The journal of Apple technology.

Volume Number: 15 (1999)
Issue Number: 9
Column Tag: Systems

Scripting Languages

by Rich Morin and Vicki Brown

A Cross-OS Perspective

Background

Ask most computer users (or programmers) about scripting languages and you'll get a wide variety of answers. Most users, however, will give you examples of languages they "know" are scripting languages - AppleScript, the Unix shells, Visual Basic, JavaScript, Perl, Tcl/Tk...

Ask about system programming languages and you'll be given a different set of examples. This time C, C++, Java, and/or Pascal will be mentioned. Everyone knows which languages fit which categories, but not everyone can explain why. And what about languages such as Perl, which seem to have elements of both?

Although "scripting languages" are increasingly popular, their exact definition can be elusive. Are they simply variations on "command languages"? Can they be used for "real" programming? Because of the way languages evolve, no definition we could make can be definitive, but some background notes may help to clarify the situation.

Scripting languages have their roots in the "Job Control" languages (e.g., IBM's OS JCL) that are used on "batch processing" computer systems. JCL commands tell the system to run specified programs, using particular I/O resources (e.g., files and tape drives). The "command languages" found on "time-sharing" systems (e.g., Unix and VMS) add interactive window-dressing, but perform precisely the same functions.

Unix "shells" (e.g., bash, csh, ksh, and sh), along with Apple's MPW, are strongly influenced by these command languages. As a result, users of these shells still "program" their way through the day. By issuing commands to the system, they can move files, compile programs, and more. For example, a Unix user might issue these commands:

```
% mv foo.c bar.c      # rename "foo.c" to "bar.c"
% cc -o bar bar.c      # compile "bar.c" into "bar"
% bar           # run "bar"
```

Scripts

The line between entering commands and scripting is very fine. On most modern computer systems, it is trivial to record a "script" of user commands, edit it as desired, and reuse it later to perform similar operations. A typical Unix "shell script" of this sort might look like:

```
# wl - run word count, then print (lpr) a set of files
#
# Usage: wl file1 [file2] ...
```

```
T=/tmp/wl.$$ # temporary file
wc  -l $* > $T # get line counts for files
lpr $T $* # print line counts, then files
rm  $T # remove temporary file
```

Shell scripts tend to be easy to write, understand, and modify, at least in comparison to C programs that perform the same tasks. Many of the "standard" Unix tools are implemented as shell scripts. (A quick look over our FreeBSD system found 47 scripts, a little less than 10% of the total system commands). MPW users won't find this unusual; indeed, scripts are even more prevalent on MPW. (Approximately a quarter of the commands in the basic MPW distribution are scripts.)

"Glue" Languages

Unlike most compiled languages, which make use of code libraries written in the same language, scripting languages are often used as "glue" languages. That is, they are used to glue together tools and programs which may not, themselves, be written in the scripting language.

In the example above, the wl script glues together three Unix tools: wc, lpr, and rm. Each of these may, itself, be another script; it doesn't matter. What does matter is that we've brought them together (along with a variable definition and some comments), creating a useful tool.

When Does a Script Become a Program?

OS JCL has relatively primitive support for programming. Variables can be defined and conditional execution is supported, along with a bit of modularity, but that's about it. Today's command languages, in contrast, support all sorts of programming constructs.

The Unix and MPW shells each allow conditional execution and generalized control flow (e.g., if...else, looping), some level of modularity, variable creation and evaluation, and other desirable features. As a result, rather than sticking to simple, line-by-line scripts, it is quite possible to write "real programs" in these languages. As we said before, many of the normal administrative tools found in Unix and MPW are, in fact, are scripts.

Because scripting languages are "interpreted" (rather than compiled into machine code), they can show some amazing bits of run-time flexibility. For instance, the variables ($* and $T) in the Unix shell script above could contain any sort of text strings, including data values, file names, or even shell commands!

Similarly, the "backquote" operator can be used to generate data "on the fly", as:

```
% mv 'grep -l Mac *' dir  # move "Mac" files into "dir"
```

In the Unix command above, grep searches all files in the current working directory, listing (the names of) files which contain the string "Mac". This list is then substituted back into the command line, yielding something like:

```
% mv abc bcd cde ... dir
```

If the user had to write the script this way, it would lose much of its value, as the names of the files would have to be "hard-coded" into the script. Instead, the filenames are calculated at runtime, and they will likely be different every time this script is run.

