COMPARING WEB LANGUAGES IN THEORY AND PRACTICE

by

Kristofer J. Carlson

A Graduate Research Report Submitted for INSS 690
in Partial Fulfillment of the Requirements of the Degree of
Master of Science in Management Information Systems

Bowie State University
Maryland in Europe
May 2005

TABLE OF CONTENTS

Abstract

Scripting languages are weakly typed, interpreted languages. Scripting languages complement traditional systems programming languages. Scripting languages are highly flexible languages that operate on a higher level of abstraction then systems programming languages. By being weakly typed and interpreted, scripting languages are able to glue together applications written in a variety of different systems programming languages. Because they operate on a higher level of abstraction, they are especially well suited to rapid prototyping. Also, the best programmers want the best, most versatile tools, and prohibiting the use of the best tools will ultimately limit the ability of an enterprise to attract and retain the best programming talent. The Information Systems professional needs background information about scripting languages and their relationship to systems programming languages so they can separate the hype from reality; understand the advantages and disadvantages of scripting; and pick a language based on the needs of the project rather than the sometimes irrational preferences of the individual programmer.

Comparing Web Languages

In Theory and Practice

Introduction

Over the last twenty years, programming has changed from programming entire systems in a single systems programming language, to designing systems built of individual software components and gluing these components together with a scripting language (Ousterhout, Introduction, ¶ 1, 1998). This change has gone unnoticed by the academicians (Ousterhout, The role of objects, ¶ 1), making it difficult for the Information Systems professional to 1) determine the value of scripting in their development and operational environment and 2) decide upon the appropriate scripting language. Addressing these issues requires 1) an examination of the specific language features that separate scripting languages from systems programming languages and 2) the development of a theoretical framework that can be used to evaluate the different languages. Only after those two issues have been addressed is it valuable to begin comparing different languages.

Systems applications have historically been "developed monolithically," using a single programming language (Tcl, From monolithic applications to components, ¶ 1, 2001). Eventually programmers began writing systems components instead of entire systems applications. They then glued these components together using a scripting language. "Systems programming languages such as C, C++, and Java provide excellent tools for creating components, but they are not well-suited to integrating and extending components: their compiled nature and strong typing make them too static and inflexible for component integration. In contrast, scripting languages…are interpreted and weakly typed. This makes them much more flexible (an essential attribute when combining components that weren't originally designed for each other) and provides dramatically faster development and evolution of applications (Tcl, Scripting: From monolithic applications to components, ¶ 2, 2001).

What is Scripting?

Scripting languages are typically defined by their suitability to a specific problem domain. Mark Lutz, author of *Programming Python*, says a scripting language "makes it easy to utilize and direct other software components" (Lutz, p. xix, 2001). A more colorful and typical description is provided by John K. Ousterhout, who says "Scripting languages are designed for 'gluing' applications" (Ousterhout, Abstract, 1998). Scripting languages were initially developed to automate a series of systems tasks on a specific system or to extend the capabilities of several languages by enabling them to connect to each other (Cowlishaw, ¶ 2, 4, n.d.). With the rise of the World Wide Web and the development of componentized software, scripting languages have become increasingly important, as they are the glue allowing interactive connections between the web client and the server, and between different applications and software components on the server (Ousterhout, Scripting is on the rise, ¶ 3). Successfully performing these functions requires a great deal of language flexibility (Tcl, Scripting: the right platform for component integration, ¶ 1).

A change in the programming environment from designing systems applications towards the development of componentized software encourages the use of scripting as part of systems development. Even Microsoft has discovered the benefits of componentized software. Their new operating system, code-names "Longhorn," is being developed using software components, primarily to speed development. Automated tools and processes can complete the test of a component in "less than an hour," instead of the 18 hours it took using more traditional software development practices (Foley, 2005). Because of the speed and ease of testing, "Microsoft will be more likely to hit its deadlines than in the past" (Foley, 2005). This does not mean that Microsoft is using scripting languages to glue these software components together. Due to the relatively static nature of these operating systems components and their relationships to each other, this is unnecessary. Still, for Microsoft to shift to componentized software indicates this programming approach is here in a big way and bodes well for the future of scripting languages.

Two specific language features help determine whether a language is suitable for scripting; other language features determine whether a scripting language will be successful. John Ousterhout points out that scripting languages "tend to be typeless;" that scripting languages are interpreted instead of compiled; and that scripting languages assume the existence of "a collection of useful *components* written in other languages" (Ousterhout, Scripting languages, ¶ 1). Successful scripting languages must easily "call and execute operating system utilities and services" (O'Reilly, When and Why a Scripting Language, ¶ 7, 2004). To be successful, a scripting language must be the scripting language of a popular system (Graham, External Factors, ¶ 1, 2001). Furthermore, a successful scripting language requires extensive library functions (Graham, Libraries, ¶ 1, 2001).

Abstraction

Scripting languages operate at a higher level of abstraction than systems programming languages, (Ousterhout), allowing for more succinct programs. As Paul Graham says, "Succinctness is what programming languages are *for*" (Graham, Hypothesis, ¶ 2, 2002). After all, if we weren't striving for higher levels of abstraction, we would be programming directly in machine code. A common measure of succinctness is how many lines of machine code are executed by a single line of code in a higher level language. John Ousterhout points out that a single statement in a scripting language executes "hundreds or thousands of machine instructions, whereas a typical statement in a system programming language executes about five machine instructions (Ousterhout, Scripting Languages, ¶13).

Strong vs. Weak Typing

A big part of what makes scripting languages more abstract is because they are weakly typed, or as John Ousterhout puts it---"typeless" (Ousterhout, Scripting languages, ¶ 2). Being typeless, scripts can be more dynamic and flexible. Machine code is created as needed instead of being compiled in advance. When we say a language is typeless, or weakly typed, we are saying the language uses dynamic typing rather than static typing. Static typing means we declare both the variable and its type when the variable is created. Dynamic typing, by contract, enables the programmer to say what he wants the program to do without having to specify in detail how to do it. To put it in more concrete terms, dynamic typing allows the programmer determine a way to solve a problem without imposing artificial limitations on the variable type. For example, let's

say we had a script that took an array of dollar amounts and averaged them. If the array held dollar values only, then at execution the scripting language would determine the variable type to be an integer. But if even one of the array values has a fractional portion, (meaning dollars and cents,) then at execution the scripting language would determine the variable type of *all* the variables to be floats. By way of comparison, a systems programming language would require the types to be declared prior to compilation and would require a great deal of code to deal with incompatible data types.

