New programming languages are born every day. Why do some succeed and some fail?

JOHN R. MASHEY, TECHVISER

Languages,
Levels, Libraries, and Longevity

In 50 years, we've already seen numerous programming systems come and (mostly) go, although some have remained a long time and will probably do so for: decades? centuries? millennia? The questions about language designs, levels of abstraction, libraries, and resulting longevity are numerous. Why do new languages arise? Why is it sometimes easier to write new software than to adapt old software that works? How many different levels of languages make sense? Why do some languages last in the face of "better" ones?

We can gather insights from the last 50 years of programming systems to the current time. For the far future, Vernor Vinge's fine science-fiction novel, *A Deepness in the Sky*, rings all too true. The young protagonist, Pham, has joined a starship crew and is

Languages,
Levels, Libraries, and Longevity

learning the high-value vocation of "programmer archaeologist," as the crew's safety depends on the ability to find needed code, use it, and modify it without breaking something. He is initially appalled at the code he finds:

> The programs were crap…Programming went back to the beginning of time…There were programs here that had been written five thousand years ago, before Humankind ever left Earth. The wonder of it—the horror of it…these programs still worked… down at the very bottom of it was a little program that ran a counter. Second by second, the Qeng Ho counted from the instant that a human had first set foot on Old Earth's moon. But if you looked at it still more closely… the starting instant was actually about fifteen million seconds later, the 0-second of one of Humankind's first computer operating systems…
>
> "We should rewrite it all," said Pham.
>
> "It's been done," said Sura.
>
> "It's been tried," corrected Bret…"You and a thousand friends would have to work for a century or so to reproduce it… And guess what—even if you did, by the time you finished, you'd have your own set of inconsistencies. And you still wouldn't be consistent with all the applications that might be needed now and then…"
>
> "The word for all this is 'mature programming environment.'"[1]

Any old Unix person would be amused to think that Unix's January 1, 1970, date would be enshrined so long. We have begun a process in which many people's lives are already dependent on the correct working of software, and likely to become even more so. Software once runnable only on large systems migrates downward onto larger numbers of smaller computers. Some current cellphones use 300-MHz CPUs, running at a rate higher than any CPU commercially produced by 1990. Some have 64 MB of memory, competitive with many expensive systems of the late 1980s. Vinge's book extrapolates from current small "smart dust" computers to assume that 5,000 years from now, most computing will be done by their hyper-

powerful, barely visible descendants, containing layers of software (and more than a few trapdoors). In the United States, we already have approximately 100 CPUs per person, and this number has traditionally increased tenfold each decade. As wireless sensor networks proliferate, we face a future in which most objects have CPUs and are linked together via radio.

Software already matters, will continue to matter even more pervasively, and language choice will always be an important element of software quality, understandability, and usability.

LANGUAGE WARS ARE FOREVER
Language wars seem to go on forever. Classic references on early languages are Jean Sammet's *Programming Languages—History and Fundamentals*,[2] which discussed approximately 120 important languages as of 1969, and Richard Wexelblat's *History of Progarmming Languages*,[3] which recorded a conference that chose 10 important languages created before 1967 and still in use in 1977. Of the 10, substantial new code is still written by many people in Basic, Cobol, and Fortran. Others remain popular in their specific domains (Lisp, APT, and occasionally Snobol), and some long-established IBM languages (PL/I, GPSS) remain. Most of the 120 are gone.

Successful languages continue to arise from small groups in industry or universities, from commercial vendors, or via consortia. In fact, with current CPUs and software, it is easier for individuals to create interesting languages. By 2020, when those CPUs are laughably ancient, it should become even easier.

Several Web sites extensively catalog computer languages, including http://hopl.murdoch.edu.au.

LEVELS AND LEVERAGE
Unlike computers, human beings are not easily reengineered for higher performance. Programmers vary wildly in ability, but each person has real I/O limits in reading and writing code. Much software progress has come from using faster computers, more efficient for people, if less so for the computer. Dramatic increases in computer perfor-

mance and storage are matched by the expansion of code.

In each computer class (mainframe, minicomputer, microcomputer), people tended first to write assembly code for performance, then use higher-level languages as the computer class became more powerful. Commonly, more powerful languages are later-binding, moving more decisions closer to execution time. A typical progression is as follows:

Assembly language is normally one-to-one with CPU instructions.

Macro-assembler is one-to-many with CPU instructions, good for parameterized expansion of standard code sequences. Humans are still burdened with substantial work in arranging data storage, allocating registers, and choosing efficient instruction sequences. These first two levels have mostly (and thankfully) disappeared for most programmers, but some small embedded micros and many DSPs (digital signal processors) are still programmed this way.

