Converging on Program Simplicity

Neville Holmes, University of Tasmania

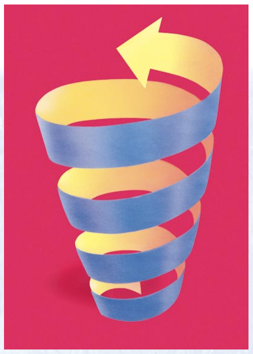
Among the letters published in the June issue of *Computer* was one that deserved a screaming headline. In it, Hank Walker proclaimed the virtues of reusability through commercial components "assembled by the programmer using a scripting language," for which he averred that "productivity increases of 10 times or more are common compared to development from scratch." This is a truth long understood by many practicing programmers, but too widely ignored by theorists and others.

When I read the immediately following report on PC-TV convergence, a bell tinkled faintly in the back of my mind. Some time later, I worked out what it was telling me: Scripting languages like those advocated by Walker must also converge if they are to succeed as they should. What, then, are they to converge with, and why?

EARLY SCRIPTING LANGUAGES

Although Walker was talking of Visual Basic and Tcl/Tk, what he described was also applicable to much older scripting languages, those used to write batch files and shell scripts, the *job control languages* that introduced program reusability. I well remember the awe with which programmers in the mid-1960s regarded the IBM/360 Job Control Language (JCL). Not only did JCL provide for files to be named outside the program itself, but—wonder of wonders—you could even switch files from cards to tape to disk without rewriting your program. Device independence, it was called; at the time, it was a giant step toward program reusability.

Guest Critic: Neville Holmes, Department of Computing, University of Tasmania, PO Box 1214, Launceston 7250, Australia; neville.holmes@utas.edu.au.



Why hasn't program reuse evolved through steadily developed scripting languages and component libraries?

Later came Unix, an operating system based on reusability. Its glory was that it provided

- lots of little utility programs designed to be cobbled together by the user,
- a powerful shell-scripting language for cobbling programs together, and
- operating system support for data to be passed simply from program to program through environment variables and pipes.

Why haven't scripting languages become more widely used, promoting the theme of reusability as started with JCL and Unix? Why hasn't program reuse evolved through steadily developed scripting languages and libraries of simple reusable components? Why have monstrous, icon-based systems triumphed over elegant, command-based ones? The answer, the bell in the back of my mind was telling me, was from lack of convergence!

DIVERGENCE IN SCRIPTING

Consider the *divergence* in Unix shell scripting. First there was the Bourne shell, which is still popular, and still the only one you can be sure of having on a Unix system. Then came the C shell, a ministep on the laudable road to convergence with the Unix language of choice, C. More recently, there has been a shift away from convergence with C in a variety of other popular shells—the Korn, Z, and Bourne-Again shells, for example—which separately strive for similarity to the original Bourne shell.

As an attempt to bring the Unix scripting language closer to a conventional programming language, the C shell was a great idea that was stillborn or at least grossly malnourished. But why, for crying out loud, should there be any difference between scripting and programming languages at all?

Perhaps some people perceive an interpreted language and a compiled language as somehow basically different. For instance, maybe interpreters need external variables transferred through pipes or sockets, while compilers need typed variables transferred through data files. Yet this notion was long ago given lie to when IBM's programmers at Hursley produced—as standard products—both a PL/I interpreter (for testing) and a compiler (for production). IBM warranted these products to produce the same result

Continued on page 134

Binary Critic

Continued from page 136

when presented with the same source program—proof indeed that compiled languages and interpreted languages are not essentially different. The error, then, must lie in seeing what differences there are as distinctions rather than as opportunities for convergence.

TOWARD CONVERGENCE

Distinguishing interpreters and compilers based on their data transfer mechanisms actually represents two distinctions. The first relates to data streams brought into, or put out from, a program. Scripts frequently use pipes for this, but conventional programs use traditional data files. The second distinction relates to named data, which scripts can easily bring in from the shell. Conventional programs bring this type of data in from a database server. These distinctions are probably the ones people most often think of as distinguishing scripting languages from conventional programming languages. What opportunities for convergence are hidden here?

Data streams

The pipes or sockets made popular by the early Unix shell are just means for passing or exchanging data streams. So are the data files favored by conventional programming languages. Any difference lies more in perception than in reality, perhaps because their time scale is usually (but not essentially) quite different.

Any difference should be transparent to a scripter or other programmer, just as the difference between a sequential file on cards or magnetic tape or magnetic disk was apparent to the System/360 programmer. Via JCL, the operating system absorbed the actual difference.

A device independence that treats pipes and sockets interchangeably with data files would promote the convergence of scripting and programming languages, simplifying both. After all, there is no essential difference between a stream of data passed from a magnetic disk and one passed from a program.

Named data

Unlike data streams, individual named values are maintained by the shell and supplied to scripts and other programs as *environment variables*. Most scripting languages allow a simple and basic use of environment variables. Programming languages do not.

But the use of named variables by a script is not fundamentally different from the use of named values requested from a database server by a traditionally written client program. Why should an operating system not provide a database of named values to any program it runs? Why should it not do so for any program running anywhere on the Internet, for that matter, whether that program is in a scripting or programming language? Why should an operating system not provide facilities to exchange named values between programs, so that these values look the same to a scripting language as they look to a programming language?

A related distinction between scripting and programming languages is their treatment of numeric values after they are brought into a script or program. Programming languages usually distinguish at least between integer and floating-point values and between the types of arithmetic carried out on them. In contrast, many interpreted languages treat the two types of numbers interchangeably within a single arithmetic. Programming for more than one type of arithmetic is an unnecessary burden, and I've proposed standardization of composite arithmetic ("Composite Arithmetic: Proposal for a New Standard," Mar. 1997, pp. 65-73), which moves decisions about the best way to represent numeric values out of the program and away from the programmer's control. A subroutine library, the operating system, or even the hardware would make these decisions.