In dynamically typed languages, variable type is determined not through a variable declaration, but when the variable is assigned a value (Sebesta, 2002, p. 188). The cost of dynamic typing is increased execution time because type checking can only be done at run time, and in increased storage requirements as the amount of memory reserved for a variable must be large enough to accommodate the larger data types (Sebesta, 2002, p. 187). This is wasteful of memory, but in certain problem domains the waste is acceptable given the flexibility dynamic typing affords. Dynamically typed languages are often interpreted (Sebesta, 2002, p. 187). This "has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units" (Sebesta, 2002, p. 30). In addition, debugging a program can occur more quickly because "compilation can occur while the code is being edited" (Scripting languages, 2001, ¶ 16).

The use of dynamic typing makes a language much more flexible, but comes at a cost. Dynamic typing almost always requires a language to be interpreted rather than compiled, since the type of the variable cannot be determined until program execution. Ousterhout points out that "scripting languages are less efficient than systems programming languages" (Ousterhout, Scripting Languages, ¶ 11). In practice, the use of an opcode cache (Turck, What is Turck MMCache?, ¶1, 2002) can reduce the performance hit by caching compiled scripts. The Perl scripting language has a built in opcode cache; by maintaining compiled script in memory, Perl executes at "nearly the speed of a compiled language" (O'Reilly, 2004). Dr. Nikolai Bezroukov gives us a different way of looking at the problem. He points out that for "computationally intensive tasks…such languages as Python and Perl are often (but not always!) competitive with C++, C# and, especially, Java. The reason is that when you are operating at a higher level, you are often able to find a better, more optimal, algorithm, data structures, problem decomposition schema or all of the above" (Bezroukov, If a project had died, then it does not matter what was the implementation language, ¶ 1, n.d.).

The performance of a scripting language is rarely a problem if the scripting language is used as a glue language. In this case, the scripted portion of the system is small relative to the size of the compiled portion. The performance of the system as a whole is driven by the performance of the components, not the scripts (Ousterhout, Scripting Languages, ¶ 12). The issue of execution speed arises only when a scripting language is used to do things that appear better suited to systems programming languages. John Ousterhout describes the tradeoff this way: "For complex algorithms and data structures, the strong typing of a systems programming languages makes programs easier to manage. Where execution speed is key, a system programming language can often run 10-20x faster that a scripting language" (Ousterhout, Different tools for different tasks, ¶ 1).

Memory Management

An important feature of scripting language is automated memory management. This is a process where the language allocates and deallocates memory as needed. Older languages such as C require the programmer to manually allocate and deallocate storage. Such manual allocation and deallocation requires the language to be statically typed so the size of the memory cell can be known ahead of time. Languages using manual memory management are more efficient at execution, as variable locations can be addressed directly (Sebesta, p. 191). But languages using automated memory management are more efficient to develop in (Spolsky, 2004, Automatic Transmissions Win the Day, ¶ 2).

Automated memory management increases programmer productivity, but this productivity comes at a cost. Eric Raymond writes that except for things "like kernel hacking, scientific computing and 3-D graphics," it is "far better to trade a few cycles and a few kilobytes of memory for the overhead of a scripting language's memory manager and economize on far more valuable human time" (Raymond, 2000, ¶ 14, 15).

Application (Contributor)	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months Tcl version: 1 day		60	C++ version implemented first; Tcl version had more functionality
Computer system test and installation (Andy Belsey)	C test application: 272 Klines, 120 months. C MIS application: 90 Klines, 60 months. Tcl/Perl version: 7.7K lines, 8 months	47	22	C version implemented first. Tcl/Perl version replaced both C applications.
Database library (Ken Corey)	C++ version: 2-3 months Tcl version: 1 week		8-12	C++ version implemented first.
Security scanner (Jim Graham)	C version: 3000 lines Tcl version: 300 lines	10		C version implemented first. Tcl version had more functionality.
Display oil well production curves (Dan Schaack)	C version: 3 months Tcl version: 2 weeks		6	Tcl version implemented first.
Query dispatcher (Paul Healy)	C version: 1200 lines, 4-8 weeks Tcl version: 500 lines, 1 week	2.5	4-8	C version implemented first, unmaintained. Tcl version had comments, more functionality.
Spreadsheet tool	C version: 1460 lines Tcl version: 320 lines	4		Tcl version implemented first.
Simulator and GUI (Randy Wang)	Java version: 3400 lines, 3-4 weeks. Tcl version: 1600 lines, < 1 week.	2	3-4	Tcl version had 10-20% more functionality, was implemented first.

Table 1. Each row of the table describes an application that was implemented twice, once with a system programming language such as C or Java and once with a scripting language such as Tcl. The Code Ratio column gives the ratio of lines of code for the two implementations (>1 means the system programming language required more lines); the Effort Ratio column gives the ratio of development times. In most cases the two versions were implemented by different people. The information in the table was provided by various Tcl developers in response to an article posted on the comp.lang.tcl newsgroup; see [7] for details.

Table 1: Program Development Comparison (Ousterhout, Scripting languages, Table 1).

Rapid Prototyping

Scripting languages operate on a higher level of abstraction than systems programming languages, as we have previously stated. The language characteristics that go into creating this higher level of abstraction make the language especially useful for rapid development. Table 1, provided by John Ousterhout, is a table listing 8 different applications, each of which were developed in a systems programming language and in a scripting language. In each case the scripting application took less time to develop, used fewer lines of code, and in some cases provided greater functionality. "The true difference between scripting and system programming is…a factor of 5-10x" (Ousterhout, Scripting languages, ¶ 16).

According to B. Jacobs, "Rapid Development is the primary reason for using a scripting language" (Jacobs, B. 2004, Rapid Development, ¶ 1). The reason to use a scripting for rapid prototyping is because scripting languages are "concise, malleable, and easily maintainable" (Udell, 2003, Shipping the prototype, ¶ 5). Given the increasingly rapid software development cycle, combined with increasingly well implemented scripting languages running on ever more powerful hardware, producing finished code in a scripting language, or "shipping the prototype", is increasingly likely (Udel, Shipping the prototype, ¶ 1, 8, 10).

