In Praise of Scripting: Real Programming Pragmatism

Ronald P. Loui
Washington University in St. Louis

The author recommends that scripting, not Java, be taught first, asserting that students should learn to love their own possibilities before they learn to loathe other people's restrictions.

Today, the boldness of John K. Ousterhout's 1998 *Computer* article, "Scripting: Higher Level Programming for the 21st Century," is vindicated.[1] Every major observation and benefit appears genuine. Significantly, *IEEE Software* recently printed a canonical attack on scripting, "Java Makes Scripting Languages Irrelevant?"[2]

This attack is interesting because the author seems unconvinced of his own title; the paper concludes with more text devoted to praising scripting languages than it expends in its declaration of Java's progress toward improved usability. Which is a better recommendation for scripting remains unclear: the durability of Ousterhout's text or this recent critic's indecisiveness.

Most shocking, the academic programming language community continues to reject the change in programming practices brought about by scripting. Enamored of the object-oriented paradigm, especially in the undergraduate curriculum, never quite ready to accept LAMP (Linux-Apache-MySql-Perl/Python/Php), and firmly believing that more programming theory leads to better programming practice, the academics remain deaf to Ousterhout.

That scripting has developed in the shadow of object-oriented programming explains part of the problem. The two are not incompatible, but one philosophy has received the most attention. Scripting has been appearing language by language. Those who might advocate a scripting philosophy will more likely praise their favorite language, including Ousterhout, who spent much of his article praising Tcl. Today, many questions about scripting persist:

- Is there a scripting language appropriate for teaching CS1 (the first programming course for majors in the undergraduate computing curriculum)?
- Is there room for scripting in enterprise or real-time applications?
- Is there a way for scripting practices to scale to larger software engineering projects?

Fortunately, all these questions now have legitimate answers.

RECENT HISTORY

The years 1996 through 1998 were perhaps the most interesting in the phylogeny of scripting. During that time, Perl "held the Web together" and, along with a new Posix Awk and Gnu Gawk, shipped

Published by the IEEE Computer Society
0018-9162/08/$25.00 © 2008 IEEE

with every new Linux implementation, considerably improving on older shell scripting practices.

Meanwhile, Web developers furiously deployed JavaScript—itself bearing no important relation to Java, having been renamed from "livescript" for purely corporate purposes, apparently a sign of Netscape's solidarity with Sun (and even renamed "Jscript" under Microsoft, now officially "ecmascript"). Also, a handoff from Tcl/Tk to Python took place as the language of choice for GUI developers who would not yield to Microsoft's Visual Basic.

Php appeared about the same time, although it would take another round of development before it would start displacing server-side Perl, ColdFusion, and Asp. All these are now considered classic, even prototypical, scripting languages. The most recent scripting language to capture the imagination, Ruby, has actually been around awhile, promising object-oriented cleanliness with Perl-like productivity. In the Java world, various forms of scripting have been produced in the past three years that are compatible with their virtual machines.

Java and the Web

By the mid-1990s, the shift from Scheme to Java as the dominant CS1 language had already been completed, and the industry had ceased questioning C++'s superiority over C. But Java applets were not well supported in early browsers, so the appeal of "write once, run everywhere" quickly became derided as "write once, debug everywhere." Webpage forms, which used the common gateway interface (CGI), proliferated, and systems programming languages like C became recognized as overkill for server-side programming.

Web developers quickly discovered the main advantage of Perl for CGI forms processing, especially in the dotcom setting: It minimized the programmer's write time. The algorithms were simple, network latency masked small delays, and developers built database performance into the database software—at which point the bottleneck became programming. Maintenance proved less troublesome than feared as well, with developers and management both happy to rewrite code for redesigned services rather than deal with legacy code. Scripting, it turned out, was so powerful and programmer-friendly that creating new scripts from scratch was easier than modifying old programs.

