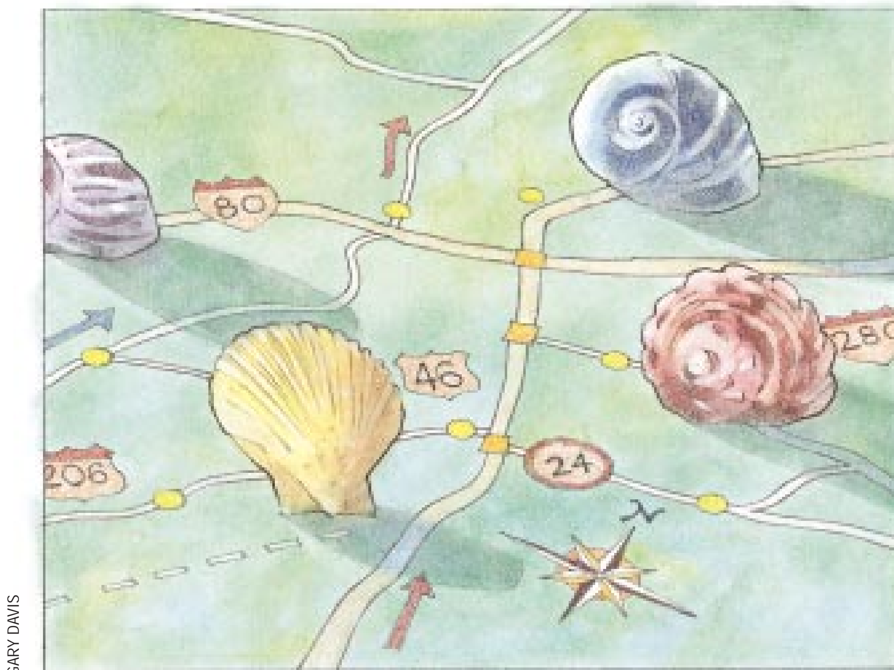
by Peter Collinson, Hillside Systems



GARY DAVIS

A Shell Road Map

One of the early design requirements for UNIX was the need to generate a small kernel, the memory-resident part of the operating system that interfaces the user's applications to the hardware. There were pragmatic reasons for smallness. The kernel had to fit completely into the 32-KB 16-bit words of kernel memory address space afforded by the PDP-11. It had to fit and also leave enough memory space available so that the user could run programs. Small was beautiful.

To reduce the amount of code in the kernel, many aspects of the "operating system" were moved into user processes. When I first encountered UNIX, some 20 years ago, it was novel to see that the contents of `/bin` were a set of commands whose names you typed to get your work done. Likewise, it was novel to see that `/bin/sh`, your command interpreter, was not a protected part of the operating system. It was just another user-level program whose job was to take typed information and launch processes.

Small was beautiful in the shell too. The part of the command interpreter that performed filename expansion–changing `*` to a list of filenames–was actually a separate program called `/etc/glob`. The shell scanned the input line, and if filename expansion was needed, `/etc/glob` was executed to do the job.

A consequence of the use of programs sitting in separate files for what are considered to be parts of the operating system is the ability to replace those standard utilities with alternatives. Of course, you didn't want to physically replace, say, the standard `rm` command, but you may want some users to use one version while others see the standard one.

Providing personalized utilities was made easier when UNIX Version 7 appeared. Its kernel introduced the per-process environment strings whose contents are retained when a process uses the `fork()` system call to create a new process. One of the standard environment strings is the `PATH` variable. It provides an ordered list of places to search for the file containing the binary program that matches the command name typed by the user on the input line. The use of the `PATH` variable means that by controlling the search path, it's possible to place alternative utilities *in front of* existing ones for some users. Also, it is now simple for users to include a private directory in their search path (often called `bin`) that contains private copies of standard commands.

It didn't take much time before alternatives to the standard `sh` program were created, providing different user interfaces to the system. We now have a plethora of different shells to choose from; most derive from either the Bourne shell (`/bin/sh`) or the C shell (`/bin/csh`).