Using Scripting to Hire and Retain Talent

Another less empirical reason to use scripting languages is because the best programmers like them. Dr. Paul Graham (August, 2004) calls this "the Python paradox: if a company chooses to write its software in a comparatively esoteric language, they'll be able to hire better programmers, because they'll attract only those who cared enough to learn it." In his essay entitled Great Hackers, Dr. Graham (July, 2004) makes the following specific claim: "The programmers you'll be able to hire to work on a Java project won't be as smart as the ones you could get to work on a project written in Python." Some IT shops don't see it that way, dividing their people into scripters and programmers. Chad Dickerson, CIO of InfoWorld (2003), says that in this environment scripters are treated as lesser programmers and are mentored and trained to become "'real' developers." What happens next is predictable:

> 1) Many of the most talented scripters eventually become disgruntled and leave for scripting-friendly pastures, and 2) the "real" developers spend days and weeks writing Java and C++ code to solve problems that those talented Perl and Python programmers could have knocked out in a few hours.

No company intentionally tries to hire lesser qualified programmers, but when they require the use of a lesser language they end up limiting their pool of talented candidates. It is likely that the better programmers self-select out of the pool of available talent, choosing instead to work for scripting-friendly companies. The scripters they do hire are less productive than they could be, and company projects take longer to complete they need be. All this affects the bottom line, indicating companies are not as rationale as economic theory would indicate.

A Framework for Comparing Languages

An interesting addition to a computer science curriculum is a course on programming language theory. The course lays out the history of language design and describes a variety of language features. In other words, this course emphasizes history and theory and provides an extensive framework for the budding programmer's knowledge. This framework is essential when selecting a programming language for a project. We have already discussed those aspects of computer language theory that separate systems programming from scripting. What we need, now, is a theoretical framework for evaluating individual scripting languages.

One such framework is provided in Robert Sebesta's text, *Concepts of Programming Languages*. In it Dr. Sebesta provides a short list of criteria used when evaluating languages: readability, writability, reliability and cost (Sebesta, pp. 8-18, 2002). Each of these criteria can be broken down into several lower-level criteria, yielding a strong theoretical framework with which to evaluate and compare programming languages. Using the theoretical foundations provided by programming theory courses, it is possible to make comparisons across a wide spectrum of systems programming languages.

While it is easy to find empirical comparisons and academic descriptions of systems programming languages, it is difficult to find the same empirical comparisons and academic descriptions for the class of programming languages known as scripting languages. (The descriptions of these systems and scripting language categories are presented in Appendix A, entitled "Language Comparisons.") What comparisons are made are rarely empirical or academically neutral. Most often, comparisons are made between two scripting languages, and are meant to prove that language A is better than language B. Instead of using a theoretical framework, like readability, writability, reliability and cost, the arguments are sometimes based on nothing more than how pretty the code is in language "A" versus how ugly the code is in language "B". ("Pretty" isn't an empirical description, but it does have a close relationship to readability and writability, and therefore to maintainability.)

Comparisons between scripting languages are long on advocacy but short on theory. Perhaps this is because academics do not consider scripting languages to be real programming languages and therefore not worth the time and effort to study and analyze them. Dr. Sebesta's textbook uses only four paragraphs to describe scripting languages, and then dismisses them by stating "they have contributed little to the development of more conventional programming languages" (Sebesta, pp. 7,8). Perhaps it is because they are considered niche programs, and not used as widely as the "conventional" programming languages, (Dr. Sebesta's term from the quote above,) as shown in the first table provided in the TIOBE Programming Community Index for January, 2005 (TIOBE, table 1, 2005). John Ousterhout suggests the "experts in programming languages and software engineering" have "focused their attention on object-oriented system programming languages (Ousterhout, The role of objects, ¶ 1). This attention has come because of the promise of increased programmer productivity. Unfortunately, object-oriented programming only increases programmer productivity by 20-30% (Ousterhout, The role of objects, ¶ 2), while scripting increases programmer productivity by a factor of 5 - 10 (Ousterhout, Scripting languages, ¶ 15). The disregard of scripting among the academicians appears to be shortsighted at best; the disregard of scripting among business IT departments may very well put their companies at a competitive advantage.

One can easily find documents in which the adherents of one programming language praise its virtues and deride all others. The quality of argument in these tirades rarely rises to the level of civilized discourse. Several people and organizations have made attempts to rectify the lack of rigorous analysis. Lutz Prechelt, for example, provided a scientific analysis of three popular programming languages and four scripting languages. He developed an innovative way to measure these languages for programmer productivity, program reliability, and computational load (Prechelt, p. 1, 2000). Dr. Paul Graham appreciates the results of Lutz Prechelt's tests, but notes that the problems used are too short to provide truly meaningful results. "A better test of a language," he writes, "is what happens in programs that take a month to write" (Graham, Comparison, ¶ 3, 2002). Dr. Graham goes on to concede that "we are never likely to have accurate comparisons of the relative power of programming languages. We'll have precise comparisons, but not accurate ones" (Graham, Comparisons, ¶ 5, 2002). Brian W. Kernighan, (the developer of "C",) and Christopher J. Van Wyk devised a series of trials to see how fast a variety of scripting languages ran in controlled trials (Kernighan, B. W. & Van Wyk, C. J., Abstract, ¶ 1, 1997). Eight years have passed since those trials, and many of the languages tested have undergone significant changes and updates since. Both Kernighan and Van Wyk told me the project was a "one off" and neither of them had any plans to update it, (B.W. Kernighan & C. J. Van Wyk, personal communications, December 23, 2004), which proves Dr. Graham's point.

