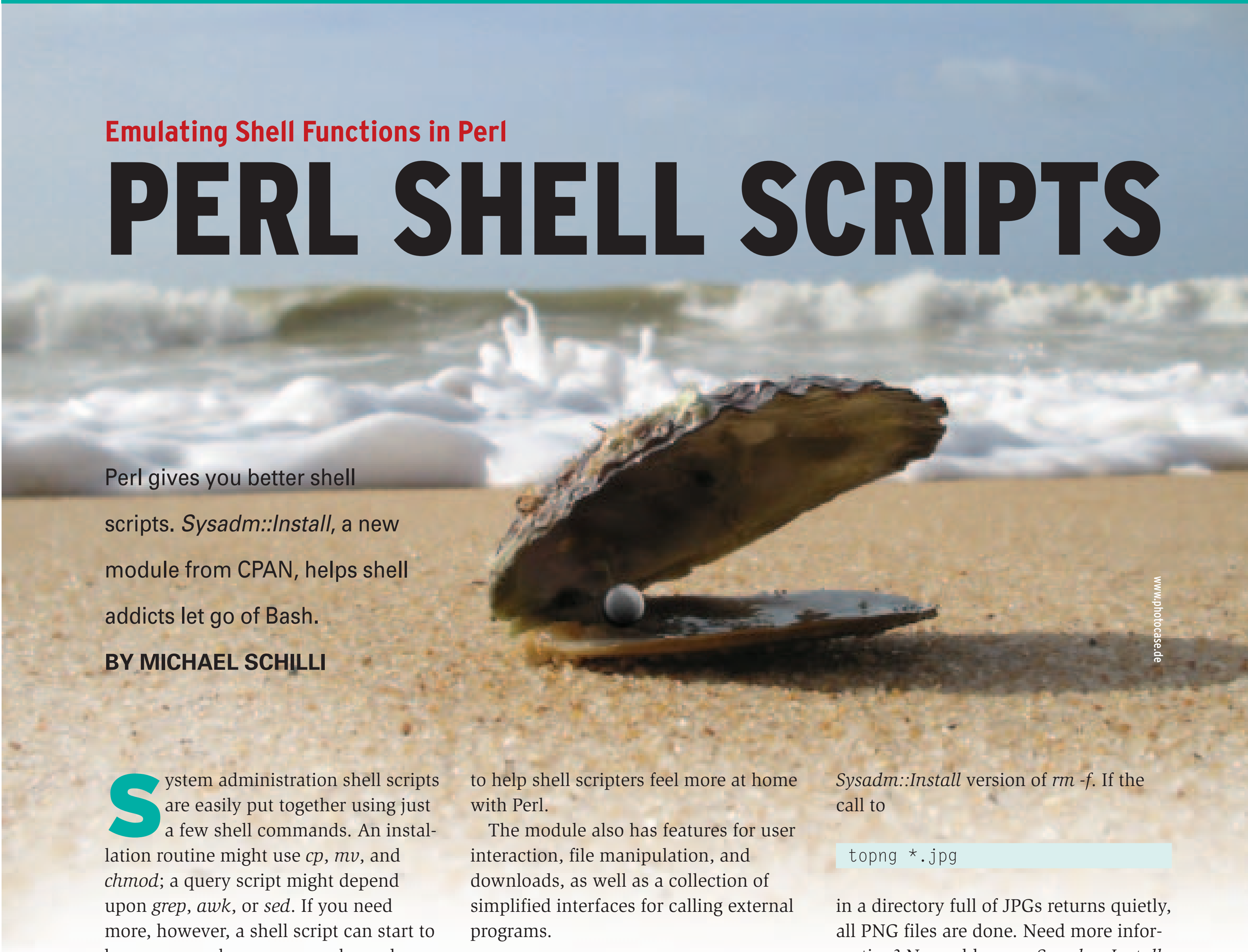


Emulating Shell Functions in Perl

PERL SHELL SCRIPTS

Perl gives you better shell scripts. *Sysadm::Install*, a new module from CPAN, helps shell addicts let go of Bash.

BY MICHAEL SCHILLI

System administration shell scripts are easily put together using just a few shell commands. An installation routine might use *cp*, *mv*, and *chmod*; a query script might depend upon *grep*, *awk*, or *sed*. If you need more, however, a shell script can start to become complex, or even ugly; and more times than not, you come to a dead end. The shell takes a very roundabout route to get to some places, and there are other places a shell script just can't go.