The Bourne Shell

UNIX Version 7 was released in 1978 sporting a brand-new shell that was significantly different from the Version 6 shell. It has since become known as the Bourne shell, after its author. Steve Bourne had arrived at Bell Labs directly

from Cambridge University in the United Kingdom and was an Algol 68 aficionado. The syntax of this language found its way into the control structures of his shell. The shell itself was coded using an Algol 68-like language that used the C preprocessor to convert its source into C.

As I said, the shell made use of various new features of the system and also made its own impact on the kernel design. The shell treated commands as subroutines and could test their success or failure, meaning that each command had to be recoded to return a result when it exited. There's a story that when Bourne first developed his shell, he made it print a rude message when a command did not return a valid return value. Soon people were embarrassed into converting their programs. I think the ability to detect the success or failure of commands has made UNIX scripts significantly more useful, so those peoples' red faces were worth it.

The Bourne shell was (and is) a fully functional programming language using text replacement ideas that were prevalent in macro processors. The language comes with a full set of program control structures, allowing tests and loops. In the early '80s, when UNIX System V was released, the shell was extended to include *shell functions*, allowing the user to collect commonly used command sequences into private shell built-in commands. Shell functions also made it easier to write complex scripts, allowing the programmer to split the task into small chunks.

The C Shell

The names for UNIX commands are littered with puns (usually bad) and jokes (usually obscure), and the C shell, csh, is no exception. I expect you to sea that. Csh was created by Bill Joy when he was at the University of California at Berkeley. I think he started at about the same time that Bourne was making his shell. Joy took the shell's programming syntax from the C language.

The shell was groundbreaking because it contained many features that were executed in the shell itself. For example, csh has the ability to perform arithmetic in the shell. It has aliases for command names, allowing the user to create command names without recourse to a specific command file. Many people use aliases

to add preferred switches into commands. For example,

```
alias rm 'rm -i'
```

adds the interactive switch to every invocation of the rm command made from the shell.

Csh also contains many built-in commands. For example, echo is the shell's print statement and is used considerably. The code for the command is resident csh, making it much faster to print messages.

Testing the values of variables is another very common operation, and csh evaluates the values internally. So a csh statement like this:

```
if ($var == "hello") echo yes
```

would execute quickly because the whole line is interpreted in the shell.

The original Bourne shell relied on executing commands in /bin to perform the equivalent task:

```
if test $var = "hello"
then
       echo yes
fi
```

The test and echo commands lived in /bin, so by the time you had executed the test, you had also created two new processes. Incidentally, in Bourne shell, you can write the test using square bracket syntax:

```
if [ $var = "hello" ]
then
       echo yes
fi
```

The opening square bracket was originally a command in /bin (a link to the test command). The shell checks for a closing square bracket and removes it from the invocation of the command. You won't find a command called [on your system today. The command echo and variable test have become built-ins in the System V version of the Bourne shell, following csh's lead.

I first started to use csh when we switched to 4.1BSD in around 1981. There were two reasons: First, csh had implemented command history; for example, typing

```
% !!
```

would execute the last command, and

```
% !m
```

would execute the last command starting with the character m. As a perpetually dreadful typist, history substitution makes my work go faster.

To stop people from doing dumb things and changing their shell to something that was not a legal program, the passwd program used the /etc/shells file as a reference source to determine if the shell that the user selected was sensible.

Second, 4.1BSD came with job control that gave us the ability to manipulate background and foreground processes. Typing Control-Z puts the current foreground job into the background. The shell notices this and presents you with a prompt so you can start a new command. There are various commands built into the shell that allow you to control the background processes. Actually, job control was my sole reason for switching to 4.1BSD, and because it was not supported by the extant Bourne shell, you had to use csh as your primary interface to the system.

I still find myself using job control even though I obviously run an X Window system. It's still quicker to type Control-Z, suspending the current job on the desktop, than it is to locate the mouse and click here and there on the screen.

Changing Shells

When csh appeared on the BSD system, UNIX suddenly had a choice of two real shells providing different interfaces for the user. Ideally, you want to run one shell or the other when you log in. The UNIX password file format has always contained a field that tells the login program which shell is to be started for that user when they first connect to the system. So using an alternative shell is not a problem.

The Berkeley system handed this choice directly to the user by supplying a utility that could be used to change the shell field in your password file entry. The commands were actually links to the passwd program, and options to the passwd command could be used to perform the edit too.