Although neither mv nor grep "knows" how to do this sort of selective file move, users are able to combine them in ways that take advantage of each command's strong points. This re-use of low-level tools is very characteristic of scripting and can be extremely powerful.

The following bit of code from an MPW Startup file combines use of backquotes with if...else statements and variable evaluation (in {braces}) to create a tidy little programmatic construct, as well as a reusable bit of code that can be shared between users.

```
if "'Exists "{PrefsFolder}"UserProjectMenu'"
 Execute "{PrefsFolder}"UserProjectMenu
else if "'Exists "{MPW}"UserProjectMenu'"
 Execute "{MPW}"UserProjectMenu
else
 ProjectMenu # the default project menu
end
```

There is no need for the script to know how a given user has his menus set up, nor even to know (ahead of time) what the path to the PrefsFolder or the MPW folder will be. As long as the pertinent variables are set by the user, this script will run correctly. More to the point, it will be flexible enough to serve the needs of different users.

AppleScript

Although users of conventional (i.e., command line) scripting languages may find AppleScript to be somewhat unusual, it does fit the model of what a scripting language can be. Under Mac OS, commands are issued, not by typing, but by selecting - menus, buttons, icons - and clicking. AppleScript allows users to save these commands in a file, suitable for executing (by double-clicking) at a later time. These files can then be annotated and extended in a manner similar to editing a shell script under Unix or MPW. Thus, it is reasonable to think of AppleScript, along with MPW and the Unix shells, as a scripting language.

The Record feature (analogous to the Unix script command) allows a user to record an exact sequence of events into a script. After recording, the user can then edit the script with a specialized Script Editor, adding variables, conditionals, and looping. Indeed, while a simple recorded AppleScript may bear little resemblance to what we think of as a program (being nothing more than a repetitive list of captured tasks), with the addition of a few variables and a loop or too, AppleScripts begin to look quite... real.

Some tasks may not be recordable, unfortunately, so you'll need to read the documentation carefully (and study examples) to get the most mileage out of your scripts. Even then, menu selection, button presses, and the like still won't be scriptable with vanilla AppleScript. If you're writing an automated script, you may not want the script to stop part way through and wait for the user to click the OK button!

Fear not - PreFab Player <http://www.prefab.com> is a commercial application created to work with AppleScript and provide access to all the controls that Apple and other application vendors left out. Specifically, Player "adds verbs that query and manipulate the Macintosh user interface, giving your scripts access to otherwise non-scriptable applications, desk accessories and control panels." This lets you include statements such as

```
do menu menu item "Page Setup" of menu "File"
```

within your scripts, removing the need for user intervention (and making your AppleScripts even more "program-like").

Advanced Scripting Languages

As useful as they are, conventional scripting languages have syntactic and other limitations, however, which tend to limit their efficiency and expressive power. For instance, even "advanced" Unix shells (e.g., bash and ksh) have very limited support for concurrency, data structuring, information hiding, object-oriented programming, regular expressions, etc.

Responding to these deficiencies, language developers have created extended scripting languages such as Perl, Python, and Tcl/Tk, among others. These languages are still able to invoke and glue together other programs, manipulate files, and set values on-the-fly, but they are also appropriate for writing much larger applications.

Perl, for example, provides compact but very powerful sets of data types (numbers, strings, and references) and structures (hashes and lists). Hashes, also known as an associative arrays, use text strings as indices. Lists, in Perl, are numerically indexable and can also act as stacks, queues, or even dequeues (double-ended queues). Perl emphasizes support for common application-oriented tasks; important feature of Perl include built-in regular expressions, "text munging", file I/O, and report generation.

Python has a great deal in common with Perl, but it emphasizes different things. Python's focus is on support for common programming methodologies such as data structure design and object-oriented programming. While such things are generally matters of opinion, Python also claims to have a more "elegant" (i.e., "less cryptic") syntactic notation.

Tcl/Tk shines at creating graphical user interfaces. A few dozen lines of Tcl/Tk can produce a very substantial collection of buttons, sliders, fill-in areas, and other widgets, nicely arranged and capable of doing calculations and/or invoking other applications. Tcl alone makes an excellent language for writing low-level system and kernel test tools; Tcl does not require a user interface, nor do Tcl scripts require the complete Operating System, with all its utilities, to be running.