TOWARD SIMPLICITY

The story goes that a hopeful inventor once approached Henry Ford with a gadget to attach to Ford's cars. Ford is reported to have told him to come back when he had worked out how to *remove* something from the car, rather than add to it. Both scripting and programming languages abound with distinctive features that could be removed or simplified through convergence, as would have pleased Ford.

COMPUTER
Innovative technology for computer professionals

Circulation: *Computer* (ISSN 0018-9162) is published monthly by the IEEE Computer Society. IEEE Headquarters, 345 East 47th St., New York, NY 10017-2394; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; voice (714) 821-8380; fax (714) 821-4010; IEEE Computer Society Headquarters,1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Annual subscription included in society member dues. Nonmember subscription rate available upon request. Single-copy prices: members $10.00; nonmembers $20.00. This magazine is also available in microfiche form.

Postmaster: Send undelivered copies and address changes to *Computer*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855. Periodicals Postage Paid at New York, New York, and at additional mailing offices. Canadian GST #125634188. Canada Post International Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487910. Printed in USA.

Editorial: Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *Computer* does not necessarily constitute endorsement by the IEEE or the Computer Society. All submissions are subject to editing for style, clarity, and space.

Logical and numeric values

Perhaps the most subtle, unnecessary numerical distinction lies buried deep within almost all scripting and programming languages: the separation of logical values True and False from numeric values. While True is often in theory considered equivalent to 1, and False to 0, in practice scripting and programming languages keep logical and numeric representations completely distinct. Interestingly enough, international standards require 1 and 0 to represent the closely related concepts on and off. Computer manufacturers usually observe this standard in labeling power switches.

In the name of convergence, though, programs should treat a logical 1 exactly the same as a numeric 1, and a logical 0 as a numeric 0. Ken Iverson recog-nized this long ago, and Donald Knuth recently endorsed the idea (*Am. Mathematical Monthly*, May 1992, pp. 403-422). If logical and numeric representations converged, min and max built-in functions would make the usual AND and OR functions redundant. If scripting and programming languages belatedly followed this convergence, we could remove unnecessary data types and their attendant built-in functions. Then simple array element selection (subscripting) could replace most, if not all, case and if structures—two desirable simplifications.

Element selection as a function

In addition, what possible justification is there for different notations to invoke functions and to select array elements? There is no basic difference to the operations used behind the two notations, and there are many benefits that might flow from their convergence. One such benefit, apart from the principal benefit of greater notational simplicity, would be the possibility of implying interpolation by using noninteger arguments in selecting from a numeric array.

Character sets

There is also the matter of the characters that represent programs. Most of the computing world uses either ASCII or the EBCDIC character set. ASCII is a warmed-up 7-bit code suited mainly to driving teletypewriters. It's grotesquely adapted in a variety of incompatible ways to different uses in different parts of the world. EBCDIC is nearly as grotesque. It adapts a 6-bit binary encoding of a single-case punched card character set to an 8-bit computing world that uses printers with both uppercase and lowercase letters. Jokes, both of them; tragic jokes. Unicode is an even bigger and more tragic joke. A 16-bit sledgehammer of a character set, Unicode strives to be all things to all people, just as did Algol 68 and PL/I.

Scripting and programming languages abound with distinctive features that could be removed or simplified.

The computing profession should consider changing character sets, particularly now that the burgeoning of the World Wide Web increases the cost of change every day. Character sets are primarily about writing systems, not about languages. Alphabetic writing systems predominate, and the so-called Latin alphabet is the most prevalent. Even the Chinese government has officially adopted a Latin representation for its national spoken language.

Since most computing and data transmission machinery uses 8-bit bytes, indirectly or directly, the industry urgently needs a single 8-bit standard character set for representing alphabetic text. It should particularly accommodate the Latin alphabet and allow for a general use of diacritical marks to suit simultan-eously the Latin-alphabet representation of even such languages as Turkish and Vietnamese.

REWARDS OF CONVERGENCE

If this convergence and simplification took place, the same language could serve for both scripting and programming—indeed there need be no distinction between the two activities. This would allow programmers to write simpler program components since they would be more general. The stage would be set for leaner software ("A Plea for Lean Software," Feb. 1995, pp. 64-68) and for more software jewels ("Why Software Jewels Are Rare," Feb. 1996, pp. 57-60). Programs would use fewer program components since they could be more easily cobbled together and adapted by scripting, that is, be more reusable. Using a common character set, programmers could write components that process alphabetic text properly, whatever language it represented, and do so for users and programmers they couldn't talk to. Programmers could more thoroughly develop these fewer program components using the shell interpreter before compilation.

Why bother proposing convergence when the industry seems to thrive on divergence? Because although the battle between simplicity and complexity in technology is fundamental, real progress comes from simplifying first, then extending. To perpetuate complexity is to prevent fundamental innovation. For example, the windowing interface is a good one, but all it really does is make a single display screen behave as though it were many. Perhaps one day a windowing PC or network computer will provide windowing in hardware where it properly should be. Then programmers can simply use operating systems to do what they used to do primarily (run programs), and applications could compete for user loyalty in a standard environment.

If hardware provided windowing (perhaps as set forth in an industry standard), an operating system could run programs with whatever windows they needed, the programs running independently of each other or pipe-connected, as specified by commands or shell scripts. This would become more feasible if programming languages became simpler (along the lines I've suggested), and if we can eliminate the divergence between scripting languages and conventional programming languages. ❖

Neville Holmes is a senior lecturer at the University of Tasmania. To the Internet he is neville.holmes@utas.edu.au.