While the specific language comparisons developed in Kernighan & Van Wyk's study are no longer valid due to changes in the languages themselves, their general principles and themes are still useful. (This is the value of empirical comparisons—their precise but inaccurate comparisons provide the foundation for discussions about the benefits of various language features.) First, as we have discussed, "compilation wins over interpretation, and interpretation from an intermediate representation, [byte code, for example], wins over repeated interpretation of the original program text" (Kernighan, B. W. & Van Wyk, C. J., Introduction, ¶ 6, 1997). Second, "memory-related issues" greatly affect program runtimes (Kernighan, B. W. & Van Wyk, C. J., Abstract, ¶ 2, 1997). The amount of memory available on a machine; the various memory buffers and caches; and the "implementation of memory allocation and garbage collection can…have dramatic effects on performance (Kernighan, B. W. & Van Wyk, C. J., Introduction, ¶ 6, 1997). In the case of hardware caches, these may not even be language specific features, yet can have dramatic impacts on computational speeds. To the general principles and themes provided by Kernighan and Van Wyk, we must also remember Dr. Bezroukov's point that programming in a higher level of abstraction allows you to find a "better, more optimal, algorithm, data structures, problem decomposition schema or all of the above" (Bezroukov, If a project had died, then it does not matter what was the implementation language, ¶ 1, n.d.), thus possibly negating the performance hit incurred by interpretation rather than compilation.

Scripting Theory and Development?

General language features are important, so a short description of these features will help explain the difference between a systems programming language and a scripting language. Before we do that, we should clarify what we mean by "programming language." The most succinct definition of a programming language may be: "any computer language which is either interpreted or compiled and is capable of manipulating data" (TIOBE, Frequently Asked Questions, ¶ 1, 2005). Markup languages such as HTML and XML are not programming

languages. ASP is also not a programming language; instead, it is a technique or framework that makes use of programming languages such as JavaScript and VBScript (TIOBE, Frequently Asked Questions, ¶ 1, 2005). The definition for programming languages given above may be *too* succinct, as it would include SQL as a programming language. SQL is applicable to only one problem domain—manipulating data in a relational database management system. Therefore a better definition may read as follows: A programming language is one which is either interpreted or compiled, and is capable of manipulating data *across a variety of problem domains*.

Language Classification

A literature review will find programming languages variously categorized as systems languages, scripting languages, extension languages, interface languages, and macro languages. These categories have a certain degree of overlap. As mentioned previously, systems, (or conventional,) programming languages are generally compiled, while scripting languages are interpreted. Both Python and Java are compiled into an intermediate state. Both Python and Java are systems programming languages. Python, however, is useful for scripting, while Java is not. Scripting and extension languages are seemingly the same thing; the Unix and Open Source camp, (Richard Stallman, for example,) often use the term extension instead of scripting languages (Stallman, ¶ 2, ¶ 6, 1994). IBM uses the terms "extension language" and "macro language" interchangeably (Cowlishaw, ¶ 4, n.d.).

A macro is basically a collection of commands and programming logic that controls a single system; the macro language is the language of that system, and is specific to that system (Cowlishaw, ¶ 2, n.d.). The next step in this evolutionary process was the "extension language—or 'macro' language—for a wide variety of applications" (Cowlishaw, ¶ 4, n.d.). Once programmers were able to extend and link different systems, it was a natural step to move toward treating software as components rather than systems.

The term "extension" has to do with the idea of extending systems or components, while the term "scripting" comes from the theatre; the script provides the lines and some basic directions, but the details of the performance are left to the actors and directors. The script which connects two components links one component to another, but doesn't control what each component does. Flexibility is a key characteristic of scripting languages, as they are connecting components which were not necessarily designed to be integrated together, and which may have been written in completely different programming languages. Interface languages, like Visual Basic, are generally used to develop user interfaces; these languages are sometimes called scripting languages as well. These broad categories have a great deal of overlap. Java, a systems programming languages, is also used to create user interfaces. As mentioned above, Python may be categorized both as a scripting language, but can be used for programming large, complex systems.

Scripting Languages---limited vs. full-featured

In 1994 Richard Stallman threw down the gauntlet when he declared that Tcl would not be used in GNU[1] project software. In doing so, he exposed an interesting and important dividing line between programmers---between those who use and approve of scripting, and those who are wary of it.

> A language for extensions should not be a mere "extension language." It should be a real programming language, designed for writing and maintaining substantial programs. Because people will want to do that! [sic]
>
> Extensions are often large, complex programs in their own right, and the people who write them deserve the same facilities that other programmers rely on. (Stallman, ¶ 2 & 3, 1994)

Dr. Paul Graham, develops this thought in more detail.

> A throwaway program is a program you write quickly for some limited task: a program to automate some system administration task, or generate test data for a simulation, or convert data from one format to another.
>
> [One] way to get a big program is to start with a throwaway program and keep improving it…Those that did evolve this way are probably still written in whatever language they were first written in, because it's rare for a program to be ported, except for political reasons. And so, paradoxically, if you want to make a language that is used for big systems, you have to make it good for writing throwaway programs, because that's where big systems come from (Graham, 2001, ¶ 2, 4).

Those who accept that a scripting language is merely a "glue" language, that its sole function is to glue different software components together, would select a small, flexible language. Those who want a scripting language to be able to develop program extensions, extensions that may evolve into large applications, would select a scripting language used in a rich programming environment. Scripting languages are themselves often created to accomplish limited tasks and create "throwaway programs," (Graham, *Being Popular*, 5 Throwaway Programs, ¶ 2, 5), but add features and complexity until they become full-featured programming languages. Perl, for example, "began life as a collection of utilities for generating reports," (Graham, *Being Popular*, Throwaway Programs, ¶ 5). PHP began life as a set of Personal Home Page Tools (ITworld.com, 2001, January, Humble Beginnings, ¶ 1). As programmers begin using a (typically open source) language they begin building in features they need until, after several versions, the language becomes the rich programming environment they were after.

---

[1] GNU was originally a project to develop an open source, Unix-compatible operating system. GNU stands for "Gnu's Not Unix."

Scripting language development

Scripting languages are derived from macro languages, but are applicable to a wide variety of systems. They are interpreted rather than compiled. Compilation means the program is translated into the machine language native to the specific systems architecture before they are run (Sebesta, p. 27, 2002). This compilation then runs on only that specific systems architecture. For a script to be able to connect different systems and components running on multiple systems architectures it cannot be compiled, or the compilation will only run on one particular type of machine. Instead each contains an interpreter for the particular scripting language. "The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with a high-level language program statements rather than machine instructions" (Sebesta, p. 30, 2002).