In addition, through the efforts of these language enthusiasts, these languages have very substantial collections of add-on code (e.g., modules and libraries) for accessing databases, creating or parsing HTML, writing network daemons, etc. By leveraging the available add-ons, programmers can accomplish a great deal in an astonishingly small amount of new code.

System Programming Languages

Where do we draw the line between a scripting language and a system programming language? Perl, for example, is widely considered to be a scripting language, but it shares many characteristics with so-called "system" languages such as C. As an example, Perl can take advantage of sockets, fork processes (under Unix), and handle various aspects of input/output and process control "normally" reserved for "system" languages.

Perl can still be used as a glue language, but the Perl language itself has become so capable that this is no longer a requirement. In simple scripting languages, such as the Unix shell, there are only a small number of built-in commands; it's difficult to write much code without invoking another, "external" program or two. In contrast, the Perl language is so extensive (and extensible, via modules and XS code), that it is quite possible to write a complete Perl script that relies on nothing but Perl. (This is part of what makes Perl so portable).

MacPerl, for example, is a nearly complete port of Perl to the Macintosh, leaving out only those features that are specifically unsupported by the Mac OS (e.g., fork, exec, and various process and user control commands). MacPerl even sports a more extensive socket library than Unix Perl (specifically, it supports AppleTalk).

When combined with ToolServer (part of the MPW suite), MacPerl can even act as a glue language, taking advantage of backquoted commands or the system() function to call Apple application programs. MacPerl scripts can also call AppleScript (and vice versa), providing even more programming power and extensibility.

At some point, with so many readily available, easy to use (and inexpensive or free!) high-level scripting languages to choose from, why would anyone still resort to C, C++, or Java? We can think of several reasons, but execution speed and the desire to keep algorithms private are probably the dominant ones. Both of these reasons are going away, however, with increases in processor speed and the advent of compilers for traditionally interpreted scripting languages.

If you want to get down to the lowest level and work closely with the actual bits, or if you need special memory allocation or character I/O, a system programming language may still be the best choice. And, of course, you may not want to write the next great OS in a scripting language (then again, you might!). In any event, scripting languages definitely have a useful role to fill in most programmer's toolchests; if you haven't yet done so, we suggest that you give them a try!

References

MPW - Macintosh Programmers Workbench, is available for free download from Apple Computer. See http://developer.apple.com/tools/mpw-tools/index.html.

An extensive set of pointers to scripting language comparisons can be found on the Python language web site at http://www.python.org/doc/Comparisons.html.

Idiom Consulting has compiled a Catalog of Free Compilers and Interpreters, "... freely available software for ... things whose user interface is a language." The list is available at http://www.idiom.com/free-compilers.

For a comparison of Perl, Python, and Tcl/Tk (as well as another perspective on what makes a scripting language), see "Choosing a Scripting Language" in the Oct, 1997 issue of SunWorld (available at http://www.sunworld.com/swol-10-1997/swol-10-scripting.html).

Further information on Perl, Python, and Tcl/Tk can be found, respectively, at http://www.perl.com, http://www.python.org, and http://www.scriptics.com. Ports of all three languages are available for Mac OS.

A 30-year veteran of the computer industry, **Rich Morin** (rdm@ptf.com) writes the Silicon Carny column for SunExpert magazine and is a Contributing Editor for MacTech. He programs almost entirely in Perl these days, on Mac and Unixish systems. Rich is also the president of Prime Time Freeware (www.ptf.com), which publishes mixed-media (book/CD-ROM) collections of freely redistributable software. PTF's Mac-specific products include "MacPerl: Power and Ease" and "MkLinux: Microkernel Linux for the Power Macintosh".

Vicki Brown (vlb@cfcl.com) has been programming professionally since 1984. Unix is her favorite Operating System; the Mac OS is her favorite User Interface. Vicki is co-author of "MacPerl: Power and Ease" and co-host of the MacPerl feature in PerlMonth magazine. When she isn't writing, Vicki is employed as a Scientific (Perl) Programmer at a BioTech company on the San Francisco Peninsula. Her interests include programming, the World Wide Web, reading, writing, and spending time with her spouse and their cats.