GUI surprise

As far back as 1990 most of the programming effort had already shifted to writing the GUI, and the object-oriented paradigm had much of its momentum in the inheritance of interface widget behaviors. Surprisingly, the interface that most programmers needed could be had in a browser.

The HTML/JavaScript/CGI trio became the GUI and, if more was needed, ambitious client-side JavaScript proved more reliable than the browser's Java virtual machine. Moreover, the server-side program simply offered a better way to distribute automation in a heterogeneous Internet than the downloadable client-side program, whether the download was in binary or bytecode.

Strong typing, a naming regimen, and verbosity were supposed to help programmers avoid errors.

PROGRAMMING POWER

Although developers disagreed on the exact necessary and sufficient properties that characterized scripting and distinguished it from "more serious" programming, several things had become clear about scripting:

- it permitted rapid development, often regarded as merely "rapid prototyping," but subsequently recognized as a kind of agile programming;
- it provided the kind of high-level programming that had always been envisioned in the ascent from low-level assembly language programming to higher levels of abstraction—it was concise and shielded programmers from concerning themselves with many performance and memory management details;
- it was well suited to working with data in heterogeneous, mixed-user settings where the majority of a programming task consisted of transforming user data, as opposed to the connecting of components, which Java did well, or the control of a well-designed system, which was C++'s realm; and
- it was easier to get things right with short source code, in which code that was not too terse or verbose determined behavior, when all types could be coerced into strings for debugging, when identifiers were short, and when programmers could turn ideas into code quickly without losing focus.

This last point was extremely counterintuitive. Strong typing, a naming regimen, and verbosity were supposed to help programmers avoid errors. But the programmers who had to generate too many keystrokes and consult too many pages, who had to search through many different files to be sure of semantics, who had to follow too many rules, and who had to sustain motivation and concentration over a long period, became distracted and consequently inefficient. The language's promise to discipline the programmer quite simply did not reduce the tendency of humans to err. It exchanged one kind of frequent error for another.

Independent minds

Scripting languages became the distinctive tools of independent-minded programmers: the hackers, yes,

but also the gifted and genius programmers who tended to drive a project's design and development, according to Paul Graham.[3] Scripting became the mark of autodidacts, prodigies, and Third World programmers, the inspired class, the people who had never had to "think outside the box" because they had never been stuck inside it.

Proper and professional software engineering supposedly permits managers to level the playing field and extract considerable productivity from less talented and less motivated programmers. This makes software productivity a commodity, and programmers become disposable and exchangeable. Scripting does not promise this kind of disposability. Some languages, notably Python, Php, and Ruby, can support large-scale professional software engineering practice, but they are also quite usable by the rugged individual, the eccentric, and the rebel.

If recording and replaying macros simulates a kind of autopilot, then scripting offers a kind of proxy for human decision-making.

A corollary to this difference between the mundane and the liberating: Scripting was not enervating but actually renewing. Programmers who viewed code generation in "real languages" as tedious and tiresome viewed scripting in contrast as rewarding self-expression or recreation.

Semantics

The distinct characteristics of scripting languages that produce these effects are usually enumerated as semantic features, starting with low I/O specification costs, the use of implicit coercion and weak typing, automatic variable initialization with optional declaration, predominant use of associative arrays for storage and regular expressions for pattern matching, reduced syntax, and terse control structures. But the main reason for the productivity gains can be found in the name *scripting* itself. Scripting powerfully embeds a developer in an environment. In the same way that the dolphin reigns over the open ocean, Lisp provides a powerful language for those who would customize their Emacs, JavaScript is feral among browsers, and many older scripting beasts still rule the Linux jungle.

The basic idea of scripting even includes a hint of AI: The scripting language grants high-level control to automate by capturing the intentions and routines normally provided by a user or administrator. If recording and replaying macros simulates a kind of autopilot, then scripting offers a kind of proxy for human decision-making. Nowhere is this clearer than in one-line embedded Php, or in sysadmin shell scripting, or in the scripting of artificial agents in computer games.