Higher-level algorithmic languages such as Fortran and C automate much low-level detail so the human can concentrate on algorithms. Object orientation, inspired by Simula and Smalltalk, and found widely in C++, Java, and C#, improves code and eases maintenance with better data structures.

Domain-specific languages such as APT (Automatically Programmed Tools) aim at a target domain, and so can supply specific operations needed there and ignore everything else. Text-processing languages, such as roff, troff, SGML, Scribe, TeX, Postscript, and HTML, are familiar members of this group, some of which are powerful programming languages in their own right.

Very high-level languages such as APL, Mathematica, and computer spreadsheets let people perform extensive calculations with minimal programming. APL was first implemented in the late 1960s, with strong leverage for many kinds of problems, but was loved or hated because it was deemed write-only. Many spreadsheet users do not think of themselves as programmers, but they do the same work as would have required much Fortran years ago.

Scripting languages such as Perl,

PHP, Javascript, and Python are widely used where code need not be so efficient, but where human efficiency, ease of expression, and maintenance are the highest priorities. Sometimes a change in computing environment requires new types of languages to allow widespread use. Programming distributed applications was for decades a difficult task that could be handled only by experts, despite repeated attempts to write better languages or toolkits for creating them. The Web changed that substantially.

As an example of the evolution of different levels of languages, let's go to the Bell Laboratories of the 1970s, one of several environments that helped create important foundations of current computing. Of course, its roots go even further back.

In 1970, "real computers" were still mainframes, although minicomputers were seeing increasing use. The DEC (Digital Equipment Corporation) 16-bit PDP-11 was introduced in 1970, and of particular importance, the PDP-11/45 appeared in 1972, with up to 248 KB of MOS (metal-oxide semiconductor) memory. By 1975, the PDP-11/70 allowed a huge increase to 4 MB, although each program was still restricted to 64 KB instructions and 64 KB data. Some sites supported 16 simultaneous users on an 11/45, and with heroic effort, 48 on an 11/70. The VAX-11/780 was introduced in 1977 and spread lower-cost 32-bit computing more widely. By the end of the decade, minicomputers were "real computers," and 32-bit microcomputers were beginning to appear.

In 1970, there was widespread use of applications languages such as Fortran, Cobol, and PL/I, but many applications' and most systems' codes were still written in assembly language, and the idea that an operating system would be portable among machines was laughable. In the end, Unix was ported to many systems, C was widely used, and applications were being written with various combinations of higher-level tools.

In 1973, the PWB (Programmer's Workbench) began in a Bell Labs software tools department.[4] It supported a 1,000-person division that produced database and communications application software products that ran on various mainframes and
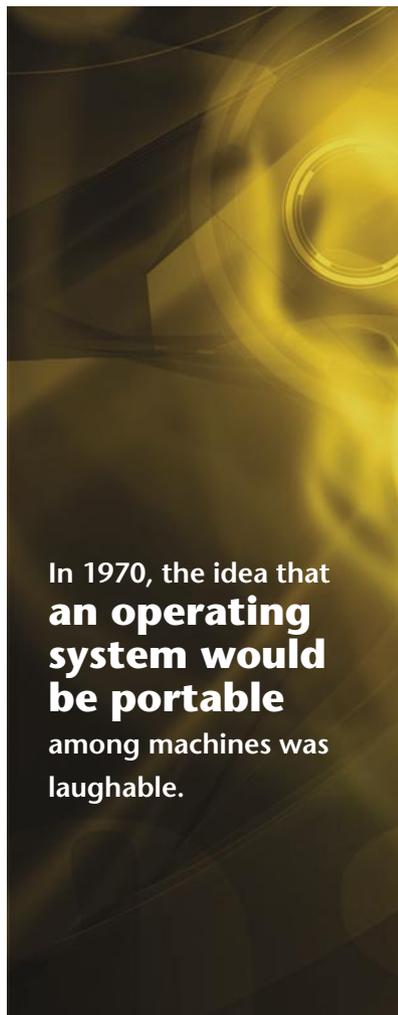
In 1970, the idea that **an operating system would be portable** among machines was laughable.

Languages,
Levels, Libraries, and Longevity

minicomputers. It wished to move many programming activities off expensive mainframes onto a common Unix-based development environment to avoid the creation of unique support software for each target system. Programming departments needed to be convinced to change their ways, that Unix was a good thing, that minicomputers were not toys, and that they should transfer budget to the tools department for more PDP-11s.

In 1973, most of the several dozen existing Unix systems were the property of individual departments, used by small numbers of people for their own projects and administered informally, sometimes with minimal security. The PWB site was the first in Bell Labs to run a "Unix computer center" for shared general use among departments, including typing pools. For years it was the largest single Unix site, and it often endured early encounters with problems of scalability, system administration, charging, security, automation, and usability for nontechnical users.