Creative programmers will always find an (admittedly obscure) way to work around the obstacles, and shells such as Bash, Ksh, and Tcsh have many features found in genuine programming languages. Also, if you prefer to avoid typing, standard Perl is not recommended for simple shell tasks. After all, who wants to have to enter something like the following:

```
open FILE, "<filename" or
  die "Cannot open filename
  ($!)";
```

when a simple *cat filename* would open a file for processing in a subsequent pipe? Enter *Sysadm::Install*, a new CPAN module that exports functions such as *cp*, *mv*, *untar*, *mkd*, *rmf* (rm -f), and *cd* to help shell scripters feel more at home with Perl.

The module also has features for user interaction, file manipulation, and downloads, as well as a collection of simplified interfaces for calling external programs.

Transformer

topng in Listing 1 is a script that calls the *convert* tool to convert a group of JPG images to PNG. It uses *use Sysadm::Install qw(:all)* to import all available functions into the current namespace before going on to use two of them: *sysrun()* to call an external program, and *rmf()*, which is the *Sysadm::Install* version of *rm -f*. If the call to

```
topng *.jpg
```

in a directory full of JPGs returns quietly, all PNG files are done. Need more information? No problem: as *Sysadm::Install* supports the *Log::Log4perl* system, we can simply add

```
use Log::Log4perl qw(:easy);
Log::Log4perl->➘
easy_init($DEBUG);
```

to the start of the script to make it more talkative:

```
2004/12/03 23:25:20 sysrun:➘
convert 1.jpg 1.png
2004/12/03 23:25:32 rmf 1.jpg
2004/12/03 23:25:32 sysrun:➘
convert 2.jpg 2.png
2004/12/03 23:25:44 rmf 2.jpg
```

But this is not the only reason why the script uses *sysrun()* and *rmf()* from the *Sysadm::Install* treasure trove, rather than the equivalent standard Perl functions *system()* and *unlink()*. *Sysrun()* and *rmf()* automatically use run-or-die mode, as do all *Sysadm::Install* functions. Under the hood, the results are

Listing 1: *topng*

```
#!/usr/bin/perl -w
################################
# topng -- convert jpgs->png
################################
use Sysadm::Install qw(:all);

for $jpg (@ARGV) {
  ($png = $jpg) =~
                s/jpg$/png/;
  sysrun "convert",
         $jpg, $png;
  rmf $jpg;
}
```

carefully checked, and in case of error, a call to *die()* automatically interrupts the script. Shell programmers who previously needed to type in something like the following:

```
cp a b || exit 1
mv c d || exit 1
```

can now relax. *topng* doesn't even make use of perl's *strict* mode, which is otherwise used religiously in the world of Perl. This is a quick-and-dirty script that does not try to deny its heritage.

Elegantly Unpacked

If you do not use *tar* on a daily basis, you may not always be able to recall the correct syntax for a specific *tar* command. This memory lapse could be fatal. Imagine, for instance, using the *cf* option instead of *xf*. This unfortunate mistake would overwrite the archive instead of unpacking it. Also, some packagers forget to add a top-level directory. If this happens, the *tar* archive will contain a whole bundle of files that will clutter up your directory when you unpack them. *untar()* prevents these problems from happening. The *untar()* script expects the name of a *tar* file, discovers whether the file needs to be decompressed, and then unpacks the contents of the archive. If the archive is the free-for-all type that does not give you a top-level directory, the tool creates a directory and stores the archive content in that directory.

Listing 2: *untar*

```
#!/usr/bin/perl -w
#################################
# untar -- Untar tarballs
#################################
use strict;

use Log::Log4perl qw(:easy);
use Getopt::Std;
use Sysadm::Install 'untar';

getopts('v', \my %opts);

Log::Log4perl->easy_init(
 $opts{v} ? $DEBUG : $ERROR);

for my $tar (@ARGV) {
   untar($tar);
}
```

Listing 3: *mkperl*

```
#!/usr/bin/perl -w
#################################
# mkperl - Download the
#   latest stable perl,
#   configure and install it.
#################################
use strict;

use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init(
                     $DEBUG);

use Sysadm::Install qw(
  download hammer untar
  cd sysrun);

download
  "http://www.perl.com/" .
  "CPAN/src/stable.tar.gz";

untar "stable.tar.gz";
cd "stable";

hammer("./Configure", "-d",
 "-D", "prefix=/home/" .
      "mschilli/PERL-test");

sysrun("make install");
```

The sample *untar* script in Listing 2 is simply called with the name of the *tar* archive:

```
untar pari-2.1.4.tgz
```

Since *pari-2.1.4.tgz* properly contains a top directory, this will just unpack the contents into *pari-2.1.4*. But it's comforting to know that we'd be covered in either case!

If you prefer information on the proceedings rather than silence, you can set the *-v* flag for more details on what is going on under the hood.