Interpretation vs. Compilation

While we have previously discussed interpretation vs. compilation, we must now address it in a different context. As previously mentioned, interpretation is "10 to 100 times slower" than running compiled machine code, and takes more space (Sebesta, p. 30, 2002). The more complicated the language and the more symbols it has, the more space the symbol table takes and the more time the machine takes to interpret the language and convert it into a specific machine language. "Languages with a simpler structure lend themselves to faster interpretation," (Sebesta, p. 30, 2002), so if speed was the only issue, a scripting language should have a simple structure. This simple language structure, however, would not lend itself to the creation of complex applications.

If compilation is many times faster than interpretation, wouldn't that be a good argument for a compiled language over an interpreted one? Not necessarily. In a rich client application, the cost of interpretation can almost be ignored. "Developers have been able to assume that each user would have an increasingly powerful processor sitting on their desk" (Graham, 2001, Efficiency, ¶ 11). Thus the cost of determining "the meaning of each expression and statement…directly from the source program at run time" (Sebesta, 2002, p. 30) is manageable.

The advantage to compilation, which is speed at execution, comes at its own cost in conventional systems programming languages. That cost is compile time. Robert Morris notes that "as code size increases, compile time rises in a distinctly nonlinear manner (Are scripting languages the wave of the future, 2001, ¶ 13). Compile time takes so long because the languages are statically typed, and every use of a variable has to be checked against its declaration. In a large program, the declarations in *all* modules must be read and checked against, even to recompile a *single* module (Scripting languages, 2001, ¶ 6, 9, 12).

Scripting languages: minimal size vs. maximal power

What Richard Stallman indicated back in 1994 was that programmers tended to use scripting languages for things they were never intended to do. Scripting languages are sometimes called "glue languages" because of the way they are used to glue components together. But because of their ease of use, programmers often begin using these languages to build entire components. Sometimes the script, which was originally intended solely to connect

two different components, ends up growing into a component of its own. For this reason Richard Stallman declared that scripting languages needed to be fully featured, because programmers used them to build full-featured programs (Stallman, ¶ 2 & 3, 1994). Of course this means that the scripting language must be more complex than it needs to be if it is used simply to glue two components together, meaning run time compilation is longer and more problematic than it needs to be. And because a scripting language is weakly typed, the program requires more storage, takes longer to convert to machine code, and the programmer is not protected from making type errors.

It is important to understand the history of a language as that may help someone either anticipate or explain certain language features. Perl, for example, is partly derived from the Unix operating system (B. Jacobs, Context Insensitivity, (Footnotes), ¶ 1, 2004). Knowing that, one should not be surprised to find Perl contains Unix derived symbols as they are what the language developers are familiar with. If a person knows the Unix operating system, then the symbols and their meanings used in Perl are familiar. It is then a natural step to move from the Unix operating system to the use of Perl as a scripting language. If a person is unfamiliar with the Unix operating system, then the symbols used in Perl lack any context and therefore any meaning (B. Jacobs, 2004), making Perl—for these programmers—neither writeable nor readable.

Conclusion

Scripting languages can be defined by the language features they hold in common, (weakly typed, interpreted, automated memory management,) or by their uses, (automating operating system functions, gluing components together, rapid prototyping.) In either case, these features and uses separate them from the more traditional systems programming languages which are strongly typed, compiled, and are especially useful for "complex algorithms and data structures" (Ousterhout, Different tools for different tasks, ¶ 1).

It is clear that programming has shifted from developing monolithic systems to developing software components. These components can be compiled, tested and modified individually, speeding development. Using scripting languages, software components developed at different times, in different languages, and using different data formats can be glued together into a common systems application. The rise of the Internet and the World Wide Web (WWW,) has given increased urgency to the shift to scripting, as WWW applications derive information from data contained in often disparate and dislocated databases.

Information Systems managers must understand the underlying advantages and disadvantages of both scripting and systems programming languages to ensure each is used where and when it is appropriate. They must develop an appreciation for the complementary nature of these two types of programming languages, because Internet enabled applications demand it. Furthermore, Information Systems managers must be able to determine when it is appropriate to use a scripting language for rapid prototyping, and when it is advantageous to ship the prototype rather than port it or rewrite it in a systems programming language. This requires an appreciation of the different levels of difficulty in programming between a statically typed vs. a dynamically types language; between a language with automated memory management vs. a language where managing memory allocation is the programmer's responsibility; and between an

interpreted language vs. a compiled language. The differences are not black and white; they are not even in shades of gray. Instead, these should be thought of as two different approaches that are in tension; it is the function of the Information Systems manager to find the balance between these competing forces and thus to provide an optimum return on investment.

References

Bezroukov, N. (n.d.) *A Slightly Skeptical View on Scripting Languages*. Retrieved April 5, 2005 from http://www.softpanorama.org/Scripting/index.shtml

Cowlishaw, M. (n.d.) *A Brief History of 'Classic' REXX*. Retrieved January 9, 2005 from http://www-360.ibm.com/software/awdtools/rexx/library/rexxhist.html

Dickerson, C. (2003, February 21). *Tools for the short hike*. Retrieved April 5, 2005 from http://www.infoworld.com/article/03/02/21/08connection_1.html

Foley, J. (2005, April 18). Microsoft: Longhorn Is Till Compelling. *Information Week, 1035*, 24.