In 1973, Ken Thompson's Unix shell was primarily used as an interactive interpreter, but had some rudimentary scripting ability, including separate IF and GOTO commands. In 1974, I used shell scripts to build a small document management package for a potential client department and found this to be a great way to build such software quickly, but with awkward restrictions. In 1975 and 1976 the PWB's shell got simple variables, better control structures (IF-THEN-ELSE-ENDIF, SWITCH, WHILE), and interrupt-catching. The variables that later became $HOME (home directory) and $PATH (variable search path for commands) date from this effort.

Shell programming rapidly became a widespread mechanism for PWB users to help automate their work.[5] Substantial CPU time was consumed by shell procedures, to the point where previously separate commands, such as IF, GOTO, and SWITCH, were moved into the shell itself with substantial performance improvements. Steve Bourne was then working on a brand-new shell in Computing Research and, after much discussion, evolved a fresh design whose performance and features were interesting enough to eventually replace the PWB shell.

The variables got generalized into the "environment variables" designed for 7th Edition Unix.

Al Aho, Peter Weinberger, and Brian Kernighan had written awk, the philosophical ancestor of some popular current scripting languages. In the 1970s, Bell Labs was busily constructing computer systems to improve Bell System operations, and many were built on Unix and even used scripting languages in delivered software. CRAS (Cable Repair Adminstrative System) was a data-mining software package that integrated data from several other systems, was distributed between IBM mainframes and Unix minicomputers, had to be deployed quickly, and was sensitive to organization-dependent requirements in a time of major reorganization.[6] The first version included 10 KLOC (thousands of lines of code) of C plus 15 KLOC of shell+awk scripts and was modified quickly in the field to adapt to newly revealed customer requirements. A large listing of these scripts appeared on Kernighan's desk—to his great surprise, as the awk writers had never expected such extensive use in production.

Shell scripting used late-binding, high-level interpretation to combine higher-performance, compiled components. Awk gave us a more flexible language above C, although we sometimes later converted heavily used awk to C for performance, after requirements had settled. We sometimes wished for an awk compiler. Raising the language level enabled vast improvements in productivity in that decade, as C replaced assembly language, and script-level languages greatly augmented C.

LIBRARIES

It is preferable to reuse existing code rather than to write new code. Subroutine libraries that enable such reuse go back at least to David Wheeler on EDSAC in 1947.

A partner to the level issue is the "programming-in-the-large" problem. It is all too easy to write code, then fail to organize and document it well enough that someone else can find and reuse it, or even harder, modify it. This was the problem of Vinge's protagonist, and sometimes the problem of "write-only" APL. There has been substantial progress over the years—from simple libraries,

to software development environment systems, to today's Web-based tools for sharing and searching—but improvements must continue.

Bell Labs's Doug McIlroy's 1969 words remain relevant to this day:

> "Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality, and time-space performance.… We undoubtedly produce software by backwards means. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters." [7]

ACCEPTANCE AND LONGEVITY

Many people have proposed, and even implemented, good languages that were never widely accepted. To gain widespread support, a new language needs to achieve one of several goals.

First, it might effectively address some new problem domain. If it's early, if people tolerate its flaws, if it gains support, and if the flaws keep getting fixed, then it may be difficult for a successor to supplant, even if successors are more elegant.

Second, it might substantially raise the level of abstraction, greatly ease some programming task, and appear at a time when additional performance consumption is acceptable. Some fine ideas have simply appeared too early. Some other ideas proved surprisingly difficult to implement on most hardware, such as Algol's "call-by-name." C raised the level compared with assembly language, offered facilities that were close to efficiently implementable hardware, and compiler optimization kept improving fast enough to fend off lower-level competitors.

Third, it might handle new data types poorly addressed by existing languages. Features such as vectorization, parallelization, or parallel multimedia demand language extensions and sometimes new languages, as old ones may have difficulty adapting. Interesting experimentation is happening in the use of C for describing the wild profusion of new features being created in embedded processors, as C's normal data types deal poorly with hardware that is efficient for mixtures of 10-, 12-, and 24-bit packed data items. Extensions such as SystemC are used for hardware description. C was a strong influence on the design of RISC processors in the 1980s, and now it appears that current hardware innovations may influence variations of C.

Finally, it might be supported by a large consortium or a strong vendor, and thus might persist a long time.

Anyone who is tempted to create a new language esthetically cleaner than, but incompatible with, some existing, widely used language, and 10 percent better, should resist temptation. In an existing domain, a new language needs to be much better on important metrics to have any chance. The best opportunities happen with big performance jumps or with major changes in applications, such as the Web.