To stop people from doing dumb things and changing their shell to something that was not a legal program, the passwd program used the /etc/shells file as a reference source to determine if the shell that the user selected was sensible. This file has subsequently taken on other uses, mostly in checking that a user coming in with FTP has a legal shell.

The chsh command can be still be found in SunOS, and you can easily select your own shell. Things have changed a bit on Solaris. The explicit commands have disappeared, and you can change your shell using options to the passwd command, but only if your site uses NIS or NIS+. You need to discuss the situation with your systems administrator if the password file exists on your machine. This may not be a bad thing. Changing shells can be complicated because they use start-up files that will need tailoring for your environment.

Once a system had more than one viable shell, then users were able to write scripts in either. They wanted to put the script in a file, turn on the execute bit and use that file as a standard command. However, in the early systems, a problem arose of exactly what syntax the file contained. The file needed to be passed to an appropriate interpreter to execute its commands. Early systems used a bit of hackery. If the first character of a file was #, introducing a comment in csh, then the script was passed to csh; otherwise the script was executed by the Bourne shell.

Later, a more general-purpose approach was adopted. If the file starts with the sequence #!, then the remainder

of the line is taken to be the full pathname of an interpreter to which the script is passed. This solution permits users to write scripts for an arbitrary interpreter, which may or may not be a shell. So, for example, you'll see that Perl scripts will start with something like this:

```
#!/usr/local/bin/perl -w
```

The Korn Shell

The Korn Shell, ksh, is a development of the Bourne shell, adding many features that people liked from csh and introducing some new ones too. You do need to have Solaris to have access to the shell; it's not supplied on SunOS.

The main change in ksh as far as the terminal user is concerned is the provision of command-line editing using either `emacs` or `vi` keystroke patterns. You can edit the line that you are typing, using full visual editor capability. Also, ksh stores all the commands, so you can treat the command sequence as a file, stepping back to a command, pulling it into the current line and editing it to suit. As I found when csh was introduced, the ability to deal with your command history is a huge win. However, it was hard, and somewhat nonintuitive, to reuse bits of lines in csh. With ksh, you use a set of familiar editing keys to create just that tiny change to the line that you need. Incidentally, line editing is turned off by default. You need to say

```
set -o emacs
```

or

```
set -o vi
```

in your start-up file to ensure that it's enabled.

Ksh has several other additions, many picked up from aspects of csh. It supports integer arithmetic in the shell itself and has new syntax to aid expression evaluation. It has arrays of variables, a long-missed feature of the Bourne shell. It allows the use of tilde before a login name to mean "the home directory" of the user. It also implements aliasing, allowing the shell to store replacements for commands. A new idea uses aliasing to automatically

track commands. Once a command is found by searching the PATH list, an alias to the full pathname of that command is established, meaning that subsequent use of the command will not require directory searching. Finally, ksh supports job control.

David Korn, the author of ksh, spent many hours sitting in the POSIX committees that were working on the shell and its utilities and, as a result, some of the new features that were present in ksh became enshrined in the standard POSIX shell. Lacking some of these new features, the Bourne shell cannot fulfill the requirements of the POSIX standard. So we've seen ksh appear as an alternative shell on various systems that need to be POSIX compatible.

The POSIX committee liked ksh's alternative method of expressing the Bourne shell's backquote operator for command substitution. Let's say we want to look in a set of files for some string or other, and then put those files into an editor. Well, we know that we can use the grep command to search for a string. Also, we can make grep output a list of filenames by giving it the -l option. The output would look something like this:

```
$ grep -l lookfor *
file1
file10
file9
```

so we can see that the string lookfor was found in file1, file10 and file9. If we want to use command substitution to load the output of grep into the editing command in Bourne shell, we'd say

```
$ vi `grep -l lookfor *`
```

The backquote syntax makes the shell run the command and capture the standard output. The output of the quoted command replaces the backquoted section of the line, so the command that we would execute is this:

```
$ vi file1 file10 file9
```

The problem with the backquote syntax is that it doesn't nest. In ksh, the example above would be written as:

```
$ vi $(grep -l lookfor *)
```

Because there are start and finish markers, we can embed other commands inside the command substitution. Actually, there are still some problems, and these are highlighted on the ksh manual page.