CURRENT CLAIMS

While it might have been risky for Ousterhout to proclaim scripting on the rise in 1998, it would be folly to dismiss the success of scripting today. Scripting languages are excellent choices for CS1. To me, Java-based CS1 is the single greatest mistake in the history of computing curricula. Students should learn to love their own possibilities before they learn to loathe other people's restrictions.

There is a lot of the old fascist-versus-anarchist dispute here, but there is also empirical evidence. I reported in 1996[4] that only the scripting programmers could generate code fast enough to keep up with the demands of the artificial intelligence laboratory class. Even though students could choose any language they wanted, and many had to unlearn top-down ways of doing things, few could turn new ideas into code without scripting. In the intervening decade, little has changed.

Students who learn to script early are empowered throughout their college years, especially in the crucial Unix and Web environments. Those who learn Java and C++ first are stifled by enterprise-sized correctness. Early programmers must learn to be creative and inventive, and they need programming tools that support exploration rather than production. Software engineering aesthetics should come after programming, not the other way around.

Scripting CS1

What scripting language could be used for CS1? I personally prefer Gawk, JavaScript, Php, and Asp, mainly because of their gentle learning curves. I don't think Perl would be a disaster; its imperfection would create many teaching moments. But an emerging consensus in the scripting community holds that Python is the right solution for freshman programming. Ruby would also be a defensible choice.

Python and Ruby have the enviable properties that almost no one dislikes them, and almost everyone respects them. Both languages support a variety of programming styles and paradigms and satisfy practitioners and theoreticians equally. Both languages are carefully enough designed that developers can demonstrate "correct" programming practices and enforce high standards of code quality. That Google stands by Python provides added motivation for undergraduate majors. Google originally used Python because Scott Hassan, who wrote much of the prototype for Brin and Page, had been mentored by an early Python guru and because I was unable to convince him over many years in St. Louis to switch to Gawk or Perl.

But do scripting solutions scale? What about the performance gap when the algorithm faces large n? What about software engineering on big projects? There has been extensive discussion about scriptings' scalability. In the past, these debates have simply ended with the

concession that developers must rewrite large systems in C or C++, once the scripting had served its prototyping duty.

Multiple languages

Indeed, scripting languages are not the answer for long-lasting, CPU-intensive nested loops. Matrix multiplication is simply faster in other languages. Developers can easily identify such bottlenecks and rewrite the code in a more appropriate language. But a harder language or one with black-box libraries of objects and methods does not always offer the best performance choice.

Often, we see that a team did not implement efficient data organization because it would have required more code—code that they would have attempted and likely successfully written in an easier programming language. We saw this in the AI class with heuristic search and computer vision, where brute force is better in C, but complex heuristics are better than brute force, and scripting is better for complex heuristics.

When algorithms are exponential, it usually doesn't matter what language developers use because most practical n will incur too great a cost. Again, the solution is to write heuristics, and scripting is the top dog in that house. Processors are so much faster than disks these days that a single extra disk read can erase the CPU advantage of using a compiled language instead of an interpreted one.

Programmers also benefit from using multiple paradigms and languages. Projects can and do contain a mix of scripting, high-performance programming, and professional componentware. I weep when I think about the text processing written in C under my managerial watch because the programmer didn't know Perl. Considering that there are much better scripting tools for much of what gets programmed in Java and C++, perhaps the question should be whether Java and C++ scale to enterprise projects.

PRAGMATICS

The real reason scripting blindsided academics is because they lack practicality. Academia understandably holds software practice at a distance. There is, however, a purely intellectual reason why programming language courses have only now warmed to scripting.

The historical concerns of programming language theory have been syntax and semantics. Java's amazing contribution to computer science is that it raised so many old-fashioned questions that occupied existing programming language experts: How can it be compiled? Why aren't all of its data types first-class objects? Ruby has perhaps had a calming influence lately because we can also puzzle over its syntax and semantics, looking inward, instead of the disruptive influence that Ruby on Rails might have on Web programming, looking outward.