Graham, P. (2001, May). *Being Popular*. Retrieved January 25, 2005 from http://www.paulgraham.com/popular.html

Graham, P. (2002, May). *Revenge of the Nerds*. Retrieved January 25, 2005 from http://paulgraham.com/icad.html

Graham, P. (2002, May). *Succinctness is Power*. Retrieved January 25, 2005 from http://www.paulgraham.com/power.html

Graham, P. (2004, July). *Great Hackers*. Retrieved January 25, 2005 from http://www.paulgraham.com/gh.html

Graham, P. (2004, August). *The Python Paradox*. Retrieved January 25, 2005 from http://paulgraham.com/pypar.html

Kernighan, B. W. & Van Wyk, C. J. (1997, November 30). *Timing Trials, or, the Trials of Timing: Experiments with Scripting and User-Interface Languages.* Retrieved December 23, 2004 from http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html

Jacobs, B. (2004). *How To Design A Programming Language: A survey of scripting programming language feature options.* Findy Services. Retrieved January 13, 2005, from http://www.geocities.com/tablizer/langopts.htm

ITworld.com. (2001, March 14). *Are scripting languages the wave of the future?* Retrieved January 30, 2005 from http://www.itworld.com/AppDev/1262/itw-0314-rcmappdevint/pfindex.html

ITworld.com. (2001, January 23). *When should you use PHP?*. Retrieved March 17, 2005, from http://www.itworld.com/AppDev/4072/lw-01-php/pfindex.html

Lutz, M. (2001). *Programming Python* (2nd ed.). Cambridge: O'Reilly

O'Reilly & Associates, Inc., & Smith, B. (2004). *The Importance of Perl*. Retrieved March 14, 2005, from http://perl.oreilly.com/lpt/a/2100

Ousterhout, J. (1998, March) Scripting: Higher Level Programming for the 21st Century. [Electronic Version] *IEEE Computer*, 3, 23-30.

Prechelt, L. (2000, October). *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl*. Unpublished Manuscript. Retrieved 23 December, 2004, from http://page.mi.fu-berlin.de/~prechelt/Biblio/jccpprt_computer2000.pdf

Raymond, E. (2000, April 30). *Why Python?*. Retrieved March 14, 2005 from http://www.linuxjournal.com/article/3882

Sebesta, R. W. (2002). *Concepts of Programming Languages* (5th ed.). Boston: Addison Wesley

Spolsky, J. (2004, June 13). *How Microsoft Lost the API War*. Retrieved January 22, 2005, from http://www.joelonsoftware.com/articles/APIWar.html

Stallman, R. (1994, September 23). Why you should not use Tcl [Msg 1]. Retrieved December 22, 2004 from http://www.vanderburg.org/OldPages/Tcl/war/0000.html

Tcl Developer Xchange, (2001, April 25). *Scripting: Next-Generation Software Development.* Retrieved January 9, 2005 from http://www.tcl.tk/advocacy/whyscript.html

TIOBE Programming Community Index for January 2005. (2005, January). Retrieved January 9, 2005, from http://www.tiobe.com/tpci.htm

Turck MMCache for PHP. (2002). Retrieved December 28, 2004 from http://turck-mmcache.sourceforge.net/index_old.html

Udell, J. (2003, February 06). *Shipping the prototype*. Retrieved April 5, 2005, from http://www.infoworld.com/article/03/02/06/06stratdev_1.html

APPENDIX A

Language Comparisons

Language Comparisons

Numerous comparisons can and have been made between the various programming languages. To the Information Systems (IS) professional, these comparisons are not always as useful as they should be. Such comparisons do have some value if they highlight the language features that determine whether a language operates at a high level of abstraction or not. Unfortunately, most language comparisons focus on comparing the low-level features of various systems programming languages and ignore the scripting languages. This appendix describes the language features that differentiate systems programming languages from scripting languages, moves on to language features important in certain applications and organizations, and finishes with features that determine the success of a language.

Using the TIOBE Programming Index[2] is a convenient way of determining the most important languages in use on the World Wide Web. The index really measures a language's popularity, not its usefulness or suitability. This programming language comparison will use a subset of the languages represented on the TIOBE index, based on a judgment as to whether the inclusion of a particular language will be either useful or at least interesting. For example, C is on the list, because it is the most widely used language on the World Wide Web. C# is not on the list, because it is not widely used, and for the purposes of this comparison is not sufficiently different from C++ to require separate evaluation. Ruby is on the list because it is a new scripting language with some interesting qualities, and to provide a glimpse into a way to evaluate languages. Few systems programming languages are on the list, because most are used for legacy programs created before the advent of the World Wide Web and therefore add nothing to our discussion. Unlike some language comparisons, the list focuses on languages that may prove useful to the Information Systems professional. The list is as follows:

Systems Programming Languages

- C
- Java
- C++

Scripting Languages

- Perl
- Python
- PHP
- Ruby

1. Data Typing – Some languages require the type of a variable to be declared when the variable is created. This language feature is called strong typing, also known as static typing. Static typing allows a programmer to determine the amount of memory necessary for that specific variable, and forces the programmer to determine all the possible interactions with that specific variable ahead of time, ensuring that all the variables that interact have the same data type. A strongly typed language increase safety by reducing the chance of programmer error, but limits the flexibility of the language. A weakly typed language, or one utilizing dynamic typing,

[2] http://www.tiobe.com/tiobe_index/tekst.htm

determines the type of a variable as the program is being run. Dynamic typing gives a programmer a great deal of flexibility; languages that provide this level of flexibility are almost always interpreted instead of compiled. (See item #4 for more information on compilation and interpretation.

Statically typed languages	Dynamically typed languages
C, Java, C++	Perl, Python, PHP, Ruby

Table A-1: Static vs. Dynamic Typing

2. Memory Management – Earlier programming languages, including some still in wide use today, require the programmer to manually allocate and deallocate memory. This required the programmer to know how much memory to reserve for a particular variable, which required that the data type of the variable be known in advance. If the programmer makes an error and doesn't deallocate memory once the program is through with a particular memory location, the memory stays allocated. These errors tend to accumulate in large programs, and reduce the amount of free memory available to the program. These errors are commonly known as "memory leaks." Languages that use manual memory management must be strongly typed. Languages that use automated memory management, where the program allocates and deallocates memory as needed, may be strongly typed, like Java, (and C#, by the way,) but can also be weakly typed. Automated memory management should aid programmer productivity by reducing the number of lines of code they have to produce for a specific routine.

Manual Memory Management	Automated Memory Management
C, C++	Java, Perl, Python, PHP, Ruby

Table A-2: Memory Management

3. Compilation or Interpretation – Early computers were programmed directly in the native language of that machine's CPU; this language is called machine code, and is unique for each specific chip. Programming directly in machine code is highly inefficient, as machine code is not inherently readable, writeable, or maintainable. Higher level languages were developed that enable a programmer to write a single line of code that is executed as several lines of machine code. Converting high level languages into machine code is the task of a special piece of software called either a compiler or an interpreter. These programs are written specifically to convert programs written in a particular high level language into the machine code of a specific system. Compilers do this conversion ahead of time, so when the program is executed the machine code ready to be run. Interpreters perform the conversion into machine code as the program is being run.