If I had a Hammer: Perl on Your Coffee Break

Some install scripts prompt users for interactive input. In most cases you simply need to press the Enter key, and this is exactly what the *hammer()* function does. Perl-Build is a typical example. You download the *tarball* from *perl.com*, unpack the archive, go to the top directory, launch *./Configure*, and you are bombarded with a plethora of questions. If you are sure that the defaults will be fine for your machine, you can either set the *-d* flag or keep pressing enter.

The *mkperl* script in Listing 3 *download()*s the current stable Perl tarball from perl.com, runs *untar()* to unpack it, configures the release, and launches into the build. It uses the *-d* option of *Configure* to cut down on the noise, but runs *hammer()* on the last prompt to brush off the configurator's last question on editing the configuration file manually.

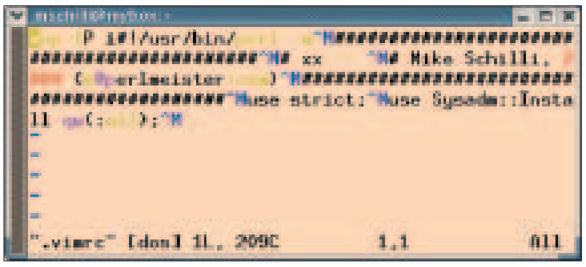
User Input

Some scripts need the user to confirm that he or she is happy with the default, or to select one of the five options on display. *Sysadm::Install* gives you the *ask* and *pick* functions for this task. *ask* prompts the user to accept the default text or enter another. *pick* shows the user an enumerated list of options and prompts the user to select the number corresponding with a specific option.

The script in Listing 4 *input* first gets a text string from the user then gives the user a choice of three options:

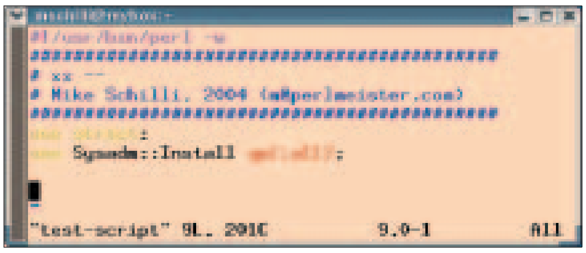
```
Name ↵
[No-Name-Entered]> Bill Gates
  Name: Bill Gates

[1] 0-100K
[2] 100K-200K
[3] 300K-
Salary [1]> 3
  Salary: 300K-
```

The values returned by *ask* and *pick* correspond to the entry or the selected



```
imap !P i#!/usr/bin/perl -w^M################################
########################^M# xx   ^M# Mike Schilli,
     (m@perlmeister.com)^M#################################
######################^Muse strict;^Muse Sysadm::Insta
ll qw(:all);^M
~
~
~
~
~
".vimrc" [dos] 1L, 209C        1,1           All
```

Figure 1: This vim macro maps the initial lines of a Perl script to the [!]+[P] key combination and toggles the editor to insert mode.



```
#!/usr/bin/perl -w
############################################
# xx --
# Mike Schilli, 2004 (m@perlmeister.com)
############################################
use strict;
use Sysadm::Install qw(:all);

~
"test-script" 9L, 201C                9,0-1            All
```

Figure 2: The results of the macro from Figure 1. Script authors can press [!]+[P] and get down to writing their scrips without further ado.

value. In this case, the values are "*Bill Gates*" and *300K-*.

Backslashitis

If you make frequent use of Perl one-liners, you may be familiar with the problem of escaping Perl code in the command line to prevent the shell from eating it up. For example,

```
perl -e "print "Hi!\n""
```

will not work, as the shell will mangle the internal quotes and the backslash, and the exclamation mark will dig up a command from the history list. However, you could escape the endangered characters using a backslash (not forgetting to escape the backslash with yet another backslash, of course):

```
perl -e "print
\"Hi\!\\n\""
```

Single quotes are an alternative, but then this stops the shell from substituting variables, and single quotes again need to be escaped in the code. This gets worse if you need to run the command on a remote machine, rather than locally, using the *ssh -t* host command. This means escaping any non-standard characters (including the special characters and escape characters you escaped previously!) This can lead to a severe attack of backslashitis:

```
ssh -t somehost "perl -e ⮐
\"print \\\"Hi\\\!\\\\n\\\"\""
```

Confused? Thought so. But there is a way of avoiding this using the *qquote()* function that *Sysadm::Install* exports to add double quotes to your string and escape any quotes and backslashes contained in the string. If the second para-

Listing 4: *input*

```
#!/usr/bin/perl -w
##############################
# input -- ask() and pick()
##############################
use strict;

use Sysadm::Install qw(:all);

my $name = ask "Name",
           "No-Name-Entered";

print "  Name: $name\n";

my $salary = pick "Salary",
   ["0-100K", "100K-200K",
    "300K-"], 1;

print "  Salary: $salary\n";
```

Listing 5: *fixinittab*

```
#!/usr/bin/perl
#################################
# fixinittab
#################################
use Sysadm::Install qw(:all);

$file = "/etc/inittab";
$data = slurp $file;
$data =~ s/id:5:initdefault:/
            id:3:initdefault:/x;
blurt $data, $file;
```

meter you specify is *:shell*, *qquote()* extends this protection to the dangerous dollar sign, the explosive exclamation mark, and the big bad backquote.