But new questions beyond syntax and semantics should be asked, such as what a particular language is well-suited to achieve inexpensively, quickly, or elegantly, especially with the new mix of platforms. People have informal answers, but they have no frameworks to help organize their knowledge. A need to control environments with specific programming and engineering preferences drives the proliferation of scripting languages. Their rise demands a new age of innovation in the study of programming languages, a new taxonomy, and a new set of issues.

We are entering an era of comparative programming language study in which the issues are higher level, social, and cognitive.

Linguists recognize something above syntax and semantics, what they call *pragmatics*,[5] which addresses the more abstract social and cognitive functions of language: situations, speakers and hearers, discourse, goals and uses, and performance. We are entering an era of comparative programming language study in which the issues are higher level, social, and cognitive. We have experienced a primitive version of programming language pragmatics when referring to broad purposes: "this is a database query language" or "this is a good language for teaching" or even "this is a suitable language for scientists."

Some basic questions have received little formal study as language questions, although they have certainly been noted as software engineering questions:

- What is the average lifetime of a program written in language X for programmers of type Y, for a program of type Z?
- What is the average time to complete a program in language X for programmers of type Y, for a program of type Z?
- What is the average time spent authoring versus debugging a program in language X for programmers of type Y, for a program of type Z?
- What is the learning curve for language X, to reach a specific competence—for example, to write a program of type Z?
- What level of concentration is required, on average, to write in language X for programmers of type Y, for a program of type Z?
- What is the consumption of short-term memory?
- What is the likelihood that a library function, method, or object will be available in a user's environment?
- What is the average time to start a runtime environment? What is the typical impact on the rest of the system?

- Does language preference correlate with gender, generation, IQ, or personality type?

Internet programming language "shootouts" and "scriptometers" have sought to address some of the questions relevant to the choice of scripting language, but these have only been first steps. For example, one site reports on the shortest script in each scripting language that can perform a simple task. But absolute brevity for trivial tasks, such as "print hello world" is not as illuminating as typical brevity for real tasks, such as XML parsing. Formulating meaningful, technically informed measures of pragmatic desirability will require some cleverness. Prechelt's study[6] is one of the best of this kind so far, but it still lacks an essential framework.

Michael Scott's popular textbook, *Programming Language Pragmatics*,[7] is a fairly traditional tome that concerns itself with parameter passing, types, and bindings. It's hard to see why this book merits *pragmatics* in its title, even as it goes to a second edition with a chapter on scripting added. We need a programming language pragmatics to go past the analysis of syntax and semantics in the same way that linguistics studies perlocution and illocution.

Pragmatic questions are not the easiest for mathematically inclined computer scientists to address. They refer by nature to people and their habits, sociology, and the day's technological demands. An industrial psychology literature, apart from computing, has sometimes addressed questions of this kind. But this kind of study must become part of programming language theory within computing. It's the importance of just these kinds of questions that makes programmers choose scripting languages.

Ousterhout declared scripting on the rise, but perhaps so too are programming language pragmatics.

References

1. J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *Computer*, Mar. 1998, pp. 23-30.
2. D. Spinellis, "Java Makes Scripting Languages Irrelevant?" *IEEE Software*, vol. 22, no. 3, 2005, pp. 70-71.
3. P. Graham, *Hackers and Painters*, O'Reilly, 2004.
4. R.P. Loui, "Why Gawk for AI?" *SIGPLAN Notices*, vol. 31, no. 8, 1996, pp. 8-9.
5. S. Levinson, *Pragmatics*, Cambridge University Press, 1983.
6. L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, no. 10, 2000, pp. 23-29.
7. M.L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, 2000.

Ronald P. Loui *is an associate professor of computer science and engineering at Washington University in St. Louis. His research interests include Web programming, artificial intelligence, and hardware-software codesign. Loui received a PhD in computer science from the University of Rochester. Contact him at loui@cse.wustl.edu.*