David Korn, the author of ksh, spent many hours sitting in the POSIX committees that were working on the shell and its utilities and, as a result, some of the new features that were present in ksh became enshrined in the standard POSIX shell.

If you are using Bourne shell to talk to the machine, then switching to ksh makes sense. You might want to think a little if you use csh; the cultural shock may be too great. However, it depends on how complex your use of the shell actually is. If you just type commands and use the occasional loop, then switching will not be too difficult.

The Bourne-Again Shell

The shell that I actually use is bash from the Free Software Foundation. Of course, bash is another quirky acronym, standing for the Bourne-Again Shell. Bash is intended to be a complete implementation of a shell that complies with the POSIX standard. It basically has the feel of the Bourne and Korn shells, but with several extra features that make it considerably more usable.

Bash has command-line editing, providing emacs and vi keystrokes to edit the current line or the command history, but added to that is the ability to automatically complete commands and filenames, a feature aimed at lazy or inept typists.

When you are typing a filename, you can hit the tab key to force the shell to look in the directory for a file that starts with just the characters you have typed so far. So if your directory contains jim.txt and fred.txt, you can type

```
$ vi j<TAB>
```

and you will see the filename be completed,

```
$ vi jim.txt
```

now you can hit the return key to start the editing command. The completion will not occur unless the string you have typed is unique. If you had another file in the directory, say jane.txt, then when you hit the tab key in the example above, the shell would make the system beep at you, demanding that you create a unique string. Typing another tab will give you a list of alternatives. You then need to enter enough characters to select the appropriate file. This feature also works for command-name completion, although I rarely use that. Most UNIX commands are short anyway.

Bash also implements csh-style history invocation, which is not supported in ksh. So you can use the old !! to execute the last command that you typed. Sometimes my fingers do this; mostly they type the Control-P character that is the emacs "previous line" command, pulling the last command from the history list. However, if you are seeking the last command starting with say "m," then !m is easier and faster to type than the equivalent emacs search command.

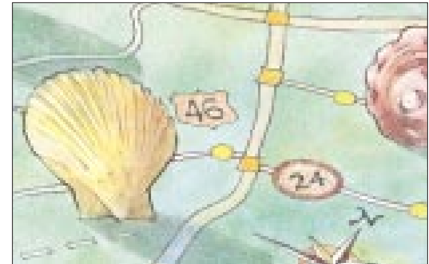
Bash has all the advantages of the Bourne shell, with powerful redirection features and sympathetic treatment of multiline quoted strings and few real annoyances.

Finally

If you are going to use bash, then probably the easiest way to obtain it is to get hold of a binary distribution for your machine. If you are running Solaris 2.5 on a SPARC, then take a look at http://smc.vnet.net/ (also named http://sunfreeware.com). These good folks have done the compilation work for you and created Sun packages that can be loaded using the standard installation facilities. The Web site says that 2.6 packages will be along soon, and

undoubtedly things will change by the time you read this.

If you have a C compiler, then it is actually very easy to install bash from the sources. Check out your nearest Free Software Foundation mirror site, or failing that use anonymous FTP to `prep.ai.mit.edu`. If you are unsure about FTP, then locate `ftp://prep.ai.mit.edu/pub/gnu` using your Web browser. This will give you a list of files, and you need to find the most recent version of bash. As I write, that file is `bash-2.01.tar.gz`. There are lots of earlier bash versions, and some files containing differences between versions, so ignore these. If you just want to look at the documentation, then `bash-doc-2.01.tar.gz` contains it in various formats.



Tcsh is a variant of the csh that supports line editing and also command completion. It's become the shell of choice for people who like csh. I've not mentioned it in full, because I've never used it. My loss, I suspect. Confirmed csh users may want to check out this shell; it seems well liked by its users. You can obtain tcsh from `ftp://ftp.gw.com/pub/unix/tcsh`.

I got the idea of writing this article from reading the excellent *UNIX Power Tools*, edited by Jerry Peek, Tim O'Reilly and Mike Loukides, published by O'Reilly and Associates Inc. The second edition of this tome has just arrived, complete with a supporting CD-ROM (ISBN 1-56592-260-3).

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever… He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.