Machine Escapism

Starting in line 10, Listing 7 *ips* defines a script that runs the *ifconfig* command and extracts the IP addresses of all network interfaces. Line 18 strips the newlines and superfluous blanks out of the script text, and in line 21 the *qquote()* function formulates a compact double-quoted string that the script then appends to *perl -e*.

Line 23 adds another round of *quote()* for the SSH command, before *system()* finally runs the following command to collect the IP addresses from *somehost* without you needing to install a script on that machine:

```
ssh -t somehost "perl -e \" ➥
\\\$data = \\\`ifconfig\\\`; ➥
while(\\\$data =~ /inet addr:➥
(\\\\S+)/g) { print ➥
\\\"\\\$1\\\\n\\\"; } \""
```

Of course, you could just as easily return the *ifconfig* results to your local host and continue the process there, but a mobile script can be a more practical approach

Listing 6: *fixinittab-pie*

```
#!/usr/bin/perl
#################################
# fixinittab-pie
#################################
use Sysadm::Install qw(:all);

pie(sub {
   s/id:5:initdefault
    /id:3:initdefault/gx; $_;
   }, "/etc/inittab");
```

Listing 7: *ips*

```
01 #!/usr/bin/perl -w
02 #################################
03 # ips -- run a script on a
04 #        remote machine
05 #################################
06 use strict;
07
08 use Sysadm::Install 'qquote';
09
10 my $script = q{
11     $data = `ifconfig`;
12     while($data =~
13         /inet addr:(\S+)/g) {
14          print "$1\n";
15      }
16 };
17
18 $script =~ s/\s+/ /g;
19
20 my $cmd = "perl -e " .
21     qquote($script, ":shell");
22
23 $cmd = "ssh -t somehost " .
24       qquote($cmd, ":shell");
25
26 system($cmd);
```

if you need to handle large amounts of data.

Slurp

Scripts often need the whole set of data from a file. Perl 6 will address this problem by providing a *slurp()* function. *Sysadm::Install* already has the useful *slurp()* function, along with the opposite function, *blurt()*, which has the effect of moving stored data back into a file in one fell swoop.

For example, if you wish to tell Linux not to enter the X GUI after booting but to display a text mode login for the user instead, you need to modify */etc/inittab* by changing the 5 in the following line to a 3:

```
id:5:initdefault:
```

slurp() and *blurt()* make this change quite simple, as Listing 5 (*fixinittab*) shows.

You can make this even more compact, as Listing 6 demonstrates. In a similar way to Perl's inline edit mode (*perl -p -i -e "..."*), *Sysadm::Install* gives you a *pie()* function. The *pie()* function expects at least two arguments: a reference to user defined callback, and one or more file names. *pie()* parses the files specified with the command one by one, calls the callback for each line, and replaces the line with the value returned by the function. After completing these changes, *pie()* writes the results back to the original file(s).

If you use the substitution operator, remember that *s/a/b/* does not return the resulting string but the number of replacements. If the callback is a simple substitution function, *s/a/b/; $_;* makes sure the resulting string is substituted back into the file.

Tired of Typing?

Finally, a tip for those who want to cut down on keyboard penetration. Instead of typing *#!/usr/bin/perl* and *use Sysadm::Install qw(:all)*, you can simply define a *vim* macro, as shown in Figure 1. The *vim* macro assigns an insert command, followed by the first seven lines of a *Sysadm::Install* script, to the *!P* (P for Perl) key combination in *vi*'s command mode: a Perl-Shebang line, some dressing, a template for the script name, what the script does, and your name. In *.vimrc*, the command has to be written in a single line. The newlines, which are represented by *^M* later, are created by pressing the [CTRL] + [V] key combination, followed by [Return], in Vim input mode.

You can simply type *vi test-script* to start a new script. [!] + [P] adds the header and switches *vi* to insert mode. ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/52/Perl*

[2] Sysadm::Install: *http://search.cpan.org/~mschilli/Sysadm-Install-0.09/*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.