are going to rewrite the whole enchilada when things go beyond the prototype stage.

Do It Yourself

For incomplete command-line entries, you can just as easily generate the suggestions that Bash returns with a Perl script. For example, if you have a

```
complete -C helper command
```

directive in your *.bashrc* where the shell can parse it on launching, the shell will always rely on the *helper* program for suggestions concerning the *command*.

When a user types *command* (followed by a blank) and presses the Tab key, Bash will call the Perl *helper* script and pass the *COMP_LINE* and *COMP_POINT* environment variables to it. This gives the helper script the command line up to that point and the cursor position when the user pressed the Tab key. Additionally, the arguments the helper receives (from *@ARGV* in Perl) are the first word in the line (typically the command), the word to be completed, and, as a third argument, the previous word. The shell expects the helper to output a number of suggestions separated by line breaks to stdout.

The shell session in Figure 2 defines the *complete-dump* script as a helper for the *ls* command. As you can see from Listing 1, this experimental helper script only returns the content of the *COMP_LINE* and *COMP_POINT* environment variables and the *@ARGV* array as the stderr output. It doesn't return anything on stdout, and the completion mechanism thus makes no suggestions.

The output shows that Bash passes the command line, including all blanks, to the helper script in the *COMP_LINE* variable. It returns the cursor position in *COMP_POINT* along with the *COMP_LINE* because the user could press the cursor keys to edit the command line and then suddenly press Tab in the middle of a command, although this is probably of little practical use. Under normal circumstances, *COMP_POINT* will be exactly the length of the string in *COMP_LINE*.

Line 3 attempts to complete the first argument for the *ls* command and receives *ls* (the first word), */etc/p* (the word to complete) and *ls* again (the previous word) as arguments in *@ARGV*. In the fourth case, in which it wants to complete the second argument for *ls*, a

Listing 1: complete-dump

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use Data::Dump qw(dump);
04
05 my %matches = ();
06
07 for my $env_var (keys %ENV) {
08  next if
09      $env_var !~ /^COMP_/;
10
11  $matches{$env_var} =
12    $ENV{$env_var};
13 }
14
15 $matches{ARGV} = \@ARGV;
16
17 print STDERR "\n",
18   dump(\%matches);
```

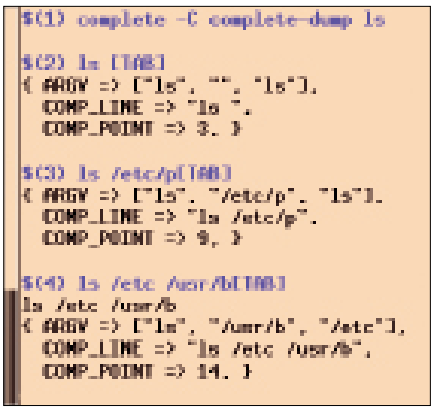

Figure 2: The shell complete command uses the -C option to assign the complete-dump script to the ls command, which the shell then calls as a completion helper.

third helper argument is returned: */etc* – again this is the word before the word to be completed.

Built-In Helper

Even self-programmed scripts can serve as helper functions. The following command

```
complete -C myscript myscript
```

tells Bash to ask *myscript* itself for help if a user presses the Tab key after typing *myscript* followed by a blank. The script *myscript* then checks whether *COMP_LINE* is set and, if so, returns some suggestions; otherwise, it executes normally.

Of course, this is a balancing act. A programming error in the script could trigger a destructive function even though the user hasn't even entered a command and is just waiting for a suggestion. The CPAN Getopt::Complete module, which gives scripts an elegant approach to completing their own options, thus suggests a conservative solution of only allowing the script to enter Perl's compile phase in helper mode with *perl -c myscript 2>/dev/null* rather

Listing 2: getopt-complete

```
1 #!/usr/local/bin/perl -w
2 use strict;
3
4 use Getopt::Complete(
5  'bgcolor' => [
6   'red', 'blue', 'green'
7  ],
8 );
```

than actually running the script [4] for completion. Listing 2 provides a short example that uses this neat module. The script offers the *--bgcolor* option for setting the background color and accepts three color values. If the user calls *complete -C getopt-complete getopt-complete*, the shell will complete not only the color values but also the option names:

```
$ getopt-complete [TAB]
-> getopt-complete --bgcolor=
$ getopt-complete --bgcolor=r[TAB]
-> getopt-complete --bgcolor=red
```

Compiled programs that you can't rewrite to add completion functions need an external helper. For example, Listing 3 shows an example in which a user, who obviously wants to run *git clone* to clone a Git repository, is given a list of all his repositories on github.com as a suggestion.

Tried and Trusted

Unfortunately, if the helper doesn't find anything useful, the shell's default completion file matching mechanism also fails. This eventuality is ugly – imagine a user typing *git add* and waiting for the shell to suggest files as potential candidates. It won't get any, because *github-helper* (Listing 3) doesn't have any suggestions for this case. The *-o default* option for the *complete* command resolves this problem by reverting to the shell's own completion mechanism if the helper has nothing to offer. Thus, adding the line

```
complete -C github-helper ➥
   -o default git
```

to your *.bashrc* solves this problem. If you also want Bash to take any completions defined in the Getopt::Complete module [4] into consideration, you can add *-o bashdefault*. If the user calls a program by its full pathname (i.e., */usr/bin/git* instead of just *git*), the shell first looks for a completion entry for the full

Mike: Getopt::Complete module OK as meant by "defined in [4]"?? -rls

Listing 3: github-helper

```
001 #!/usr/local/bin/perl -w
002 ############################
003 # github-helper -
004 #       Complete github repos
005 # Mike Schilli, 2010
006 # (m@perlmeister.com)
007 ############################
008 use strict;
009 use Pod::Usage;
010 use LWP::UserAgent;
011 use XML::Simple;
012
013 my $netloc =
014   'git@github.com';
015 my $user = 'mschilli';
016
017 if (!defined $ENV{COMP_LINE})
018 {
019  pod2usage(
020   "COMP_LINE missing");
021 }
022
023 my ($git, $clone, $args) =
024   split /\s+/,
025         $ENV{COMP_LINE}, 3;
026
027 $args = ""
028   unless defined $args;
029
030 if (!defined $clone
031  or $clone ne "clone") {
032
033  # Only 'clone' suggestions
034  exit(0);
035 }
036
037 if ($ARGV[2] ne "clone") {
038
039  # Do nothing unless clone
040  exit 0;
041 }
042
043  # Two pseudo choices
044 if (!length $args) {
045  for (1 .. 2) {
046   print "$netloc/$user/$_\n";
047  }
048  exit 0;
049 }
050
051 for my $repo (
052      remote_repos($user))
053 {
054  my $remote =
055    "$netloc/$user/$repo";
056
057  if ( $args eq substr(
058         $remote, 0,
059         length $args
060   )) {
061   print "$remote\n";
062  }
063 }
064
065 ############################
066 sub remote_repos {
067 ############################
068  my ($user) = @_;
069
070  my @repos = ();
071
072  my $ua =
073    LWP::UserAgent->new();
074
075  my $resp =
076    $ua->get(
077      "http://github.com/api/".
078      "v1/xml/$user"
079    );
080
081  if ($resp->is_error) {
082   die "API fetch failed: ",
083     $resp->message();
084  }
085
086  my $xml =
087    XMLin(
088   $resp->decoded_content());
089
090  for my $repo (
091   keys %{
092    $xml->{repositories}
093      ->{repository} })
094  {
095   push @repos, $repo;
096  }
097
098  return @repos;
099 }
100
101 __END__
102
103 =head1 NAME
104
105 github-helper - Complete github
    repos
106
107 =head1 SYNOPSIS
108
109 COMP_LINE=... github-helper
```

Mike: "Complete" OK with cap (in the "If the *git* subcommand" paragraph)?? -rls

path and, if it doesn't find one, reverts to the program name (i.e., *git*). The entry generated above will thus work in both cases.

If a user calls the *github-helper* without having the *COMP_LINE* environment variable set, the script quits by calling the CPAN Pod::Usage module's *pod2usage()* function to output the POD documentation. Otherwise, line 24 breaks down the command-line string entry into a maximum of three parts separated by blanks.

If the *git* subcommand happens to be, say, *add* rather than *clone*, the script will quit in line 34 without any output and tell the Complete mechanism that it can't help. Instead, the Complete mechanism will then use completion functions defined elsewhere. Line 37 checks to see whether the user entry really is positioned after the word *clone* (with a blank) or whether the cursor is still positioned directly behind *clone* (without a blank).

To hurry the shell into immediately writing *git@github.com/mschilli* without contacting the Github server when the user presses Tab after *git clone* (followed by a blank), the script relies on a trick in line 45: It outputs two pseudo-repositories:

```
git@github.com/mschilli/1
git@github.com/mschilli/2
```

and the shell immediately performs partial completion up to the largest common denominator, as you can see in line 2 of Figure 3. If two further tabs occur, the user must be interested in the remote repositories on the server, and line 52 calls *remote_repos()*.

Querying the Github Web API

To find the repositories belonging to a specific Github user, the script issues a request to the Github server's web API [5] in the *remote_repos()* function (lines 66-99). You can do this without logging in and using a very intuitive interface that returns data formatted in either XML or JSON.

The CPAN XML::Simple module and the *XMLin()* function it exports help *github-helper* accept the XML string that the web API call returns and convert it into a Perl data structure. The hash entry {*repositories*}->{*repository*} in this structure comprises keys containing the user's repository names. Perl's *keys()* function returns them as a list, and the *for* loop in lines 90-96 bundles them into the *@repos* array, which the function finally returns to the calling program.

The third command in Figure 3 shows how pressing the Tab key twice after the clone command gives you a choice of all available repository paths. If the user then types two more letters, as shown in the final line, thus making the selection unique, the shell will complete the results when you press Tab again, and you only need to press Enter to start the cloning process.

```
$(1) git clone[TAB]

$(2) git clone [TAB]
  -> git clone git@github.com/mschilli/

$(3) git clone git@github.com/mschilli/[TAB][TAB]
git@github.com/mschilli/app-daemon
git@github.com/mschilli/bot-wootoff
git@github.com/mschilli/data-throttler-perl
git@github.com/mschilli/dns-zoneparse-perl
git@github.com/mschilli/google-chart
git@github.com/mschilli/libuuu-perl
git@github.com/mschilli/log4perl
...
git@github.com/mschilli/yaml-logic-perl

$(4) git clone git@github.com/mschilli/lo[TAB]
  -> git clone git@github.com/mschilli/log4perl
```

Figure 3: The helper script completing Git repository names.

The *if* statement in lines 57-62 of Listing 3 checks each repository it finds to see whether it matches the user input up to the number of characters typed in so far. If so, the *print()* command in line 61 will output the full repository string, followed by a newline character, to the script's standard output, from where the completion mechanism then gobbles it up.

Fortunately, repository names don't contain any non-standard characters that could irritate the shell. Otherwise, you would need to escape all the results to protect them from being expanded by the shell. For example, if the hit contained a blank or a dollar sign, you would want the helper script to return \ or \$ to prevent the shell from interpreting the results and confusing the completion mechanism.

Installation

Lines 14 and 15 in Listing 3 must be modified to match your needs, and you need to replace the user *mschilli* with the user nick that the script will be using to query Github.com. Of course, nothing stops you from cloning my repositories; that's what Github is here for, after all. To make sure the shell finds it, the script must be executable and installed in a directory somewhere in your *$PATH*.

After doing this, you need to add the *complete* command line shown previously to your *.bashrc* file, which you can force the current shell to reparse by entering *source .bashrc*, and which will automatically be used by any new Bash shell you call.

As always, you can install the CPAN modules XML::Simple, LWP::UserAgent, and Pod::Usage that the script needs quickly and easily in a CPAN shell or with your distribution-specific package manager.

Users who are too impatient to wait for the network lookup to complete (it takes somewhere between one and two seconds), can additionally define a cache to store the results. Note that the shell calls a new instance of the *github-helper* script every time – the script thus needs to store its data persistently on disk.

The scripts given in this article [6] are designed only as simple examples of what you can achieve with Bash completion; the possibilities are endless, and Unix programmers are well known for wanting to save typing wherever they can. I'll leave it up to your imagination to save time with Bash completion! ■

INFO

[1] Bash Completion homepage: *http://bash-completion.alioth.debian.org*

[2] Vossen, JP, and Cameron Newham, *Bash Cookbook*. O'Reilly, 2007.

[3] Kiddle, Oliver, Jerry Peek, and Peter Stephenson, *From Bash to Z Shell: Conquering the Command Line*. Apress, 2004.

[4] CPAN Getopt::Complete module: *http://search.cpan.org/dist/Getopt-Complete/*

[5] Github API: *http://github.com/guides/the-github-api*

[6] Listings for this article: *ftp://ftp.linux-magazin.com/pub/listings/magazine/114/Perl/*