In any given problem area, there is room for a stack of languages at different levels, but there is not much room for many different competitors at the same level in the same domain. Sometimes languages get squeezed by combinations of competitors from above and below. For example, once-popular languages above assembly language but below C—such as Intel PL/M, P. J. Plauger's LIL, Niklaus Wirth's PL/360, and IBM PL/S—got squeezed out by C.

A FEW FUTURE POSSIBILITIES

First, when new hardware supports new data types, either languages must be extended, or new ones arise. We are in the early stages of great instruction-set innovation in embedded CPUs. In some cases, people can invent new instruction sets and try them out in hours. C and C++ are either going to stretch far, or be replaced in that domain.

In an existing domain, **a new language needs to be much better** on important metrics to have any chance.

Second, tight-coupled parallelism keeps increasing, via multiprocessors, increasingly on single chips, of which some already have 100-plus CPUs. Some applications can use ordinary sequential code, but for others to harness this compute power, better languages will help. Of course, people have been working on this for decades, and many big parallel applications are still written in Fortran, and parallel programming is still hard. Cheap on-chip multiprocessors may enable successful new languages.

Third, loose-coupled parallelism will need help, especially as the scale and nature of networks change with

Languages,
Levels, Libraries, and Longevity

more smart-dust and mobile systems, in which nodes come and go. Harnessing all this is even harder than managing close-coupled parallelism, and there should be language opportunities here, as has emerged with the Internet.

Fourth, higher-performance hardware always allows later binding time, more interpretation, or more just-in-time compiling. The ideal for many is to write at the highest level, with late binding, and have the software system take care of dynamically recompiling heavily used parts of software to make it faster, as is done in some dynamic binary translation systems. There may be room for new languages designed for such ideas.

BUILDING BETTER LANGUAGES

Software designers must continue to build better languages that harness increasing performance to unchanging human characteristics. They must continually raise the level of abstraction, so that humans can ignore more details. Continual improvement is needed in tools for organizing software, so that people can more easily discover existing code, reuse it, and adapt it.

In 50 years, we've seen languages come and go, sometimes leaving behind archaeological digs of crucial software whose original hardware has long since become inoperative, and some writers likewise. Some languages have shown amazing longevity, despite the later development of much "better" ones. In practice, anything successful seems to build such a code base that it may never go away. We need to keep finding better ways to express our programming ideas that let us safely rewrite old code.

Otherwise, in 7000 A.D., will Vinge's Pham still be puzzling over a Unix C timer routine? Quite possibly! Q

"The evil that men do lives after them. The good is oft interred with their bones." –William Shakespeare, *Julius Caesar*, Act III, scene ii.

REFERENCES
1. Vinge, V. 1999. *A Deepness in the Sky*. New York: Tor, 222-228.
2. Sammet, J. 1969. *Programming Languages—History and Fundamentals.* Englewood Cliffs, NJ: Prentice-Hall.
3. Wexelblat, R., ed. 1981. *History of Programming Languages*. New York: Academic Press. (Final proceedings of ACM SIGPLAN History of Programming Languages Conference, Los Angeles, CA, June 1-3, 1978.)
4. Dolotta, T., and J. Mashey. 1976. An Introduction to the Programmer's Workbench. *Proceedings of the 2nd International Conference on Software Engineering* (October 13-15), IEEE 76CH1125-4 C: 164-168.
5. Mashey, J. 1976. Using a Command Language as a High-level Programming Language. *Proceedings of the 2nd International Conference on Software Engineering* (October 13-15), IEEE 76CH1125-4 C: 169-176.
6. Boggs, P., and J. Mashey. 1982. Cable Repair Administrative System. *Bell System Technical Journal* 61 (July-August), Part 2: 1275-1291.
7. McIlroy, M.D. 1969. Mass-produced Software Components. In *Software Engineering*, ed. P. Naur and B. Randell. Garmisch, Germany: NATO Report (October 7-11).

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

JOHN MASHEY, a consultant for venture capitalists and technology companies, sits on various technology advisory boards and is a trustee of the Computer History Museum. He spent 10 years at Bell Laboratories, working on PWB/Unix, including text-processing (the troff MM macros), command languages, Unix process accounting, security, and other issues needed to help Unix become widely accessible. He later managed development of Unix-based applications and software development systems. He was one of the designers of the MIPS RISC architecture, one of the founders of the SPEC (Standard Performance Evaluation Corporation) benchmarking group, and chief scientist at Silicon Graphics. His interests have long included interactions between hardware and software. He received a B.S. in mathematics and an M.S. and Ph.D. in computer science from Pennsylvania State University.