Compilation of a large program can take a very long time, even on the fastest computers. Today, instead of writing one large program, systems are designed as components, and each component can be independently compiled. This makes compilation faster but also reduces the need for compilation. Instead, a component can be written in a very high level language and interpreted at execution.

A third technique is a hybrid between these two approaches. The hybrid approach compiles the high level program code into some form of intermediate code, sometimes called byte code. This intermediate code can then be run by virtual machines written for a specific computer platform. This approach can be faster than pure interpretation, but may prove to be less flexible.

Compilation	Hybrid	Pure Intpretation
C, C++	Java, Python	Perl, PHP, Ruby

Table A-3: Compilation and Interpretation

4. Object Orientation – Object Oriented Programming, (OOP), is a modern implementation of hierarchal program design. In this approach a data structure is encapsulated with the routines, called methods, which operate on the data. The data can only be affected by the routines encapsulated with the data. This approach provides a high degree of safety and regularity, (inflexibility,) since an object can only be acted upon in predetermined ways.

Some languages were built to support object orientation. Some hybrid languages started out as procedural languages, but had object oriented features grafted on later. Finally, some languages do not support object orientation at all.

Pure procedural languages	Hybrid languages	Pure OOP languages
C, PHP (v4 and earlier)	C++, Perl, Python, PHP(v5)	Java, Ruby

Table A-4: Languages – Object Oriented?

5. Inheritance - OOP is organized into classes and subclasses, with the subclass inheriting features from the higher level class. This is a pure hierarchal relationship. OOP can be very fast and efficient if the data is hierarchal. But most things in life are not organized into neat hierarchies. Thus we get into messy arrangements such as multiple inheritance, where a child subclass inherits properties from multiple parent classes. This is necessary because most data sets can be examined in more than one dimension. Without multiple inheritance, the OOP program is efficient only as long as the data is organized in the way the programmer anticipated, but if the data is organized differently, or if a user requests the data be examined using a different dimension, a program that cannot handle multiple inheritance becomes extremely inefficient. Some OOP languages use single inheritance, but have techniques that approximate the benefits of multiple inheritance. These techniques are interesting, but are outside the scope of this brief discussion.

Not Applicable	Single Inheritance	Hybrid Implementation	Multiple Inheritance
C, PHP (v4 and earlier)	PHP	Java, Ruby	C++, Perl, Python

Table A-5: OOP Inheritance

6. Pointer Arithmetic – Systems programming languages often require the programmer to manually allocate and deallocate memory. (See item #2.) This is done through memory address pointers directly to the physical locations. In certain cases, where speed and efficiency are paramount, some languages allow the programmer to perform addition on the pointers instead of going through a complicated and time consuming name lookup procedure. In languages that use automated memory management, the program uses virtual memory; references to memory are to a virtual location, which is converted into the actual physical location. In this arrangement, memory is not allocated in contiguous blocks, and pointer arithmetic is impossible.

Pointer Arithmetic	Not Applicable
C, C++	Java, Perl, Python, PHP, Ruby

Table A-6: Pointer Arithmetic

7. Libraries – When developing programs, programmers often come across difficult and time consuming problems, or situations that tend to arise over and over again. Solutions to these problems are maintained in program libraries. A good programming language provides a substantial set of libraries available to the programmer, so they can call a specific function or subroutine rather than develop a new solution from scratch. Libraries are crucial to the success of a language, so much so that the developers of C++ made it backwards compatible in part so programmers using C++ could use the extensive C libraries.

Most of the languages being discussed here have extensive library support; the older and more popular the language, the more extensive the libraries. A language like C, or C++, has a huge set of libraries; some of them predate the World Wide Web and are not germane to the issue of suitability for the web. A language like Ruby is only now gaining acceptance, and has fewer libraries. Java, Python, Perl & PHP have more than adequate library support, or they would not have gained such widespread acceptance.

8. Language Integration – Because software is developed as individual components, and because components need not always be written in the same programming language, a useful programming language will interface with a variety of other languages. Language integration is key to interoperability. Systems programming languages, being relatively inflexible, have to have this integration built into the system.

Languages	Language Integration
C	C++, Java, Perl, Python, PHP, Ruby
C++	C, Assembler
Java	C, C++ (limited)
Perl	C, C++
Python	C, C++, Java

PHP	
Ruby	

Table A-7: Language Integration

Specific language support may not be built into a scripting language because the flexibility of scripting makes such support unnecessary. Scripting languages often require only a runtime (interpreter) for a specific operating system and platform; if that is available, a scripting language should be flexible enough to interconnect two otherwise incompatible programs.

APPENDIX B

Bibliography of Online Resources

Bibliography of Online Resources

The information in the following web sites made important contributions to this paper, and are presented here, with commentary, to help you quickly find information of value to you.

http://www.tiobe.com/tpci.htm

This site categorizes languages by their use in the World Wide Web. It is a measure of a languages popularity, pervasiveness, and growth.

http://shootout.alioth.debian.org/index.php?sort=kb

This site provides the results from a variety of language tests. By carefully selecting a set of tests, one can derive a language's suitability for a particular problem domain.

http://www.jvoegele.com/software/langcomp.html

Personal evaluation and comparison of many popular programming languages. It is intended to provide very high-level information about the respective languages to anyone who is trying to decide which language(s) to learn or to use for a particular project. Has some nice charts comparing language features.

http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html

"Timing Trials, or, the Trials of Timing" The Authors ran an empirical comparison of various scripting and user interface languages. They derived a set of general principles and themes that are relevant, even if the data is no longer.

http://page.mi.fu-berlin.de/~prechelt/Biblio/jccprt_computer2000.pdf

"An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl." This is a really good article with an interesting approach to determining the "productivity" of a given language. It is useful, but due to the relatively simple nature of the problems it is not as accurate as a real world comparison would be, (with programs that take a month or more to complete.) However, a real world comparison would be far too expensive, so this will have to do.

http://www.mvps.org/scripting/languages/

This site provides a brief description of Windows scripting languages, and provides a variety of links for further study.

http://directory.google.com/Top/Computers/Programming/Languages/Comparison_and_Review/

Google Directory page listing a variety of links to language comparisons. Some are useful, some are playful, some are useless.

http://home.pacbell.net/ouster.html

Article originally published in IEEE Computer magazine. This is a great article for describing the difference between systems programming languages and scripting languages. I would have saved a lot of time had I come across this first.

http://www.python.org/doc/essays/comparisons.html

An article by Guido van Rossum comparing Python to different languages. Guido says it

is dated, (1997), but it is still valuable in setting the parameters for language comparisons.

http://www.paulgraham.com/power.html

"Succinctness is Power" This article takes issue with a particular statement that places "regularity and readability" over "power." Power equates to succinctness, and succinctness is what abstraction is all about. Everything else being equal, a more abstract language is more succinct and powerful; power is what we are after in a computer language. [COBOL is very readable, and is very regular. It is extremely verbose, however, meaning it operates at a relatively low level of abstraction.]

http://www.linuxjournal.com/article/3882

"Why Python" The author describes his switch from Perl to Python. The reasons for the change illuminate various language comparison features such as writebility, readability, and maintainability.

http://perl.oreilly.com/lpt/a/2100

This article is a really good introduction to scripting in general. It highlights the problem domain for which scripting is optimized, describes how scripting is a much higher level language than systems programming languages, and being higher level, cuts development time dramatically. Now that computing time is cheap, developer time matters more. [As an aside to this, if programs were written in higher level languages programmers would be more productive, which should lessen the rush to outsource programming to India.]

http://www.wdvl.com/Authoring/Languages/PHP/Welcome/index.html

"Welcome to PHP" This article describes PHP and its suitability to a specific problem domain---developing dynamic websites. It then goes on to describe a variety of PHP features and structures. It does state that PHP "remains an immature language," although this statement applies to PHP3, not the more modern and advanced version available today.

http://www.ukuug.org/events/linux2002/papers/html/php/index.html

"Experience of Using PHP in Large Websites" This article describes how PHP's features make it difficult to use the language for a large commercial website. [Last.fm has had to reduce its use of opcode caches and recode portions of the site in Java because PHP didn't scale well.]

http://www.tcl.tk/advocacy/whyscript.html

Explains what scripting is and how it differs from systems programming.

http://www.tcl.tk/advocacy/scriptHistory.html

A basic history of scripting, from shell languages to JavaScript

http://www.geocities.com/tablizer/langopts.htm

A programmer describes what would go into the ideal programming language. In developing this idea, the author delves into a variety of language features. An opinion piece, this is nonetheless an interesting and valuable essay.

APPENDIX C

Language Recommendation

Language Recommendation

The temptation of providing a language recommendation is overwhelming. No one who studies the various languages can help but develop strong opinions on the subject. Nevertheless, any language recommendation, divorced from an understanding of a particular organization, the hardware in use, and the application in question would be foolhardy at best, and damaging at worst.

The Information Systems professional must understand some language fundamentals, must evaluate the available languages, must inventory the hardware, and must determine the structure of the application in question. Only then can the proper programming language be determined. This is no casual undertaking. It takes research, thought, insight, and the dedication to see the decision through.

Once a programming or scripting language is chosen it impacts everything. The language determines which people you hire, what equipment you purchase, what operating system you install and how long it takes to develop the application. The language may determine the elegance of the programming solution. The language may determine how fast the application runs and how many individual users or transactions the application can support.

It is impossible to say for sure that one language is faster than another as that is highly dependent upon the runtime for that language and not necessarily the language itself. It is also dependent upon the skill of the programmer; a poorly written program will run slowly even when written in a faster language. In addition, the languages are constantly evolving and any side-by-side comparisons are soon out of date.

Nevertheless, in the summer of 2005 the following appear to be true. PHP is suited for small scripting projects but does not appear to scale. Perl and Python scale well; Python in particular appears to be powerful enough to build large applications as well as tiny scripts. Perl scales but does have issues involving maintainability. Both PHP and Perl have different issues with writability: PHP is not suitable for programs developed by large teams, and Perl has so many ways of doing everything that individual Perl programmers deal only with the subset of language features they are familiar with. Thus no two Perl programmers will program the same problem in the same way and using the same features, making it difficult for Perl programmers to work in teams.

Readability and Maintainability are closely related. A language with an awkward and inconsistent syntax will be difficult to read, and difficult to maintain. Many put Perl in this category, including some Perl coders. Python uses indentation to define different sections; the indentation that some programmers add for readability actually means something in Python. Perl programmers, being used to the curly braces, dislike the nested indentation in Python. This is partly a matter of taste, so it is up to the Information Security professional to make an objective determination.

Writability is important. Some argue that the curly braces in Perl are awkward, while others coming from a Unix background understand and embrace the syntax. Yet these might be the same people who dislike the nested parenthesis in Lisp. Any language is easy for a single

programmer to write in once that programmer learns the language. But non-trivial applications are created by teams of programmers writing, editing, and interacting with each other's code. A language must have the proper software configuration management tools to control versioning. PHP, for example, does not have good software configuration management tools and is therefore not a good choice for a large scale project. Writability has many facets, and the Information Systems professional must balance them all when choosing a language.

As an example of how to approach a language, let's examine one of the newer scripting languages, Ruby. Ruby is a pure object-oriented scripting language, dynamically typed, and is highly regarded in some circles, (especially in Japan, where it was developed.) A successful a scripting language usually needs to come bundled with an operating system, be part of a server application, or be supported by the major Internet browsers. In the Spring of 2005, this support is limited. Ruby can be added to Apache servers, but Apache does not provide native support for Ruby, limiting its appeal. Ruby can interact with a variety of languages, and does have useful libraries. Ruby has only now begun to be used outside Japan, and very little has been written on the language. Ruby may be a fine scripting language, but would require careful study to determine its usefulness to the enterprise.

If Information Systems professionals evaluate individual languages in this way, adding to that evaluation information regarding a language's Readability, Writability and Maintainability, they will be ready to make informed decisions for themselves. Developing the framework for that sort of decision making was the point of this whole paper.